Contents

1 Notation and conventions **2**
1.0.1 Background Information . . . 3
1.1 Some Useful Mathematical Facts . . . 4
1.2 Acknowledgements . . . 4

2 Some Preliminaries **5**
2.1 Notation and conventions . . . 5
2.1.1 Background Information . . . 6
2.2 Some Useful Mathematical Facts . . . 7
2.3 Acknowledgements . . . 7
2.4 The Curse of Dimension . . . 7
2.4.1 The Curse: Data isn't Where You Think it is . . . 7
2.4.2 Minor Banes of Dimension . . . 9

3 Learning to Classify **10**
3.1 Classification, Error, and Loss . . . 10
3.1.1 Loss and the Cost of Misclassification . . . 11
3.1.2 Building a Classifier from Probabilities . . . 11
3.1.3 Building a Classifier using Decision Boundaries . . . 12
3.1.4 What will happen on Test Data? . . . 12
3.1.5 The Class Confusion Matrix . . . 14
3.1.6 Statistical Learning Theory and Generalization . . . 15
3.2 Classifying with Naive Bayes . . . 16
3.3 The Support Vector Machine . . . 20
3.3.1 Choosing a Classifier with the Hinge Loss . . . 21
3.3.2 Finding a Minimum: General Points . . . 23
3.3.3 Finding a Minimum: Stochastic Gradient Descent . . . 24
3.3.4 Example: Training a Support Vector Machine with Stochastic Gradient Descent 26
3.3.5 Multi-Class Classifiers . . . 28
3.4 Classifying with Random Forests . . . 29
3.4.1 Building a Decision Tree . . . 29
3.4.2 Entropy and Information Gain . . . 32
3.4.3 Entropy and Splits . . . 34
3.4.4 Choosing a Split with Information Gain . . . 35
3.4.5 Forests . . . 37
3.4.6 Building and Evaluating a Decision Forest . . . 37
3.4.7 Classifying Data Items with a Decision Forest . . . 38
3.5 Classifying with Nearest Neighbors . . . 40
3.6 You should . . . 43
3.6.1 be able to: . . . 43
3.6.2 remember: . . . 43

4 Extracting Important Relationships in High Dimensions 49
4.1 Some Plots of High Dimensional Data . . . 49
4.1.1 Understanding Blobs with Scatterplot Matrices - CLEANUP 49
4.1.2 Parallel Plots . . . 49
4.1.3 Scatterplot Matrices . . . 50
4.2 Summaries of High Dimensional Data . . . 59
4.2.1 The Mean . . . 59
4.2.2 Using Covariance to encode Variance and Correlation . . . 59
4.3 Blob Analysis of High-Dimensional Data . . . 63
4.3.1 Transforming High Dimensional Data . . . 63
4.3.2 Transforming Blobs . . . 64
4.3.3 Whitening Data . . . 67
4.4 Principal Components Analysis . . . 69
4.4.1 The Blob Coordinate System and Smoothing . . . 70
4.4.2 The Low-Dimensional Representation of a Blob . . . 72
4.4.3 Smoothing Data with a Low-Dimensional Representation . . . 74
4.4.4 The Error of the Low-Dimensional Representation . . . 76
4.4.5 Example: Representing Spectral Reflectances . . . 78
4.4.6 Example: Representing Faces with Principal Components . . . 79
4.5 High Dimensions, SVD and NIPALS . . . 81
4.5.1 Principal Components by SVD . . . 81
4.5.2 Just a few Principal Components with NIPALS . . . 82
4.5.3 Projection and Discriminative Problems . . . 84
4.5.4 Just a few Discriminative Directions with PLS1 . . . 85
4.6 Multi-Dimensional Scaling . . . 86
4.6.1 Principal Coordinate Analysis . . . 86
4.6.2 Example: Mapping with Multidimensional Scaling . . . 88
4.7 Example: Understanding Height and Weight . . . 90
4.8 What you should remember - NEED SOMETHING . . . 93

5 Clustering: Models of High Dimensional Data 97
5.1 Agglomerative and Divisive Clustering . . . 97
5.1.1 Clustering and Distance . . . 99
5.2 The K-Means Algorithm and Variants . . . 104
5.2.1 How to choose K . . . 106
5.2.2 Soft Assignment . . . 108
5.2.3 General Comments on K-Means . . . 109
5.3 Describing Repetition with Vector Quantization . . . 111
5.3.1 Vector Quantization . . . 112
5.3.2 Example: Groceries in Portugal . . . 113
5.3.3 Efficient Clustering and Hierarchical K Means . . . 115
5.3.4 Example: Activity from Accelerometer Data . . . 116
5.4 You should . . . 120
5.4.1 remember: . . . 120

6 Clustering using Probability Models 122
6.1 The Multivariate Normal Distribution . . . 122

6.1.1 Affine Transformations and Gaussians . . . 123
6.1.2 Plotting a 2D Gaussian: Covariance Ellipses . . . 123
6.2 Mixture Models and Clustering . . . 124
6.2.1 A Finite Mixture of Blobs . . . 125
6.2.2 Topics and Topic Models . . . 126
6.3 The EM Algorithm . . . 129
6.3.1 Example: Mixture of Normals: The E-step . . . 130
6.3.2 Example: Mixture of Normals: The M-step . . . 132
6.3.3 Example: Topic Model: The E-Step . . . 133
6.3.4 Example: Topic Model: The M-step . . . 134
6.3.5 EM in Practice . . . 134
6.4 You should . . . 136
6.4.1 remember: . . . 136

7 Regression 138
7.1 Overview . . . 138
7.1.1 Regression to Spot Trends . . . 140
7.2 Linear Regression and Least Squares . . . 142
7.2.1 Linear Regression . . . 142
7.2.2 Choosing β . . . 143
7.2.3 Residuals . . . 145
7.2.4 R-squared . . . 146
7.2.5 Transforming Variables . . . 148
7.2.6 Can you Trust Your Regression? . . . 151
7.3 Problem Data Points . . . 152
7.3.1 Problem Data Points have Significant Impact . . . 153
7.3.2 The Hat Matrix and Leverage . . . 155
7.3.3 Cook's Distance . . . 156
7.3.4 Standardized Residuals . . . 157
7.4 Many Explanatory Variables . . . 159
7.4.1 Functions of One Explanatory Variable . . . 160
7.4.2 Regularizing Linear Regressions . . . 161
7.4.3 Example: Weight against Body Measurements . . . 165
7.5 You should . . . 169
7.5.1 remember: . . . 169

8 Regression: Choosing and Managing Models 173
8.1 Model Selection: Which Model is Best? . . . 173
8.1.1 Bias and Variance . . . 173
8.1.2 Choosing a Model using Penalties: AIC and BIC . . . 175
8.1.3 Choosing a Model using Cross-Validation . . . 177
8.1.4 A Search Process: Forward and Backward Stagewise Regression 177
8.1.5 Significance: What Variables are Important? . . . 178
8.2 Robust Regression . . . 179
8.2.1 M-Estimators and Iteratively Reweighted Least Squares . . . 180
8.2.2 Scale for M-Estimators . . . 182
8.3 Generalized Linear Models . . . 183

8.3.1 Logistic Regression . . . 183
8.3.2 Multiclass Logistic Regression . . . 185
8.3.3 Regressing Count Data . . . 185
8.4 L1 Regularization and Sparse Models . . . 186
8.4.1 Dropping Variables with L1 Regularization . . . 186
8.5 Bayesian Regression . . . 190
8.5.1 Bayesian Regression with a Normal Prior . . . 191
8.5.2 Tricks with Normal Distributions . . . 192
8.5.3 Bayesian Regression - Overview . . . 196
8.5.4 Bayesian Regression with Everything Normal . . . 196
8.6 You should . . . 198
8.6.1 remember: . . . 198

9 Non-Parametric Regression 202
9.1 Modelling with Bumps . . . 202
9.1.1 Scattered Data: Smoothing and Interpolation . . . 202
9.1.2 Density Estimation . . . 206
9.1.3 Kernel Smoothing . . . 208
9.2 Exploiting Your Neighbors for Regression . . . 210
9.2.1 Local Polynomial Regression . . . 212
9.2.2 Using your Neighbors to Predict More than a Number . . . 214

10 Neural Networks 217
10.1 Units and Classification . . . 217
10.1.1 Building a Classifier out of Units: The Cost Function . . . 217
10.1.2 Building a Classifier out of Units: Training . . . 219
10.2 Layers and Networks . . . 220
10.2.1 Notation . . . 220
10.2.2 Training, Gradients and Backpropagation . . . 221
10.2.3 Training Multiple Layers . . . 222
10.2.4 Gradient Scaling Tricks . . . 223
10.3 Convolution and Orientation Features for Images . . . 224
10.4 Convolutional neural networks . . . 224

11 Classification II 225
11.1 Logistic Regression . . . 225
11.1.1 The Logistic Loss . . . 225
11.1.2 Logistic Regression and Softmax . . . 227
11.2 Neural Nets . . . 227
11.2.1 Two Layers of Logistic Regression . . . 227
11.2.2 Training, Gradients and Backpropagation . . . 227
11.2.3 Variants of SGD . . . 227
11.3 Convolution and orientation features . . . 227
11.4 Convolutional neural networks . . . 227

12 Boosting 228
12.1 GradientBoost . . . 228

12.2 ADAboost 228

13 Some Important Models **229**
13.1 HMM's 229
13.2 CRF's 229
13.3 Fitting and inference with MCMC? 229

14 Math Resources **230**
14.1 Useful Material about Matrices 230
14.1.1 The Singular Value Decomposition 231
14.1.2 Approximating A Symmetric Matrix 232

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 212).
- $P(\{\mathcal{E}\})$ is the probability of event $\mathcal{E}$ (page 212).
- $P(\{\mathcal{E}\}|\{\mathcal{F}\})$ is the probability of event $\mathcal{E}$, conditioned on event $\mathcal{F}$ (page 212).
- $p(x)$ is the probability that random variable X will take the value x; also written $P(\{X = x\})$ (page 212).
- $p(x, y)$ 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 212).
- $\underset{x}{\text{argmax}}\ f(x)$ 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 SOME USEFUL MATHEMATICAL FACTS

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

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

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

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

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

1.2 ACKNOWLEDGEMENTS

Typos spotted by: Han Chen (numerous!), Henry Lin (numerous!), 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.

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). 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 variance 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 **??**).

- $\mathsf{corr}(\{(x,y)\})$ 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 212).
- $P(\{\mathcal{E}\})$ is the probability of event $\mathcal{E}$ (page 212).
- $P(\{\mathcal{E}\}|\{\mathcal{F}\})$ is the probability of event $\mathcal{E}$, conditioned on event $\mathcal{F}$ (page 212).
- $p(x)$ is the probability that random variable X will take the value x; also written $P(\{X = x\})$ (page 212).
- $p(x,y)$ 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 212).
- $\underset{x}{\operatorname{argmax}}\, f(x)$ means the value of x that maximises $f(x)$.
- $\underset{x}{\operatorname{argmin}}\, f(x)$ 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.

2.4 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.

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

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

The first surprising fact about high dimensional data is that most of the data can lie quite far away from the mean. For example, we can divide our dataset into two pieces. $\mathcal{A}(\epsilon)$ consists of all data items where *every* component of the data has a value in the range $[-(1-\epsilon), (1-\epsilon)]$. $\mathcal{B}(\epsilon)$ consists of all the rest of the data. If you think of the data set as forming a cubical orange, then $\mathcal{B}(\epsilon)$ is the rind (which has thickness ϵ) and $\mathcal{A}(\epsilon)$ is the fruit.

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

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

to compute as

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

and

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

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

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

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

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

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

but we can rearrange, so that

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

Now each component of $\mathbf{x}$ is independent, so that $P(\mathbf{x}) = P(x_1)P(x_2)\dots P(x_d)$. 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

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

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.

2.4.2 Minor Banes of Dimension

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

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

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

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

CHAPTER 3

Learning to Classify

A **classifier** is a procedure that accepts a set of features and produces a class label for them. There could be two, or many, classes. 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. In this (crude) model, the doctor accepts a set of features, which might be 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).

Classifiers are built by taking a set of labeled examples and using them to come up with a procedure that assigns a label to any new example. In the general problem, we have a training dataset $(\mathbf{x}_i, y_i)$; each of the **feature vectors** $\mathbf{x}_i$ consists of measurements of the properties of different types of object, and the y_i are labels giving the type of the object that generated the example. We will then use the training dataset to find a procedure that will predict an accurate label (y) for any new object ($\mathbf{x}$).

Definition: 3.1 *Classifier*

A classifier is a procedure that accepts a feature vector and produces a label.

3.1 CLASSIFICATION, ERROR, AND LOSS

You should think of a classifier as a procedure — we pass in a feature vector, and get a class label in return. We want to use the training data to find the procedure that is "best" when used on the test data. This problem has two tricky features.

First, we need to be clear on what a good procedure is. Second, we really want the procedure to be good on test data, which we haven't seen and won't see; we only get to see the training data. These two considerations shape much of what we do.

3.1.1 Loss and the Cost of Misclassification

The choice of procedure must depend on the cost of making a mistake. This cost can be represented with a **loss function**, which specifies the cost of making each type of mistake. I will write $L(j \rightarrow k)$ for the loss incurred when classifying an example of class j as having class k.

A two-class classifier can make two kinds of mistake. Because two-class classifiers are so common, there is a special name for each kind of mistake. A **false positive** occurs when a negative example is classified positive (which we can write $L(- \rightarrow +)$ and avoid having to remember which index refers to which class); a **false negative** occurs when a positive example is classified negative (similarly $L(+ \rightarrow -)$). By convention, the loss of getting the right answer is zero, and the loss for any wrong answer is non-negative.

The choice of procedure should depend quite strongly on the cost of each mistake. For example, pretend there is only one disease; then doctors would be classifiers, deciding whether a patient had it or not. If this disease is dangerous, but is safely and easily treated, false negatives are expensive errors, but false positives are cheap. In this case, procedures that tend to make more false positives than false negatives are better. Similarly, if the disease is not dangerous, but the treatment is difficult and unpleasant, then false positives are expensive errors and false negatives are cheap, and so we prefer false negatives to false positives.

You might argue that the best choice of classifier makes no mistake. But for most practical cases, the best choice of classifier is guaranteed to make mistakes. As an example, consider an alien who tries to classify humans into male and female, using only height as a feature. However the alien's classifier uses 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 whatever gender it chooses for that height.

For many practical problems, it is difficult to know what loss function to use. There is seldom an obvious choice. One common choice is to assume that all errors are equally bad. This gives the **0-1 loss** — every error has loss 1, and all right answers have loss zero.

3.1.2 Building a Classifier from Probabilities

Assume that we have a reliable model of $p(y|\mathbf{x})$. This case occurs less often than you might think for practical data, because building such a model is often very difficult. However, when we do have a model and a loss function, it is easy to determine the best classifier. We should choose the rule that gives minimum expected loss.

We start with a two-class classifier. At $\mathbf{x}$, the expected loss of saying $-$ is $L(+ \rightarrow -)p(+|\mathbf{x})$ (remember, $L(- \rightarrow -) = 0$); similarly, the expected loss of saying $+$ is $L(- \rightarrow +)p(-|\mathbf{x})$. At most points, one of $L(- \rightarrow +)p(-|\mathbf{x})$ and

$L(+ \to -)p(+|\mathbf{x})$ is larger than the other, and so the choice is clear. The remaining set of points (where $L(- \to +)p(-|\mathbf{x}) = L(+ \to -)p(+|\mathbf{x})$) is "small" (formally, it has zero measure) for most models and problems, and so it doesn't matter what we choose at these points. This means that the rule

$$\text{say} \begin{cases} + & \text{if } L(+ \to -)p(+|\mathbf{x}) > L(- \to +)p(-|\mathbf{x}) \\ - & \text{if } L(+ \to -)p(+|\mathbf{x}) < L(- \to +)p(-|\mathbf{x}) \\ \text{random choice} & \text{otherwise} \end{cases}$$

is the best available. Because it doesn't matter what we do when $L(+ \to -)p(+|\mathbf{x}) = L(- \to +)p(-|\mathbf{x})$, it is fine to use

$$\text{say} \begin{cases} + & \text{if } L(+ \to -)p(+|\mathbf{x}) > L(- \to +)p(-|\mathbf{x}) \\ - & \text{otherwise} \end{cases}$$

The same reasoning applies in the multi-class case. We choose the class where the expected loss from that choice is smallest. In the case of 0-1 loss, this boils down to:

$$\text{choose } k \text{ such that } p(k|\mathbf{x}) \text{ is largest.}$$

3.1.3 Building a Classifier using Decision Boundaries

Building a classifier out of posterior probabilities is less common than you might think, for two reasons. First, it's often very difficult to get a good posterior probability model. Second, most of the model doesn't matter to the choice of classifier. What is important is knowing which class has the lowest expected loss, not the exact values of the expected losses, so we should be able to get away without an exact posterior model.

Look at the rules in section 3.1.2. Each of them carves up the domain of $\mathbf{x}$ into pieces, and then attaches a class – the one with the lowest expected loss – to each piece. There isn't necessarily one piece per class, (though there's always one class per piece). The important factor here is the boundaries between the pieces, which are known as **decision boundaries**. A powerful strategy for building classifiers is to choose some way of building decision boundaries, then adjust it to perform well on the data one has. This involves modelling considerably less detail than modelling the whole posterior.

For example, in the two-class case, we will spend some time discussing the decision boundary given by

$$\text{choose} \begin{cases} - & \text{if } \mathbf{x}^T\mathbf{a} + b < 0 \\ + & \text{otherwise} \end{cases}$$

often written as $\text{sign}\mathbf{x}^T\mathbf{a} + b$ (section 9.9). In this case we choose $\mathbf{a}$ and b to obtain low loss.

3.1.4 What will happen on Test Data?

What we really want from a classifier is to have small loss on test data. But this is difficult to measure or achieve directly. For example, think about the case of classifying credit-card transactions as "good" or "bad". We could certainly obtain

a set of examples that have been labelled for training, because the card owner often complains some time after a fraudulent use of their card. But what is important here is to see a new transaction and label it without holding it up for a few months to see what the card owner says. The classifier may never know if the label is right or not.

Generally, we will assume that the training data is "like" the test data, and so we will try to make the classifier perform well on the training data. Classifiers that have small training error might not have small test error. One example of this problem is 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 and otherwise chooses randomly between the classes. This classifier has been learned from data, and has a zero error rate on the training dataset; it is likely to be unhelpful on any other dataset, however.

Test error is usually worse than training error, because of an effect that is sometimes called **overfitting**, so called because the classification procedure fits the training data better than it fits the test data. 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 fails to generalize. The effect occurs because the classifier has been trained to perform well *on the training dataset*, and the training dataset is not the same as the test dataset. First, it is quite likely smaller. Second, it might be biased through a variety of accidents. This means that small training error may have to do with quirks of the training dataset that don't occur in other sets of examples. One consequence of overfitting is that classifiers should always be evaluated on data that was not used in training.

Remember this: *Classifiers should always be evaluated on data that was not used in training.*

Now assume that we are using the 0-1 loss, so that the loss of using a classifier is the same as the **error rate**, that is, the percentage of classification attempts on a test set that result in the wrong answer. We could also use the **accuracy**, which is the percentage of classification attempts that result in the right answer. We cannot estimate the error rate of the classifier using training data, 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 often called a **test set**), then train the classifier on the rest of the data, and evaluate on the validation set. This has the difficulty that the classifier will not be the best estimate possible, because we have 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 repeatedly: 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. This allows an estimate of the likely future performance of a classifier, at the expense of substantial computation. You should notice that cross-validation, in some sense, looks at the sensitivity of the classifier to a change in the training set. The most usual form of this algorithm involves omitting single items from the dataset and is known as **leave-one-out cross-validation**.

You should usually compare the error rate of a classifier to two important references. The first is the error rate if you assign classes to examples uniformly at random, which for a two class classifier is 50%. A two class classifier should never have an error rate higher than 50%. If you have one that does, all you need to do is swap its class assignment, and the resulting error rate would be lower than 50%. The second is the error rate if you assign all data to the most common class. If one class is uncommon and the other is common, this error rate can be hard to beat. Data where some classes occur very seldom requires careful, and quite specialized, handling.

3.1.5 The Class Confusion Matrix

Evaluating a multi-class classifier is more complex than evaluating a binary classifier. The error rate if you assign classes to examples uniformly at random can be rather high. If each class has about the same frequency, then this error rate is $(1 - 100/\text{number of classes})\%$. A multi-class classifier can make many more kinds of mistake than a binary classifier can. It is useful to know the total error rate of the classifier (percentage of classification attempts that produce the wrong answer) or the accuracy, (the percentage of classification attempts that produce the *right* answer). If the error rate is low enough, or the accuracy is high enough, there's not much to worry about. But if it's not, you can look at the **class confusion matrix** to see what's going on.

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

TABLE 3.1: The class confusion matrix for a multiclass classifier. Further details about the dataset and this example appear in worked example 3.3.

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.

3.1.6 Statistical Learning Theory and Generalization

What is required in a classifier is an ability to *predict*—we should like to be confident that the classifier chosen on a particular data set has a low risk on future data items. The family of decision boundaries from which a classifier is chosen is an important component of the problem. Some decision boundaries are more flexible than others (in a sense we don't intend to make precise). This has nothing to do with the number of parameters in the decision boundary. For example, if we were to use a point to separate points on the line, there are very small sets of points that are not linearly separable (the smallest set has three points in it). This means that relatively few sets of points on the line are linearly separable, so that if our dataset is sufficiently large and linearly separable, the resulting classifier is likely to behave well in future. However, using the sign of $\sin \lambda x$ to separate points on the line produces a completely different qualitative phenomenon; for *any* labeling of distinct points on the line into two classes, we can choose a value of λ to achieve this labeling. This flexibility means that the classifier is wholly unreliable—it can be made to fit any set of examples, meaning the fact that it fits the examples is uninformative.

There is a body of theory that treats this question, which rests on two important points.

- **A large enough dataset yields a good representation of the source of the data:** this means that if the dataset used to train the classifier is very large, there is a reasonable prospect that the performance on the training set will represent the future performance. However, for this to be helpful, we need the data set to be large with respect to the "flexibility" of the family
- **The "flexibility" of a family of decision boundaries can be formalized:** yielding the **Vapnik-Chervonenkis dimension** (or **V-C dimension**) of the family. This dimension is independent of the number of parameters of the family. Families with finite V-C dimension can yield classifiers whose future performance can be bounded using the number of training elements; families with infinite V-C dimension (like the $\sin \lambda x$ example above) cannot be used to produce reliable classifiers.

The essence of the theory is as follows: if one chooses a decision boundary from an inflexible family, and the resulting classifier performs well on a large data set, there is strong reason to believe that it will perform well on future items drawn from the same source. This statement can be expressed precisely in terms of bounds on total risk to be expected for particular classifiers as a function of the size of the data set used to train the classifier. These bounds hold in probability. These bounds

tend not to be used in practice, because they appear to be extremely pessimistic. Space doesn't admit an exposition of this theory—which is somewhat technical—but interested readers can look it up in (**?**, **?**, **?**).

3.2 CLASSIFYING WITH NAIVE BAYES

One reason it is difficult to build a posterior probability model is the dependencies between features. However, if we *assume* that features are conditionally independent conditioned on the class of the data item, we can get a simple expression for the posterior. This assumption is hardly ever true in practice. Remarkably, this doesn't matter very much, and the classifier we build from the assumption often works extremely well. It is the classifier of choice for very high dimensional data.

Recall bayes' rule. If we have $p(\mathbf{x}|y)$ (often called either a **likelihood** or **class conditional probability**), and $p(y)$ (often called a **prior**) then we can form

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

(the **posterior**). We write x_j for the j'th component of $\mathbf{x}$. Our assumption is

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

(again, this isn't usually the case; it just turns out to be fruitful to *assume* that it is true). This assumption means that

$$\begin{aligned} p(y|\mathbf{x}) &= \frac{p(\mathbf{x}|y)p(y)}{p(\mathbf{x})} \\ &= \frac{\prod_i p(x_i|y)p(y)}{p(\mathbf{x})} \\ &\propto \prod_i p(x_i|y)p(y). \end{aligned}$$

Now because we need only to know the posterior values up to scale at $\mathbf{x}$ to make a decision (check the rules above if you're unsure), we don't need to estimate $p(\mathbf{x})$. In the case of 0-1 loss, this yields the rule

$$\text{choose } y \text{ such that } \prod_i p(x_i|y)p(y) \text{ is largest.}$$

Naive bayes is particularly good when there are a large number of features, but there are some things to be careful about. 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. A model with many different features is likely to have many strongly negative log probabilities, so you should not just add up all the log probabilities then exponentiate, or else you will find that each class has a posterior probability of zero. Instead, subtract the largest log from all the others, then exponentiate; you will obtain a vector proportional to the class probabilities, where the largest element has the value 1.

We still need models for $p(x_i|y)$ for each x_i. It turns out that simple parametric models work really well here. For example, one could fit a normal distribution to each x_i in turn, for each possible value of y, using the training data. The logic of the measurements might suggest other distributions, too. If one of the x_i's was a count, we might fit a Poisson distribution. If it was a 0-1 variable, we might fit a Bernoulli distribution. If it was a numeric variable that took one of several values, then we might use either a multinomial model.

Many effects cause missing values: measuring equipment might fail; a record could be damaged; it might be too hard to get information in some cases; survey respondents might not want to answer a question; and so on. As a result, missing values are quite common in practical datasets. A nice feature of naive bayes classifiers is that they can handle missing values for particular features rather well.

Dealing with missing data during learning is easy. 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.

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. If there are some classes with very little data, then the classifier is likely to work poorly, because you will have trouble getting reasonable estimates of the parameters for the $p(x_i|y)$.

Worked example 3.1 *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: 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. The class-confusion matrix on the test set was:

Prediction	adi	car	con	fad	gla	mas
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 16%. The κ is 4.34. These numbers, and the class-confusion matrix, will vary with test-train split. I have not averaged over splits, which would be the next thing.

Worked example 3.2 *Classifying mouse protein expression*

Build a naive bayes classifier to classify the "mouse protein" dataset from the UC Irvine machine learning repository. The dataset is at http://archive.ics.uci.edu/ml/datasets/Mice+Protein+Expression.

Solution: There's only one significant difficulty here; many of the data items are incomplete. I dropped all incomplete data items, which is about half of the dataset. One can do somewhat more sophisticated things, but we don't have the tools yet. 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, then trained with cross-validation on the remainder. The class-confusion matrix on the test set was:

Pred'n	c-CS-m	c-CS-s	c-SC-m	c-SC-s	t-CS-m	t-CS-s	t-SC-m	t-SC-s
c-CS-m	9	0	0	0	0	0	0	0
c-CS-s	0	15	0	0	0	0	0	0
c-SC-m	0	0	12	0	0	0	0	0
c-SC-s	0	0	0	15	0	0	0	0
t-CS-m	0	0	0	0	18	0	0	0
t-CS-s	0	0	0	0	0	15	0	0
t-SC-m	0	0	0	0	0	0	12	0
t-SC-s	0	0	0	0	0	0	0	14

which is as accurate as you can get. Again, I have not averaged over splits, which would be the next thing.

Naive bayes with normal class-conditional distributions takes an interesting and suggestive form. Assume we have two classes. Recall our decision rule is

$$\text{say } \begin{cases} + & \text{if } L(+ \to -)p(+|\mathbf{x}) > L(- \to +)p(-|\mathbf{x}) \\ - & \text{otherwise} \end{cases}$$

Now as p gets larger, so does $\log p$ (logarithm is a monotonically increasing function), and the rule isn't affected by adding the same constant to both sides, so we can rewrite as:

$$\text{say } \begin{cases} + & \text{if } \log L(+ \to -) + \log p(\mathbf{x}|+) + \log p(+) > \log L(- \to +) \log p(\mathbf{x}|-) + \log p(-) \\ - & \text{otherwise} \end{cases}$$

Write μ_j^+, σ_j^+ respectively for the mean and standard deviation for the class-conditional density for the j'th component of $\mathbf{x}$ for class + (and so on); the comparison becomes $\log L(+ \to -) - \sum_j \frac{(x_j - \mu_j^+)^2}{2(\sigma_j^+)^2} - \sum_j \log \sigma_j^+ + \log p(+) > \log L(- \to +) - \sum_j \frac{(x_j - \mu_j^-)^2}{2(\sigma_j^-)^2} - \sum_j \log \sigma_j^- + \log p(-)$ Now we can expand and collect terms really aggressively to get

$$\left(\sum_j c_j x_j^2 - d_j x_j \right) - e > 0$$

(where c_j, d_j, e are functions of the means and standard deviations and losses and priors). Rather than forming these by estimating the means, etc., we could directly search for good values of c_j, d_j and e.

3.3 THE SUPPORT VECTOR MACHINE

Assume we have a set of N example points $\mathbf{x}_i$ that belong to two classes, which we indicate by 1 and -1. These points come with their class labels, which we write as y_i; thus, our dataset can be written as

$$\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\}.$$

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

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

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 $\mathbf{a} \cdot \mathbf{x} + b = 0$. This hyperplane separates the positive data from the negative data, and is known as the **decision boundary**. Notice that the magnitude of $\mathbf{a} \cdot \mathbf{x} + b$ grows as the point $\mathbf{x}$ moves further away from the hyperplane.

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

Assume we use a linear model with one feature. Then the model has the form $y_i^{(p)} = \text{sign}(ax_i + b)$. For any particular example which has the feature value x^*, this means we will test whether x^* is larger than, or smaller than, $-b/a$.

Example: 3.2 *A linear model with two features*

Assume we use a linear model with two features. Then the model has the form $y_i^{(p)} = \text{sign}(\mathbf{a}^T\mathbf{x}_i + b)$. 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 fact, examples that are classified badly by the linear rule usually are classified badly because there are two 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, the error rate would be extremely small. 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 $\mathbf{x}$.

Recall that using naive bayes with a normal model for the class conditional distributions boiled down to testing $\left(\sum_j c_j x_j^2 - d_j x_j\right) - e > 0$ for some values of c_j, d_j, and e. This may not look to you like a linear classifier, but it is. Imagine that, for an example $\mathbf{u}_i$, you form the feature vector

$$\mathbf{x} = \left(u_{i,1}^2, u_{i,1}, u_{i,2}^2, u_{i,2}, \ldots, u_{i,d}\right)^T.$$

Then we can interpret testing $\mathbf{a}^T\mathbf{x} + b > 0$ as testing $a_1 u_{i,1}^2 - (-a_2)u_{i,1} + a_3 u_{i,2}^2 - (-a_4)u_{i,2} + ... - (-b) > 0$, and pattern matching to the expression for naive bayes suggests that the two cases are equivalent (i.e. for any choice of $\mathbf{a}$, b, there is a corresponding naive bayes case and vice versa; exercises).

3.3.1 Choosing a Classifier with the Hinge Loss

We will choose $\mathbf{a}$ and b by choosing values that minimize a cost function. We will adopt a cost function of the form:

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

For the moment, we will ignore the penalty term and focus on the training error cost. 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 $(\mathbf{a} \cdot \mathbf{x} + b)$ 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 C should not be zero in this case. Finally, if γ_i and y_i have the same

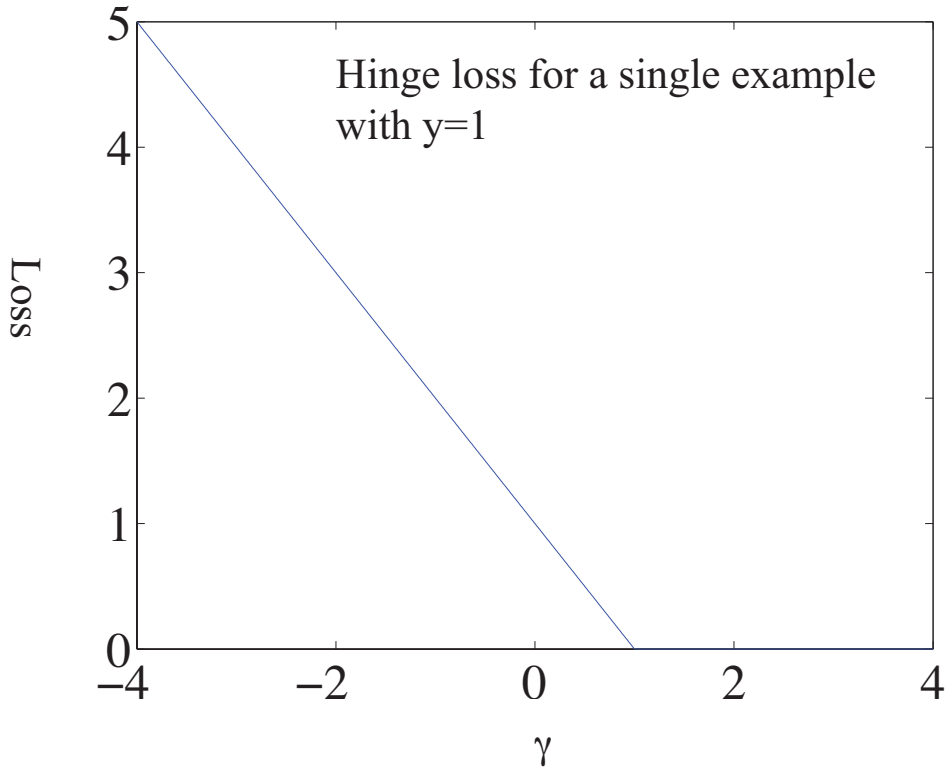

FIGURE 3.1: *The hinge loss, plotted for the case $y_i = 1$. The horizontal variable is the $\gamma_i = \mathbf{a} \cdot \mathbf{x}_i + b$ of the text. Notice that giving a strong negative response to this positive example causes a loss that grows linearly as the magnitude of the response grows. Notice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

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 $\mathbf{x}_i$.

The choice

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

has these properties. 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. But in any other case, there is a cost. The cost rises if $\mathbf{x}_i$ moves toward the decision boundary from the correct side, and grows linearly as $\mathbf{x}_i$ moves further away from the boundary on the wrong side (Figure 3.1). This means that minimizing the loss will encourage the classifier to (a) make strong positive (or negative) predictions for positive (or negative) examples and (b) for examples it gets wrong, make the most positive (negative) prediction that it can. This choice is known as the **hinge loss**.

Now we 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 an example

with feature vector $\mathbf{x}$ will be classified with the rule sign $((\mathbf{)a} \cdot \mathbf{x} + b)$. If we classify this example wrongly, we should like $|\mathbf{a} \cdot \mathbf{x} + b|$ to be small. Achieving this would mean that at least some nearby examples will have the right sign. The way to achieve this is to ensure that $\|\mathbf{a}\|$ is small. By this argument, we would like to achieve a small value of the hinge loss using a small value of $\|\mathbf{a}\|$. Thus, we add a penalty term to the loss so that pairs $(\mathbf{a}, b)$ that have small values of the hinge loss and large values of $\|\mathbf{a}\|$ are expensive. We minimize

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

where λ is some weight that balances the importance of a small hinge loss against the importance of a small $\|\mathbf{a}\|$. There are now two problems to solve. First, assume we know λ; we will need to find $\mathbf{a}$ and $\mathbf{b}$ that minimize $S(\mathbf{a}, b; \lambda)$. Second, we will need to estimate λ.

3.3.2 Finding a Minimum: General Points

I will first summarize general recipes for finding a minimum. 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 this problem in closed form by constructing the gradient and finding a value of $\mathbf{u}$ the makes the gradient zero. This happens mainly for specially chosen problems that occur in textbooks. For practical problems, we tend to need a numerical method.

Typical methods take a point $\mathbf{u}^{(i)}$, update it to $\mathbf{u}^{(i+1)}$, 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 it won't matter. The update is usually obtained by computing a direction $\mathbf{p}^{(i)}$ such that for small values of h, $g(\mathbf{u}^{(i)} + h\mathbf{p}^{(i)})$ is smaller than $g(\mathbf{u}^{(i)})$. 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**.

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

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

and that

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

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

$$g(\mathbf{u}^{(i)} + h\mathbf{p}^{(i)}) = g(\mathbf{u}^{(i)}) + h(\nabla g)^T\mathbf{p}^{(i)} + O(h^2)$$

This means that we can expect that if

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

we expect that, at least for small values of h, $g(\mathbf{u}^{(i)}+h\mathbf{p}^{(i)})$ will be less than $g(\mathbf{u}^{(i)})$. 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.

3.3.3 Finding a Minimum: Stochastic Gradient Descent

Assume we wish to minimize some function $g(\mathbf{u}) = g_0(\mathbf{u}) + (1/N)\sum_{i=1}^{N} g_i(\mathbf{u})$, as a function of $\mathbf{u}$. Gradient descent would require us to form

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

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 of examples. For some cases, there might be trillions of examples. Touching each example at each step really is impractical.

Instead, assume that, at each step, we choose a number k in the range $1 \ldots N$ uniformly and at random, and form

$$\mathbf{p}_k = -\left(\nabla g_0(\mathbf{u}) + \nabla g_k(\mathbf{u})\right)$$

and then take a small step along $\mathbf{p}_k$. Our new point becomes

$$\mathbf{a}^{(i+1)} = \mathbf{a}^{(i)} + \eta\mathbf{p}_k^{(i)},$$

where η is called the **steplength** (even though it very often isn't the length of the step we take!). It is easy to show that

$$\mathbb{E}[\mathbf{p}_k] = \nabla g(\mathbf{u})$$

(where the expectation is over the random choice of k). This implies that if we take many small steps along $\mathbf{p}_k$, 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 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 a steplength 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. One useful strategy is to divide training into **epochs**. Each epoch is a block of a fixed number of iterations. Each iteration is one of the steps given above, with fixed steplength. However, the steplength changes from epoch to epoch. In particular, in the r'th epoch, the steplength is

$$\eta^{(r)} = \frac{a}{r+b}$$

where a and b are constants chosen by experiment with small subsets of the dataset.

One cannot really test whether stochastic gradient descent has converged to the right answer. A better approach is to plot the error as a function of epoch on a validation set. This should vary randomly, but generally go down as the epochs proceed. I have summarized this discussion in box 3.1. You should be aware that the recipe there admits many useful variations, though.

Procedure: 3.1 *Stochastic Gradient Descent*

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, either 1 or -1. Choose a set of possible values of the regularization weight λ. We wish to train a model that minimizes a cost function of the form $g(\mathbf{u}) = \frac{\lambda}{2}\mathbf{u}^T\mathbf{u} + (\frac{1}{N})\sum_{i=1}^{N} g_i(\mathbf{u})$. Separate the data into three sets: test, training and validation. For each value of the regularization weight, train a model, and evaluate the model on validation data. Keep the model that produces the lowest error rate on the validation data, and report its performance on the test data.

Train a model by choosing a fixed number of epochs N_e, and the number of steps per epoch N_s. Choose a random start point, $\mathbf{u}_0 = [\mathbf{a}, b]$. For each epoch, first compute the steplength. In the e'th epoch, the steplength is typically $\eta = \frac{1}{ae+b}$ for constants a and b chosen by small-scale experiments (you try training a model with different values and see what happens). For the e'th epoch, choose a subset of the training set for validation for that epoch. Now repeat until the model has been updated N_s times:

- Take k steps. Each step is taken by selecting a single data item uniformly and at random. Assume we select the i'th data item. We then compute $\mathbf{p} = -\nabla g_i(\mathbf{u}) - \lambda\mathbf{u}$, and update the model by computing
$$\mathbf{u}_{n+1} = \mathbf{u}_n + \eta\mathbf{p}$$
- Evaluate the current model by computing the accuracy on the validation set for that epoch. Plot the accuracy as a function of step number.

3.3.4 Example: Training a Support Vector Machine with Stochastic Gradient Descent

I have summarized stochastic gradient descent in algorithm 3.1, but here is an example in more detail. We need to choose $\mathbf{a}$ and b to minimize

$$C(\mathbf{a}, b) = (1/N) \sum_{i=1}^{N} \max(0, 1 - y_i (\mathbf{a} \cdot \mathbf{x}_i + b)) + \frac{\lambda}{2} \mathbf{a}^T \mathbf{a}.$$

This is a support vector machine, because it uses hinge loss. For a support vector machine, stochastic gradient descent is particularly easy. We have estimates $\mathbf{a}^{(n)}$ and $b^{(n)}$ of the classifier parameters, and we want to improve the estimates. We pick the k'th example at random. We must now compute

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

Assume that $y_k (\mathbf{a} \cdot \mathbf{x}_k + b) > 1$. 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 $y_k (\mathbf{a} \cdot \mathbf{x}_k + b) < 1$, we can ignore the max, and the first term is $1 - y_k (\mathbf{a} \cdot \mathbf{x}_k + b)$; the gradient is again easy. But what if $y_k (\mathbf{a} \cdot \mathbf{x}_k + b) = 1$? there are two distinct values we could choose for the gradient, because the max term isn't differentiable. It turns out not to matter which term we choose (Figure **??**), so we can write the gradient as

$$p_k = \begin{cases} \begin{bmatrix} \lambda \mathbf{a} \\ 0 \end{bmatrix} & \text{if } y_k (\mathbf{a} \cdot \mathbf{x}_k + b) \geq 1 \\ \begin{bmatrix} \lambda \mathbf{a} - y_k \mathbf{x} \\ -y_k \end{bmatrix} & \text{otherwise} \end{cases}$$

We choose a steplength η, and update our estimates using this gradient. This yields:

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

and

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

To construct figures, 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). I used two different training regimes.

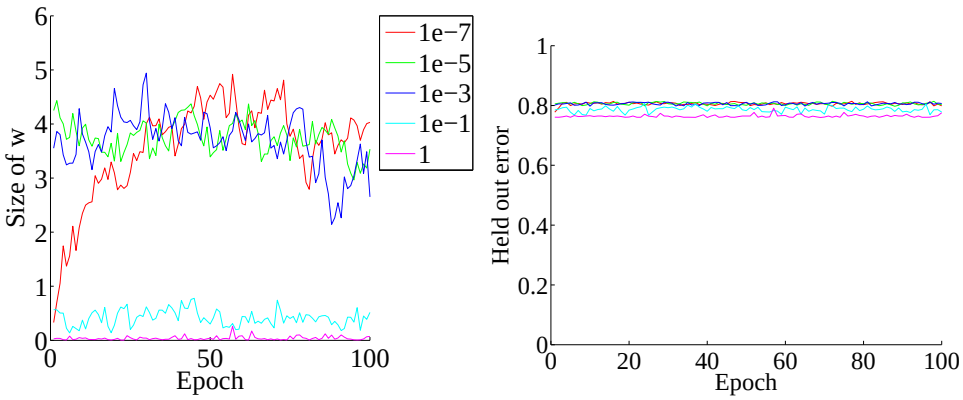

FIGURE 3.2: *On the* **left**, *the magnitude of the weight vector* $\mathbf{a}$ *at the end of each epoch for the first training regime described in the text. On the* **right**, *the accuracy on held out data at the end of each epoch. Notice how different choices of regularization parameter lead to 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.*

In the first training regime, there were 100 epochs. In each epoch, 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 $1/(0.01 * e + 50)$, where e is the epoch. At the end of each epoch, I computed $\mathbf{a}^T\mathbf{a}$ and the accuracy (fraction of examples correctly classified) of the current classifier on the held out test examples. Figure 3.2 shows the results. You should notice that the accuracy changes slightly each epoch; 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 epochs. In each epoch, 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, 216 unique data items — it hasn't seen the whole training set. I chose 5 different values for the regularization parameter and trained with a steplength of $1/(0.01 * e + 50)$, where e is the epoch. At the end of each epoch, I computed $\mathbf{a}^T\mathbf{a}$ and the accuracy (fraction of examples correctly classified) of the current classifier on the held out test examples. Figure 3.3 shows the results. You should notice that the accuracy changes slightly each epoch; that for larger regularizer values $\mathbf{a}^T\mathbf{a}$ is smaller; and that the accuracy settles down to about 0.8 very quickly; and that there isn't much difference between the two training regimes. All of these points are relatively typical of stochastic gradient

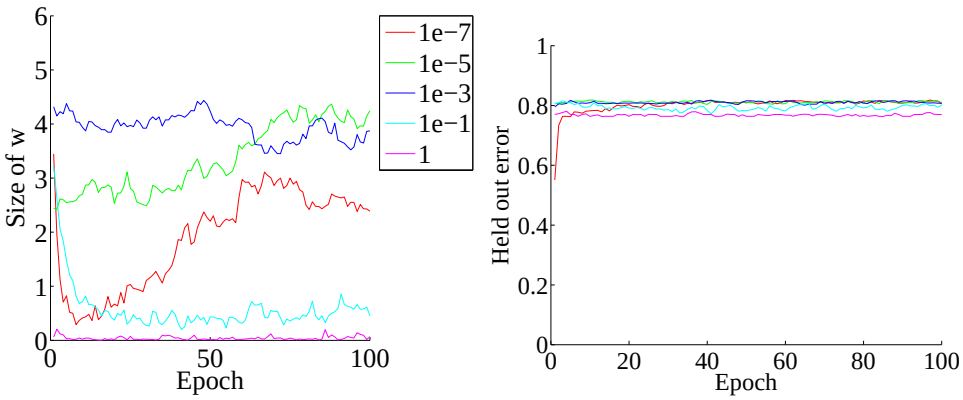

FIGURE 3.3: *On the* **left**, *the magnitude of the weight vector* $\mathbf{a}$ *at the end of each epoch for the second training regime described in the text. On the* **right**, *the accuracy on held out data at the end of each epoch. Notice how different choices of regularization parameter lead to 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.*

descent with very large datasets.

> **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. There is an immense quantity of good software available.*

3.3.5 Multi-Class Classifiers

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

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

In the **one-vs-all** approach, we build a binary classifier for each class. This classifier must distinguish its class from all the other classes. We then take the class

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

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

3.4 CLASSIFYING WITH RANDOM FORESTS

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

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

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

3.4.1 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

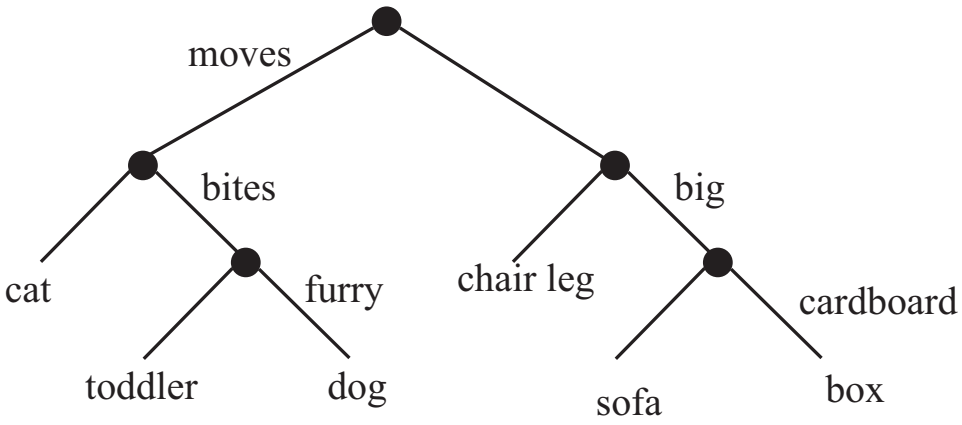

FIGURE 3.4: *This — the household robot's guide to obstacles — is a typical decision tree. I have labelled only one of the outgoing branches, because the other is the negation. So if the obstacle moves, bites, but isn't furry, then it's a toddler. In general, an item is passed down the tree until it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

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 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

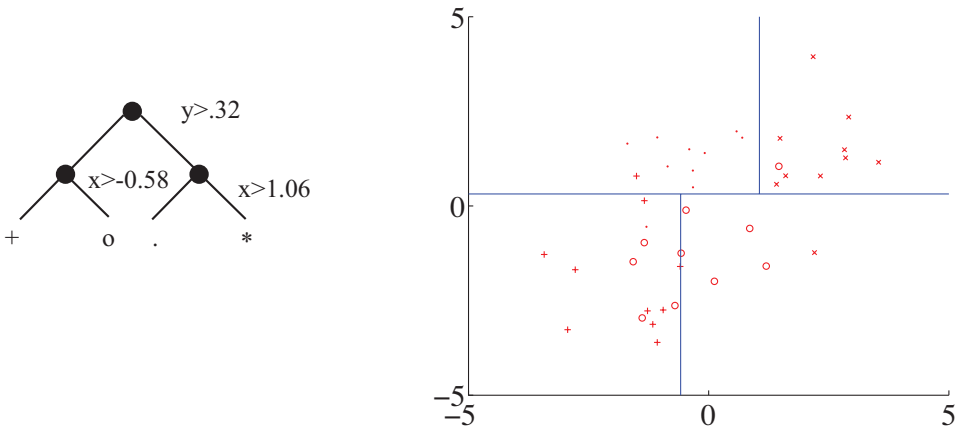

FIGURE 3.5: *A straightforward decision tree, illustrated in two ways. On the* **left**, *I have given the rules at each split; on the* **right**, *I have shown the data points in two dimensions, and the structure that the tree produces in the feature space.*

case by testing the amount of data against a threshold, chosen by experiment. If all the data at a node belongs to a single class, there is no point in splitting. Finally, constructing a tree that is too deep tends to result in generalization problems, so we usually allow no more than a fixed depth D of splits. Choosing the best splitting threshold is more complicated.

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

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

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

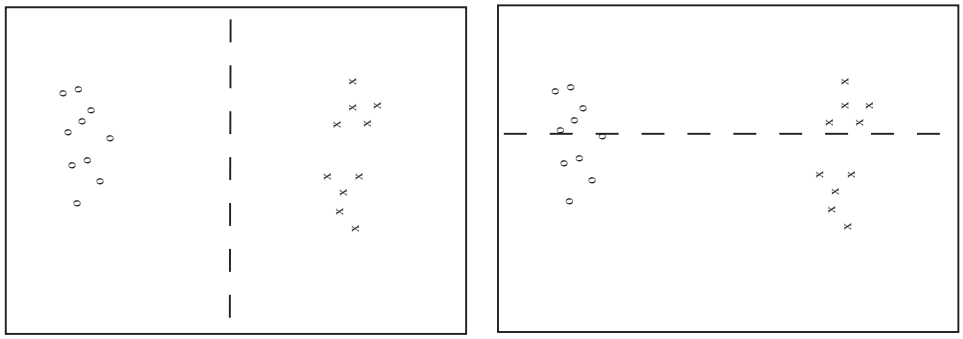

FIGURE 3.6: *Two possible splits of a pool of training data. Positive data is represented with an 'x', negative data with a 'o'. Notice that if we split this pool with the informative line, all the points on the left are 'o's, and all the points on the right are 'x's. This is an excellent choice of split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

split means that my uncertainty about what class the data item belongs to is significantly reduced if I know whether it goes left or right. To choose a good threshold, we need to keep track of how informative the split is.

3.4.2 Entropy and Information Gain

It turns out to be straightforward to keep track of information, in simple cases. We will start with an example. Assume I have 4 classes. There are 8 examples in class 1, 4 in class 2, 2 in class 3, and 2 in class 4. How much information *on average* will you need to send me to tell me the class of a given example? Clearly, this depends on how you communicate the information. You could send me the complete works of Edward Gibbon to communicate class 1; the Encyclopaedia for class 2; and so on. But this would be redundant. The question is how little can you send me. Keeping track of the amount of information is easier if we encode it with bits (i.e. you can send me sequences of '0's and '1's).

Imagine the following scheme. If an example is in class 1, you send me a '1'. If it is in class 2, you send me '01'; if it is in class 3, you send me '001'; and in class 4, you send me '101'. Then the expected number of bits you will send me is

$$p(\text{class} = 1)1 + p(2)2 + p(3)3 + p(4)3 = \frac{1}{2}1 + \frac{1}{4}2 + \frac{1}{8}3 + \frac{1}{8}3$$

which is 1.75 bits. This number doesn't have to be an integer, because it's an expectation.

Notice that for the i'th class, you have sent me $-\log_2 p(i)$ bits. We can write

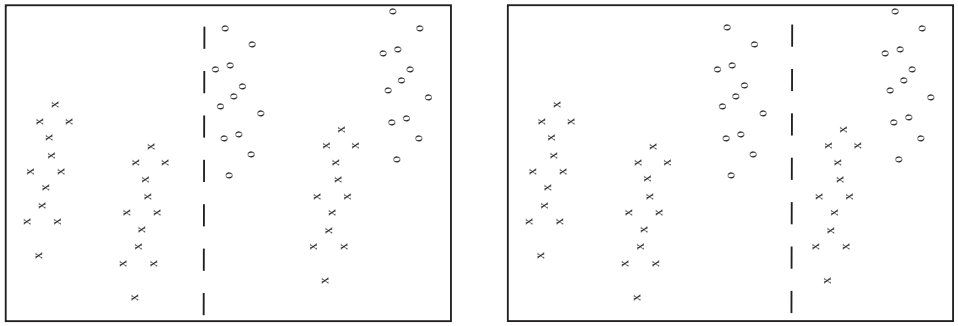

FIGURE 3.7: *Two possible splits of a pool of training data. Positive data is represented with an 'x', negative data with a 'o'. Notice that if we split this pool with the informative line, all the points on the left are 'x's, and two-thirds of the points on the right are 'o's. This means that knowing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

the expected number of bits you need to send me as

$$-\sum_i p(i) \log_2 p(i).$$

This expression handles other simple cases correctly, too. You should notice that it isn't really important *how many* objects appear in each class. Instead, the *fraction* of all examples that appear in the class is what matters. This fraction is the prior probability that an item will belong to the class. You should try what happens if you have two classes, with an even number of examples in each; 256 classes, with an even number of examples in each; and 5 classes, with $p(1) = 1/2$, $p(2) = 1/4$, $p(3) = 1/8$, $p(4) = 1/16$ and $p(5) = 1/16$. If you try other examples, you may find it hard to construct a scheme where you can send as few bits *on average* as this expression predicts. It turns out that, in general, the smallest number of bits you will need to send me is given by the expression

$$-\sum_i p(i) \log_2 p(i)$$

under all conditions, though it may be hard or impossible to determine what representation is required to achieve this number.

The **entropy** of a probability distribution is a number that scores how many bits, on average, would need to be known to identify an item sampled from that probability distribution. For a discrete probability distribution, the entropy is computed as

$$-\sum_i p(i) \log_2 p(i)$$

where i ranges over all the numbers where $p(i)$ is not zero. For example, if we have two classes and $p(1) = 0.99$, then the entropy is 0.0808, meaning you need very little information to tell which class an object belongs to. This makes sense, because there is a very high probability it belongs to class 1; you need very little information to tell you when it is in class 2. If you are worried by the prospect of having to send 0.0808 bits, remember this is an average, so you can interpret the number as meaning that, if you want to tell which class each of 10^4 independent objects belong to, you could do so in principle with only 808 bits.

Generally, the entropy is larger if the class of an item is more uncertain. Imagine we have two classes and $p(1) = 0.5$, then the entropy is 1, and this is the largest possible value for a probability distribution on two classes. You can always tell which of two classes an object belongs to with just one bit (though you might be able to tell with even less than one bit).

3.4.3 Entropy and Splits

Now we return to the splits. 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 $\mathcal{C}$ that 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 is $H(\mathcal{C})$ 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

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

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.

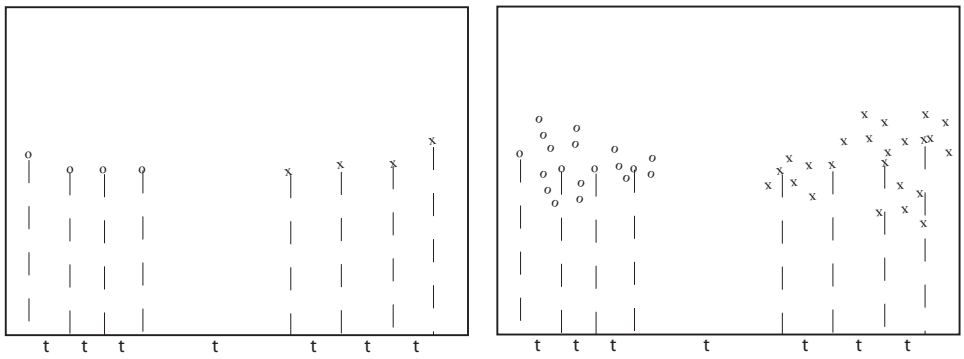

FIGURE 3.8: *We search for a good splitting threshold by looking at values of the chosen component that yield different splits. On the* **left**, *I show a small dataset and its projection onto the chosen splitting component (the horizontal axis). For the 8 data points here, there are only 7 threshold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

3.4.4 Choosing a Split with 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.

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

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

Now assume we happen to have chosen to work with a feature that isn't ordinal, and so can't be tested against a threshold. A natural, and effective, strategy

is as follows. We can split such a feature into two pools by flipping an unbiased coin for each value — if the coin comes up H, any data point with that value goes left, and if it comes up T, any data point with that value goes right. We chose this split at random, so it might not be any good. We can come up with a good split by repeating this procedure F times, computing the information gain for each split, then keeping the one that has the best information gain. We choose F in advance, and it usually depends on the number of values the categorical variable can take.

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

Procedure: 3.2 *Building a decision tree*

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 the best splitting threshold. Do so by selecting a set of possible values for the threshold, then for each value splitting the dataset (every data item with a value of the component below the threshold goes left, others go right), and computing 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.

3.4.5 Forests

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

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

3.4.6 Building and Evaluating a Decision Forest

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

Procedure: 3.3 *Building a decision forest*

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

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

Procedure: 3.4 *Building a decision forest using bagging*

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

3.4.7 Classifying Data Items with a Decision Forest

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

Procedure: 3.5 *Classification with a decision forest*

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

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

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

Worked example 3.3 *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. There is sample code in listing **??**. 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	Predict 1	Predict 2	Predict 3	Predict 4	Class error
True 0	151	7	2	3	1	7.9%
True 1	32	5	9	9	0	91%
True 2	10	9	7	9	1	81%
True 3	6	13	9	5	2	86%
True 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 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	Predict 1	Class error
True 0	138	26	16%
True 1	31	108	22%

Notice that the false positive rate (16%, from 26/164) is rather better than the false negative rate (22%). Looking at these class confusion matrices, you might wonder whether it is better to predict $0, \ldots, 4$, then quantize. But this is not a particularly good idea. While the false positive rate is 7.9%, the false negative rate 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.

Listing 3.1: R code used for the random forests of worked example 3.3

```
setwd('/users/daf/Current/courses/Probcourse/Trees/RCode');
install.packages('randomForest')
library(randomForest)
heart<-read.csv('processed.cleveland.data.txt', header=FALSE)
heart$levels<-as.factor(heart$V14)
heartforest.allvals<-
  randomForest(formula=levels~V1+V2+V3+V4+V5+V6
               +V7+V8+V9+V10+V11+V12+V13,
               data=heart, type='classification', mtry=5)
# this fits to all levels
# I got the CCM by typing
heartforest.allvals
heart$yesno<-cut(heart$V14, c(-Inf, 0.1, Inf))
heartforest<-
  randomForest(formula=yesno~V1+V2+V3+V4+V5+V6
               +V7+V8+V9+V10+V11+V12+V13,
               data=heart, type='classification', mtry=5)
# this fits to the quantized case
# I got the CCM by typing
heartforest
```

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

3.5 CLASSIFYING WITH NEAREST NEIGHBORS

Generally we expect that, if two points $\mathbf{x}_i$ and $\mathbf{x}_j$ are close, then their labels will be the same most of the time. If we were unlucky, the two points could lie on either side of a decision boundary, but most pairs of nearby points will have the same labels.

This observation suggests the extremely useful and general strategy of exploiting a data item's neighbors. If you want to classify a data item, find the closest example, and report the class of that example. Alternatively, you could find the closest k examples, and vote. Imagine we have a data point $\mathbf{x}$ that we wish to classify (a query point). Our strategy will be to find the closest training example, and report its class.

How well can we expect this strategy to work? Remember that any classifier will slice up the space of examples into cells (which might be quite complicated) where every point in a cell has the same label. The boundaries between cells are decision boundaries — when a point passes over the decision boundary, the label changes. Now assume we have a large number of labelled training examples, and we know the best possible set of decision boundaries. If there are many training examples, there should be at least one training example that is close to the query

point. If there are enough training examples, then the closest point should be inside the same cell as the query point.

You may be worried that, if the query point is close to a decision boundary, the closest point might be on the other side of that boundary. But if it were, we could improve things by simply having more training points. All this suggests that, *with enough training points*, our classifier should work about as well as the best possible classifier. This intuition turns out to be correct, though the number of training points required is wholly impractical, particularly for high-dimensional feature vectors.

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). A $(k, 0)$-nearest neighbor classifier is usually known as a **k-nearest neighbor classifier**, and a $(1, 0)$-nearest neighbor classifier is usually known as a **nearest neighbor classifier**. In practice, one seldom uses more than three nearest neighbors. Finding the k nearest points for a particular query can be difficult, and Section **??** reviews this point.

There are three practical difficulties in building nearest neighbor classifiers. You need a lot of labelled examples. You need to be able to find the nearest neighbors for your query point. And 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? One possibility is to whiten the features (section 4.1). This may be hard if the dimension is so large that the covariance matrix is hard to estimate. 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. Notice that nearest neighbors (fairly obviously) doesn't like categorical data. If you can't give a clear account of how far apart two things are, you shouldn't be doing nearest neighbors. It is possible to fudge this point a little, by (say) putting together a distance between the levels of each factor, but it's probably unwise.

Nearest neighbors is wonderfully flexible about the labels the classifier predicts. Nothing changes when you go from a two-class classifier to a multi-class classifier.

Worked example 3.4 *Classifying using nearest neighbors*

Build a nearest neighbor classifier to classify the digit data originally constructed by Yann Lecun. You can find it at 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: 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. This uses a bagging strategy. There is sample code in listing **??**. I trained on 1000 of the 42000 examples, so you could see how in the code. I tested on the next 200 examples. For this (rather small) case, I found the following class confusion matrix

	P 0	1	2	3	4	5	6	7	8	9
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 was in a rush and couldn't recall the magic line of R to get them. However, you can see the classifier works rather well for this case.

Remember this: *Nearest neighbor classifiers are often very effective. They can predict any kind of label. You do need to be careful to have enough data, and to have a meaningful distance function.*

3.6 YOU SHOULD

3.6.1 be able to:

- construct a naive bayes classifier for continuous and discrete features
- train a linear SVM with stochastic gradient descent and evaluate the resulting classifier
- train a random forest using a package and evaluate the resulting classifier
- train and evaluate a nearest neighbors classifier

3.6.2 remember:

New term: classifier . . . 10
New term: feature vectors . . . 10
Definition: Classifier . . . 10
New term: decision boundaries . . . 12
New term: overfitting . . . 13
New term: selection bias . . . 13
New term: generalizing badly . . . 13
Do not evaluate a classifier on training data. . . . 13
New term: validation set . . . 13
New term: test set . . . 13
New term: class confusion matrix . . . 14
New term: Vapnik-Chervonenkis dimension . . . 15
New term: V-C dimension . . . 15
New term: likelihood . . . 16
New term: class conditional probability . . . 16
New term: prior . . . 16
New term: posterior . . . 16
New term: decision boundary . . . 20
New term: descent direction . . . 23
New term: line search . . . 23
New term: gradient descent . . . 23
New term: steplength . . . 24
New term: stochastic gradient descent . . . 24
Linear SVM's are a go-to classifier. . . . 28
Any SVM package should build a multi-class classifier for you. . . . 29
New term: decision tree . . . 29
New term: decision forest . . . 29
New term: decision function . . . 30
New term: entropy . . . 33
New term: entropy . . . 34
New term: information gain . . . 34
New term: bagging . . . 37
New term: bag . . . 37
Random forests are good and easy. . . . 40
Nearest neighbors are good and easy. . . . 42

PROBLEMS

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. For several attributes in this data set, a value of 0 may indicate a missing value of the variable.

- **(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 (it's quite straightforward), but you may find the function `createDataPartition` in the R package `caret` helpful to get the random partition.
- **(b)** Now adjust your code so that, for attribute 3 (Diastolic blood pressure), attribute 4 (Triceps skin fold thickness), attribute 6 (Body mass index), and attribute 8 (Age), it regards a value of 0 as a missing value when estimating the class-conditional distributions, and the posterior. R uses a special number `NA` to flag a missing value. Most functions handle this number in special, but sensible, ways; but you'll need to do a bit of looking at manuals to check. Does this affect the accuracy of your classifier?
- **(c)** Now use the `caret` and `klaR` packages to build a naive bayes classifier for this data, assuming that no attribute has a missing value. 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. I have not been able to persuade the combination of `caret` and `klaR` to handle missing values the way I'd like them to, but that may be ignorance (look at the `na.action` argument).
- **(d)** 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. You should NOT substitute `NA` values for zeros for attributes 3, 4, 6, and 8.

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, EUROSIS, ISBN 978-9077381-39-7.

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 search for an appropriate value of the regularization constant, trying at least the values $\lambda = [1e-3, 1e-2, 1e-1, 1]$. Use the validation set for this search
You should use at least 50 epochs of at least 100 steps each. In each epoch, you should separate out 50 training examples at random for evaluation. You should compute the accuracy of the current classifier on the set held out for the epoch every 10 steps. You should produce:

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

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

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

3.5. 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 (¿=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 search for an appropriate value of the regularization constant, trying at least the values $\lambda = [1e-3, 1e-2, 1e-1, 1]$. Use the validation set for this search
You should use at least 50 epochs of at least 300 steps each. In each epoch, you should separate out 50 training examples at random for evaluation. You should compute the accuracy of the current classifier on the set held out for the epoch every 30 steps. You should produce:

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

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

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

3.6. 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. (2009) Predicting Positive p53 Cancer Rescue Regions Using Most Informative Positive (MIP) Active Learning, PLOS Computational Biology, 5(9); Danziger, S.A., Zeng, J., Wang, Y., Brachmann, R.K. and Lathrop, R.H. (2007) Choosing where to look next in a mutation sequence space: Active Learning of informative p53 cancer rescue mutants, Bioinformatics, 23(13), 104-114; 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. (2006) Functional

census of mutation sequence spaces: the example of p53 cancer rescue mutants, IEEE/ACM transactions on computational biology and bioinformatics / IEEE, ACM, 3, 114-125.

You can find this data at https://archive.ics.uci.edu/ml/datasets/p53+Mutants. There are a total of 16772 instances, with 5409 attributes per instance. Attribute 5409 is the class attribute, which is either active or inactive. There are several versions of this dataset. You should use the version `K8.data`.

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

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

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

3.7. 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?

CODE SNIPPETS

Listing 3.2: R code used for the naive bayes example of worked example 3.1

```
setwd('/users/daf/Current/courses/Probcourse/Classification/RCode/BreastTissue')
wdat<-read.csv('cleanedbreast.csv')
library(klaR)
library(caret)
bigx<-wdat[,-c(1:2)]
bigy<-wdat[,2]
wtd<-createDataPartition(y=bigy, p=.8, list=FALSE)
trax<-bigx[wtd,]
tray<-bigy[wtd]
modelk<-train(trax, tray, 'nb', trControl=trainControl(method='cv', number=10))
teclasses<-predict(model,newdata=bigx[-wtd,])
confusionMatrix(data=teclasses, bigy[-wtd])
```

Listing 3.3: R code used for the naive bayes example of worked example 3.2

```
setwd('/users/daf/Current/courses/Probcourse/Classification/RCode/MouseProtein')
wdat<-read.csv('Data_Cortex_Nuclear.csv')
#install.packages('klaR')
#install.packages('caret')
library(klaR)
library(caret)
cci<-complete.cases(wdat)
bigx<-wdat[cci,-c(82)]
bigy<-wdat[cci,82]
wtd<-createDataPartition(y=bigy, p=.8, list=FALSE)
trax<-bigx[wtd,]
tray<-bigy[wtd]
modelk<-train(trax, tray, 'nb', trControl=trainControl(method='cv', number=10))
teclasses<-predict(model,newdata=bigx[-wtd,])
confusionMatrix(data=teclasses, bigy[-wtd])
```

CHAPTER 4

Extracting Important Relationships in High Dimensions

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

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

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

4.1 SOME PLOTS OF HIGH DIMENSIONAL DATA

4.1.1 Understanding Blobs with Scatterplot Matrices - CLEANUP

Plotting high dimensional data is tricky.

4.1.2 Parallel Plots

Parallel plots can sometimes reveal information, particularly when the dimension of the dataset is low. To construct a parallel plot, you compute a normalized representation of each component of each data item. The component is normalized by translating and scaling so that the minimum value over the dataset is zero, and the maximum value over the dataset is one. Now write the i'th normalised data item as $(n_1, n_2, \ldots, n_d)$. For this data item, you plot a broken line joining $(1, n_1)$ to $(2, n_2)$ to $(3, n_3$, etc. These plots are superimposed on one another. In the case of the bodyfat dataset, this yields the plot of figure 4.1.

Some structures in the parallel plot are revealing. Outliers often stick out (in figure 4.1, it's pretty clear that there's a data point with a very low height value, and also one with a very large weight value). Outliers affect the scaling, and so make other structures difficult to spot. I have removed them for figure 4.2. In this figure, you can see that two negatively correlated components next to one another produce a butterfly like shape (bodyfat and density). In this plot, you can also see that there are still several data points that are very different from others (two data

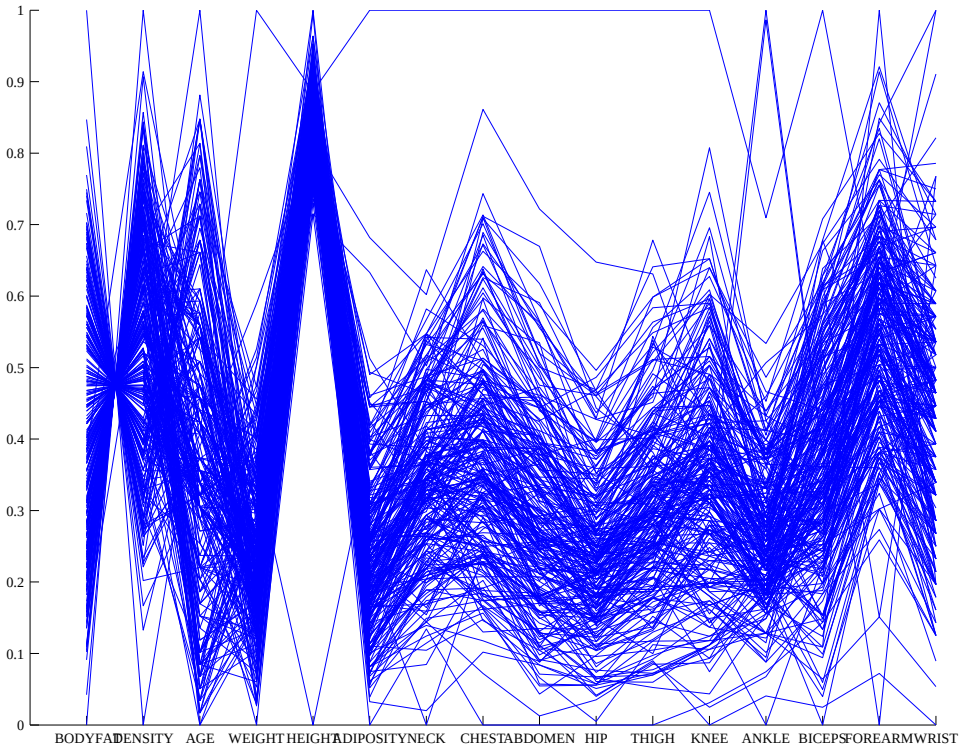

FIGURE 4.1: *A parallel plot of the bodyfat dataset, including all data points. I have named the components on the horizontal axis. It is easy to see that large values of bodyfat correspond to small values of density, and vice versa. Notice that one datapoint has height very different from all others; similarly, one datapoint has weight very different from all others.*

items have ankle values that are very different from the others, for example).

4.1.3 Scatterplot Matrices

One 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 of variables in a matrix. On the diagonal, you name the variable that is the vertical axis for each plot in the row, and the horizontal axis in the column. This sounds more complicated than it is; look at the example of figure 4.3, which shows a scatterplot 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.

What is nice about this kind of plot is that it's quite easy to spot correlations between pairs of variables, though you do need to take into account the coordinates have not been normalized. For figure 4.3, you can see that weight and adiposity

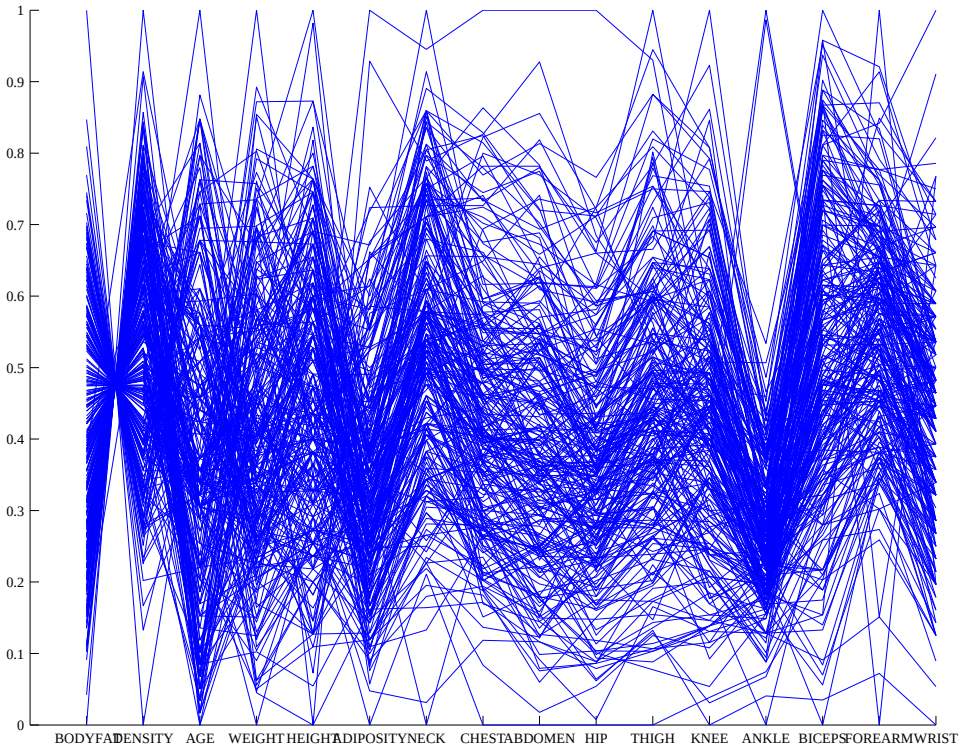

FIGURE 4.2: *A plot with those data items removed, so that those components are renormalized. Two datapoints have rather distinct ankle measurements. Generally, you can see that large knees go with large ankles and large biceps (the v structure).*

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. To show what might happen, figure 4.4 shows a scatter plot matrix with some points shown as circles. Notice how they lie inside the "blob" of data in some views, and outside in others. This is an effect of projection.

UC Irvine keeps a large repository of datasets that are important in machine learning. You can find the repository at http://archive.ics.uci.edu/ml/index.html. Figures 4.5 and 4.6 show visualizations of a famous dataset to do with the botanical classification of irises.

Figures **??**, **??** and 4.9 show visualizations of another dataset to do with forest fires in Portugal, also from the UC Irvine repository (look at http://archive.ics.uci.edu/ml/datasets/Forest+Fires). In this dataset, there are a variety of measurements of location, time, temperature, etc. together with the area burned by a wildfire. It would be nice to know what leads to large fires, and a visualization is the place to start. Many fires are tiny (or perhaps there was no area measurement?) and so

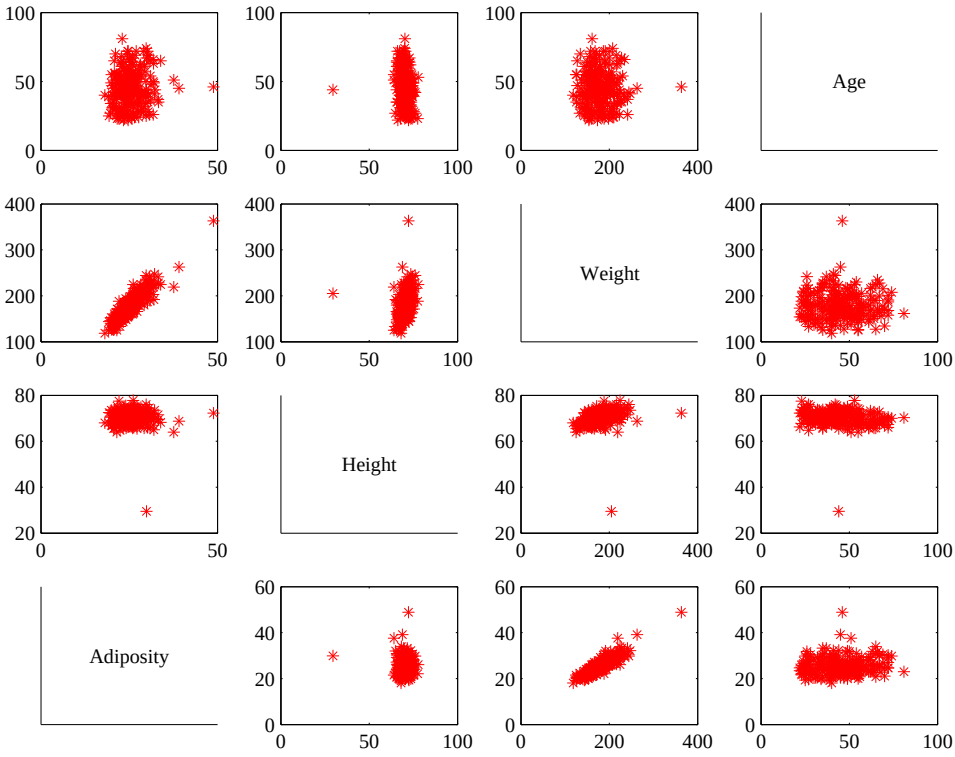

FIGURE 4.3: *This is a scatterplot matrix for four of the variables in the height weight dataset of http://www2.stetson.edu/~jrasp/data.htm. Each plot is a scatterplot of a pair of variables. The name of the variable for the horizontal axis is obtained by running your eye down the column; for the vertical 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).*

many values of the area are zero. I found it helpful to take the log of area, and then to divide the values of the logarithm into seven categories. I ignored the first four variables, because I didn't think they'd be too important. **Exercise:** was I right? I made two scatterplot matrices, because an eight by eight matrix is too big to view. Generally, this visualization suggests that it would be hard to predict the size of a fire from these variables.

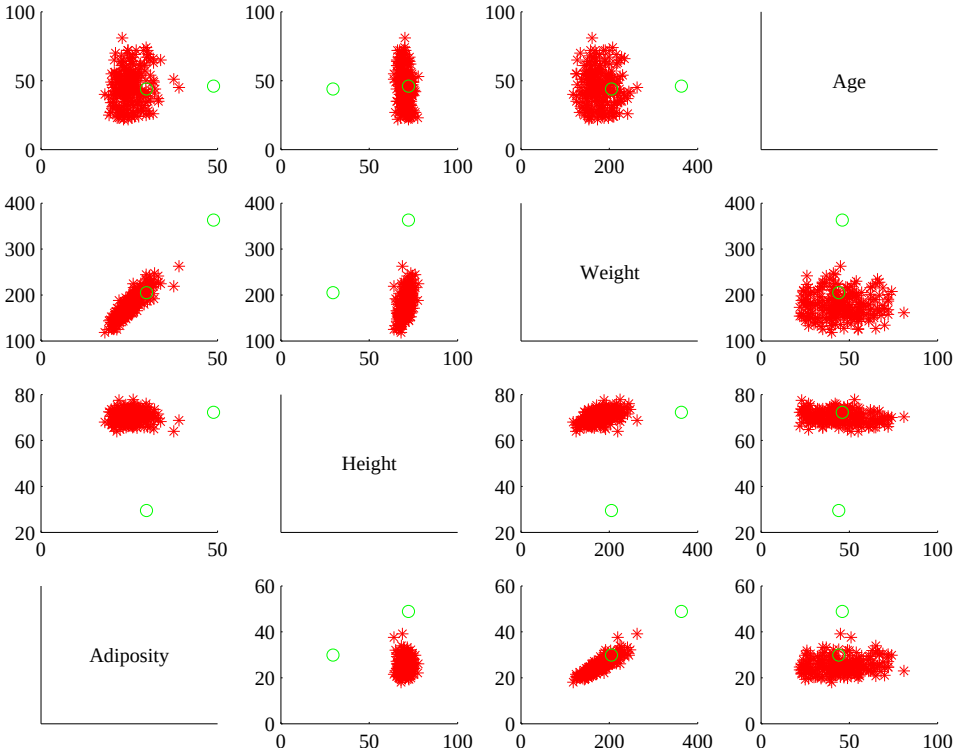

FIGURE 4.4: *You should compare this figure with figure 4.3. I have marked two data points with circles in this figure; notice that in some panels these are far from the rest of the data, in others close by. A "brush" in an interactive application can be used to mark data like this to allow a user to explore a dataset.*

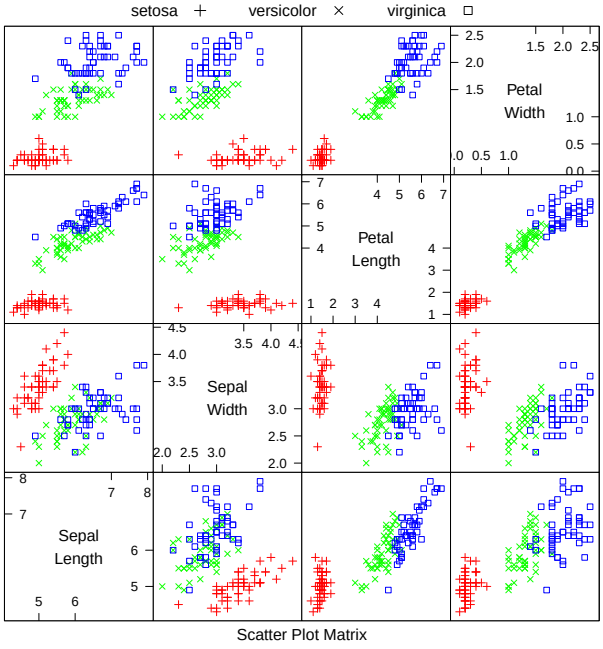

FIGURE 4.5: *This is a scatterplot matrix for the famous Iris data, originally due to ***. There are four variables, measured for each of three species of iris. I have plotted each species with a different marker. You can see from the plot that the species cluster quite tightly, and are different from one another. R code for this plot is on the website.*

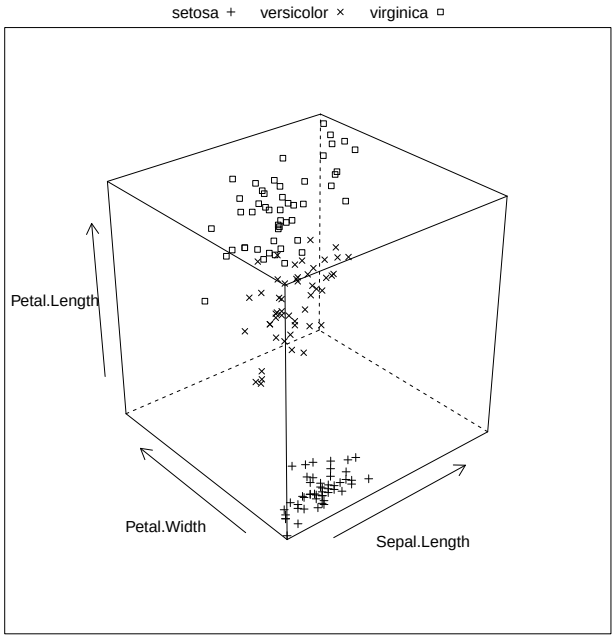

FIGURE 4.6: *This is a 3D scatterplot for the famous Iris data, originally due to ***. I have chosen three variables from the four, and have plotted each species with a different marker. You can see from the plot that the species cluster quite tightly, and are different from one another. R code for this plot is on the website.*

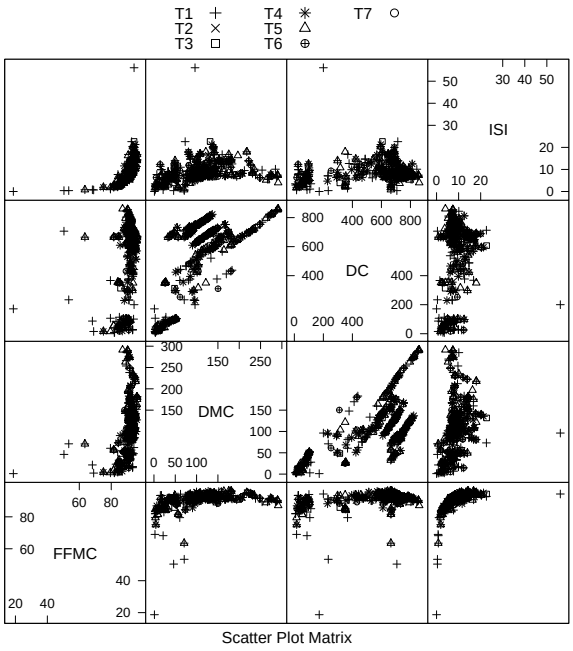

FIGURE 4.7: *This is a scatterplot matrix for the fire dataset from the UC Irvine repository. The smallest area fire is 'T1', and the largest is 'T7'; each is plotted with a different marker. These plots show severity of the fire, plotted against variables 5-8 of the dataset. You should notice that there isn't much separation between the markers. It might be very hard to predict the severity of a fire from these variables. R code for this plot is on the website.*

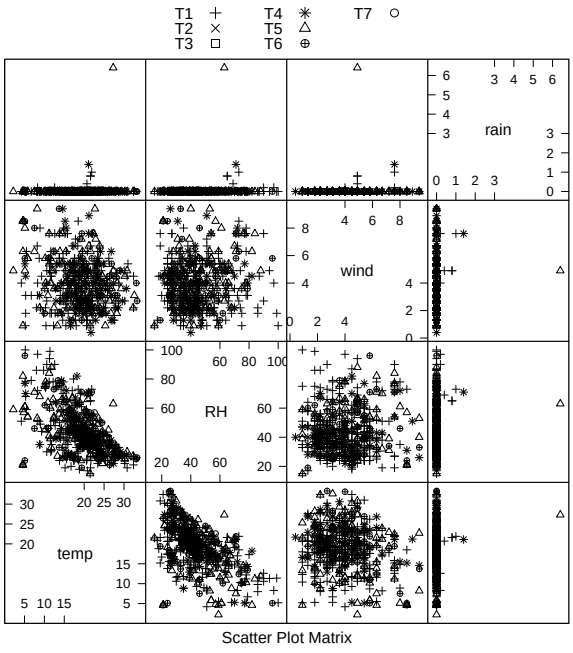

FIGURE 4.8: *This is a scatterplot matrix for the fire dataset from the UC Irvine repository. The smallest area fire is 'T1', and the largest is 'T7'; each is plotted with a different marker. These plots show severity of the fire, plotted against variables 9-12 of the dataset. You should notice that there isn't much separation between the markers. It might be very hard to predict the severity of a fire from these variables. R code for this plot is on the website.*

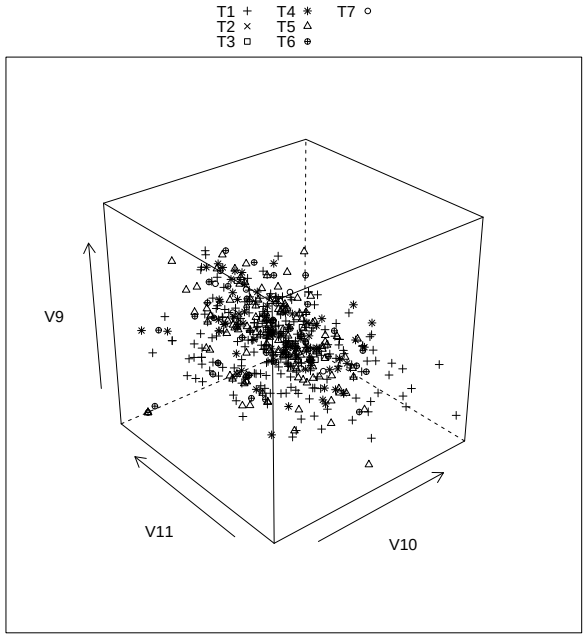

FIGURE 4.9: *This is a 3D scatterplot for the fire dataset from the UC Irvine repository. The smallest area fire is 'T1', and the largest is 'T7'; each is plotted with a different marker. These plots show severity of the fire, plotted against variables 9-11 of the dataset. You should notice that there isn't much separation between the markers. It might be very hard to predict the severity of a fire from these variables. R code for this plot is on the website.*

4.2 SUMMARIES OF HIGH DIMENSIONAL DATA

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

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

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

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

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

4.2.2 Using Covariance to encode Variance and Correlation

Variance, standard deviation and correlation can each be seen as an instance of a more general operation on data. Assume that we have two one dimensional data sets $\{x\}$ and $\{y\}$. Then we can define the **covariance** of $\{x\}$ and $\{y\}$.

Definition: 4.1 *Covariance*

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

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

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

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

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

which you can prove by substituting the expressions. Recall that variance measures the tendency of a dataset to be different from the mean, so the covariance of a dataset with itself is a measure of its tendency not to be constant.

More important, notice that

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

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.

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

Definition: 4.2 *Covariance Matrix*

The covariance matrix is:

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

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

Properties of the Covariance Matrix 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.

- 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:

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

Proof:

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

Proposition:

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

Proof: We have

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

Proposition: *Write* $\Sigma = \mathsf{Covmat}(\{\mathbf{x}\})$. *If there is no vector* $\mathbf{a}$ *such that* $\mathbf{a}^T(\mathbf{x}_i - \mathsf{mean}(\{\mathbf{x}\})) = 0$ *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

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

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

4.3 BLOB ANALYSIS OF HIGH-DIMENSIONAL DATA

When we plotted histograms, we saw that mean and variance were a very helpful description of data that had a unimodal histogram. If the histogram had more than one mode, one needed to be somewhat careful to interpret the mean and variance; in the pizza example, we plotted diameters for different manufacturers to try and see the data as a collection of unimodal histograms.

Generally, mean and covariance are a good description of data that lies in a "blob" (Figure 4.10). You might not believe that this is a technical term, but it's quite widely used. This is because mean and covariance supply a natural coordinate system in which to interpret the blob. Mean and covariance are less useful as descriptions of data that forms multiple blobs (Figure 4.10). In chapter 9.9, we discuss methods to model data that forms multiple blobs, or other shapes that we will interpret as a set of blobs. But many datasets really are single blobs, and we concentrate on such data here. The way to understand a blob is to think about the coordinate transformations that place a blob into a particularly convenient form.

4.3.1 Transforming High Dimensional Data

Assume we apply an affine transformation to our data set $\{\mathbf{x}\}$, to obtain a new dataset $\{\mathbf{u}\}$, where $\mathbf{u}_i = \mathcal{A}\mathbf{x}_i + \mathbf{b}$. Here $\mathcal{A}$ is any matrix (it doesn't have to be

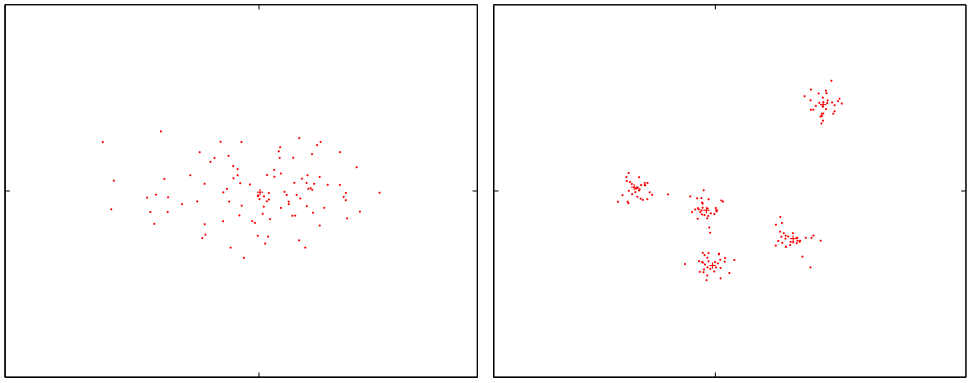

FIGURE 4.10: *On the* **left**, *a "blob" in two dimensions. This is a set of data points that lie somewhat clustered around a single center, given by the mean. I have plotted the mean of these data points with a '+'. On the* **right**, *a data set that is best thought of as a collection of five blobs. I have plotted the mean of each with a '+'. We could compute the mean and covariance of this data, but it would be less revealing than the mean and covariance of a single blob. In chapter 9.9, I will describe automatic methods to describe this dataset as a series of blobs.*

square, or symmetric, or anything else; it just has to have second dimension d). It is easy to compute the mean and covariance of $\{\mathbf{u}\}$. We have

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

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

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

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

4.3.2 Transforming Blobs

The trick to interpreting high dimensional data is to use the mean and covariance to understand the blob. Figure 4.11 shows a two-dimensional data set. Notice that there is obviously some correlation between the x and y coordinates (it's a diagonal 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 center (Figure 4.11). In coordinates, this means we compute the new

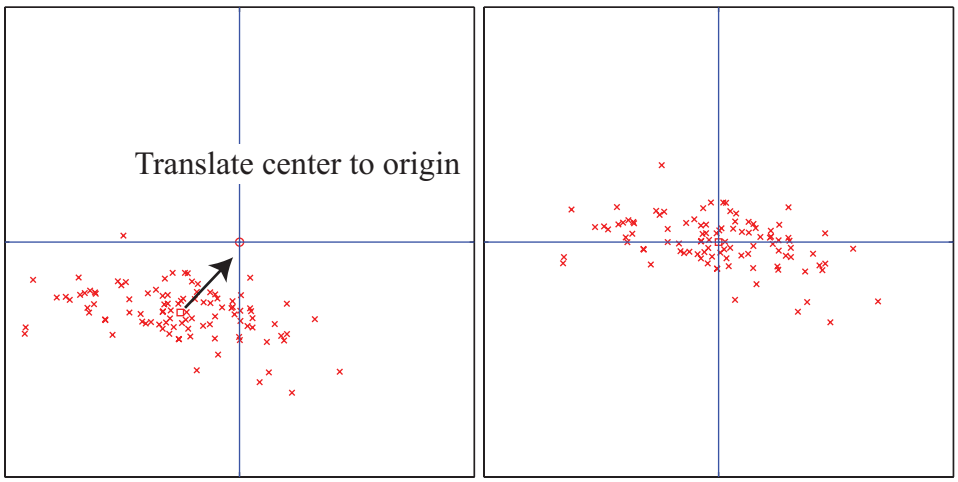

FIGURE 4.11: *On the* **left**, *a "blob" in two dimensions. This is a set of data points that lie somewhat clustered around a single center, given by the mean. I have plotted the mean of these data points with a hollow square (it's easier to see when there is a lot of data). To translate the blob to the origin, we just subtract the mean from each datapoint, yielding the blob on the* **right**.

dataset $\{\mathbf{u}\}$ from the old dataset $\{\mathbf{x}\}$ by the rule $\mathbf{u}_i = \mathbf{x}_i - \text{mean}(\{\mathbf{x}\})$. This new dataset has been translated so that the mean is zero.

Once this blob is translated (Figure 4.12, left), we can rotate it as well. It is natural to try to rotate the blob so that there is no correlation between distinct pairs of dimensions. We can do so by diagonalizing the covariance matrix. In particular, let $\mathcal{U}$ be the matrix formed by stacking the eigenvectors of $\text{Covmat}(\{\mathbf{x}\})$ into a matrix (i.e. $\mathcal{U} = [\mathbf{v}_1, \ldots, \mathbf{v}_d]$, where $\mathbf{v}_j$ are eigenvectors of the covariance matrix). We now form the dataset $\{\mathbf{n}\}$, using the rule

$$\mathbf{n}_i = \mathcal{U}^T \mathbf{u}_i = \mathcal{U}^T (\mathbf{x}_i - \text{mean}(\{\mathbf{x}\})).$$

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

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

where Λ is a diagonal matrix of eigenvalues of $\text{Covmat}(\{\mathbf{x}\})$. 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. We now have two very useful facts about $\{\mathbf{n}\}$: (a) every pair of distinct components has covariance zero, and so has correlation zero; (b) the first component has the highest variance, the second component has the second highest variance, and so on. We can rotate and translate any blob into a coordinate system that has these properties. *In this coordinate system*, we

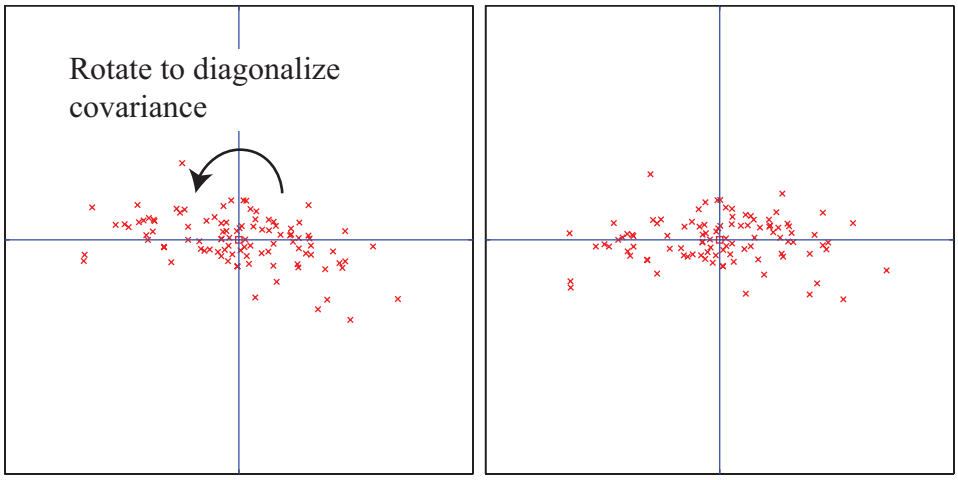

FIGURE 4.12: *On the* **left**, *the translated blob of figure 4.11. This blob lies somewhat diagonally, because the vertical and horizontal components are correlated. On the* **right**, *that blob of data rotated so that there is no correlation between these components. We can now describe the blob 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.*

can describe the blob simply by giving the variances of each component — the covariances are zero.

Translating a blob of data doesn't change the scatterplot matrix in any interesting way (the axes change, but the picture doesn't). Rotating a blob produces really interesting results, however. Figure 4.14 shows the dataset of figure 4.3, translated to the origin and rotated to diagonalize it. Now we do not have names for each component of the data (they're linear combinations of the original components), but each pair is now not correlated. This blob has some interesting shape features. Figure 4.14 shows the gross shape of the blob best. Each panel of this figure has the same scale in each direction. You can see the blob extends about 80 units in direction 1, but only about 15 units in direction 2, and much less in the other two directions. You should think of this blob as being rather cigar-shaped; it's long in one direction, but there isn't much in the others. The cigar metaphor isn't perfect because there aren't any 4 dimensional cigars, but it's helpful. You can think of each panel of this figure as showing views down each of the four axes of the cigar.

Now look at figure **??**. 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

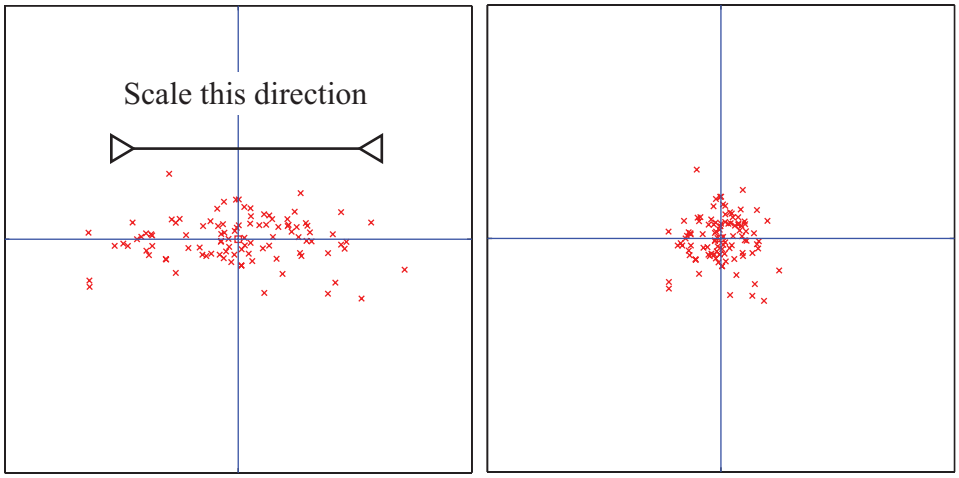

FIGURE 4.13: *On the* **left**, *the translated and rotated blob of figure 4.12. This blob is stretched — one direction has more variance than another. Because all covariances are zero, it is easy to scale the blob so that all variances are one (the blob on the* **right**). *You can think of this as a standard blob. All blobs can be reduced to a standard blob, by relatively straightforward linear algebra.*

with a number. These points are clearly special in some way.

We could now scale the data in this new coordinate system so that all the variances are either one (if there is any variation in that direction) or zero (directions where the data doesn't vary — these occur only if some directions are functions of others). Figure 4.13 shows the final scaling. The result is a standard blob. Our approach applies to any dimension — I gave 2D figures because they're much easier to understand. There is a crucial point here: we can reduce any blob of data, in any dimension, to a standard blob of that dimension. All blobs are the same, except for some stretching, some rotation, and some translation. This is why blobs are so well-liked.

4.3.3 Whitening Data

It is sometimes useful to actually reduce a dataset to a standard blob. 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. For example, if we have a dataset where one component ranges from 1e5 to 2e5, and the other component ranges from -1e-5 to 1e-5, we are likely to face numerical problems in many computations (adding small numbers to big numbers is often unwise). Often, this kind of thing follows from a poor choice of units, rather than any kind of useful property of the data. In such a case, it could be quite helpful to whiten the data. Another reason to whiten the data might be that we know relatively little about the meaning of each component. In this case, the original choice of coordinate system was somewhat arbitrary anyhow, and

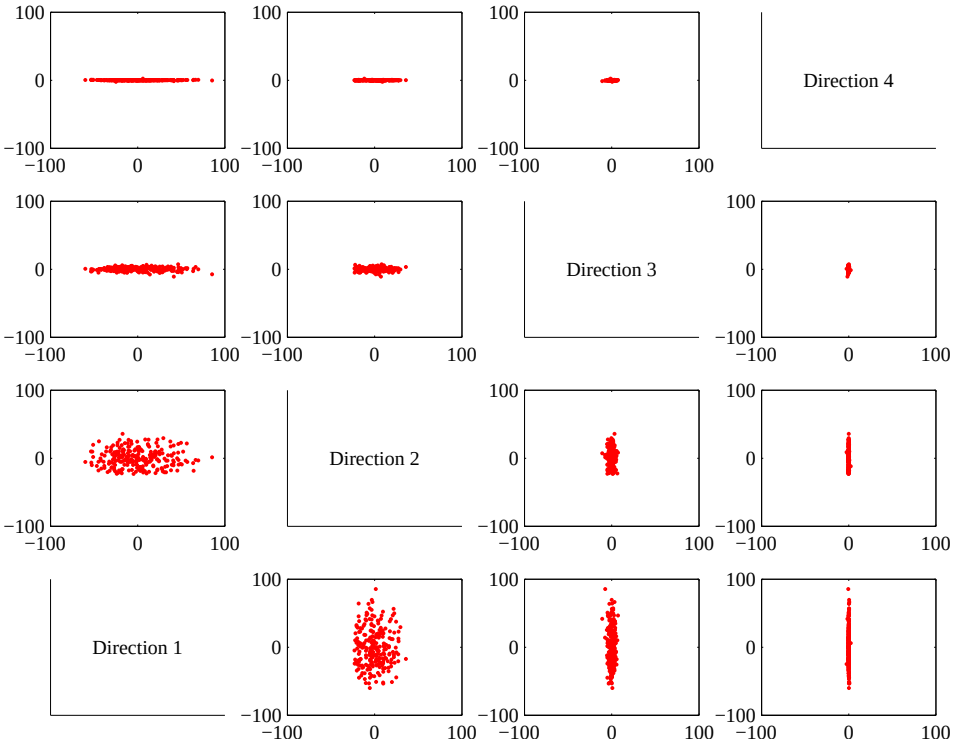

FIGURE 4.14: *A panel plot of the bodyfat dataset of figure 4.3, now rotated so that the covariance between all pairs of distinct dimensions is zero. Now we do not know names for the directions — they're linear combinations of the original variables. Each scatterplot is on the same set of axes, so you can see that the dataset extends more in some directions than in others.*

transforming data to a uniform blob could be helpful.

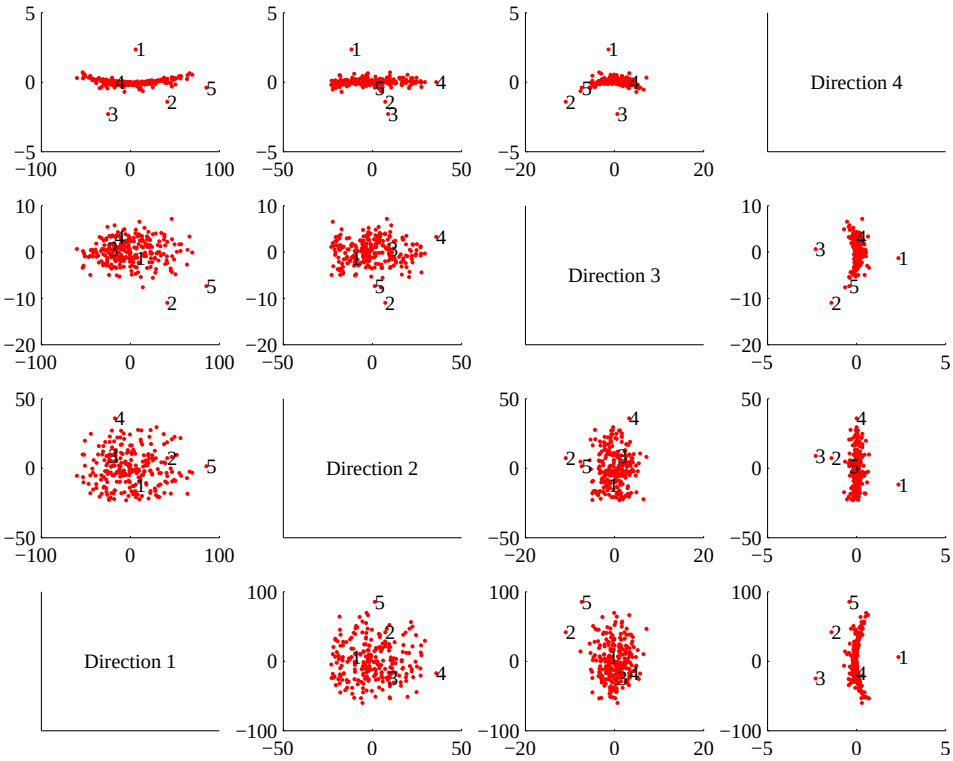

FIGURE 4.15: *A panel plot of the bodyfat dataset of figure 4.3, now rotated so that the covariance between all pairs of distinct dimensions is zero. Now we do not know names for the directions — they're linear combinations of the original variables. I have scaled the axes so you can see details; notice that the blob is a little curved, and there are several data points that seem to lie some way away from the blob, which I have numbered.*

Useful Facts: 4.1 *Whitening a dataset*

For a dataset $\{\mathbf{x}\}$, compute:

- $\mathcal{U}$, the matrix of eigenvectors of $\mathsf{Covmat}(\{\mathbf{x}\})$;
- and $\mathsf{mean}(\{\mathbf{x}\})$.

Now compute $\{\mathbf{n}\}$ using the rule

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

Then $\mathsf{mean}(\{\mathbf{n}\}) = \mathbf{0}$ and $\mathsf{Covmat}(\{\mathbf{n}\})$ is diagonal.
Now write Λ for the diagonal matrix of eigenvalues of $\mathsf{Covmat}(\{\mathbf{x}\})$ (so that $\mathsf{Covmat}(\{\mathbf{x}\})\mathcal{U} = \mathcal{U}\Lambda$). Assume that each of the diagonal entries of Λ is greater than zero (otherwise there is a redundant dimension in the data). Write λ_i for the i'th diagonal entry of Λ, and write $\Lambda^{-(1/2)}$ for the diagonal matrix whose i'th diagonal entry is $1/\sqrt{\lambda_i}$. Compute $\{\mathbf{z}\}$ using the rule

$$\mathbf{z}_i = \Lambda^{(-1/2)}\mathcal{U}(\mathbf{x}_i - \mathsf{mean}(\{\mathbf{x}\})).$$

We have that $\mathsf{mean}(\{\mathbf{z}\}) = 0$ and $\mathsf{Covmat}(\{\mathbf{z}\}) = \mathcal{I}$. The dataset $\{\mathbf{z}\}$ is often known as **whitened data**.

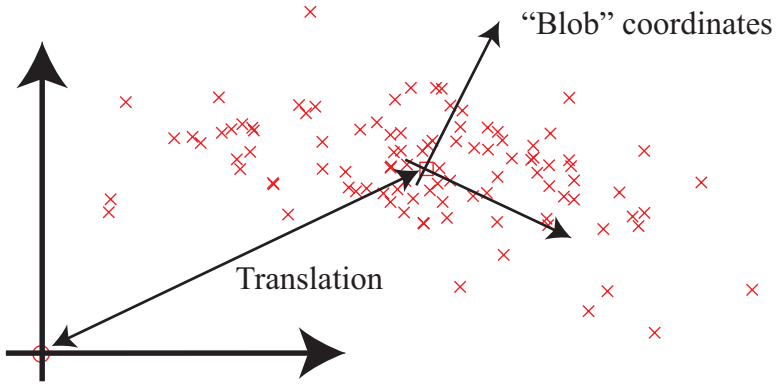

FIGURE 4.16: *A 2D blob, with its natural blob coordinate system. The origin of this coordinate system is at the mean of the data. The coordinate axes are (a) at right angles to one another and (b) are directions that have no covariance.*

It isn't always a good idea to whiten data. In some circumstances, each separate component is meaningful, and in a meaningful set of units. For example, one of the components might be a length using a natural scale and the other might be a time on a natural scale. When this happens, we might be reluctant to transform the data, either because we don't want to add lengths to times or because we want to preserve the scales.

4.4 PRINCIPAL COMPONENTS ANALYSIS

Mostly, when one deals with high dimensional data, it isn't clear which individual components are important. As we have seen with the height weight dataset (for example, in the case of density and weight) some components can be quite strongly correlated. Equivalently, as in Figure 9.9, the blob is not aligned with the coordinate axes.

4.4.1 The Blob Coordinate System and Smoothing

We can use the fact that we *could* rotate, translate and scale the blob to define a coordinate system within the blob. The origin of that coordinate system is the mean of the data, and the coordinate axes are given by the eigenvectors of the covariance matrix. These are orthonormal, so they form a set of unit basis vectors at right angles to one another (i.e. a coordinate system). You should think of these as blob coordinates; Figure 4.16 illustrates a set of blob coordinates.

The blob coordinate system is important because, *once we know the blob coordinate system*, we can identify important scales of variation in the data. For example, if you look at Figure 4.16, you can see that this blob is extended much further along one direction than along the other. We can use this information to identify the most significant forms of variation in very high dimensional data. In

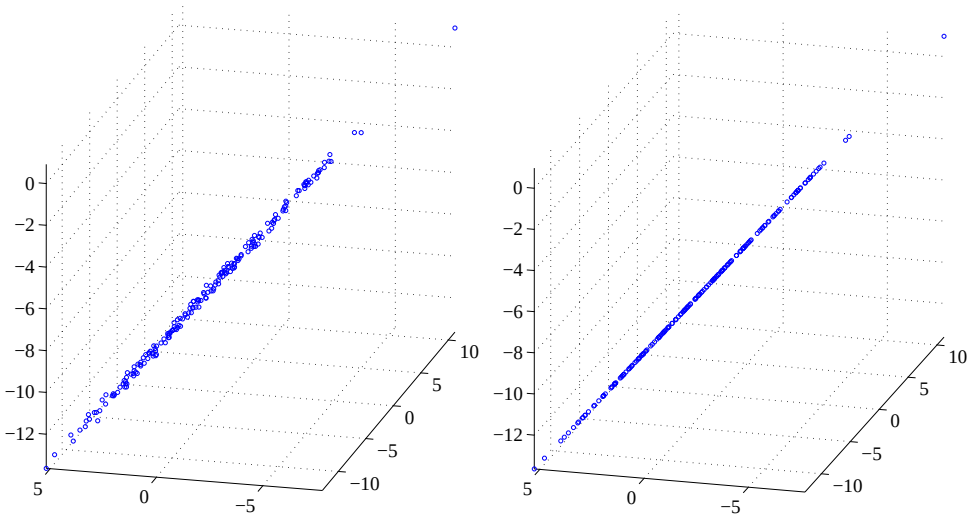

FIGURE 4.17: *On the* **left**, *a blob of 3D data that has very low variance in two directions in the blob coordinates. As a result, all the data points are very close to a 1D blob. Experience shows that this is a common phenomenon. Although there might be many components in the data items, all data points are very close to a much lower dimensional object in the high dimensional space. When this is the case, we could obtain a lower dimensional representation of the data by working in blob coordinates, or we could smooth the data (as on the* **right**), *by projecting each data point onto the lower dimensional space.*

some directions in the blob coordinate system, the blob will be spread out — ie have large variance — but in others, it might not be.

Equivalently, imagine we choose to represent each data item *in blob coordinates*. Then the mean over the dataset will be zero. Each pair of distinct coordinates will be uncorrelated. Some coordinates — corresponding to directions where the blob is spread out — will have a large range of values. Other coordinates — directions in which the blob is small — will have a small range of values. We could choose to replace these coordinates with zeros, with little significant loss in accuracy. The advantage of doing so is that we would have lower dimensional data to deal with.

However, it isn't particularly natural to work in blob coordinates. Each component of a data item may have a distinct meaning and scale (i.e. feet, pounds, and so on), but this is not preserved in any easy way in blob coordinates. Instead, we should like to (a) compute a lower dimensional representation in blob coordinates then (b) transform that representation into the original coordinate system of the data item. Doing so is a form of **smoothing** — suppressing small, irrelevant variations by exploiting multiple data items.

For example, look at Figure 4.17. Imagine we transform the blob on the left to blob coordinates. The covariance matrix in these coordinates is a 3×3 diagonal matrix. One of the values on the diagonal is large, because the blob is extended on

one direction; but the other two are small. This means that, in blob coordinates, the data varies significantly in one direction, but very little in the other two directions.

Now imagine we project the data points onto the high-variation direction; equivalently, we set the other two directions to zero for each data point. Each of the new data points is very close to the corresponding old data point, because by setting the small directions to zero we haven't moved the point very much. In blob coordinates, the covariance matrix of this new dataset has changed very little. It is again a 3×3 diagonal matrix, but now two of the diagonal values are zero, because there isn't any variance in those directions. The third value is large, because the blob is extended in that direction. We take the new dataset, and rotate and translate it into the original coordinate system. Each point must lie close to the corresponding point in the original dataset. However, the new dataset lies along a straight line (because it lay on a straight line in the blob coordinates). This process gets us the blob on the right in Figure 4.17. This blob is a smoothed version of the original blob.

Smoothing works because when two data items are strongly correlated, the value of one is a good guide to the value of the other. This principle works for more than two data items. Section 9.9 describes an example where the data items have dimension 101, but all values are extremely tightly correlated. In a case like this, there may be very few dimensions in blob coordinates that have any significant variation (3-6 for this case, depending on some details of what one believes is a small number, and so on). The components are so strongly correlated in this case that the 101-dimensional blob really looks like a slightly thickened 3 (or slightly more) dimensional blob that has been inserted into a 101-dimensional space (Figure 4.17). If we project the 101-dimensional data onto that structure in the original, 101-dimensional space, we may get much better estimates of the components of each data item than the original measuring device can supply. This occurs because each component is now estimated using correlations between all the measurements.

4.4.2 The Low-Dimensional Representation of a Blob

We wish to construct an r dimensional representation of a blob, where we have chosen r in advance. First, we compute $\{\mathbf{v}\}$ by translating the blob so its mean is at the origin, so that $\mathbf{v}_i = \mathbf{x}_i - \mathsf{mean}\,(\{\mathbf{x}\})$. Now write $\mathcal{V} = [\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_N]$. The covariance matrix of $\{\mathbf{v}\}$ is then

$$\mathsf{Covmat}\,(\{\mathbf{v}\}) = \frac{1}{N}\mathcal{V}\mathcal{V}^T = \mathsf{Covmat}\,(\{\mathbf{x}\}).$$

Now write Λ for the diagonal matrix of eigenvalues of $\mathsf{Covmat}\,(\{\mathbf{x}\})$ and $\mathcal{U}$ for the matrix of eigenvectors, so that $\mathsf{Covmat}\,(\{\mathbf{x}\})\mathcal{U} = \mathcal{U}\Lambda$. We assume that the elements of Λ are sorted in decreasing order along the diagonal. The covariance matrix for the dataset transformed into blob coordinates will be Λ. Notice that

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

This means we can interpret $(\mathcal{U}^T\mathcal{V})$ as a new dataset $\{\mathbf{b}\}$. This is our data, rotated into blob coordinates.

Now write Π_r for the $d \times d$ matrix

$$\begin{bmatrix} \mathcal{I}_r & 0 \\ 0 & 0 \end{bmatrix}$$

which projects a d dimensional vector onto its first r components, and replaces the others with zeros. Then we have that

$$\Lambda_r = \Pi_r \Lambda \Pi_r^T$$

is the covariance matrix for the reduced dimensional data in blob coordinates. Notice that Λ_r keeps the r largest eigenvalues on the diagonal of Λ, and replaces all others with zero.

We have

$$\begin{aligned} \Lambda_r &= \Pi_r \Lambda \Pi_r^T \\ &= \Pi_r \mathcal{U}^T \mathsf{Covmat}\left(\{\mathbf{x}\}\right) \mathcal{U} \Pi_r^T \\ &= (\Pi_r \mathcal{U}^T \mathcal{V})(\mathcal{V}^T \mathcal{U} \Pi_r^T) \\ &= \mathcal{P}\mathcal{P}^T \end{aligned}$$

where $\mathcal{P} = (\Pi_r \mathcal{U}^T \mathcal{V})$. This represents our data, rotated into blob coordinates, and then projected down to r dimensions, with remaining terms replaced by zeros. Write $\{\mathbf{b}_r\}$ for this new dataset.

Occasionally, we need to refer to this representation, and we give it a special name. Write

$$\mathsf{pcaproj}\left(\mathbf{x}_i, r, \{\mathbf{x}\}\right) = \Pi_r \mathcal{U}^T (\mathbf{x}_i - \mathsf{mean}\left(\{\mathbf{x}\}\right))$$

where the notation seeks to explicitly keep track of the fact that the low dimensional representation of a particular data item *depends on the whole dataset* (because you have to be able to compute the mean, and the eigenvectors of the covariance). Notice that $\mathsf{pcaproj}\left(\mathbf{x}_i, r, \{\mathbf{x}\}\right)$ is a representation of the dataset with important properties:

- The representation is r-dimensional (i.e. the last $d - r$ components are zero).
- Each pair of distinct components of $\{\mathsf{pcaproj}\left(\mathbf{x}_i, r, \{\mathbf{x}\}\right)\}$ has zero covariance.
- The first component of $\{\mathsf{pcaproj}\left(\mathbf{x}_i, r, \{\mathbf{x}\}\right)\}$ has largest variance; the second component has second largest variance; and so on.

4.4.3 Smoothing Data with a Low-Dimensional Representation

We would now like to construct a low dimensional representation of the blob, in the original coordinates. We do so by rotating the low-dimensional representation back to the original coordinate system, then adding back the mean to translate the origin back to where it started. We can write this as

$$\begin{aligned} \mathsf{pcasmooth}\left(\mathbf{x}_i, r, \{\mathbf{x}\}\right) &= \mathcal{U}\Pi_r^T \left(\Pi_r \mathcal{U}^T (\mathbf{x}_i - \mathsf{mean}\left(\{\mathbf{x}\}\right))\right) + \mathsf{mean}\left(\{\mathbf{x}\}\right) \\ &= \mathcal{U}\Pi_r^T \, \mathsf{pcaproj}\left(\mathbf{x}_i, r, \{\mathbf{x}\}\right) + \mathsf{mean}\left(\{\mathbf{x}\}\right) \end{aligned}$$

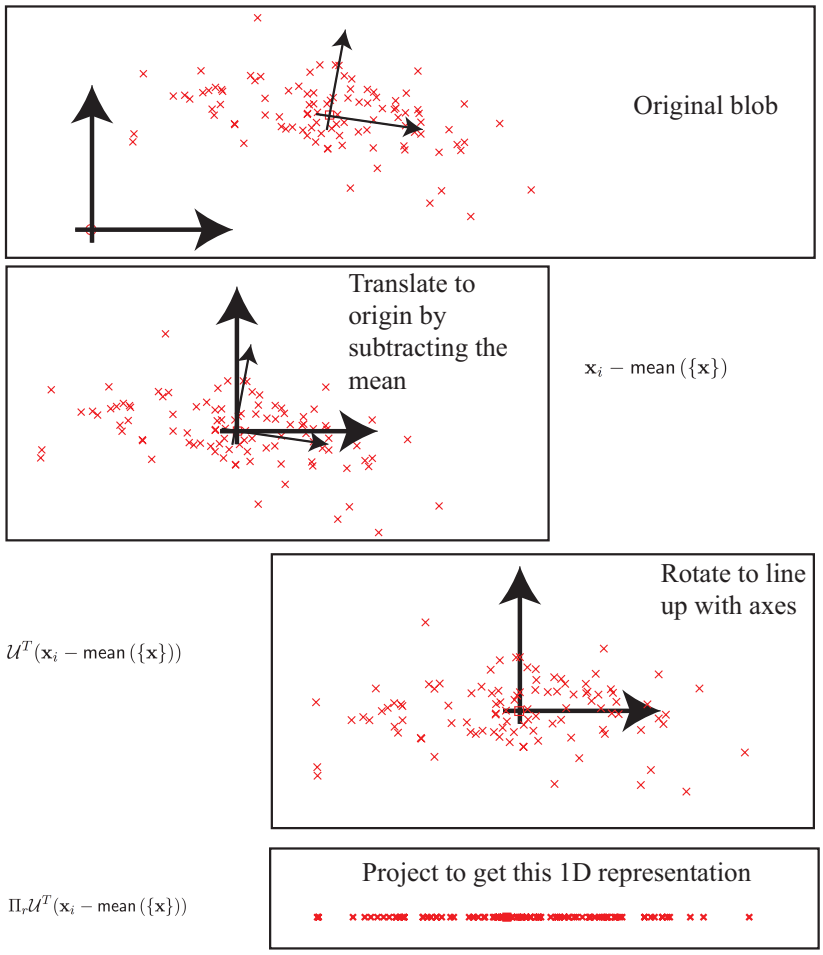

FIGURE 4.18: *Computing a low dimensional representation for principal components analysis.*

we have a new representation of the i'th data item *in the original space* (Figure 9.9). Now consider the dataset obtained by smoothing each of our data items. We write this dataset as $\{\mathsf{pcasmooth}\,(\mathbf{x}_i, r, \{\mathbf{x}\})\}$.

You should think of $\{\mathsf{pcasmooth}\,(\mathbf{x}_i, r, \{\mathbf{x}\})\}$ as a smoothed version of the original data. One way to think of this process is that we have chosen a low-dimensional basis that represents the main variance in the data rather well. It is quite usual to think of a data item as being given by a the mean plus a weighted sum of these basis elements. In this view, the first weight has larger variance than the second, and so on. By construction, this dataset lies in an r dimensional affine subspace of the original space. We constructed this r-dimensional space to preserve the largest variance directions of the data. Each column of this matrix is known as a **principal component**. In particular, we have constructed this dataset so that

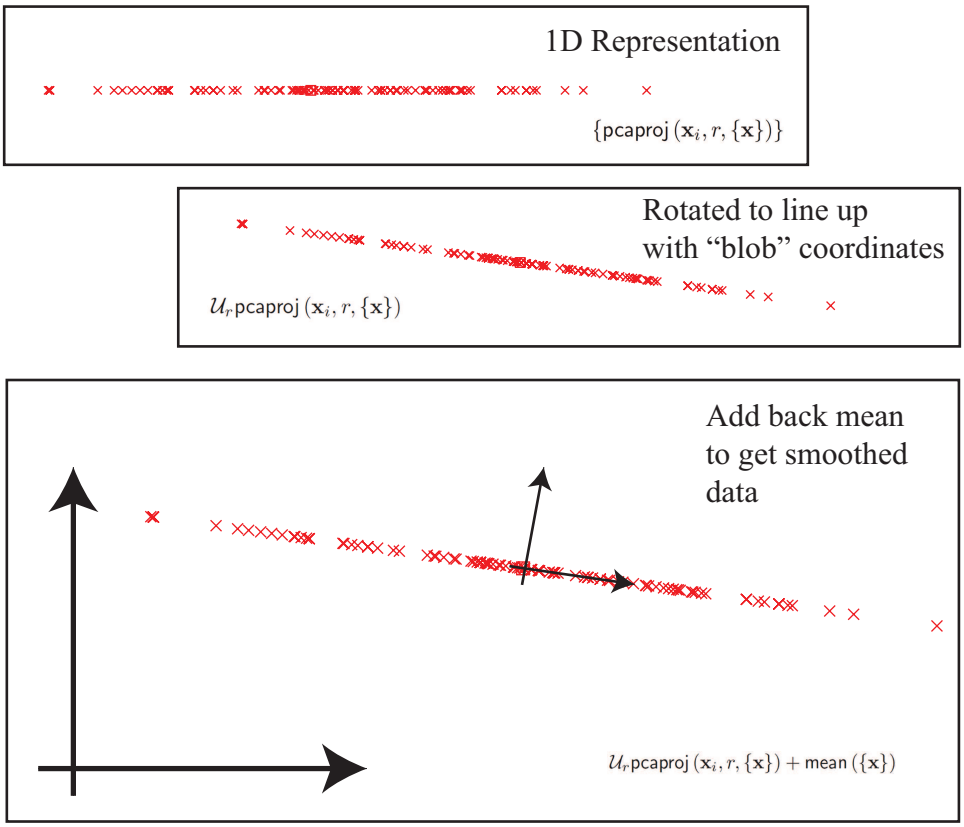

FIGURE 4.19: *Smoothing data with principal components analysis.*

- $\mathsf{mean}\left(\{\mathsf{pcasmooth}\left(\mathbf{x}_i, r, \{\mathbf{x}\}\right)\}\right) = \mathsf{mean}\left(\{\mathbf{x}\}\right)$;
- $\mathsf{Covmat}\left(\{\mathsf{pcasmooth}\left(\mathbf{x}_i, r, \{\mathbf{x}\}\right)\}\right)$ has rank r;
- $\mathsf{Covmat}\left(\{\mathsf{pcasmooth}\left(\mathbf{x}_i, r, \{\mathbf{x}\}\right)\}\right)$ is the best approximation of $\mathsf{Covmat}\left(\{\mathbf{x}\}\right)$ with rank r.

Figure 4.19 gives a visualization of the smoothing process. By comparing figures 4.14 and 4.20, you can see that a real dataset can lose several dimensions without much significant going wrong. As we shall see in the examples, some datasets can lose many dimensions without anything bad happening.

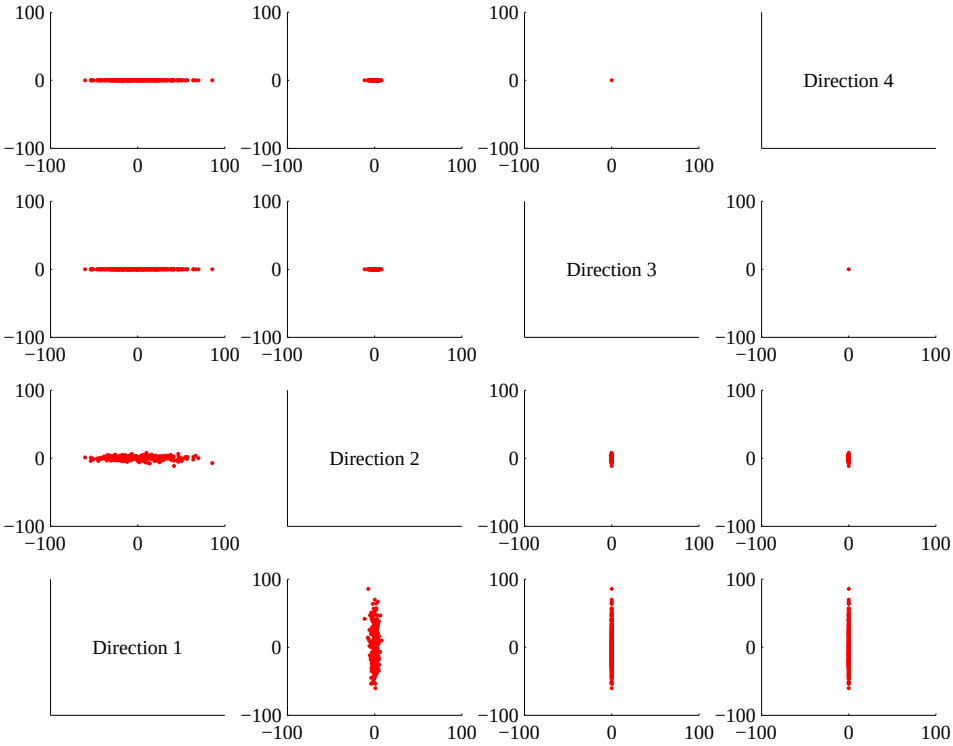

FIGURE 4.20: *A panel plot of the bodyfat dataset of figure 4.3, with the dimension reduced to two using principal components analysis. Compare this figure to figure 4.14, which is on the same set of axes. You can see that the blob has been squashed in direction 3 and direction 4. But not much has really happened, because there wasn't very much variation in those directions in the first place.*

Procedure: 4.1 *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 **??**). It is quite usual to do this by hand.
Constructing a low-dimensional representation: Form $\mathcal{U}_r$, a matrix consisting of the first r columns of $\mathcal{U}$. Now compute $\{\mathsf{pcaproj}(\mathbf{x}_i, r, \{\mathbf{x}\})\} = \{(\Pi_r \mathcal{U}^T(\mathbf{x}_i - \mathsf{mean}(\{\mathbf{x}\})))\}$. This is a set of data vectors which are r dimensional, and where each component is independent of each other component (i.e. the covariances of distinct components are zero).
Smoothing the data: Form $\{\mathsf{pcasmooth}(\mathbf{x}_i, r, \{\mathbf{x}\})\} = \{(\mathcal{U}_r \mathsf{pcaproj}(\mathbf{x}_i, r, \{\mathbf{x}\}) + \mathsf{mean}(\{\mathbf{x}\}))\}$. These are d dimensional vectors that lie in a r-dimensional subspace of d-dimensional space. The "missing dimensions" have the lowest variance, and are independent.

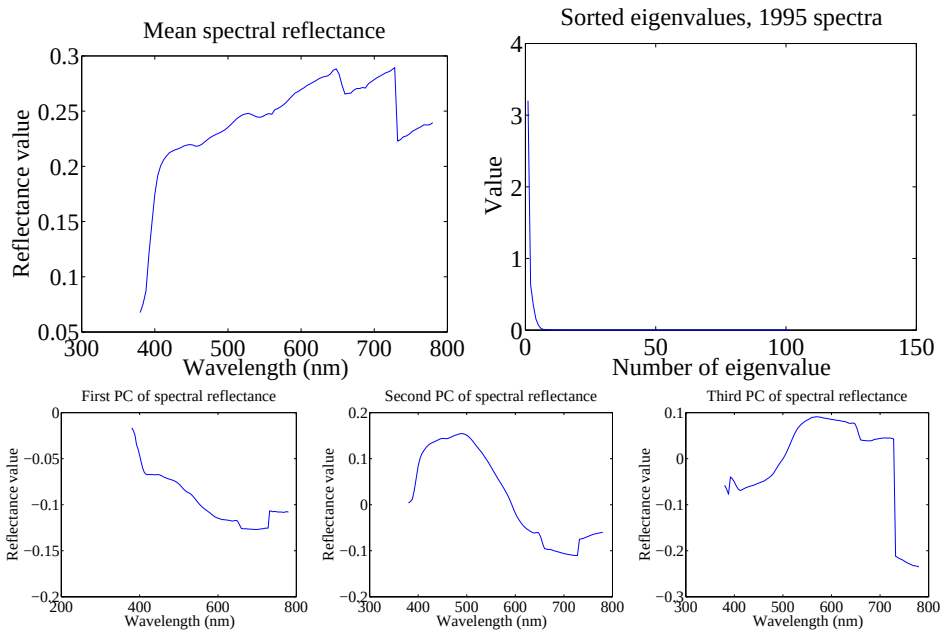

FIGURE 4.21: *On the* **top left**, *the mean spectral reflectance of a dataset of 1995 spectral reflectances, collected by Kobus Barnard (at* http://www.cs.sfu.ca/~colour/data/*). On the* **top right**, *eigenvalues of the covariance matrix of spectral reflectance data, from a dataset of 1995 spectral reflectances, collected by Kobus Barnard (at* http://www.cs.sfu.ca/~colour/data/*). Notice how the first few eigenvalues are large, but most are very small; this suggests that a good representation using few principal components is available. The* **bottom row** *shows the first three principal components. A linear combination of these, with appropriate weights, added to the mean of figure* **??**, *gives a good representation of the dataset.*

4.4.4 The Error of the Low-Dimensional Representation

We took a dataset, $\{\mathbf{x}\}$, and constructed a d-dimensional dataset $\{\mathbf{b}\}$ in blob coordinates. We did so by translating, then rotating, the data, so no information was lost; we could reconstruct our original dataset by rotating, then translating $\{\mathbf{b}\}$. But in blob coordinates we projected each data item down to the first r components to get an r-dimensional dataset $\{\mathbf{b}_r\}$. We then reconstructed a smoothed dataset by rotating, then translating, $\{\mathbf{b}_r\}$. Information has been lost here, but how much?

The answer is easy to get if you recall that rotations and translations do not change lengths. This means that

$$\| \mathbf{x}_i - \text{pcasmooth}\left(\mathbf{x}_i, r, \{\mathbf{x}\}\right) \|^2 = \| \mathbf{b}_i - \mathbf{b}_{r,i} \|^2.$$

This expression is easy to evaluate, because $\mathbf{b}_i$ and $\mathbf{b}_{r,i}$ agree in their first r com-

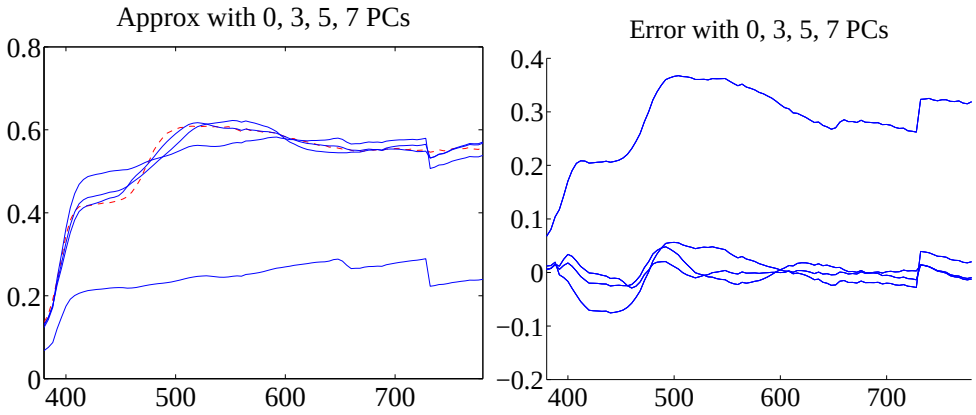

FIGURE 4.22: *On the* **left**, *a spectral reflectance curve (dashed) and approximations using the mean, the mean and 3 principal components, the mean and 5 principal components, and the mean and 7 principal components. Notice the mean is a relatively poor approximation, but as the number of principal components goes up, the error falls rather quickly. On the* **right** *is the error for these approximations. Figure plotted from a dataset of 1995 spectral reflectances, collected by Kobus Barnard (at http://www.cs.sfu.ca/~colour/data/).*

ponents. The remaining $d - r$ components of $\mathbf{b}_{r,i}$ are zero. So we can write

$$\| \mathbf{x}_i - \text{pcasmooth}\left(\mathbf{x}_i, r, \{\mathbf{x}\}\right) \|^2 = \sum_{u=r+1}^{d} (\mathbf{b}_i^{(u)})^2.$$

Now a natural measure of error is the average over the dataset of this term. We have that

$$\frac{1}{N} \sum_{u=r+1}^{d} (\mathbf{b}_i^{(u)})^2 = \sum_{u=r+1}^{d} \text{var}\left(\left\{\mathbf{b}^{(u)}\right\}\right)$$

which is easy to evaluate, because we know these variances — they are the values of the $d-r$ eigenvalues that we decided to ignore. So the mean error can be written as

$$\mathbf{1}^T(\Lambda - \Lambda_r)\mathbf{1}.$$

Now we could choose r 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 9.9. You can see that the sum of remaining eigenvalues is small.

4.4.5 Example: Representing Spectral Reflectances

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

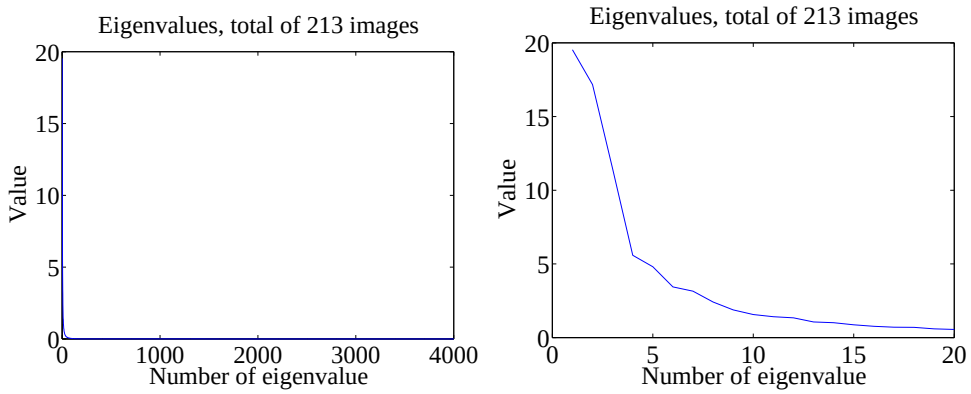

FIGURE 4.23: *On the* **left**,*the eigenvalues of the covariance of the Japanese facial expression dataset; there are 4096, so it's hard to see the curve (which is packed to the left). On the* **right**, *a zoomed version of the curve, showing how quickly the values of the eigenvalues get small.*

be colored, because the surface reflects different fractions of the light falling on it at different wavelengths. This effect can be represented by measuring the spectral reflectance of a surface, which is the fraction of light the surface reflects as a function of wavelength. This is usually measured in the visual range of wavelengths (about 380nm to about 770 nm). Typical measurements are every few nm, depending on the measurement device. I obtained data for 1995 different surfaces from http://www.cs.sfu.ca/~colour/data/ (there are a variety of great datasets here, from Kobus Barnard).

Each spectrum has 101 measurements, which are spaced 4nm apart. This represents surface properties to far greater precision than is really useful. Physical properties of surfaces suggest that the reflectance can't change too fast from wavelength to wavelength. It turns out that very few principal components are sufficient to describe almost any spectral reflectance function. Figure 4.21 shows the mean spectral reflectance of this dataset, and Figure 4.21 shows the eigenvalues of the covariance matrix.

This is tremendously useful in practice. One should think of a spectral reflectance as a function, usually written $\rho(\lambda)$. What the principal components analysis tells us is that we can represent this function rather accurately on a (really small) finite dimensional basis. This basis is shown in figure 4.21. This means that there is a mean function $r(\lambda)$ and k functions $\phi_m(\lambda)$ such that, for any $\rho(\lambda)$,

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

where $e(\lambda)$ is the error of the representation, which we know is small (because it consists of all the other principal components, which have tiny variance). In the case of spectral reflectances, using a value of k around 3-5 works fine for most applications (Figure 4.22). This is useful, because when we want to predict what

FIGURE 4.24: *The mean and first 16 principal components of the Japanese facial expression dataset.*

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.

4.4.6 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

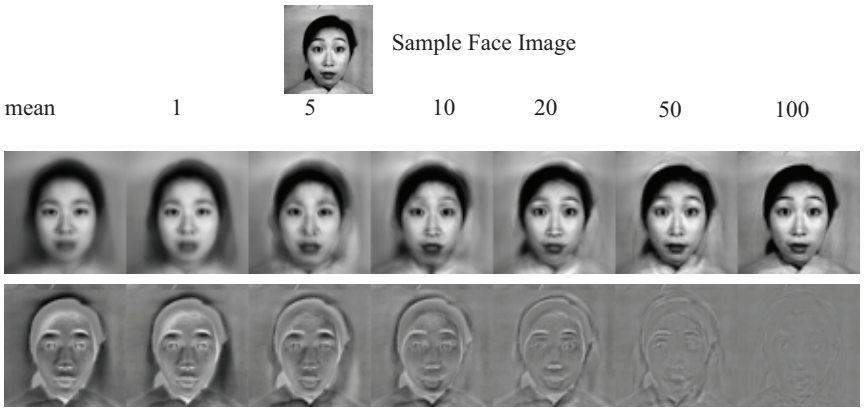

FIGURE 4.25: *Approximating a face image by the mean and some principal components; notice how good the approximation becomes with relatively few components.*

(use `reshape` in Matlab). This means you can take the principal components of a set of images. Doing so was something of a fashionable pastime in computer vision for a while, though there are some reasons that this is not a great representation of pictures. However, the representation yields pictures that can give great intuition into a dataset.

Figure **??** shows the mean of a set of face images encoding facial expressions of Japanese women (available at http://www.kasrl.org/jaffe.html; there are tons of face datasets at http://www.face-rec.org/databases/). I reduced the images to 64x64, which gives a 4096 dimensional vector. The eigenvalues of the covariance of this dataset are shown in figure 4.23; there are 4096 of them, so it's hard to see a trend, but the zoomed figure suggests that the first couple of hundred contain most of the variance. Once we have constructed the principal components, they can be rearranged into images; these images are shown in figure 4.24. Principal components give quite good approximations to real images (figure 4.25).

The principal components sketch out the main kinds of variation in facial expression. Notice how the mean face in Figure 4.24 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.

4.5 HIGH DIMENSIONS, SVD AND 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.

4.5.1 Principal Components by SVD

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

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

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

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

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

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

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

4.5.2 Just a few Principal Components with NIPALS

For really big datasets, even taking the SVD is hard. Usually, we don't really want to recover all the principal components, because we want to recover a reasonably accurate low dimensional representation of the data. We continue to work with a data matrix $\mathcal{X}$, whose *rows* are data items. Now assume we wish to recover the first principal component. This means we are seeking a vector $\mathbf{u}$ and a set of N numbers w_i such that $w_i\mathbf{u}$ is a good approximation to $\mathbf{x}_i$. In particular, we would like the dataset made of $w_i\mathbf{u}$ to encode as much of the variance of the original dataset as possible. Now we can stack the w_i into a column vector $\mathbf{w}$. The **Frobenius norm** is a term for the matrix norm obtained by summing squared entries of the matrix. We write

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

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

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

which we can write as

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

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

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

which can be written in matrix vector form as

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

Similarly, the partial with respect to u_l is

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

which can be written in matrix vector form as

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

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

$$\mathbf{w}^{(n+\frac{1}{2})} = \frac{\mathcal{X}\mathbf{u}^{(n)}}{(\mathbf{u}^{(n)})^T\mathbf{u}^{(n)}}.$$

Now we can update the estimate of $\mathbf{u}$ by choosing a value that makes the partial wrt $\mathbf{u}$ zero, using our estimate $\mathbf{w}^{(n+\frac{1}{2})}$, to get

$$\mathbf{u}^{(n+\frac{1}{2})} = \frac{\mathcal{X}^T\mathbf{w}^{(n+\frac{1}{2})}}{(\mathbf{w}^{(n+\frac{1}{2})})^T\mathbf{w}}.$$

We need to rescale to ensure that our estimate of $\mathbf{u}$ has unit length. Write $s = ((\mathbf{w}^{(n+\frac{1}{2})})^T\mathbf{w})^{\frac{1}{2}}$ We get

$$\mathbf{u}^{(n+1)} = \frac{\mathbf{u}^{(n+\frac{1}{2})}}{s}$$

and

$$\mathbf{w}^{(n+1)} = s\mathbf{w}^{(n+\frac{1}{2})}.$$

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).

NIPALS is quite forgiving of missing values, though missing values make it hard to use matrix notation. Recall I wrote the cost function as $C(\mathbf{w}, \mathbf{u}) = \sum_{ij}(x_{ij} - w_i u_j)^2$. We change the sum so that it ranges over only the known values, to get

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

then write

$$\frac{\partial C}{\partial w_k} = \sum_{j \in \text{known values for } k} (x_{kj} - w_k u_j) u_j$$

and

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

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

$$w_k^{(n+\frac{1}{2})} = \frac{\sum_{j \in \text{known values for } k} x_{kj} u_j}{\sum_{j \in \text{known values for } k} u_j^{(n+\frac{1}{2})} u_j^{(n+\frac{1}{2})}}$$

and

$$u_k^{(n+\frac{1}{2})} = \frac{\sum_{i \in \text{known values for } l} x_{il} w_l}{\sum_{i \in \text{known values for } l} w_l^{(n+\frac{1}{2})} w_l^{(n+\frac{1}{2})}}.$$

We then normalize as before.

Procedure: 4.2 *Obtaining some principal components with NIPALS*

4.5.3 Projection and Discriminative Problems

You should remind yourself how Lagrange multipliers work before reading this section.

Principal components analysis identifies the directions in high dimensional space that best describe the dataset. This may not always be what we are looking for. For a simple example, look at figure **??**. In this case, we have two classes of data item. Each varies a lot in the x direction and little in the y direction. If we project onto the first principal component, the outcome represents variance in the data well, but suppresses the difference between the two classes. For many applications – for example, classification – we are interested in directions that emphasize the

differences between classes. One very important construction for such directions can be used for classification and for regression.

Assume we have a dataset of N items, each with two parts $\mathbf{x}_i$ and $\mathbf{y}_i$. We assume that at least $\mathbf{x}_i$ has high dimension. We also assume that $\mathsf{mean}\,(\{\mathbf{x}\}) = 0$ and $\mathsf{mean}\,(\{\mathbf{y}\}) = 0$. This simplifies notation, and is easy to achieve (subtract the mean). This is a mild generalization of what occurred in classification, where we had a feature vector $\mathbf{x}_i$ and a label y_i for each data item; but now instead of having just a label, we have a vector. This situation arises in practice quite often. For example, $\mathbf{x}_i$ might be a vector that describes an image and $\mathbf{y}_i$ might be a vector that describes a caption for that image. If we have a classification problem with C classes, we might choose $\mathbf{y}_i$ to be a vector with zero in all components except the one corresponding to the class (this is sometimes known as a **one-hot vector**). We wish to choose projections of $\mathbf{x}$ and $\mathbf{y}$ to a shared low dimensional space, so that the projections in this space are strongly correlated to one another.

For the moment, assume the low dimensional space is one dimensional. Then there is some unit vector $\mathbf{a}$ so that projecting $\mathbf{x}_i$ to that space is given by $\mathbf{a}^T\mathbf{x}_i$; similarly, there is some unit vector $\mathbf{b}$ so that projecting $\mathbf{y}_i$ to the space is given by $\mathbf{b}^T\mathbf{y}_i$. Now we stack the vectors into data matrices whose *rows* are data items, as above, so the i'th row of $\mathcal{X}$ is $\mathbf{x}_i^T$ and the i'th row of $\mathcal{Y}$ is $\mathbf{y}_i^T$. Then $\mathbf{p}_x = matxX\mathbf{a}$ is a vector containing all the projections of the x part, and $\mathbf{p}_y = \mathcal{Y}\mathbf{b}$ is a vector containing all the projections of the y part. We want these projections to be "like" one another to the extent possible.

One criterion we can use is to maximize $\mathbf{p}_x^T\mathbf{p}_y$ by choice of $\mathbf{a}$ and $\mathbf{b}$. We must maximise

$$\mathbf{a}^T \max X^T\mathcal{Y}\mathbf{b} \text{ subject to } \mathbf{a}^T\mathbf{a} = 1 \text{ and } \mathbf{b}^T\mathbf{b} = 1.$$

Write λ_a and λ_b for Lagrange multipliers. To solve this problem, we must find $\mathbf{a}$ and $\mathbf{b}$ such that

$$\mathcal{X}^T\mathcal{Y}\mathbf{b} = \lambda_a\mathbf{a} \text{ and } \mathcal{Y}^T\mathcal{X}\mathbf{a} = \lambda_b\mathbf{b}.$$

We can substitute using the second equation to get

$$(\mathcal{X}^T\mathcal{Y}\mathcal{Y}^T\mathcal{X})\mathbf{a} = \lambda_a\lambda_b\mathbf{a}$$

which means that $\mathbf{a}$ is an eigenvector of $(\mathcal{X}^T\mathcal{Y}\mathcal{Y}^T\mathcal{X})$. It turns out we seek the eigenvector corresponding to the largest eigenvalue; equivalently, we must solve

$$\mathbf{a}^T(\mathcal{X}^T\mathcal{Y}\mathcal{Y}^T\mathcal{X})\mathbf{a} \text{ subject to } \mathbf{a}^T\mathbf{a} = 1$$

There is a straightforward way to obtain a second, third, etc. dimension. We take the dataset, and subtract the $\mathbf{a}$ component from each $\mathbf{x}_i$ and the $\mathbf{b}$ component from each $\mathbf{y}_i$; we now have a new dataset, and seek another $\mathbf{a}$, $\mathbf{b}$ using our procedure. These new directions must (a) maximise the covariance we are interested in and (b) are orthogonal to the original directions. As Figure **??** suggests, these projections of the data separate blobs of data with different labels.

****PROBLEM FIGURE

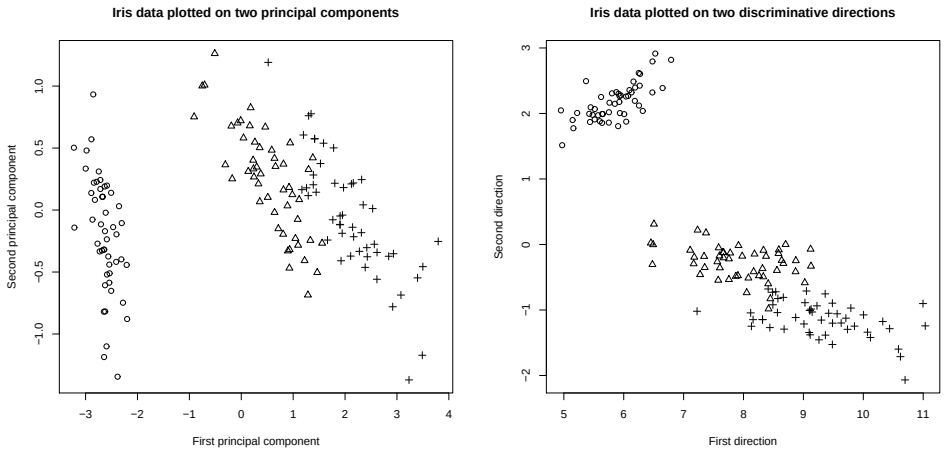

FIGURE 4.26:

4.5.4 Just a few Discriminative Directions with PLS1

Notice that we have, again, formed a (kind of) covariance matrix here, meaning that this method might not apply if the dimension of the data is big. But we could obtain $\mathbf{a}$ from an SVD. An SVD yields $\mathcal{U}\Sigma\mathcal{V}^T = \mathcal{Y}^T\mathcal{X}$, where $\mathcal{U}$ and $\mathcal{V}$ are orthonormal and Σ is diagonal. In turn, we have

$$\mathbf{a}^T(\mathcal{X}^T\mathcal{Y}\mathcal{Y}^T\mathcal{X})\mathbf{a} = \mathbf{a}^T\mathcal{V}\Sigma^2\mathcal{V}^T\mathbf{a}$$

so that $\mathbf{a}$ must be the column of $\mathcal{V}$ corresponding to the largest singular value. Now the argument I used for extracting principal components works for this problem, too. Recall I was looking for the unit $\mathbf{a}$ that maximized $\mathbf{a}^T\mathcal{X}^T\mathcal{X}\mathbf{a}$; I showed I could obtain this from an SVD of $\mathcal{X}$; then I argued that I could recover $\mathbf{a}$ by finding $\mathbf{a}$, $\mathbf{w}$ such that $\|\mathcal{X} - \mathbf{w}\mathbf{a}\|_F^2$ was minimized.

When we are looking for a discriminative direction, we would look for $\mathbf{a}$, $\mathbf{w}$ such that $\|\mathcal{Y}^T\mathcal{X} - \mathbf{a}\mathbf{w}^T\|_F^2$ is minimized. The procedure above applies. This algorithm is usually called **PLS1** (for partial least squares one).

4.6 MULTI-DIMENSIONAL SCALING

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

4.6.1 Principal Coordinate Analysis

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

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

(where the superscript is to remind you that this is a squared distance). We could build an $N \times N$ matrix of squared distances, which we write $\mathcal{D}^{(2)}(\mathbf{x})$. The i, j'th entry in this matrix is $D_{ij}^{(2)}(\mathbf{x})$, and the $\mathbf{x}$ argument means that the distances are between points in the high-dimensional space. Now we could choose the $\mathbf{v}_i$ to make

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

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

In its current form, the expression is difficult to deal with, but we can refine it. Because translation does not change the distances between points, it cannot change either of the $\mathcal{D}^{(2)}$ matrices. So it is enough to solve the case when the mean of the points $\mathbf{x}_i$ is zero. We can assume that $\frac{1}{N}\sum_i \mathbf{x}_i = \mathbf{0}$. Now write $\mathbf{1}$ for the n-dimensional vector containing all ones, and $\mathcal{I}$ for the identity matrix. Notice that

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

Now write

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

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

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

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

We can choose a set of $\mathbf{v}_i$ that makes $\mathcal{D}^{(2)}(\mathbf{v})$ close to $\mathcal{D}^{(2)}(\mathbf{x})$ quite easily, using the method of the previous section. Take the dataset of N d-dimensional column vectors $\mathbf{x}_i$, and form a matrix $\mathcal{X}$ by stacking the vectors, so

$$\mathcal{X} = \left[\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N \right].$$

In this notation, we have

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

This matrix is symmetric, and it is positive semidefinite. It can't be positive definite, because the data is zero mean, so $\mathcal{M}(\mathbf{x})\mathbf{1} = 0$. The $\mathcal{M}(\mathbf{v})$ we seek must (a) be as close as possible to $\mathcal{M}(\mathbf{x})$ and (b) have rank 2. It must have rank 2 because

there must be some $\mathcal{V}$ which is $2 \times N$ so that $\mathcal{M}(\mathbf{v}) = \mathcal{V}^T\mathcal{V}$. The columns of this $\mathcal{V}$ are our $\mathbf{v}_i$.

We can use the method of section 14.1.2 to construct $\mathcal{M}(\mathbf{v})$ and $\mathcal{V}$. As usual, we write $\mathcal{U}$ for the matrix of eigenvectors of $\mathcal{M}(\mathbf{x})$, Λ for the diagonal matrix of eigenvalues sorted in descending order, Λ_2 for the 2×2 upper left hand block of Λ, and $\Lambda_2^{(1/2)}$ for the matrix of positive square roots of the eigenvalues. Then our methods yield

$$\mathcal{M}(\mathbf{v}) = \mathcal{U}_2\Lambda_2^{(1/2)}\Lambda_2^{(1/2)}\mathcal{U}_2^T$$

and

$$\mathcal{V} = \Lambda_2^{(1/2)}\mathcal{U}_2^T$$

and we can plot these $\mathbf{v}_i$ (example in section 9.9). This method for constructing a plot is known as **principal coordinate analysis**.

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

Procedure: 4.3 *Principal Coordinate Analysis*

Assume we have a matrix $D^{(2)}$ consisting of the squared differences between each pair of N points. We do not need to know the points. We wish to compute a set of points in r dimensions, such that the distances between these points are as similar as possible to the distances in $D^{(2)}$. Form $\mathcal{A} = \left[\mathcal{I} - \frac{1}{N}\mathbf{1}\mathbf{1}^T\right]$. Form $\mathcal{W} = \frac{1}{2}\mathcal{A}\mathcal{D}^{(2)}\mathcal{A}^T$. Form $\mathcal{U}$, Λ, such that $\mathcal{W}\mathcal{U} = \mathcal{U}\Lambda$ (these are the eigenvectors and eigenvalues of $\mathcal{W}$). Ensure that the entries of Λ are sorted in decreasing order. Choose r, the number of dimensions you wish to represent. Form Λ_r, the top left $r \times r$ block of Λ. Form $\Lambda_r^{(1/2)}$, whose entries are the positive square roots of Λ_r. Form $\mathcal{U}_r$, the matrix consisting of the first r columns of $\mathcal{U}$. Then $\mathcal{V} = \Lambda_2^{(1/2)}\mathcal{U}_2^T = [\mathbf{v}_1, \ldots, \mathbf{v}_N]$ is the set of points to plot.

4.6.2 Example: Mapping with Multidimensional Scaling

Multidimensional scaling gets positions (the $\mathcal{V}$ of section 4.6.1) from distances (the $\mathcal{D}^{(2)}(\mathbf{x})$ of section 4.6.1). 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 apply multidimensional scaling. Table **??** shows distances between the South African provincial capitals, in kilometers, rounded to the nearest kilometer. I then used principal coordinate analysis to find positions for each capital, and rotated, translated and scaled the resulting plot to check it against a real map (Figure 4.27).

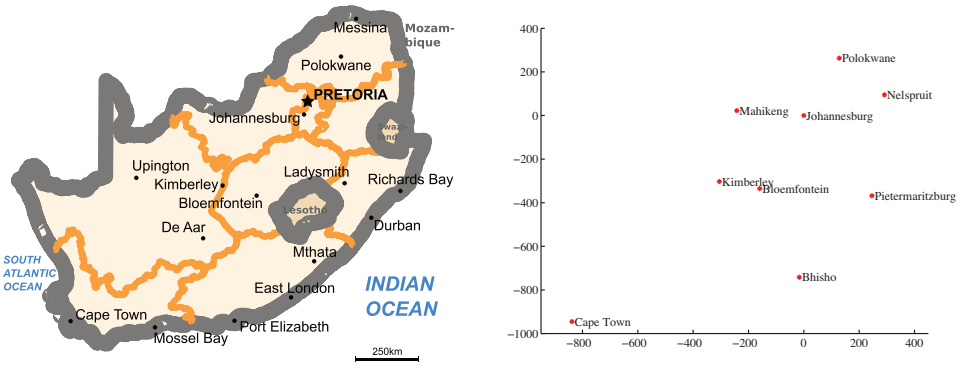

FIGURE 4.27: *On the* **left**, *a public domain map of South Africa, obtained from* http://commons.wikimedia.org/wiki/File:Map_of_South_Africa.svg, *and edited to remove surrounding countries. On the* **right**, *the locations of the cities inferred by multidimensional scaling, rotated, translated and scaled to allow a comparison to the map by eye. The map doesn't have all the provincial capitals on it, but it's easy to see that MDS has placed the ones that are there in the right places (use a piece of ruled tracing paper to check).*

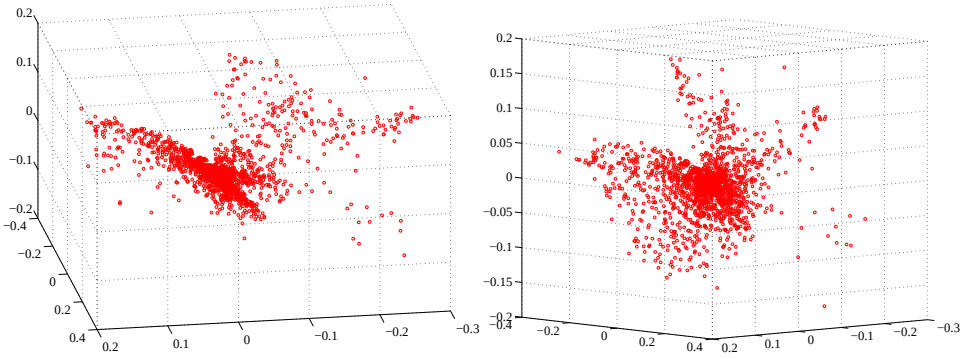

FIGURE 4.28: *Two views of the spectral data of section 4.4.5, plotted as a scatter plot by applying principal coordinate analysis to obtain a 3D set of points. Notice that the data spreads out in 3D, but seems to lie on some structure; it certainly isn't a single blob. This suggests that further investigation would be fruitful.*

One natural use of principal coordinate analysis is to see if one can spot any structure in a dataset. Does the dataset form a blob, or is it clumpy? This isn't a perfect test, but it's a good way to look and see if anything interesting is happening. In figure 4.28, I show a 3D plot of the spectral data, reduced to three dimensions using principal coordinate analysis. The plot is quite interesting. You should notice that the data points are spread out in 3D, but actually seem to lie on a complicated curved surface — they very clearly don't form a uniform blob. To me, the structure looks somewhat like a butterfly. I don't know why this occurs, 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.

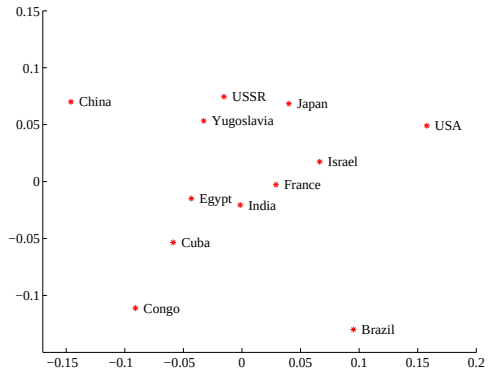

FIGURE 4.29: *A map of country similarity, prepared from the data of figure* **??**. *The map is often interpreted as showing a variation in development or wealth (poorest at bottom left to richest at top right); and freedom (most repressed at top left and freeest at bottom right). I haven't plotted these axes, because the interpretation wouldn't be consistent with current intuition (the similarity data is forty years old, and quite a lot has happened in that time).*

Our algorithm has one really interesting property. In some cases, we do not actually know the datapoints as vectors. Instead, we *just* know distances between the datapoints. This happens often in the social sciences, but there are important cases in computer science as well. As a rather contrived example, one could survey people about breakfast foods (say, eggs, bacon, cereal, oatmeal, pancakes, toast, muffins, kippers and sausages for a total of 9 items). We ask each person to rate the similarity of each pair of distinct items on some scale. We advise people that similar 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 $\mathbf{V}$ and produce a scatter plot. This plot is quite revealing, because items that most people think are easily substituted appear close together, and items that are hard to substitute are far apart. The neat trick here is that we did not start with a $\mathcal{X}$, but with just a set of distances; but we were able to associate a vector with "eggs", and produce a meaningful plot.

Table **??** shows data from one such example. Students were interviewed (in 1971! things may have changed since then) about their perceptions of the similarity of countries. The averaged perceived similarity is shown in table **??**. Large numbers

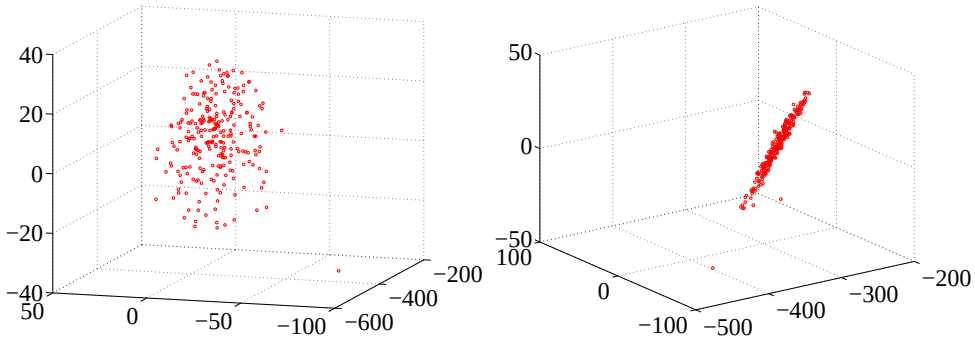

FIGURE 4.30: *Two views of a multidimensional scaling to three dimensions of the height-weight dataset. Notice how the data seems to lie in a flat structure in 3D, with one outlying data point. This means that the distances between data points can be (largely) explained by a 2D representation.*

reflect high similarity, so we can't use these numbers directly. It is reasonable to turn these numbers into distances by (a) using 0 as the distance between a country and itself and (b) using $e^{-s_{ij}}$ as the distance between countries i and j (where s_{ij} is the similarity between them). Once we have distances, we can apply the procedure of section 4.6.1 to get points, then plot a scatter plot (Figure 4.29).

4.7 EXAMPLE: UNDERSTANDING HEIGHT AND WEIGHT

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

Figure **??** shows a multidimensional scaling of this dataset down to three dimensions. The dataset seems to lie on a (fairly) flat structure in 3D, meaning that inter-point distances are relatively well explained by a 2D representation. Two points seem to be special, and lie far away from the flat structure. The structure isn't perfectly flat, so there will be small errors in a 2D representation; but it's clear that a lot of dimensions are redundant. Figure 4.31 shows a 2D representation of these points. They form a blob that is stretched along one axis, and there is no sign of multiple blobs. There's still at least one special point, which we shall ignore but might be worth investigating further. The distortions involved in squashing this dataset down to 2D seem to have made the second special point less obvious than it was in figure **??**.

The next step is to try a principal component analysis. Figure 4.32 shows the mean of the dataset. The components of the dataset have different units, and shouldn't really be compared. But it is difficult to interpret a table of 16 numbers, so I have plotted the mean by showing a vertical bar for each component. Figure 4.33

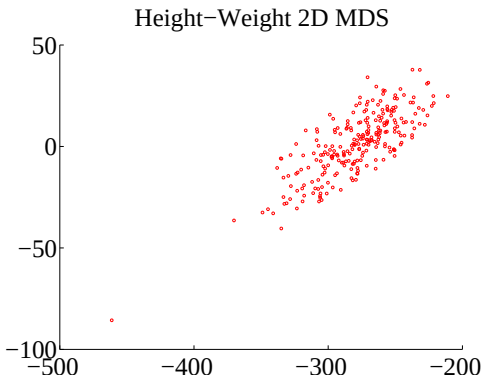

FIGURE 4.31: *A multidimensional scaling to two dimensions of the height-weight dataset. One data point is clearly special, and another looks pretty special. The data seems to form a blob, with one axis quite a lot more important than another.*

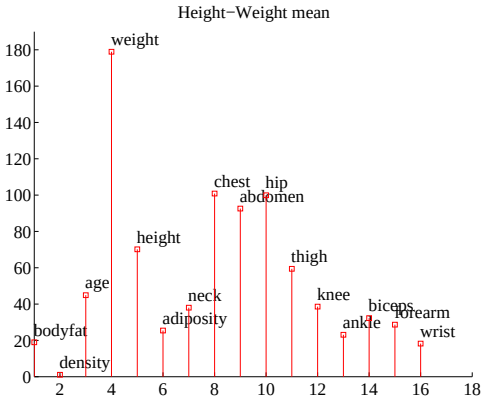

FIGURE 4.32: *The mean of the bodyfat.xls dataset. Each component is likely in a different unit (though I don't know the units), making it difficult to plot the data without being misleading. I've adopted one solution here, by plotting each component as a vertical bar, and labelling the bar. You shouldn't try to compare the values to one another. Instead, think of this plot as a compact version of a table.*

shows the eigenvalues of the covariance for this dataset. Notice how one dimension is very important, and after the third principal component, the contributions become small. Of course, I could have said "fourth", or "fifth", or whatever — the precise choice depends on how small a number you think is "small".

Figure 4.33 also shows the first principal component. The eigenvalues justify thinking of each data item as (roughly) the mean plus some weight times this principal component. From this plot you can see that data items with a larger value of *weight* will also have larger values of most other measurements, except *age* and *density*. You can also see how much larger; if the weight goes up by 8.5 units, then the abdomen will go up by 3 units, and so on. This explains the main variation

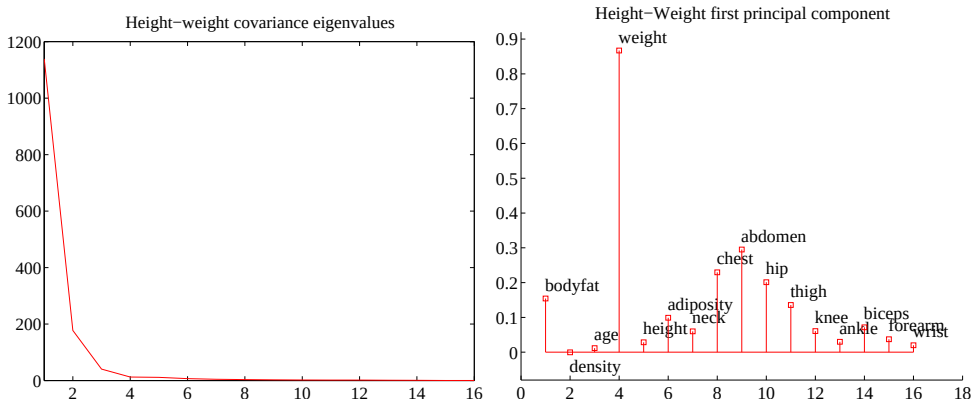

FIGURE 4.33: *On the* **left**, *the eigenvalues of the covariance matrix for the bodyfat data set. Notice how fast the eigenvalues fall off; this means that most principal components have very small variance, so that data can be represented well with a small number of principal components. On the* **right**, *the first principal component for this dataset, plotted using the same convention as for figure 4.32.*

in the dataset.

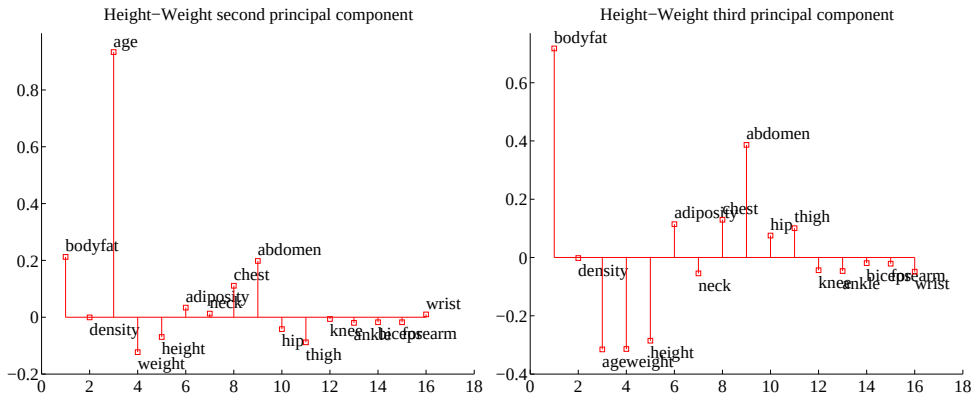

FIGURE 4.34: *On the* **left**, *the second principal component, and on the* **right** *the third principal component of the height-weight dataset.*

In the rotated coordinate system, the components are not correlated, and they have different variances (which are the eigenvalues of the covariance matrix). You can get some sense of the data by adding these variances; in this case, we get 1404. This means that, in the translated and rotated coordinate system, the average data point is about $37 = \sqrt{1404}$ units away from the center (the origin). Translations and rotations do not change distances, so the average data point is about 37 units from the center in the original dataset, too. If we represent a datapoint by using the mean and the first three principal components, there will be some error. We

can estimate the average error from the component variances. In this case, the sum of the first three eigenvalues is 1357, so the mean square error in representing a datapoint by the first three principal components is $\sqrt{(1404 - 1357)}$, or 6.8. The relative error is $6.8/37 = 0.18$. Another way to represent this information, which is more widely used, is to say that the first three principal components explain all but $(1404 - 1357)/1404 = 0.034$, or 3.4% of the variance; notice that this is the square of the relative error, which will be a much smaller number.

All this means that explaining a data point as the mean and the first three principal components produces relatively small errors. Figure 4.35 shows the second and third principal component of the data. These two principal components suggest some further conclusions. As *age* gets larger, *height* and *weight* get slightly smaller, but the weight is redistributed; *abdomen* gets larger, whereas *thigh* gets smaller. A smaller effect (the third principal component) links *bodyfat* and *abdomen*. As *bodyfat* goes up, so does *abdomen*.

4.8 WHAT YOU SHOULD REMEMBER - NEED SOMETHING

PROBLEMS

Summaries

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

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

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

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

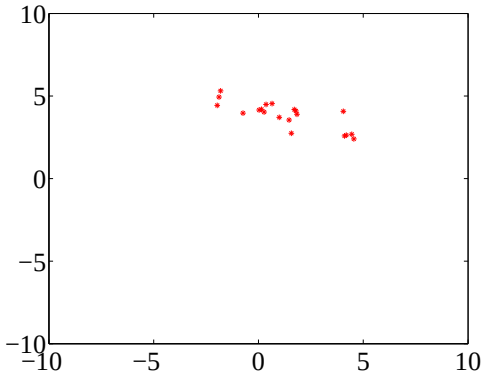

FIGURE 4.35: *Figure for the question*

4.2. On Figure 4.35, mark the mean of the dataset, the first principal component,

Listing 4.1: R code for iris example.

```
# r code for scatterplot of iris data
irisdat<-read.csv('iris.dat', header=FALSE);
library('lattice')
numiris=irisdat[,c(1, 2, 3, 4)]
postscript("irisscatterplot.eps")
# so that I get a postscript file
speciesnames<-c('setosa','versicolor', 'virginica')
pchr<-c(1, 2, 3)
colr<-c('red', 'green', 'blue', 'yellow', 'orange')
ss<-expand.grid(species=1:3)
parset<-with(ss, simpleTheme(pch=pchr[species],
                             col=colr[species]))
splom(irisdat[, c(1:4)], groups=irisdat$V5,
      par.settings=parset,
      varnames=c('Sepal\nLength', 'Sepal\nWidth',
          'Petal\nLength', 'Petal\nWidth'),
      key=list(text=list(speciesnames),
           points=list(pch=pchr), columns=3))
dev.off()
```

and the second principal component.

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

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

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

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

PROGRAMMING EXERCISES

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

(a) Plot a scatterplot matrix of this dataset, showing each species with a different marker. The fragment of R code in listing 4.1 should take you most of the way.

(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

(c) Now use PLS1 to obtain two discriminative directions, and project the data on to those directions. Does the plot look better? Explain *Keep in mind that the most common error here is to forget that the $\mathcal{X}$ and the $\mathcal{Y}$ in PLS1 are centered - i.e. we subtract the mean.*

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

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

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

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

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

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

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

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

(a) Plot this dataset on the first three principal components, using different markers for benign and malignant cases. What do you see?

(b) Now use PLS1 to obtain three discriminative directions, and project the data on to those directions. Does the plot look better? Explain *Keep in mind that the most common error here is to forget that the $\mathcal{X}$ and the $\mathcal{Y}$ in PLS1 are centered - i.e. we subtract the mean. Once you have computed the first (etc. direction, you should subtract it from $\mathcal{X}$,* but leave $\mathcal{Y}$ alone.

CHAPTER 5

Clustering: Models of High Dimensional Data

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

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

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

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

5.1 AGGLOMERATIVE AND DIVISIVE CLUSTERING

There are two natural algorithms for clustering. In **divisive clustering**, the entire data set is regarded as a cluster, and then clusters are recursively split to yield a good clustering (Algorithm 5.2). In **agglomerative clustering**, each data item is regarded as a cluster, and clusters are recursively merged to yield a good clustering (Algorithm 5.1).

```
Make each point a separate cluster
Until the clustering is satisfactory
     Merge the two clusters with the
       smallest inter-cluster distance
end
```

Algorithm 5.1: *Agglomerative Clustering or Clustering by Merging.*

```
Construct a single cluster containing all points
Until the clustering is satisfactory
     Split the cluster that yields the two
       components with the largest inter-cluster distance
end
```

Algorithm 5.2: *Divisive Clustering, or Clustering by Splitting.*

There are two major issues in thinking about clustering:

- *What is a good inter-cluster distance?* Agglomerative clustering uses an inter-cluster distance to fuse nearby clusters; divisive clustering uses it to split insufficiently coherent clusters. Even if a natural distance between data points is available (which might not be the case for vision problems), 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**).

- *How many clusters are there?* This is an intrinsically difficult task if there is no model for the process that generated the clusters. The algorithms we 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 5.1).

The main difficulty in using a divisive model is knowing where to split. This is sometimes made easier for particular kinds of data. For example, we could segment an image by clustering pixel values. In this case, you can sometimes find good splits by constructing a histogram of intensities, or of color values.

Another important thing to notice about clustering from the example of figure 5.1 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.

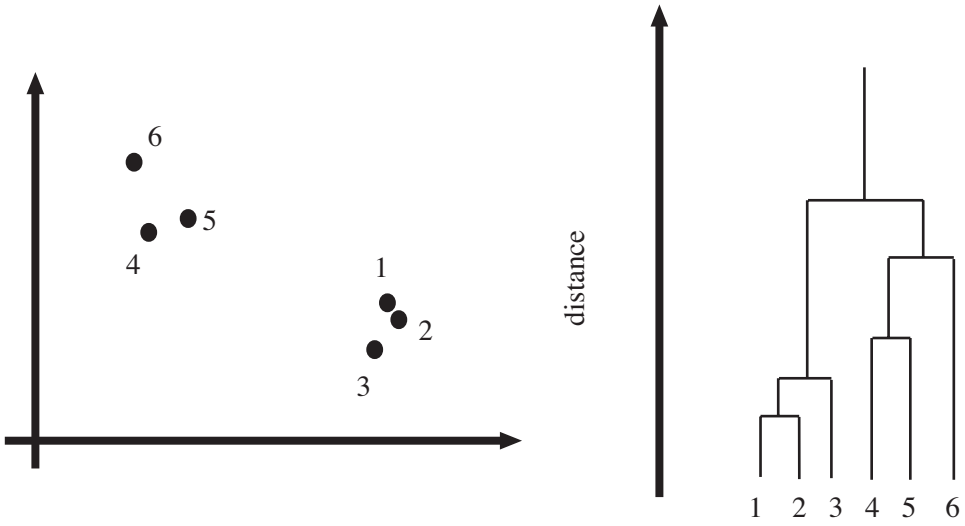

FIGURE 5.1: **Left**, *a data set;* **right**, *a dendrogram obtained by agglomerative clustering using single-link clustering. If one selects a particular value of distance, then a horizontal line at that distance splits the dendrogram into clusters. This representation makes it possible to guess how many clusters there are and to get some insight into how good the clusters are.*

5.1.1 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 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**

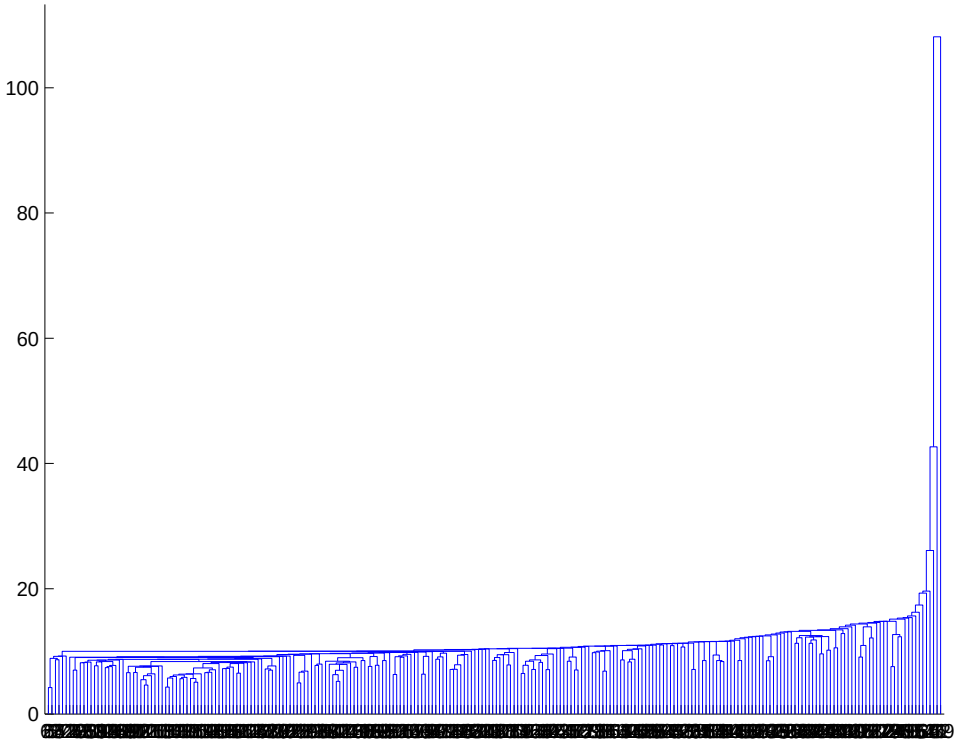

FIGURE 5.2: *A dendrogram obtained from the body fat dataset, using single link clustering. Recall that the data points are on the horizontal axis, and that the vertical axis is distance; there is a horizontal line linking two clusters that get merged, established at the height at which they're merged. I have plotted the entire dendrogram, despite the fact it's a bit crowded at the bottom, because it shows that most data points are relatively close (i.e. there are lots of horizontal branches at about the same height).*

(because you make the data more like white noise); I described how to do this in section **??**.

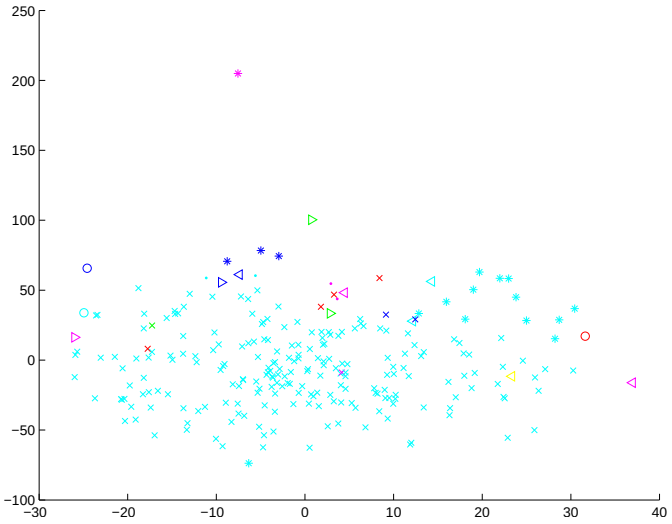

FIGURE 5.3: *A clustering of the body fat dataset, using agglomerative clustering, single link distance, and requiring a maximum of 30 clusters. I have plotted each cluster with a distinct marker (though some markers differ only by color; you might need to look at the PDF version to see this figure at its best). Notice that one cluster contains much of the data, and that there are a set of small isolated clusters. The original data is 16 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 16 dimensional space).*

Worked example 5.1 *Agglomerative clustering in Matlab*

Cluster the height-weight dataset of http://www2.stetson.edu/~jrasp/data.htm (look for bodyfat.xls) using an agglomerative clusterer, and describe the results.

Solution: Matlab provides some tools that are useful for agglomerative clustering. These functions use a scheme where one first builds the whole tree of merges, then analyzes that tree to decide which clustering to report. `linkage` will determine which pairs of clusters should be merged at which step (there are arguments that allow you to choose what type of inter-cluster distance it should use); `dendrogram` will plot you a dendrogram; and `cluster` will extract the clusters from the linkage, using a variety of options for choosing the clusters. I used these functions to prepare the dendrogram of figure 5.2 for the height-weight dataset of section **??** (from http://www2.stetson.edu/~jrasp/data.htm; look for bodyfat.xls). I deliberately forced Matlab to plot the whole dendrogram, which accounts for the crowded look of the figure (you can allow it to merge small leaves, etc.). I used a single-link strategy. In particular, notice that many data points are about the same distance from one another, which suggests a single big cluster with a smaller set of nearby clusters. The clustering of figure 5.3 supports this view. I plotted the data points on the first two principal components, using different colors and shapes of marker to indicate different clusters. There are a total of 30 clusters here, though most are small.

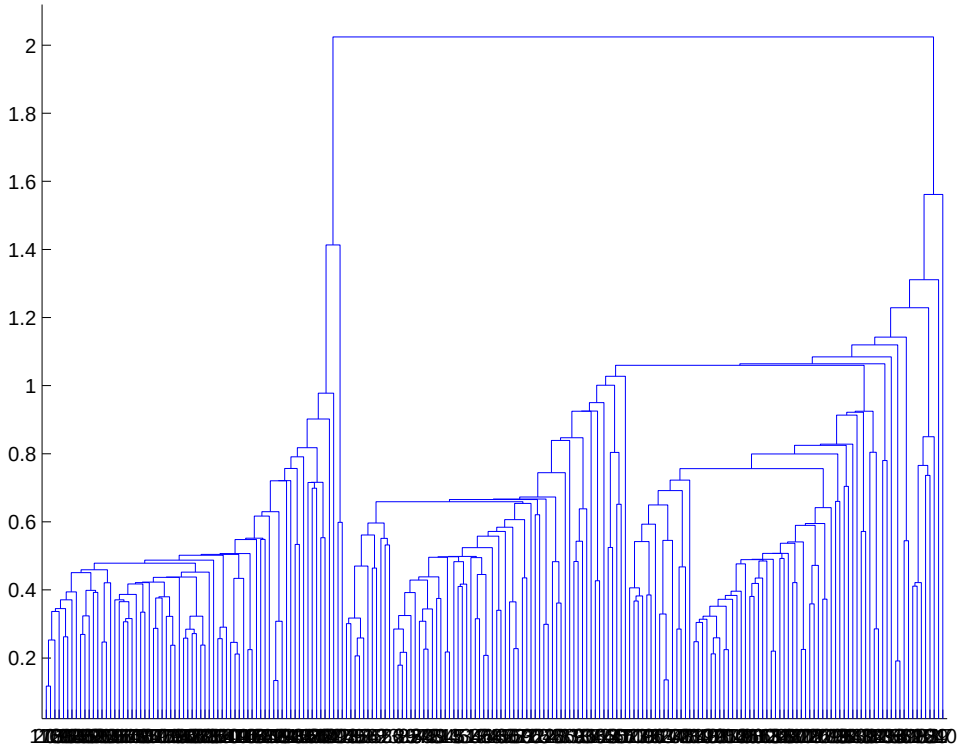

FIGURE 5.4: *A dendrogram obtained from the seed dataset, using single link clustering. Recall that the data points are on the horizontal axis, and that the vertical axis is distance; there is a horizontal line linking two clusters that get merged, established at the height at which they're merged. I 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 5.2 *Agglomerative clustering in Matlab, 2*

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. As you can see in figures 5.4 and 5.5, this data clusters rather well.

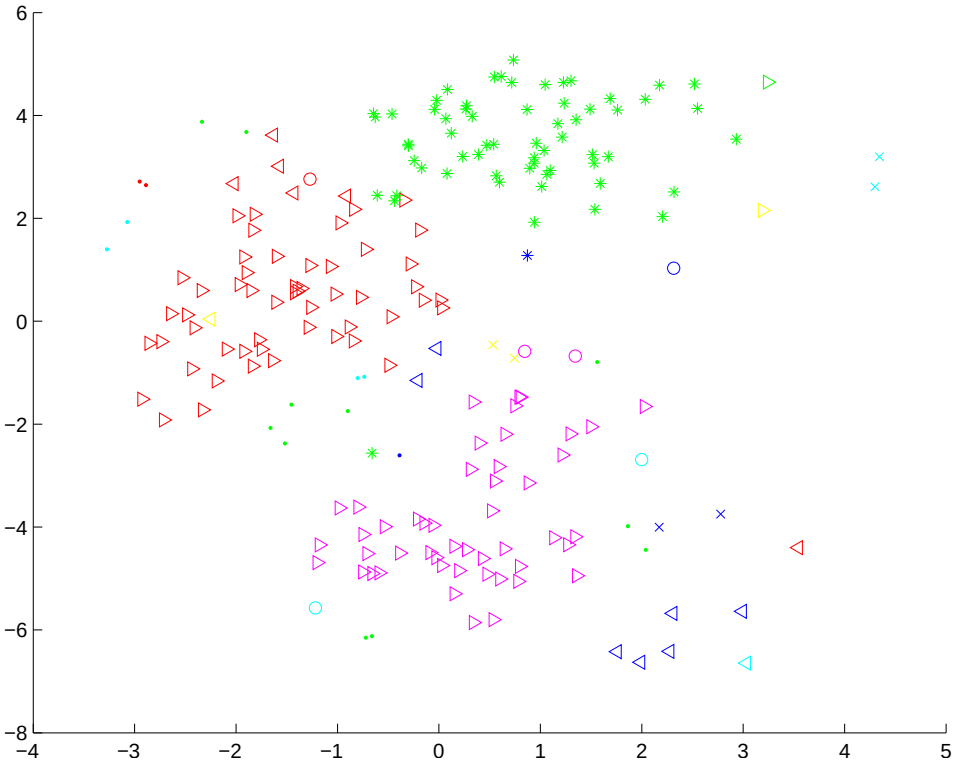

FIGURE 5.5: *A clustering of the seed dataset, using agglomerative clustering, single link distance, and requiring a maximum of 30 clusters. I have plotted each cluster with a distinct marker (though some markers differ only by color; you might need to look at the PDF version to see this figure at its best). Notice that there are a set of fairly natural isolated clusters. The original data is 8 dimensional, which presents plotting problems; I show a scatter plot on the first two principal components (though I computed distances for clustering in the original 8 dimensional space).*

Worked example 5.3 *Agglomerative clustering in R*

Cluster the questions in the student evaluation dataset from the UC Irvine Machine Learning Dataset Repository (you can find it at https://archive.ics.uci.edu/ml/datasets/Turkiye+Student+Evaluation; this dataset was donated by G. Gunduz and E. Fokoue), and display it on the first two principal components. You should use 5 clusters, but investigate the results of choosing others.

Solution: R provides tools for agglomerative clustering, too. I used the block of code shown in listing 5.1 to produce figure 5.6. The clustering shown in that figure is to 5 clusters. To get some idea of what the clusters are like, you can compute the per-cluster means, using a line in that listing. I found means where: all students in the cluster gave moderately low, low, moderately high and high answers to all questions, respectively; and one where all students answered 1 to all questions. There are 5820 students in this collection. The clusters suggest that answers tend to be quite strongly correlated — a student who gives a low answer to a question will likely give low answers to others, too. Choosing other numbers of clusters wasn't particularly revealing, though there were more levels to the answers.

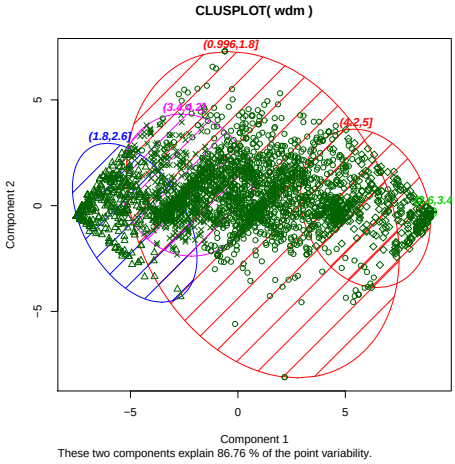

FIGURE 5.6: A clustering of the student evaluation dataset, using agglomerative clustering and 5 clusters. I produced this clustering with the code fragment in listing 5.1. One difficulty with this dataset is there are 28 questions (Q1-Q28), meaning the data is quite difficult to plot. You should be able to see the clusters on the figure, but notice that they are smeared on top of one another; this is because I had to project off 26 dimensions to produce the plot. The plotting function I used (`clusplot`) plots a covariance ellipse for each cluster.

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

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

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

Listing 5.1: R code for student example.

```
setwd('/users/daf/Current/courses/Probcourse/Clustering/RCode')
wdat<-read.csv('turkiye-student-evaluation_R_Specific.csv')
wdm<-wdat[,c(6:33)] # choose the questions
d <- dist(wdm, method = "euclidean") # distance matrix
fit <- hclust(d, method="ward") # the clustering
cm<-cov(wdm)
ev<-eigen(cm)
princ<-ev$vectors
proj<-t(rbind(t(princ[, 1]), t(princ[, 2])))
wdzm<-scale(wdm, center=TRUE, scale=FALSE) #subtract the mean
wdzm<-data.matrix(wdzm)
wdpcs<-wdzm%*%proj
kv<-c(5)
groups<-cutree(fit , k=kv)
wdbig<-wdm
wdbig$group<-groups
cg<-cut(groups , kv)
p<-ggplot(as.data.frame(wdpcs))+geom_point(aes(y=V1, x=V2, shape=cg))
setEPS()
postscript("clusters.eps")
p
dev.off()
cm1<-colMeans(subset(wdbig, group==1)) # mean of the 1st cluster
```

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

We then choose a start point by randomly choosing cluster centers, and then iterate these stages alternately. This process eventually converges to a local minimum of the objective function (the value either goes down or is fixed at each step, and it is bounded below). It is not guaranteed to converge to the global minimum of the objective function, however. It is also not guaranteed to produce k clusters, unless we modify the allocation phase to ensure that each cluster has some nonzero number of points. This algorithm is usually referred to as **k-means** (summarized in Algorithm 5.3).

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. A natural dataset to use to explore k-means is the iris data, where we know that the data should form three clusters (because there are three species). Recall this dataset from section **??**. I reproduce figure 4.5 from that section as figure 5.11, for comparison. Figures 5.8, **??** and **??** show different k-means clusterings of that data.

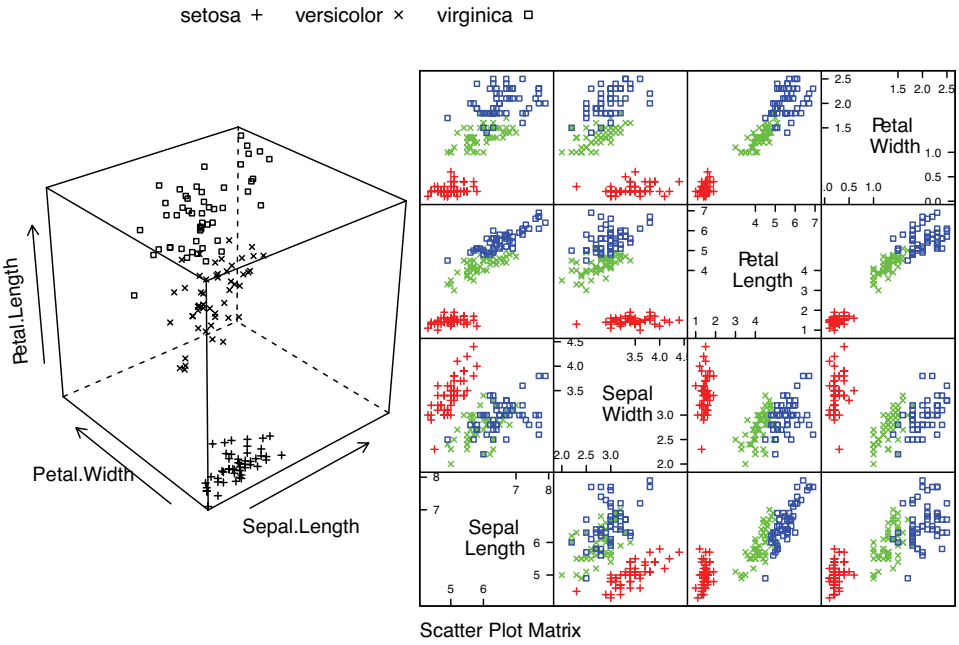

FIGURE 5.7: **Left:** *a 3D scatterplot for the famous Iris data, originally due to ***. I have chosen three variables from the four, and have plotted each species with a different marker. You can see from the plot that the species cluster quite tightly, and are different from one another.* **Right:** *a scatterplot matrix for the famous Iris data, originally due to ***. 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.*

Worked example 5.4 *K-means clustering in R*

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

Solution: I used the code fragment in listing 5.2, which produced figure **??**

5.2.1 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

Choose k data points 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.
end

Algorithm 5.3: *Clustering by K-Means.*

Listing 5.2: R code for iris example.

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

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 5.11 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

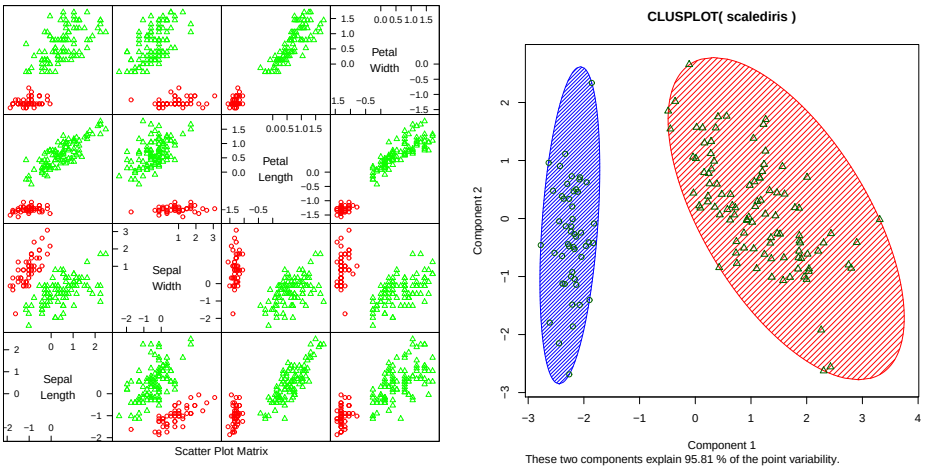

FIGURE 5.8: *On the* **left**, *a panel plot of the iris data clustered using k-means with* $k = 2$. *By comparison with figure 5.11, notice how the* versicolor *and* verginica *clusters appear to have been merged. On the* **right**, *this data set projected onto the first two principal components, with one blob drawn over each cluster.*

important, vector quantization, is described in section 5.3). There are usually natural ways to evaluate this application. For example, vector quantization is often used as an early step in texture recognition or in image matching; here one can evaluate the error rate of the recognizer, or the accuracy of the image matcher. One then chooses the k that gets the best evaluation score on validation data. In this view, the issue is not how good the clustering is; it's how well the system that uses the clustering works.

5.2.2 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.

We allow each point to carry a total weight of 1. In the conventional k-means algorithm, it must choose a single cluster, and assign its weight to that cluster alone. In soft-assignment k-means, the point can allocate some weight to each cluster center, as long as (a) the weights are all non-negative and (b) the weights sum to one. Write $w_{i,j}$ for the weight connecting point i to cluster center j. We interpret these weights as the degree to which the point participates in a particular cluster. We require $w_{i,j} \geq 0$ and $\sum_j w_{i,j} = 1$.

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

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

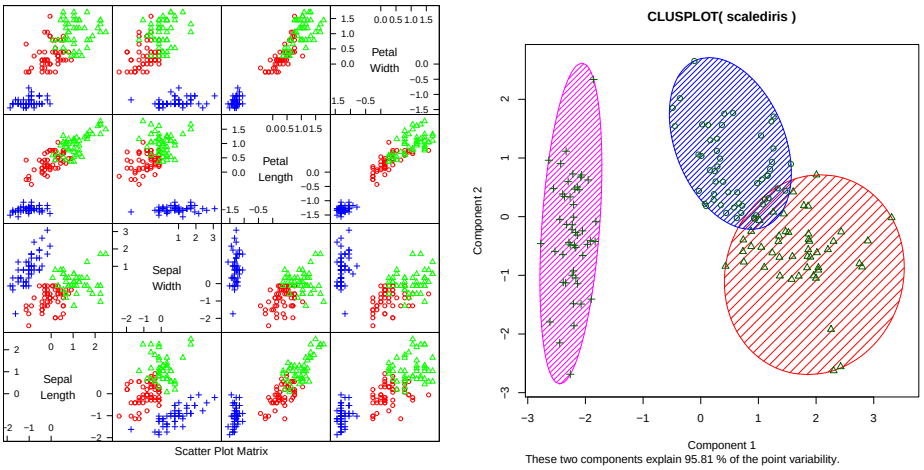

FIGURE 5.9: *On the* **left**, *a panel plot of the iris data clustered using k-means with* $k = 3$. *By comparison with figure 5.11, notice how the clusters appear to follow the species labels. On the* **right**, *this data set projected onto the first two principal components, with one blob drawn over each cluster.*

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

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

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

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

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

Notice that k-means is a special case of this algorithm where σ limits to zero. In this case, each point has a weight of one for some cluster, and zero for all others, and the weighted mean becomes an ordinary mean. I have collected the description into Algorithm 5.4 for convenience.

5.2.3 General Comments on K-Means

If you experiment with k-means, you will notice one irritating habit of the algorithm. It almost always produces either some rather spread out clusters, or some single element clusters. Most clusters are usually rather tight and blobby clusters, but there is usually one or more bad cluster. This is fairly easily explained. Because every data point must belong to some cluster, data points that are far from all

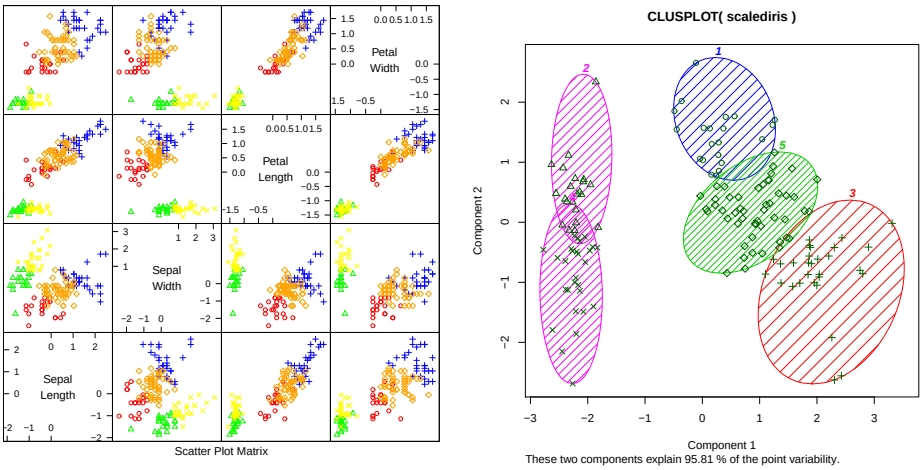

FIGURE 5.10: *On the* **left**, *a panel plot of the iris data clustered using k-means with* $k = 5$. *By comparison with figure 5.11, notice how* setosa *seems to have been broken in two groups, and* versicolor *and* verginica *into a total of three . On the* **right**, *this data set projected onto the first two principal components, with one blob drawn over each cluster.*

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.

In some cases, we want to cluster objects that can't be averaged. For example, you can compute distances between two trees but you can't meaningfully average them. In some cases, you might have a table of distances between objects, but not know vectors representing the objects. For example, one could collect data on the similarities between countries (as in Section 4.6.2, particularly Figure 4.29), then try and cluster using this data (similarities can be turned into distances by, for example, taking the negative logarithm). 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, but the rest of the algorithm has a familiar form. We assume the number of medoids is known, and initialize these randomly. We then iterate two procedures. In the first,

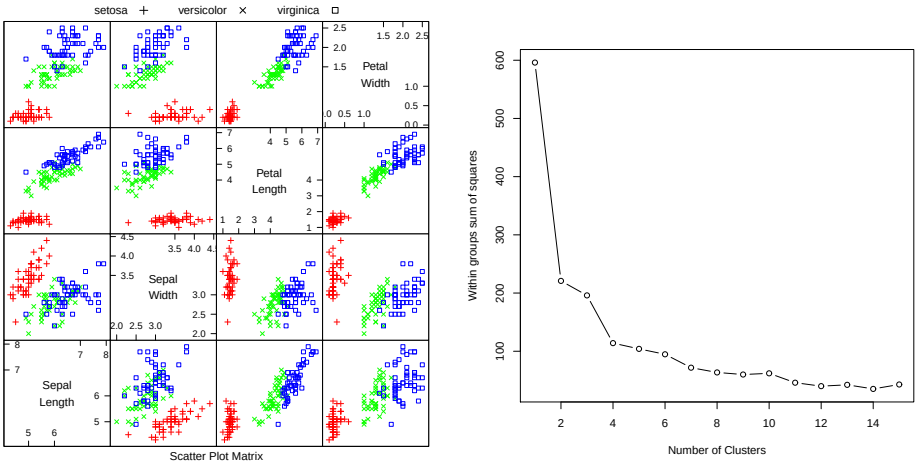

FIGURE 5.11: *On the* **left**, *the scatterplot matrix for the Iris data, for reference. On the* **right**, *a plot of the value of the cost function for each of several different values of* k. *Notice how there is a sharp drop in cost going from* $k = 1$ *to* $k = 2$, *and again at* $k = 4$; *after that, the cost falls off slowly. This suggests using* $k = 2$, $k = 3$, *or* $k = 4$, *depending on the precise application.*

we allocate data points to medoids. In the second, we choose the best medoid for each cluster by finding the medoid that minimizes the sum of distances of points in the cluster to that medoid (blank search is fine).

5.3 DESCRIBING REPETITION WITH VECTOR QUANTIZATION

Repetition is an important feature of many interesting signals. For example, images contain *textures*, which are orderly patterns that look like large numbers of small structures that are repeated. Examples include the spots of animals such as leopards or cheetahs; the stripes of animals such as tigers or zebras; the patterns on bark, wood, and skin. Similarly, speech signals contain *phonemes* — characteristic, stylised sounds that people assemble together to produce speech (for example, the "ka" sound followed by the "tuh" sound leading to "cat"). Another example comes from accelerometers. If a subject wears an accelerometer while moving around, the signals record the accelerations during their movements. So, for example, brushing one's teeth involves a lot of repeated twisting movements at the wrist, and walking involves swinging the hand back and forth.

Repetition occurs in subtle forms. The essence is that a small number of local patterns can be used to represent a large number of examples. You see this effect in pictures of scenes. If you collect many pictures of, say, a beach scene, you will expect most to contain some waves, some sky, and some sand. The individual patches of wave, sky or sand can be surprisingly similar, and different images are made by selecting some patches from a vocabulary of patches, then placing them down to form an image. Similarly, pictures of living rooms contain chair patches, TV patches, and carpet patches. Many different living rooms can be made from

Choose k data points to act as cluster centers
Until the cluster centers change very little
    First, we estimate the weights

    For i indexing data points
        For j indexing cluster centers
            Compute $s_{i,j} = e^{\frac{-\|\mathbf{x}_i - \mathbf{c}_j\|}{2\sigma^2}}$

    For i indexing data points
        For j indexing cluster centers
            Compute $w_{i,j} = s_{i,j} / \sum_{l=1} k s_{i,l}$

    Now, we re-estimate the centers

    For j indexing cluster centers
        Compute $\mathbf{c}_j = \frac{\sum_i w_{i,j} \mathbf{x}_i}{\sum_i w_{i,j}}$

end

Algorithm 5.4: *Soft Clustering by K-Means.*

small vocabularies of patches; but you won't often see wave patches in living rooms, or carpet patches in beach scenes.

An important part of representing signals that repeat is building a vocabulary of patterns that repeat, then describing the signal in terms of those patterns. For many problems, problems, knowing what vocabulary elements appear and how often is much more important than knowing where they appear. For example, if you want to tell the difference between zebra's 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.

5.3.1 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; in a sound file, which is likely represented as a vector, it might be a subvector of fixed size. But finding patterns that appear often is hard to do, because the signal is continuous — each pattern will be slightly different, so we cannot simply count how many times a particular pattern occurs.

Here is a strategy. First, we take a training set of signals, and cut each signal into vectors of fixed dimension (say d). It doesn't seem to matter too much if these

vectors overlap or not. We then build a set of clusters out of these vectors; this set of clusters is often thought of as a dictionary. We can now now describe any new vector with the cluster center closest to that vector. This means that a vector in a continuous space is described with a number in the range $[1, \ldots, k]$ (where you get to choose k), and two vectors that are close should be described by the same number. This strategy is known as **vector quantization**.

We can now build features that represent important repeated structure in signals. We take a signal, and cut it up into vectors of length d. These might overlap, or be disjoint; we follow whatever strategy we used in building the dictionary. We then take each vector, and compute the number that describes it (i.e. the number of the closest cluster center, as above). We then compute a histogram of the numbers we obtained for all the vectors in the signal. This histogram describes the signal.

Notice several nice features to this construction. First, it can be applied to anything that can be thought of in terms of vectors, so it will work for speech signals, sound signals, accelerometer signals, images, and so on. You might need to adjust some indices. For example, you cut the image into patches, then rearrange the patch to form a vector. As another example, accelerometer signals are three dimensional vectors that depend on time, so you cut out windows of a fixed number of time samples (say t), then rearrange to get a $3t$ dimensional vector.

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 **??**.

5.3.2 Example: Groceries in Portugal

At http://archive.ics.uci.edu/ml/datasets/Wholesale+customers, you will find a dataset giving sums of money spent annually on different commodities by customers in Portugal. The commodities are divided into a set of categories (fresh; milk; grocery; frozen; detergents and paper; and delicatessen) relevant for the study. These customers are divided by channel (two channels) and by region (three regions). You can think of the data as being divided into six groups (one for each channel-region pair). There are 440 records, and so there are many customers per group. Figure 5.12 shows a panel plot of the customer data; the data has been clustered, and I gave each of 20 clusters its own marker. Relatively little structure is apparent in this scatter plot. You can't, for example, see evidence of six groups that are cleanly separated.

It's unlikely that all the customers in a group are the same. Instead, we expect that there might be different "types" of customer. For example, customers who prepare food at home might spend more money on fresh or on grocery, and those who mainly buy prepared food might spend more money on delicatessan; similarly, coffee drinkers with cats or children might spend more on milk than the lactose-intolerant, and so on. Because some of these effects are driven by things like wealth and the tendency of people to like to have neighbors who are similar to

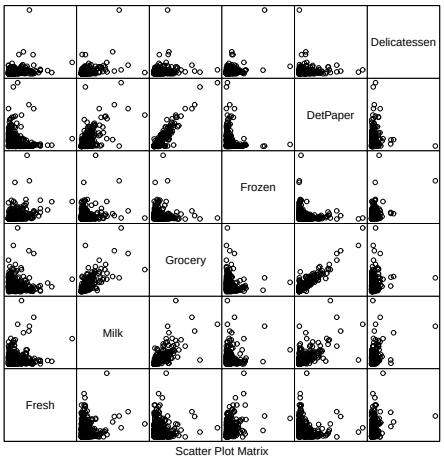

FIGURE 5.12: *A panel plot of the wholesale customer data of http://archive.ics.uci.edu/ml/datasets/Wholesale+customers, which records sums of money spent annually on different commodities by customers in Portugal. This data is recorded for six different groups (two channels each within three regions). I have plotted each group with a different marker, but you can't really see much structure here, for reasons explained in the text.*

them, you could expect that different groups contain different numbers 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.

An effect like this is hard to see on a panel plot (Figure 5.12). 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. There is another effect. If customers are grouped in the way I suggested above, then each group might look the same in a panel plot. A group of some milk-spenders and more detergent-spenders will have many 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 another group with more milk-spenders and some detergent-spenders will also look like two blobs, in about the same place. It will be hard to spot the difference. A histogram of the types within each group will make this difference obvious.

I used k-means clustering to cluster the customer data to 20 different clusters (Figure 5.14). I chose 20 rather arbitrarily, but with the plot of error against k in mind. Then I described the each group of data by the histogram of customer types that appeared in that group (Figure **??**). Notice how the distinction between the groups is now apparent — the groups do appear to contain quite different distributions of customer type. It looks as though the channels (rows in this figure) are more different than the regions (columns in this figure). To be more confident in this analysis, we would need to be sure that different types of customer really

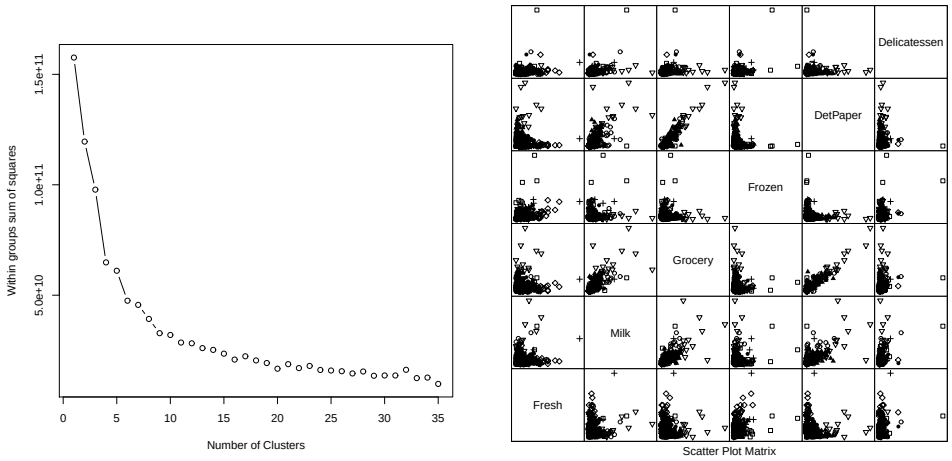

FIGURE 5.13: *On the* **left**, *the sum of squared error for clusterings of the customer data with k-means for k running from 2 to 35. This suggests using a k somewhere in the range 10-30; I chose 20. On the* **right**, *I have clustered this data to 20 cluster centers with k-means. The clusters do seem to be squashed together, but the plot on the left suggests that clusters do capture some important information. Using too few clusters will clearly lead to problems. Notice that I did not scale the data, because each of the measurements is in a comparable unit. For example, it wouldn't make sense to scale expenditures on fresh and expenditures on grocery with a different scale.*

are different. We could do this by repeating the analysis for fewer clusters, or by looking at the similarity of customer types.

5.3.3 Efficient Clustering and Hierarchical K Means

One important difficulty occurs in applications. We might need to have an enormous dataset (millions of image patches are a real possibility), and so a very large k. In this case, k means clustering becomes difficult because identifying which cluster center is closest to a particular data point scales linearly with k (and we have to do this for every data point at every iteration). There are two useful strategies for dealing with this problem.

The first is to notice that, if we can be reasonably confident that each cluster contains many data points, some of the data is redundant. We could randomly subsample the data, cluster that, then keep the cluster centers. This works, but doesn't scale particularly well.

A more effective strategy is to build a hierarchy of k-means clusters. We randomly subsample the data (typically, quite aggressively), then cluster this with a small value of k. Each data item is then allocated to the closest cluster center, and the data in each cluster is clustered again with k-means. We now have something that looks like a two-level tree of clusters. Of course, this process can be repeated to produce a multi-level tree of clusters. It is easy to use this tree to vector quantize a

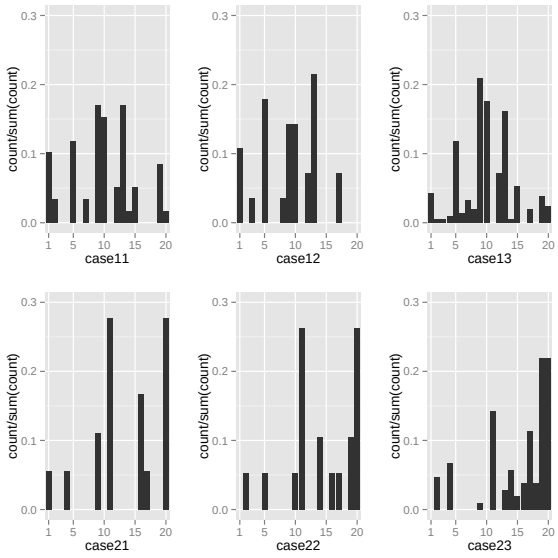

FIGURE 5.14: *The histogram of different types of customer, by group, for the customer data. Notice how the distinction between the groups is now apparent — the groups do appear to contain quite different distributions of customer type. It looks as though the channels (rows in this figure) are more different than the regions (columns in this figure).*

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.

5.3.4 Example: Activity from Accelerometer Data

A complex example dataset appears at https://archive.ics.uci.edu/ml/datasets/Dataset+for+ADL+Recognition+w
This dataset consists of examples of the signal from a wrist mounted accelerometer, produced as different subjects engaged in different activities of daily life. Activities include: brushing teeth, climbing stairs, combing hair, descending stairs, and so on. Each is performed by sixteen volunteers. The accelerometer samples the data at 32Hz (i.e. this data samples and reports the acceleration 32 times per second). The accelerations are in the x, y and z-directions. Figure 5.15 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

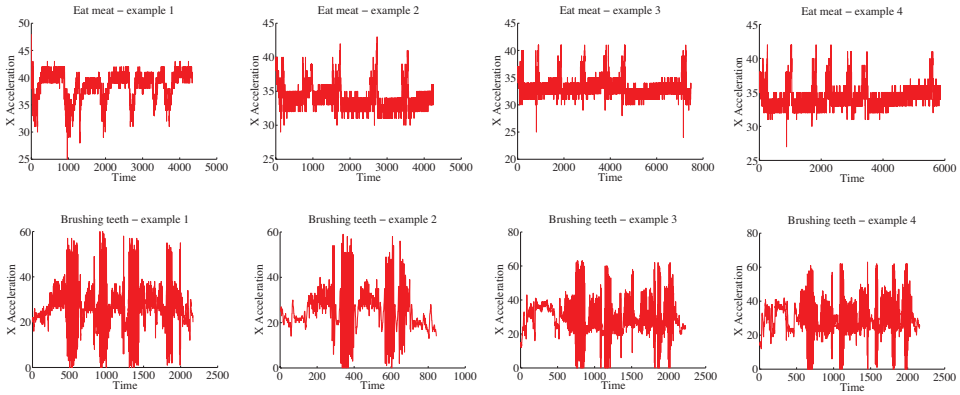

FIGURE 5.15: *Some examples from the accelerometer dataset at https://archive.ics.uci.edu/ml/datasets/Dataset+for+ADL+Recognition+with+Wrist-worn+Accelerometer. I have labelled each signal by the activity. These show acceleration in the X direction (Y and Z are in the dataset, too). There are four examples 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).*

with this by warping time and resampling the signal. For example, doing so will make a thorough toothbrusher look as though they are moving their hands very fast (or a careless toothbrusher look ludicrously slow: think speeding up or slowing down a movie). So we need a representation that can cope with signals that are a bit longer or shorter than other signals.

Another important property of these signals is that all examples of a particular activity should contain repeated patterns. For example, brushing teeth should show fast accelerations up and down; walking should show a strong signal at somewhere around 2 Hz; and so on. These two points should suggest vector quantization to you. Representing the signal in terms of stylized, repeated structures is probably a good idea because the signals probably contain these structures. And if we represent the signal in terms of the relative frequency with which these structures occur, the representation will have a fixed length, even if the signal doesn't. To do so, we need to consider (a) over what time scale we will see these repeated structures and (b) how to ensure we segment the signal into pieces so that we see these structures.

Generally, repetition in activity signals is so obvious that we don't need to be smart about segment boundaries. I broke these signals into 32 sample segments, one following the other. Each segment represents one second of activity. This is long enough for the body to do something interesting, but not so long that our representation will suffer if we put the segment boundaries in the wrong place. This resulted in about 40, 000 segments. I then used hierarchical k-means to cluster these segments. I used two levels, with 40 cluster centers at the first level, and 12 at the second. Figure 5.16 shows some cluster centers at the second level.

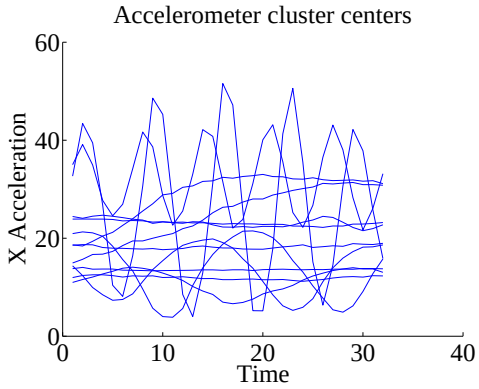

FIGURE 5.16: *Some cluster centers from the accelerometer dataset. Each cluster center represents a one-second burst of activity. There are a total of 480 in my model, which I built using hierarchical k-means. Notice there are a couple of centers that appear to represent movement at about 5Hz; another 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).*

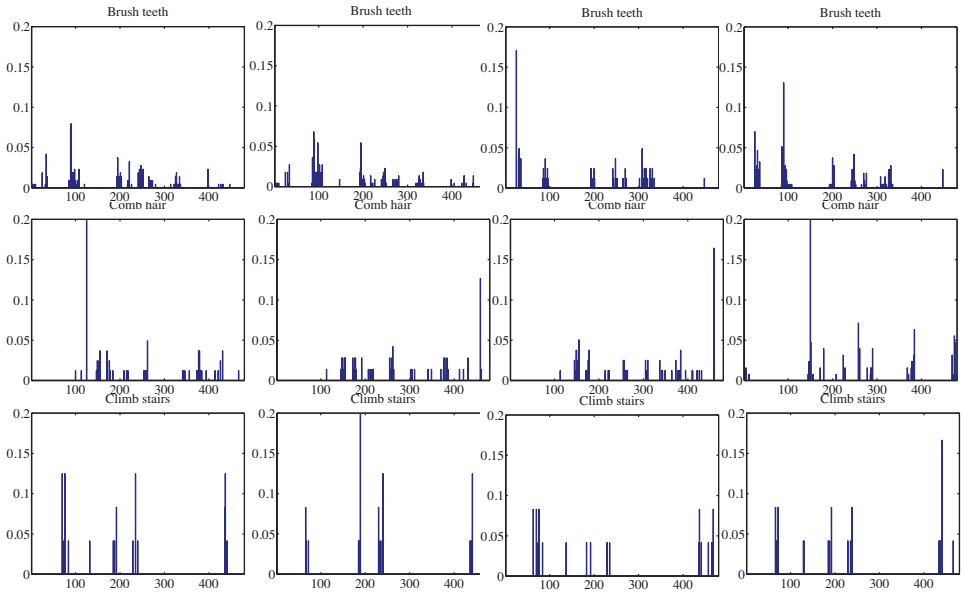

FIGURE 5.17: *Histograms of cluster centers for the accelerometer dataset, for different activities. You should notice that (a) these histograms look somewhat similar for different actors performing the same activity and (b) these histograms look somewhat different for different activities.*

I then computed histogram representations for different example signals (Figure 5.17). You should notice that when the activity label is different, the histogram looks different, too.

Another useful way to check this representation is to compare the average within class chi-squared distance with the average between class chi-squared distance. I computed the histogram for each example. Then, for each pair of examples, I computed the chi-squared distance between the pair. Finally, for each pair of *activity labels*, I computed the average distance between pairs of examples where one example has one of the activity labels and the other example has the other activity label. In the ideal case, all the examples with the same label would be very close to one another, and all examples with different labels would be rather different. Table 5.1 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.

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 5.1: *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.*

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.

5.4 YOU SHOULD

5.4.1 remember:

New term: clustering 97
New term: decorrelation 99
New term: whitening 99
New term: k-means 105
New term: vector quantization 113

PROGRAMMING EXERCISES

5.1. Obtain the activities of daily life dataset from the UC Irvine machine learning website (https://archive.ics.uci.edu/ml/datasets/Dataset+for+ADL+Recognition+with+Wrist-worn+Acceleron data provided by Barbara Bruno, Fulvio Mastrogiovanni and Antonio Sgorbissa).

(a) Build a classifier that classifies sequences into one of the 14 activities provided. To make features, you should vector quantize, then use a histogram of cluster centers (as described in the subsection; this gives a pretty explicit set of steps to follow). You will find it helpful to use hierarchical k-means to vector quantize. You may use whatever multi-class classifier you wish, though I'd start with R's decision forest, because it's easy to use and effective. You should report (a) the total error rate and (b) the class confusion matrix of your classifier.

(b) Now see if you can improve your classifier by (a) modifying the number of cluster centers in your hierarchical k-means and (b) modifying the size of the fixed length samples that you use.

CHAPTER 6

Clustering using Probability Models

6.1 THE MULTIVARIATE NORMAL DISTRIBUTION

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

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

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

where Σ is a positive definite matrix. Notice that if Σ is not positive definite, then we cannot have a probability distribution, because there are some directions $\mathbf{d}$ such that $\exp\left(-\frac{1}{2}(t\mathbf{d}-\mu)^T\Sigma^{-1}(t\mathbf{d}-\mu)\right)$ does not fall off to zero as t limits to infinity. In turn, this means we can't compute the integral, and so can't normalize.

The following facts explain the names of the parameters:

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

Assuming a multivariate normal distribution, we have

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

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

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

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

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

(which is rather a nuisance to show, because you need to know how to differentiate a determinant). You should be aware that this estimate is not guaranteed to be positive definite, even though the covariance matrix of a gaussian must be positive definite. We deal with this problem by checking the estimate. If its smallest eigenvalue is too close to zero, then we add some small positive constant times the identity to get a positive definite matrix.

6.1.1 Affine Transformations and Gaussians

Gaussians behave very well under affine transformations. In fact, we've already worked out all the math. Assume I have a dataset $\{\mathbf{x}\}$ with N data points $\mathbf{x}_i$, each a d-dimensional vector. The mean of the maximum likelihood gaussian model is $\mathsf{mean}\left(\{\mathbf{x}\}\right)$, and the covariance is $\mathsf{Covmat}\left(\{\mathbf{x}\}\right)$. We assume that this is positive definite, or adjust it as above.

I can now transform the data with an affine transformation, to get $\mathbf{y}_i = \mathcal{A}\mathbf{x}_i + \mathbf{b}$. We assume that $\mathcal{A}$ is a square matrix with full rank, so that this transformation is 1-1. The mean of the maximum likelihood gaussian model for the transformed dataset is $\mathsf{mean}\left(\{\mathbf{y}\}\right) = \mathcal{A}\mathsf{mean}\left(\{\mathbf{x}\}\right) + \mathbf{b}$. Similarly, the covariance is $\mathsf{Covmat}\left(\{\mathbf{y}\}\right) = \mathcal{A}\mathsf{Covmat}\left(\{\mathbf{x}\}\right)\mathcal{A}^T$.

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

6.1.2 Plotting a 2D Gaussian: Covariance Ellipses

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

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

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

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

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

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

Now rotating the ellipse is means we will pre- and post-multiply the covariance matrix with some rotation matrix. Translating it will move the origin to the mean. As a result, the ellipse has its center at the mean, its major axis is in the direction of the eigenvector of the covariance with largest eigenvalue, and its minor axis is in the direction of the eigenvector with smallest eigenvalue. A plot of this ellipse, which can be coaxed out of most programming environments with relatively little effort, gives us a great deal of information about the underlying Gaussian. These ellipses are known as **covariance ellipses**.

6.2 MIXTURE MODELS AND CLUSTERING

It is natural to think of clustering in the following way. The data was created by a collection of distinct models (one per cluster). For each data item, something (nature?) chose which model was to produce a point, and then the model produced a point. We see the results: crucially, we'd like to know what the models were, but we don't know which model produced which point. If we knew the models, it would be easy to decide which model produced which point. Similarly, if we knew which point went to which model, we could determine what the models were.

One encounters this situation – or problems that can be mapped to this situation – again and again. It is very deeply embedded in clustering problems. It is pretty clear that a natural algorithm is to iterate between estimating which model gets which point, and the model parameters. We have seen this approach before, in the case of k-means.

A particularly interesting case occurs when the models are probabilistic. There is a standard, and very important, algorithm for estimation here, called **EM** (or **expectation maximization**, if you want the long version). I will develop this algorithm in two simple cases, and we will see it in a more general form later.

Notation: This topic lends itself to a glorious festival of indices, limits of sums and products, etc. I will do one example in quite gory detail; the other follows the same form, and for that we'll proceed more expeditiously. Writing the

limits of sums or products explicitly is usually even more confusing than adopting a compact notation. When I write $\sum_i$ or $\prod_i$, I mean a sum (or product) over all values of i. When I write $\sum_{i,\hat{j}}$ or $\prod_{i,\hat{j}}$, I mean a sum (or product) over all values of i *except* for the j'th item. I will write vectors, as usual, as $\mathbf{x}$; the i'th such vector in a collection is $\mathbf{x}_i$, and the k'th component of the i'th vector in a collection is x_{ik}. In what follows, I will construct a vector δ_i corresponding to the i'th data item $\mathbf{x}_i$ (it will tell us what cluster that item belongs to). I will write δ to mean all the δ_i (one for each data item). The j'th component of this vector is δ_{ij}. When I write $\sum_{\delta_u}$, I mean a sum over all values that δ_u can take. When I write $\sum_{\delta}$, I mean a sum over all values that each δ can take. When I write $\sum_{\delta,\hat{\delta_v}}$, I mean a sum over all values that all δ can take, *omitting* all cases for the v'th vector δ_v.

6.2.1 A Finite Mixture of Blobs

A blob of data points is quite easily modelled with a single normal distribution. Obtaining the parameters is straightforward (estimate the mean and covariance matrix with the usual expressions). Now imagine I have t blobs of data, and I know t. A normal distribution is likely a poor model, but I could think of the data as being produced by t normal distributions. I will assume that each normal distribution has a fixed, *known* covariance matrix Σ, but the mean of each is unknown. Because the covariance matrix is fixed, and *known*, we can compute a factorization $\Sigma = \mathcal{A}\mathcal{A}^T$. The factors must have full rank, because the covariance matrix must be positive definite. This means that we can apply $\mathcal{A}^{-1}$ to all the data, so that each blob covariance matrix (and so each normal distribution) is the identity.

Write μ_j for the mean of the j'th normal distribution. We can model a distribution that consists of t distinct blobs by forming a weighted sum of the blobs, where the j'th blob gets weight π_j. We ensure that $\sum_j \pi_j = 1$, so that we can think of the overall model as a probability distribution. We can then model the data as samples from the probability distribution

$$p(\mathbf{x}|\mu_1, \ldots, \mu_k, \pi_1, \ldots, \pi_k) = \sum_j \pi_j \left[\frac{1}{\sqrt{(2\pi)^d}} \exp\left(-\frac{1}{2}(\mathbf{x} - \mu_j)^T(\mathbf{x} - \mu_j) \right) \right].$$

The way to think about this probability distribution is that a point is generated by first choosing one of the normal distributions (the j'th is chosen with probability π_j), then generating a point from that distribution. This is a pretty natural model of clustered data. Each mean is the center of a blob. Blobs with many points in them have a high value of π_j, and blobs with few points have a low value of π_j. We must now use the data points to estimate the values of π_j and μ_j (again, I am assuming that the blobs – and the normal distribution modelling each – have the identity as a covariance matrix). A distribution of this form is known as a **mixture of normal distributions**.

Writing out the likelihood will reveal a problem: we have a product of many sums. The usual trick of taking the log will not work, because then you have a sum of logs of sums, which is hard to differentiate and hard to work with. A much more productive approach is to think about a set of hidden variables which tell us which blob each data item comes from. For the i'th data item, we construct a vector δ_i. The j'th component of this vector is δ_{ij}, where $\delta_{ij} = 1$ if $\mathbf{x}_i$ comes from blob

(equivalently, normal distribution) j and zero otherwise. Notice there is exactly one 1 in δ_i, because each data item comes from one blob. I will write δ to mean all the δ_i (one for each data item). Assume we know the values of these terms. I will write $\theta = (\mu_1, \ldots, \mu_k, \pi_1, \ldots, \pi_k)$ for the unknown parameters. Then we can write

$$p(\mathbf{x}_i|\delta_i, \theta) = \prod_j \left[\frac{1}{\sqrt{(2\pi)^d}} \exp\left(-\frac{1}{2}(\mathbf{x}_i - \mu_j)^T(\mathbf{x}_i - \mu_j) \right) \right]^{\delta_{ij}}$$

(because $\delta_{ij} = 1$ means that $\mathbf{x}_i$ comes from blob j, so the terms in the product are a collection of 1's and the probability we want). We also have

$$p(\delta_{ij} = 1|\theta) = \pi_j$$

allowing us to write

$$p(\delta_i|\theta) = \prod_j [\pi_j]^{\delta_{ij}}$$

(because this is the probability that we select blob j to produce a data item; again, the terms in the product are a collection of 1's and the probability we want). This means that

$$p(\mathbf{x}_i, \delta_i|\theta) = \prod_j \left\{ \left[\frac{1}{\sqrt{(2\pi)^d}} \exp\left(-\frac{1}{2}(\mathbf{x}_i - \mu_j)^T(\mathbf{x}_i - \mu_j) \right) \right] \pi_j \right\}^{\delta_{ij}}$$

and we can write a log-likelihood. The data are the observed values of $\mathbf{x}$ and δ (remember, we pretend we know these; I'll fix this in a moment), and the parameters are the unknown values of $\mu_1, \ldots, \mu_k$ and $\pi_1, \ldots, \pi_k$. We have

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

where K is a constant that absorbs the normalizing constants for the normal distributions. You should check this expression gives the right answer. I have used the δ_{ij} as a "switch" – for one term, $\delta_{ij} = 1$ and the term in curly brackets is "on", and for all others that term is multiplied by zero. The problem with all this is that we don't know δ. I will deal with this when we have another example.

6.2.2 Topics and Topic Models

A real attraction of probabilistic clustering methods is that we can cluster data where there isn't a clear distance function. One example occurs in document processing. For many kinds of document, we obtain a good representation by (a) choosing a list of different words then (b) representing the document by a vector of word counts, where we simply ignore every word outside the list. 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. The problem is to cluster the documents.

It isn't a particularly good idea to cluster on the distance between word vectors. This is because quite small changes in word use might lead to large differences between count vectors. For example, some authors might write "car" when others write "auto". In turn, two documents might have a large (resp. small) count for "car" and a small (resp. large) count for "auto". Just looking at the counts would significantly overstate the difference between the vectors. However, the counts are informative: a document that uses the word "car" often, and the word "lipstick" seldom, is likely quite different from a document that uses "lipstick" often and "car" seldom.

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

We cluster documents together if they come from the same topic. Imagine we know which document comes from which topic. Then we could estimate the word probabilities using the documents in each topic. Now imagine we know the word probabilities for each topic. Then we could tell (at least in principle) which topic a document comes from by looking at the probability each topic generates the document, and choosing the topic with the highest probability. This should strike you as being a circular argument. It has a form you should recognize from k-means, though the details of the distance have changed.

To construct a probabilistic model, we will assume that a document is generated in two steps. We will have t topics. First, we choose a topic, choosing the j'th topic with probability π_j. Then we will obtain a set of words by repeatedly drawing IID samples from that topic, and record the count of each word in a count vector. Assume we have N vectors of word counts, and write $\mathbf{x}_i$ for the i'th such vector. Each topic is a multinomial probability distribution. Write $\mathbf{p}_j$ for the vector of word probabilities for the j'th topic. We assume that words are generated independently, conditioned on the topic. Write x_{ik} for the k'th component of $\mathbf{x}_i$, and so on. Notice that $\mathbf{x}_i^T\mathbf{1}$ is the sum of entries in $\mathbf{x}_i$, and so the number of words in document i. Then the probability of observing the counts in $\mathbf{x}_i$ when the document was generated by topic j is

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

We can now write the probability of observing a document. Again, we write $\theta =$

$(\mathbf{p}_1, \ldots, \mathbf{p}_t, \pi_1, \ldots, \pi_t)$ for the vector of unknown parameters. We have

$$\begin{aligned} p(\mathbf{x}_i|\theta) &= \sum_l p(\mathbf{x}_i|\text{topic is l})p(\text{topic is l}|\theta) \\ &= \sum_l \left[\left(\frac{(\mathbf{x}_i^T\mathbf{1})!}{\prod_v x_{iv}!} \right) \prod_u p_{lu}^{x_{iu}} \right] \pi_l. \end{aligned}$$

This model is widely called a **topic model**; be aware that there are many kinds of topic model, and this is a simple one. The expression should look unpromising, in a familiar way. If you write out a likelihood, you will see a product of sums; and if you write out a log-likelihood, you will see a sum of logs of sums. Neither is enticing. We could use the same trick we used for a mixture of normals. Write $\delta_{ij} = 1$ if $\mathbf{x}_i$ comes from topic j, and $\delta_{ij} = 0$ otherwise. Then we have

$$p(\mathbf{x}_i|\delta_{ij} = 1, \theta) = \left[\left(\frac{(\mathbf{x}_i^T\mathbf{1})!}{\prod_v x_{iv}!} \right) \prod_u p_{ju}^{x_{iu}} \right]$$

(because $\delta_{ij} = 1$ means that $\mathbf{x}_i$ comes from topic j). This means we can write

$$p(\mathbf{x}_i|\delta_i, \theta) = \prod_j \left\{ \left[\left(\frac{(\mathbf{x}_i^T\mathbf{1})!}{\prod_v x_{iv}!} \right) \prod_u p_{ju}^{x_{iu}} \right] \right\}^{\delta_{ij}}$$

(because $\delta_{ij} = 1$ means that $\mathbf{x}_i$ comes from topic j, so the terms in the product are a collection of 1's and the probability we want). We also have

$$p(\delta_{ij} = 1|\theta) = \pi_j$$

(because this is the probability that we select topic j to produce a data item), allowing us to write

$$p(\delta_i|\theta) = \prod_j [\pi_j]^{\delta_{ij}}$$

(again, the terms in the product are a collection of 1's and the probability we want). This means that

$$p(\mathbf{x}_i, \delta_i|\theta) = \prod_j \left[\left(\frac{(\mathbf{x}_i^T\mathbf{1})!}{\prod_v x_{iv}!} \right) \prod_u \left(p_{ju}^{x_{iu}}\right) \pi_j \right]^{\delta_{ij}}$$

and we can write a log-likelihood. The data are the observed values of $\mathbf{x}$ and δ (remember, we pretend we know these for the moment), and the parameters are the unknown values collected in θ. We have

$$\mathcal{L}(\theta; \mathbf{x}, \delta) = \sum_i \left\{ \sum_j \left[\sum_u x_{iu} \log p_{ju} + \log \pi_j \right] \delta_{ij} \right\} + K$$

where K is a term that contains all the

$$\log \left(\frac{(\mathbf{x}_i^T\mathbf{1})!}{\prod_v x_{iv}!} \right)$$

terms. This is of no interest to us, because it doesn't depend on any of our parameters. It takes a fixed value for each dataset. You should check this expression, noticing that, again, I have used the δ_{ij} as a "switch" – for one term, $\delta_{ij} = 1$ and the term in curly brackets is "on", and for all others that term is multiplied by zero. The problem with all this, as before, is that we don't know δ_{ij}. But there is a recipe.

6.3 THE EM ALGORITHM

There is a straightforward, natural, and very powerful recipe. In essence, we will average out the things we don't know. But this average will depend on our estimate of the parameters, so we will average, then re-estimate parameters, then re-average, and so on. If you lose track of what's going on here, think of the example of k-means with soft weights (section 9.9; this is what the equations for the case of a mixture of normals will boil down to). In this analogy, the δ tell us which cluster center a data item came from. Because we don't know the values of the δ, we assume we have a set of cluster centers; these allow us to make a soft estimate of the δ; then we use this estimate to re-estimate the centers; and so on.

This is an instance of a general recipe. Recall we wrote θ for a vector of parameters. In the mixture of normals case, θ contained the means and the mixing weights; in the topic model case, it contained the topic distributions and the mixing weights. Assume we have an estimate of the value of this vector, say $\theta^{(n)}$. We could then compute $p(\delta|\theta^{(n)}, \mathbf{x})$. In the mixture of normals case, this is a guide to which example goes to which cluster. In the topic case, it is a guide to which example goes to which topic.

We could use this to compute the expected value of the likelihood with respect to δ. We compute

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

(where the sum is over all values of δ). Notice that $Q(\theta; \theta^{(n)})$ is a *function* of θ (because $\mathcal{L}$ was), but now does not have any unknown δ terms in it. This $Q(\theta; \theta^{(n)})$ encodes what we know about δ.

For example, assume that $p(\delta|\theta^{(n)}, \mathbf{x})$ has a single, narrow peak in it, at (say) $\delta = \delta^0$. In the mixture of normals case, this would mean that there is one allocation of points to clusters that is significantly better than all others, given $\theta^{(n)}$. For this example, $Q(\theta; \theta^{(n)})$ will be approximately $\mathcal{L}(\theta; \mathbf{x}, \delta^0)$.

Now assume that $p(\delta|\theta^{(n)}, \mathbf{x})$ is about uniform. In the mixture of normals case, this would mean that any particular allocation of points to clusters is about as good as any other. For this example, $Q(\theta; \theta^{(n)})$ will average $\mathcal{L}$ over all possible δ values with about the same weight for each.

We obtain the next estimate of θ by computing

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

and iterate this procedure until it converges (which it does, though I shall not prove that). The algorithm I have described is extremely general and powerful, and is

known as **expectation maximization** or (more usually) **EM**. The step where we compute $Q(\theta; \theta^{(n)})$ is called the **E step**; the step where we compute the new estimate of θ is known as the **M step**.

One trick to be aware of: it is quite usual to ignore additive constants in the log-likelihood, because they have no effect. When you do the E-step, taking the expectation of a constant gets you a constant; in the M-step, the constant can't change the outcome. As a result, I will tend to be careless about it. In the mixture of normals example, below, I've tried to keep track of it; for the mixture of multinomials, I've ignored it.

6.3.1 Example: Mixture of Normals: The E-step

Now let us do the actual calculations for a mixture of normal distributions. The E step requires a little work. We have

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

If you look at this expression, it should strike you as deeply worrying. There are a very large number of different possible values of δ. In this case, there are $N \times t$ cases (there is one δ_i for each data item, and each of these can have a one in each of t locations). It isn't obvious how we could compute this average.

But notice

$$p(\delta | \theta^{(n)}, \mathbf{x}) = \frac{p(\delta, \mathbf{x} | \theta^{(n)})}{p(\mathbf{x} | \theta^{(n)})}$$

and let us deal with numerator and denominator separately. For the numerator, notice that the $\mathbf{x}_i$ and the δ_i are independent, identically distributed samples, so that

$$p(\delta, \mathbf{x} | \theta^{(n)}) = \prod_i p(\delta_i, \mathbf{x}_i | \theta^{(n)}).$$

The denominator is slightly more work. We have

$$\begin{aligned} p(\mathbf{x} | \theta^{(n)}) &= \sum_{\delta} p(\delta, \mathbf{x} | \theta^{(n)}) \\ &= \sum_{\delta} \left[\prod_i p(\delta_i, \mathbf{x}_i | \theta^{(n)}) \right] \\ &= \prod_i \left[\sum_{\delta_i} p(\delta_i, \mathbf{x}_i | \theta^{(n)}) \right]. \end{aligned}$$

You should check the last step; one natural thing to do is check with $N = 2$ and

$t = 2$. This means that we can write

$$\begin{aligned} p(\delta|\theta^{(n)}, \mathbf{x}) &= \frac{p(\delta, \mathbf{x}|\theta^{(n)})}{p(\mathbf{x}|\theta^{(n)})} \\ &= \frac{\prod_i p(\delta_i, \mathbf{x}_i|\theta^{(n)})}{\prod_i \left[\sum_{\delta_i} p(\delta_i, \mathbf{x}_i|\theta^{(n)})\right]} \\ &= \prod_i \frac{p(\delta_i, \mathbf{x}_i|\theta^{(n)})}{\sum_{\delta_i} p(\delta_i, \mathbf{x}_i|\theta^{(n)})} \\ &= \prod_i p(\delta_i|\mathbf{x}_i, \theta^{(n)}) \end{aligned}$$

Now we need to look at the log-likelihood. We have

$$\mathcal{L}(\theta; \mathbf{x}, \delta) = \sum_{ij} \left\{ \left[\left(-\frac{1}{2}(\mathbf{x}_i - \mu_j)^T (\mathbf{x}_i - \mu_j) \right) \right] + \log \pi_j \right\} \delta_{ij} + K.$$

The K term is of no interest – it will result in a constant – but we will try to keep track of it. To simplify the equations we need to write, I will construct a t dimensional vector $\mathbf{c}_i$ for the i'th data point. The j'th component of this vector will be

$$\left\{ \left[\left(-\frac{1}{2}(\mathbf{x}_i - \mu_j)^T (\mathbf{x}_i - \mu_j) \right) \right] + \log \pi_j \right\}$$

so we can write

$$\mathcal{L}(\theta; \mathbf{x}, \delta) = \sum_i \mathbf{c}_i^T \delta_i + K.$$

Now all this means that

$$\begin{aligned} \mathcal{Q}(\theta; \theta^{(n)}) &= \sum_{\delta} \mathcal{L}(\theta; \mathbf{x}, \delta) p(\delta|\theta^{(n)}, \mathbf{x}) \\ &= \sum_{\delta} \left(\sum_i \mathbf{c}_i^T \delta_i + K \right) p(\delta|\theta^{(n)}, \mathbf{x}) \\ &= \sum_{\delta} \left(\sum_i \mathbf{c}_i^T \delta_i + K \right) \prod_u p(\delta_u|\theta^{(n)}, \mathbf{x}) \\ &= \sum_{\delta} \left(\mathbf{c}_1^T \delta_1 \prod_u p(\delta_u|\theta^{(n)}, \mathbf{x}) + \ldots \mathbf{c}_N^T \delta_N \prod_u p(\delta_u|\theta^{(n)}, \mathbf{x}) \right). \end{aligned}$$

We can simplify further. We have that $\sum_{\delta_i} p(\delta_i|\mathbf{x}_i, \theta^{(n)}) = 1$, because this is a probability distribution. Notice that, for any index v,

$$\begin{aligned} \sum_{\delta} \left(\mathbf{c}_v^T \delta_v \prod_u p(\delta_u|\theta^{(n)}, \mathbf{x}) \right) &= \sum_{\delta_v} \left(\mathbf{c}_v^T \delta_v p(\delta_v|\theta^{(n)}, \mathbf{x}) \right) \left[\sum_{\delta,\ \hat{\delta_v}} \prod_{u,\hat{v}} p(\delta_u|\theta^{(n)}, \mathbf{x}) \right] \\ &= \sum_{\delta_v} \left(\mathbf{c}_v^T \delta_v p(\delta_v|\theta^{(n)}, \mathbf{x}) \right) \end{aligned}$$

So we can write

$$\begin{aligned}
\mathcal{Q}(\theta;\theta^{(n)}) &= \mathcal{L}(\theta;\mathbf{x},\delta)p(\delta|\theta^{(n)},\mathbf{x}) \\
&= \sum_i \left[\sum_{\delta_i} \mathbf{c}_i^T \delta_i p(\delta_i|\theta^{(n)},\mathbf{x})\right] + K \\
&= \sum_i \left[\left(\sum_j \left\{\left[\left(-\frac{1}{2}(\mathbf{x}_i-\mu_j)^T(\mathbf{x}_i-\mu_j)\right)+\log\pi_j\right] w_{ij}\right\}\right)\right] + K
\end{aligned}$$

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

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

6.3.2 Example: Mixture of Normals: The M-step

The M-step is more straightforward. Recall

$$\mathcal{Q}(\theta;\theta^{(n)}) = \left(\sum_{ij}\left\{\left[\left(-\frac{1}{2}(\mathbf{x}_i-\mu_j)^T(\mathbf{x}_i-\mu_j)\right)\right]+\log\pi_j\right\}w_{ij}+K\right)$$

and we have to maximise this with respect to μ and π, and the terms w_{ij} are known. This maximization is easy. We compute

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

and

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

You should check these expressions by differentiating and setting to zero. When you do so, remember that, because π is a probability distribution, $\sum_j \pi_j = 1$ (otherwise you'll get the wrong answer).

6.3.3 Example: Topic Model: The E-Step

We need to work out two steps. The E step requires a little calculation. We have

$$\begin{aligned}
\mathcal{Q}(\theta;\theta^{(n)}) &= \sum_{\delta} \mathcal{L}(\theta;\mathbf{x},\delta)p(\delta|\theta^{(n)},\mathbf{x}) \\
&= \sum_{\delta}\left(\sum_{ij}\left\{\left[\sum_u x_{iu}\log p_{ju}\right] + \log\pi_j\right\}\delta_{ij}\right)p(\delta|\theta^{(n)},\mathbf{x}) \\
&= \left(\sum_{ij}\left\{\left[\sum_k x_{i,k}\log p_{j,k}\right] + \log\pi_j\right\}w_{ij}\right)
\end{aligned}$$

Here the last two steps follow from the same considerations as in the mixture of normals. The $\mathbf{x}_i$ and δ_i are IID samples, and so the expectation simplifies as in that case. If you're uncertain, rewrite the steps of section 6.3.1. The form of this Q function is the same as that (a sum of $\mathbf{c}_i^T\delta_i$ terms, but using a different expression for $\mathbf{c}_i$). In this case, as above,

$$\begin{aligned}
w_{ij} &= 1p(\delta_{ij}=1|\theta^{(n)},\mathbf{x}) + 0p(\delta_{ij}=0|\theta^{(n)},\mathbf{x}) \\
&= p(\delta_{ij}=1|\theta^{(n)},\mathbf{x}).
\end{aligned}$$

Again, we have

$$\begin{aligned}
p(\delta_{ij}=1|\theta^{(n)},\mathbf{x}) &= \frac{p(\mathbf{x}_i,\delta_{ij}=1|\theta^{(n)})}{p(\mathbf{x}_i|\theta^{(n)})} \\
&= \frac{p(\mathbf{x}_i,\delta_{ij}=1|\theta^{(n)})}{\sum_l p(\mathbf{x}_i,\delta_{il}=1|\theta^{(n)})}
\end{aligned}$$

and so the main question is to obtain $p(\mathbf{x}_i,\delta_{ij}=1|\theta^{(n)})$. But

$$\begin{aligned}
p(\mathbf{x}_i,\delta_{ij}=1|\theta^{(n)}) &= p(\mathbf{x}_i|\delta_{ij}=1,\theta^{(n)})p(\delta_{ij}=1|\theta^{(n)}) \\
&= = \left[\prod_k p_{j,k}^{x_k}\right]\pi_j.
\end{aligned}$$

Substituting yields

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

6.3.4 Example: Topic Model: The M-step

The M-step is more straightforward. Recall

$$\mathcal{Q}(\theta; \theta^{(n)}) = \left(\sum_{ij} \left\{ \left[\sum_k x_{i,k} \log p_{j,k} \right] + \log \pi_j \right\} w_{ij} \right)$$

and we have to maximise this with respect to μ and π, and the terms w_{ij} are known. This maximization is easy, but remember that the probabilities sum to one, so you need either to use a Lagrange multiplier or to set one probability to $(1 -$ all others$)$. You should get

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

and

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

You should check these expressions by differentiating and setting to zero.

6.3.5 EM in Practice

The algorithm we have seen is amazingly powerful; I will use it again, ideally with less notation. One could reasonably ask whether it produces a "good" answer. Slightly surprisingly, the answer is yes. The algorithm produces a local minimum of $p(\mathbf{x}|\theta)$, the likelihood of the data conditioned on parameters. This is rather surprising because we engaged in all the activity with δ to avoid directly dealing with this likelihood (which in our cases was an unattractive product of sums). I did not prove this, but it's true anyway.

There are some practical issues. First, how many cluster centers should there be? Mostly, the answer is a practical one. We are usually clustering data for a reason (vector quantization is a really good reason), and then we search for a k that yields the best results.

Second, how should one start the iteration? This depends on the problem you want to solve, but for the two cases I have described, a rough clustering using k-means usually provides an excellent start. In the mixture of normals problem, you can take the cluster centers as initial values for the means, and the fraction of points in each cluster as initial values for the mixture weights. In the topic model problem, you can cluster the count vectors with k-means, use the overall counts within a cluster to get an initial estimate of the multinomial model probabilities, and use the fraction of documents within a cluster to get mixture weights. You need to be careful here, though. You really don't want to initialize a topic probability with a zero value for any word (otherwise no document containing that word can ever go into the cluster, which is a bit extreme). For our purposes, it will be enough

to allocate a small value to each zero count, then adjust all the word probabilies to be sure they sum to one. More complicated approaches are possible.

Third, we need to avoid numerical problems in the implementation. Notice that you will be evaluating terms that look like

$$\frac{\pi_k e^{-(\mathbf{x}_i-\mu_k)^T(\mathbf{x}_i-\mu_k)/2}}{\sum_u \pi_u e^{-(\mathbf{x}_i-\mu_u)^T(\mathbf{x}_i-\mu_u)/2}}.$$

Imagine you have a point that is far from all cluster means. If you just blithely exponentiate the negative distances, you could find yourself dividing zero by zero, or a tiny number by a tiny number. This can lead to trouble. There's an easy alternative. Find the center the point is closest to. Now subtract the square of this distance (d^2_{min} for concreteness) from all the distances. Then evaluate

$$\frac{\pi_k e^{-\left[(\mathbf{x}_i-\mu_k)^T(\mathbf{x}_i-\mu_k)-d^2_{\text{min}}\right]/2}}{\sum_u \pi_u e^{-\left[(\mathbf{x}_i-\mu_u)^T(\mathbf{x}_i-\mu_u)-d^2_{\text{min}}\right]/2}}$$

which is a better way of estimating the same number (notice the $e^{-d^2_{\text{min}}/2}$ terms cancel top and bottom).

The last problem is more substantial. EM will get me to a local minimum of $p(\mathbf{x}|\theta)$, but there might be more than one local minimum. For clustering problems, the usual case is there are lots of them. One doesn't really expect a clustering problem to have a single best solution, as opposed to a lot of quite good solutions. Points that are far from all clusters are a particular source of local minima; placing these points in different clusters yields somewhat different sets of cluster centers, each about as good as the other. It's not usual to worry much about this point. A natural strategy is to start the method in a variety of different places (use k means with different start points), and choose the one that has the best value of Q when it has converged.

However, EM isn't magic. There are problems where computing the expectation is hard, typically because you have to sum over a large number of cases which don't have the nice independence structure that helped in the examples I showed. There are strategies for dealing with this problem — essentially, you can get away with an approximate expectation – but they're beyond our reach at present.

6.4 YOU SHOULD

6.4.1 remember:

Useful facts: Parameters of a Multivariate Normal Distribution . . . 122
New term: covariance ellipses . . . 124
New term: EM . . . 124
New term: expectation maximization . . . 124
New term: mixture of normal distributions . . . 125
New term: topic model . . . 128
New term: expectation maximization . . . 130
New term: EM . . . 130
New term: E step . . . 130
New term: M step . . . 130

PROGRAMMING EXERCISES

6.1. Obtain the activities of daily life dataset from the UC Irvine machine learning website (https://archive.ics.uci.edu/ml/datasets/Dataset+for+ADL+Recognition+with+Wrist-worn+Acceleron data provided by Barbara Bruno, Fulvio Mastrogiovanni and Antonio Sgorbissa).

(a) Build a classifier that classifies sequences into one of the 14 activities provided. To make features, you should vector quantize, then use a histogram of cluster centers (as described in the subsection; this gives a pretty explicit set of steps to follow). You will find it helpful to use hierarchical k-means to vector quantize. You may use whatever multi-class classifier you wish, though I'd start with R's decision forest, because it's easy to use and effective. You should report (a) the total error rate and (b) the class confusion matrix of your classifier.

(b) Now see if you can improve your classifier by (a) modifying the number of cluster centers in your hierarchical k-means and (b) modifying the size of the fixed length samples that you use.

CHAPTER 7

Regression

Classification tries to predict a class from a data item. **Regression** tries to predict a value. For example, we know the zip code of a house, the square footage of its lot, the number of rooms and the square footage of the house, and we wish to predict its likely sale price. As another example, we know the cost and condition of a trading card for sale, and we wish to predict a likely profit in buying it and then reselling it. As yet another example, we have a picture with some missing pixels – perhaps there was text covering them, and we want to replace it – and we want to fill in the missing values. As a final example, you can think of classification as a special case of regression, where we want to predict either $+1$ or -1; this isn't usually the best way to proceed, however. Predicting values is very useful, and so there are many examples like this.

7.1 OVERVIEW

Some formalities are helpful here. In the simplest case, we have a dataset consisting of a set of N pairs $(\mathbf{x}_i, y_i)$. We think of y_i as the value of some function evaluated at $\mathbf{x}_i$, but with some random component. This means there might be two data items where the $\mathbf{x}_i$ are the same, and the y_i are different. We refer to the $\mathbf{x}_i$ as **explanatory variables** and the y_i is a **dependent variable**. We regularly say that we are regressing the dependent variable against the explanatory variables. We want to use the examples we have — the **training examples** — to build a model of the dependence between y and $\mathbf{x}$. This model will be used to predict values of y for new values of $\mathbf{x}$, which are usually called **test examples**. By far the most important model has the form $y = \mathbf{x}^T\beta + \xi$, where β are some set of parameters we need to choose and ξ are random effects. Now imagine that we have one independent variable. An appropriate choice of $\mathbf{x}$ (details below) will mean that the predictions made by this model will lie on a straight line. Figure 7.1 shows two regressions. The data are plotted with a scatter plot, and the line gives the prediction of the model for each value on the x axis.

We do not guarantee that different values of $\mathbf{x}$ produce different values of y. Data just isn't like this (see the crickets example Figure 7.1). Traditionally, regression produces some representation of a probability distribution for y conditioned on $\mathbf{x}$, so that we would get (say) some representation of a distribution on the houses likely sale value. The best prediction would then be the expected value of that distribution.

It should be clear that none of this will work if there is not some relationship between the training examples and the test examples. If I collect training data on the height and weight of children, I'm unlikely to get good predictions of the weight of adults from their height. We can be more precise with a probabilistic framework. We think of $\mathbf{x}_i$ as IID samples from some (usually unknown) probability distribution $P(X)$. Then the test examples should also be IID samples from $P(X)$,

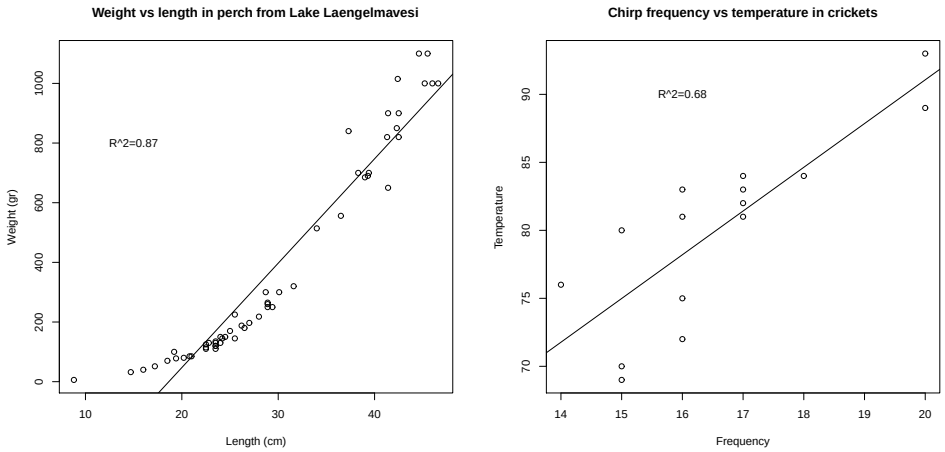

FIGURE 7.1: *On the* **left**, *a regression of weight against length for perch from a Finnish lake (you can find this dataset, and the back story at* http://www.amstat.org/publications/jse/jse_data_archive.htm*; look for "fishcatch" on that page). Notice that the linear regression fits the data fairly well, meaning that you should be able to predict the weight of a perch from its length fairly well. On the* **right**, *a regression of air temperature against chirp frequency for crickets. The data is fairly close to the line, meaning that you should be able to tell the temperature from the pitch of cricket's chirp fairly well. This data is from* http://mste.illinois.edu/patel/amar430/keyprob1.html*. The* R^2 *you see on each figure is a measure of the goodness of fit of the regression (section 7.2.4).*

or, at least, rather like them – you usually can't check this point with any certainty. A probabilistic formalism can help be precise about the y_i, too. Assume another random variable Y has joint distribution with X given by $P(Y, X)$. We think of each y_i as a sample from $P(Y|\{X = \mathbf{x}_i\})$. Then our modelling problem would be: given the training data, build a model that takes a test example $\mathbf{x}$ and yields a model of $P(Y|\{X = \mathbf{x}_i\})$.

Thinking about the problem this way should make it clear that we're not relying on any exact, physical, or causal relationship between Y and X. It's enough that their joint probability makes useful predictions possible, something we will test by experiment. This means that you can build regressions that work in somewhat surprising circumstances. For example, regressing childrens' reading ability against their foot size can be quite successful. This isn't because having big feet somehow helps you read; it's because on the whole, older children read better, and also have bigger feet.

To do anything useful with this formalism requires some aggressive simplifying assumptions. There are very few circumstances that require a comprehensive representation of $P(Y|\{X = \mathbf{x}_i\})$. Usually, we are interested in $\mathbb{E}[Y|\{X = \mathbf{x}_i\}]$ (the mean of $P(Y|\{X = \mathbf{x}_i\})$) and in $\mathsf{var}(\{P(Y|\{X = \mathbf{x}_i\})\})$. To recover this representation, we assume that, for any pair of examples $(\mathbf{x}, y)$, the value of y is obtained

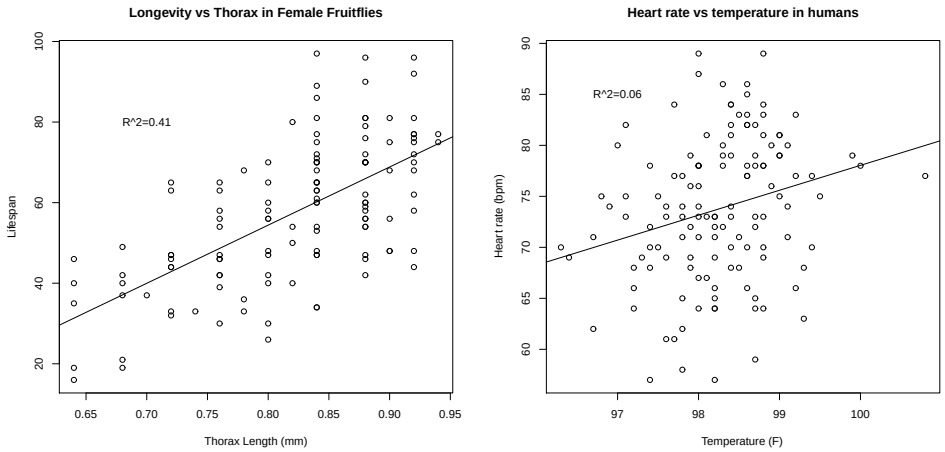

FIGURE 7.2: *Regressions do not necessarily yield good predictions or good model fits. On the* **left**, *a regression of the lifespan of female fruitflies against the length of their torso as adults (apparently, this doesn't change as a fruitfly ages; you can find this dataset, and the back story at http://www.amstat.org/publications/jse/jse_data_archive.htm; look for "fruitfly" on that page). The figure suggests you can make some prediction of how long your fruitfly will last by measuring its torso, but not a particularly accurate one. On the* **right**, *a regression of heart rate against body temperature for adults. You can find the data at http://www.amstat.org/publications/jse/jse_data_archive.htm as well; look for "temperature" on that page. Notice that predicting heart rate from body temperature isn't going to work that well, either.*

by applying some (unknown) function f to $\mathbf{x}$, then adding some random variable ξ with zero mean. We can write $y(\mathbf{x}) = f(\mathbf{x}) + \xi$, though it's worth remembering that there can be many different values of y associated with a single $\mathbf{x}$. Now we must make some estimate of f — which yields $\mathbb{E}[Y|\{X = \mathbf{x}_i\}]$ — and estimate the variance of ξ. The variance of ξ might be constant, or might vary with $\mathbf{x}$.

7.1.1 Regression to Spot Trends

Regression isn't only used to predict values. Another reason to build a regression model is to compare trends in data. Doing so can make it clear what is really happening. Here is an example from Efron ("Computer-Intensive methods in statistical regression", B. Efron, SIAM Review, 1988). The table in the appendix shows some data from medical devices, which sit in the body and release a hormone. The data shows the amount of hormone currently in a device after it has spent some time in service, and the time the device spent in service. The data describes devices from three production lots (A, B, and C). Each device, from each lot, is supposed to have the same behavior. The important question is: Are the lots the same? The amount of hormone changes over time, so we can't just compare the amounts currently in each device. Instead, we need to determine the relationship between time in service

and hormone, and see if this relationship is different between batches. We can do so by regressing hormone against time.

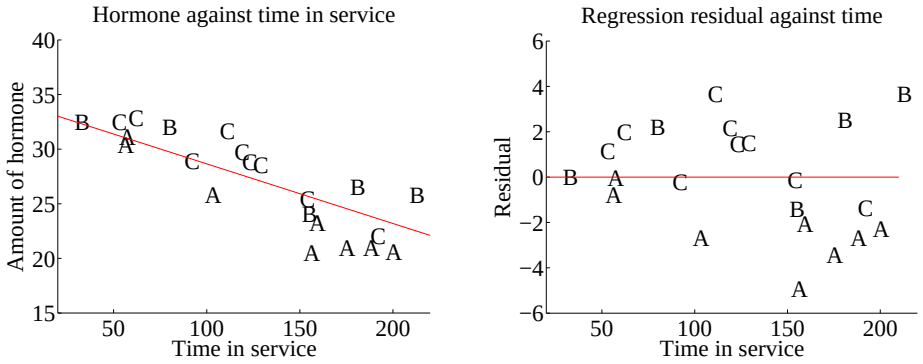

FIGURE 7.3: *On the* **left**, *a scatter plot of hormone against time for devices from tables 8.1 and 8.1. Notice that there is a pretty clear relationship between time and amount of hormone (the longer the device has been in service the less hormone there is). The issue now is to understand that relationship so that we can tell whether lots A, B and C are the same or different. The best fit line to all the data is shown as well, fitted using the methods of section 7.2. On the* **right**, *a scatter plot of residual — the distance between each data point and the best fit line — against time for the devices from tables 8.1 and 8.1. Now you should notice a clear difference; some devices from lots B and C have positive and some negative residuals, but all lot A devices have negative residuals. This means that, when we account for loss of hormone over time, lot A devices still have less hormone in them. This is pretty good evidence that there is a problem with this lot.*

Figure 7.3 shows how a regression can help. In this case, we have modelled the amount of hormone in the device as

$$a \times (\text{time in service}) + b$$

for a, b chosen to get the best fit (much more on this point later!). This means we can plot each data point on a scatter plot, together with the best fitting line. This plot allows us to ask whether any particular batch behaves differently from the overall model in any interesting way.

However, it is hard to evaluate the distances between data points and the best fitting line by eye. A sensible alternative is to subtract the amount of hormone predicted by the model from the amount that was measured. Doing so yields a **residual** — the difference between a measurement and a prediction. We can then plot those residuals (Figure 7.3). In this case, the plot suggests that lot A is special — all devices from this lot contain less hormone than our model predicts.

Definition: 7.1 *Regression*

Regression accepts a feature vector and produces a prediction, which is usually a number, but can sometimes have other forms. You can use these predictions as predictions, or to study trends in data. It is possible, but not usually particularly helpful, to see classification as a form of regression.

7.2 LINEAR REGRESSION AND LEAST SQUARES

Assume we have a dataset consisting of a set of N pairs $(\mathbf{x}_i, y_i)$. We think of y_i as the value of some function evaluated at $\mathbf{x}_i$, with some random component added. This means there might be two data items where the $\mathbf{x}_i$ are the same, and the y_i are different. We refer to the $\mathbf{x}_i$ as **explanatory variables** and the y_i is a **dependent variable**. We want to use the examples we have — the **training examples** — to build a model of the dependence between y and $\mathbf{x}$. This model will be used to predict values of y for new values of $\mathbf{x}$, which are usually called **test examples**. It can also be used to understand the relationships between the $\mathbf{x}$. The model needs to have some probabilistic component; we do not expect that y is a function of $\mathbf{x}$, and there is likely some error in evaluating y anyhow.

7.2.1 Linear Regression

We cannot expect that our model makes perfect predictions. Furthermore, y may not be a function of $\mathbf{x}$ — it is quite possible that the same value of $\mathbf{x}$ could lead to different y's. One way that this could occur is that y is a measurement (and so subject to some measurement noise). Another is that there is some randomness in y. For example, we expect that two houses with the same set of features (the $\mathbf{x}$) might still sell for different prices (the y's).

A good, simple model is to assume that the dependent variable (i.e. y) is obtained by evaluating a linear function of the explanatory variables (i.e. $\mathbf{x}$), then adding a zero-mean normal random variable. We can write this model as

$$y = \mathbf{x}^T\beta + \xi$$

where ξ represents random (or at least, unmodelled) effects. We will always assume that ξ has zero mean. In this expression, β is a vector of weights, which we must estimate. When we use this model to predict a value of y for a particular set of explanatory variables $\mathbf{x}^*$, we cannot predict the value that ξ will take. Our best available prediction is the mean value (which is zero). Notice that if $\mathbf{x} = 0$, the model predicts $y = 0$. This may seem like a problem to you — you might be concerned that we can fit only lines through the origin — but remember that $\mathbf{x}$ contains explanatory variables, and we can choose what appears in $\mathbf{x}$. The two

examples show how a sensible choice of $\mathbf{x}$ allows us to fit a line with an arbitrary y-intercept.

Definition: 7.2 *Linear regression*

A linear regression takes the feature vector $\mathbf{x}$ and predicts $\mathbf{x}^T\beta$, for some vector of coefficients β. The coefficients are adjusted, using data, to produce the best predictions.

Example: 7.1 *A linear model fitted to a single explanatory variable*

Assume we fit a linear model to a single explanatory variable. Then the model has the form $y = x\beta + \xi$, where ξ is a zero mean random variable. For any value x^* of the explanatory variable, our best estimate of y is βx^*. In particular, if $x^* = 0$, the model predicts $y = 0$, which is unfortunate. We can draw the model by drawing a line through the origin with slope β in the x, y plane. The y-intercept of this line must be zero.

Example: 7.2 *A linear model with a non-zero y-intercept*

Assume we have a single explanatory variable, which we write u. We can then create a *vector* $\mathbf{x} = [u, 1]^T$ from the explanatory variable. We now fit a linear model to this vector. Then the model has the form $y = \mathbf{x}^T\beta + \xi$, where ξ is a zero mean random variable. For any value $\mathbf{x}^* = [u^*, 1]^T$ of the explanatory variable, our best estimate of y is $(\mathbf{x}^*)^T\beta$, which can be written as $y = \beta_1 u^* + \beta_2$. If $x^* = 0$, the model predicts $y = \beta_2$. We can draw the model by drawing a line through the origin with slope β_1 and y-intercept β_2 in the x, y plane.

7.2.2 Choosing β

We must determine β. We can proceed in two ways. I show both because different people find different lines of reasoning more compelling. Each will get us to the same solution. One is probabilistic, the other isn't. Generally, I'll proceed as if they're interchangeable, although at least in principle they're different.

Probabilistic approach: we could assume that ξ is a zero mean normal random variable with unknown variance. Then $P(y|x, \beta)$ is normal, with mean $\mathbf{x}^T\beta$, and so we can write out the log-likelihood of the data. Write σ^2 for the variance of ξ, which we don't know, but will not worry about right now. We have that

$$\begin{aligned}\log \mathcal{L}(\beta) &= \sum_i \log P(y_i|\mathbf{x}_i, \beta) \\ &= \frac{1}{2\sigma^2}\sum_i (y_i - \mathbf{x}_i^T\beta)^2 + \text{term not depending on } \beta\end{aligned}$$

Maximizing the log-likelihood of the data is equivalent to minimizing the negative log-likelihood of the data. Furthermore, the term $\frac{1}{2\sigma^2}$ does not affect the location of the minimum, so we must have that β minimizes $\sum_i (y_i - \mathbf{x}_i^T\beta)^2$, or anything proportional to it. It is helpful to minimize an expression that is an average of squared errors, because (hopefully) this doesn't grow much when we add data. We therefore minimize

$$\left(\frac{1}{N}\right)\left(\sum_i (y_i - \mathbf{x}_i^T\beta)^2\right).$$

Direct approach: notice that, if we have an estimate of β, we have an estimate of the values of the unmodelled effects ξ_i for each example. We just take $\xi_i = y_i - \mathbf{x}_i^T\beta$. It is quite natural to make the unmodelled effects "small". A good measure of size is the mean of the squared values, which means we want to minimize

$$\left(\frac{1}{N}\right)\left(\sum_i (y_i - \mathbf{x}_i^T\beta)^2\right).$$

We can write all this more conveniently using vectors and matrices. Write $\mathbf{y}$ for the vector

$$\begin{pmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{pmatrix}$$

and $\mathcal{X}$ for the matrix

$$\begin{pmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \dots \mathbf{x}_n^T \end{pmatrix}.$$

Then we want to minimize

$$\left(\frac{1}{N}\right)(\mathbf{y} - \mathcal{X}\beta)^T(\mathbf{y} - \mathcal{X}\beta)$$

which means that we must have

$$\mathcal{X}^T\mathcal{X}\beta - \mathcal{X}^T\mathbf{y} = 0.$$

For reasonable choices of features, we could expect that $\mathcal{X}^T\mathcal{X}$ — which should strike you as being a lot like a covariance matrix — has full rank. If it does, which

Listing 7.1: R code used for the linear regression example of worked example 7.1

```
efd<-read.table('efrontable.txt',header=TRUE)
# the table has the form
#N1   Ah   Bh   Ch N2  At   Bt   Ct
# now we need to construct a new dataset
hor<-stack(efd, select=2:4)
tim<-stack(efd, select=6:8)
foo<-data.frame(time=tim[, c("values")],
                hormone=hor[, c("values")])
foo.lm<-lm(hormone~time,data=foo)
plot(foo)
abline(foo.lm)
```

is the usual case, this equation is easy to solve. If it does not, there is more to do, which we will do in section 7.4.2.

Remember this: *The vector of coefficients β for a linear regression is usually estimated using a least-squares procedure.*

Worked example 7.1 *Simple Linear Regression with R*

Regress the hormone data against time for all the devices in the Efron example.

Solution: This example is mainly used to demonstrate how to regress in R. There is sample code in listing 7.1. The summary in the listing produces a great deal of information (try it). Most of it won't mean anything to you yet. You can get a figure by doing `plot(foo.lm)`, but these figures will not mean anything yet, either. In the code, I've shown how to plot the data and a line on top of it.

7.2.3 Residuals

Assume we have produced a regression by solving

$$\mathcal{X}^T\mathcal{X}\hat{\beta} - \mathcal{X}^T\mathbf{y} = 0$$

for the value of $\hat{\beta}$. I write $\hat{\beta}$ because this is an *estimate*; we likely don't have the true value of the β that generated the data (the model might be wrong; etc.). We cannot expect that $\mathcal{X}\hat{\beta}$ is the same as $\mathbf{y}$. Instead, there is likely to be some error. The **residual** is the vector

$$\mathbf{e} = \mathbf{y} - \mathcal{X}\hat{\beta}$$

which gives the difference between the true value and the model's prediction at each point. Each component of the residual is an estimate of the unmodelled effects for that data point. The **mean square error** is

$$m = \frac{\mathbf{e}^T\mathbf{e}}{N}$$

and this gives the average of the squared error of prediction on the training examples.

Notice that the mean squared error is not a great measure of how good the regression is. This is because the value depends on the units in which the dependent variable is measured. So, for example, if you measure y in meters you will get a different mean squared error than if you measure y in kilometers.

7.2.4 R-squared

There is an important quantitative measure of how good a regression is which doesn't depend on units. Unless the dependent variable is a constant (which would make prediction easy), it has some variance. If our model is of any use, it should explain some aspects of the value of the dependent variable. This means that the variance of the residual should be smaller than the variance of the dependent variable. If the model made perfect predictions, then the variance of the residual should be zero.

We can formalize all this in a relatively straightforward way. We will ensure that $\mathcal{X}$ always has a column of ones in it, so that the regression can have a non-zero y-intercept. We now fit a model

$$\mathbf{y} = \mathcal{X}\beta + \mathbf{e}$$

(where $\mathbf{e}$ is the vector of residual values) by choosing β such that $\mathbf{e}^T\mathbf{e}$ is minimized. Then we get some useful technical results.

Useful Facts: 7.1 *Regression*

We write $\mathbf{y} = \mathcal{X}\hat{\beta} + \mathbf{e}$, where $\mathbf{e}$ is the residual. Assume $\mathcal{X}$ has a column of ones, and $\hat{\beta}$ is chosen to minimize $\mathbf{e}^T\mathbf{e}$. Then we have

1. $\mathbf{e}^T\mathcal{X} = \mathbf{0}$, i.e. that $\mathbf{e}$ is orthogonal to any column of $\mathcal{X}$. This is because, if $\mathbf{e}$ is not orthogonal to some column of $\mathbf{e}$, we can increase or decrease the $\hat{\beta}$ term corresponding to that column to make the error smaller. Another way to see this is to notice that $\hat{\beta}$ is chosen to minimize $\frac{1}{N}\mathbf{e}^T\mathbf{e}$, which is $\frac{1}{N}(\mathbf{y} - \mathcal{X}\hat{\beta})^T(\mathbf{y} - \mathcal{X}\hat{\beta})$. Now because this is a minimum, the gradient with respect to $\hat{\beta}$ is zero, so $(\mathbf{y} - \mathcal{X}\hat{\beta})^T(-\mathcal{X}) = -\mathbf{e}^T\mathcal{X} = 0$.
2. $\mathbf{e}^T\mathbf{1} = 0$ (recall that $\mathcal{X}$ has a column of all ones, and apply the previous result).
3. $\mathbf{1}^T(\mathbf{y} - \mathcal{X}\hat{\beta}) = 0$ (same as previous result).
4. $\mathbf{e}^T\mathcal{X}\hat{\beta} = 0$ (first result means that this is true).

Now $\mathbf{y}$ is a one dimensional dataset arranged into a vector, so we can compute $\mathsf{mean}\,(\{y\})$ and $\mathsf{var}[y]$. Similarly, $\mathcal{X}\hat{\beta}$ is a one dimensional dataset arranged into a vector (its elements are $\mathbf{x}_i^T\hat{\beta}$), as is $\mathbf{e}$, so we know the meaning of mean and variance for each. We have a particularly important result:

$$\mathsf{var}[y] = \mathsf{var}\left[\mathcal{X}\hat{\beta}\right] + \mathsf{var}[e].$$

This is quite easy to show, with a little more notation. Write $\overline{\mathbf{y}} = (1/N)(\mathbf{1}^T\mathbf{y})\mathbf{1}$ for the vector whose entries are all $\mathsf{mean}\,(\{y\})$; similarly for $\overline{\mathbf{e}}$ and for $\overline{\mathcal{X}\hat{\beta}}$. We have

$$\mathsf{var}[y] = (1/N)(\mathbf{y} - \overline{\mathbf{y}})^T(\mathbf{y} - \overline{\mathbf{y}})$$

and so on for $\mathsf{var}[e_i]$, etc. Notice from the facts that $\overline{\mathbf{y}} = \overline{\mathcal{X}\hat{\beta}}$. Now

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

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 7.3 has an R^2 value of 0.87. Figures 7.1 and 7.2 show the R^2 values for the regressions plotted there; notice how better models yield larger values of R^2. Notice that if you look at the summary that R provides for a linear regression, it will offer you *two* estimates of the value for R^2. These estimates are obtained in ways that try to account for (a) the amount of data in the regression, and (b) the number of variables in the regression. For our purposes, the differences between these numbers and the R^2 I defined are not significant. For the figures, I computed R^2 as I described in the text above, but if you substitute one of R's numbers nothing terrible will happen.

Remember this: *The quality of predictions made by a regression can be evaluated by looking at the fraction of the variance in the dependent variable that is explained by the regression. This number is called R^2, and lies between zero and one; regressions with larger values make better predictions.*

7.2.5 Transforming Variables

Sometimes the data isn't in a form that leads to a good linear regression. In this case, transforming explanatory variables, the dependent variable, or both can lead to big improvements. Figure 7.4 shows one example, based on the idea of word frequencies. Some words are used very often in text; most are used seldom. The dataset for this figure consists of counts of the number of time a word occurred for the 100 most common words in Shakespeare's printed works. It was originally collected from a concordance, and has been used to attack a variety of interesting questions, including an attempt to assess how many words Shakespeare knew. This is hard, because he likely knew many words that he didn't use in his works, so one can't just count. If you look at the plot of Figure 7.4, you can see that a linear regression of count (the number of times a word is used) against rank (how common a word is, 1-100) is not really useful. The most common words are used very often, and the number of times a word is used falls off very sharply as one looks at less common words. You can see this effect in the scatter plot of residual against dependent variable in Figure 7.4 — the residual depends rather strongly on the dependent variable. This is an extreme example that illustrates how poor linear regressions can be.

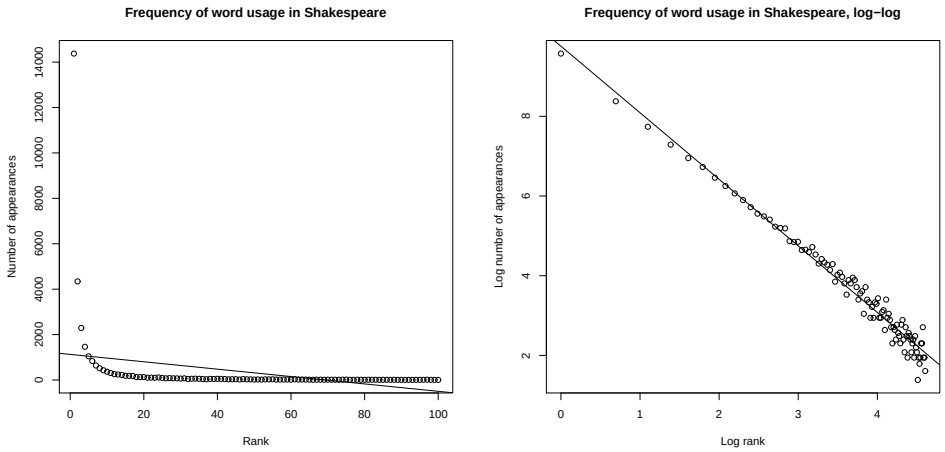

FIGURE 7.4: *On the* **left**, *word count plotted against rank for the 100 most common words in Shakespeare, using a dataset that comes with R (called "bard", and quite likely originating in an unpublished report by J. Gani and I. Saunders). I show a regression line too. This is a poor fit by eye, and the* R^2 *is poor, too (*$R^2 = 0.1$*). On the* **right**, *log word count plotted against log rank for the 100 most common words in Shakespeare, using a dataset that comes with R (called "bard", and quite likely originating in an unpublished report by J. Gani and I. Saunders). The regression line is very close to the data.*

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*

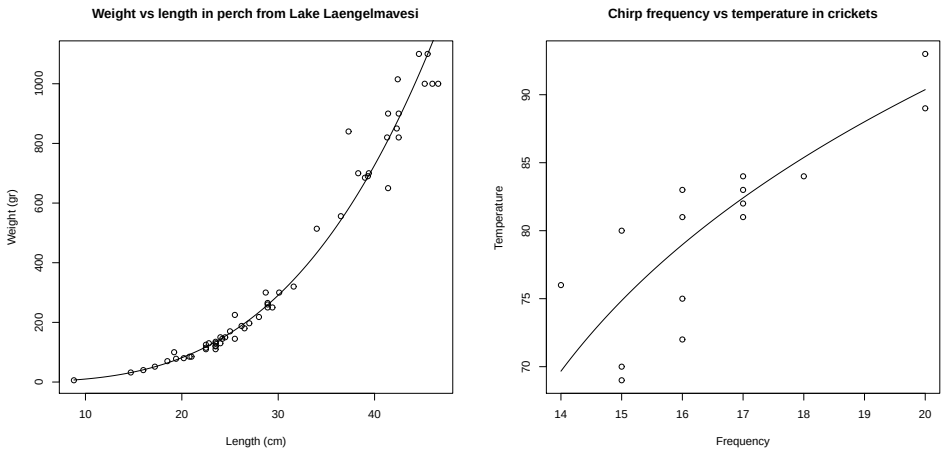

FIGURE 7.5: *The Box-Cox transformation suggests a value of* $\lambda = 0.303$ *for the regression of weight against height for the perch data of Figure 7.1. You can find this dataset, and the back story at* http://www.amstat.org/publications/jse/jse_data_archive.htm*; look for "fishcatch" on that page). On the* **left**, *a plot of the resulting curve overlaid on the data. For the cricket temperature data of that figure (from* http://mste.illinois.edu/patel/amar430/keyprob1.html*), the transformation suggests a value of* $\lambda = 4.75$. *On the* **right**, *a plot of the resulting curve overlaid on the data.*

The **Box-Cox transformation** is a method that can search for a transformation of the dependent variable that improves the regression. The method uses a one-parameter family of transformations, with parameter λ, then searches for the best value of this parameter using maximum likelihood. A clever choice of transformation means that this search is relatively straightforward. We define the Box-Cox transformation of the dependent variable to be

$$y_i^{(bc)} = \begin{cases} \frac{y_i^\lambda - 1}{\lambda} & \text{if } \lambda \neq 0 \\ \log y_i & \text{if } \lambda = 0 \end{cases} .$$

It turns out to be straightforward to estimate a good value of λ using maximum likelihood. One searches for a value of λ that makes residuals look most like a normal distribution. Statistical software will do it for you; the exercises sketch out the method. This transformation can produce significant improvements in a

regression. For example, the transformation suggests a value of $\lambda = 0.303$ for the fish example of Figure 7.1. It isn't natural to plot weight$^{0.303}$ against height, because we don't really want to predict weight$^{0.303}$. Instead, we plot the predictions of weight that come from this model, which will lie on a curve with the form $(ax + b)^{\frac{1}{0.303}}$, rather than on a straight line. Similarly, the transformation suggests a value of $\lambda = 0.475$ for the cricket data. Figure 7.5 shows the result of these transforms.

7.2.6 Can you Trust Your Regression?

Linear regression is useful, but it isn't magic. Some regressions make poor predictions (recall the regressions of figure 7.2). As another example, regressing the first digit of your telephone number against the length of your foot won't work.

We have some straightforward tests to tell whether a regression is working. You can **look at a plot** for a dataset with one explanatory variable and one dependent variable. You plot the data on a scatter plot, then plot the model as a line on that scatterplot. Just looking at the picture can be informative (compare Figure 7.1 and Figure 7.2).

You can check if the regression **predicts a constant**. This is usually a bad sign. You can check this by looking at the predictions for each of the training data items. If the variance of these predictions is small compared to the variance of the independent variable, the regression isn't working well. If you have only one explanatory variable, then you can plot the regression line. If the line is horizontal, or close, then the value of the explanatory variable makes very little contribution to the prediction. This suggests that there is no particular relationship between the explanatory variable and the independent variable.

You can also check, by eye, if **the residual isn't random**. If $y - \mathbf{x}^T\beta$ is a zero mean normal random variable, then the value of the residual vector should not depend on the corresponding y-value. Similarly, if $y - \mathbf{x}^T\beta$ is just a zero mean collection of unmodelled effects, we want the value of the residual vector to not depend on the corresponding y-value either. If it does, that means there is some phenomenon we are not modelling. Looking at a scatter plot of $\mathbf{e}$ against $\mathbf{y}$ will often reveal trouble in a regression (Figure 7.7). In the case of Figure 7.7, the trouble is caused by a few data points that are very different from the others severely affecting the regression. We will discuss how to identify and deal with such points in Section **??**. Once they have been removed, the regression improves markedly (Figure 7.8).

Remember this: *Linear regressions can make bad predictions. You can check for trouble by: evaluating R^2; looking at a plot; looking to see if the regression makes a constant prediction; or checking whether the residual is random. Other strategies exist, but are beyond the scope of this book.*

Procedure: 7.1 *Linear Regression using Least Squares*

We have a dataset containing N pairs $(\mathbf{x}_i, y_i)$. Each x_i is a d-dimensional explanatory vector, and each y_i is a single dependent variable. We assume that each data point conforms to the model

$$y_i = \mathbf{x}_i^T \beta + \xi_i$$

where ξ_i represents unmodelled effects. We assume that ξ_i are samples of a random variable with 0 mean and unknown variance. Sometimes, we assume the random variable is normal. Write

$$\mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{pmatrix}$$

and

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

We estimate $\hat{\beta}$ (the value of β) by solving the linear system

$$\mathcal{X}^T \mathcal{X} \hat{\beta} - \mathcal{X}^T \mathbf{y} = 0.$$

For a data point $\mathbf{x}$, our model predicts $\mathbf{x}^T \hat{\beta}$. The residuals are

$$\mathbf{e} = \mathbf{y} - \mathcal{X}\hat{\beta}.$$

We have that $\mathbf{e}^T \mathbf{1} = 0$. The mean square error is given by

$$m = \frac{\mathbf{e}^T \mathbf{e}}{N}.$$

The R^2 is given by

$$\frac{\mathsf{var}\left(\left\{\mathbf{x}_i^T \hat{\beta}\right\}\right)}{\mathsf{var}\left(\{\mathbf{y}\}\right)}.$$

Values of R^2 range from 0 to 1; a larger value means the regression is better at explaining the data.

7.3 PROBLEM DATA POINTS

I have described regressions on a single explanatory variable, because it is easy to plot the line in this case. You can find most problems by looking at the line and the data points. But a single explanatory variable isn't the most common or useful

case. If we have many explanatory variables, it can be hard to plot the regression in a way that exposes problems. This section mainly describes methods to identify and solve difficulties that don't involve looking at the line.

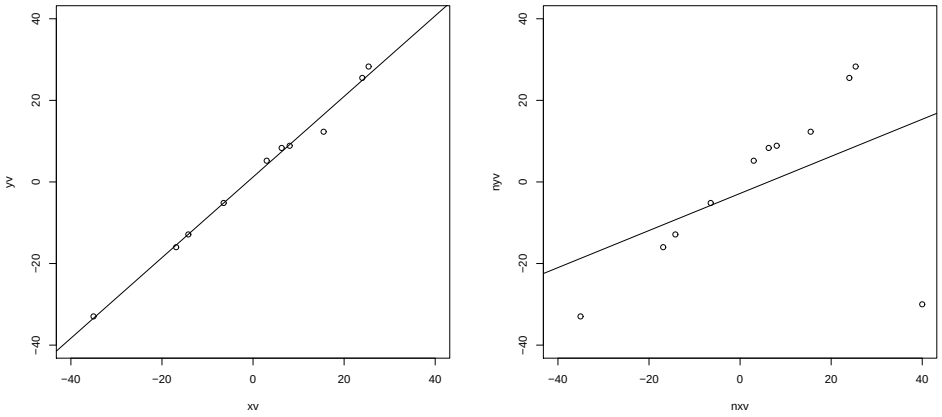

FIGURE 7.6: *On the* **left**, *a synthetic dataset with one independent and one explanatory variable, with the regression line plotted. Notice the line is close to the data points, and its predictions seem likely to be reliable. On the* **right**, *the result of adding a single outlying datapoint to that dataset. The regression line has changed significantly, because the regression line tries to minimize the sum of squared vertical distances between the data points and the line. Because the outlying datapoint is far from the line, the squared vertical distance to this point is enormous. The line has moved to reduce this distance, at the cost of making the other points further from the line.*

7.3.1 Problem Data Points have Significant Impact

Outlying data points can significantly weaken the usefulness of a regression. For some regression problems, we can identify data points that might be a problem, and then resolve how to deal with them. One possibility is that they are true outliers — someone recorded a data item wrong, or they represent an effect that just doesn't occur all that often. Another is that they are important data, and our linear model may not be good enough. If the data points really are outliers, we can drop them from the data set. If they aren't, we may be able to improve the regression by transforming features or by finding a new explanatory variable.

When we construct a regression, we are solving for the β that minimizes $\sum_i (y_i - \mathbf{x}_i^T \beta)^2$, equivalently for the β that produces the smallest value of $\sum_i e_i^2$. This means that residuals with large value can have a very strong influence on the outcome — we are squaring that large value, resulting in an enormous value. Generally, many residuals of medium size will have a smaller cost than one large residual and the rest tiny. As figure 7.6 illustrates, this means that a data point that lies far from the others can swing the regression line significantly.

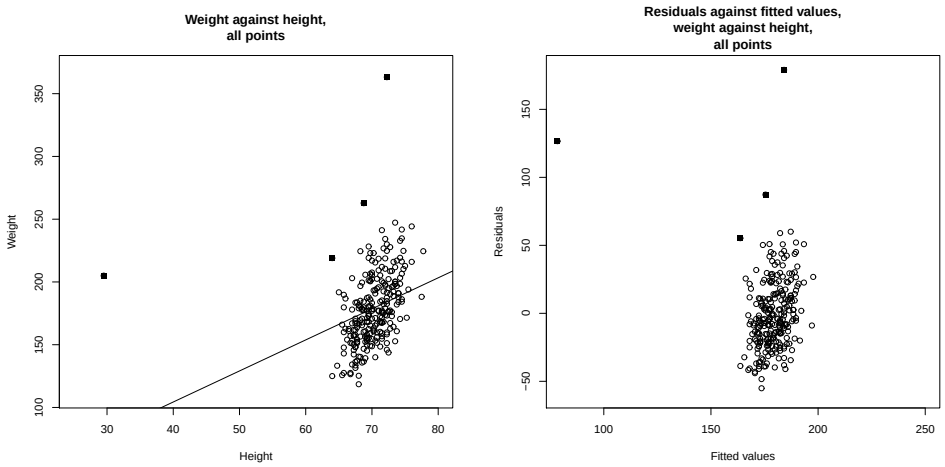

FIGURE 7.7: *On the* **left**, *weight regressed against height for the bodyfat dataset. The line doesn't describe the data particularly well, because it has been strongly affected by a few data points (filled-in markers). On the* **right**, *a scatter plot of the residual against the value predicted by the regression. This doesn't look like noise, which is a sign of trouble.*

This creates a problem, because data points that are very different from most others (sometimes called **outliers**) can also have the highest influence on the outcome of the regression. Figure 7.8 shows this effect for a simple case. When we have only one explanatory variable, there's an easy method to spot problem data points. We produce a scatter plot and a regression line, and the difficulty is usually obvious. In particularly tricky cases, printing the plot and using a see-through ruler to draw a line by eye can help (if you use an opaque ruler, you may not see some errors).

These data points can come from many sources. They may simply be errors. Failures of equipment, transcription errors, someone guessing a value to replace lost data, and so on are some methods that might produce outliers. Another possibility is your understanding of the problem is wrong. If there are some rare effects that are very different than the most common case, you might see outliers. Major scientific discoveries have resulted from investigators taking outliers seriously, and trying to find out what caused them (though you shouldn't see a Nobel prize lurking behind every outlier).

What to do about outliers is even more fraught. The simplest strategy is to find them, then remove them from the data. I will describe some methods that can identify outliers, but you should be aware that this strategy can get dangerous fairly quickly. First, you might find that each time you remove a few problematic data points, some more data points look strange to you. This process is unlikely to end well. Second, you should be aware that throwing out outliers can *increase* your future prediction error, particularly if they're caused by real effects. An alternative strategy is to build methods that can either discount the effects of outliers, or

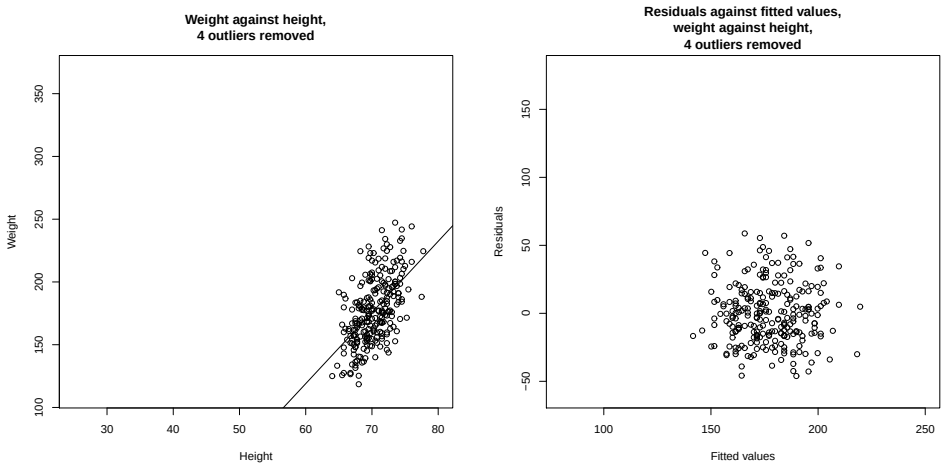

FIGURE 7.8: *On the* **left**, *weight regressed against height for the bodyfat dataset. I have now removed the four suspicious looking data points, identified in Figure 7.7 with filled-in markers; these seemed the most likely to be outliers. On the* **right**, *a scatter plot of the residual against the value predicted by the regression. Notice that the residual looks like noise. The residual seems to be uncorrelated to the predicted value; the mean of the residual seems to be zero; and the variance of the residual doesn't depend on the predicted value. All these are good signs, consistent with our model, and suggest the regression will yield good predictions.*

model them; I describe some such methods, which can be technically complex, in the following chapter.

Remember this: *Outliers can affect linear regressions significantly. Usually, if you can plot the regression, you can look for outliers by eyeballing the plot. Other methods exist, but are beyond the scope of this text.*

7.3.2 The Hat Matrix and Leverage

Write $\hat{\beta}$ for the estimated value of β, and $\mathbf{y}^{(p)} = \mathcal{X}\hat{\beta}$ for the predicted y values. Then we have

$$\hat{\beta} = \left(\mathcal{X}^T\mathcal{X}\right)^{-1}\left(\mathcal{X}^T\mathbf{y}\right)$$

so that

$$\mathbf{y}^{(p)} = (\mathcal{X}\left(\mathcal{X}^T\mathcal{X}\right)^{-1}\mathcal{X}^T)\mathbf{y}.$$

What this means is that the values the model predicts at training points are a linear function of the true values at the training points. The matrix $(\mathcal{X}\left(\mathcal{X}^T\mathcal{X}\right)^{-1}\mathcal{X}^T)$ is

sometimes called the **hat matrix**. The hat matrix is written $\mathcal{H}$, and I shall write the i, j'th component of the hat matrix h_{ij}.

Remember this: *The predictions of a linear regression at training points are a linear function of the y-values at the training points. The linear function is given by the hat matrix.*

The hat matrix has a variety of important properties. I won't prove any here, but the proofs are in the exercises. It is a symmetric matrix. The eigenvalues can be only 1 or 0. And the row sums have the important property that

$$\sum_j h_{ij}^2 \leq 1.$$

This is important, because it can be used to find data points that have values that are hard to predict. The **leverage** of the i'th training point is the i'th diagonal element, h_{ii}, of the hat matrix $\mathcal{H}$. Now we can write the prediction at the i'th training point $y_{p,i} = h_{ii}y_i + \sum_{j \neq i} h_{ij}y_j$. But if h_{ii} has large absolute value, then all the other entries in that row of the hat matrix must have small absolute value. This means that, if a data point has high leverage, the model's value at that point is predicted almost entirely by the observed value at that point. Alternatively, it's hard to use the other training data to predict a value at that point.

Here is another way to see this importance of h_{ii}. Imagine we change the value of y_i by adding Δ; then $y_i^{(p)}$ becomes $y_i^{(p)} + h_{ii}\Delta$. In turn, a large value of h_{ii} means that the predictions at the i'th point are very sensitive to the value of y_i.

Remember this: *Ideally, the value predicted for a particular data point depends on many other data points. Leverage measures the importance of a data point in producing a prediction at that data point. If the leverage of a point is high, other points are not contributing much to the prediction for that point, and it may well be an outlier.*

7.3.3 Cook's Distance

Another way to find points that may be creating problems is to look at the effect of omitting the point from the regression. We could compute $\mathbf{y}^{(p)}$ using the whole data set. We then omit the i'th point from the dataset, compute the regression coefficients from the remaining data (which I will write $\hat{\beta}_{\hat{i}}$), then compare $\mathbf{y}^{(p)}$ to $\mathcal{X}\hat{beta}_{\hat{i}}$. If there is a large difference, the point is suspect, because omitting it

strongly changes the predictions. The score for the comparison is called **Cook's distance**. If a point has a large value of Cook's distance, then it has a strong influence on the regression and might well be an outlier. Typically, one computes Cook's distance for each point, and takes a closer look at any point with a large value. This procedure is described in more detail in procedure 96

Notice the rough similarity to cross-validation (omit some data and recompute). But in this case, we are using the procedure to identify points we might not trust, rather than to get an unbiased estimate of the error.

Procedure: 7.2 *Computing Cook's distance*

We have a dataset containing N pairs $(\mathbf{x}_i, y_i)$. Each x_i is a d-dimensional explanatory vector, and each y_i is a single dependent variable. Write $\hat{\beta}$ for the coefficients of a linear regression (see procedure 7.1), and $\hat{\beta}_{\hat{i}}$ for the coefficients of the linear regression computed by omitting the i'th data point, $\mathbf{y}^{(p)}$ for $\mathcal{X}\hat{\beta}$, and m for the mean square error. The Cook's distance of the i'th data point is

$$\frac{(\mathbf{y}^{(p)} - \mathcal{X}\hat{\beta}_{\hat{i}})^T(\mathbf{y}^{(p)} - \mathcal{X}\hat{\beta}_{\hat{i}})}{dm}.$$

Large values of this distance suggest a point may present problems. Statistical software will compute and plot this distance for you.

Remember this: *The Cook's distance of a training data point measures the effect on predictions of leaving that point out of the regression. A large value of Cook's distance suggests other points are poor at predicting the value at a given point, so a point with a large value of Cook's distance may be an outlier.*

7.3.4 Standardized Residuals

The hat matrix has another use. It can be used to tell how "large" a residual is. The residuals that we measure depend on the units in which y was expressed, meaning we have no idea what a "large" residual is. For example, if we were to express y in kilograms, then we might want to think of 0.1 as a small residual. Using exactly the same dataset, but now with y expressed in grams, that residual value becomes 100 — is it really "large" because we changed units?

Now recall that we assumed, in section 7.2.1, that $y - \mathbf{x}^T\beta$ was a zero mean normal random variable, but we didn't know its variance. It can be shown that,

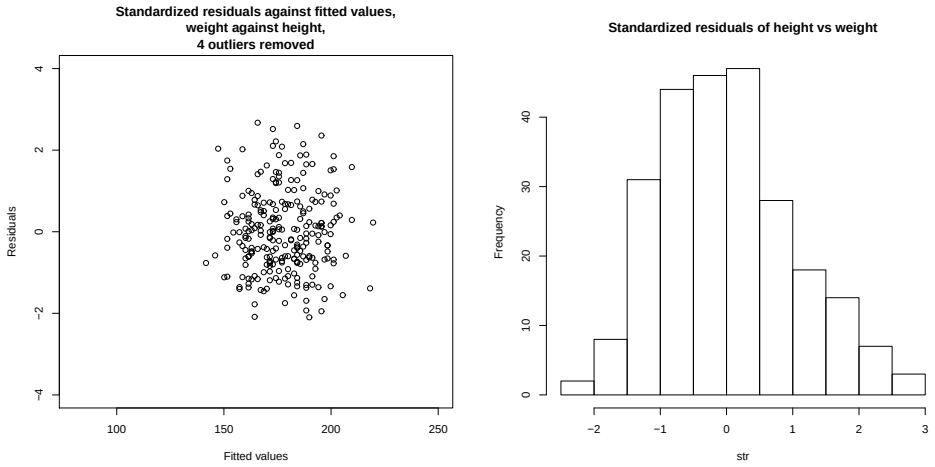

FIGURE 7.9: *On the* **left**, *standardized residuals plotted against predicted value for weight regressed against height for the bodyfat dataset. I removed the four suspicious looking data points, identified in Figure 7.7 with filled-in markers ; these seemed the most likely to be outliers. You should compare this plot with the residuals in figure 7.8, which are not standardized. On the* **right**, *a histogram of the residual values. Notice this looks rather like a histogram of a standard normal random variable, though there are slightly more large positive residuals than one would like. This suggests the regression is working tolerably.*

under our assumption, the i'th residual value, e_i, is a sample of a normal random variable whose variance is

$$\left(\frac{(\mathbf{e}^T\mathbf{e})}{N}\right)(1-h_{ii}).$$

This means we can tell whether a residual is large by **standardizing** it – that is, dividing by its standard deviation. Write s_i for the standard residual at the i'th training point. Then we have that

$$s_i = \frac{e_i}{\sqrt{\left(\frac{(\mathbf{e}^T\mathbf{e})}{N}\right)(1-h_{ii})}}.$$

When the regression is behaving, this standard residual should look like a sample of a standard normal random variable. In turn, this means that if all is going well, about 66% of the residuals should have values in the range $[-1, 1]$, and so on. Large values of the standard residuals are a sign of trouble.

R produces a nice diagnostic plot that can be used to look for problem data points (code and details in the appendix). The plot is a scatter plot of the standardized residuals against leverage, with level curves of Cook's distance superimposed. Figure 7.10 shows an example. Some bad points that are likely to present problems are identified with a number (you can control how many, and the number, with arguments to `plot`; appendix). Problem points will have high leverage and/or high

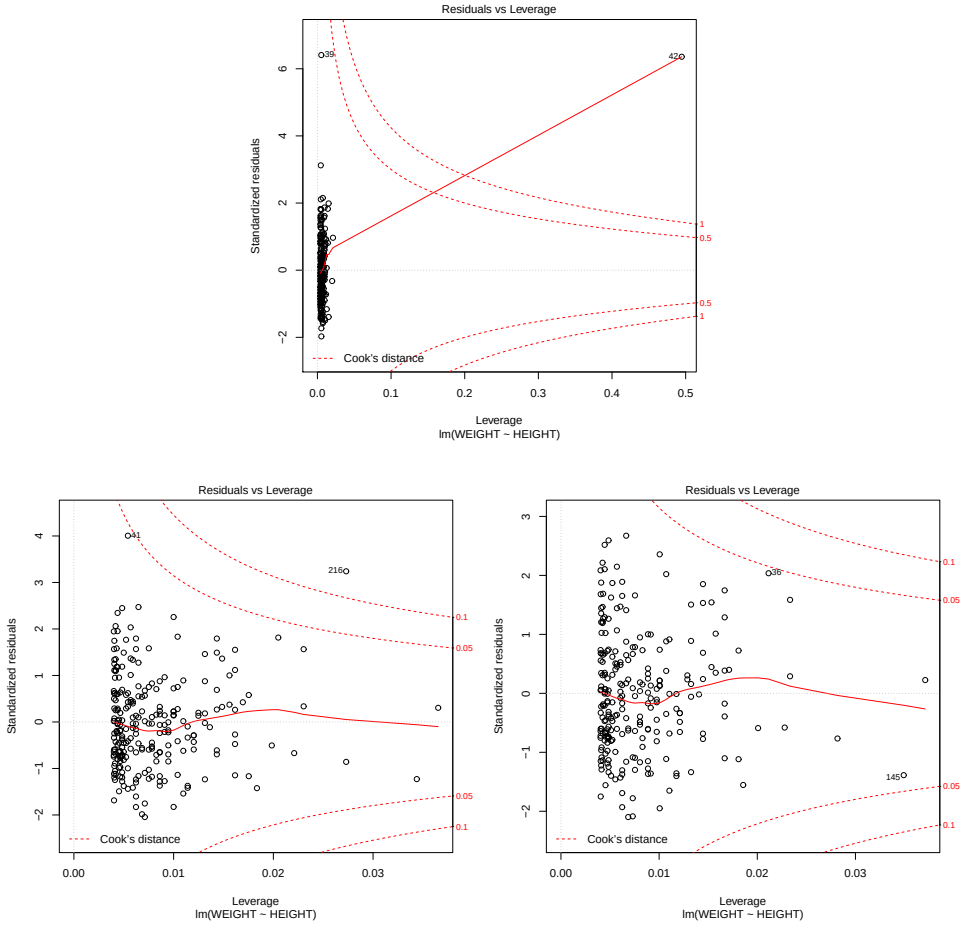

FIGURE 7.10: *A diagnostic plot, produced by R, of a linear regression of weight against height for the bodyfat dataset.* **Top:** *the whole dataset;* **bottom left:** *with the two most extreme points in the top figure removed;* **bottom right:** *with two further points (highest residual) removed. Details in text.*

Cook's distance and/or high residual. The figure shows this plot for three different versions of the dataset (original; two problem points removed; and two further problem points removed).

7.4 MANY EXPLANATORY VARIABLES

In earlier sections, I implied you could put anything into the explanatory variables. This is correct, and makes it easy to do the math for the general case. However, I have plotted only cases where there was one explanatory variable (together with a constant, which hardly counts). In some cases (section 7.4.1), we can add explana-

tory variables and still have an easy plot. Adding explanatory variables can cause the matrix $\mathcal{X}^T\mathcal{X}$ to have poor condition number; there's an easy strategy to deal with this (section 7.4.2).

Most cases are hard to plot successfully, and one needs better ways to visualize the regression than just plotting. The value of R^2 is still a useful guide to the goodness of the regression, but the way to get more insight is to use the tools of the previous section.

7.4.1 Functions of One Explanatory Variable

Imagine we have only one measurement to form explanatory variables. For example, in the perch data of Figure 7.1, we have only the length of the fish. If we evaluate functions of that measurement, and insert them into the vector of explanatory variables, the resulting regression is still easy to plot. It may also offer better predictions. The fitted line of Figure 7.1 looks quite good, but the data points look as though they might be willing to follow a curve. We can get a curve quite easily. Our current model gives the weight as a linear function of the length with a noise term (which we wrote $y_i = \beta_1 x_i + \beta_0 + \xi_i$). But we could expand this model to incorporate other functions of the length. In fact, it's quite suprising that the weight of a fish should be predicted by its length. If the fish doubled in each direction, say, its weight should go up by a factor of eight. The success of our regression suggests that fish do not just scale in each direction as they grow. But we might try the model $y_i = \beta_2 x_i^2 + \beta_1 x_i + \beta_0 + \xi_i$. This is easy to do. The i'th row of the matrix $\mathcal{X}$ currently looks like $[x_i, 1]$. We build a new matrix $\mathcal{X}^{(b)}$, where the i'th row is $[x_i^2, x_i, 1]$, and proceed as before. This gets us a new model. The nice thing about this model is that it is easy to plot – our predicted weight is still a function of the length, it's just not a linear function of the length. Several such models are plotted in Figure 7.11.

You should notice that it can be quite easy to add a lot of functions like this (in the case of the fish, I tried x_i^3 as well). However, it's hard to decide whether the regression has actually gotten better. The least-squares error *on the training data* will *never* go up when you add new explanatory variables, so the R^2 will *never* get worse. This is easy to see, because you could always use a coefficient of zero with the new variables and get back the previous regression. However, the models that you choose are likely to produce worse and worse predictions as you add explanatory variables. Knowing when to stop can be tough (Section 8.1), though it's sometimes obvious that the model is untrustworthy (Figure 7.11).

Remember this: *If you have only one measurement, you can construct a high dimensional* $\mathbf{x}$ *by using functions of that measurement. This produces a regression that has many explanatory variables, but is still easy to plot. Knowing when to stop is hard. An understanding of the problem is helpful.*

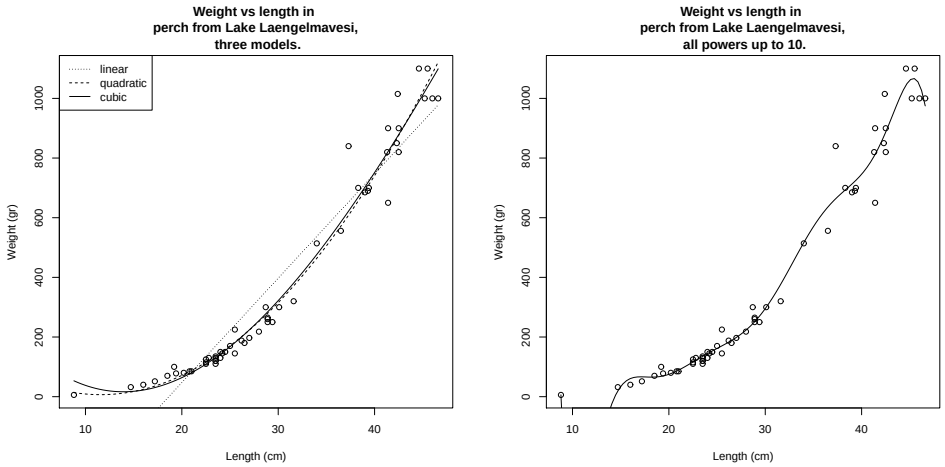

FIGURE 7.11: *On the* **left**, *several different models predicting fish weight from length. The line uses the explanatory variables* 1 *and* x_i*; and the curves use other monomials in* x_i *as well, as shown by the legend. This allows the models to predict curves that lie closer to the data. It is important to understand that, while you can make a curve go closer to the data by inserting monomials, that doesn't mean you necessarily have a better model. On the* **right**, *I have used monomials up to* x_i^{10}*. This curve lies very much closer to the data points than any on the other side, at the cost of some very odd looking wiggles inbetween data points (look at small lengths; the model goes quite strongly negative there, but I can't bring myself to change the axes and show predictions that are obvious nonsense). I can't think of any reason that these structures would come from true properties of fish, and it would be hard to trust predictions from this model.*

7.4.2 Regularizing Linear Regressions

When we have many explanatory variables, some might be significantly correlated. This means that we can predict, quite accurately, the value of one explanatory variable using the values of the other variables. This means there must be a vector $\mathbf{w}$ so that $\mathcal{X}\mathbf{w}$ is small (exercises). In turn, that $\mathbf{w}^T\mathcal{X}^T\mathcal{X}\mathbf{w}$ must be small, so that $\mathcal{X}^T\mathcal{X}$ has some small eigenvalues. These small eigenvalues lead to bad predictions, as follows. The vector $\mathbf{w}$ has the property that $\mathcal{X}^T\mathcal{X}\mathbf{w}$ is small. This means that $\mathcal{X}^T\mathcal{X}(\hat{\beta} + \mathbf{w})$ is not much different from $\mathcal{X}^T\mathcal{X}\hat{\beta}$ (equivalently, the matrix can turn large vectors into small ones). All this means that $(\mathcal{X}^T\mathcal{X})^{-1}$ will turn some small vectors into big ones. A small change in $\mathcal{X}^T\mathbf{Y}$ can lead to a large change in the estimate of $\hat{\beta}$.

This is a problem, because we can expect that different samples from the same data will have somewhat different values of $\mathcal{X}^T\mathbf{Y}$. For example, imagine the person recording fish measurements in Lake Laengelmavesi recorded a different set of fish; we expect changes in $\mathcal{X}$ and $\mathbf{Y}$. But, if $\mathcal{X}^T\mathcal{X}$ has small eigenvalues, these changes could produce large changes in our model.

The problem is relatively easy to control. When there are small eigenvalues

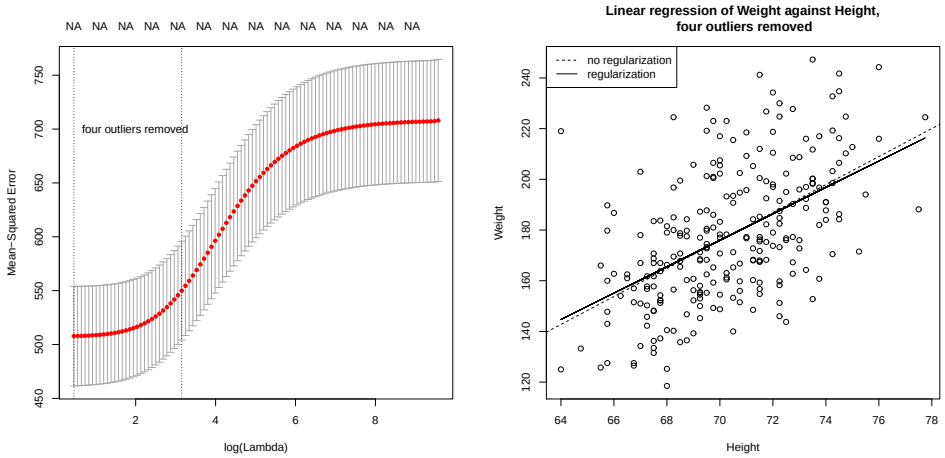

FIGURE 7.12: *On the* **left**, *cross-validated error estimated for different choices of regularization constant for a linear regression of weight against height for the bodyfat dataset, with four outliers removed. The horizontal axis is log regression constant; the vertical is cross-validated error. The mean of the error is shown as a spot, with vertical error bars. The vertical lines show a range of reasonable choices of regularization constant (***left*** yields the lowest observed error,* **right** *the error whose mean is within one standard error of the minimum). On the* **right**, *two regression lines on a scatter plot of this dataset; one is the line computed without regularization, the other is obtained using the regularization parameter that yields the lowest observed error. In this case, the regularizer doesn't change the line much, but may produce improved values on new data (notice how the cross-validated error is fairly flat with low values of the regularization constant).*

in $\mathcal{X}^T\mathcal{X}$, we expect that $\hat{\beta}$ will be large (because we can add components in the direction of $\mathbf{w}$ without changing all that much), and the largest components in $\hat{\beta}$ might be very inaccurately estimated. If we are trying to predict new y values, we expect that large components in $\hat{\beta}$ turn into large errors in prediction (exercises).

An important and useful way to suppress these errors is to try to find a $\hat{\beta}$ that isn't large, and also gives a low error. We can do this by regularizing, using the same trick we saw in the case of classification. Instead of choosing the value of β that minimizes

$$\left(\frac{1}{N}\right)(\mathbf{y}-\mathcal{X}\beta)^T(\mathbf{y}-\mathcal{X}\beta)$$

we minimize

$$\begin{array}{rcl}\left(\frac{1}{N}\right)(\mathbf{y}-\mathcal{X}\beta)^T(\mathbf{y}-\mathcal{X}\beta) & + & \lambda\beta^T\beta \\ \text{Error} & + & \text{Regularizer}\end{array}$$

Here $\lambda > 0$ is a constant that weights the two requirements (small error; small $\hat{\beta}$) relative to one another. Notice also that dividing the total error by the number of

data points means that our choice of λ shouldn't be affected by changes in the size of the data set.

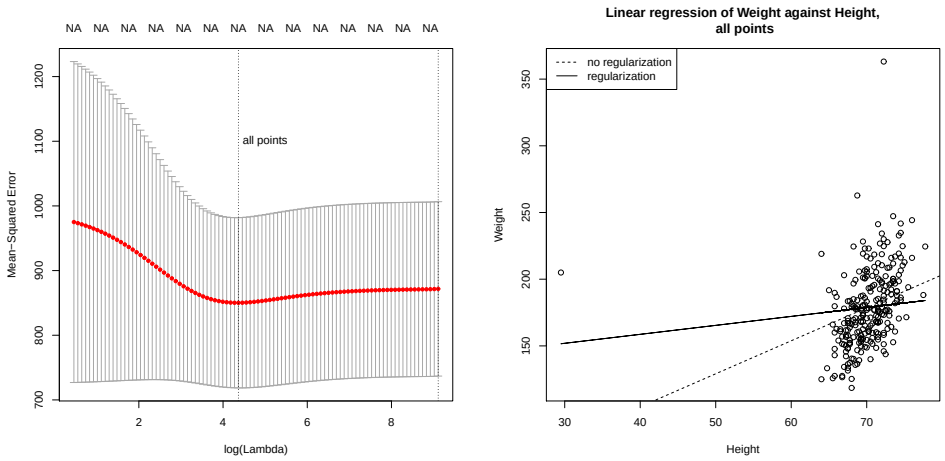

FIGURE 7.13: *Regularization doesn't make outliers go away. On the* **left**, *cross-validated error estimated for different choices of regularization constant for a linear regression of weight against height for the bodyfat dataset, with all points. The horizontal axis is log regression constant; the vertical is cross-validated error. The mean of the error is shown as a spot, with vertical error bars. The vertical lines show a range of reasonable choices of regularization constant (***left** *yields the lowest observed error,* **right** *the error whose mean is within one standard error of the minimum). On the* **right**, *two regression lines on a scatter plot of this dataset; one is the line computed without regularization, the other is obtained using the regularization parameter that yields the lowest observed error. In this case, the regularizer doesn't change the line much, but may produce improved values on new data (notice how the cross-validated error is fairly flat with low values of the regularization constant).*

Regularization helps deal with the small eigenvalue, because to solve for β we must solve the equation

$$\left[\left(\frac{1}{N}\right)\mathcal{X}^T\mathcal{X} + \lambda\mathcal{I}\right]\hat{\beta} = \left(\frac{1}{N}\right)\mathcal{X}^T\mathbf{y}$$

(obtained by differentiating with respect to β and setting to zero) and the smallest eigenvalue of the matrix $\left(\left(\frac{1}{N}\right)\left(\mathcal{X}^T\mathcal{X} + \lambda\mathcal{I}\right)\right.$ will be at least λ (exercises). Penalizing a regression with the size of β in this way is sometimes known as **ridge regression**.

We choose λ in the same way we used for classification; split the training set into a training piece and a validation piece, train for different values of λ, and test the resulting regressions on the validation piece. The error is a random variable, random because of the random split. It is a fair model of the error that would occur on a randomly chosen test example (assuming that the training set is "like" the test set, in a way that I do not wish to make precise yet). We could use multiple

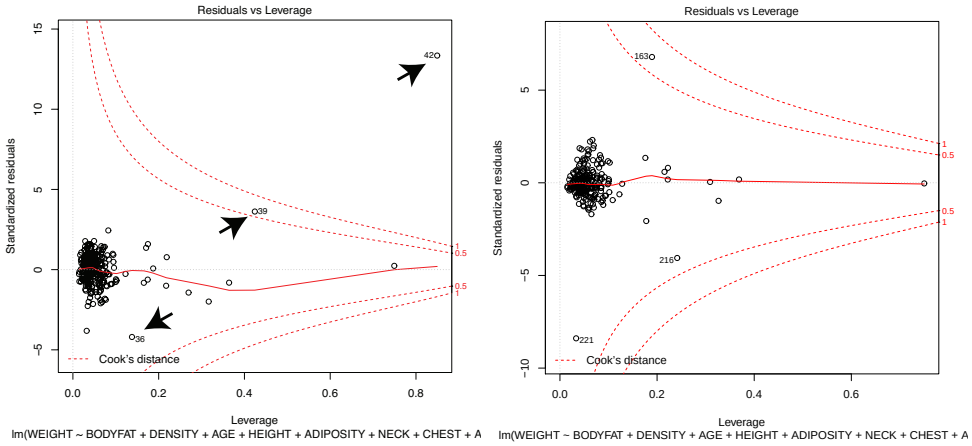

FIGURE 7.14: *On the* **left**, *residuals plotted against leverage for a regression of weight against all other measurements for the bodyfat dataset. I did* not *remove the outliers. The contours on the plot are contours of Cook's distance; I have overlaid arrows showing points with suspiciously large Cook's distance. Notice also that several points have high leverage, without having a large residual value. These points may or may not present problems. On the* **right**, *the same plot for this dataset with points 36, 39, 41 and 42 removed (these are the points I have been removing for each such plot). Notice that another point now has high Cook's distance, but mostly the residual is much smaller.*

splits, and average over the splits. Doing so yields both an average error for a value of λ and an estimate of the standard deviation of error.

Statistical software will do all the work for you. I used the `glmnet` package in R (see exercises for details). Figure 7.12 shows an example, for weight regressed against height. Notice the regularization doesn't change the model (plotted in the figure) all that much. For each value of λ (horizontal axis), the method has computed the mean error and standard deviation of error using cross-validation splits, and displays these with error bars. Notice that $\lambda = 0$ yields poorer predictions than a larger value; large $\hat{\beta}$ really are unreliable. Notice that now there is now no λ that yields the smallest validation error, because the value of error depends on the random splits used in cross-validation. A reasonable choice of λ lies between the one that yields the smallest error encountered (one vertical line in the plot) and the largest value whose mean error is within one standard deviation of the minimum (the other vertical line in the plot).

All this is quite similar to regularizing a classification problem. We started with a cost function that evaluated the errors caused by a choice of β, then added a term that penalized β for being "large". This term is the squared length of β, as a vector. It is sometimes known as the L_2 **norm** of the vector. In section 9.9, I describe the consequences of using other norms.

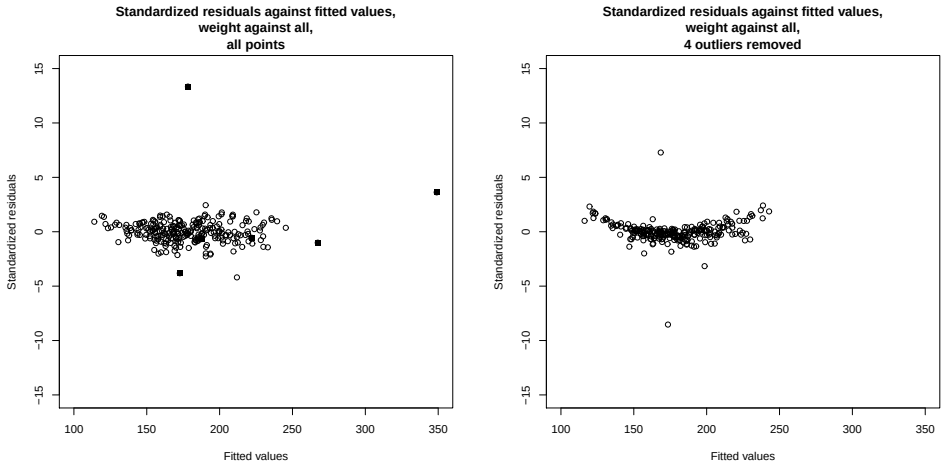

FIGURE 7.15: *On the* **left**, *standardized residuals plotted against predicted value for weight regressed against all variables for the bodyfat dataset. Four data points appear suspicious, and I have marked these with a filled in marker. On the* **right**, *standardized residuals plotted against predicted value for weight regressed against all variables for the bodyfat dataset, but with the four suspicious looking data points removed. Notice two other points stick out markedly.*

> **Remember this:** *The performance of a regression can be improved by regularizing, particularly if some explanatory variables are correlated. The procedure is similar to that used for classification.*

7.4.3 Example: Weight against Body Measurements

We can now look at regressing weight against all body measurements for the bodyfat dataset. We can't plot this regression (too many independent variables), but we can approach the problem in a series of steps.

Finding suspect points: Figure 7.14 shows the R diagnostic plots for a regression of weight against all body measurements for the bodyfat dataset. We've already seen there are outliers, so the odd structure of this plot should be no particular surprise. There are several really worrying points here. As the figure shows, removing the four points identified in the caption, based on their very high standardized residuals, high leverage, and high Cook's distance, yields improvements. We can get some insight by plotting standardized residuals against predicted value (Figure 7.9). There is clearly a problem here; the residual seems to depend quite strongly on the predicted value. Removing the four outliers we have already identified leads to a much improved plot, also shown in Figure 7.15. This is banana-shaped, which is suspicious. There are two points that seem to come from some

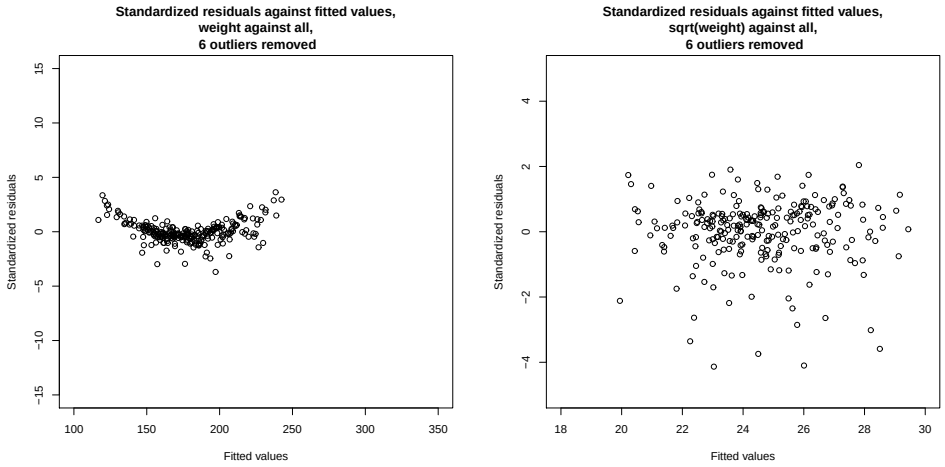

FIGURE 7.16: *On the* **left**, *standardized residuals plotted against predicted value for weight regressed against all variables for the bodyfat dataset. I removed the four suspicious data points of Figure 7.15, and the two others identified in that figure. Notice a suspicious "banana" shape – the residuals are distinctly larger for small and for large predicted values. This suggests a non-linear transformation of something might be helpful. I used a Box-Cox transformation, which suggested a value of 0.5 (i.e. regress* $2(\sqrt{weight} - 1)$*) against all variables. On the* **right**, *the standardized residuals for this regression. Notice that the "banana" has gone, though there is a suspicious tendency for the residuals to be smaller rather than larger. Notice also the plots are on different axes. It's fair to compare these plots by eye; but it's not fair to compare details, because the residual of a predicted square root means something different than the residual of a predicted value.*

other model (one above the center of the banana, one below). Removing these points gives the residual plot shown in Figure 7.16.

Transforming variables: The banana shape of the plot of standardized residuals against value is a suggestion that some non-linearity somewhere would improve the regression. One option is a non-linear transformation of the independent variables. Finding the right one might require some work, so it's natural to try a Box-Cox transformation first. This gives the best value of the parameter as 0.5 (i.e. the dependent variable should be $\sqrt{\text{weight}}$, which makes the residuals look much better (Figure 7.16).

Choosing a regularizing value: Figure 7.17 shows the `glmnet` plot of cross-validated error as a function of regularizer weight. A sensible choice of value here seems to be a bit smaller than -2 (between the value that yields the smallest error encountered – one vertical line in the plot – and the largest value whose mean error is within one standard deviation of the minimum – the other vertical line in the plot). I chose -2.2

How good are the resulting predictions likely to be: the standardized residuals don't seem to depend on the predicted values, but how good are the

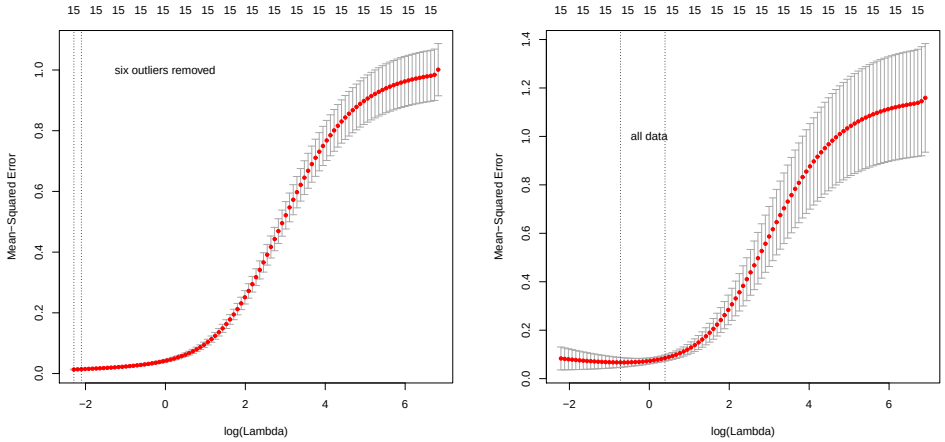

FIGURE 7.17: *Plots of mean-squared error as a function of log regularization parameter (i.e.* $\log \lambda$*) for a regression of weight*$^{1/2}$ *against all variables for the bodyfat dataset. These plots show mean-squared error averaged over cross-validation folds with a vertical one standard deviation bar. On the* **left***, the plot for the dataset with the six outliers identified in Figure 9.9 removed. On the* **right***, the plot for the whole dataset. Notice how the outliers increase the variability of the error, and the best error.*

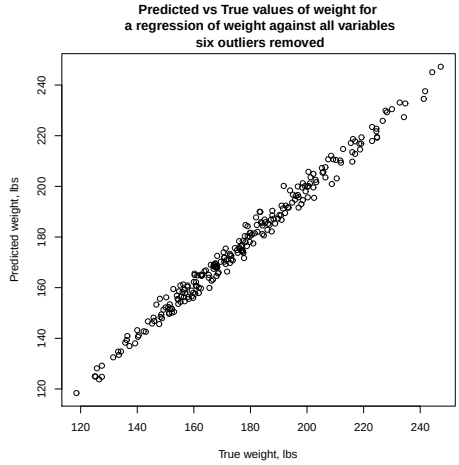

FIGURE 7.18: *A scatter plot of the predicted weight against the true weight for the bodyfat dataset. The prediction is made with all variables, but the six outliers identified above are omitted. I used a Box-Cox transformation with parameter* $1/2$*, and the regularization parameter that yielded the smallest mean square error in Figure 7.17.*

predictions? We already have some information on this point. Figure 7.17 shows cross-validation errors for regressions of weight$^{1/2}$ against height for different regularization weights, but some will find this slightly indirect. We want to predict weight, not weight$^{1/2}$. I chose the regularization weight that yielded the lowest mean-square-error for the model of Figure 7.17, omitting the six outliers previously mentioned. I then computed the predicted weight for each data point using that model (which predicts weight$^{1/2}$, remember; but squaring takes care of that). Figure 7.18 shows the predicted values plotted against the true values. You should not regard this plot as a safe way to estimate generalization (the points were used in training the model; Figure 7.17 is better for that), but it helps to visualize the errors. This regression looks as though it is quite good at predicting bodyweight from other measurements.

7.5 YOU SHOULD

7.5.1 remember:

New term: Regression . . . 138
New term: explanatory variables . . . 138
New term: dependent variable . . . 138
New term: training examples . . . 138
New term: test examples . . . 138
New term: residual . . . 141
Definition: Regression . . . 142
New term: explanatory variables . . . 142
New term: dependent variable . . . 142
New term: training examples . . . 142
New term: test examples . . . 142
Definition: Linear regression . . . 143
Estimating β . . . 145
New term: residual . . . 145
New term: mean square error . . . 146
Useful facts: Regression . . . 147
R^2 evaluates the quality of predictions made by a regression . . . 148
New term: Zipf's law . . . 149
Transforming variables is useful . . . 150
New term: Box-Cox transformation . . . 150
Linear regressions can fail. . . . 151
New term: outliers . . . 154
Outliers can affect linear regressions significantly. . . . 155
New term: hat matrix . . . 156
The hat matrix mixes training y-values to produce predictions. . . . 156
New term: leverage . . . 156
Be suspicious of points with high leverage. . . . 156
New term: Cook's distance . . . 157
Be suspicious of points with high Cook's distance. . . . 157
New term: standardizing . . . 158
Appending functions of a measurement to $\mathbf{x}$ is useful. . . . 160
New term: ridge regression . . . 163
New term: L_2 norm . . . 164
You can regularize a regression . . . 165
New term: condition number . . . 171
New term: condition number . . . 171

APPENDIX: DATA

Batch A		Batch B		Batch C	
Amount of Hormone	Time in Service	Amount of Hormone	Time in Service	Amount of Hormone	Time in Service
25.8	99	16.3	376	28.8	119
20.5	152	11.6	385	22.0	188
14.3	293	11.8	402	29.7	115
23.2	155	32.5	29	28.9	88
20.6	196	32.0	76	32.8	58
31.1	53	18.0	296	32.5	49
20.9	184	24.1	151	25.4	150
20.9	171	26.5	177	31.7	107
30.4	52	25.8	209	28.5	125

TABLE 7.1: *A table showing the amount of hormone remaining and the time in service for devices from lot A, lot B and lot C. The numbering is arbitrary (i.e. there's no relationship between device 3 in lot A and device 3 in lot B). We expect that the amount of hormone goes down as the device spends more time in service, so cannot compare batches just by comparing numbers.*

PROBLEMS

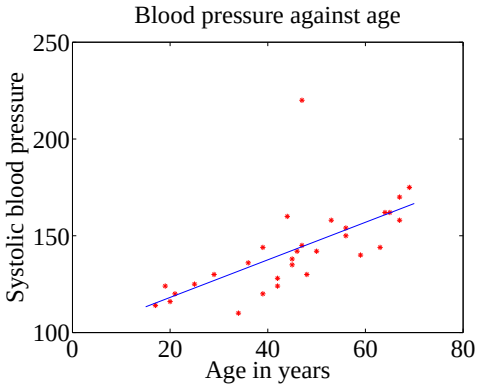

FIGURE 7.19: *A regression of blood pressure against age, for 30 data points.*

7.1. Figure 8.7 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)?

7.2. This exercise investigates the effect of correlation on a regression. Assume we have N data items, $(\mathbf{x}_i, y_i)$. 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 $\mathbf{x}_i$. Assume our data has the property that $x_{i1} = \mathbf{x}_{i\hat{1}}^T \mathbf{u} + r_i$. Write $\mathbf{r}$ for the vector of residuals (i.e. the i'th component of $\mathbf{r}$ is r_i). Now assume $\mathbf{r}^T \mathbf{1} = 0$ (i.e. the average of the r_i's is zero) and $\mathbf{r}^T \mathbf{r} \leq \epsilon$. All this means that the data have the property that the first component is relatively accurately predicted by the other components.

(a) Write $\mathbf{w} = [-1, \mathbf{u}]^T$. Show that

$$\mathbf{w}^T \mathcal{X}^T \mathcal{X} \mathbf{w} \leq \epsilon.$$

(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 $s\max$ for $\max(s_1, \ldots, s_d)$. Show that the largest eigenvalue of $\mathcal{X}^T \mathcal{X}$ is greater than or equal to $s\max$.

(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

$$(\mathcal{X}^T \mathbf{Y} - \mathcal{X}^T \mathcal{X}(\hat{\beta} + \mathbf{w}))^T (\mathcal{X}^T \mathbf{Y} - \mathcal{X}^T \mathcal{X}(\hat{\beta} + \mathbf{w}))$$

is bounded above by

$$\epsilon^2 (1 + \mathbf{u}^T \mathbf{u})$$

(f) Use the last sub exercises to explain why correlated data will lead to a poor estimate of $\hat{\beta}$.

7.3. 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) Show that $r = \mathsf{mean}\left(\{(x - \mathsf{mean}\left(\{x\}\right))(y - \mathsf{mean}\left(\{y\}\right))\}\right)/(\mathsf{std}\left(x\right)\mathsf{std}\left(y\right))$.

(b) Now write $s = \frac{\mathsf{std}(y)}{\mathsf{std}(x)}$. Now 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

$$sr(x_0 - \mathsf{mean}\left(\{x\}\right)) + \mathsf{mean}\left(\{y\}\right).$$

(c) Show that $sr = \mathsf{mean}\left(\{(xy)\}\right) - \mathsf{mean}\left(\{x\}\right)\mathsf{mean}\left(\{y\}\right)$.

(d) 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

$$\mathcal{X}^T\mathcal{X} = N \begin{pmatrix} \mathsf{mean}\left(\left\{x^2\right\}\right) & \mathsf{mean}\left(\{x\}\right) \\ \mathsf{mean}\left(\{x\}\right) & 1 \end{pmatrix}$$

(e) Now show that $\mathsf{var}\left(\{x\}\right) = \mathsf{mean}\left(\left\{(x - \mathsf{mean}\left(\{x\}\right))^2\right\}\right) = \mathsf{mean}\left(\left\{x^2\right\}\right) - \mathsf{mean}\left(\{x\}\right)^2$.

(f) Now show that $\mathsf{std}\left(x\right)\mathsf{std}\left(y\right)\mathsf{corr}\left(\{(x, y)\}\right) = \mathsf{mean}\left(\{(x - \mathsf{mean}\left(\{x\}\right))(y - \mathsf{mean}\left(\{y\}\right))\}\right)$.

CHAPTER 8

Regression: Choosing and Managing Models

8.1 MODEL SELECTION: WHICH MODEL IS BEST?

It is usually quite easy to have many explanatory variables in a regression problem. Even if you have only one measurement, you could always compute a variety of non-linear functions of that measurement. As we have seen, inserting variables into a model will reduce the fitting cost, but that doesn't mean that better predictions will result (section 7.4.1). We need to choose which explanatory variables we will use. A linear model with few explanatory variables may make poor predictions because the model itself is incapable of representing the independent variable accurately (an effect known as bias). A linear model with many explanatory variables may make poor predictions because we can't estimate the coefficients well (an effect known as variance). Choosing which explanatory variables we will use (and so which model we will use) requires that we balance these effects, described in greater detail in section 8.1.1. In the following sections, we describe straightforward methods of doing so.

8.1.1 Bias and Variance

We now look at the process of finding a model in a fairly abstract way. Doing so makes plain three distinct and important effects that cause models to make predictions that are wrong. One is **irreducible error**. Even a perfect choice of model can make mistake 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 $P(Y|\mathbf{x})$. This 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

$$\begin{aligned}\mathbb{E}\left[(Y - \hat{f}(\mathbf{X}))^2\right] &= \mathbb{E}\left[Y^2\right] - 2\mathbb{E}\left[Y\hat{f}\right] + \mathbb{E}\left[\hat{f}^2\right] \\ &= \mathsf{var}\left(\{Y\}\right) + \mathbb{E}[Y]^2 + \mathsf{var}\left(\left\{\hat{f}\right\}\right) + \mathbb{E}\left[\hat{f}\right]^2 - 2\mathbb{E}\left[Y\hat{f}\right].\end{aligned}$$

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

$$\begin{aligned}\mathbb{E}\left[(Y - \hat{f}(\mathbf{X}))^2\right] &= \mathsf{var}\left(\{Y\}\right) + \mathbb{E}[f]^2 + \mathsf{var}\left(\left\{\hat{f}\right\}\right) + \mathbb{E}\left[\hat{f}\right]^2 - 2\mathbb{E}\left[f\hat{f}\right] \\ &= \sigma_\xi^2 + \mathbb{E}\left[(f - \hat{f})^2\right] + \mathsf{var}\left(\left\{\hat{f}\right\}\right) \\ &= \sigma_\xi^2 + (f - \mathbb{E}\left[\hat{f}\right])^2 + \mathsf{var}\left(\left\{\hat{f}\right\}\right) \ (f \text{ isn't random}).\end{aligned}$$

The expected error on all future data is the sum of three terms. The irreducible error is σ_ξ^2; even the true model must produce this error, on average. The best model to choose would be $\mathbb{E}\left[\hat{f}\right]$ (remember, the expectation is over choices of training data; this model would be the one that best represented all possible attempts to train). But we don't have $\mathbb{E}\left[\hat{f}\right]$. Instead, we have $\hat{f}$. The variance is $\mathsf{var}\left(\left\{\hat{f}\right\}\right) = \mathbb{E}\left[(\hat{f} - \mathbb{E}\left[\hat{f}\right])^2\right]$. This term represents the fact that the model we chose ($\hat{f}$) is different from the mean model ($\mathbb{E}\left[\hat{f}\right]$). The difference arises because our training data is a subset of all data, and our model is chosen to be good on the training data, rather than on every possible training set. The bias is $(f - \mathbb{E}\left[\hat{f}\right])^2$. This term reflects the fact that even the best choice of model ($\mathbb{E}\left[\hat{f}\right]$) may not be the same as the true source of data ($\mathbb{E}[f]$ which is the same as f, because f is deterministic).

There is usually a tradeoff between bias and variance. Generally, when a model comes from a "small" or "simple" family, we expect that (a) we can estimate the best model in the family reasonably accurately (so the variance will be low) but (b) the model may have real difficulty reproducing the data (meaning the bias

is large). Similarly, if the model comes from a "large" or "complex" family, the variance is likely to be high (because it will be hard to estimate the best model in the family accurately) but the bias will be low (because the model can more accurately reproduce the data). All modelling involves managing this tradeoff between bias and variance. I am avoiding being precise about the complexity of a model because it can be tricky to do. One reasonable proxy is the number of parameters we have to estimate to determine the model.

You can see a crude version this tradeoff in the perch example of section 7.4.1 and Figure 7.11. Recall that, as I added monomials to the regression of weight against length, the fitting error went down; but the model that uses length^{10} as an explanatory variable makes very odd predictions away from the training data. When I use low degree monomials, the dominant source of error is bias; and when I use high degree monomials, the dominant source of error is variance. A common mistake is to feel that the major difficulty is bias, and so to use extremely complex models. Usually the result is poor estimates of model parameters, and huge variance. Experienced modellers fear variance far more than they fear bias.

The bias-variance discussion suggests it isn't a good idea simply to use all the explanatory variables that you can obtain (or think of). Doing so might lead to a model with serious variance problems. Instead, we must choose a model that uses a subset of the explanatory variables that is small enough to control variance, and large enough that the bias isn't a problem. We need some strategy to choose explanatory variables. The simplest (but by no means the best; we'll see better in this chapter) approach is to search sets of explanatory variables for a good set. The main difficulty is knowing when you have a good set.

8.1.2 Choosing a Model using Penalties: AIC and BIC

We would like to choose one of a set of models. We cannot do so using just the training error, because more complex models will tend to have lower training error, and so the model with the lowest training error will tend to be the most complex model. Training error is a poor guide to test error, because lower training error is evidence of lower bias on the models part; but with lower bias, we expect to see greater variance, and the training error doesn't take that into account.

One strategy is to penalize the model for complexity. We add some penalty, reflecting the complexity of the model, to the training error. We then expect to see the general behavior of figure 8.1. The training error goes down, and the penalty goes up as the model gets more complex, so we expect to see a point where the sum is at a minimum.

There are a variety of ways of constructing penalties. **AIC** (short for An Information Criterion) is a method due originally to Akaike, in ****. Rather than using the training error, AIC uses the maximum value of the log-likelihood of the model. Write $\mathcal{L}$ for this value. Write k for the number of parameters estimated to fit the model. Then the AIC is

$$2k - 2\mathcal{L}$$

and a better model has a smaller value of AIC (remember this by remembering 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

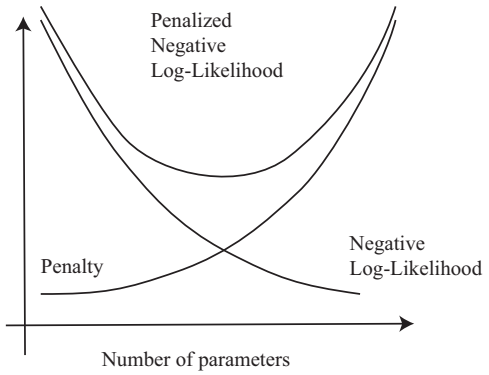

FIGURE 8.1: *When we add explanatory variables (and so parameters) to a model, the value of the negative log-likelihood of the best model can't go up, and usually goes down. This means that we cannot use the value as a guide to how many explanatory variables there should be. Instead, we add a penalty that increases as a function of the number of parameters, and search for the model that minimizes the sum of negative long-likelihood and penalty. AIC and BIC grow linearly with the number of parameters, but I am following the usual convention of plotting the penalty as a curve rather than a straight line.*

normal random variable. You estimate the mean-squared error, which gives the variance of the noise, and so the log-likelihood of the model. You do have to keep track of two points. First, k is the total number of parameters estimated to fit the model. For example, in a linear regression model, where you model y as $\mathbf{x}^T\beta + \xi$, you need to estimate d parameters to estimate $\hat{\beta}$ *and* the variance of ξ (to get the log-likelihood). So in this case $k = d + 1$. Second, log-likelihood is usually only known up to a constant, so that different software implementations often use different constants. This is wildly confusing when you don't know about it (why would `AIC` and `extractAIC` produce different numbers on the same model?) but of no real significance – you're looking for the smallest value of the number, and the actual value doesn't mean anything. Just be careful to compare only numbers computed with the same routine.

An alternative is **BIC** (Bayes' Information Criterion), given by

$$2k \log N - 2\mathcal{L}$$

(where N is the size of the training data set). You will often see this written as $2\mathcal{L} - 2k \log N$; I have given the form above so that one always wants the smaller value as with AIC. There is a considerable literature comparing AIC and BIC. AIC has a mild reputation for overestimating the number of parameters required, but is often argued to have firmer theoretical foundations.

Worked example 8.1 *AIC and BIC*

Write M_d for the model that predicts weight from length for the perch dataset as $\sum_{j=0}^{j=d} \beta_j \text{length}^j$. Choose an appropriate value of $d \in [1, 10]$ using AIC and BIC.

Solution: I used the R functions `AIC` and `BIC`, and got the table below.

	1	2	3	4	5	6	7	8	9	10
AIC	677	617	617	613	615	617	617	612	613	614
BIC	683	625	627	625	629	633	635	633	635	638

The best model by AIC has (rather startlingly!) $d = 8$. One should not take small differences in AIC too seriously, so models with $d = 4$ and $d = 9$ are fairly plausible, too. BIC suggests $d = 2$.

8.1.3 Choosing a Model using Cross-Validation

AIC and BIC are estimates of error on future data. An alternative is to measure this error on held out data, using a cross-validation strategy (as in section 3.1.4). One splits the training data into F **folds**, where each data item lies in exactly one fold. The case $F = N$ is sometimes called "leave-one-out" cross-validation. One then sets aside one fold in turn, fitting the model to the remaining data, and evaluating the model error on the left-out fold. The model error is then averaged. This process gives us an estimate of the performance of a model on held-out data. Numerous variants are available, particularly when lots of computation and lots of data are available. For example: one might not average over all folds; one might use fewer or more folds; and so on.

Worked example 8.2 *Cross-validation*

Write M_d for the model that predicts weight from length for the perch dataset as $\sum_{j=0}^{j=d} \beta_j \text{length}^j$. Choose an appropriate value of $d \in [1, 10]$ using leave-one-out cross validation.

Solution: I used the R functions `CVlm`, which takes a bit of getting used to. There is sample code in the exercises section. I found:

1	2	3	4	5	6	7	8	9	10
1.9e4	4.0e3	7.2e3	4.5e3	6.0e3	5.6e4	1.2e6	4.0e6	3.9e6	1.9e8

where the best model is $d = 2$.

8.1.4 A Search Process: Forward and Backward Stagewise Regression

Assume we have a set of explanatory variables and we wish to build a model, choosing some of those variables for our model. Our explanatory variables could be many distinct measurements, or they could be different non-linear functions of

the same measurement, or a combination of both. We can evaluate models relative to one another fairly easily (AIC, BIC or cross-validation, your choice). However, choosing which set of explanatory variables to use can be quite difficult, because there are so many sets. The problem is that you cannot predict easily what adding or removing an explanatory variable will do. Instead, when you add (or remove) an explanatory variable, the errors that the model makes change, and so the usefulness of all other variables changes too. This means that (at least in principle) you have to look at every subset of the explanatory variables. Imagine you start with a set of F possible explanatory variables (including the original measurement, and a constant). You don't know how many to use, so you might have to try every different group, of each size, and there are far too many groups to try. There are two useful alternatives.

In **forward stagewise regression**, you start with an empty working set of explanatory variables. You then iterate the following process. For each of the explanatory variables not in the working set, you construct a new model using the working set and that explanatory variable, and compute the model evaluation score. If the best of these models has a better score than the model based on the working set, you insert the appropriate variable into the working set and iterate. If no variable improves the working set, you decide you have the best model and stop. This is fairly obviously a greedy algorithm.

Backward stagewise regression is pretty similar, but you start with a working set containing all the variables, and remove variables one-by-one and greedily. As usual, greedy algorithms are very helpful but not capable of exact optimization. Each of these strategies can produce rather good models, but neither is guaranteed to produce the best model.

8.1.5 Significance: What Variables are Important?

Imagine you regress some measure of risk of death against blood pressure, whether someone smokes or not, and the length of their thumb. Because high blood pressure and smoking tend to increase risk of death, you would expect to see "large" coefficients for these explanatory variables. Since changes in the thumb length have no effect, you would expect to see "small" coefficients for these explanatory variables. This suggests a regression can be used to determine what effects are important in building a model.

One difficulty is the result of sampling variance. Imagine that we have an explanatory variable that has absolutely no relationship to the dependent variable. If we had an arbitrarily large amount of data, and could exactly identify the correct model, we'd find that, in the correct model, the coefficient of that variable was zero. But we don't have an arbitrarily large amount of data. Instead, we have a sample of data. Hopefully, our sample is random, so that the reasoning of section 9.9 can be applied. Using that reasoning, our estimate of the coefficient is the value of a random variable whose expected value is zero, but whose variance isn't. As a result, we are very unlikely to see a zero. This reasoning applies to each coefficient of the model. To be able to tell which ones are small, we would need to know the standard deviation of each, so we can tell whether the value we observe is a small number of standard deviations away from zero. This line of reasoning is very like hypothesis

testing. It turns out that the sampling variance of regression coefficients can be estimated in a straightforward way. In turn, we have an estimate of the extent to which their difference from zero could be a result of random sampling. R will produce this information routinely; use `summary` on the output of `lm`.

A second difficulty has to do with practical significance, and is rather harder. We could have explanatory variables that are genuinely linked to the independent variable, but might not matter very much. This is a common phenomenon, particularly in medical statistics. It requires considerable care to disentangle some of these issues. Here is an example. Bowel cancer is an unpleasant disease, which could kill you. Being screened for bowel cancer is at best embarrassing and unpleasant, and involves some startling risks. There is considerable doubt, from reasonable sources, about whether screening has value and if so, how much (as a start point, you could look at Ransohoff DF. How Much Does Colonoscopy Reduce Colon Cancer Mortality?. Ann Intern Med. 2009). There is some evidence linking eating red or processed meat to incidence of bowel cancer. A good practical question is: should one abstain from eating red or processed meat based on increased bowel cancer risk?

Coming to an answer is tough; the coefficient in any regression is clearly not zero, but it's pretty small as these numbers indicate. The UK population in 2012 was 63.7 million (this is a summary figure from Google, using World Bank data; there's no reason to believe that it's significantly wrong). I obtained the following figures from the UK cancer research institute website, at http://www.cancerresearchuk.org/health-professional/cancer-statistics/statistics-by-cancer-type/bowel-cancer. There were 41, 900 new cases of bowel cancer in the UK in 2012. Of these cases, 43% occurred in people aged 75 or over. 57% of people diagnosed with bowel cancer survive for ten years or more after diagnosis. Of diagnosed cases, an estimated 21% are linked to eating red or processed meat, and the best current estimate is that the risk of incidence is between 17% and 30% higher per 100g of red meat eaten per day (i.e. if you eat 100g of red meat per day, your risk increases by some number between 17% and 30%; 200g a day gets you twice that number; and – rather roughly – so on). These numbers are enough to confirm that there is a non-zero coefficient linking the amount of red or processed meat in your diet with your risk of bowel cancer (though you'd have a tough time estimating the exact value of that coefficient from the information here). If you eat more red meat, your risk of dying of bowel cancer really will go up. But the numbers I gave above suggest that (a) it won't go up much and (b) you might well die rather late in life, where the chances of dying of something are quite strong. The coefficient linking eating red meat and bowel cancer is clearly pretty small, because the incidence of the disease is about 1 in 1500 per year. Does it matter? you get to choose, and your choice has consequences.

8.2 ROBUST REGRESSION

We have seen that outlying data points can result in a poor model. This is caused by the squared error cost function: squaring a large error yields an enormous number. One way to resolve this problem is to identify and remove outliers before fitting a model. This can be difficult, because it can be hard to specify precisely when

a point is an outlier. Worse, in high dimensions most points will look somewhat like outliers, and we may end up removing all most all the data. The alternative solution I offer here is to come up with a cost function that is less susceptible to problems with outliers. The general term for a regression that can ignore some outliers is a **robust regression**.

8.2.1 M-Estimators and Iteratively Reweighted Least Squares

One way to reduce the effect of outliers on a least-squares solution would be to weight each point in the cost function. We need some method to estimate an appropriate set of weights. This would use a large weight for errors at points that are "trustworthy", and a low weight for errors at "suspicious" points.

We can obtain such weights using an **M-estimator**, which estimates parameters by replacing the negative log-likelihood with a term that is better behaved. In our examples, the negative log-likelihood has always been squared error. Write β for the parameters of the model being fitted, and $r_i(\mathbf{x}_i, \beta)$ for the residual error of the model on the ith data point. For us, r_i will always be $y_i - \mathbf{x}_i^T\beta$. So rather than minimizing

$$\sum_i (r_i(\mathbf{x}_i, \beta))^2$$

as a function of β, we will minimize an expression of the form

$$\sum_i \rho(r_i(\mathbf{x}_i, \beta); \sigma),$$

for some appropriately chosen function ρ. Clearly, our negative log-likelihood is one such estimator (use $\rho(u;\sigma) = u^2$). The trick to M-estimators is to make $\rho(u;\sigma)$ look like u^2 for smaller values of u, but ensure that it grows more slowly than u^2 for larger values of u.

The **Huber loss** is one important M-estimator. We use

$$\rho(u;\sigma) = \begin{cases} \frac{u^2}{2} & |u| < \sigma \\ \sigma|u| - \frac{\sigma^2}{2} & \end{cases}$$

which is the same as u^2 for $-\sigma \leq u \leq \sigma$, and then switches to $|u|$ for larger (or smaller) σ (Figure **??**). The Huber loss is convex (meaning that there will be a unique minimum for our models) and differentiable, but its derivative is not continuous. The choice of the parameter σ (which is known as **scale**) has an effect on the estimate. You should interpret this parameter as the distance that a point can lie from the fitted function while still being seen as an **inlier** (anything that isn't even partially an outlier).

Generally, M-estimators are discussed in terms of their **influence function**. This is

$$\frac{\partial \rho}{\partial u}.$$

Its importance becomes evidence when we consider algorithms to fit $\hat{\beta}$ using an

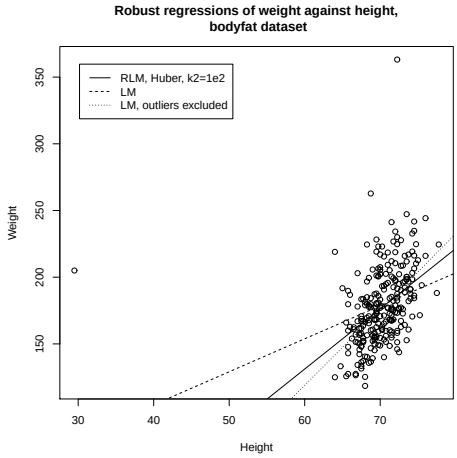

FIGURE 8.2: *Comparing three different linear regression strategies on the bodyfat data, regressing weight against height. Notice that using an M-estimator gives an answer very like that obtained by rejecting outliers by hand. The answer may well be "better" because it isn't certain that each of the four points rejected is an outlier, and the robust method may benefit from some of the information in these points. I tried a range of scales for the Huber loss (the 'k2' parameter), but found no difference in the line resulting over scales varying by a factor of 1e4, which is why I plot only one scale.*

M-estimator. Our minimization criterion is

$$\begin{aligned}\nabla_\beta \left(\sum_i \rho(y_i - \mathbf{x}_i^T\beta; \sigma) \right) &= \sum_i \left[\frac{\partial \rho}{\partial u}(y_i - \mathbf{x}_i^T\beta; \sigma) \right] (-\mathbf{x}_i) \\ &= 0.\end{aligned}$$

Now write $w_i(\beta)$ for

$$\frac{\frac{\partial \rho}{\partial u}(y_i - \mathbf{x}_i^T\beta; \sigma)}{y_i - \mathbf{x}_i^T\beta}.$$

We can write the minimization criterion as

$$\sum_i \left[w_i(\beta) \right] (y_i - \mathbf{x}_i^T\beta)(-\mathbf{x}_i) = 0.$$

Now write $\mathcal{W}(\beta)$ for the diagonal matrix whose i'th diagonal entry is $w_i(\beta)$ Then our fitting criterion is equivalent to

$$\mathcal{X}^T \left[\mathcal{W}(\beta) \right] \mathbf{Y} = \mathcal{X}^T \left[\mathcal{W}(\beta) \right] \mathcal{X}\beta.$$

The difficulty in solving this is that $w_i(\beta)$ depend on β, so we can't just solve a linear system in β. We could use the following strategy. Use $\mathcal{W}$ tries to downweight points that are suspiciously inconsistent with our current estimate of β, then update

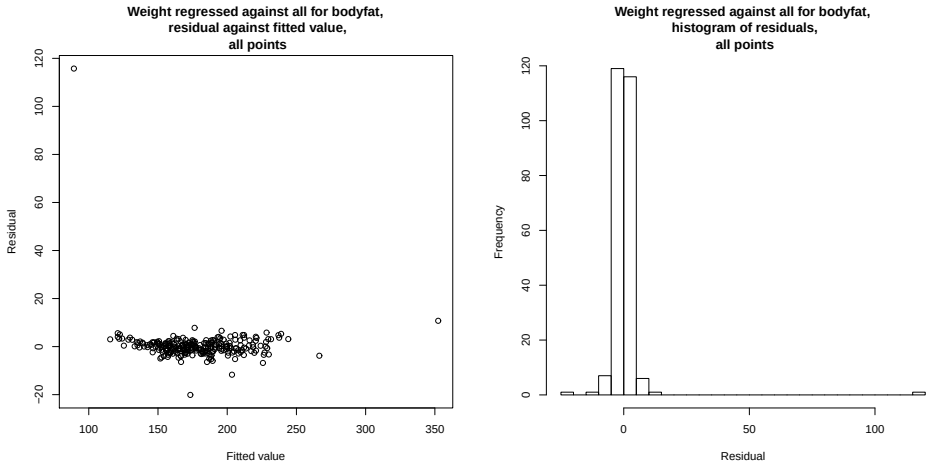

FIGURE 8.3: *A robust linear regression of weight against all variables for the bodyfat dataset, using the Huber loss and all data points. On the* **left**, *residual plotted against fitted value (the residual is not standardized). Notice that there are some points with very large residual, but most have much smaller residual; this wouldn't happen with a squared error. On the* **right**, *a histogram of the residual. If one ignores the extreme residual values, this looks normal. The robust process has been able to discount the effect of the outliers, without us needing to identify and reject outliers by hand.*

β using those weights. The strategy is known as **iteratively reweighted least squares**, and is very effective.

We assume we have an estimate of the correct parameters $\hat{\beta}^{(n)}$, and consider updating it to $\hat{\beta}^{(n+1)}$. We compute

$$w_i^{(n)} = w_i(\hat{\beta}^{(n)}) = \frac{\frac{\partial \rho}{\partial u}(y_i - \mathbf{x}_i^T\beta^{(n)}; \sigma)}{y_i - \mathbf{x}_i^T\hat{\beta}^{(n)}}.$$

We then estimate $\hat{\beta}^{(n+1)}$ by solving

$$\mathcal{X}^T\mathcal{W}^{(n)}\mathbf{Y} = \mathcal{X}^T\mathcal{W}^{(n)}\mathcal{X}\hat{\beta}^{(n+1)}.$$

The key to this algorithm is finding good start points for the iteration. One strategy is randomized search. We select a small subset of points uniformly at random, and fit some $\hat{\beta}$ to these points, then use the result as a start point. If we do this often enough, one of the start points will be an estimate that is not contaminated by outliers.

8.2.2 Scale for M-Estimators

The estimators require a sensible estimate of σ, which is often referred to as **scale**. Typically, the scale estimate is supplied at each iteration of the solution method.

One reasonable estimate is the **MAD** or **median absolute deviation**, given by

$$\sigma^{(n)} = 1.4826 \text{ median}_i | r_i^{(n)}(x_i; \hat{\beta}^{(n-1)}) |.$$

Another a popular estimate of scale is obtained with **Huber's proposal 2** (that is what everyone calls it!). Choose some constant $k_1 > 0$, and define $\Xi(u) = \min(|u|, k_1)^2$. Now solve the following equation for σ:

$$\sum_i \Xi(\frac{r_i^{(n)}(x_i; \hat{\beta}^{(n-1)})}{\sigma}) = Nk_2$$

where k_2 is another constant, usually chosen so that the estimator gives the right answer for a normal distribution (exercises). This equation needs to be solved with an iterative method; the MAD estimate is the usual start point. R provides `hubers`, which will compute this estimate of scale (and figures out k_2 for itself). The choice of k_1 depends somewhat on how contaminated you expect your data to be. As $k_1 \to \infty$, this estimate becomes more like the standard deviation of the data.

8.3 GENERALIZED LINEAR MODELS

We have used a linear regression to predict a value from a feature vector, but implicitly have assumed that this value is a real number. Other cases are important, and some of them can be dealt with using quite simple generalizations of linear regression. When we derived linear regression, I said one way to think about the model was

$$y = \mathbf{x}^T\beta + \xi$$

where ξ was a normal random variable with zero mean and variance σ_ξ^2. Another way to write this is to think of y as the value of a random variable Y. In this case, Y has mean $\mathbf{x}^T\beta$ and variance σ_ξ^2. This can be written as

$$Y \sim N(\mathbf{x}^T\beta, \sigma_\xi^2).$$

This offers a fruitful way to generalize: we replace the normal distribution with some other parametric distribution, and predict the parameter using $\mathbf{x}^T\beta$. Two examples are particularly important.

8.3.1 Logistic Regression

Assume the y values can be either 0 or 1. You could think of this as a two class classification problem, and deal with it using an SVM. There are sometimes advantages to seeing it as a regression problem. One is that we get to see a new classification method that explicitly models class posteriors, which an SVM doesn't do.

We build the model by asserting that the y values represent a draw from a Bernoulli random variable (definition below, for those who have forgotten). The parameter of this random variable is θ, the probability of getting a one. But $0 \leq \theta \leq 1$, so we can't just model θ as $\mathbf{x}^T\beta$. We will choose some **link function** g so that we can model $g(\theta)$ as $\mathbf{x}^T\beta$. This means that, in this case, g must map the interval between 0 and 1 to the whole line, and must be 1-1. The link function maps θ to $\mathbf{x}^T\beta$; the direction of the map is chosen by convention. We build our model by asserting that $g(\theta) = \mathbf{x}^T\beta$.

Definition: 8.1 *Bernoulli random variable*

A Bernoulli random variable with parameter θ takes the value 1 with probability θ and 0 with probability $1-\theta$. This is a model for a coin toss, among other things.

Notice that, for a Bernoulli random variable, we have that

$$\log\left[\frac{P(y=1|\theta)}{P(y=0|\theta)}\right] = \log\left[\frac{\theta}{1-\theta}\right]$$

and the **logit function** $g(u) = \log\left[\frac{u}{1-u}\right]$ meets our needs for a link function (it maps the interval between 0 and 1 to the whole line, and is 1-1). This means we can build our model by asserting that

$$\log\left[\frac{P(y=1|\mathbf{x})}{P(y=0|\mathbf{x})}\right] = \mathbf{x}^T\beta$$

then solving for the β that maximizes the log-likelihood of the data. Simple manipulation yields

$$P(y=1|\mathbf{x}) = \frac{e^{\mathbf{x}^T\beta}}{1+e^{\mathbf{x}^T\beta}} \text{ and } P(y=0|\mathbf{x}) = \frac{1}{1+e^{\mathbf{x}^T\beta}}.$$

In turn, this means the log-likelihood of a dataset will be

$$\mathcal{L}(\beta) = \sum_i \left[\mathbb{I}_{[y=1]}(y_i)\mathbf{x}_i^T\beta - \log\left(1+e^{\mathbf{x}_i^T\beta}\right)\right].$$

You can obtain β from this log-likelihood by gradient ascent (or rather a lot faster by Newton's method, if you know that).

A regression of this form is known as a **logistic regression**. It has the attractive property that it produces estimates of posterior probabilities. Another interesting property is that a logistic regression is a lot like an SVM. To see this, we replace the labels with new ones. Write $\hat{y}_i = 2y_i - 1$; this means that $\hat{y}_i$ takes the values -1 and 1, rather than 0 and 1. Now $\mathbb{I}_{[y=1]}(y_i) = \frac{\hat{y}_i+1}{2}$, so we can write

$$\begin{aligned}
-\mathcal{L}(\beta) &= -\sum_i \left[\frac{\hat{y}_i+1}{2}\mathbf{x}_i^T\beta - \log\left(1+e^{\mathbf{x}_i^T\beta}\right)\right] \\
&= \sum_i \left[\frac{\hat{y}_i+1}{2}\mathbf{x}_i^T\beta - \log\left(1+e^{\mathbf{x}_i^T\beta}\right)\right] \\
&= \sum_i \left[\log\left(\frac{1+e^{\mathbf{x}_i^T\beta}}{e^{\frac{\hat{y}_i+1}{2}\mathbf{x}_i^T\beta}}\right)\right] \\
&= \sum_i \left[\log\left(e^{\frac{-(\hat{y}_i+1)}{2}\mathbf{x}_i^T\beta} + e^{\frac{1-\hat{y}_i}{2}\mathbf{x}_i^T\beta}\right)\right]
\end{aligned}$$

and we can interpret the term in square brackets as a loss function. If you plot it, you will notice that it behaves rather like the hinge loss. When $\hat{y}_i = 1$, if $\mathbf{x}^T\beta$ is positive the loss is very small, but if $\mathbf{x}^T\beta$ is strongly negative, the loss grows linearly in $\mathbf{x}^T\beta$. There is similar behavior when $\hat{y}_i = -1$. The transition is smooth, unlike the hinge loss. Logistic regression should (and does) behave well for the same reasons the SVM behaves well.

Be aware that logistic regression has one annoying quirk. When the data are linearly separable (i.e. there exists some β such that $y_i\mathbf{x}_i^T\beta > 0$ for all data items), logistic regression will behave badly. To see the problem, choose the β that separates the data. Now it is easy to show that increasing the magnitude of β will increase the log likelihood of the data; there isn't any limit. These situations arise fairly seldom in practical data.

8.3.2 Multiclass Logistic Regression

Imagine $y \in [0, 1, \ldots, C-1]$. Then it is natural to model $p(y|\mathbf{x})$ with a discrete probability distribution on these values. This can be specified by choosing $(\theta_0, \theta_1, \ldots, \theta_{C-1})$ where each term is between 0 and 1 and $\sum_i \theta_i = 1$. Our link function will need to map this constrained vector of θ values to a $\Re^{C-1}$. We can do this with a fairly straightforward variant of the logit function, too. Notice that there are $C-1$ probabilities we need to model (the C'th comes from the constraint $\sum_i \theta_i = 1$). We choose one vector β for each probability, and write β_i for the vector used to model θ_i. Then we can write

$$\mathbf{x}^T\beta_i = \log\left(\frac{\theta_i}{1 - \sum_u \theta_u}\right)$$

and this yields the model

$$\begin{aligned}
P(y = 0|\mathbf{x}, \beta) &= \frac{e^{\mathbf{x}^T\beta_0}}{1 + \sum_i e^{\mathbf{x}^T\beta_i}} \\
P(y = 1|\mathbf{x}, \beta) &= \frac{e^{\mathbf{x}^T\beta_1}}{1 + \sum_i e^{\mathbf{x}^T\beta_i}} \\
&\cdots \\
P(y = C-1|\mathbf{x}, \beta) &= \frac{1}{1 + \sum_i e^{\mathbf{x}^T\beta_i}}
\end{aligned}$$

and we would fit this model using maximum likelihood. The likelihood is easy to write out, and gradient descent is a good strategy for actually fitting models.

8.3.3 Regressing Count Data

Now imagine that the y_i values are counts. For example, y_i might have the count of the number of animals caught in a small square centered on $\mathbf{x}_i$ in a study region. As another example, $\mathbf{x}_i$ might be a set of features that represent a customer, and y_i might be the number of times that customer bought a particular product. The natural model for count data is a Poisson model, with parameter θ representing the intensity (reminder below).

Definition: 8.2 *Poisson distribution*

A non-negative, integer valued random variable X has a Poisson distribution when its probability distribution takes the form

$$P(\{X = k\}) = \frac{\theta^k e^{-\theta}}{k!},$$

where $\theta > 0$ is a parameter often known as the **intensity** of the distribution.

Now we need $\theta > 0$. A natural link function is to use

$$\mathbf{x}^T\beta = \log\theta$$

yielding a model

$$P(\{X = k\}) = \frac{e^{k\mathbf{x}^T\beta}e^{-e^{k\mathbf{x}^T\beta}}}{k!}.$$

Now assume we have a dataset. The negative log-likelihood can be written as

$$\begin{aligned} -\mathcal{L}(\beta) &= -\sum_i \log\left(\frac{e^{y_i\mathbf{x}_i^T\beta}e^{-e^{y_i\mathbf{x}_i^T\beta}}}{y_i!}\right) \\ &= -\sum_i \left(y_i\mathbf{x}_i^T\beta - e^{y_i\mathbf{x}_i^T\beta} - \log(y_i!)\right). \end{aligned}$$

There isn't a closed form minimum available, but the log-likelihood is convex, and gradient descent (or Newton's method) are enough to find a minimum. Notice that the $\log(y_i!)$ term isn't relevant to the minimization, and is usually dropped.

8.4 L1 REGULARIZATION AND SPARSE MODELS

8.4.1 Dropping Variables with L1 Regularization

We have a large set of explanatory variables, and we would like to choose a small set that explains most of the variance in the independent variable. We could do this by encouraging β to have many zero entries. In section 7.4.2, we saw we could regularize a regression by adding a term to the cost function that discouraged large values of β. Instead of solving for the value of β that minimized $\sum_i(y_i - \mathbf{x}_i^T\beta)^2 = (\mathbf{y} - \mathcal{X}\beta)^T(\mathbf{y} - \mathcal{X}\beta)$ (which I shall call the **error cost**), we minimized

$$\sum_i (y_i - \mathbf{x}_i^T\beta)^2 + \lambda\beta^T\beta = (\mathbf{y} - \mathcal{X}\beta)^T(\mathbf{y} - \mathcal{X}\beta) + \lambda\beta^T\beta$$

(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 $\hat{\beta}$. Now notice that

$$\beta_k = \frac{b(\beta_{-k})}{(a + \lambda)}.$$

Notice that λ doesn't appear in the numerator. This means that, to force β_k to zero by increasing λ, we may have to make λ arbitrarily large. This is because the improvement in the penalty obtained by going from a small β_k to $\beta_k = 0$ is tiny – the penalty is proportional to β_k^2.

To force some components of β to zero, we need a penalty that grows linearly around zero rather than quadratically. This means we should use the $\mathbf{L}_1$ **norm** of β, given by

$$\| \beta \|_1 = \sum_k | \beta_k |.$$

To choose β, we must now solve

$$(\mathbf{y} - \mathcal{X}\beta)^T(\mathbf{y} - \mathcal{X}\beta) + \lambda \| \beta \|_1$$

for an appropriate choice of λ. An equivalent problem is to solve a constrained minimization problem, where one minimizes

$$(\mathbf{y} - \mathcal{X}\beta)^T(\mathbf{y} - \mathcal{X}\beta) \text{ subject to } \| \beta \|_1 \leq t$$

where t is some value chosen to get a good result, typically by cross-validation. There is a relationship between the choice of t and the choice of λ (with some thought, a smaller t will correspond to a bigger λ) but it isn't worth investigating in any detail.

Actually solving this system is quite involved, because the cost function is not differentiable. You should *not* attempt to use stochastic gradient descent, because this will not compel zeros to appear in $\hat{\beta}$ (exercises). There are several methods, which are beyond our scope. As the value of λ increases, the number of zeros in $\hat{\beta}$ will increase too. We can choose λ in the same way we used for classification; split the training set into a training piece and a validation piece, train for different values of λ, and test the resulting regressions on the validation piece. However, one consequence of modern methods is that we can generate a very good approximation to the path $\hat{\beta}(\lambda)$ for all values of $\lambda \geq 0$ about as easily as we can choose $\hat{\beta}$ for a particular value of λ.

One way to understand the models that result is to look at the behavior of cross-validated error as λ changes. The error is a random variable, random because of the random split. It is a fair model of the error that would occur on a randomly chosen test example (assuming that the training set is "like" the test set, in a way that I do not wish to make precise yet). We could use multiple splits, and average over the splits. Doing so yields both an average error for each value of λ and an estimate of the standard deviation of error. Figure 8.4 shows the

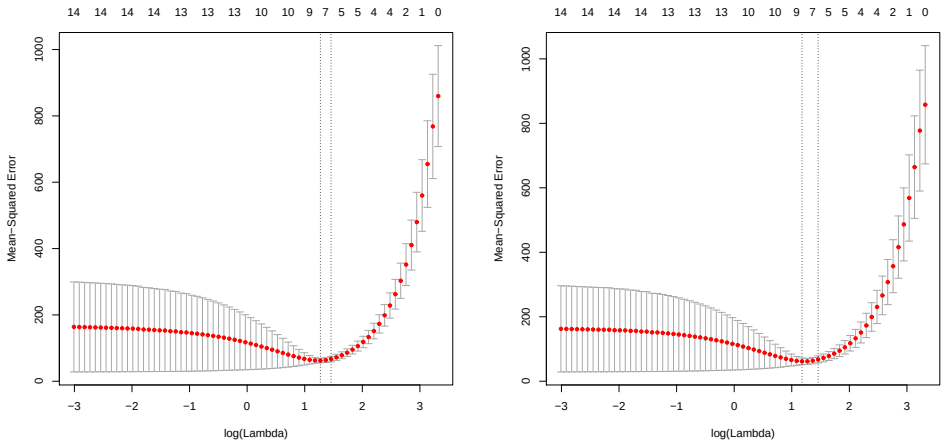

FIGURE 8.4: *Plots of mean-squared error as a function of log regularization parameter (i.e.* $\log \lambda$*) for a regression of weight against all variables for the bodyfat dataset. These plots show mean-squared error averaged over cross-validation folds with a vertical one standard deviation bar. On the* **left***, the plot for the dataset with the six outliers identified in Figure 9.9 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. The penalty ensures that explanatory variables with small coefficients are dropped as* λ *gets bigger.*

result of doing so for two datasets. Again, there is no λ that yields the smallest validation error, because the value of error depends on the random split cross-validation. A reasonable choice of λ lies between the one that yields the smallest error encountered (one vertical line in the plot) and the largest value whose mean error is within one standard deviation of the minimum (the other vertical line in the plot). It is informative to keep track of the number of zeros in $\hat{\beta}$ as a function of λ, and this is shown in Figure 8.4.

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 **??** shows 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. For different values of λ, one gets different solutions for $\hat{\beta}$. When λ is very large, the penalty dominates, and so the norm of $\hat{\beta}$ must be small. In turn, most components of $\hat{\beta}$ are zero. As λ gets smaller, the norm of $\hat{\beta}$ falls and some components of become non-zero. At first glance, the variable whose coefficient grows very large seems important. Look more carefully; this is the last component introduced into the model. But Figure 8.4 implies that the right model has $\log \lambda \approx 1.3$, 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

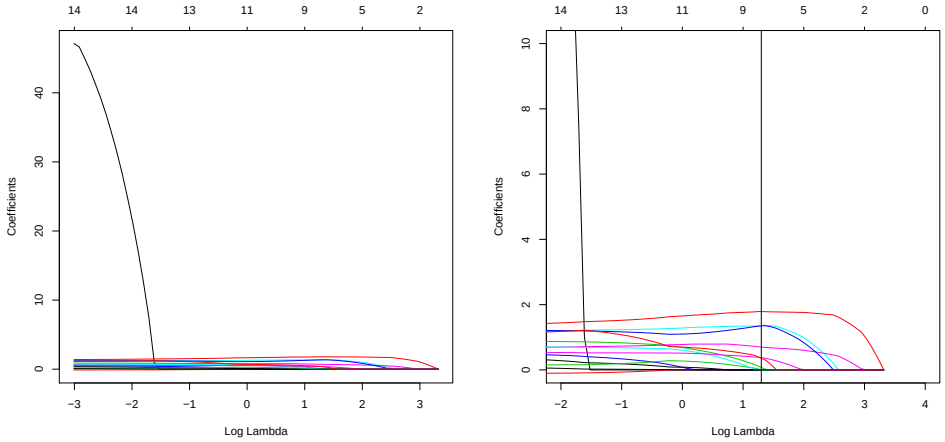

FIGURE 8.5: *Plots of coefficient values as a function of* $\log \lambda$ *for a regression of weight against all variables for the bodyfat dataset, penalised using the* L_1 *norm. In each case, the six outliers identified in Figure 9.9 were removed. On the* **left**, *the plot of the whole path for each coefficient (each curve is one coefficient). On the* **right**, *a detailed version of the plot. The vertical line shows the value of* $\log \lambda$ *the produces the model with smallest cross-validated error (look at Figure 8.4). Notice that the variable that appears to be important, because it would have a large weight with* $\lambda = 0$, *does not appear in this model.*

large number of variables. In the UC Irvine Machine Learning repository, there is a dataset to do with the geographical origin of music (https://archive.ics.uci.edu/ml/datasets/Geographical+Original+of+Music). The dataset was prepared by Fang Zhou, and donors were Fang Zhou, Claire Q, and Ross D. King. Further details appear on that webpage, and in the paper: "Predicting the Geographical Origin of Music" by Fang Zhou, Claire Q and Ross. D. King, which appeared at ICDM in 2014. There are two versions of the dataset. One has 116 explanatory variables (which are various features representing music), and 2 independent variables (the latitude and longitude of the location where the music was collected). Figure 8.6 shows the results of a regression of latitude against the independent variables using L_1 regularization. Notice that the model that achieves the lowest cross-validated prediction error uses only 38 of the 116 variables.

Regularizing a regression with the L_1 norm is sometimes known as a **lasso**. A nuisance feature of the lasso is that, if several explanatory variables are correlated, it will tend to choose one for the model and omit the others (example in exercises). This can lead to models that have worse predictive error than models chosen using the L_2 penalty. One nice feature of good minimization algorithms for the lasso is

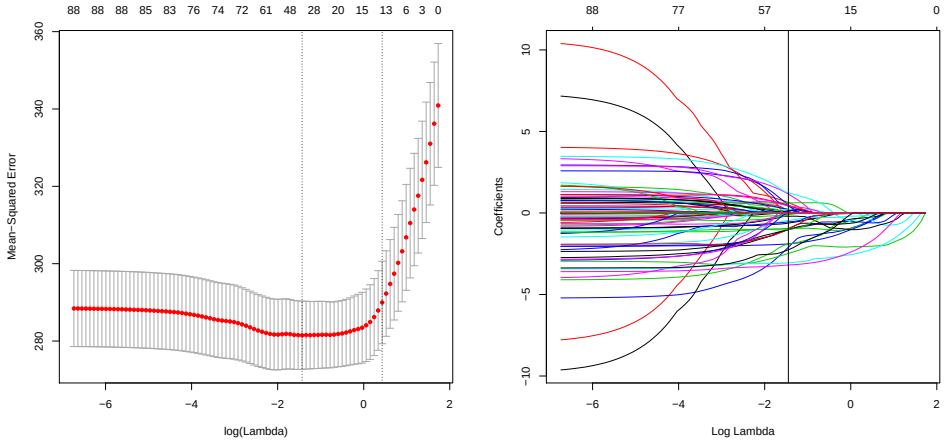

FIGURE 8.6: *Mean-squared error as a function of log regularization parameter (i.e.* $\log \lambda$*) for a regression of latitude against features describing music (details in text), using the dataset at* https://archive.ics.uci.edu/ml/datasets/Geographical+Original+of+Music *and penalized with the* 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.*

that it is easy to use both an L_1 penalty and an L_2 penalty together. One can form

$$\underbrace{\left(\frac{1}{N}\right)\left(\sum_i (y_i - \mathbf{x}_i^T\beta)^2\right)}_{\text{Error}} + \underbrace{\lambda\left(\frac{(1-\alpha)}{2}\|\beta\|_2{}^2 + \alpha\|\beta\|_1\right)}_{\text{Regularizer}}$$

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).

8.5 BAYESIAN REGRESSION

To predict a y value for $\mathbf{x}$ with our current linear regression procedure, we take a training data set, produce an estimate $\hat{\beta}$ of β, then use that estimate to predict the value $\mathbf{x}^T\hat{\beta}$. This is simple, and often effective, but it ignores one important

phenomenon. Our estimate of β might not be right, and errors in $\hat{\beta}$ will result in errors in the predicted y. We would like to account for these errors.

It is common to write $\mathbf{x} \sim N(\mu, \Sigma)$ to mean that $p(\mathbf{y})$ is a normal distribution, with mean μ and covariance Σ; similarly, $\mathbf{y}|\mathbf{x} \sim N(\mu, \Sigma)$ means that $p(\mathbf{y}|\mathbf{x})$ is a normal distribution, with mean μ and covariance Σ. Quite often, you will see something like $\mathbf{y}|\mathbf{x} \sim N(\mathcal{A}\mathbf{x}, \Sigma)$, which means that the mean of $p(\mathbf{y}|\mathbf{x})$ is $\mathcal{A}\mathbf{x}$.

Assume we have N training data items consisting of pairs of vectors and values to be predicted $(\mathbf{x}_i, y_i)$. We stack these as before into a matrix $\mathcal{X}$ and a vector $\mathbf{Y}$. We adopt the model that a y value is obtained by forming some parametric function $f(\mathbf{x}; \beta)$ where β are the parameters, then adding a zero mean normal random variable with *known* standard deviation σ_ξ. This means we can write

$$y|\mathbf{X}, \beta \sim N(f(\mathbf{x}; \beta), \sigma_\xi^2).$$

In the previous chapter, we estimated $\hat{\beta}$ (the best estimate of β we could obtain, given the data), then predicted new values using that. We don't actually know β – we just have a best estimate. Different values of $\hat{\beta}$ might give fairly similar behavior on the training data. This would be an important problem if they gave quite different predictions on test data.

One way to keep track of this problem is to incorporate the uncertainty in our knowledge of β into the prediction. The strategy is as follows. Assume we wish to make a prediction $\hat{y}$ for a new data item $\hat{\mathbf{x}}$. We obtain a posterior on β, which we write $p(\beta|\mathcal{X}, \mathbf{Y})$. We now construct $p(\hat{y}|\beta, \mathbf{x})$. Then the **predictive distribution** $p(\hat{y}|\mathcal{X}, \mathbf{Y}, \hat{\mathbf{x}})$ is

$$p(\hat{y}|\mathcal{X}, \mathbf{Y}, \hat{\mathbf{x}}) = \int p(\hat{y}|\beta, \mathbf{x}) p(\beta|\mathcal{X}, \mathbf{Y}) d\beta.$$

Notice that this predictive distribution is *not* a function of β. Note also it is *not* a prediction of $\hat{y}$ – instead, it is a probability distribution over predicted values. You should think of the predictive distribution in this way. For any particular β, $p(\hat{y}|\beta, \mathbf{x})$ is a distribution over possible values of $\hat{y}$ conditioned on that β. We now take all of these distributions, and form a weighted average, where the weights are $p(\beta|\mathcal{X}, \mathbf{Y})$. So the $p(\hat{y}|\beta, \mathbf{x})$ corresponding to values of β that have a large $p(\beta|\mathcal{X}, \mathbf{Y})$ will tend to dominate. But if $p(\beta|\mathcal{X}, \mathbf{Y})$ has multiple peaks, then $p(\hat{y}|\mathcal{X}, \mathbf{Y}, \hat{\mathbf{x}})$ may very well have multiple peaks, too.

8.5.1 Bayesian Regression with a Normal Prior

Generally, doing the sums for Bayesian regression would take us out of our way. I have provided some notes below, but won't join the dots in detail. Assume that our model is

$$y = \mathbf{x}^T \beta + \xi,$$

where ξ is a normal random variable with mean 0 and variance σ^2. Assume this variance is known, so we can avoid some technical complications. Assume we have a normal prior on β with mean 0 and covariance Λ (we choose this by hand; it's usually a diagonal matrix with the same value in every diagonal entry). Then we can use the tricks, below, to show that

$$P(\beta|\mathcal{X}, \mathbf{Y}, \sigma_\xi, \Lambda)$$

is normal, and

$$\begin{aligned}\beta|\mathcal{X},\mathbf{Y},\sigma_\xi,\Lambda &\sim N(\mu_\beta,\Sigma_\beta)\\ \mu_\beta &= \left(\mathcal{X}^T\mathcal{X}+\sigma_\xi^2\Lambda\right)^{-1}\mathcal{X}\mathbf{Y}\\ \Sigma_\beta &= \sigma_\xi^2\left(\mathcal{X}^T\mathcal{X}+\sigma_\xi^2\Lambda\right)^{-1}.\end{aligned}$$

You can also use these tricks to show that

$$P(y|\mathbf{x},\mathcal{X},\mathbf{Y},\sigma_\xi,\Lambda)$$

is normal, and

$$\begin{aligned}y|\mathbf{x},\mathcal{X},\mathbf{Y},\sigma_\xi,\Lambda &\sim N(\mathbf{x}^T\mu_\beta,\sigma_p^2)\\ \sigma_p^2 &= \sigma_\xi^2+\mathbf{x}^T\Sigma_\beta\mathbf{x}.\end{aligned}$$

It is possible to place a prior on σ_ξ^2 as well, and average over that. Doing so would lead us into unnecessary complications.

8.5.2 Tricks with Normal Distributions

At this point, it is helpful to know some useful tricks with normal distributions. Normal distributions are nice, because many calculations are in fact quite simple. I have collected some useful facts here, and will prove them below, in order. You should remember that these expressions exist, and where to look them up (I find them hard to memorize, and derive them when I need them). The proofs are just careful chasing of terms, but you should likely look at them once, and again remember where to find them.

Useful Facts: 8.1 *Marginalizing a normal distribution*

If $\mathbf{x}^T=(\mathbf{u},\mathbf{v})^T$ is a normal random vector, and $\mathbf{x}\sim N(\mu,\Sigma)$, with mean $\mu=(\mu_u,\mu_v)$ and covariance Σ. Write

$$\begin{pmatrix}\Sigma_{uu} & \Sigma_{uv}\\ \Sigma_{vv} & \Sigma_{vv}\end{pmatrix}=\Sigma$$

(where the blocks correspond to the $\mathbf{u}$, $\mathbf{v}$ decomposition of $\mathbf{x}$). Then

$$\mathbf{u}\sim N(\mu_u,\Sigma_{uu}).$$

Useful Facts: 8.2 *Conditioning jointly normal random variables*

Assume $\mathbf{x}^T = (\mathbf{u}, \mathbf{v})^T$ is a normal random vector, and $\mathbf{x} \sim N(\mu, \Sigma)$, with mean $\mu = (\mu_u, \mu_v)$ and covariance Σ. Write

$$\begin{pmatrix} \Sigma_{uu} & \Sigma_{uv} \\ \Sigma_{vv} & \Sigma_{vv} \end{pmatrix} = \Sigma$$

(where the blocks correspond to the $\mathbf{u}$, $\mathbf{v}$ decomposition of $\mathbf{x}$). Then

$$\mathbf{u}|\mathbf{v} \sim N(\mu_c, \Sigma_c)$$

where

$$\begin{aligned} \Sigma_c &= \left[\Sigma_{uu} - \Sigma_{uv}\Sigma_{vv}^{-1}\Sigma_{uv}^T\right] \\ \mu_c &= \mu_u - \Sigma_{uv}\Sigma_{vv}^{-1}(\mathbf{v} - \mu_v). \end{aligned}$$

Useful Facts: 8.3 *Normal likelihood and normal prior yield normal posterior*

Assume $\mathbf{y}|\mathbf{x} \sim N(\mathcal{H}\mathbf{x}, \Sigma_1)$ and $\mathbf{x} \sim N(\mu_0, \Sigma_0)$. Then

$$\mathbf{x}|\mathbf{y} \sim N(\mu_p, \Sigma_p)$$

where

$$\begin{aligned} \Sigma_p &= \left[\Sigma_0^{-1} + \mathcal{H}^T\Sigma_1^{-1}\mathcal{H}\right]^{-1} \\ \mu_p &= \mu_0 + \Sigma_0\mathcal{H}^T\left[\Sigma_1 + \mathcal{H}\Sigma_0\mathcal{H}^T\right]^{-1}(\mathbf{y} - \mathcal{H}\mu_0) \end{aligned}$$

Normal distributions and quadratic forms: Assume you know that $p(\mathbf{x})$ is normal, and you need to determine the particular normal distribution you are dealing with. Write out the log of the probability distribution; you must have an expression that looks like

$$-2\log p(\mathbf{x}) = \mathbf{x}^T\mathcal{M}\mathbf{x} - 2\mathbf{b}^T\mathbf{x} + K$$

(otherwise, it isn't normal). Now we can determine expressions in the parameters by pattern matching. If $p(\mathbf{x})$ is a normal distribution with mean μ and covariance Σ, then

$$-2\log p(\mathbf{x}) = \mathbf{x}^T\Sigma^{-1}\mathbf{x} - 2\mu^T\Sigma^{-1}\mathbf{x} + C$$

so we must have that $\mathcal{M}^{-1} = \Sigma$ and $\mu = \Sigma\mathbf{b}$. This observation may strike you as trivial, but it is extremely useful.

Marginalizing a normal distribution: Assume that $\mathbf{x}^T = (\mathbf{u}, \mathbf{v})^T$ is a normal random vector, with mean $\mu = (\mu_u, \mu_v)$ and covariance Σ. Write

$$\begin{pmatrix} \Sigma_{uu} & \Sigma_{uv} \\ \Sigma_{vv} & \Sigma_{vv} \end{pmatrix} = \Sigma \text{ and } \begin{pmatrix} \mathcal{A} & \mathcal{B}^T \\ \mathcal{B} & \mathcal{C} \end{pmatrix} = \Sigma^{-1}$$

(where the blocks correspond to the $\mathbf{u}$, $\mathbf{v}$ decomposition of $\mathbf{x}$). I claim that

$$p(\mathbf{u}) = \int p(\mathbf{u}, \mathbf{v}) d\mathbf{v}$$

is normal with mean μ_u and covariance Σ_{uu}. Assume also that $\mathbf{v}$ is d_v dimensional. I will do this example in detail, as a model for the next ones. First, we will change variables, writing $\mathbf{m} = (\mathbf{u} - \mu_u)$ and $\mathbf{n} = (\mathbf{v} - \mu_v)$. We then seek

$$p(\mathbf{m}) = \int_D p(\mathbf{m}, \mathbf{n}) d\mathbf{n}$$

where D is all values of $\mathbf{n}$, so R^{d_v} (you should check nothing interesting has happened with the change of variables). Now we have

$$\begin{aligned} -2\log p(\mathbf{m}, \mathbf{n}) &= (\mathbf{x} - \mu)^T\Sigma^{-1}(\mathbf{x} - \mu) + K \\ &= (\mathbf{u} - \mu_u)^T\mathcal{A}(\mathbf{u} - \mu_u) + 2(\mathbf{u} - \mu_u)^T\mathcal{B}^T(\mathbf{v} - \mu_v) + (\mathbf{v} - \mu_v)^T\mathcal{C}(\mathbf{v} - \mu_v) \\ &= \mathbf{m}^T\mathcal{A}\mathbf{m} + 2\mathbf{m}^T\mathcal{B}^T\mathbf{n} + \mathbf{n}^T\mathcal{C}\mathbf{n} \\ &= \mathbf{m}^T(\mathcal{A} - \mathcal{B}^T\mathcal{C}^{-1}\mathcal{B})\mathbf{m} + (\mathbf{n} - \mathcal{C}^{-1}\mathcal{B}\mathbf{m})^T\mathcal{C}(\mathbf{n} - \mathcal{C}^{-1}\mathcal{B}\mathbf{m}). \end{aligned}$$

(where you should check the last step, which I got by completing the square; you could expand out the expression). Now write $p(\mathbf{m}, \mathbf{n}) = f(\mathbf{m})g(\mathbf{n} - \mathcal{C}^{-1}\mathcal{B}\mathbf{m})$ where

$$-2\log g(\mathbf{u}) = \mathbf{u}^T\mathcal{C}\mathbf{u}$$

I claim $\int g(\mathbf{n} - \mathcal{C}^{-1}\mathcal{B}\mathbf{m})d\mathbf{n}$ does not depend on $\mathbf{m}$. Choose some particular value of $\mathbf{m}$, and compute the integral for that value, recalling that the domain of the integral is over all values of $\mathbf{n}$ – the domain does not change with $\mathbf{m}$. Different choices of $\mathbf{m}$ shift the peak of the integrand, but do not change the value of the integral. Equivalently, for some fixed $\mathbf{m} = \mathbf{m}_0$, note that $\int_D g(\mathbf{n} - \mathcal{C}^{-1}\mathcal{B}\mathbf{m}_0)d\mathbf{n} = \int_D g(\mathbf{w})d\mathbf{w}$, where $\mathbf{w} = \mathbf{n} - \mathcal{C}^{-1}\mathcal{B}\mathbf{m}_0$. This is the same for all $\mathbf{m}_0$, so $\int g(\mathbf{n} - \mathcal{C}^{-1}\mathcal{B}\mathbf{m})d\mathbf{n}$ does not depend on $\mathbf{m}$. As a result

$$-2\log p(\mathbf{u}) = (\mathbf{u} - \mu_u)^T(\mathcal{A} - \mathcal{B}^T\mathcal{C}^{-1}\mathcal{B})(\mathbf{u} - \mu_u) + K$$

so $p(\mathbf{u})$ is normal with mean μ_u. Obtaining its covariance requires a very little more work. Write

$$\Sigma = \begin{pmatrix} \Sigma_{uu} & \Sigma_{vu}^T \\ \Sigma_{vu} & \Sigma_{vv} \end{pmatrix}$$

and multiply this by Σ^{-1}, yielding

$$\begin{pmatrix} \mathcal{A}\Sigma_{uu} + \mathcal{B}^T\Sigma vu & \mathcal{A}\Sigma_{vu}^T + \mathcal{B}^T\Sigma_{vv} \\ \mathcal{B}\Sigma_{uu} + \mathcal{C}\Sigma vu & \mathcal{B}\Sigma_{vu}^T + \mathcal{C}\Sigma_{vv} \end{pmatrix} = \Sigma = \begin{pmatrix} \mathcal{I} & 0 \\ 0 & \mathcal{I} \end{pmatrix}$$

from which you can derive $(\mathcal{A} - \mathcal{B}^T\mathcal{C}^{-1}\mathcal{B}) = \Sigma_{uu}$.

Conditioning jointly normal random variables: Assume $\mathbf{x}^T = (\mathbf{u}, \mathbf{v})^T$ is a normal random vector, and $\mathbf{x} \sim N(\mu, \Sigma)$, with mean $\mu = (\mu_u, \mu_v)$ and covariance Σ. Write

$$\begin{pmatrix} \Sigma_{uu} & \Sigma_{uv} \\ \Sigma_{vv} & \Sigma_{vv} \end{pmatrix} = \Sigma \text{ and } \begin{pmatrix} \mathcal{A} & \mathcal{B}^T \\ \mathcal{B} & \mathcal{C} \end{pmatrix} = \Sigma^{-1}$$

(where the blocks correspond to the $\mathbf{u}$, $\mathbf{v}$ decomposition of $\mathbf{x}$). Now

$$-2\log p(\mathbf{u}|\mathbf{v}) = \mathbf{u}^T\mathcal{A}\mathbf{u} - 2(\mu_u^T\mathcal{A} + [\mathbf{v} - \mu_v]^T\mathcal{B})\mathbf{u}) + D$$

so the covariance is $\Sigma_c = \mathcal{A}$ and the mean is $\mu_c = (\mu_u + \mathcal{A}^{-1}\mathcal{B}^T[\mathbf{v} - \mu_v])$. Now

$$\begin{pmatrix} \mathcal{A}\Sigma_{uu} + \mathcal{B}^T\Sigma vu & \mathcal{A}\Sigma_{vu}^T + \mathcal{B}^T\Sigma_{vv} \\ \mathcal{B}\Sigma_{uu} + \mathcal{C}\Sigma vu & \mathcal{B}\Sigma_{vu}^T + \mathcal{C}\Sigma_{vv} \end{pmatrix} = \Sigma = \begin{pmatrix} \mathcal{I} & 0 \\ 0 & \mathcal{I} \end{pmatrix}$$

from which we can recover $\mathcal{B}^T = -\Sigma_{vv}^{-1}\Sigma_{uv}^T\mathcal{A}$, and so $\left[\Sigma_{uu} - \Sigma_{uv}\Sigma_{vv}^{-1}\Sigma_{uv}^T\right]\mathcal{A} = \mathcal{I}$ and $\mathcal{A}^{-1}\mathcal{B}^T = -\Sigma_{vv}^{-1}\Sigma_{uv}^T$. In turn, this yields

$$\begin{aligned} \Sigma_c &= \left[\Sigma_{uu} - \Sigma_{uv}\Sigma_{vv}^{-1}\Sigma_{uv}^T\right] \\ \mu_c &= \mu_u - \Sigma_{uv}\Sigma_{vv}^{-1}(\mathbf{v} - \mu_v). \end{aligned}$$

Normal likelihood and normal prior yield normal posterior: Assume $\mathbf{y}|\mathbf{x} \sim N(\mathcal{H}\mathbf{x}, \Sigma_1)$ and $\mathbf{x} \sim N(\mu_0, \Sigma_0)$. Then $p(\mathbf{x}|\mathbf{y})$ is normal. We write

$$\mathbf{x}|\mathbf{y} \sim N(\mu_p, \Sigma_p).$$

Now

$$\begin{aligned} -2\log p(\mathbf{x}|\mathbf{y}) &= (\mathcal{H}\mathbf{x} - \mathbf{y})^T\Sigma_1^{-1}(\mathcal{H}\mathbf{x} - \mathbf{y}) + (\mathbf{x} - \mu_0)^T\Sigma_0^{-1}(\mathbf{x} - \mu_0) \\ &= \mathbf{x}^T(\mathcal{H}^T\Sigma_1^{-1}\mathcal{H} + \Sigma_0^{-1})\mathbf{x} - 2(\mathcal{H}\Sigma_1^{-1}\mathbf{y} + \Sigma_0^{-1}\mu_0)^T\mathbf{x} + \text{terms not involving } \mathbf{x}. \end{aligned}$$

This means that

$$\Sigma_p = (\mathcal{H}^T\Sigma_1^{-1}\mathcal{H} + \Sigma_0^{-1})^{-1}.$$

An expression for μ_p takes a bit more work. We have

$$\mu_p = \left[\mathcal{H}^T\Sigma_1^{-1}\mathcal{H} + \Sigma_0^{-1}\right]^{-1}(\mathcal{H}\Sigma_1^{-1}\mathbf{y} + \Sigma_0^{-1}\mu_0)$$

but this could do with simplifying. The **Woodbury matrix identity** gives

$$(\mathcal{A} + \mathcal{U}\mathcal{C}\mathcal{V})^{-1} = \mathcal{A}^{-1} - \mathcal{A}^{-1}\mathcal{U}(\mathcal{C}^{-1} + \mathcal{V}\mathcal{A}^{-1}\mathcal{U})^{-1}\mathcal{V}\mathcal{A}^{-1}$$

(by the way, this is useful, but I can never remember it; you should remember how to find it, though). Applying this identity gives

$$\mu_p = \left[\Sigma_0 - \Sigma_0\mathcal{H}^T\left(\Sigma_1 + \mathcal{H}\Sigma_0\mathcal{H}^T\right)^{-1}\mathcal{H}\Sigma_0\right]^{-1}(\mathcal{H}\Sigma_1^{-1}\mathbf{y} + \Sigma_0^{-1}\mu_0).$$

Write $\mathcal{K} = \Sigma_0 \mathcal{H}^T \left[\mathcal{H}\Sigma_0\mathcal{H}^T + \Sigma_1\right]^{-1}$ (this is known as the **Kalman gain matrix**), and notice that $\mathcal{K}\left[\mathcal{I} + \mathcal{H}\Sigma_0\mathcal{H}^T\Sigma_1^{-1}\right] = \Sigma_0\mathcal{H}^T\Sigma_1^{-1}$. Then we get

$$\mu_p = \mu_0 + \mathcal{K}(\mathbf{y} - \mathcal{H}\mu_0)$$

and you can derive

$$\Sigma_p = (\mathcal{I} - \mathcal{K}\mathcal{H}))\,\Sigma_0$$

in the same way (which gives a small hint of the importance of $\mathcal{K}$)

8.5.3 Bayesian Regression - Overview

8.5.4 Bayesian Regression with Everything Normal

The primary problem here is doing the sums. I will make some simplifications. I assume that $\beta \sim N(\mu_0, \Sigma_0)$, and that σ_n^2 is *known*. Unknown σ_n^2 can be dealt with, but complicate the treatment significantly without necessarily making anything much better. I assume that $f(\mathbf{x};\beta) = \mathbf{x}^T\beta$, the form that we are accustomed to. This means we adopt the model that a y value is obtained by forming a linear prediction with an (unknown) β, then adding a zero mean normal random variable with *known* standard deviation σ_n. This means we can write

$$\mathbf{Y}|\mathcal{X}, \beta \sim N(\mathcal{X}\beta, \sigma_n^2).$$

The main reason for doing all this is that we will end up with a predictive distribution that is normal.

First, we compute the posterior on β. We have

$$\mathbf{Y}|\mathcal{X}, \beta \sim N(\mathcal{X}\beta, \sigma_n^2\mathcal{I})$$

so $\beta|\mathbf{Y}, \mathcal{X} \sim N(\mu_1, \Sigma_1)$ where (by plugging into the expressions for a posterior, above), we obtain

$$\begin{aligned}
\Sigma_1 &= \left(\frac{\mathcal{X}^T\mathcal{X}}{\sigma_n^2} + \Sigma_0^{-1}\right)^{-1} \\
\mu_1 &= \Sigma_1\left(\Sigma_0^{-1}\mu_0 + \frac{1}{\sigma_n^2}\mathcal{X}^T\mathcal{Y}\right).
\end{aligned}$$

Next, we need $p(\hat{y}|\beta, \hat{x}, \mathcal{X}, \mathbf{Y}$. We assume the same prediction process applies to $\hat{x}$ as to other points (the reason we have an issue is we don't know β, rather than the model changed). This yields

$$\hat{y}|\hat{x}, \beta, \mathcal{X}, \mathbf{Y} \sim N(\hat{x}^T\beta, \sigma_n^2\mathcal{I}).$$

The last step is, by far, the nastiest. I will plod through this with some care. We have that

$$-2\log p(\hat{y}, \beta|\mathbf{X}, \mathbf{Y}, \hat{x}) = (\beta - \mu_1)^T\Sigma_1^{-1}(\beta - \mu_1) + \frac{(\hat{y} - \hat{x}^T\beta)^2}{\sigma_n^2}$$

which I can write as

$$-2\log p(\hat{y},\beta|\mathbf{X},\mathbf{Y},\hat{x}) = \left(\hat{y}-0,\beta-\Sigma_1^{-1}\mu_1\right)\left(\begin{array}{cc}\frac{1}{\sigma_{n_x}^2} & -\frac{\hat{x}^T}{\sigma_n^2}\\ -\frac{\hat{x}}{\sigma_n^2} & \Sigma_1^{-1}\end{array}\right)\left(\begin{array}{c}\hat{y}-0\\ \beta-\Sigma_1^{-1}\end{array}\right).$$

For convenience, write Σ_b for covariance of this distribution and μ_b for its mean. Write

$$\Sigma_b = \left[\begin{array}{cc} a & \mathbf{b}^T \\ \mathbf{b} & \mathcal{C}\end{array}\right]$$

and

$$\mu_b = \left[\begin{array}{c}\mu_{\hat{y}}\\ \mu_\beta\end{array}\right].$$

We want $p(\hat{y}|\hat{\mathbf{x}},\mathcal{X},\mathbf{Y})$, which is normal, with mean $\mu_{\hat{y}}$ and variance $\sigma_{\hat{y}}^2$. Now we know that, to compute the marginalized distribution, we need to figure out $\mu_{\hat{y}}$ and $a=\sigma_{\hat{y}}^2$. But

$$\left[\begin{array}{cc}\frac{1}{\sigma_{n_x}^2} & -\frac{\hat{x}^T}{\sigma_n^2}\\ -\frac{\hat{x}}{\sigma_n^2} & \Sigma_1^{-1}\end{array}\right]\left[\begin{array}{cc} a & \mathbf{b}^T \\ \mathbf{b} & \mathcal{C}\end{array}\right] = \left[\begin{array}{cc}\mathcal{I} & 0\\ 0 & \mathcal{I}\end{array}\right]$$

so we have

$$\begin{aligned}\frac{a}{\sigma_n^2} - \frac{\hat{\mathbf{x}}^T\mathbf{b}}{\sigma_n^2} &= 1\\ -\frac{a\hat{\mathbf{x}}}{\sigma_n^2} + \Sigma_1^{-1}\mathbf{b} &= 0\\ \mathbf{b}^T\Sigma_1^{-1}\mu_1 &= \mu_{\hat{y}}\end{aligned}$$

yielding

$$\begin{aligned}\sigma_{\hat{y}}^2 &= \frac{\sigma_n^2}{\sigma_n^2+\hat{x}^T\Sigma_1\hat{x}}\\ \mu_{\hat{y}} &= \hat{\mathbf{x}}^T\beta\end{aligned}$$

8.6 YOU SHOULD

8.6.1 remember:

New term: irreducible error . . . 173
New term: bias . . . 173
New term: variance . . . 173
New term: AIC . . . 175
New term: BIC . . . 176
New term: forward stagewise regression . . . 178
New term: Backward stagewise regression . . . 178
New term: robust regression . . . 180
New term: Huber loss . . . 180
New term: scale . . . 180
New term: inlier . . . 180
New term: iteratively reweighted least squares . . . 182
New term: MAD . . . 183
New term: median absolute deviation . . . 183
New term: Huber's proposal 2 . . . 183
New term: link function . . . 183
Definition: Bernoulli random variable . . . 184
New term: logit function . . . 184
New term: logistic regression . . . 184
New term: intensity . . . 186
Definition: Poisson distribution . . . 186
New term: error cost . . . 186
New term: L2 regularized error . . . 186
New term: lasso . . . 189
New term: elastic net . . . 190
New term: predictive distribution . . . 191
Useful facts: Marginalizing a normal distribution . . . 192
Useful facts: Conditioning jointly normal random variables . . . 193
Useful facts: Normal likelihood and normal prior yield normal posterior193
New term: Woodbury matrix identity . . . 195
New term: Kalman gain matrix . . . 196

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 8.1: *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

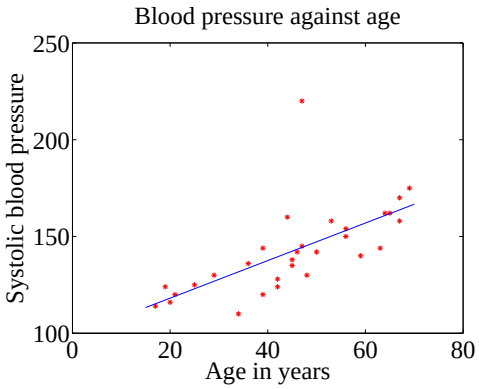

FIGURE 8.7: *A regression of blood pressure against age, for 30 data points.*

8.1. Figure 8.7 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)?

8.2. 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) Show that $r = \mathsf{mean}\,(\{(x - \mathsf{mean}\,(\{x\}))(y - \mathsf{mean}\,(\{y\}))\})/(\mathsf{std}\,(x)\mathsf{std}\,(y))$.

(b) Now write $s = \frac{\mathsf{std}(y)}{\mathsf{std}(x)}$. Now 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

$$sr(x_0 - \mathsf{mean}\,(\{x\})) + \mathsf{mean}\,(\{y\}).$$

(c) Show that $sr = \mathsf{mean}\,(\{(xy)\}) - \mathsf{mean}\,(\{x\})\mathsf{mean}\,(\{y\})$.

(d) 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

$$\mathcal{X}^T\mathcal{X} = N \begin{pmatrix} \mathsf{mean}\left(\left\{x^2\right\}\right) & \mathsf{mean}\,(\{x\}) \\ \mathsf{mean}\,(\{x\}) & 1 \end{pmatrix}$$

(e) Now show that $\mathsf{var}\left(\{x\}\right) = \mathsf{mean}\left(\left\{(x - \mathsf{mean}\left(\{x\}\right))^2\right\}\right) = \mathsf{mean}\left(\left\{x^2\right\}\right) - \mathsf{mean}\left(\{x\}\right)^2$.

(f) Now show that $\mathsf{std}\left(x\right)\mathsf{std}\left(y\right)\mathsf{corr}\left(\{(x, y)\}\right) = \mathsf{mean}\left(\{(x - \mathsf{mean}\left(\{x\}\right))(y - \mathsf{mean}\left(\{y\}\right))\}\right)$.

CHAPTER 9

Non-Parametric Regression

9.1 MODELLING WITH BUMPS

For many data sets, linear models just don't fit very well. However, we expect that points near a given $\mathbf{x}$ will have similar y values. We can exploit this observation in a variety of ways.

9.1.1 Scattered Data: Smoothing and Interpolation

*** 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. 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.

Although this problem is very like regression, there is an important difference: we are interested only in the predicted value at each point, rather than in the conditional distribution. Typical methods for dealing with this problem are very like regression methods, but typically the probabilistic infrastructure required to predict variances, standard errors, and the like are not developed. Interpolation and smoothing problems in one dimension have the remarkable 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.

Now think about the function y that we wish to interpolate, and assume that $\mathbf{x}$ is "reasonably" scaled, meaning that distances between points are a good guide to their similarity. There are several ways to achieve this. We could whiten the points (section 9.9), or we could use our knowledge of the underlying problem to scale the different features relative to one another. Once we have done this properly, we expect 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 expect that the dependency should decline with increasing $\|\mathbf{x} - \mathbf{x}_i\|$. In most problems, we don't know how quickly the weights should decline with increasing distance, and it is usual to have a scaling parameter to handle this. The scaling parameter will need to be selected.

All this suggests constructing an interpolate using **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.

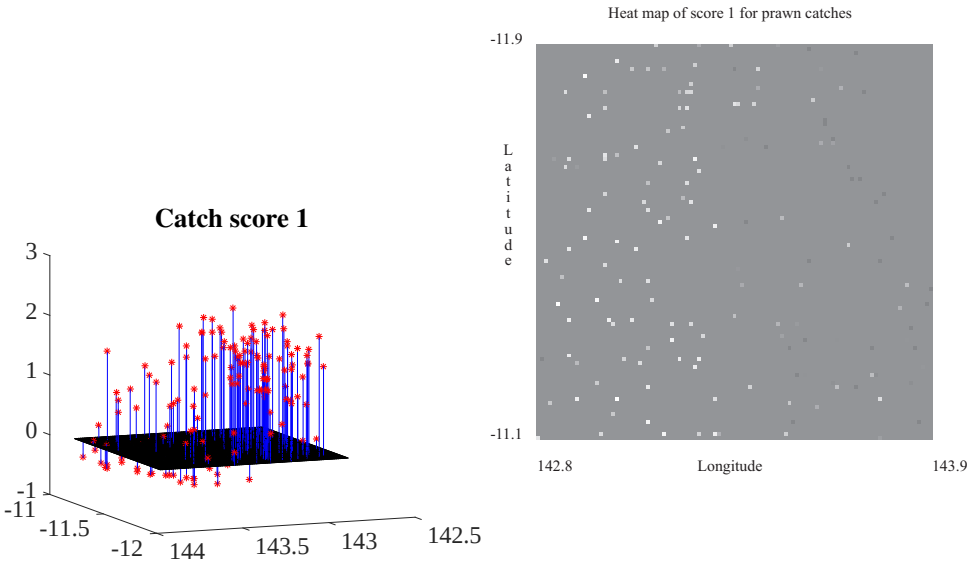

FIGURE 9.1: *A dataset recording scores of prawn trawls around the Great Barrier Reef, from http://www.statsci.org/data/oz/reef.html. There are two scores; this is score 1. On the* **left** *I have plotted the data as a 3D scatter plot. This form of plot isn't usually very successful, though it helps 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.*

- **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**, $K(u) = \frac{1}{\exp -u + \exp u + 2}$. 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.

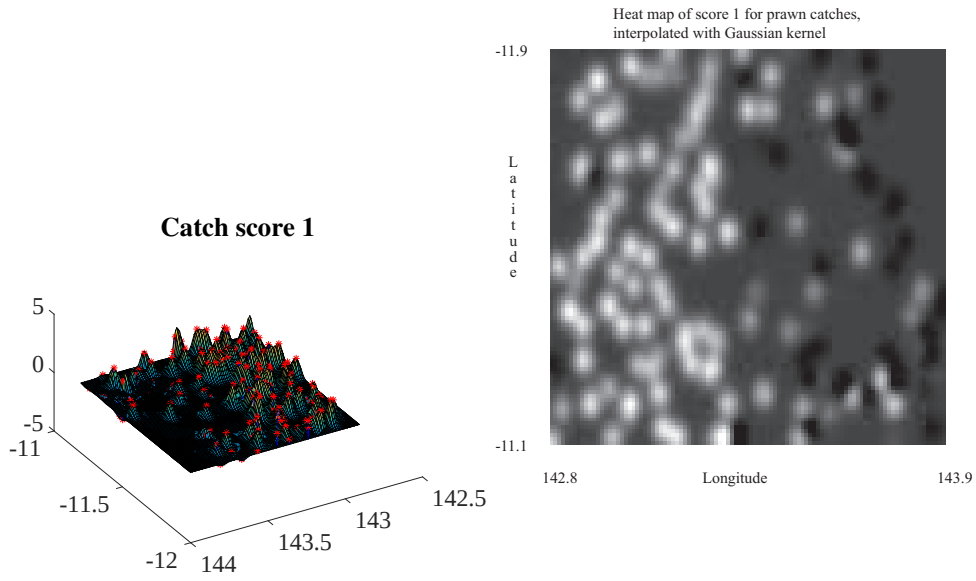

FIGURE 9.2: *The prawn data of figure 9.1, interpolated with radial basis functions (in this case, a Gaussian kernel) with scale chosen by cross-validation. On the* **left**, *a surface shown with the 3D scatter plot. On the* **right**, *a heat map. I ignored the curvature of the earth, small at this scale, when computing distances between points. This figure is a good example of why interpolation is usually not what one wants to do (text).*

We choose a kernel function K, then build a function

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

where $\mathbf{b}_j$ is a set of R points *which don't have to be training points*. These are sometimes referred to as **base points**. You can think of this process as placing a weighted bump at each base point. Consider the values that this function takes at the training points $\mathbf{x}_i$. We have

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

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

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

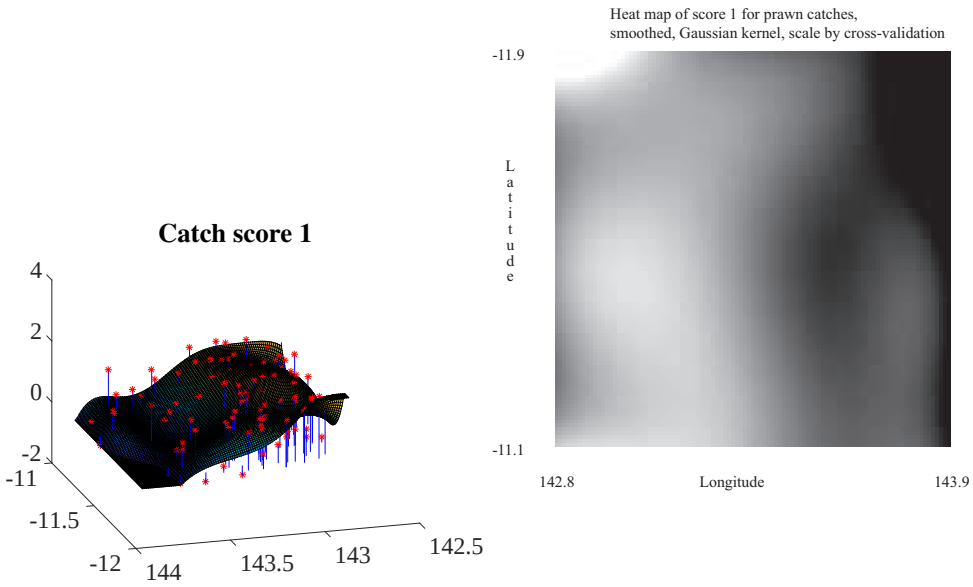

FIGURE 9.3: *The prawn data of figure 9.1, smoothed with radial basis functions (in this case, a Gaussian kernel) with scale chosen by cross-validation. On the* **left**, *a surface shown with the 3D scatter plot. On the* **right**, *a heat map. I ignored the curvature of the earth, small at this scale, 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 spoat. 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 *lambda* to be small enough to make the linear algebra work ($1e-9$ works for me), and ignore it.

As Figure **??** 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.

Now look at Figure 9.3. 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.

9.1.2 Density Estimation

One specialized application of kernel functions related to smoothing is **density estimation**. Here we have a set of N data points $\mathbf{x}_i$ which we believe to be IID samples from some $p(X)$, and we wish to estimate the probability density function $p(X)$. In the case that we have a parametric model for $p(X)$, we could do so by estimating the parameters with (say) maximum likelihood. But we may not have such a model, or we may have data that does not comfortably conform to any model.

A natural, and important, model is to place probability $1/N$ at each data point, and zero elsewhere. This is sometimes called an **empirical distribution**. However, this model implies that we can only ever see the values we have already seen, which is often implausible or incovenient. We should like to "smooth" this model. If the $\mathbf{x}_i$ have low enough dimension, we can construct a density estimate with kernels in a straightforward way.

Recall that a kernel function is non-negative, and has the property that

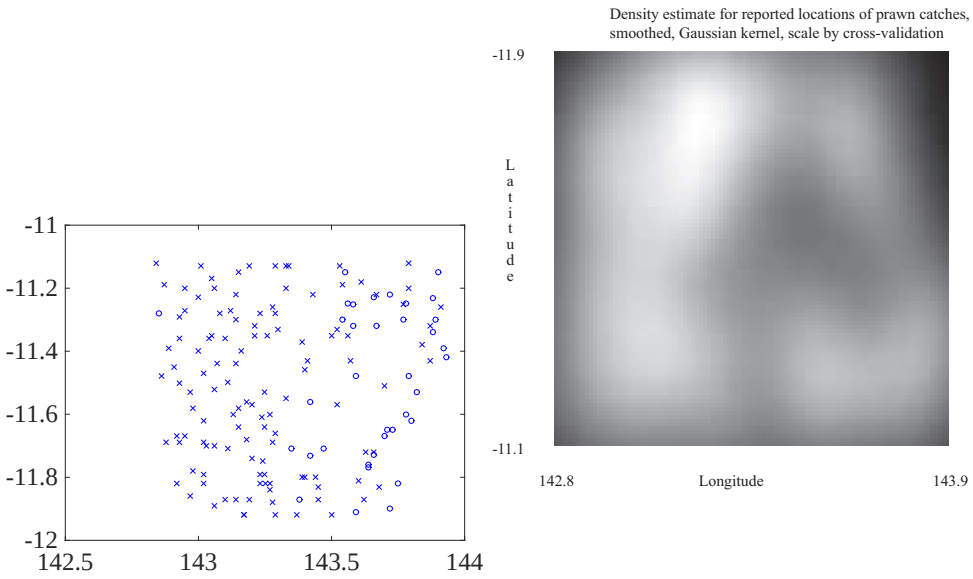

FIGURE 9.4: *The prawn data of figure 9.1, now shown on the* **left** *as a scatter plot of locations from which scores were reported. The 'o's correspond to negative scores, and the '+'s to positive scores. On the* **right**, *a density estimate of the probability distribution from which the fishing 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.*

$\int_{-\infty}^{\infty} K(u)du = 1$. This means that if

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

we have

$$\int_{-\infty}^{\infty} y(\mathbf{x})d\mathbf{x} = \sum_{j=1}^{R} a_j.$$

Now imagine we choose a basepoint at each data point, and we choose $a_j = 1/N$ for all j. The resulting function is non-negative, and integrates to one, so can be seen as a probability density function. We are placing a bump function of scale h, weighted by $1/N$ on top of each data point. If there are many data points close together, these bump functions will tend to reinforce one another and the resulting function will be large in such regions. The function will also be small in gaps between data points that are large compared to h. The resulting model captures the need to (a) represent the data and (b) allow values that have not been observed to have non-zero probability.

Choosing h using cross-validation is straightforward. We choose the h that maximises the log-likelihood of omitted "test" data, averaged over folds. There is

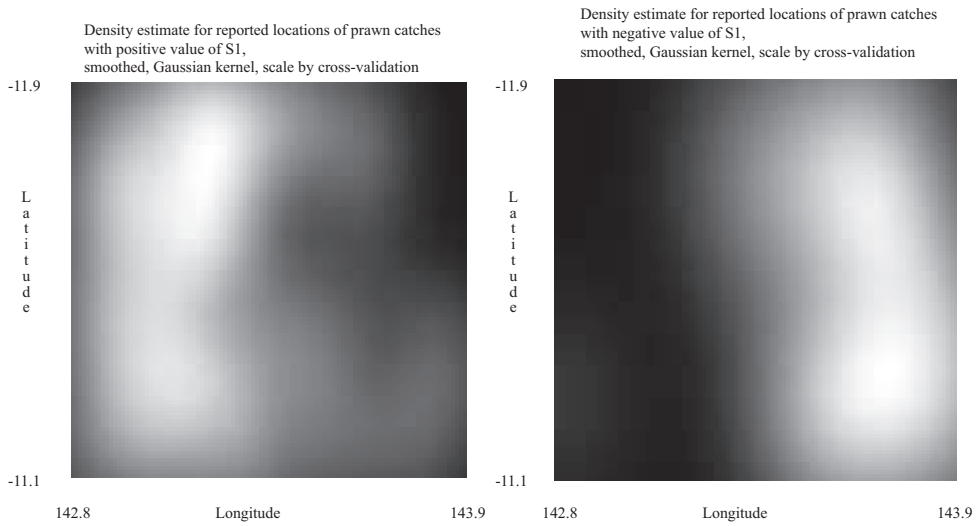

FIGURE 9.5: *Further plots of the prawn data of figure 9.1. On the* **left**, *a density estimate of the probability distribution from which the fishing locations which achieved positive values of the first score were drawn, where lighter pixels correspond to larger density values. On the* **right**, *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.*

one nuisance effect here to be careful of. If you use a kernel that has finite support, you could find that an omitted test item has zero probability; this leads to trouble with logarithms, etc. You could avoid this by obtaining an initial scale estimate with a kernel that has infinite support, then refining this with a kernel with finite support.

*** example *** careful about how kernel functions scale with dimension

9.1.3 Kernel Smoothing

We expect that, if $\mathbf{x}$ is near a training example $\mathbf{x}_i$, then $y(\mathbf{x})$ will be similar to y_i. This suggests constructing an estimate of $y(\mathbf{x})$ as a weighted sum of the values at nearby examples. Write $W(\mathbf{x}, \mathbf{x}_i)$ for the weight applied to y_i when estimating $y(\mathbf{x})$. Then our estimate of $y(\mathbf{x})$ is

$$\sum_{i=1}^{N} y_i W(\mathbf{x}, \mathbf{x}_i).$$

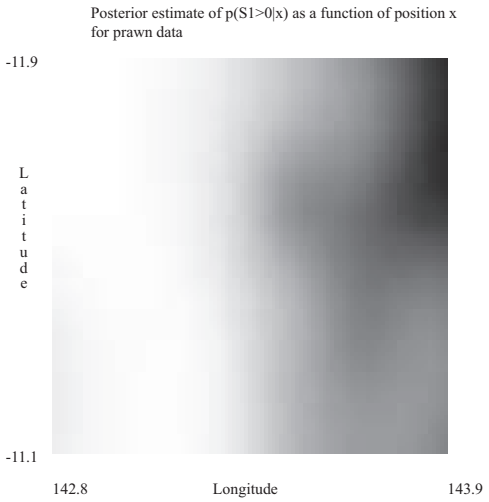

FIGURE 9.6: *An estimate of the posterior probability of obtaining a positive value of the first catch score for prawns, as a function of position, from the prawn data of figure 9.1, using the density estimate of figure* **??**. *Lighter pixels correspond to larger density values. I ignored the curvature of the earth, small at this scale, when computing distances between points.*

We need good choices of $W(\mathbf{x}, \mathbf{x}_i)$. There are some simple, natural constraints we can impose. We should like y to be a convex combination of the observed values. This means we want $W(\mathbf{x}, \mathbf{x}_i)$ to be non-negative (so we can think of them as weights) and $\sum_i W(\mathbf{x}, \mathbf{x}_i) = 1$

Assume we have a kernel function $K(u)$. Then a natural choice of weights is

$$W(\mathbf{x}, \mathbf{x}_i; h_i) = \frac{K(\frac{\|\mathbf{x}-\mathbf{x}_i\|}{h_i})}{\sum_{j=1}^{k} K(\frac{\|\mathbf{x}-\mathbf{x}_j\|}{h_i})}$$

where I have expanded the notation for the weight function to keep track of the scaling parameter, h_i, which we will need to select. Notice that, at each data point, we are using a kernel of different width to set weights. This should seem natural to you. If there are few points near a given data point, then we should have weights that vary slowly, so that we can for a weighted average of those points. But if there are many points nearby, we can have weights that vary fast. The weights are non-negative because we are using a non-negative kernel function. The weights sum to 1, because we divide at each point by the sum of all kernels evaluated at that point. For reference, this gives the expression

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

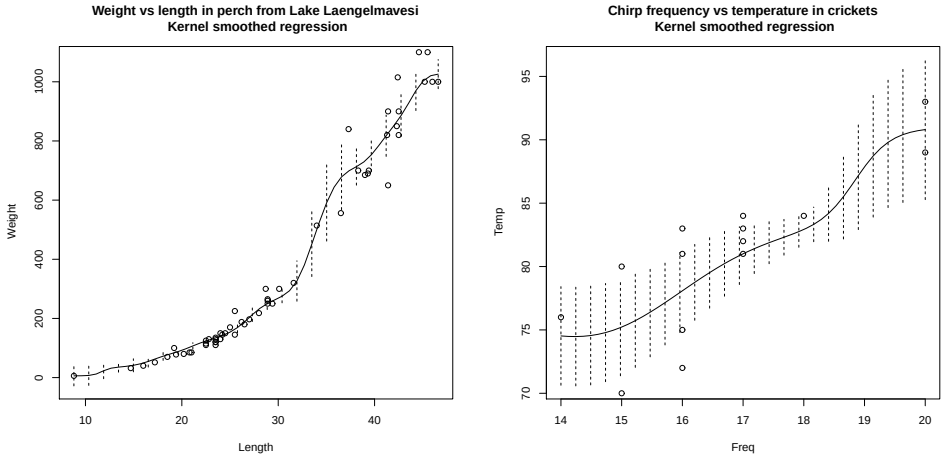

FIGURE 9.7: *Non parametric regressions for the datasets of Figures 7.1 and 7.5. The curve shows the expected value of the independent variable for each value of the explanatory variable. The vertical bars show the standard error of the predicted density for the independent variable, for each value of 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.*

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` (see appendix for code samples).

*** robustness isn't really an issue here - copes rather well with outliers *** not completely correct - locfit will use an alternative *** for that, you need to discuss biweight

*** distinction between (a) the standard error of prediction and (b) the sd of predictive distribution

*** you could get away with poor scaling of $\mathbf{x}$ if you scale each dimension separately

*** Curse of dimension and multiple here

*** appendix to each chapter - R CODE for each figure?

9.2 EXPLOITING YOUR NEIGHBORS FOR REGRESSION

TODO: work in local polynomial stuff

Nearest neighbors can clearly predict a number for a query example — you find the closest training example, and report its number. This would be one way to use nearest neighbors for regression, but it isn't terribly effective. One important difficulty is that the regression prediction is piecewise constant (Figure 9.10). If

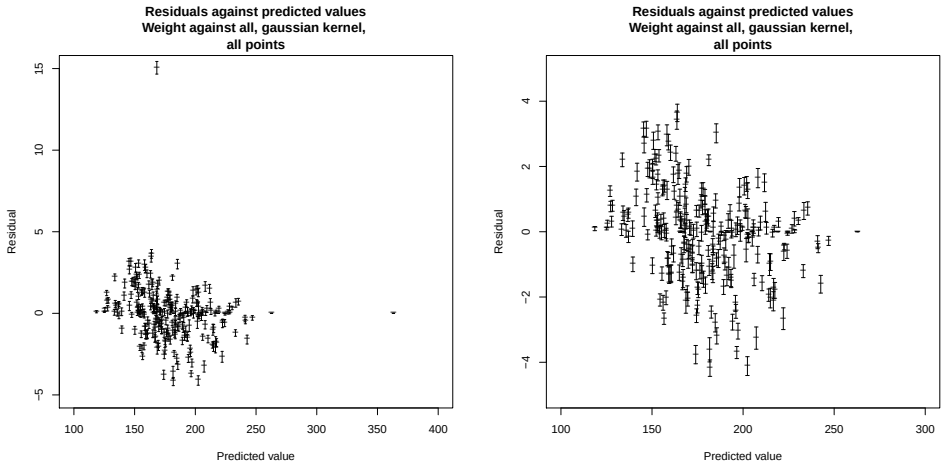

FIGURE 9.8:

there is an immense amount of data, this may not present major problems, because the steps in the prediction will be small and close together. But it's not generally an effective use of data.

A more effective strategy is to find several nearby training examples, and use them to produce an estimate. This approach can produce very good regression estimates, because every prediction is made by training examples that are near to the query example. However, producing a regression estimate is expensive, because for every query one must find the nearby training examples.

Write $\mathbf{x}$ for the query point, and assume that we have already collected the N 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. Write w_i for the weight at the i'th point. Then the estimate is

$$y_{pred} = \frac{\sum_i w_i y_i}{\sum_i w_i}.$$

A variety of weightings are reasonable choices. 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 $w_i = 1/d_i$. Alternatively, we could use an exponential function to strongly weight down more distant points, using

$$w_i = \exp\left(\frac{-d_i^2}{2\sigma^2}\right).$$

We will need to choose a scale σ, which can be done by cross-validation. 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. Alternatively,

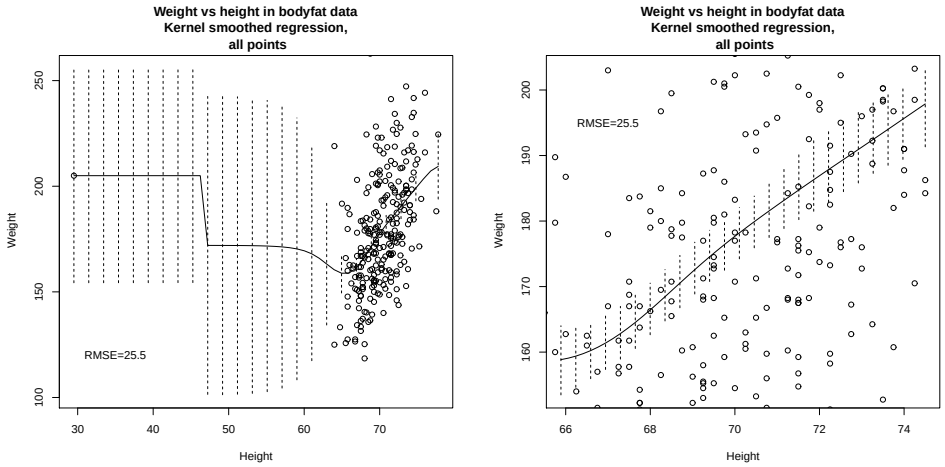

FIGURE 9.9:

if there are enough nearest neighbors, we could form a distance weighted linear regression, then predict the value at the query point from that regression.

Each of these strategies presents some difficulties when $\mathbf{x}$ has high dimension. In that case, it is usual that the nearest neighbor is a lot closer than the second nearest neighbor. If this happens, then each of these weighted averages will boil down to evaluating the dependent variable at the nearest neighbor (because all the others will have very small weight in the average).

Remember this: *Nearest neighbors can be used for regression. In the simplest approach, you find the nearest neighbor to your feature vector, and take that neighbor's number as your prediction. More complex approaches smooth predictions over multiple neighbors.*

**** link this to non parametric and kernel methods

9.2.1 Local Polynomial Regression

Imagine wish to predict a value at a point $\mathbf{x}_0$. Write y_0 for the value we predict. A reasonable choice would be to choose a number that minimizes a least squares difference to the training values, weighted in some way. We would like weights to be large for training points that are close to the test point, and small for points that are far away. We could achieve this using a kernel function, and choose the y_o to be the value of α_0 that minimizes

$$\sum_i (y_i - \alpha_0)^2 K(\frac{\| \mathbf{x}_0 - \mathbf{x}_i \|}{h_i}).$$

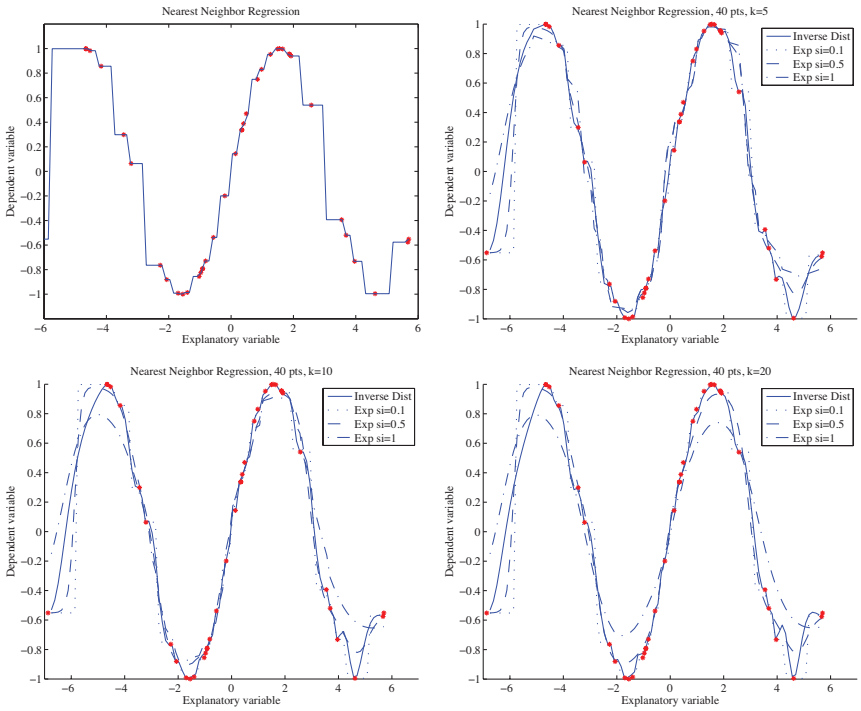

FIGURE 9.10: *Different forms of nearest neighbors regression, predicting y from a one-dimensional x, using a total of 40 training points.* **Top left:** *reporting the nearest neighbor leads to a piecewise constant function.* **Top right:** *improvements are available by forming a weighted average of 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.*

If you differentiate and set to zero, etc., you will find you have the familiar expression for kernel regression

$$y_0 = \sum_{i=1}^{N} y_i \left(\frac{K(\frac{\|\mathbf{x}_0 - \mathbf{x}_i\|}{h_i})}{\sum_{j=1}^{k} K(\frac{\|\mathbf{x}_0 - \mathbf{x}_j\|}{h_i})} \right).$$

This suggests something really fruitful. You can read the equation

$$\sum_i (y_i - \alpha_0)^2 K(\frac{\| \mathbf{x}_0 - \mathbf{x}_i \|}{h_i})$$

as choosing a local function at each $\mathbf{x}_i$ such that the weighted errors are minimized. The local function is constant (the value of α_0), but doesn't have to be. Instead, it could be polynomial. A polynomial function about the point $\mathbf{x}_i$

9.2.2 Using your Neighbors to Predict More than a Number

Linear regression takes some features and predicts a number. But in practice, one often wants to predict something more complex than a number. For example, I might want to predict a parse tree (which has combinatorial structure) from a sentence (the explanatory variables). As another example, I might want to predict a map of the shadows in an image (which has spatial structure) against an image (the explanatory variables). As yet another example, I might want to predict which direction to move the controls on a radio-controlled helicopter (which have to be moved together) against a path plan and the current state of the helicopter (the explanatory variables).

Looking at neighbors is a very good way to solve such problems. The general strategy is relatively simple. We find a large collection of pairs of training data. Write $\mathbf{x}_i$ for the explanatory variables for the i'th example, and $\mathbf{y}_i$ for the dependent variable in the i'th example. This dependent variable could be anything — it doesn't need to be a single number. It might be a tree, or a shadow map, or a word, or anything at all. I wrote it as a vector because I needed to choose some notation.

In the simplest, and most general, approach, we obtain a prediction for a new set of explanatory variables $\mathbf{x}$ by (a) finding the nearest neighbor and then (b) producing the dependent variable for that neighbor. We might vary the strategy slightly by using an approximate nearest neighbor. If the dependent variables have enough structure that it is possible to summarize a collection of different dependent variables, then we might recover the k nearest neighbors and summarize their dependent variables. How we summarize rather depends on the dependent variables. For example, it is a bit difficult to imagine the average of a set of trees, but quite straightforward to average images. If the dependent variable was a word, we might not be able to average words, but we can vote and choose the most popular word. If the dependent variable is a vector, we can compute either distance weighted averages or a distance weighted linear regression.

Example: Filling Large Holes with Whole Images 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.

If one has a large hole in a large image, we may not be able to just extend a texture to fill the hole. Instead, entire objects might need to appear in the hole (Figure 9.11). There is a straightforward, and extremely effective, way to achieve this. We match the image to a large collection of images, to find the nearest neighbors (the details of the distance function are below). This yields a set of example images where all the pixels we didn't want to replace are close to those of the query image. From these, we choose one, and fill in the pixels from that image.

There are several ways to choose. If we wish to do so automatically, we could

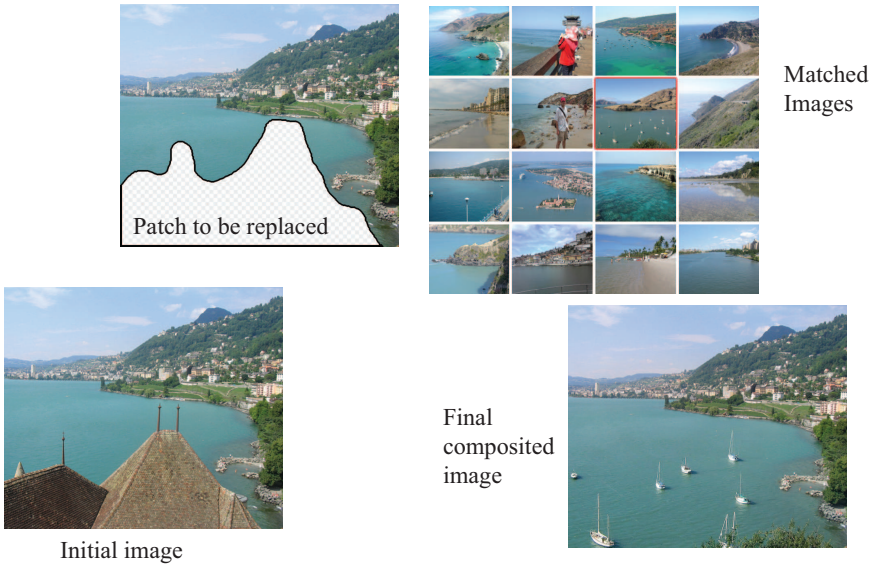

FIGURE 9.11: *We can fill large holes in images by matching the image to a collection, choosing one element of the collection, then cutting out an appropriate block of pixels and putting them into the hole in the query image. In this case, the hole has been made by an artist, who wishes to remove the 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".*

use the example with the smallest distance to the image. Very often, an artist is involved, and then we could prepare a series of alternatives — using, perhaps, the k closest examples — then show them to the artist, who will choose one. This method, which is very simple to describe, is extremely effective in practice.

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 $\sharp\mathcal{K}$ for the size of this set. We can now use

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

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.

Remember this: *Nearest neighbors can be used to predict more than numbers. Examples include parse trees, blocks of pixels, and so on.*

CHAPTER 10

Neural Networks

10.1 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 w_0. 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.

10.1.1 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

$$p(\text{class} = i|\mathbf{x}, \theta) = s_i(\mathbf{o}(\mathbf{x}, \theta)).$$

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. 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.

The essential difficulty here is to choose θ that results in the best behavior. We will do so by writing a cost function that measures the "goodness" of the classification. 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 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). This yields a loss function

$$\frac{1}{N}\sum_{i\in\text{data}}\left[\sum_{j\in\text{classes}}\left\{-\mathbf{y}_i^T\log\mathbf{s}(\mathbf{x}_i,\theta)\right\}\right].$$

Notice that this cost function is written in a clean way that will 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 9.9), 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. We will penalize large sets of weights. 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 9.9), we write λ for a weight applied to

the penalty. Our cost function is then

$$S(\theta, \mathbf{x}; \lambda) = \frac{1}{N} \sum_{i \in \text{data}} \left[\sum_{j \in \text{classes}} \left\{ -\mathbf{y}_i^T \log \mathbf{s}(\mathbf{x}_i, \theta) \right\} \right] + \frac{\lambda}{2} \sum_{u \in \text{units}} \mathbf{w}_u^T \mathbf{w}_u$$

(misclassification loss) (penalty)

10.1.2 Building a Classifier out of Units: Training

I have described a simple classifier built out of units. We must now train this classifier. 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. This is mainly an exercise in 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 $s_k(\mathbf{x}_i, \theta_k)$ is the output of the softmax function for the k'th unit for input $\mathbf{x}_i$. This represents the probability that example is of class k under the current model. Then we must compute

$$\nabla_{\theta_u} \frac{1}{M} \sum_{i \in \text{minibatch}} \left[\sum_{j \in \text{classes}} \left\{ -\mathbf{y}_i^T \log \mathbf{s}(\mathbf{x}_i, \theta) \right\} \right] + \frac{\lambda}{2} \sum_{j \in \text{classes}} \mathbf{w}_j^T \mathbf{w}_j.$$

The gradient is easily computed using the chain rule. I will write $c(i)$ for the class of example i, and $\mathbb{I}_{[u=c(i)]}(u)$ for an indicator function that takes the value 1 when $u = c(i)$, and 0 otherwise. We have that

$$\begin{aligned} \nabla_{\theta_u} \log s_j(\mathbf{x}_i, \theta_j) &= [-\nabla_{\theta_u} o_u(\mathbf{x}_i, \theta_u)] \, \mathbb{I}_{[u=c(i)]}(u) + \\ &\quad s_u(\mathbf{x}_i, \theta_u) \nabla_{\theta_u} o_u(\mathbf{x}_i, \theta_u) \end{aligned}$$

and

$$\nabla_{\theta_u} o_u(\mathbf{x}_i, \theta_u) = \begin{cases} \mathbf{0} & \text{if } \mathbf{w}_u^T \mathbf{x}_i + b_u \leq 0 \\ \begin{bmatrix} \mathbf{x}_i \\ 0 \end{bmatrix} & \text{otherwise} \end{cases}.$$

The gradient computed for a minibatch is

$$\begin{bmatrix} \nabla_{\theta_1} (\text{misclassification loss} + \text{penalty}) \\ \ldots \nabla_{\theta_C} (\text{misclassification loss} + \text{penalty}) \end{bmatrix}$$

where

$$\nabla_{\theta_u}(\text{misclassification loss}) = \sum_{i \in \text{minibatch}} \left[-\nabla_{\theta_u} o_u(\mathbf{x}_i, \theta_u)\right] \mathbb{I}_{[u=c(i)]}(u) + s_u(\mathbf{x}_i, \theta_u) \nabla_{\theta_u} o_u(\mathbf{x}_i, \theta_u)$$

and

$$\nabla_{\theta_u}(\text{penalty}) = \lambda \begin{bmatrix} \mathbf{w}_u \\ 0 \end{bmatrix}.$$

10.2 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 9.9 - 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; 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.

10.2.1 Notation

The main issue here is training this object. 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.

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

$$L(\mathbf{y}_i, \mathbf{s}(\mathbf{o}^L(\mathbf{x}_i, \theta))) = -\log \mathbf{y}_i^T \mathbf{s}(\mathbf{o}^L(\mathbf{x}_i, \theta)))$$

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 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.

10.2.2 Training, Gradients and Backpropagation

Now consider $\nabla_\theta E$. You can think of this as a vector of stacked vectors, one per layer. There is a trick for computing these vectors. We have, using the chain rule,

$$\nabla_{\theta^L} E = \sum_u \left(\frac{\partial E}{\partial o_u^L} \right) \nabla_{\theta^L} o_u^L(\mathbf{x}^L, \theta^L).$$

Yet more notation; I will write $\#(L)$ for the number of parameters in layer L. You should think of this as

$$\mathbf{v}^T \mathcal{J} = \begin{bmatrix} \frac{\partial E}{\partial o_1^L} \\ \ldots \\ \frac{\partial E}{\partial o_C^L} \end{bmatrix}^T \begin{pmatrix} \frac{\partial o_1^L(\mathbf{x}^L, \theta^L)}{\partial \theta_1^L} & \ldots & \frac{\partial o_1^L(\mathbf{x}^L, \theta^L)}{\partial \theta_{\#(L)}^L} \\ \ldots & \ldots & \ldots \\ \frac{\partial o_C^L(\mathbf{x}^L, \theta^L)}{\partial \theta_1^L} & \ldots & \frac{\partial o_C^L(\mathbf{x}^L, \theta^L)}{\partial \theta_{\#(L)}^L} \end{pmatrix}$$

The matrix of first partials is clearly important. I will write $\mathcal{J}_{\mathbf{o}^l; \theta^l}$ to mean

$$\begin{pmatrix} \frac{\partial o_1^l(\mathbf{x}^l, \theta^l)}{\partial \theta_1^l} & \ldots & \frac{\partial o_1^l(\mathbf{x}^l, \theta^l)}{\partial \theta_{\#(l)}^l} \\ \ldots & \ldots & \ldots \\ \frac{\partial o_C^l(\mathbf{x}^l, \theta^l)}{\partial \theta_1^l} & \ldots & \frac{\partial o_C^l(\mathbf{x}^l, \theta^l)}{\partial \theta_{\#(l)}^l} \end{pmatrix}$$

and $\mathcal{J}_{\mathbf{o}^l;\mathbf{x}^l}$ to mean

$$\begin{pmatrix} \frac{\partial o_1^l(\mathbf{x}^l,\theta^l)}{\partial x_1^l} & \dots & \frac{\partial o_1^l(\mathbf{x}^l,\theta^l)}{\partial x_{\#(l)}^l} \\ \dots & \dots & \dots \\ \frac{\partial o_C^l(\mathbf{x}^l,\theta^l)}{\partial x_1^l} & \dots & \frac{\partial o_C^l(\mathbf{x}^l,\theta^l)}{\partial x_{\#(l)}^l} \end{pmatrix}.$$

We can now get to the point. We have

$$\begin{aligned} \nabla_{\theta^L} E &= (\nabla_{\mathbf{o}^L} E)\, \mathcal{J}_{\mathbf{o}^L;\theta^L} \\ \nabla_{\theta^{L-1}} E &= (\nabla_{\mathbf{o}^L} E)\, \mathcal{J}_{\mathbf{o}^L;\mathbf{x}^L} \mathcal{J}_{\mathbf{o}^{L-1};\theta^{L-1}}) \\ \nabla_{\theta^{L-2}} E &= (\nabla_{\mathbf{o}^L} E)\, \mathcal{J}_{\mathbf{o}^L;\mathbf{x}^L} \mathcal{J}_{\mathbf{o}^{L-1};\mathbf{x}^{L-1}}) \mathcal{J}_{\mathbf{o}^{L-2};\theta^{L-2}}) \\ &\dots \end{aligned}$$

We can make the recursion more obvious with some notation. We have

$$\begin{aligned} \mathbf{v}^L &= \left(\nabla_{\mathbf{o}}^L E\right) \\ \nabla_{\theta^L} E &= \mathbf{v}^L \mathcal{J}_{\mathbf{o}^L;\theta^L} \\ \mathbf{v}^{L-1} &= \mathbf{v}^L \mathcal{J}_{\mathbf{o}^L;\mathbf{x}^L} \\ \nabla_{\theta^{L-1}} E &= \mathbf{v}^{L-1} \mathcal{J}_{\mathbf{o}^{L-1};\theta^{L-1}}) \\ &\dots \\ \mathbf{v}^{i-1} &= \mathbf{v}^i \mathcal{J}_{\mathbf{o}^i;\mathbf{x}^i} \\ \nabla_{\theta^{i-1}} E &= \mathbf{v}^{i-1} \mathcal{J}_{\mathbf{o}^{i-1};\theta^{i-1}}) \\ &\dots \end{aligned}$$

** ISSUE with $\mathcal{C}$ ** computing efficiently; up, then down

10.2.3 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 $\eta^{(r)}$.

Starting: Starting with $\theta^{(0)} = \mathbf{0}$ is a truly terrible idea. You should check that every gradient in this case will also be zero, meaning that your method will never move away from this point. It is usual to start with $\theta^{(0)}$ a small random vector. Different practitioners use slightly different approaches ** FAN IN/FAN OUT

The Learning Rate: *** Learning curves and spotting trouble *** examples

10.2.4 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 $f(x, y) = (1/2)(x^2 + \epsilon y^2)$, where ϵ is a small positive number. The gradient at (x, y) is $[x, \epsilon y]$. For simplicity, use a fixed learning rate η, so we have $\left[x^{(r)}, y^{(r)}\right] = \left[(1-\eta)x^{(r-1)}, (1-\epsilon\eta)y^{(r-1)}\right]$. 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 $\left[x^{(r)}, y^{(r)}\right] = \left[(1-\eta)^r x^{(0)}, (1-\epsilon\eta)^r y^{(0)}\right]$. The problem is that the gradient in x is quite large (so x must change quickly) and the gradient in y is small (so y changes slowly). In turn, for steps in x to converge we must have $|1-\eta| < 1$; but for steps in y to converge, we require only the much weaker constraint $|1-\epsilon\eta| < 1$. Imagine we choose the largest η we dare for the x constraint. The x value will very quickly have small magnitude, though its sign will change with each step. But the y steps will move you closer to the right spot only extremely slowly.

Another way to see this problem is to reason geometrically. Figure **??** 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 **??**).

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 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.

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. We choose another number, Δ, (the **decay rate**; typical values might be 0.9, 0.99 or 0.999), and iterate

$$\begin{aligned} c_i^{(r+1)} &= \Delta c_i^{(r)} + (1-\Delta)(g_i^{(r)})^2 \\ \theta_i^{(r+1)} &= \theta_i^{(r)} - \eta \frac{g_i^{(r)}}{(c_i^{(r+1)})^{\frac{1}{2}} + \alpha} \end{aligned}$$

10.3 CONVOLUTION AND ORIENTATION FEATURES FOR IMAGES

10.4 CONVOLUTIONAL NEURAL NETWORKS

CHAPTER 11

Classification II

11.1 LOGISTIC REGRESSION

11.1.1 The Logistic Loss

We will choose $\mathbf{a}$ and b by choosing values that minimize the cost of errors made by the classifier. In particular, we will adopt a cost function of the form:

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

For the moment, we will ignore the penalty term. The training error cost will be of the form

$$(1/N)\sum_{i=1}^{N} C((\mathbf{a}^T\mathbf{x}_i + b), y_i)$$

so at each point in the training data, we compute a cost from the true value of y_i and the predicted value. This cost should be large if y_i and $y_i^{(p)}(\mathbf{a}^T\mathbf{x}_i + b, y_i)$ have different signs, and small if they have the same sign. It is convenient to write

$$\gamma_i = (\mathbf{a}^T\mathbf{x}_i + b, y_i).$$

For **logistic regression**, the cost function using this notation is

$$C((\mathbf{a}^T\mathbf{x}_i + b), y_i) = C(\gamma_i, y_i) = \log\left(1 + \exp\left(-y_i\gamma_i\right)\right).$$

The function $L(1, \gamma)$ is plotted in Figure 11.1. This loss is sometimes known as the **logistic loss**. This loss very strongly penalizes a large positive γ_i if y_i is negative (and vice versa). However, there is no significant advantage to having a large positive γ_i if y_i is positive. This means that the significant components of the loss function will be due to examples that the classifier gets wrong, but also due to examples that have γ_i near zero (i.e., the example is close to the decision boundary).

You should notice another important property of this loss. Assume we wish to predict a label for a new data item. The loss we would incur depends quite strongly on the magnitude of γ. If we produce a large value of γ for that data item *with the wrong sign*, then we would incur a very large loss. This means that we should prefer values of $\mathbf{a}$ and b that will tend to produce small values of γ. In turn, we should prefer small values of $\mathbf{a}$ if they give about the same value of training loss. This is our penalty term. We should use a cost function of the form

$$\text{Training Loss} + \frac{\lambda}{2}\left(\text{Norm of } \mathbf{a}\right)$$

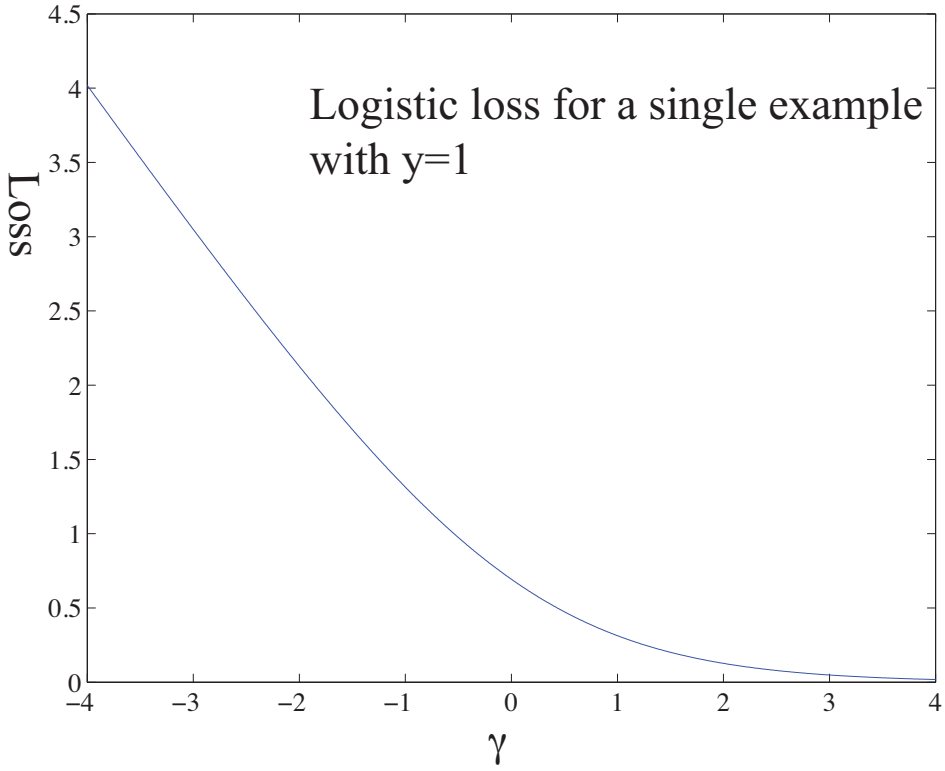

FIGURE 11.1: *The logistic loss, plotted for the case $y_i = 1$. In the case of the logistic loss, the horizontal variable is the $\gamma_i = \mathbf{a} \cdot \mathbf{x}_i$ of the text. Notice that giving a strong negative response to this positive example causes a loss that grows linearly as the magnitude of the response grows. Notice also that giving an insufficiently positive response also causes a loss. Giving a strongly positive response is cheap or free.*

which is

$$\left[\frac{1}{N}\sum_{i \in \text{examples}} \{\log(1 + \exp -y_i\gamma_i\})\right] + \frac{\lambda}{2}\mathbf{a}^T\mathbf{a}$$

where $\lambda > 0$ is a constant chosen for good performance. Too large a value of λ, and the classifier will behave poorly on training and test data; too small a value, and the classifier will behave poorly on test data.

Usually, the value of λ is set with a validation dataset. We train classifiers with different values of λ on a test dataset, then evaluate them on a validation set—data whose labels are known, but which is not used for training—and finally choose the λ that gets the best validation error. The error is not usually all that sensitive to the choice of λ, so searching values by factors of 10 is usually fine.

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.

11.1.2 Logistic Regression and Softmax
11.2 NEURAL NETS
11.2.1 Two Layers of Logistic Regression
The RELU
11.2.2 Training, Gradients and Backpropagation
11.2.3 Variants of SGD
11.3 CONVOLUTION AND ORIENTATION FEATURES
11.4 CONVOLUTIONAL NEURAL NETWORKS

CHAPTER 12

Boosting

12.1 GRADIENTBOOST

*** for classification *** for regression *** in each case with trees?

12.2 ADABOOST

**?

CHAPTER 13

Some Important Models

13.1 HMM'S
13.2 CRF'S
13.3 FITTING AND INFERENCE WITH MCMC?

CHAPTER 14

Math Resources

14.1 USEFUL MATERIAL ABOUT MATRICES

Terminology:

- A matrix $\mathcal{M}$ is **symmetric** if $\mathcal{M} = \mathcal{M}^T$. A symmetric matrix is necessarily square.
- We write $\mathcal{I}$ for the identity matrix.
- A matrix is **diagonal** if the only non-zero elements appear on the diagonal. A diagonal matrix is necessarily symmetric.
- A symmetric matrix is **positive semidefinite** if, for any $\mathbf{x}$ such that $\mathbf{x}^T\mathbf{x} > 0$ (i.e. this vector has at least one non-zero component), we have $\mathbf{x}^T\mathcal{M}\mathbf{x} \geq 0$.
- A symmetric matrix is **positive definite** if, for any $\mathbf{x}$ such that $\mathbf{x}^T\mathbf{x} > 0$, we have $\mathbf{x}^T\mathcal{M}\mathbf{x} > 0$.
- A matrix $\mathcal{R}$ is **orthonormal** if $\mathcal{R}^T\mathcal{R} = \mathcal{I} = \mathcal{I}^T = \mathcal{R}\mathcal{R}^T$. Orthonormal matrices are necessarily square.

Orthonormal matrices: 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 $\mathbf{u} = \mathcal{R}\mathbf{x}$, 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 $\mathbf{y}^T\mathbf{x}$; but, by the same argument as 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.

Eigenvectors and Eigenvalues: Assume $\mathcal{S}$ is a $d \times d$ symmetric matrix, $\mathbf{v}$ is a $d \times 1$ vector, and λ is a scalar. If we have

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

then $\mathbf{v}$ 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 $\mathcal{U} = [\mathbf{v}_1, \ldots, \mathbf{v}_d]$. This matrix is orthonormal, meaning that $\mathcal{U}^T\mathcal{U} = \mathcal{I}$. This means that there is a diagonal matrix Λ 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.

Diagonalizing a symmetric matrix: This gives us a particularly important procedure. We can convert any symmetric matrix $\mathcal{S}$ to a diagonal form by computing

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

This procedure is referred to as **diagonalizing** a matrix. Again, 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. Diagonalization allows us to show that positive definiteness is equivalent to having all positive eigenvalues, and positive semidefiniteness is equivalent to having all non-negative eigenvalues.

Factoring a matrix: Assume that $\mathcal{S}$ is symmetric and positive semidefinite. We have that

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

and all the diagonal elements of Λ are non-negative. Now construct a diagonal matrix whose diagonal entries are the positive square roots of the diagonal elements of Λ; call this matrix $\Lambda^{(1/2)}$. We have $\Lambda^{(1/2)}\Lambda^{(1/2)} = \Lambda$ and $(\Lambda^{(1/2)})^T = \Lambda^{(1/2)}$. Then we have that

$$\mathcal{S} = (\mathcal{U}\Lambda^{(1/2)})(\Lambda^{(1/2)}\mathcal{U}^T) = (\mathcal{U}\Lambda^{(1/2)})(\mathcal{U}\Lambda^{(1/2)})^T$$

so we can factor $\mathcal{S}$ into the form $\mathcal{X}\mathcal{X}^T$ by computing the eigenvectors and eigenvalues.

14.1.1 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. 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. Both $\mathcal{U}$ and $\mathcal{V}$ are orthonormal (i.e. $\mathcal{U}\mathcal{U}^T = \mathcal{I}$ and $\mathcal{V}\mathcal{V}^T = \mathcal{I}$).

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

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

14.1.2 Approximating A Symmetric Matrix

Assume we have a $k \times k$ symmetric matrix $\mathcal{T}$, and we wish to construct a matrix $\mathcal{A}$ that approximates it. We require that (a) the rank of $\mathcal{A}$ is precisely $r < k$ and (b) the approximation should minimize the **Frobenius norm**, that is,

$$\| (\mathcal{T} - \mathcal{A}) \|_F^2 = \sum_{ij} (T_{ij} - A_{ij})^2.$$

It turns out that there is a straightforward construction that yields $\mathcal{A}$.

The first step is to notice that if $\mathcal{U}$ is orthonormal and $\mathcal{M}$ is any matrix, then

$$\| \mathcal{U}\mathcal{M} \|_F = \| \mathcal{M}\mathcal{U} \|_F = \| \mathcal{M} \|_F.$$

This is true because $\mathcal{U}$ is a rotation (as is $\mathcal{U}^T = \mathcal{U}^{-1}$), and rotations do not change the length of vectors. So, for example, if we write $\mathcal{M}$ as a table of row vectors $\mathcal{M} = [\mathbf{m}_1, \mathbf{m}_2, ...\mathbf{m}_k]$, then $\mathcal{U}\mathcal{M} = [\mathcal{U}\mathbf{m}_1, \mathcal{U}\mathbf{m}_2, ...\mathcal{U}\mathbf{m}_k]$. Now $\| \mathcal{M} \|_F^2 = \sum_{j=1}^k \| \mathbf{m}_j \|^2$, so $\| \mathcal{U}\mathcal{M} \|_F^2 = \sum_{i=1}^k \| \mathcal{U}\mathbf{m}_k \|^2$. But rotations do not change lengths, so $\| \mathcal{U}\mathbf{m}_k \|^2 = \| \mathbf{m}_k \|^2$, and so $\| \mathcal{U}\mathcal{M} \|_F = \| \mathcal{M} \|_F$. To see the result for the case of $\mathcal{M}\mathcal{U}$, just think of $\mathcal{M}$ as a table of row vectors.

Notice that, if $\mathcal{U}$ is the orthonormal matrix whose columns are eigenvectors of $\mathcal{T}$, then we have

$$\| (\mathcal{T} - \mathcal{A}) \|_F^2 = \| \mathcal{U}^T (\mathcal{T} - \mathcal{A}) \mathcal{U} \|_F^2.$$

Now write Λ_r for $\mathcal{U}^T \mathcal{A} \mathcal{U}$, and Λ for the diagonal matrix of eigenvalues of $\mathcal{T}$. Then we have

$$\| (\mathcal{T} - \mathcal{A}) \|_F^2 = \| \Lambda - \Lambda_A \|_F^2,$$

an expression that is easy to solve for Λ_A. We know that Λ is diagonal, so the best Λ_A is diagonal, too. The rank of $\mathcal{A}$ must be r, so the rank of Λ_A must be r as well. To get the best Λ_A, we keep the r largest diagonal values of Λ, and set the rest to zero; Λ_A has rank r because it has only r non-zero entries on the diagonal, and every other entry is zero.

Now to recover $\mathcal{A}$ from Λ_A, we know that $\mathcal{U}^T\mathcal{U} = \mathcal{U}\mathcal{U}^T = \mathcal{I}$ (remember, $\mathcal{I}$ is the identity). We have $\Lambda_A = \mathcal{U}^T \mathcal{A} \mathcal{U}$, so

$$\mathcal{A} = \mathcal{U} \Lambda_A \mathcal{U}^T.$$

We can clean up this representation in a useful way. Notice that only the first r columns of $\mathcal{U}$ (and the corresponding rows of $\mathcal{U}^T$) contribute to $\mathcal{A}$. The remaining $k - r$ are each multiplied by one of the zeros on the diagonal of Λ_A. Remember that, by convention, Λ was sorted so that the diagonal values are in descending order (i.e. the largest value is in the top left corner). We now keep only the top left $r \times r$ block of Λ_A, which we write Λ_r. We then write $\mathcal{U}_r$ for the $k \times r$ matrix consisting of the first r columns of $\mathcal{U}$. Then

$$\mathcal{A} = \mathcal{U}_r \Lambda_r \mathcal{U}^T$$

This is so useful a result, I have displayed it in a box; you should remember it.

Procedure: 14.1 *Approximating a symmetric matrix with a low rank matrix*

Assume we have a symmetric $k \times k$ matrix $\mathcal{T}$. We wish to approximate $\mathcal{T}$ with a matrix $\mathcal{A}$ that has rank $r < k$. Write $\mathcal{U}$ for the matrix whose columns are eigenvectors of $\mathcal{T}$, and Λ for the diagonal matrix of eigenvalues of $\mathcal{A}$ (so $\mathcal{A}\mathcal{U} = \mathcal{U}\Lambda$). Remember that, by convention, Λ was sorted so that the diagonal values are in descending order (i.e. the largest value is in the top left corner).
Now construct Λ_r from Λ by setting the $k - r$ smallest values of Λ to zero, and keeping only the top left $r \times r$ block. Construct $\mathcal{U}_r$, the $k \times r$ matrix consisting of the first r columns of $\mathcal{U}$. Then

$$\mathcal{A} = \mathcal{U}_r \Lambda_r \mathcal{U}_r^T$$

is the best possible rank r approximation to $\mathcal{T}$ in the Frobenius norm.

Now if $\mathcal{A}$ is positive semidefinite (i.e. if at least the r largest eigenvalues of $\mathcal{T}$ are non-negative), then we can factor $\mathcal{A}$ as in the previous section. This yields a procedure to approximate a symmetric matrix by factors. This is so useful a result, I have displayed it in a box; you should remember it.

Procedure: 14.2 *Approximating a symmetric matrix with low dimensional factors*

Assume we have a symmetric $k \times k$ matrix $\mathcal{T}$. We wish to approximate $\mathcal{T}$ with a matrix $\mathcal{A}$ that has rank $r < k$. We assume that at least the r largest eigenvalues of $\mathcal{T}$ are non-negative. Write $\mathcal{U}$ for the matrix whose columns are eigenvectors of $\mathcal{T}$, and Λ for the diagonal matrix of eigenvalues of $\mathcal{A}$ (so $\mathcal{A}\mathcal{U} = \mathcal{U}\Lambda$). Remember that, by convention, Λ was sorted so that the diagonal values are in descending order (i.e. the largest value is in the top left corner).
Now construct Λ_r from Λ by setting the $k - r$ smallest values of Λ to zero and keeping only the top left $r \times r$ block. Construct $\Lambda_r^{(1/2)}$ by replacing each diagonal element of Λ with its positive square root. Construct $\mathcal{U}_r$, the $k \times r$ matrix consisting of the first r columns of $\mathcal{U}$. Then write $\mathcal{V} = (\mathcal{U}_r \Lambda_r^{(1/2)})$

$$\mathcal{A} = \mathcal{V}\mathcal{V}^T$$

is the best possible rank r approximation to $\mathcal{T}$ in the Frobenius norm.

Error functions and Gaussians: The **error function** is defined by

$$\mathsf{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

and programming environments can typically evaluate the error function. This fact is made useful to us by a simple change of variables. We get

$$\frac{1}{\sqrt{2\pi}} \int_0^x e^{\frac{-u^2}{2}} du = \frac{1}{\sqrt{\pi}} \int_0^{\frac{x}{\sqrt{2}}} e^{-t^2} dt = \frac{1}{2}\mathsf{erf}\left(\frac{x}{\sqrt{2}}\right).$$

A particularly useful manifestation of this fact comes by noticing that

$$\frac{1}{\sqrt{2\pi}} \int_{-\infty}^0 e^{\frac{-t^2}{2}} dt = 1/2$$

(because $\frac{1}{\sqrt{2\pi}} e^{\frac{-u^2}{2}}$ is a probability density function, and is symmetric about 0). As a result, we get

$$\frac{1}{\sqrt{2\pi}} \int_{-\infty}^x e^{\frac{-t^2}{2}} dt = 1/2(1 + \mathsf{erf}\left(\frac{x}{\sqrt{2}}\right)).$$

Inverse error functions: We sometimes wish to know the value of x such that

$$\frac{1}{\sqrt{2\pi}} \int_{-\infty}^x e^{\frac{-t^2}{2}} dt = p$$

for some given p. The relevant function of p is known as the **probit function** or the **normal quantile function**. We write

$$x = \Phi(p).$$

The probit function Φ can be expressed in terms of the **inverse error function**. Most programming environments can evaluate the inverse error function (which is the inverse of the error function). We have that

$$\Phi(p) = \sqrt{2}\mathsf{erf}^{-1}(2p - 1).$$

One problem we solve with some regularity is: choose u such that

$$\int_{-u}^u \frac{1}{\sqrt{2\pi}} \exp\left(-x^2/2\right) dx = p.$$

Notice that

$$\begin{aligned} \frac{p}{2} &= \frac{1}{\sqrt{2\pi}} \int_0^u e^{\frac{-t^2}{2}} dt \\ &= \frac{1}{2}\mathsf{erf}\left(\frac{u}{\sqrt{2}}\right) \end{aligned}$$

so that

$$u = \sqrt{2}\mathsf{erf}^{-1}(p).$$