Chapter 13
Supervised Learning

A frequent question in biological and biomedical applications is whether a property of interest (say, disease type, cell type, the prognosis of a patient) can be “predicted”, given one or more other properties, called the predictors. Often we are motivated by a situation in which the property to be predicted is unknown (it lies in the future, or is hard to measure), while the predictors are known. The crucial point is that we learn the prediction rule from a set of training data in which the property of interest is also known. Once we have the rule, we can either apply it to new data, and make actual predictions of unknown outcomes; or we can dissect the rule with the aim of better understanding the underlying biology.

Compared to unsupervised learning and what we have seen in Chapters 5, 7 and 9, where we do not know what we are looking for or how to decide whether our result is “right”, we are on much more solid ground with supervised learning: the objective is clearly stated, and there are straightforward criteria to measure how well we are doing.

The central issue in supervised learning\(^1\) is overfitting and generalisability: did we just learn the training data “by heart” by constructing a rule that has 100% accuracy on the training data, but would perform poorly on any new data? Or did our rule indeed pick up some of the pertinent patterns in the system being studied, which will also apply to yet unseen new data?

13.1 Goals for this chapter

In this chapter we will

- see exemplary applications that motivate the use of supervised learning methods
- learn what discriminant analysis does,
- define measures of performance,
- encounter the curse of dimensionality and see what overfitting is,
- find out about regularisation and understand the concepts of generisability and model complexity,
• see how to use cross-validation to tune parameters of the algorithms,
• get to see a unified framework for machine learning algorithms in R that allows you to use hundreds of methods in a consistent manner,
• discuss method hacking.

13.2 What are the data?

The basic data structure for both supervised and unsupervised learning is (at least conceptually) a dataframe, where each row corresponds to an object and the columns are different features of the objects. While in unsupervised learning we aim to find (dis)similarity relationships between the objects based on their feature values (e.g., by clustering or ordination), in supervised learning we aim to find a mathematical function (or a computational algorithm) that predicts the value of one of the features from the other features. Many implementations require that there are no missing values, whereas other methods can be generalized to work with some amount of missing data.

The feature that we select over all the others with the aim of predicting is called the objective or the response. Sometimes the choice is natural, but sometimes it is also instructive to reverse the roles, especially if we are interested in dissecting the prediction function for the purpose of biological understanding, or in disentangling correlations from causation.

The framework for supervised learning covers both continuous and categorical response variables. In the continuous case we also call it regression, in the categorical case, classification. It turns out that this distinction is not a detail, as it has quite far-reaching consequences for the choice of loss function (Section 13.5) and thus the choice of algorithm (Friedman, 1997).

The first question to consider in any supervised learning task is how the number of objects compares to the number of predictors. The more data, the better, and much of the hard work in supervised learning has to do with overcoming the limitations of having a finite (and typically, too small) training set.

Question 13.2.1. Give examples where we have encountered instances of supervised learning with a categorical response in this book.

13.2.1 Motivating examples

Predicting diabetes type

The diabetes dataset (Reaven and Miller, 1979) presents three different groups of diabetes patients and five clinical variables measured on them.

library("ggplot2")
library("readr")
library("magrittr")
diabetes = read_csv("../data/diabetes.csv", col_names = TRUE)
diabetes

## # A tibble: 144 x 7
## #  id relwt glufast glutest steady insulin group
## 1  1   0.81  80  356  124   55   3
## 2  3   0.94 105  319  143  105   3
## 3  5   1.00  90  323  240  143   3
## 4  7   0.91 100  350  221  119   3
## 5  9   0.99  97  379  142   98   3
## 6 11  0.90  91  353  221   53   3
## 7 13  0.96  78  290  136  142   3
## 8 15  0.74  86  312  208  105   3
## 9 17  1.10  90  364  152  146   3
## 10 19  0.83  85  296  116   60   3
## # ... with 134 more rows

diabetes$group %<>% factor

We used the forward-backward pipe operator %<>% to convert the group column into a factor.

library("reshape2")

ggplot(melt(diabetes, id.vars = c("id", "group")),
       aes(x = value, col = group)) +
  geom_density() + facet_wrap(~variable, ncol = 2, scales = "free")

The plot is shown in Figure 13.3.

**Predicting cellular phenotypes**

Neumann et al. (2010) observed human cancer cells using live-cell imaging. The cells were genetically engineered so that their histones were tagged with a green fluorescent protein (GFP). A genome-wide RNAi library was applied to the cells, and for each siRNA perturbation, movies of a few hundred cells were recorded for about two days, to see what effect the depletion of each gene had on cell cycle, nuclear morphology and cell proliferation. Their paper reports the use of an automated image classification algorithm that quantified the visual appearance of each cell’s nucleus and enabled the prediction of normal mitosis states or aberrant nuclei. The algorithm was trained on the data from around 3000 cells that were annotated by a human expert. It was then applied to almost 2 billions images of nuclei (Figure 13.4). Using automated image classification provided scalability (annotating 2 billion images manually would take a long time) and objectivity.
Predicting embryonic cell states

We will revisit the mouse embryo data (Ohnishi et al., 2014), which we have already seen in Chapters 3, 5 and 7, and show how we can predict the developmental state (Embryonic Days) from the gene expression measurements.

13.3 Linear discrimination

We start with one of the simplest possible discrimination problems, where we have objects described by two continuous features (so the objects can be thought of as points in the 2D plane) and falling into three groups. Our aim is to define class boundaries, which are lines in the 2D space.

13.3.1 Diabetes data

Let’s see whether we can predict the feature group from the features insulin and glutest variables in the diabetes data. It’s always a good idea to first visualise the data (Figure 13.5).
ggdb = ggplot(mapping = aes(x = insulin, y = glutest)) + geom_point(aes(colour = group), data = diabetes)
ggdb

We’ll start with a method called linear discriminant analysis (LDA). This method is a foundation stone of classification, many of the more complicated (and sometimes more powerful) algorithms are really just generalisations of LDA.

library("MASS")
diabetes_lda = lda(group ~ insulin + glutest, data = diabetes)
diabetes_lda

## Call:
## lda(group ~ insulin + glutest, data = diabetes)
##
## ## Prior probabilities of groups:
## ## 1 2 3
## ## 0.2222222 0.2500000 0.5277778
## ##
## ## Group means:
## ## insulin glutest
## ## 1 320.9375 1027.3750
## ## 2 208.9722 493.9444
## ## 3 114.0000 349.9737
## ##
## ## Coefficients of linear discriminants:
## ## LD1 LD2
## ## insulin -0.004463900 -0.01591192
## ## glutest -0.005784238 0.00480830
## ##
## ## Proportion of trace:
## ## LD1 LD2
## ## 0.9677 0.0323

ghat = predict(diabetes_lda)$class
table(ghat, diabetes$group)

##
## ghat 1 2 3
## 1 25 0 0
## 2 6 24 6
## 3 1 12 70

mean(ghat != diabetes$group)
## [1] 0.1736111

> **Question 13.3.1.** What do the different parts of the above output mean?

Now, let’s visualise the LDA result\(^4\). We are going to plot the prediction regions for each of the three groups. We do this by creating a grid of points and using our prediction rule on each of them. We’ll then also dig a bit deeper into the mechanics of LDA and plot the class centers (diabetes_lda$means) and ellipses that correspond to the fitted covariance matrix (diabetes_lda$scaling). Assembling this

\(^4\) Note how we first visualised the data, in Figure 13.5, and are now going to visualise the fitted model (Figure 13.6). The prediction regions can, in principle, be shown for any classification method, including a “black box” method. On the other hand, the cluster centers and ellipses in Figure 13.6 are a method-specific visualisation.
visualization requires us to write a bit of code.

```r
make1Dgrid = function(x) {
    rg = range(x)
    wid = diff(rg)
    rg = rg + wid * 0.05 * c(-1, 1)
    seq(from = rg[1], to = rg[2], length.out = 100)
}
```

Set up the points for prediction, a $100 \times 100$ grid that covers the data range.

```r
diabetes_grid = with(diabetes, expand.grid(insulin = make1Dgrid(insulin),
                               glutest = make1Dgrid(glutest)))
```

Do the predictions.

```r
diabetes_grid$ghat = predict(diabetes_lda, newdata = diabetes_grid)$class
```

The group centers.

```r
centers = diabetes_lda$means
```

Compute a unit circle and an affine transformation of the circle into the ellipse we want to plot.

```r
unitcircle = exp(1i * seq(0, 2*pi, length.out = 90)) %>%
             (function(x) cbind(Re(x), Im(x)))
ellipse = unitcircle %*% solve(diabetes_lda$scaling)
```

All three ellipses, one for each group center.

```r
ellipses = lapply(seq_len(nrow(centers)), function(i) {
    (ellipse +
     matrix(centers[i, ], byrow = TRUE,
            ncol = ncol(centers), nrow = nrow(ellipse))) %>%
     cbind(group = i)) %>%
     do.call(rbind, .) %>%
     data.frame
})%>%
factor
```

Now we are ready to plot (Figure 13.6).

```r
ggdb +
geom_raster(aes(fill = ghat),
            data = diabetes_grid, alpha = 0.4, interpolate = TRUE) +
geom_point(data = as_data_frame(centers), pch = "*", size = 8) +
geom_path(aes(colour = group), data = ellipses) +
scale_x_continuous(expand = c(0, 0)) +
scale_y_continuous(expand = c(0, 0))
```

**Question 13.3.2.** Why is the boundary between the prediction regions for groups 1 and 2 not perpendicular to the line between the cluster centers?

**Question 13.3.3.** How confident would you be about the predictions in those areas of the 2D plane that are far from all of the cluster centers?

**Question 13.3.4.** Why is the boundary between the prediction regions for groups 2 and 3 not half-way between the centers, but shifted in favor of class 3? (Hint: have a
look at the prior argument of lda.) Try again with uniform prior.

Answer 13.3.1. The result of the following code chunk is shown in Figure 13.7.

```r
diabetes_up = lda(group ~ insulin + glutest, data = diabetes, 
 prior = with(diabetes, rep(1/nlevels(group), nlevels(group))))

diabetes_grid$ghup = 
predict(diabetes_up, newdata = diabetes_grid$class

stopifnot(all.equal(diabetes_up$means, diabetes_lda$means))

ellipse_up = unitcircle %*% solve(diabetes_up$scaling)
ellipses_up = lapply(seq_len(nrow(centers)), function(i) {
 (ellipse_up +
 matrix(centers[i, ], byrow = TRUE, 
 ncol = ncol(centers), nrow = nrow(ellipse_up))) %>%
 cbind(group = i)) %>% do.call(rbind, .) %>% data.frame
ellipses_up$group %<>% factor

ggdb + geom_raster(aes(fill = ghup),
 data = diabetes_grid, alpha = 0.4, interpolate = TRUE) +
 geom_point(data = data.frame(centers), pch = "+", size = 8) +
 geom_path(aes(colour = group), data = ellipses_up) +
 scale_x_continuous(expand = c(0, 0)) +
 scale_y_continuous(expand = c(0, 0))
```

The `stopifnot` line confirms that the class centers are the same –they are independent of the prior–, but the joint covariance is not.

Question 13.3.5. What is the difference in the prediction accuracy if we use all 5 variables instead of just insulin and glufast?

Answer 13.3.2.

```r
diabetes_lda5 = lda(group ~ relwt + glufast + glutest + steady + insulin, data = diabetes)
diabetes_lda5

# Call:
# lda(group ~ relwt + glufast + glutest + steady + insulin, data = diabetes)
#
# Prior probabilities of groups:
# 1 2 3
# 0.2222222 0.2500000 0.5277778
#
# Group means:
# relwt glufast glutest steady insulin
# 1 0.9915625 213.65625 1027.3750 108.8438 320.9375
# 2 1.0558333 99.30556 493.9444 288.0000 208.9722
# 3 0.9372368 91.18421 349.9737 172.6447 114.0000
#
# Coefficients of linear discriminants:
# LD1 LD2
```

Figure 13.7: As Figure 13.6, but with uniform class priors.
Question 13.3.6. Instead of approximating the prediction regions by classification from a grid of points, compute the separating lines explicitly from the linear determinant coefficients.

Answer 13.3.3. See Section 4.3, Equation (4.10) in (Hastie et al., 2008).

### 13.3.2 Predicting embryonic cell state from gene expression

Assume that we already know that the four genes FN1, TIMD2, GATA4 and SOX7 are relevant to the classification task. We want to build a classifier that predict the developmental time (embryonic days, E3.25, E3.5, E4.5). We load the data and select four corresponding probes.

```r
library("Hiiragi2013")
data("x")
probes = c("1426642_at","1418765_at","1418864_at","1416564_at")
embryoCells = as_data_frame(t(exprs(x)[probes, ])) %>%
  mutate(Embryonic.day = x$Embryonic.day) %>%
  filter(x$genotype == "WT")
```

We can use the Bioconductor annotation package associated with the microarray to verify that the probes correspond to the intended genes.

```r
annotation(x)
```

```
## [1] "mouse4302"
```

```r
library("mouse4302.db")
anno = AnnotationDbi::select(mouse4302.db, keys = probes,
  columns = c("SYMBOL", "GENENAME"))
anno
```
and produce a pairs plot (Figure 13.8).

```r
library("GGally")
ggpairs(embryoCells, mapping = aes(col = Embryonic.day),
columns = anno$SYMBOL, upper = list(continuous = "points"))
```

![Figure 13.8: Expression values of the discriminating genes, with the prediction target Embryonic.day shown by color.](image)

We can now call lda on these data. The linear combinations LD1 and LD2 that serve as discriminating variables are given in the slot `ec$scaling` of the output from lda.

```r
ec$lda = lda(Embryonic.day ~ Fn1 + Timd2 + Gata4 + Sox7,
data = embryoCells)
```
For the visualisation of the learned model in Figure 13.9, we need to build the prediction regions and their boundaries by expanding the grid in the space of the two new coordinates LD1 and LD2.

```r
round(ec_lda$scaling, 1)
##     LD1 LD2
## Fn1  -0.2 -0.4
## Timd2  0.5  0.0
## Gata4 -0.1 -0.6
## Sox7  -0.7  0.5
```

For the visualisation of the learned model in Figure 13.9, we need to build the prediction regions and their boundaries by expanding the grid in the space of the two new coordinates LD1 and LD2.

```r
ec_rot = predict(ec_lda)$x %>% as_data_frame %>% mutate(ed = embryoCells$Embryonic.day)
ec_lda2 = lda(ec_rot[, 1:2], predict(ec_lda)$class)
ec_grid = with(ec_rot, expand.grid(LD1 = make1Dgrid(LD1), LD2 = make1Dgrid(LD2)))
ec_grid$edhat = predict(ec_lda2, newdata = ec_grid)$class
ggplot() + geom_point(aes(x = LD1, y = LD2, colour = ed), data = ec_rot) + geom_raster(aes(x = LD1, y = LD2, fill = edhat), data = ec_grid, alpha = 0.4, interpolate = TRUE) + scale_x_continuous(expand = c(0, 0)) + scale_y_continuous(expand = c(0, 0)) + coord_fixed()
```

**Question 13.3.7.** Repeat these analyses using quadratic discriminant analysis (qda). What difference do you see in the shape of the boundaries?

**Answer 13.3.4.** See Figure 13.10.

```r
library("gridExtra")
ec_qda = qda(Embryonic.day ~ Fn1 + Timd2 + Gata4 + Sox7, data = embryoCells)
variables = colnames(ec_qda$means)
pairs = combn(variables, 2)
lapply(seq_len(ncol(pairs)), function(i) {
  grid = with(embryoCells, 
    expand.grid(x = make1Dgrid(get(pairs[, 1])), 
               y = make1Dgrid(get(pairs[, 2])))) %>%
  `colnames<-`(pairs[, i])
  for (v in setdiff(variables, pairs[, i])) 
    grid[[v]] = median(embryoCells[[v]])
  grid$edhat = predict(ec_qda, newdata = grid)$class
})
```

Figure 13.9: LDA classification regions for Embryonic.day.
Question 13.3.8. What happens if you call lda or qda with a lot more genes, say the first 1000, in the Hiiragi dataset?

Answer 13.3.5.

```r
lda(t(exprs(x))[, 1:1000], x$Embryonic.day)
```

```
#Warning in lda.default(x, grouping, ...): variables are collinear
```

```r
qda(t(exprs(x))[, 1:1000], x$Embryonic.day)
```

```
# Error in qda.default(x, grouping, ...): some group is too small for 'qda'
```

13.4 Machine learning vs rote learning

Computers are really good at memorizing facts. In the worst case, a machine learning algorithm is a roundabout way of doing this\(^6\). The central question in statistical learning is whether the algorithm was able to generalize, i.e., interpolate and extrapolate. Let’s look at the following example. We generate random data (rnorm) for \(n\) objects, with different numbers of features (given by \(p\)). We train a LDA on these data and compute the \textit{misclassification rate}, i.e., the fraction of times the prediction is wrong (\texttt{pred} \(!=\) \texttt{resp}).
library("dplyr")
p = 2:21
n = 20

mcl = lapply(p, function(k) {
  replicate(100, {
    xmat = matrix(rnorm(n * k), nrow = n)
    resp = sample(c("apple", "orange"), n, replace = TRUE)
    fit = lda(xmat[, 1:k], resp)
    pred = predict(fit)$class
    mean(pred != resp)
  }) %>% mean %>% tibble(mcl = .)
}) %>% bind_rows %>% cbind(., p = p)

ggplot(mcl, aes(x = p, y = mcl)) + geom_line() + geom_point() + ylab("Misclassification rate")

Question 13.4.1. What is the purpose of the `replicate` loop in the above code? What happens if you omit it (or replace the 100 by 1)?

Answer 13.4.1. Averaging the misclassification rate over 100 replicates makes the estimate more stable, and since we are working with simulated data, we are at liberty to do so. For each single replicate, the curve is a noisier version of Figure 13.11.

Figure 13.11 seems to imply that we can perfectly predict random labels from random data, if we only fit a complex enough model, i.e., one with many parameters. How can we overcome such an absurd conclusion? The problem with the above code is that the model performance is evaluated on the same data on which it was trained. This generally leads to positive bias, as you see in this crass example. How can we overcome this problem? The key idea is to assess model performance on different data than those on which the model was trained.

13.4.1 Cross-validation

A naive approach might be to split the data in two halves, and use the first half for learning (“training”), the second half for assessment (“testing”). It turns out that this is needlessly variable and needlessly inefficient. Needlessly variable, since by splitting the data only once, our results can be quite affected by how the splitting happens to fall. It seems better to do the splitting many times, and average. This will give us more stable results. Needlessly inefficient, since the performance of machine learning algorithms depends on the number of samples, and the performance measured on half the data is likely7 to be worse than what it is with all the data. For this reason, it is better to use unequal sizes of training and test data. In the extreme case, we’ll use as much as n − 1 samples for training, and the remaining one for testing. After we’ve done this likewise for all samples, we can average our performance metric. This is called leave-one-out cross-validation. An alternative is k-fold cross-validation, where the samples are repeatedly split into a training set of size of around \( n(k − 1)/k \) and a test set of size of around \( n/k \). Both alternatives have pros and contras, and there is not a universally best choice. An advantage of leave-one-out is that the amount of data used for training is close to the maximally available data; this is

7 Unless we have such an excess of data that it doesn’t matter.
especially important if the sample size is limiting and “every little matters” for the algorithm. A drawback of leave-one-out is that the training sets are all very similar, so they may not sufficiently model the kind of sampling changes to be expected if a new dataset came along. For large \( n \), leave-one-out cross-validation can be needlessly time-consuming.

```r
estimate_mcl_loocv = function(x, resp) {
  vapply(seq_len(nrow(x)), function(i) {
    fit = lda(x[-i, ], resp[-i])$class
    ptst = predict(fit, newdata = x[ , drop = FALSE])$class
    c(train = mean(ptrn != resp[-i]), test = (ptst != resp[i]))
  }, FUN.VALUE = c(0, 0)) %>% rowMeans %>% as_data_frame
}
xmat = matrix(rnorm(n * last(p)), nrow = n)
resp = sample(c("apple", "orange"), n, replace = TRUE)
mcl = lapply(p, function(k) {
  estimate_mcl_loocv(xmat[, 1:k], resp)
}) %>% bind_rows %>% data.frame(p) %>% melt(id.var = "p")
ggplot(mcl, aes(x = p, y = value, col = variable)) + geom_line() + geom_point() + ylab("Misclassification rate")
```

The result is show in Figure 13.12.

**Question 13.4.2.** Why are the curves in Figure 13.12 more variable (“wiggly”) than in Figure 13.11? How can you overcome this?

**Answer 13.4.2.** Only one dataset (xmat, resp) was used to calculate Figure 13.12, whereas for Figure 13.11, we had the data generated within a `replicate` loop. You could similarly extend the above code to average the misclassification rate curves over many replicate datasets.

### 13.4.2 The curse of dimensionality

In Section 13.4.1 we have seen overfitting and cross-validation on random data, but how does it look if there is in fact a relevant class separation?

```r
p = 2:20
mcl = replicate(100, {
  xmat = matrix(rnorm(n * last(p)), nrow = n)
  resp = sample(c("apple", "orange"), n, replace = TRUE)
  lapply(p, function(k) {
    estimate_mcl_loocv(xmat[, 1:k], resp)
  }) %>% bind_rows %>% cbind(p = p) %>% melt(id.var = "p")
}, simplify = FALSE)
mcl = bind_rows(mcl) %>% group_by(p, variable) %>%
```

Figure 13.12: Cross-validation: the misclassification rate of LDA applied to random data, when evaluated on test data that were not used for learning, hovers around 0.5 independent of \( p \). The misclassification rate on the training data is also shown. It behaves similar to what we already saw in Figure 13.11.

Figure 13.13: As we increase the number of features included in the model, the misclassification rate initially improves; as we start including more and more irrelevant features, it increases again, as we are fitting noise.

\(^8\) See Chapter Model Assessment and Selection in the book by Hastie et al. (2008) for further discussion on these trade-offs.
summarise(value = mean(value))

```r
ggplot(mcl, aes(x = p, y = value, col = variable)) + geom_line() + geom_point() + ylab("Misclassification rate")
```

The result is shown in Figure 13.13. The group centers are the vectors (in \(\mathbb{R}^{20}\)) given by the coordinates \((1, 1, 1, 1, 1, 0, 0, 0, \ldots)\) (apples) and \((2, 2, 2, 2, 2, 0, 0, 0, \ldots)\) (oranges), and the optimal decision boundary is the hyperplane orthogonal to the line between them. For \(k\) smaller than 6, the decision rule cannot reach this hyperplane—it is biased. As a result, the misclassification rate is suboptimal, and it decreases with \(k\). But what happens for \(k\) larger than 6? The algorithm is, in principle, able to model the optimal hyperplane, and it should not be distracted by the additional features. The problem is that it is. The more additional features enter the dataset, the higher the probability that one or more of them happen to fall in a way that they look like good, discriminating features in the training data—only to mislead the classifier and degrade its performance in the test data. Shortly we’ll see how to use penalization to try to control this problem.

The term **curse of dimensionality** was coined by Bellman (1961). It refers to the fact that high-dimensional spaces are very hard to sample. Our intuitions about distances between points in a high-dimensional space, and the relationship between its volume and surface, break down.

**Question 13.4.3.** Assume you have a dataset with 1,000,000 data points in \(p\) dimensions. The data are uniformly distributed in the unit hypercube (i.e., all features lie in the interval \([0, 1]\)). What’s the side length of a hypercube that can be expected to contain 10 points, as a function of \(p\)?

**Answer 13.4.3.** See Figure 13.15.

```r
sideLength = function(p, pointDensity = 1e6, pointsNeeded = 10) {
  pointsNeeded / pointDensity
}
ggplot(tibble(p = 1:750, sideLength = sideLength(p)),
       aes(x = p, y = sideLength)) +
  geom_line(col = "red") + geom_hline(aes(yintercept = 1), linetype = 2)
```

Generally, prediction at the boundaries of feature space is more difficult than in its interior, as it tends to involve extrapolation, rather than interpolation.

**Question 13.4.4.** What fraction of a unit cube’s total volume is closer than 0.01 to any of its surfaces, as a function of the dimension?

**Answer 13.4.4.** See Figure 13.16.

```r
p = 1:750
volOuterCube = 1 ^ p
volInnerCube = 0.98 ^ p
ggplot(tibble(p, 'V(shell)' = volOuterCube - volInnerCube),
       aes(x = p, y = 'V(shell)')) + geom_line(col = "blue")
```

**Question 13.4.5.** What is the coefficient of variation (ratio of standard deviation over average) of the distance between two randomly picked points in the unit hypercube, as a function of the dimension?
We solve this one by simulation. We generate \( n \) pairs of random points in the hypercube \((x_1, x_2)\) and compute their Euclidean distances. See Figure 13.17. This result can also be predicted from the central limit theorem.

\[
\begin{align*}
\text{n} &= 1000 \\
\text{df} &= \text{tibble}(
    p = \text{round}(10^{-\text{seq}(0, 4, by = 0.25)}), \\
    \text{cv} = \text{vapply}(p, \text{function}(k) {
        x1 = \text{matrix}((\text{runif}(k \times n), \text{nrow} = n)) \\
        x2 = \text{matrix}((\text{runif}(k \times n), \text{nrow} = n)) \\
        d = \text{sqrt}(\text{rowSums}((x1 - x2)^2)) \\
        \text{sd}(d) / \text{mean}(d) 
    }), \text{FUN.VALUE} = \text{NA_real_})
\end{align*}
\]

\[
\begin{align*}
\text{ggplot} &\text{df, aes}(x = \log10(p), y = \text{cv}) + \text{geom_line}(\text{col} = \text{“orange”}) + \text{geom_point()}
\end{align*}
\]

13.5 Objective functions

We’ve already seen the misclassification rate (MCR) used to assess our classification performance in Figures 13.11–13.13. Its population version is defined as

\[
\text{MCR} = E \left[ \mathbb{I}_{\hat{y} \neq y} \right],
\]

and for a finite sample

\[
\overline{\text{MCR}} = \frac{1}{n} \sum_{i=1}^{n} \mathbb{I}_{\hat{y}_i \neq y_i},
\]

This is not the only choice we could make. Perhaps we care more about the misclassification of apples as oranges than vice versa, and we can reflect this by introducing weights that depend on the type of error made into the sum of Equation (13.2) (or the integral of Equation (13.1)). This can get even more elaborate if we have more than two classes. Often we do not only want to see a single numeric summary, but the whole confusion table, which in R we can get via expressions like

\[
\text{table}(\text{truth, response})
\]

An important special case is binary classification with asymmetric costs – think about, say, a medical test. Here, the sensitivity (a.k.a. true positive rate or recall) is related to the misclassification of non-sick as sick, and the specificity (or true negative rate) depends on the probability of misclassification of sick as non-sick. Often, there is a single parameter (e.g., a threshold) that can be moved up and down, allowing a trade-off between sensitivity and specificity (and thus, equivalently, between the two types of misclassification). In those cases, we usually are not content to know the classifier performance at one single choice of threshold, but at many (or all) of them. This leads to receiver operating characteristic (ROC) or precision-recall curves.

Question 13.5.1. What are the exact relationships between the per-class misclassification rates and sensitivity and specificity?
Answer 13.5.1. The sensitivity or true positive rate is

\[ TPR = \frac{TP}{P}, \]

where TP is the number of true positives and P the number of all positives. The specificity or true negative rate is

\[ SPC = \frac{TN}{N}, \]

where TN is the number of true negatives and N the number of all negatives. See also https://en.wikipedia.org/wiki/Sensitivity_and_specificity

Another cost function can be computed from the Jaccard index, which we already saw in Chapter 5.

\[ J(A, B) = \frac{|A \cap B|}{|A \cup B|}, \quad (13.3) \]

where \( A \) is the set of samples for which the true class is 1 (\( A = \{ i | y_i = 1 \} \)) and \( B \) is the set of samples for which the predicted class is 1. \( J \) is a number between 0 and 1, and a high value of \( J \) indicates high overlap of the two sets. Note that \( J \) does not depend on the number of samples for which both true and predicted class is 0 – so it is particularly suitable for measuring the performance of methods that try to find rare events.

We can also consider probabilistic class predictions, which come in the form \( \hat{P}(Y | X) \). In this case, a possible risk function would be obtained by looking at distances between the true probability distribution and the estimated probability distributions. For two classes, the finite sample version of the log loss is

\[ \log \text{loss} = -\frac{1}{n} \sum_{i=1}^{n} y_i \log(\hat{p}_i) + (1 - y_i) \log(1 - \hat{p}_i), \quad (13.4) \]

where \( \hat{p}_i \in [0, 1] \) is the prediction, and \( y_i \in \{0, 1\} \) is the truth.\(^9\)

For continuous response variables (regression), a natural choice is the mean squared error (MSE). It is the average squared error,

\[ \widehat{\text{MSE}} = \frac{1}{n} \sum_{i=1}^{n} (\hat{Y}_i - Y_i)^2. \quad (13.5) \]

The population version is defined analogously, by turning the summation into an integral as in Equations (13.1) and (13.2).

Statisticians call functions like Equations (13.1–13.5) variously (and depending on context and predisposition) risk function, cost function, objective function.\(^{10}\)

\(^9\) Note that the log loss will be infinite if a prediction is totally confident (\( \hat{p}_i \) is exactly 0 or 1) but wrong.

\(^{10}\) There is even an \( R \) package dedicated to evaluation of statistical learners called metrics.
Variance–bias trade-off

An important fact that helps us understand the tradeoffs when picking a statistical learning model is that the MSE is the sum of two terms, and often the choices we can make are such that one of those terms goes down while the other one goes up. The bias measures how different the average of all the different estimates is from the truth, and variance, how much an individual one might scatter from the average value (Figure 13.18). In applications, we often only get one shot, therefore being reliably almost on target can beat being right on the long term average but really off today. The decomposition

\[
\text{MSE} = \text{Var}(\hat{Y}) + E[(\hat{Y} - Y)^2]
\]

follows by straightforward algebra.

When trying to minimize the MSE, it is important to remember that sometimes we can pay the price of some bias to obtain a much smaller variance and thus an overall estimator of lower MSE. In classification (with categorical response variables), different objective functions than the MSE are used, and there is usually no such straightforward decomposition as in Equation (13.6). In general, we can go much further in classification applications than in regression with trading biases for variance, since the discreteness of the response neutralizes certain biases (Friedman, 1997).

13.6.1 Penalization

In high-dimensional statistics, we are constantly plagued by variance: there is just not enough data to fit all the possible parameters. One of the most fruitful ideas in high-dimensional statistics is penalization: a tool to actively control and exploit the variance-bias tradeoff.

Although generalisation of LDA to high-dimensional settings is possible (Clemmensen et al., 2011; Witten and Tibshirani, 2011), it turns out that logistic regression is a more general approach\(^\text{11}\), and therefore we’ll now switch to that, using the glmnet package.

Multinomial\(^\text{12}\) logistic regression models the posterior log-odds between \(k\) classes and can be written in the form\(^\text{13}\)

\[
\log \frac{P(Y = i | X = x)}{P(Y = k | X = x)} = \beta_i^0 + \beta_i x,
\]

where \(i = 1, \ldots, k - 1\); \(x\) is the \(n \times p\) data matrix (\(n\): number of samples, \(p\): number of features), and \(\beta_i\) is a \(p\)-dimensional vector that determines how the classification odds for class \(i\) versus class \(k\) depend on \(x\). The numbers \(\beta_i^0\) are intercepts and depend, among other things, on the classes’ prior probabilities. Instead of the log odds (13.7) (i.e., ratios of class probabilities), we can also write down an equivalent model for the class probabilities themselves, and the fact that we here used the \(k\)-th...
class as a reference is an arbitrary choice, as the model estimates are equivariant under this choice (Hastie et al., 2008). The model is fit by maximising the log-likelihood \( l(\beta, \beta_0; x) \), where \( \beta = (\beta_1, \ldots, \beta_{k-1}) \) and analogously for \( \beta_0 \).

So far, so good. But as \( p \) gets larger, there is an increasing chance that some of the estimates go wildly off the mark, due to random sampling happenstances in the data. This is true even if for each individual coordinate of the vector \( \beta_i \), the error distribution is bounded: the probability of there being one coordinate that is in the far tails increases the more coordinates there are, i.e., the larger \( p \) is.

A related problem can also occur, not in (13.7), but in other, non-linear models, as the model dimension \( p \) increases while the sample size \( n \) remains the same: the likelihood landscape around its maximum becomes increasingly flat, and the maximum-likelihood estimate of the model parameters becomes more and more variable. Eventually, the maximum is no longer a point, but a submanifold, and the maximum likelihood estimate is unidentifiable.

Both of these limitations can be overcome with a modification of the objective: instead of maximising the bare log-likelihood, we maximise a penalized version of it,

\[
\hat{\beta} = \arg \max_{\beta} l(\beta, \beta_0; x) + \lambda \text{pen}(\beta),
\]

(13.8)

where \( \lambda \geq 0 \) is a real number, and pen is a convex function, called the penalty function. Popular choices are \( \text{pen}(\beta) = |\beta|^2 \) (ridge regression) and \( \text{pen}(\beta) = |\beta|^1 \) (lasso). In the elastic net, ridge and lasso are hybridized by using the penalty function \( \text{pen}(\beta) = (1 - \alpha)|\beta|^1 + \alpha|\beta|^2 \) with some further parameter \( \alpha \in [0, 1] \). The crux is, of course, how to choose the right \( \lambda \), and we will discuss that in the following.

### 13.6.2 Example: predicting colon cancer from stool microbiome composition

Zeller et al. (2014) studied metagenome sequencing data from fecal samples of 156 humans that included colorectal cancer patients and tumor-free controls. Their aim was to see whether they could identify biomarkers (presence or abundance of certain taxa) that could help with early tumor detection. The data are available from Bioconductor through its ExperimentHub service under the identifier EH359.

```r
library("ExperimentHub")
zh = ExperimentHub()
zeller = zh["EH361"]
```

<table>
<thead>
<tr>
<th>disease</th>
<th>cancer</th>
<th>large_adenoma</th>
<th>small_adenoma</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>53</td>
<td>15</td>
<td>61</td>
</tr>
<tr>
<td></td>
<td></td>
<td>27</td>
<td></td>
</tr>
</tbody>
</table>

**Question 13.6.1.** Explore the `eh` object to see what other datasets there are.

**Answer 13.6.1.**
For the following, let’s focus on the normal and cancer samples and set the adenomas aside.

```r
zellerNC = zeller[, zeller$disease %in% c("n", "cancer")]
```

Before jumping into model fitting, it is always a good idea to do some exploration of the data. First, let’s look at the sample annotations for some of the samples. We pick them randomly, since this can be more representative of the whole dataset than only looking at the first or last ones.

```r
data(zellerNC)[ sample(ncol(zellerNC), 3), ]
```

Next, let’s explore the feature names.

```r
formatfn = function(x)
gsub("|", "| ", x, fixed = TRUE) %>% lapply(strwrap)

rownames(zellerNC)[1:4]
```

15 We define the helper function `formatfn` to line wrap these long character strings for the available space here.
As you can see, the features are a mixture of abundance quantifications at different taxonomic levels, from kingdom over phylum to species. We could select only some of these, but here we continue with all of them. Next, let’s look at the distribution of some of the features. Here, we show two; in practice, it is helpful to scroll through many such plots quickly to get an impression.

```r
ggplot(melt(exprs(zellerNC)[c(510, 527), ]), aes(x = value)) + geom_histogram(bins = 25) + facet_wrap(~ Var1, ncol = 1, scales = "free")
```

In the simplest case, we fit model (13.7) as follows.

```r
library("glmnet")
glmfit = glmnet(x = t(exprs(zellerNC)), y = factor(zellerNC$disease), family = "binomial")
```

A remarkable feature of the `glmnet` function is that it fits (13.7) not only for one choice of $\lambda$, but for all possible $\lambda$s at once. For now, let’s look at the prediction performance for, say, $\lambda = 0.04$. The name of the function parameter is `s`:

```r
predTrsf = predict(glmfit, newx = t(exprs(zellerNC)), type = "class", s = 0.04)
table(predTrsf, zellerNC$disease)
```

### Question 13.6.2.
What is the x-axis in Figure 13.20? What are the different lines?

### Answer 13.6.2.
Consult the manual page of the function `plot.glmnet` in the `glmnet` package.

Let’s get back to the question of how to choose the parameter $\lambda$. We could try many different choices—and indeed, all possible choices—of $\lambda$, assess classification performance in each case using cross-validation, and then choose the best $\lambda$. We could do so by writing a loop as we did in the `estimate_mcl_loocv` function in Section 13.4.1. It turns out that the `glmnet` package already has built-in functionality for that, with the function `cv.glmnet`, which we can use instead.

```r
cvglmfit = cv.glmnet(x = t(exprs(zellerNC)), y = factor(zellerNC$disease), family = "binomial")
plot(cvglmfit)
```

16 But remember that this is on the training data, without cross-validation.

---

### Figure 13.19: Histograms of the distributions for two randomly selected features. The distributions are highly skewed, with many zero values and a thin, long tail of non-zero values.

### Figure 13.20: Regularization paths for `glmfit`.

17 You’ll already realize from the description of this strategy that if we optimize $\lambda$ in this way, the resulting apparent classification performance will likely be exaggerated. We need a truly independent dataset, or at least another, outer cross-validation loop to get a more realistic impression of the generalizability. We will get back to this question at the end of the chapter.
The diagnostic plot is shown in Figure 13.21. We can access the optimal value with

```r
cvglmfit$lambda.min
```

```
## [1] 0.08830775
```

As this value results from finding a minimum in an estimated curve, it turns out that it is often too small, i.e., that the implied penalization is too weak. A heuristic recommended by the authors of the `glmnet` package is to use a somewhat larger value instead, namely the largest value of $\lambda$ such that the performance measure is within 1 standard error of the minimum.

```r
cvglmfit$lambda.1se
```

```
## [1] 0.1015325
```

**Question 13.6.3.** How does the confusion table look like for $\lambda = \text{lambda.1se}$?

**Answer 13.6.3.**

```r
s0 = cvglmfit$lambda.1se
predict(glmfit, newx = t(exprs(zellerNC)), type = "class", s = s0) %>%
table(zellerNC$disease)
```

```
##
## cancer n
cancer 35 7
n 18 54
```

**Question 13.6.4.** What features drive the classification?

**Answer 13.6.4.**

```r
coeffs = coef(glmfit)[which.min(abs(glmfit$lambda - s0))]
topthree = order(abs(coefs), decreasing = TRUE)[1:3]

as.vector(coefs[topthree])
```

```
```

```r
formatfn(names(coefs)[topthree])
```

```
[[1]]

[1] "k__Bacteria| p__Candidatus_Saccharibacteria|
[[2]]

[1] "c__Candidatus_Saccharibacteria| o__Candidatus_Saccharibacteria_noname|
[[3]]

[1] "f__Candidatus_Saccharibacteria_noname|
[[4]]

[1] "g__Candidatus_Saccharibacteria_noname|
[[5]]

[1] "s__candidate_division_TM7_single_cell_isolate_TM7b"

[[2]]

[[1]]

[1] "k__Bacteria| p__Firmicutes| c__Clostridia| o__Clostridiales|
[[2]]

[1] "f__Ruminococcaceae| g__Subdoligranulum|
[[3]]

[1] "s__Subdoligranulum_variabile"

[[3]]

[[1]]

[1] "k__Bacteria| p__Firmicutes| c__Clostridia| o__Clostridiales|
[[2]]

[1] "f__Lachnospiraceae| g__Lachnospiraceae_noname|
[[3]]

[1] "s__Lachnospiraceae_bacterium_7_1_58FAA"
```
Question 13.6.5. How do the results change if we transform the data, say, with the asinh transformation as we saw in Chapter 5?

Answer 13.6.5. See Figure 13.22.

cv.glmnet(x = t(asinh(exprs(zellerNC))), y = factor(zellerNC$disease), family = "binomial")

Question 13.6.6. Would a good classification performance on these data mean that this assay is ready for screening and early cancer detection?

Answer 13.6.6. No. The performance here is measured on a set of samples in which the cases have similar prevalence as the controls. This serves well enough to explore the biology. However, in a real-life application, the cases will be much less frequent. To be practically useful, the assay must have a much higher specificity, i.e., not wrongly diagnose disease where there is none. To establish specificity, a much larger set of normal samples need to be tested.

13.6.3 Example: classifying mouse cells from their expression profiles

Figures 13.21 and 13.22 are textbook examples of how we expect the dependence of (cross-validated) classification performance versus model complexity (λ) to look. Now let’s get back to the mouse embryo cells data. We’ll try to classify the cells from embryonic day E3.25 with respect to their genotype.

```r
sx = x[, x$Embryonic.day == "E3.25"]
embryoCellsClassifier = cv.glmnet(t(exprs(sx)), sx$genotype, family = "binomial", type.measure = "class")
plot(embryoCellsClassifier)
```

In Figure 13.23 we see that the misclassification error is (essentially) monotonously increasing with λ, and is smallest for λ → 0, i.e., if we apply no penalization at all.

Question 13.6.7. What is going on with these data?

Answer 13.6.7. It looks that inclusion of more, and even of all features, does not harm the classification performance. In a way, these data are “too easy”. Let’s do a t-test for all features:

```r
mouse_de = rowtests(sx, "genotype")
ggplot(mouse_de, aes(x = p.value)) + geom_histogram(boundary = 0, breaks = seq(0, 1, by = 0.01))
```

The result, shown in Figure 13.24, shows that large number of genes are differentially expressed, and thus informative for the class distinction. We can also compute the pairwise distances between all samples, using all features.

```r
dists = as.matrix(dist(scale(t(exprs(sx)))))
diag(dists) = Inf
```

and then for each sample determine the class of its nearest neighbor

```r
nn = sapply(seq_len(ncol(dists)), function(i) which.min(dists[, i]))
table(x$sampleGroup, x$sampleGroup[nn]) %>% `colnames<-`(NULL)
```

Figure 13.22: like Figure 13.21, but using an asinh transformation of the data.

Figure 13.23: Cross-validated misclassification error versus penalty parameter for the mouse cells data.

Figure 13.24: Histogram of p-values for the per-feature t-tests between genotypes in the E3.25 samples.
Using all features, the nearest neighbour classifier is correct in almost all cases, including for the E3.25 wildtype vs FGF4-KO distinction. This means that for these data, there is no apparent benefit in regularisation or feature selection. Limitations of using all features might become apparent with truly new data, but that is out of reach for cross-validation.

### 13.7 A large choice of methods

We have now seen three classification methods: linear discriminant analysis (\texttt{lda}), quadratic discriminant analysis (\texttt{qda}) and the elastic net (\texttt{glmnet}). In fact, there are hundreds of different learning algorithms\footnote{For an introduction to the subject that uses R and provides many examples and exercises, we recommend (James et al., 2013).} available in R and its add-on packages. You can get an overview in the CRAN task view Machine Learning & Statistical Learning. Some examples are:

- Support vector machines: the function \texttt{svm} in the package \texttt{e1071}; \texttt{ksvm} in \texttt{kernlab}
- Tree based methods in the packages \texttt{rpart}, \texttt{tree}, \texttt{randomForest}
- Boosting methods: the functions \texttt{glmboost} and \texttt{gamboost} in package \texttt{mboost}
- PenalizedLDA in the package \texttt{PenalizedLDA}, \texttt{dudi.discr} and \texttt{dist.pcaiv} in \texttt{ade4}).

The complexity and heterogeneity of choices of learning strategies, tuning parameters and evaluation criteria in each of these packages can be confusing. You will already have noted differences in the interfaces of the \texttt{lda}, \texttt{qda} and \texttt{glmnet} functions, i.e., in how they expect their input data to presented and what they return. There is even greater diversity across all the other packages and functions. At the same time, there are common tasks such as cross-validation, parameter tuning and performance assessment that are more or less the same no matter what specific method is used. As you have seen, e.g., in our \texttt{estimate_mcl_loocv} function, the looping and data shuffling involved leads to rather verbose code.

So what to do if you want to try out and explore different learning algorithms? Fortunately, there are several projects that provide unified interfaces to the large number of different machine learning interfaces in R, and also try to provide “best practice” implementations of the common tasks such as parameter tuning and performance assessment. The two most well-known ones are the packages \texttt{caret} and \texttt{mlr}. 

```
Here we have a look at `caret`. You can get a list of supported methods through its `getModelInfo` function. There are quite a few, here we just show the first 8.

```r
library("caret")
caretMethods = names(getModelInfo())
head(caretMethods, 8)
#> [1] "ada"    "AdaBag"  "AdaBoost.M1" "adaboost"
#> [5] "amda1"  "ANFIS"   "avNNet"   "awnb"
length(caretMethods)
#> [1] 232
```

We will check out a neural network method, the `nnet` function from the eponymous package. The `parameter` slot informs us on the the available tuning parameters\(^{19}\).

```r
getModelInfo("nnet", regex = FALSE)[[1]]$parameter
#>   parameter class label
#> 1   size     numeric #Hidden Units
#> 2 decay     numeric  Weight Decay
```

Let’s try it out.

```r
trnCtrl = trainControl(
  method = "repeatedcv",
  repeats = 3,
  classProbs = TRUE)
tuneGrid = expand.grid(
  size = c(2, 4, 8),
  decay = c(0, 1e-2, 1e-1))

nnfit = train(
  Embryonic.day ~ Fn1 + Tmld2 + Gata4 + Sox7,
  data = embryoCells,
  method = "nnet",
  tuneGrid = tuneGrid,
  trControl = trnCtrl,
  metric = "Accuracy")
```

That’s quite a mouthful, but the nice thing is that this syntax is standardized and applies across many different methods. All you need to do specify the name of the method and the grid of tuning parameters that should be explored via the `tuneGrid` argument.

Now we can have a look at the output (Figure 13.25).

```r
nnfit
#> Neural Network
#> # 66 samples
#> # 4 predictor
```

\(^{19}\) They are described in the manual of the `nnet` function.
## 3 classes: ‘E3.25’, ‘E3.5’, ‘E4.5’

## No pre-processing

Resampling: Cross-Validated (10 fold, repeated 3 times)

Summary of sample sizes: 59, 59, 59, 59, 60, 60, ...

Resampling results across tuning parameters:

<table>
<thead>
<tr>
<th>size</th>
<th>decay</th>
<th>Accuracy</th>
<th>Kappa</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.00</td>
<td>0.7158333</td>
<td>0.4270723</td>
</tr>
<tr>
<td>2</td>
<td>0.01</td>
<td>0.7622619</td>
<td>0.5710980</td>
</tr>
<tr>
<td>2</td>
<td>0.10</td>
<td>0.7726984</td>
<td>0.5926065</td>
</tr>
<tr>
<td>4</td>
<td>0.00</td>
<td>0.7455952</td>
<td>0.5284067</td>
</tr>
<tr>
<td>4</td>
<td>0.01</td>
<td>0.7633730</td>
<td>0.5902138</td>
</tr>
<tr>
<td>4</td>
<td>0.10</td>
<td>0.7726984</td>
<td>0.5919238</td>
</tr>
<tr>
<td>8</td>
<td>0.00</td>
<td>0.7813095</td>
<td>0.6169179</td>
</tr>
<tr>
<td>8</td>
<td>0.01</td>
<td>0.7911508</td>
<td>0.6257162</td>
</tr>
<tr>
<td>8</td>
<td>0.10</td>
<td>0.7563095</td>
<td>0.5595771</td>
</tr>
</tbody>
</table>

Accuracy was used to select the optimal model using the largest value.
The final values used for the model were size = 8 and decay = 0.01.

plot(nnfit)
predict(nnfit) %>% head(10)

```
Levels: E3.25 E3.5 E4.5
```

**Question 13.7.1.** Will the accuracy that we obtained above for the optimal tuning parameters generalize to a new dataset? What could you do to address that?

**Answer 13.7.1.** No, it is likely to be too optimistic, as we have picked the optimum. To get a somewhat more realistic estimate of prediction performance when generalized, we could formalize (into computer code) all our data preprocessing choices and the above parameter tuning procedure, and embed this in another, outer cross-validation loop (Ambroise and McLachlan, 2002). However, this is likely still not enough, as we discuss in the next section.

### 13.7.1 Method hacking

In Chapter 6 we encountered *p-value hacking*. A similar phenomenon exists in statistical learning: given a dataset, we explore various different methods of preprocessing (such as normalization, outlier detection, transformation, feature selection), try out different machine learning algorithms and tune their parameters until we are content with the result. The measured accuracy is likely to be too optimistic, i.e., will not generalize to a new dataset. Embedding as many of our methodical choices into a computational formalism and having an outer cross-validation loop (not to be confused with the inner loop that does the parameter tuning) will ameliorate the problem. But is unlikely to address it completely, since not all our choices can be formalized.
The gold standard remains validation on truly unseen data. In addition, it is never a bad thing if the classifier is not a black box but can be interpreted in terms of domain knowledge. Finally, report not just summary statistics, such as misclassification rates, but lay open the complete computational workflow, so that anyone (including your future self) can convince themselves of the robustness of the result or of the influence of the preprocessing, model selection and tuning choices (Holmes, 2016).

Exercises

► Exercise 13.1. Apply a kernel support vector machine, available in the kernlab package, to the zeller microbiome data. What kernel function is best?

► Exercise 13.2. It has been quipped that all classification methods are just refinements of two archetypal ideas: discriminant analysis and $k$ nearest neighbors. In what sense might that be a useful classification?

► Answer 13.1. In linear discriminant analysis, we consider our objects as elements of $\mathbb{R}^p$, and the learning task is to define regions in this space, or boundary hyperplanes between them, which we use to predict the class membership of new objects. This is archetypal for classification by partition. Generalizations of linear discriminant analysis permit more general spaces and more general boundary shapes.

In $k$ nearest neighbors, no embedding into a coordinate space is needed, but instead we require a distance (or dissimilarity) measure that can be computed between each pair of objects, and the classification decision for a new object depends on its distances to the training objects and their classes. This is archetypal for kernel-based methods.

► Exercise 13.3. Use glmnet for a prediction of a continuous variable, i.e., for regression. Explore the effects of using ridge versus lasso penalty.

► Answer 13.2. There are infinitely many possibilities here. For instance, you could explore the prostate cancer data as in Chapter 3 of (Hastie et al., 2008); the data are available in the CRAN package ElemStatLearn.

► Exercise 13.4. Consider smoothing as a regression and model selection problem. What is the equivalent quantity to the penalization parameter $\lambda$ in Equation (13.8)? How do you choose it?

► Answer 13.3. We refer to Chapter 5 of (Hastie et al., 2008)

► Exercise 13.5. Scale invariance. Consider a rescaling of one of the features in the (generalized) linear model (13.7). For instance, denote the $v$-th column of $x$ by $x_v$, and suppose that $p \geq 2$ and that we rescale $x_v \mapsto s x_v$ with some number $s \neq 0$. What will happen to the estimate $\hat{\beta}$ from Equation (13.8) in (a) the unpenalized case ($\lambda = 0$) and (b) the penalized case ($\lambda > 0$)?

► Answer 13.4. In the unpenalized case, the estimates will be scaled by $1/s$, so that the resulting model is, in effect, the same. In the penalized case, the penalty from the $v$-th component of $\hat{\beta}$ will be different. If $|s| > 1$, the amplitude of the feature is increased, smaller $\beta$-components are required for it to have the same effect in the prediction, and therefore the feature is more likely to receive a non-zero and/or
larger estimate, possibly on the cost of the other features; conversely for $|s| < 1$. 
Bibliography


Susan Holmes, Michael He, Tong Xu, and Peter P Lee. Memory t cells have gene expression patterns intermediate between naive and effector. PNAS, 102(15):5519–5523, 2005.


David Sims, Ian Sudbery, Nicholas E Ilott, Andreas Heger, and Chris P Ponting. Sequencing depth and coverage: key considerations in


R-project.org/package=GOplot. R package version 1.0.1.


Hui Zou and Trevor Hastie.  
