Use R !

Chemometrics with R

Multivariate Data Analysis in the Natural Sciences and Life Sciences

Springer

Use R!

Series Editors:
Robert Gentleman Kurt Hornik Giovanni Parmigiani

For other titles published in this series, go to
http://www.springer.com/series/6991

Ron Wehrens

Chemometrics with R

Multivariate Data Analysis in the Natural Sciences and Life Sciences

Springer

Ron Wehrens
Fondazione Edmund Mach
Research and Innovation Centre
Via E. Mach 1
38010 San Michele all'Adige
Italy
ron.wehrens@iasma.it

Series Editors:

Robert Gentleman
Program in Computational Biology
Division of Public Health Sciences
Fred Hutchinson Cancer Research Center
1100 Fairview Avenue, N. M2-B876
Seattle, Washington 98109
USA

Kurt Hornik
Department of Statistik and Mathematik
Wirtschaftsuniversität Wien
Augasse 2-6
A-1090 Wien
Austria

Giovanni Parmigiani
The Sidney Kimmel Comprehensive
Cancer Center at Johns Hopkins University
550 North Broadway
Baltimore, MD 21205-2011
USA

ISBN 978-3-642-17840-5 e-ISBN 978-3-642-17841-2
DOI 10.1007/978-3-642-17841-2
Springer Heidelberg Dordrecht London New York

Cover design: deblik, Berlin

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

For Odilia, Chris and Luc

Preface

The natural sciences, and the life sciences in particular, have seen a huge increase in the amount and complexity of data being generated with every experiment. It is only some decades ago that scientists were typically measuring single numbers – weights, extinctions, absorbances – usually directly related to compound concentrations. Data analysis came down to estimating univariate regression lines, uncertainties and reproducibilities. Later, more sophisticated equipment generated complete spectra, where the response of the system is wavelength-dependent. Scientists were confronted with the question how to turn these spectra into useable results such as concentrations. Things became more complex after that: chromatographic techniques for separating mixtures were coupled to high-resolution (mass) spectrometers, yielding a data matrix for every sample, often with large numbers of variables in both chromatographic and spectroscopic directions. A set of such samples corresponds to a data cube rather than a matrix. In parallel, rapid developments in biology saw a massive increase in the ratio of variables to objects in that area as well.

As a result, scientists today are faced with the increasingly difficult task to make sense of it all. Although most will have had a basic course in statistics, such a course is unlikely to have covered much multivariate material. In addition, many of the classical concepts have a rough time when applied to the types of data encountered nowadays – the multiple-testing problem is a vivid illustration. Nevertheless, even though data analysis has become a field in itself (or rather: a large number of specialized fields), scientists generating experimental data should know at least some of the ways to interpret their data, if only to be able to ascertain the quality of what they have generated. Cookbook approaches, involving blindly pushing a sequence of buttons in a software package, should be avoided. Sometimes the things that deviate from expected behaviour are the most interesting in a data set, rather than unfortunate measurement errors. These deviations can show up at any time point during data analysis, during data preprocessing, modelling, interpretation... Every phase in this pipeline should be carefully executed and results, also at an intermediate stage, should be checked using common sense and prior knowledge.

This also puts restrictions on the software that is being used. It should be sufficiently powerful and flexible to fit complicated models and handle large and complex data sets, and on the other hand should allow the user to exactly follow what is being calculated – black-box software should be avoided if possible. Moreover, the software should allow for reproducible results, something that is hard to achieve with many point-and-click programs: even with a reasonably detailed prescription, different users can often obtain quite different results. R [1], with its rapidly expanding user community, nicely fits the bill. It is quickly becoming the most important tool in statistical bioinformatics and related fields. The base system already provides a large array of useful procedures; in particular, the high-quality graphics system should be mentioned. The most important feature, however, is the package system, allowing users to contribute software for their own fields, containing manual pages and examples that are directly executable. The result is that many packages have been contributed by users for specific applications; the examples and the manual pages make it easy to see what is happening.

Purpose of this book.

Something of this philosophy also can be found in the way this book is set up. The aim is to present a broad field of science in an accessible way, mainly using illustrative examples that can be reproduced immediately by the reader. It is written with several goals in mind:

- **An introduction to multivariate analysis.** On an abstract level, this book presents the route from raw data to information. All steps, starting from the data preprocessing and exploratory analysis to the (statistical) validation of the results, are considered. For students or scientists with little experience in handling real data, this provides a general overview that is sometimes hard to get from classical textbooks. The theory is presented as far as necessary to understand the principles of the methods and the focus is on immediate application on data sets, either from real scientific examples, or specifically suited to illustrate characteristics of the analyses.
- **An introduction to R.** For those scientists already working in the fields of bioinformatics, biostatistics and chemometrics but using other software, the book provides an accessible overview on how to perform the most common analyses in R [1]. Many packages are available on the standard repositories, CRAN[1] and BIOCONDUCTOR[2], but for people unfamiliar with the basics of R the learning curve can be pretty steep – for software, power and complexity are usually correlated. This book is an attempt to provide a more gentle way up.

[1] `http://cran.r-project.org`

[2] `http://www.bioconductor.org`

- **Combinding multivariate data analysis and R.** The combination of the previous two goals is especially geared towards university students, at the beginning of their specialization: it is of prime importance to obtain hands-on experience on real data sets. It does take some help to start reading R code – once a certain level has been reached, it becomes more easy. The focus therefore is not just on the use of the many packages that are available, but also on showing how the methods are implemented. In many cases, simplified versions of the algorithms are given explicitly in the text, so that the reader is able to follow step-by-step what is happening. It is this insight in (at least the basics of) the techniques that is essential for fruitful application.

The book has been explicitly set up for self-study. The user is encouraged to try out the examples, and to substitute his or her own data as well. If used in a university course, it is possible to keep the classical "teaching" of theory to a minimum; during the lessons, teachers can concentrate on the analysis of real data. There is no substitute for practice.

Prior knowledge.

Some material is assumed to be familiar. Basic statistics, for example, including hypothesis tests, the construction of confidence intervals, analysis of variance and least-squares regression are referred to, but not explained. The same goes for basic matrix algebra. The reader should have some experience in programming in general (variables, variable types, functions, program control, etcetera). It is assumed the reader has installed R, and has a basic working knowledge of R, roughly corresponding to having worked through the excellent "Introduction to R" [2], which can be found on the CRAN website. In some cases, less mundane functions will receive a bit more attention in the text; examples are the `apply` and `sweep` functions. We will only focus on the comman-line interface: Windows users may find it easier to perform actions using point-and-click.

The R package **ChemometricsWithR***.*

With the book comes a package, too: **ChemometricsWithR** contains all data sets and functions used in this book. Installing the package will cause all other packages used in the book to be available as well – an overview of these packages can be found in Appendix A. In the examples it is always assumed that the **ChemometricsWithR** package is loaded; where functions or data sets from other packages are used for the first time, this is explicitly mentioned in the text.

More information about the data sets used in the book can be found in the references – no details will be given about the background or interpretation of the measurement techniques.

Acknowledgements.

This book has its origins in a reader for the Chemometrics course at the Radboud University Nijmegen covering exploratory analysis (PCA), clustering (hierarchical methods and k-means), discriminant analysis (LDA, QDA) and multivariate regression (PCR, PLS). Also material from a later course in Pattern Recognition has been included. I am grateful for all the feedback from the students, and especially for the remarks, suggestions and criticisms from my colleagues at the Department of Analytical Chemistry of the Radboud University Nijmegen. I am indebted to Patrick Krooshof and Tom Bloemberg, who have contributed in a major way in developing the material for the courses. Finally, I would like to thank all who have read (parts of) the manuscript and with their suggestions have helped improving it, in particular Tom Bloemberg, Karl Molt, Lionel Blanchet, Pietro Franceschi, and Jan Gerretzen.

Trento, *Ron Wehrens*
September 2010

Contents

Part III Modelling

Part IV Model Inspection

Part V Applications

Part VI Appendices

1

Introduction

In the life sciences, molecular biology in particular, the amount of data has exploded in the last decade. Sequencing a whole genome is becoming routine work, and shortly the amount of money needed to do so will be less than the cost of a medium-sized television set. Rather than focussing on measuring specific predefined characteristics of the sample[1] modern techniques aim at generating a holistic view, sometimes called a “fingerprint”. As a result, one analysis of one single sample can easily yield megabytes of data. These physical samples typically are complex mixtures and may, e.g., correspond to body fluids of patients and controls, measured with possibly several different spectroscopic techniques; environmental samples (air, water, soil); measurements on different cell cultures or one cell culture under different treatments; industrial samples from process industry, pharmaceutical industry or food industry; samples of competitor products; quality control samples, and many others. The types of data we will concentrate on are generated by analytical chemical measurement techniques, and are in almost all cases directly related to concentrations or amounts of specific classes of chemicals such as metabolites or proteins. The corresponding research fields are called metabolomics and proteomics, and a host of other -omics sciences with similar characteristics exist. A well-known example from molecular biology is transcriptomics, focussing on the levels of mRNA obtained by transcription from DNA strains. Although we do not include any transcriptomics data, many of the techniques treated in this book are directly applicable – in that sense, the characteristics of data of completely different origins can still be comparable.

These data can be analysed at different levels. The most direct approach is to analyse them as raw data (intensities, spectra, ...), without any prior interpretation other than a suitable pretreatment. Although this has the advantage

[1] The word “sample” will be used both for the physical objects on which measurements are performed (the chemical use of the word) and for the current realization of all possible measurements (the statistical use). Which one is meant should be clear from the context.

R. Wehrens, *Chemometrics with R: Multivariate Data Analysis in the Natural Sciences and Life Sciences*, Use R!, DOI 10.1007/978-3-642-17841-2_1,

that it is completely objective, it is usually also more difficult: typically, the number of variables is huge and the interpretability of the statistical models that are generated to describe the data often is low. A more often used strategy is to apply domain knowledge to convert the raw data into more abstract variables such as concentrations, for example by quantifying a set of compounds in a mixture based on a library of pure spectra. The advantage is that the statistical analysis can be performed on the quantities that really matter, and that the models are simpler and easier to validate and interpret. The obvious disadvantage is the dependence on the interpretation step: not always it is easy to decide which compounds are present and in what amounts. Any error at this stage cannot be corrected in later analysis stages.

The extremely rapid development of analytical techniques in biology and chemistry has left data analysis far behind, and as a result the statistical analysis and interpretation of the data has become a major bottleneck in the pipeline from measurement to information. Academic training in multivariate statistics in the life sciences is lagging. Bioinformatics departments are the primary source of scientists with such a background, but bioinformatics is a very broad field covering many other topics as well. Statistics and machine learning departments are usually too far away from the life sciences to establish joint educational programmes. As a result, scientists doing the data analysis very often have a background in biology or chemistry, and have acquired their statistical skills by training-on-the-job. This can be an advantage, since it makes it easier to interpret results and assess the relevance of certain findings. At the same time, there is a need for easily accessible background material and opportunities for self-study: books like the excellent “The Elements of Statistical Learning” [3] form an invaluable source of information but can also be a somewhat daunting read for scientists without much statistical background.

This book aims to fill the gap, at least to some extent. It is important to combine the sometimes rather abstract descriptions of the statistical techniques with hands-on experience behind a computer screen. In many ways R [1] is the ideal software platform to achieve this – it is extremely powerful, the many add-on packages provide a huge range of functionalities in different areas, and it is freely accessible. As in the other books in this series, the examples can be followed step-by-step by typing or cutting-and-pasting the code, and it is easy to plug in one’s own data. To date, there is only one other book specifically focused on the use of R in a similar field of science: “Introduction to Multivariate Statistical Analysis in Chemometrics” [4] which to some extent complements the current volume, in particular in its treatment of robust statistics.

Here, the concepts behind the most important data analysis techniques will be explained using a minimum of mathematics, but in such a way that the book still can be used as a student’s text. Its structure more or less follows the steps made in a “classical” data analysis, starting with the *data pretreatment* in Part I. This step is hugely important, yet is often treated only cursorily. An unfortunate choice here can destroy any hope of achieving good results:

background knowledge of the system under study as well as the nature of the measurements should be used in making decisions. This is where science meets art: there are no clear-cut rules, and only by experience we will learn what the best solution is.

The next phase, subject of Part II, consists of *exploratory analysis*. What structure is visible? Are there any outliers? Which samples are very similar, which are different? Which variables are correlated? Questions like these are most easily assessed by eye – the human capacity for pattern recognition in two dimensions is far superior to any statistical method. The methods at this stage all feature strong visualization capabilities. Usually, they are model-free; no model is fitted, and the assumptions about the data are kept to a minimum.

Once we are at the *modelling* phase, described in Part III, we very often do make assumptions: some models work optimally with normally distributed data, for example. The purpose of modelling can be twofold. The first is prediction. Given a set of analytical data, we want to be able to predict properties of the samples that cannot be measured easily. An example is the assessment of whether a specific treatment will be useful for a patient with particular characteristics. Such an application is known as *classification* – one is interested in modelling class membership (will or will not respond). The other major field is *regression*, where the aim is to model continuous real variables (blood pressure, protein content, ...). Such predictive models can mean a big improvement in quality of life, and save large amounts of money. The prediction error is usually taken as a quality measure: a model that is able to predict with high accuracy must have captured some real information about the system under study. Unfortunately, in most cases no analytical expressions can be derived for prediction accuracy, and other ways of estimating prediction accuracy are required in a process called *validation*. A popular example is crossvalidation.

The second aim of statistical modelling is *interpretation*, one of the topics in Part IV. Who cares if the model is able to tell me that this is a Golden Delicious apple rather than a Granny Smith? The label in the supermarket already told me so; but the question of course is why they taste different, feel different and look different. Fitting a predictive model in such a case may still be informative: when we are able to find out why the model makes a particular prediction, we may be able to learn something about the underlying physical, chemical or biological processes. If we know that a particular gene is associated with the process that we are studying, and both this gene and another one show up as important variables in our statistical model, then we may deduce that the second gene is also involved. This may lead to several new hypotheses that should be tested in the lab. Obviously, when a model has little or no predictive ability it does not make too much sense to try and extract this type of information.

Our knowledge of the system can also serve as a tool to assess the quality of our model. A model that fits the data and seems to be able to predict well is not going to be very popular when its parameters contradict what we know about the underlying process. Often, prior knowledge is available (we expect a peak at a certain position; we know that model coefficients should not be negative; this coefficient should be larger than the other), and we can use that knowledge to assess the relevance of the fitted model. Alternatively, we can constrain the model in the training phase to take prior knowledge into account, which is often done with constraints. In other cases, the model is hard to interpret because of the sheer number of coefficients that have been fitted, and graphical summaries may fail to show what variables contribute in what way. In such cases, *variable selection* can come to the rescue: by discarding the majority of the variables, hopefully without compromising the model quality, one can often improve predictions *and* make the model much more easy to interpret. Unfortunately, variable selection is an NP-complete problem (which in practice means that even for moderate-sized systems it may be impossible to assess all possible solutions) and one never can be sure that the optimal solution has been found. But then again, any improvement over the original, full, model is a bonus.

For each of the stages in this “classical” data analysis pipeline, a plethora of methods is available. It can be hard to assess which techniques should be considered in a particular problem, and perhaps even more importantly, which should not. The view taken here is that the simplest possibilities should be considered first; only when the results are unsatisfactory, one should turn to more complex solutions. Of course, this is only a very crude first approach, and experienced scientists will have devised many shortcuts and alternatives that work better for their types of data. In this book, I have been forced to make choices. It is impossible to treat all methods, or even a large subset, in detail. Therefore the focus is on an ensemble of methods that will give the reader a broad range of possibilities, with enough background information to acquaint oneself with other methods, not mentioned in this book, if needed. In some cases, methods deserve a mention because of the popularity within the bioinformatics or chemometrics communities. Such methods, together with some typical applications, are treated in the final part of the book.

Given the huge number of packages available on CRAN and the speed with which new ones appear, it is impossible to mention all that are relevant to the material in this book. Where possible, I have limited myself to the recommended packages, and those coming with a default R installation. Of course, alternative, perhaps even much simpler, solutions may be available in the packages that this book does not consider. It pays to periodically scan the CRAN and Bioconductor repositories, or, e.g., check the Task Views that provide an overview of all packages available in certain areas – there is one on Physics and Chemistry, too.

Part I

Preliminaries

2

Data

In this chapter, some typical data sets are presented, several of which will occur throughout the book. All data sets are accessible, either through one of the packages mentioned in the text, or in the **ChemometricsWithR** package. Chemical data sets nowadays are often characterized by a relatively low number of samples and a large number of variables, a result of the predominant spectroscopic measuring techniques enabling the chemist to rapidly acquire a complete spectrum for one sample. Depending on the actual technique employed, the number of variables can vary from several hundreds (typical in infrared measurements) to tens of thousands (e.g., in Nuclear Magnetic Resonance, NMR). A second characteristic is the high correlation between variables: neighbouring spectral variables usually convey very similar information. An example is shown in Figure 2.1, depicting the gasoline data set, one of several data sets that will be used throughout this book. It shows near-infrared (NIR) spectra of sixty gasolines at wavelengths from 900 to 1700 nm in 2 nm intervals [5], and is available in the **pls** package. The plot is made using the following piece of code:

```
> data(gasoline, package = "pls")
> wavelengths <- seq(900, 1700, by = 2)
> matplot(wavelengths, t(gasoline$NIR), type = "l",
+         lty = 1, xlab = "Wavelength (nm)", ylab = "1/R")
```

The `matplot` function is used to plot all columns of matrix `t(gasoline$NIR)` (or, equivalently, all rows of matrix `gasoline$NIR`) against the specified wavelengths. Clearly, all samples have very similar features – it is impossible to distinguish individual samples in the plot. NIR spectra are notoriously hard to interpret: they consist of a large number of heavily overlapping peaks which leads to more or less smooth spectra. Nevertheless, the technique has proven to be of immense value in industry: it is a rapid, non-destructive method of analysis requiring almost no sample preprocessing, and it can be used for quantitative predictions of sample properties. The data used here can be used to quantitatively assess the octane number of the gasoline samples, for instance.

R. Wehrens, *Chemometrics with R: Multivariate Data Analysis in the Natural Sciences and Life Sciences*, Use R!, DOI 10.1007/978-3-642-17841-2_2,

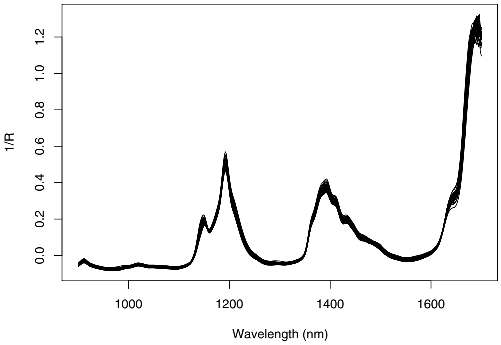

Fig. 2.1. Near-infrared spectra of sixty gasoline samples, consisting of 401 reflectance values measured at equally spaced wavelengths between 900 and 1700 nm.

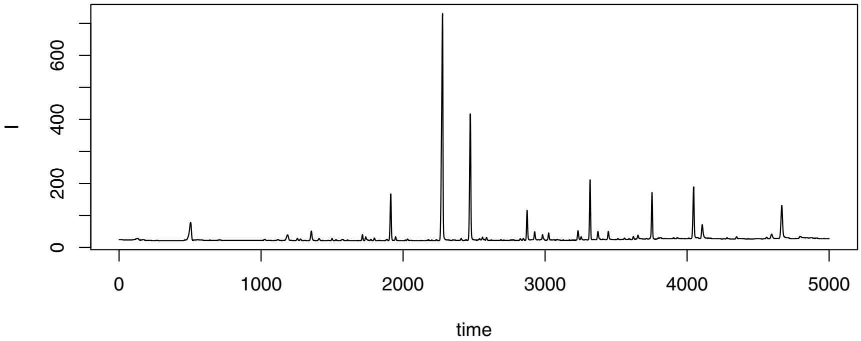

Fig. 2.2. The first gas chromatogram of data set `gaschrom` from the **ptw** package.

In other cases, specific variables can be directly related to absolute or relative concentrations. An example is the `gaschrom` data set from the **ptw** package, containing gas chromatograms measured for calibration purposes. The first sample is shown in Figure 2.2. Each feature, or peak, corresponds to the elution of a compound, or in more complex cases, a number of overlapping compounds. These peaks can be easily quantified, usually by measuring peak area, but sometimes also by peak height. Since the number of features usually is orders of magnitude smaller than the number of variables in the original data, summarising the chromatograms with a peak table containing position and intensity information can lead to significant data compression.

An example in which most of the variables correspond to concentrations is the wine data set, used throughout the book. It is a set consisting of 177 wine samples, with thirteen measured variables [6]:

```
> data(wines, package = "kohonen")
> colnames(wines)
```

```
 [1] "alcohol"          "malic acid"       "ash"
 [4] "ash alkalinity"   "magnesium"        "tot. phenols"
 [7] "flavonoids"       "non-flav. phenols" "proanth"
[10] "col. int."        "col. hue"         "OD ratio"
[13] "proline"
```

Variables are reported in different units. All variables apart from `"col. int."`, `"col. hue"` and `"OD ratio"` are concentrations. The meaning of the variables color intensity and color hue is obvious; the OD ratio is the ratio between the absorbance at wavelengths 280 and 315 nm. All wines are from the Piedmont region in Italy. Three different classes of wines are present: Barolo, Grignolino and Barberas. Barolo wine is made from Nebbiolo grapes; the other two wines have the name of the grapes from which they are made. Production areas are partly overlapping [6].

```
> table(vintages)
```

```
vintages
   Barbera     Barolo Grignolino
        48         58         71
```

The obvious aim in the analysis of such a data set is to see whether there is any structure that can be related to the three cultivars. Possible questions are: “which varieties are most similar?”, “which variables are indicative of the variety?”, “can we discern subclasses within varieties?”, etcetera.

A quick overview of the first few variables can be obtained with a so-called pairs plot[1]:

```
> pairs(wines[,1:3], pch = wine.classes, col = wine.classes)
```

This leads to the plot shown in Figure 2.3. It is clear that the three classes can be separated quite easily – consider the plot of alcohol against malic acid, for example.

A further data set comes from mass spectrometry. It contains 327 samples from three groups: patients with prostate cancer, benign prostatic hyperplasia, and normal controls [7, 8]. The data have already been preprocessed (binned, baseline-corrected, normalized – see Chapter 3). The m/z values range from 200 to 2000 Dalton. The data set is available in the R package **msProstate**:

[1] Gray-scale figures such as shown throughout the book are obtained by, e.g., `col = gray(0:2/4)[wine.classes]`.
In the text and the code we will in almost all cases use the default R colour palette.

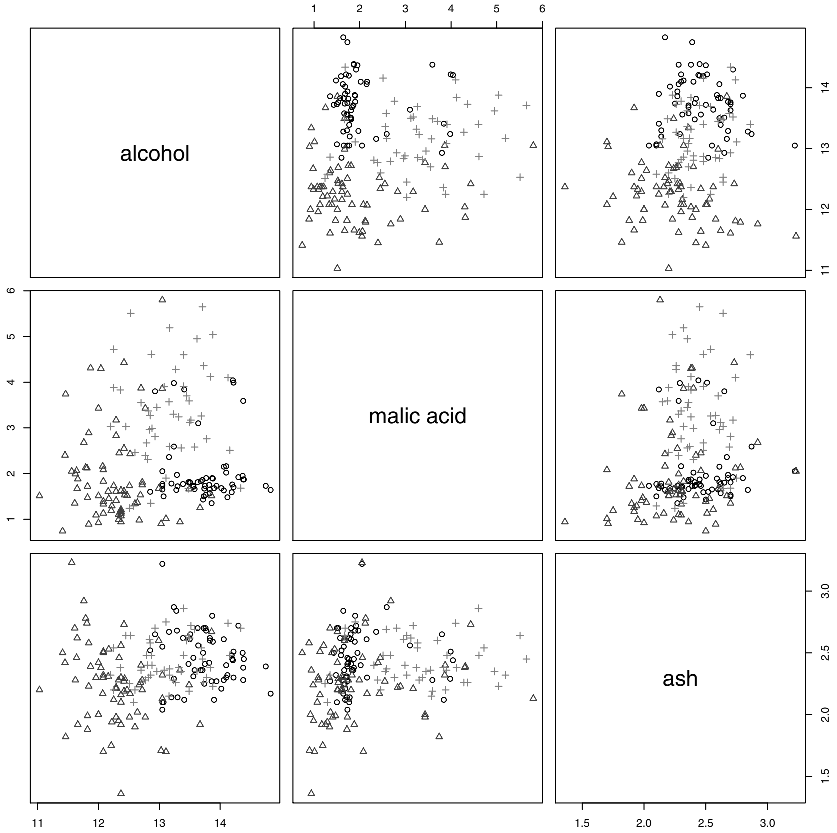

Fig. 2.3. A pairs plot of the first three variables of the wine data. The three vintages are indicated with different shades of gray and plotting symbols: Barbera wines are indicated with black circles, Barolos with dark gray triangles and Grignolinos with gray plusses.

```
> data(Prostate2000Raw, package = "msProstate")
> plot(Prostate2000Raw$mz, Prostate2000Raw$intensity[,1],
+      type = "h", xlab = "m/z", ylab = "Intensity",
+      main = "Prostate data")
```

Figure 2.4 shows the first mass spectrum, that of a healthy control sample. In total, there are 168 tumour samples, 81 controls, and 78 cases of benign prostate enlargement: all samples have been measured in duplicate.

```
> table(Prostate2000Raw$type)

    bph control     pca
    156     162     336
```

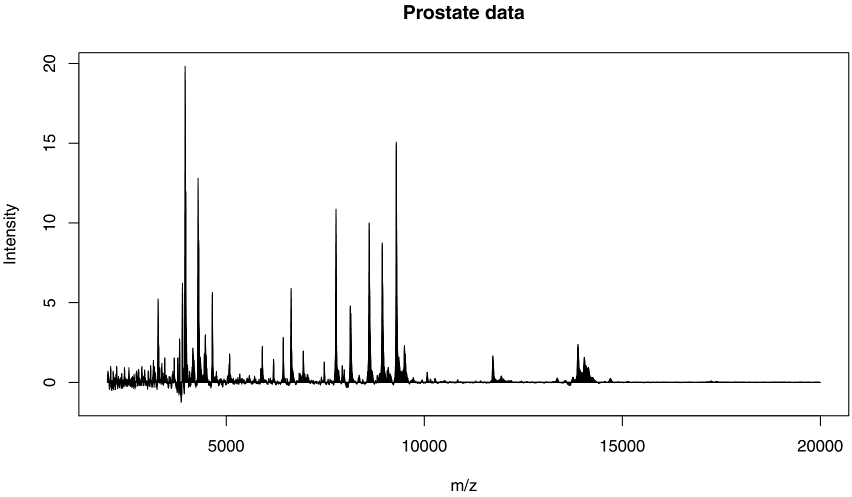

Fig. 2.4. The first mass spectrum in the prostate MS data set.

Such data can serve as diagnostic tools to distinguish between healthy and diseased tissue, or to differentiate between several disease states. The number of samples is almost always very low – for rare diseases, patients are scarce, and stratification to obtain relatively homogeneous groups (age, sex, smoking habits, ...) usually does the rest; and in cases where the measurement is unpleasant or dangerous it may be difficult or even unethical to get data from healthy controls. On the other hand, the number of variables per sample is often huge. This puts severe restrictions on the kind of analysis that can be performed and makes thorough validation even more important.

The final data set in this chapter comes from LC-MS, the combination of liquid chromatography and mass spectrometry. The chromatography step serves to separate the components of a mixture on the basis of properties like polarity, size, or affinity. At specific time points a mass spectrum is recorded, containing the counts of particles with specific mass-to-charge (m/z) ratios. Measuring several samples therefore leads to a data cube of dimensions `ntime`, `nmz`, and `nsample`; the number of timepoints is typically in the order or thousands, whereas the number of samples rarely exceeds one hundred. Mass spectra can be recorded at a very high resolution and to enable statistical analysis, m/z values are typically *binned* (or "bucketed"). Even then, thousands of variables are no exception. Package **ptw** provides a data set, `lcms`, containing data on three tryptic digests of E. coli proteins [9]. Figure 2.5 shows a top view of the first sample, with projections to the top and right of the main plot. The top projection leads to the "Total Ion Current" (TIC) chromatogram, and would be obtained if there would be no separation along the m/z axis;

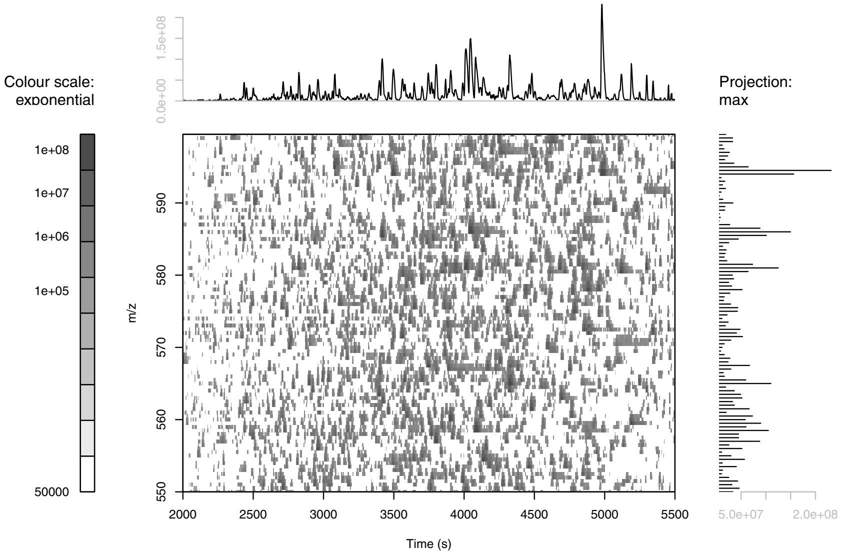

Fig. 2.5. Top view of the first sample in data set `lcms`. The TIC chromatogram is shown on the top, and the direct infusion mass spectrum on the right.

similarly, if the chromatographic dimension would be absent, the mass spectrum of the whole sample would be very close to the projection on the right (a "direct infusion" spectrum). The whole data set consists of three of such planes, leading to a data cube of size 100 × 2000 × 3.

3

Preprocessing

Textbook examples typically use clean, perfect data, allowing the techniques of interest to be explained and illustrated. However, in real life data are messy, noisy, incomplete, downright faulty, or a combination of these. The first step in any data analysis often consists of preprocessing to assess and possibly improve data quality. This step may actually take more time than the analysis itself, and more often than not the process consists of an iterative procedure where data preprocessing steps are alternated with data analysis steps.

Some problems can immediately be recognized, such as measurement noise, spikes, and unrealistic values. In these cases, taking appropriate action is rarely a problem. More difficult are the cases where it is not obvious which characteristics of the data contain information, and which do not. There are many examples where chance correlations lead to statistical models that are perfectly able to model the training data but have no predictive abilities whatsoever.

This chapter will focus on standard preprocessing techniques used in the natural sciences and the life sciences. Data are typically spectra or chromatograms, and topics include noise reduction, baseline removal, peak alignment, peak picking, and scaling. Only the basic general techniques are mentioned here; some more specific ways to improve the quality of the data will be treated in later chapters. Examples include Orthogonal Partial Least Squares for removing uncorrelated variation (Section 11.2) and variable selection (Chapter 10).

3.1 Dealing with Noise

Physico-chemical data always contain noise, where the term “noise” is usually reserved for small, fast, random fluctuations of the response. The first aim of any scientific experiment is to generate data of the highest quality, and much effort is usually put into decreasing noise levels. The simplest experimental way is to perform n repeated measurements, and average the individual spectra, leading to a noise reduction with a factor $\sqrt{n}$. In NMR spectroscopy, for

R. Wehrens, *Chemometrics with R: Multivariate Data Analysis in the Natural Sciences and Life Sciences*, Use R!, DOI 10.1007/978-3-642-17841-2_3,

example, a relatively insensitive analytical method, signal averaging is routine practice, where one has to strike a balance between measurement time and data quality.

As an example, we consider the prostate data, where each sample has been measured in duplicate. The replicate measurements of the prostate data cover consecutive rows in the data matrix. Averaging can be done using the following steps:

```
> prostate.array <- array(t(Prostate2000Raw$intensity),
+                         c(2, 327, 10523))
> prostate <- apply(prostate.array, c(2,3), mean)
> dim(prostate)
```

```
[1]   327 10523
```

The idea is to convert the matrix into an array where the first dimension contains the two replicates for every sample – each element in the first dimension thus contains one complete set of measurements. The final data matrix is obtained by averaging the replicates. The function `apply` is useful here: it (indeed) applies the function given by the third argument to the data indicated by the first argument while keeping the dimensions indicated by the second – got that? In this code snippet the outcome of `apply` is a matrix having dimensions equal to the second and third dimensions of the input array. The first dimension is averaged out. As we will see later, `apply` is extremely handy in many situations. There is, however, also another much faster possibility using `rowsum`:

```
> prostate <- rowsum(t(Prostate2000Raw$intensity),
                     group = rep(1:327, each = 2),
                     reorder = FALSE) / 2
> dim(prostate)
```

```
[1]   327 10523
```

This function sums all the rows for which the grouping variable (the second argument) is equal. Since there are two replicates for every sample, the result is divided by two to get the average values, and is stored in variable `prostate`. For this new variable we should also keep track of the corresponding class labels:

```
> prostate.type <- Prostate2000Raw$type[seq(1, 654, by = 2)]
```

To plot the result, we combine the original data with the averaged data in a three-column matrix `x`:

```
> x <- cbind(prostate[1,1:500],
+            Prostate2000Raw$intensity[1:500, 1:2])
```

The result of the signal averaging is visualized in Figure 3.1, again using the function `matplot`:

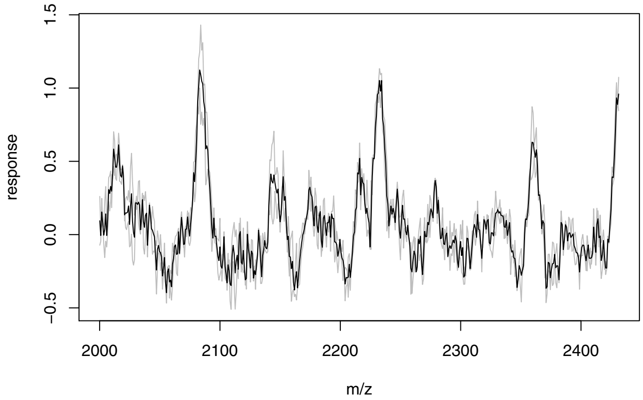

Fig. 3.1. The first averaged mass spectrum in the Prostate data set; only the first 500 m/z values are shown. Original data are in gray.

```
> matplot(Prostate2000Raw$mz[1:500], x, type = "l",
+         col = c(1, "gray", "gray"), lty = c(1,2,2),
+         xlab = "m/z", ylab = "response")
```

Clearly, despite the averaging, the noise is appreciable; reducing the noise level while taking care not to destroy the data structure would make subsequent analysis much easier.

The simplest approach is to apply a *running mean*, i.e. to replace every single value by the average of the k points around it. The value of k needs to be optimized; large values lead to a high degree of smoothing, but also to peak distortion, and low values of k can only make small changes to the signal. Running means can be easily calculated using the function `embed`, providing is a matrix containing as rows successive chunks of the original data vector; using the function `rowMeans` one then can obtain the desired running means.

```
> rmeans <- rowMeans(embed(prostate[1,1:500], 5))
> plot(Prostate2000Raw$mz[1:500], prostate[1,1:500],
+      type = "l", xlab = "m/z", ylab = "response",
+      main = "running means", col = "gray")
> lines(Prostate2000Raw$mz[3:498], rmeans, type = "l")
```

As can be seen in the left plot in Figure 3.2, the smoothing effectively reduces the noise level. Note that the points at the extremes need to be treated separately in this implementation. The price to be paid is that peak heights are decreased, and especially with larger spans one will see appreciable peak broadening. These effects can sometimes be countered by using running medi-

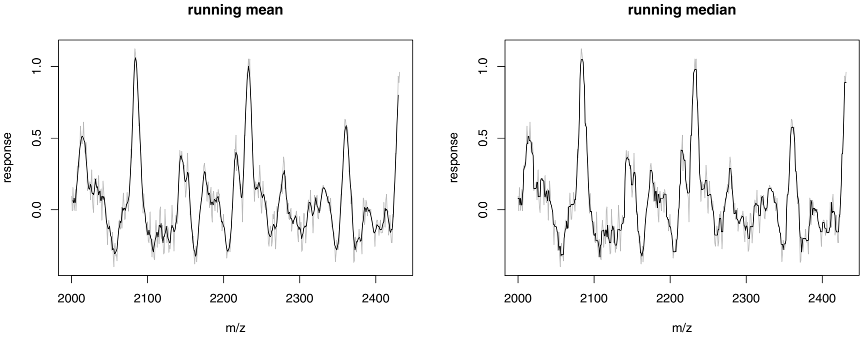

Fig. 3.2. Smoothing of the averaged mass spectrum of Figure 3.1: a running mean (left plot) and a running median (right plot), both with a window size of five.

ans instead of running means. The function `runmed`, part of the **stats** package, is available for this:

```
> plot(Prostate2000Raw$mz[1:500], prostate[1,1:500],
+      type = "l", xlab = "m/z", ylab = "response",
+      main = "running median", col = "gray")
> lines(Prostate2000Raw$mz[1:500],
+       runmed(prostate[1,1:500], k = 5), type = "l")
```

The result is shown in the right plot in Figure 3.2. Its appearance is less smooth than the running mean with the same window size; in particular, peak shapes seem less natural. Note that the function `runmed` does return a vector with the same length as the input – the points at the extremes are left unchanged. The plots of the residuals in Figure 3.3 show that both smoothing techniques do quite a good job in removing high-frequency noise components without distorting the signal too much.

Many other smoothing functions are available – only a few will be mentioned here briefly. In signal processing, Savitsky-Golay filters are a popular choice [10]; every point is replaced by a smoothed estimate obtained from a local polynomial regression. An added advantage is that derivatives can simultaneously be calculated (see below). In statistics, robust versions of locally weighted regression [11] are often used; `loess` and its predecessor `lowess` are available in R as simple-to-use implementations. The fact that the fluctuations in the noise usually are much faster than the data has led to a whole class of frequency-based smoothing methods, of which wavelets [12] are perhaps the most popular ones. The idea is to set the coefficients for the high-frequency components to zero, which should leave only the signal component.

A special kind of smoothing is formed by *binning*, also called bucketing, which not only averages consecutive values but also decreases the number of variables. To replace five data points with their average, one can use:

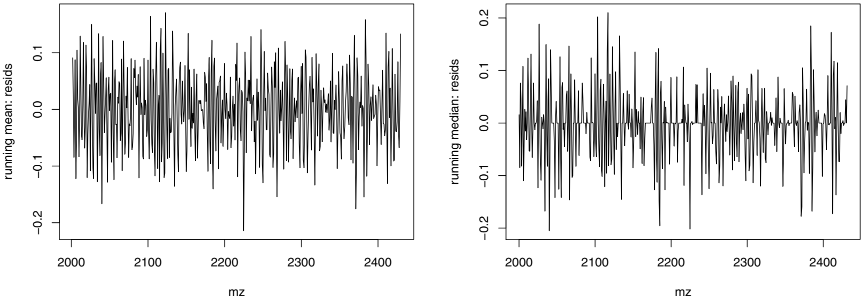

Fig. 3.3. Residuals of smoothing the first spectrum of the prostate data with a running mean (left plot) or running median (right).

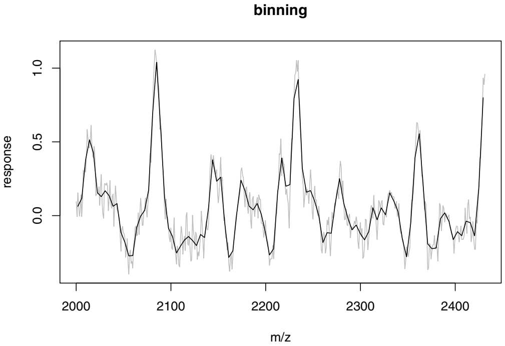

Fig. 3.4. Binned version of the mass spectrometry data from Figure 3.1. Five data points constitute one bin.

```
> mznew <- colMeans(matrix(Prostate2000Raw$mz[1:500], nrow = 5))
> xnew <- colMeans(matrix(prostate[1, 1:500], nrow = 5))
> plot(Prostate2000Raw$mz[1:500], prostate[1, 1:500],
+      type = "l", xlab = "m/z", ylab = "response",
+      main = "binning", col = "gray")
> lines(mznew, xnew)
```

We have seen the idea before: in this case we fill a matrix with five rows column-wise with the data, and then average over the rows[1]. This leads to the plot in Figure 3.4. Obviously, the binned representation gives a cruder description of the data, but still is able to follow the main features. Again, determining the optimal bin size is a matter of trial and error. Binning has several major advantages over running means and medians. First, it can be

[1] What would the code look like using `rowsum`?

applied when data are not equidistant and even when the data are given as positions and intensities of features, as is often the case with mass-spectrometric data. Second, the effect of peak shifts (see below) is decreased: even when a peak is slightly shifted, it will probably be still within the same bin. And finally, more often than not the variable-to-object ratio is extremely large in data sets from the life sciences. Summarising the information in fewer variables in many cases makes the subsequent statistical modelling more easy.

Although smoothing leads to data that are much better looking, one should also be aware of the dangers. Too much smoothing will remove features, and even when applied prudently, the noise characteristics of the data will be different. This may significantly affect statistical modelling.

3.2 Baseline Removal

In some forms of spectroscopy one can encounter a baseline, or "background signal" that is far away from the zero level. Since this influences measures like peak height and peak area, it is of utmost importance to correct for such phenomena.

Infrared spectroscopy, for instance, can lead to scatter effects – the surface of the sample influences the measurement. As a result, one often observes spectral offsets: two spectra of the same material may show a constant difference over the whole wavelength range. This may be easily removed by taking first derivatives (i.e., looking at the *differences* between intensities at sequential wavelengths, rather than the intensities themselves). Take a look at the gasoline data:

```
> nir.diff <- t(apply(gasoline$NIR, 1, diff))
> matplot(wavelengths[-1] + 1, t(nir.diff),
+         type = "l", xlab = "Wavelength (nm)",
+         ylab = "1/R (1st deriv.)", lty = 1, col = 1)
```

Note that the number of variables decreases by one. The result is shown in Figure 3.5. Comparison with the original data (Figure 2.1) shows more detailed structure; the price is an increase in noise. A better way to obtain first-derivative spectra is given by the Savitsky-Golay filter, which is not only a smoother but can also be used to calculate derivatives:

```
> nir.deriv <- apply(gasoline$NIR, 1, sgolayfilt, m = 1)
```

In this particular case, the differences between the two methods are very small.

Another way to remove scatter effects in infrared spectroscopy is Multiplicative Scatter Correction (MSC, [13,14]). One effectively models the signal of a query spectrum as a linear function of the reference spectrum:

$$y_q = a + by_r$$

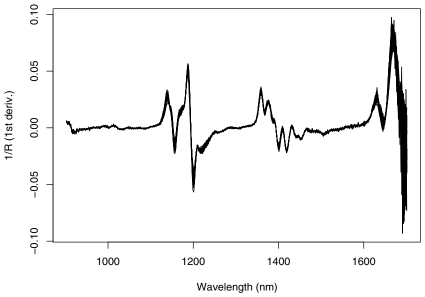

Fig. 3.5. First-derivative representation of the gasoline NIR data.

An obvious reference spectrum may not be available, and then often a mean spectrum is used. This is also the approach in the `msc` function of the **pls** package:

```
> nir.msc <- msc(gasoline$NIR)
```

For the gasoline data, the differences are quite small.

In more difficult cases, a non-constant baseline drift can be observed. First derivatives are not enough to counter such effects, and one has to resort to techniques that actually estimate the shape of the baseline. The exact function is usually not important – the baseline will be subtracted and that is it. To illustrate this point, consider the first chromatogram in the `gaschrom` data. One very simple solution is to connect local minima, obtained from, e.g., 200-point sections:

```
> x <- gaschrom[1,]
> lsection <- 200
> xmat <- matrix(x, nrow=lsection)
> ymin <- apply(xmat, 2, min)
> plot(x, type = "l", col = "gray", ylim = c(20, 50),
+      xlab = "Time", ylab = "I")
> lines(rep(ymin, each = lsection))
```

We have used the by now familiar trick to convert a vector to a matrix and calculate minimal values for every column to obtain the intensity levels of the horizontal line segments. The result is shown in the left plot of Figure 3.6. Obviously, a more smooth baseline estimate would be better. One function that can be used is `loess`, fitting local polynomials through the minimal values:

```
> minlocs <- seq(lsection/2 + 1, length(x), len = length(ymin))
> bsln.loess <- loess(ymin ~ minlocs)
> lines(predict(bsln.loess, 1:length(x)), lwd = 2)
```

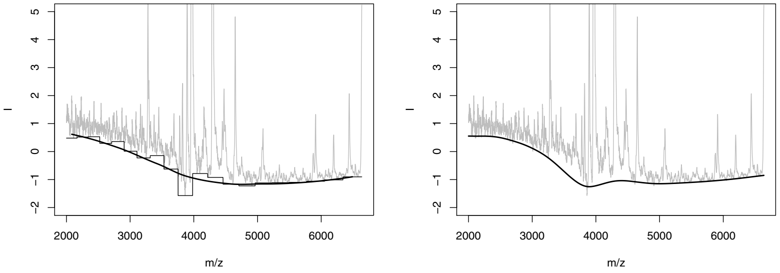

Fig. 3.6. Simple baseline correction for the first chromatogram in the `gaschrom` data: in the left plot the baseline is estimated by a series of twenty local minima, the connected horizontal segments. The thick line indicates the loess smooth (using default settings) of these minima. Right plot: asymmetric least squares estimate of the baseline.

This leads to the right plot in Figure 3.6. The `loess` solution clearly is much smoother and does not follow the data that much. For the current distortion, that is all right, but in some cases one may want a different behaviour. Twiddling with the settings (the length of the segments to find local minima, and `loess` smoother settings) can lead to even better results.

Another alternative is to use asymmetric least squares, where deviations above the fitted curve are not taken into account (or only with a very small weight). This is implemented in function `baseline.corr` in the **ptw** package, which returns a baseline-corrected signal. Internally, it uses the function `asysm` to estimate the baseline:

```
> plot(x, col = "gray", type = "l", ylim = c(20, 50),
+      xlab = "Time", ylab = "I")
> lines(asysm(x), lwd = 2)
```

The result is shown in the right plot of Figure 3.6. Again, the parameters of the `asysm` function may be tweaked to get optimal results.

3.3 Aligning Peaks – Warping

Many analytical data suffer from small shifts in peak positions. In NMR spectroscopy, for example, the position of peaks may be influenced by the pH. What complicates matters is that in NMR, these shifts are by no means uniform over the data; rather, only very few peaks shift whereas the majority will remain at their original locations. The peaks may even move in different directions. In mass spectrometry, the shift is more uniform over the m/z axis and is more easy to account for – if one aims to analyse the data in matrix form, binning is required, and in many cases a suitable choice of bins will

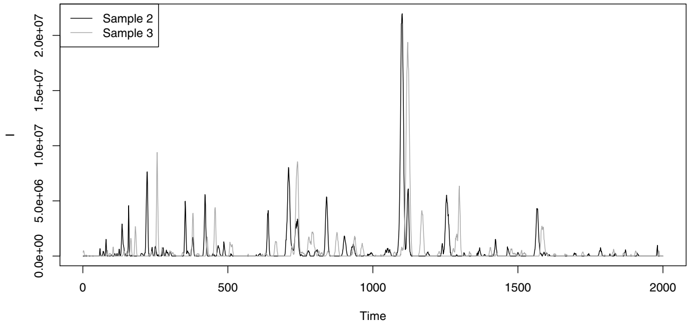

Fig. 3.7. Comparison of two mass chromatograms of the `lcms` data set. Clearly, corresponding features are not in the same positions.

already remove most if not all of the effects of shifts. Moreover, peak shifts are usually small, and may be easily corrected for by the use of standards.

The biggest shifts, however, are encountered in liquid chromatography. Two different chromatographic columns almost never give identical elution profiles, up to the extent that peaks may even swap positions. The situation is worse than in gas chromatography, since retention mechanisms are more complex in the liquid phase than in the gas phase. Moreover, ageing is an important factor in column quality, and a column that has been used for some time almost certainly will show different chromatograms than when freshly installed.

Peak shifts pose significant problems in modelling. In Figure 3.7 the first mass chromatograms in two of the samples of the `lcms` data are shown:

```
> plot(lcms[1,,2], type = "l", xlab = "Time", ylab = "I")
> lines(lcms[1,,3], type = "l", col = "gray")
```

Clearly, both chromatograms contain the same features, although at different locations – the shift is, equally clearly, not constant over the whole range. Comparing these chromatograms with a distance-based similarity function based on Euclidean distance, comparing signals at the same time points, will lead to the conclusion that there are huge differences, whereas the chromatograms in reality are very similar.

Correction of such shifts is known as “time warping”, one of the more catchy names in data analysis. The technique originates in speech processing [15, 16], and nowadays many forms exist. The most popular in the literature for natural sciences and life sciences are Dynamic Time Warping (DTW, [17]), Correlation-Optmized Warping (COW, [18]) and Parametric Time Warping (PTW, [19]). Often, the squared differences between the two signals are used as optimization criterion; this is the case for DTW and the original version of PTW [19]. The R package **ptw** [9] also provides a measure called the *weighted* *cross correlation* (WCC, [20]) to assess the similarity of two patterns – note that in this context the WCC is used as a distance measure so that a value of zero indicates perfect alignment [9]. COW maximizes the correlation between patterns, where the signals are cut into several segments which are treated separately. Both DTW and PTW currently are available as R packages on CRAN, as well as a penalized form of dynamic time warping (package **VPdtw**, [21]).

3.3.1 Parametric Time Warping

In PTW, one approximates the time axis of the reference signal by applying a polynomial transformation of the time axis of the sample [19]:

$$\hat{S}(t_k) = S(w(t_k)) \tag{3.1}$$

where $\hat{S}(t_k)$ is the value of the warped signal at time point t_k, where k is an index. The warping function, w, is given by:

$$w(t) = \sum_{j=0}^{J} a_j t^j \tag{3.2}$$

with J the maximal order of the polynomial. Only low-order polynomials are used in general. Since neighbouring points on the time axis will be warped with almost the same amount, peak shape distortions are limited. The optimization then finds the set of coefficients $a_0, ...a_J$ that minimizes the difference between the sample S and reference R, using whatever difference measure is desired.

This procedure is very suitable for modelling the gradual deterioration of chromatographic columns, so that measurements taken days or weeks apart can still be made comparable. For situations where a few individual peak shifts have to be corrected (e.g. pH-dependent shifting of patterns in NMR spectra), the technique is less ideal.

We will illustrate the use of PTW by aligning the mass chromatograms of Figure 3.7. We will use sample number 2 as a reference, and warp the third sample so that the peak positions show maximal overlap:

```
> sref <- lcms[1,,2]
> ssamp <- lcms[1,,3]
> lcms.warp <- ptw(sref, ssamp, init.coef = c(0, 1, 0))
> summary(lcms.warp)
```

```
PTW object: single alignment of 1 sample on 1 reference.

Warping coefficients:
         [,1]     [,2]         [,3]
[1,] 41.83585 0.974073 5.479969e-06

Warping criterion: WCC
Value: 0.08781744
```

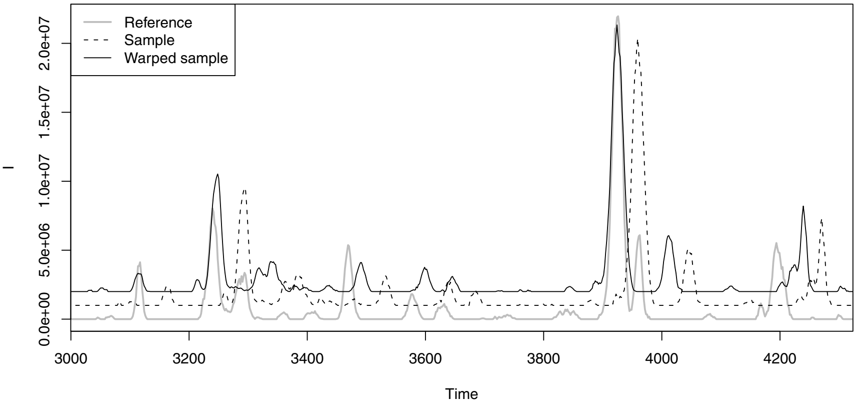

Fig. 3.8. PTW of the data shown in Figure 3.7, using a quadratic warping function. Small offsets have been added to the sample and warped sample spectra.

Using the default quadratic warping function with initial values `init.coef = c(0, 1, 0)`, corresponding to the unit warp (no shift, unit stretch, no quadratic warping), we arrive at a warping where the sample is shifted almost 42 points to the right, is compressed almost 3%, and experiences also a quadratic warping (with a small coefficient). The result is an agreement of just under 0.09, according to the default WCC criterion. A visual check confirms that the peak alignment is much improved:

```
> plot(time, sref, type = "l", lwd = 2, col = "gray",
+      xlim = c(time[600], time[1300]),
+      xlab = "Time", ylab = "I")
> lines(time, ssamp + 1e6, lty = 2)
> lines(time, lcms.warp$warped.sample + 2e6)
> legend("topleft", lty = c(1,2,1), col = c("gray", 1, 1),
+        legend = c("Reference", "Sample", "Warped sample"),
+        lwd = c(2, 1, 1))
```

The result is shown in Figure 3.8. To show the individual traces more clearly, a small vertical offset has been applied to both the unwarped and warped sample. Obviously, the biggest gains can be made at the largest peaks, and in the warped sample the features around 3250 and 3900 seconds are aligned really well. Nevertheless, some other features, such as the peaks between 3450 and 3650 seconds, and the peaks at 3950 and 4200 seconds still show shifts. This simple quadratic warping function apparently is not flexible enough to iron out these differences. Note that RMS-based warping in this case leads to very similar results:

```
> lcms.warpRMS <- ptw(sref, ssamp, optim.crit = "RMS")
> lcms.warpRMS$warp.coef
```

```
         [,1]      [,2]         [,3]
[1,] 40.84908 0.9719185 9.619694e-06
```

More complex warping functions, fitting polynomials of degrees three to five, can be tried:

```
> lcms.warp2 <- ptw(sref, ssamp, init = c(0, 1, 0, 0))
> lcms.warp3 <- ptw(sref, ssamp, init = c(0, 1, 0, 0, 0))
> lcms.warp4 <- ptw(sref, ssamp, init = c(0, 1, 0, 0, 0, 0))
```

To visualize these warping functions, we first gather all warpings in one list, and obtain the qualities of each element using a close relative of the apply function, sapply:

```
> allwarps <- list(lcms.warp, lcms.warp2, lcms.warp3, lcms.warp4)
> wccs <- sapply(allwarps, function(x) x$crit.value)
> wccs <- round(wccs*1000) / 1000
```

Where apply operates on rows or columns of a matrix, sapply performs actions on list elements, and returns the result in a simple way, in this case a matrix. We use the round trick to avoid having too many digits in the plot later on.

Because we are interested in the deviations from the identity warp (i.e. no change), we subtract that from the warping functions:

```
> allwarp.funs <- sapply(allwarps, function(x) x$warp.fun)
> warpings <- allwarp.funs - 1:length(sref)
```

Finally, we can plot the columns of the resulting matrix using the matplot function:

```
> matplot(warpings, type = "l", lty = rep(c(1,2), 2),
+         col = rep(c(1,"gray"), each = 2))
> legend("topleft", lty = rep(c(1,2), 2),
+        col = rep(c(1,"gray"), each = 2),
+        legend = paste("Degree", 2:5, " - WCC =", wccs))
```

This leads to Figure 3.9. A horizontal line here would indicate the identity warp. The fourth-degree warping shows the biggest deviation from the unwarped signal, especially in the right part of the chromatogram, but is rewarded by the lowest distance between sample and reference. The warping functions for degrees three and five are almost identical. This shows, in fact, that it is possible to end up in a local minimum: the fifth-degree warping function should be at least as good as the fourth-degree function. It is usually a good idea to use several different starting values in these situations.

One very important characteristic of LC-MS data is that it contains multiple m/z traces. If the main reason for differences in retention time is the

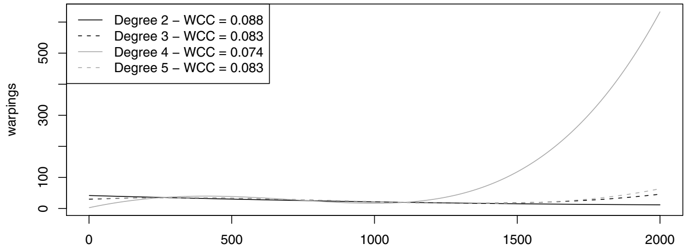

Fig. 3.9. PTW warping functions for different degrees. The fourth degree warping is the most agressive but also achieves the best agreement.

state of the column, all traces should be warped with the same warping function. When using multiple traces, one should have less trouble identifying the optimal warping since ambiguities, that may exist in single chromatograms, are resolved easily [9]. The `ptw` package also has provisions for such so-called global alignment – we will come back to this later. One can also select traces on the basis of certain criteria, such as maximal intensity. A popular criterion is Windig's Component Detection Algorithm (CODA, [22]), which is one of the selection criteriain function `select.traces` in the **ptw** package. CODA basically selects high-variance traces after smoothing, and a unit-length scaling step – the variance of a trace with one high-intensity feature will always be larger than that of a trace with the same length but many low-intensity features.

The following example chooses the ten traces which show the highest values for the CODA criterion, and uses these for constructing a warping function:

```
> sref <- lcms[,,2]
> ssamp <- lcms[,,3]
> traces <- select.traces(sref, criterion = "var")
> lcms.warpglobal <-
+   ptw(sref, ssamp, warp.type = "global",
+       selected.traces = traces$trace.nrs[1:10])
> summary(lcms.warpglobal)
```

```
PTW object: global alignment of 10 samples on 10 references.

Warping coefficients:
         [,1]      [,2]         [,3]
[1,] 91.37956 0.8727316 5.969587e-05

Warping criterion: WCC
Value: 0.1309732
```

Note that the ten selected traces are seen as separate samples for which one global warping function should be found. This is a situation that will occur often in practice – using a couple of information-rich mass traces the quality of the alignment is probably much higher than when using all traces simultaneously. If necessary, small, individual alignments can be made to correct for any remaining shifts. Let us compare an eight-degree individual warping with a global warping of degree two, followed by a four-degree individual warping. The latter strategy also generates a net warping of degree eight [9]:

```
> sample2.indiv.warp <-
+   ptw(lcms[,,3], lcms[,,2],
+       init.coef = c(0, 1, 0, 0, 0, 0, 0, 0, 0))
> sample2.global.warp <-
+   ptw(lcms[,,3], lcms[,,2], init.coef = c(0, 1, 0),
+       warp.type = "multiple")
> sample2.final.warp <-
+   ptw(lcms[,,3], lcms[,,2],
+       init.coef = c(sample2.global.warp$warp.coef, 0, 0))
```

The individual warping is initialized using estimates for the lower degree coefficients found in the global warping. We evaluate the results by looking at the total ion currents, given by the column sums of the warped samples:

```
> plot(time, colSums(lcms[,,3]), col = "gray", lwd = 3,
+      type = "l", main = "PTW (indiv.)", ylab = "I")
> lines(time, colSums(sample2.indiv.warp$warped.sample))
> legend("topleft", legend = c("Sample", "Reference"),
+        lty = 1, col = c("black", "gray"), lwd = c(1,3))
```

This gives the top panel in Figure 3.10, showing the result of the individual eight-degree warping. The bottom plot is produced with similar code, using `sample2.final.warp` instead of `sample2.indiv.warp`. Clearly, already the individual alignments lead to an overall result that is not bad at all, with very good agreement in the middle of the signal. At longer times, however, the compression is much too strong and the dominant feature just after 5000 seconds is placed too early in the warped signal. The combined strategy fares much better with a more or less perfect warping. Note the absence of the aligned signal at the extremes in both cases, the result of an overall compression.

3.3.2 Dynamic Time Warping

Also in Dynamic Time Warping (DTW), implemented in package **dtw** [23], the first step is to construct a warping function. This provides a mapping from the indices in the query signal to the points in the reference signal[2]:

[2] Note that the order of sample and reference signals is reversed compared to the `ptw` function.

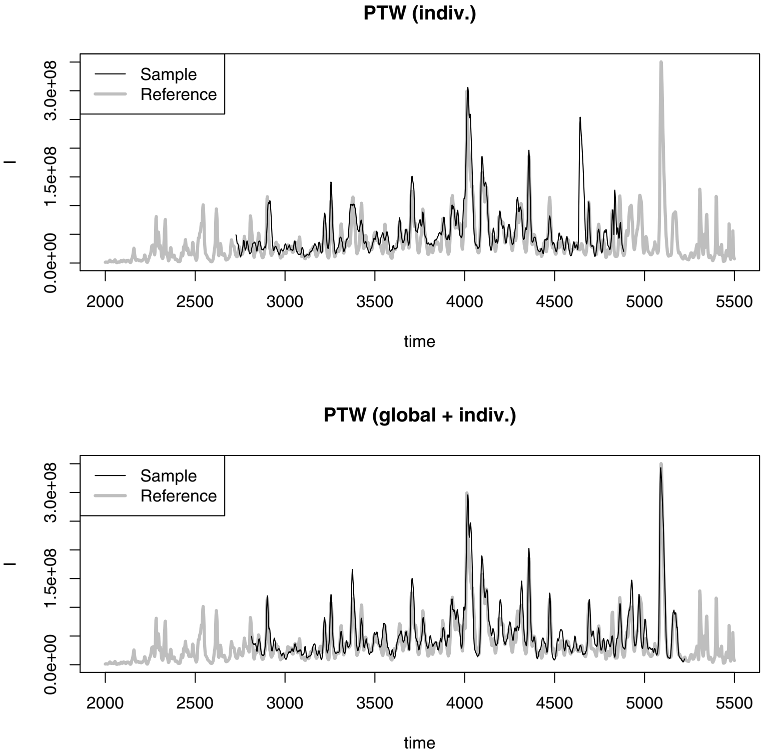

Fig. 3.10. Comparison of individual and global parametric time warping for samples two and three of the `lcms` data: the total ion current (TIC) of the aligned samples is shown. In the top panel, all mass traces have been warped individually using a warping function of degree eight – the bottom panel shows a two-stage warping using a global warping of degree two, followed by an individual warping of degree four.

```
> warpfun <- dtw(ssamp, sref)
> plot(warpfun)
> abline(-20, 1, col = "gray", lty = 2)
```

The result is shown in the left plot of Figure 3.11. A horizontal segment indicates that several points in the query signal are mapped to the same point in the reference signal; the axis of the query signal is compressed by elimination (or rather, averaging) of points. Similarly, vertical segments indicate a stretching of the query signal axis by the duplication of points. Note that these horizontal and vertical regions in the warping function of Figure 3.11 may also lead to peak shape distortions.

DTW chooses the warping function that minimizes the (weighted) distance between the warped signals:

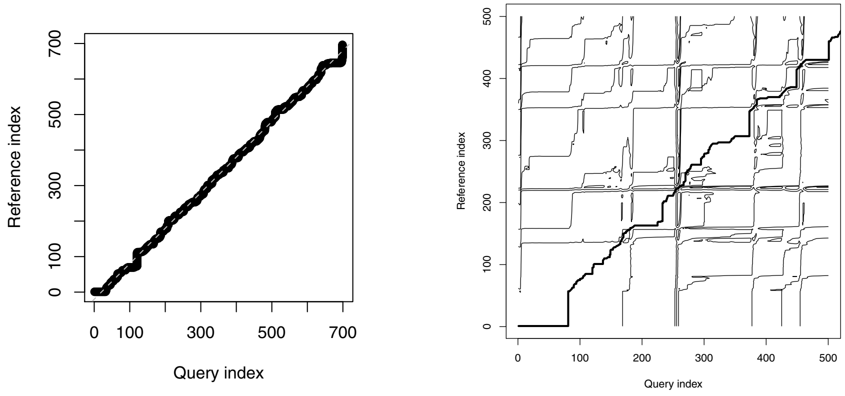

Fig. 3.11. Left plot: warping function of the data from Figure 3.7. The identity warp is indicated with a dashed gray line. Right plot: lower left corner of the warping function plot, showing the contour lines of the cost function, as well as the final warping function (fat line).

$$\sum_k \{q(m(k)) - r(n(k))\}^2 w(k) / \sum_k w(k)$$

where k is the common axis to which both the query and the reference are mapped, $m(k)$ is the warped query signal and $n(k)$ is the warped reference. Note that in this symmetric formulation there is no difference in treatment of query and reference signals: reversing the roles would lead to the same mapping. The weights are used to remove the tendency to select the shortest warping path, but should be chosen with care. The weighting scheme in the original publication [15] is for point $k+1$:

$$w(k+1) = m(k+1) - m(k) + n(k+1) - n(k)$$

That is, if both indices advance to the next point, the weight is 2; if only one of the indices advances to the next point, the weight is 1. A part of the cumulative distance from the start of both signals is shown in the right plot of Figure 3.11: the warping function finds the minimum through the (often very noisy) surface.

Obviously, such a procedure is very flexible, and indeed, one can define warping functions that put any two signals on top of each other, no matter how different they are. This is of course not what is desired, and usually several constraints are employed to keep the warping function from extreme distortions. One can, e.g., limit the maximal warping, or limit the size of individual warping steps. The `dtw` package implements these constraints and also provides the possibility to align signals of different length.

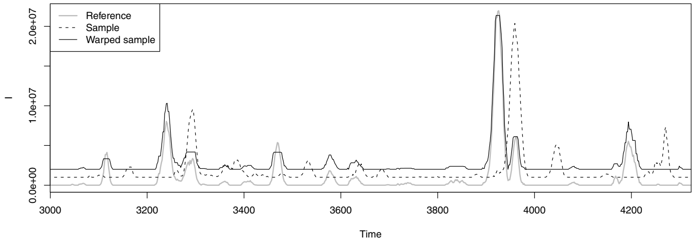

Fig. 3.12. DTW-corrected mass spectrometry data.

Once the warping function is calculated, we can use it to actually map the points in the second signal to positions corresponding to the first. For this, the `warp` function should be used, which internally performs a linear interpolation of the common axis to the original axes:

```
> wx2 <- warp(warpfun)
> plot(mz, x[,1], type = "l", xlab = "m/z", ylab = "I")
> points(mz, x[wx2,2], col = "gray")
```

The warped signal can directly be compared to the reference. The result is shown in Figure 3.12. Immediately one can see that the warping is perfect: peaks are in exactly the same positions. The only differences between the two signals are now found in areas where the peaks in the reference signal are higher than in the warped signal (e.g. m/z values at 3,100 and just below 3,300) – these peak distortions can not be corrected for.

These data are ideally suited for DTW: individual mass traces contain not too many, nicely separated peaks, so that it is clear what features should be aligned. The quality of the warping becomes clear when we align all traces individually and then compare the TIC of the warped sample with the TIC of the reference:

```
> sample2.dtw <- matrix(0, 100, 2000)
> for (i in 1:100) {
+   warpfun.dtw <- dtw(lcms[i,,3], lcms[i,,2])
+   new.indices <- warp(warpfun.dtw, index.reference = FALSE)
+   sample2.dtw[i,] <- lcms[i,new.indices,3]
+ }
```

The result is shown in the top plot of Figure 3.13 – this should be compared with the top plot in Figure 3.10. Clearly, the DTW result is much better. Note that since there is no overall compression, the length of the warped sample equals the original length. Global alignment, using one warping function for all traces simultaneously, is available using the following code:

```
> warp.dtw.gl <- dtw(t(lcms[,,3]), t(lcms[,,2]))
> samp.aligned <- lcms[,warp(warp.dtw.gl),3]
```

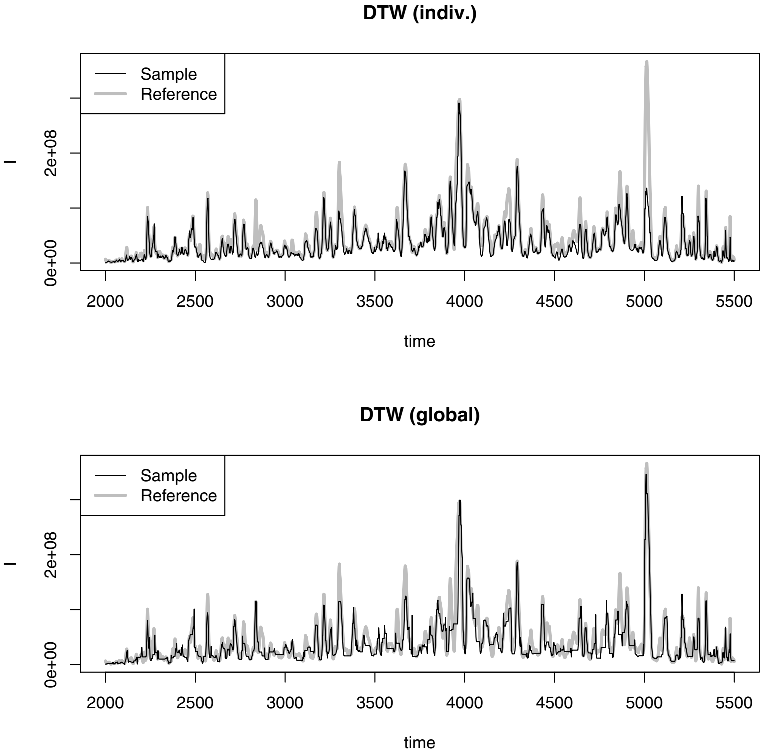

Fig. 3.13. TIC profiles of samples two and three of the `lcms` data. Top plot: DTW alignment using individual traces. Bottom plot: global DTW alignment.

The result, although still very good, is less convincing than the sum of individual DTW alignments: although the features are still aligned correctly, there are many examples of peak deformations. Global alignment with DTW is more constrained and therefore is forced to make compromises. One should be careful, however, when applying extremely flexible warping methods such as DTW to data sets in which not all peaks can be matched: in this particular example we have aligned replicate measurements. In practice, one very often will want to compare samples from different classes, and probably will want to identify those peaks that are discriminating between the classes. Alignment methods that are too flexible may be led astray by the presence of extra peaks, especially when these are of high intensity. More constrained versions of DTW, or PTW, then would probably be more appropriate.

3.3.3 Practicalities

In almost all cases, a set of signals should be aligned in such a way that all features of interest are at the same positions in every trace. One strategy is to use the column means of the data matrix as a reference. This is only possible with very small shifts and will lead to peak broadening. Simply taking a random sample from the set as a reference is better but still may be improved upon – it usually pays to perform some experiments to see which reference would lead to the smallest distortion of the other signals, while still leading to good alignment. If the number of samples is not too large, one can perform all possible combinations and see which one comes out best.

Careful data pretreatment is essential – baselines may severely influence the results and should be removed before alignment. In fact, one of the motivations of the CODA algorithm is to select traces that do not contain a baseline [22]. Another point of attention is the fact that features can have intensities differing several orders in magnitude. Often, the biggest gain in the alignment optimization is achieved by getting the prominent features in the right location. Sometimes, this dominance leads to suboptimal alignments. Also differences in intensity between sample and reference signals can distort the results. Methods to cope with these phenomena will be treated in Section 3.5. Finally, it has been shown that in some cases results can be improved when the signals are divided into segments which are aligned individually [17]. Especially with more constrained warping methods like PTW this adds flexibility, but again, there is a danger of warping too much and mapping features onto the wrong locations. Especially in cases where there may be differences between the samples (control versus diseased, for instance) there is a risk that a biomarker peak, present only in one of the two classes, is incorrectly aligned. This, again, is all the more probable when that particular peak has a high intensity.

Packages **dtw** and **ptw** are by no means alone in tackling alignment. We already mentioned the **VPdtw** package: in addition, several Bioconductor packages, such as **PROcess** and **xcms**, implement both general and more specific alignment procedures, in most cases for mass spectrometry data or hyphenated techniques like LC-MS.

3.4 Peak Picking

Several of the problems associated with misalignment can be avoided if the spectra can be transformed into lists of features, a process that is also known as peak picking. The first question of course is: what is a peak, exactly? This depends on the spectroscopic technique – usually it is a local maximum in a more or less smooth curve. In NMR, for instance, peaks usually have a specific shape (a Lorentz line shape). This knowledge can be used to fit the peaks to the data, and also to give quality assessments of the features that are identified. In chromatography, peaks can be described by a modified normal distribution, where the modification is allowing for peak tailing and other experimental imperfections. In cases where we do not want to make assumptions about peak shape, we are forced to more crude methods, e.g., finding a list of local maxima. One simple way to do this is again to make use of the embed function that splits up the spectrum in many overlapping segments. For each segment, we can calculate the location of the local maximum, and eliminate those segments where the local maximum is at the beginning or at the end. A function implementing this strategy is given in the next piece of code:

```
> pick.peaks <- function(x, span) {
+   span.width <- span * 2 + 1
+   loc.max <- span.width + 1 -
+     apply(embed(x, span.width), 1, which.max)
+   loc.max[loc.max == 1 | loc.max == span.width] <- NA
+
+   pks <- loc.max + 0:(length(loc.max)-1)
+   unique(pks[!is.na(pks)])
+ }
```

The span parameter determines the width of the segments: wider segments will cause fewer peaks to be found. Let us investigate the effect using the prostate data from Figure 3.2:

```
> pks10 <- pick.peaks(rmeans, 10)
> plot(prostate.mz[3:498], rmeans, type = "l",
+      xlab = "m/z", ylab = "Response")
> abline(v = prostate.mz[pks10 + 2], col = "gray")
> pks40 <- pick.peaks(rmeans, 40)
> plot(prostate.mz[3:498], rmeans, type = "l",
+      xlab = "m/z", ylab = "Response", main = "span = 40")
> abline(v = prostate.mz[pks40 + 2], col = "gray")
```

This leads to the plots in Figure 3.14; with the wider span, many of the smaller features are not detected. At the same time, the many noisy features that are found with the smaller span, e.g., around m/z value 2040, probably do not consitute valid features. Clearly, the results of peak picking depend crucially on the degree and quality of the smoothing – method-specific peak picking methods will lead to superior results.

Once the positions of the features have been identified, one should quantify the signals. If an explicit peak model has been fitted, the normal approach would be to use the peak area, obtained by integrating between certain limits; if not, very often the peak height is taken. Under the assumption that peak widths are relatively constant, the two measures lead to similar results. If possible, one should then identify the signals: in, e.g., mass spectrometry this would mean the identification of the corresponding fragment ion. Having these

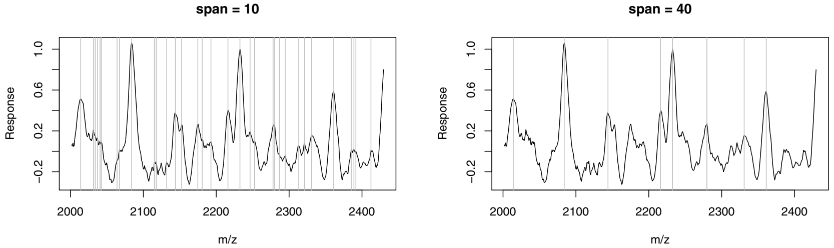

Fig. 3.14. Peak picking by identifying local maxima (prostate data): in the left figure the span is ten points, in the right forty.

assignments makes it much easier to compare spectra of different samples: even if the features are not exactly at the same position, it still is clear which signals to compare. Thus, the need for peak alignment is obviated. In practice, however, it is rare to have complete assignments of spectral data from complex samples, and an alignment step remains necessary.

3.5 Scaling

The scaling method that is employed can totally change the result of an analysis. One should therefore carefully consider what scaling method (if any) is appropriate. Scaling can serve several purposes. Many analytical methods provide data that are not on an absolute scale; the raw data in such a case are cannot be used directly when comparing different samples. If some kind of internal standard is present, it can be used to calibrate the intensities. In NMR, for instance, the TMS (tetramethylsilane, added to indicate the position of the origin on the x-axis) peak can be used for this if its concentration is known. Peak heights can then be compared directly. However, even in that situation it may be necessary to further scale intensities, since samples may contain different concentrations. A good example is the analysis of a set of urine samples by NMR. These samples will show appreciable global differences in concentrations, perhaps due to the amount of liquid the individuals have been consuming. This usually is not of interest – rather, one is interested in finding one or perhaps a couple of metabolites with concentrations that deviate from the general pattern. As an example, consider the first ten spectra of the `prostate` data:

```
> range(apply(prostate[1:10,], 1, max))

[1] 16.35960 68.89841

> range(rowSums(prostate[1:10,]))

[1]  2531.694 15207.851
```

The intensity differences within these first ten spectra are already a factor five for both statistics. If these differences are not related to the phenomenon we are interested in but are caused, e.g., by the nature of the measurements, then it is important to remove them. As stated earlier, also in cases where alignment is necessary, this type of differences between samples can hamper the analysis.

Several options exist to make peak intensities comparable over a series of spectra. The most often-used are *range scaling*, *length scaling* and *variance scaling*. In range scaling, one makes sure that the data have the same minimal and maximal values. Often, only the maximal value is considered important since for many forms of spectroscopy zero is the natural lower bound. Length scaling sets the length of each spectrum to one; variance scaling sets the variance to one. The implementation in R is easy. Here, these three methods are shown for the first ten spectra of the prostate data. Range scaling can be performed by

```
> prost10.rangesc <- sweep(prostate[1:10,], MARGIN = 1,
+                          apply(prostate[1:10,], 1, max),
+                          FUN = "/")
> apply(prost10.rangesc, 1, max)
```

```
 [1] 1 1 1 1 1 1 1 1 1 1
```

```
> range(rowSums(prost10.rangesc))
```

```
[1] 103.3278 220.7286
```

The sweep function is very similar to apply – it performs an action for every row or column of a data matrix. The MARGIN argument states which dimension is affected. In this case the MARGIN = 1 indicates the rows; column-wise sweeping would be achieved with MARGIN = 2. The third argument is the statistic that is to be swept out, here the vector of the per-row maximal values. The final argument states how the sweeping is to be done. The default is to use subtraction; here we use division. Clearly, the differences between the spectra have decreased.

Length scaling is done by dividing each row by the square root of the sum of its squared elements:

```
> prost10.lengthsc <- sweep(prostate[1:10,], MARGIN = 1,
+                           apply(prostate[1:10,], 1,
+                                 function(x) sqrt(sum(x^2))),
+                           FUN = "/")
> range(apply(prost10.lengthsc, 1, max))
```

```
[1] 0.1107480 0.2058059
```

```
> range(rowSums(prost10.lengthsc))
```

```
[1] 18.93725 30.23589
```

The difference between the smallest and largest values is now less than a factor of two. Variance scaling has a similar effect:

```
> prost10.varsc <- sweep(prostate[1:10,], MARGIN = 1,
+                        sd(t(prostate[1:10,])),
+                        FUN = "/")
> range(apply(prost10.varsc, 1, max))
```

```
[1] 11.69716 21.65927
```

```
> range(rowSums(prost10.varsc))
```

```
[1] 1976.494 3245.692
```

In this case we use the function `sd` which returns the standard deviations of the columns of a matrix – hence the transpose.

The underlying hypothesis in these scaling methods is that the maximal intensities, or the vector lengths, or the variances, should be equal in all objects. However, there is no real way of assessing whether these assumptions are correct, and it is therefore always advisable to assess different options.

Often in statistical modelling, especially in a regression context, we are more interested in deviations from a mean value than in the values per se. These deviations can be obtained by *mean-centering*, where one subtracts the mean value from every column in the data matrix, for example with the gasoline data:

```
> NIR.mc <- t(sweep(gasoline$NIR, 2, colMeans(gasoline$NIR)))
```

Subtraction is the default operation in `sweep`, but one can also use `sweep` to perform other functions using the `FUN` argument. An even easier way to achieve the same effect is to use the `scale` function:

```
> NIR.mc <- scale(gasoline$NIR, scale = FALSE)
> matplot(wavelengths, t(NIR.mc),
+         type = "l", xlab = "Wavelength (nm)",
+         ylab = "1/R (mean-centered)", lty = 1, col = 1)
```

The result is shown in Figure 3.15; note the differences with the raw data shown in Figure 2.1 and the first derivatives in Figure 3.5.

When variables have been measured in different units or have widely different scales, we should take this into account. Obviously, one does not want the scale in which a variable is measured to have a large influence on the model: just switching to other units would lead to different results. One popular way of removing this dependence on units is called *autoscaling*, where every column $\boldsymbol{x}_i$ is replaced by

$$(\boldsymbol{x}_i - \hat{\mu}_i)/\hat{\sigma}_i$$

In statistics, this is often termed *standardization* of data; the effect is that all variables are considered equally important. This type of scaling is appropriate

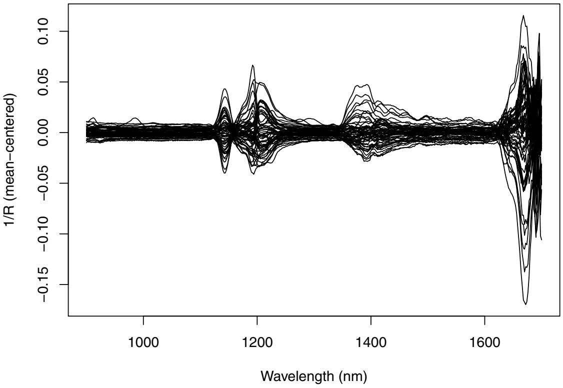

Fig. 3.15. Mean-centered gasoline NIR data.

for the wine data, since the variables have different units and very different ranges:

```
> apply(wines, 2, range)

     alcohol malic acid  ash ash alkalinity magnesium
[1,]   11.03       0.74 1.36           10.6        70
[2,]   14.83       5.80 3.23           30.0       162
     tot. phenols flavonoids non-flav. phenols proanth
[1,]         0.98       0.34              0.13    0.41
[2,]         3.88       5.08              0.66    3.58
     col. int. col. hue OD ratio proline
[1,]      1.28     0.48     1.27     278
[2,]     13.00     1.71     4.00    1680
```

The apply function in this case returns the range of every column. Clearly, the last variable (proline) has values that are quite a lot bigger than the other variables. The scale function, already mentioned, also does autoscaling: simply use the argument scale = TRUE, or do not mention it at all (it is the default):

```
> wines.sc <- scale(wines)
> boxplot(wines.mc ~ col(wines.mc),
+         main = "Mean-centered wine data")
> boxplot(wines.sc ~ col(wines.sc),
+         main = "Autoscaled wine data")
```

The result is shown in Figure 3.16. In the left plot, showing the mean-centered data, the dominance of proline is clearly visible, and any structure that may

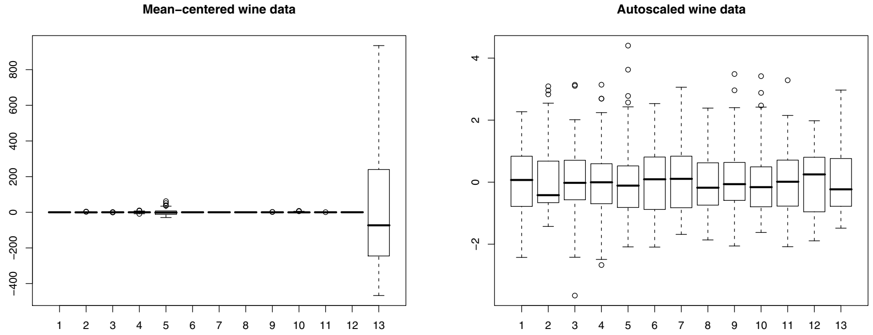

Fig. 3.16. Boxplots for the thirteen mean-centered variables (left) and auto-scaled variables (right) in the wine data set.

be present in the other variables is hard to detect. The right plot is much more informative. In almost all cases where variables indicate concentrations or amounts of chemical compounds, or represent measurements in unrelated units, autoscaling is a good idea. For the wine data, we will use always autoscaled data in examples, even in cases where scaling does not matter.

For spectral data, prevalent in the natural sciences and the life sciences, on the other hand, autoscaling is usually not recommended. Very often, the data consist of areas of high information content, *viz.* containing peaks of different intensities, and areas containing only noise. When every spectral variable is set to the same standard deviation, the noise is blown up to the same size as the signal that contains the actual information. Clearly, this is not a desirable situation, and in such cases mean-centering is much preferred.

Specialized preprocessing methods are used a lot in spectroscopy. When the total intensity in the spectra is sample-dependent, spectra should be scaled in such a way that intensities can be compared. A typical example is given by the analysis of urine spectra (of whatever type): depending on how much a person has been drinking, urine samples can be more or less diluted, and therefore there is no direct relation between peak intensities in spectra from different subjects. Apart from the range scaling method that we saw earlier, one other form of scaling is often used, especially in NIR applications: Standard Normal Variate scaling (SNV). This method essentially does autoscaling on the rows instead of the columns. That is, every spectrum will after scaling have a mean of zero and a standard deviation of 1. This gets rid of arbitrary offsets and multiplication factors. Obviously, the assumption that all spectra should have the same mean and variance is not always realistic! In some cases, the fact that the overall intensity in one spectrum is consistently higher may contain important information. In most cases, SNV gives results that are very close to MSC.

When noise is multiplicative in nature rather than additive, the level of variation depends on the signal strength. Since most noise reduction methods assume additive noise, a simple solution is to perform a log-transformation of the data. This decreases the influence of the larger features, and makes the noise more constant over the whole range. Next, regular noise reduction methods can be applied. Log transformation can also be used to reduce the dominance of the largest features, which can be disturbing the analysis, e.g., in alignment applications. A similar effect can be obtained by using *Pareto scaling*, which is the same as autoscaling with the exception that the square root of the standard deviation is used in rescaling, rather than the standard deviation itself. This form of scaling has become popular in biomarker identification applications, e.g., in metabolomics: one would like to find variables that behave differently in two populations, and one is mostly interested in those variables that have high intensities.

3.6 Missing Data

Missing data are measurements that for some reason have not led to a valid result. In spectroscopic measurements, missing data are not usually encountered, but in many other areas of science they occur frequently. The main question to be answered is: are the data missing at random? If yes, then we can probably get around the problem, provided there are not too many missing data. If no, then it means that there is some reason behind the distribution of NAs in the data set – and that means trouble. We may never be able to tell whether the missingness of some data points is not related to the process that we are studying. Therefore, in many practical applications the missing-at-random hypothesis is taken for granted.

Then, there are several strategies: one may use only those samples for which there are no missing values; one may leave out all variables containing missing values, or one can try to estimate the missing values from the other data points, a process that is known as *imputation*. Given the fact that missing values are not playing a prominent part in the analysis of spectral data in the life sciences, the main focus of this book, we will leave it at the observation that R provides ample mechanisms for dealing with missing values, which are usually represented by `NA`. Many functions have built-in capabilities of handling them. The simplest is just to remove the missing values (`na.rm = TRUE`, e.g. one of the arguments of functions like `mean`). Matrix functions like `cov` and `cor` have the `use` argument, that for instance can choose to consider only complete observations. Consult the manual pages for more information and examples.

3.7 Conclusion

Data preprocessing is an art, in most cases requiring substantial background knowledge. Because several steps are taken sequentially, the number of possible schemes is often huge. Should one scale first and then remove noise or the other way around? Individual steps will influence each other: noise removal may make it more easy to correct for a sloping baseline, but the presence of a baseline may also influence your estimate of what is noise. General recipes are hard to give, but some problems are more serious than others. The presence of peak shifts, for instance, will make any multivariate analysis very hard to interpret.

Finally, one should realise that bad data preprocessing can never be compensated for in the subsequent analysis. One should *always* inspect the data before and after preprocessing and assess whether the relevant information has been kept while disturbing signals have been removed. Of course, that is easier said than done – and probably one will go through a series of modelling cycles before one is completely satisfied with the result.

Part II

Exploratory Analysis

4

Principal Component Analysis

Principal Component Analysis (PCA, [24, 25]) is a technique which, quite literally, takes a different viewpoint of multivariate data. In fact, PCA defines new variables, consisting of linear combinations of the original ones, in such a way that the first axis is in the direction containing most variation. Every subsequent new variable is orthogonal to previous variables, but again in the direction containing most of the remaining variation. The new variables are examples of what often is called *latent variables* (LVs), and in the context of PCA they are also termed *principal components* (PCs).

The central idea is that more often than not high-dimensional data are not of full rank, implying that many of the variables are superfluous. If we look at high-resolution spectra, for example, it is immediately obvious that neighbouring wavelengths are highly correlated and contain similar information. Of course, one can try to pick only those wavelengths that appear to be informative, or at least differ from the other wavelengths in the selected set. This could, e.g., be based on clustering the variables, and selecting for each cluster one “representative”. However, this approach is quite elaborate and will lead to different results when using different clustering methods and cutting criteria. Another approach is to use variable selection, given some criterion – one example is to select the limited set of variables leading to a matrix with maximal rank. Variable selection is notoriously diffucult, especially in high-dimensional cases. In practice, many more or less equivalent solutions exist, which makes the interpretation quite difficult. We will come back to variable selection methods in Chapter 10.

PCA is an alternative. It provides a direct mapping of high-dimensional data into a lower-dimensional space containing most of the information in the original data. The coordinates of the samples in the new space are called *scores*, often indicated with the symbol $\boldsymbol{T}$. The new dimensions are linear combinations of the original variables, and are called *loadings* (symbol $\boldsymbol{P}$). The term Principal Component (PC) can refer to both scores and loadings; which is meant is usually clear from the context. Thus, one can speak of

R. Wehrens, *Chemometrics with R: Multivariate Data Analysis in the Natural Sciences and Life Sciences*, Use R!, DOI 10.1007/978-3-642-17841-2_4,

sample coordinates in the space spanned by PC 1 and 2, but also of variables contributing greatly to PC 1.

The matrix multiplication of scores and loadings leads to an approximation $\tilde{\boldsymbol{X}}$ of the original data $\boldsymbol{X}$:

$$\tilde{\boldsymbol{X}} = \boldsymbol{T}_a \boldsymbol{P}_a^T \tag{4.1}$$

Superscript ^{T}, as usual, indicates the transpose of a matrix. The subscript a indicates how many components are taken into account: the largest possible number of PCs is the minimum of the number of rows and columns of the matrix:

$$a_{\max} = \min(n, p) \tag{4.2}$$

If $a = a_{\max}$, the approximation is perfect and $\tilde{\boldsymbol{X}} = \boldsymbol{X}$.

The PCs are orthogonal combinations of variables defined in such a way that [25]:

- the variances of the scores are maximal;
- the sum of the Euclidean distances between the scores is maximal;
- the reconstruction of $\boldsymbol{X}$ is as close as possible to the original: $||\boldsymbol{X} - \tilde{\boldsymbol{X}}||$ is minimal.

These three criteria are equivalent [24]; the next section will show how to find the PCs.

PCA has many advantages: it is simple, has a unique analytical solution optimizing a clear and unambiguous criterion, and often leads to a more easily interpretable data representation. The price we have to pay is that we do not have a small set of wavelengths carrying the information but a small set of principal components, in which *all* wavelengths are represented. Note that the underlying assumption is that variation equals information. Intuitively, this makes sense: one can not learn much from a constant number.

Once PCA has defined the latent variables, one can plot all samples in the data set while ignoring all higher-order PCs. Usually, only a few PCs are needed to capture a large fraction of the variance in the data set (although this is highly dependent on the type of data). That means that a plot of (the scores of) PC 1 versus PC 2 can already be highly informative. Equally useful is a plot of the contributions of the (original) variables to the important PCs. These visualizations of high-dimensional data perhaps form the most important application of PCA. Later, we will see that the scores can also be used in regression and classification problems.

4.1 The Machinery

Currently, PCA is implemented even in low-level numerical software such as spreadsheets. Nevertheless, it is good to know the basics behind the computations. In almost all cases, the algorithm used to calculate the PCs is *Singular*

Value Decomposition (SVD)[1]. It decomposes an $n \times p$ mean-centered data matrix $\boldsymbol{X}$ into three parts:

$$\boldsymbol{X} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{V}^T \tag{4.3}$$

where $\boldsymbol{U}$ is a $n \times a$ orthonormal matrix containing the left singular vectors, $\boldsymbol{D}$ is a diagonal matrix ($a \times a$) containing the singular values, and $\boldsymbol{V}$ is a $p \times a$ orthonormal matrix containing the right singular vectors. The latter are what in PCA terminology is called the loadings – the product of the first two matrices forms the scores:

$$\boldsymbol{X} = (\boldsymbol{U}\boldsymbol{D})\boldsymbol{V}^T = \boldsymbol{T}\boldsymbol{P}^T \tag{4.4}$$

The interpretation of matrices $\boldsymbol{T}$, $\boldsymbol{P}$, $\boldsymbol{U}$, $\boldsymbol{D}$ and $\boldsymbol{V}$ is straightforward. The loadings, columns in matrix $\boldsymbol{P}$ (or equivalently, the right singular vectors, columns in matrix $\boldsymbol{V}$) give the weights of the original variables in the PCs. Variables that have very low values in a specific column of $\boldsymbol{V}$ contribute only very little to that particular latent variable. The scores, columns in $\boldsymbol{T}$, constitute the coordinates in the space of the latent variables. Put differently: these are the coordinates of the samples as we see them from our new PCA viewpoint. The columns in $\boldsymbol{U}$ give the same coordinates in a normalized form – they have unit variances, whereas the columns in $\boldsymbol{T}$ have variances corresponding to the variances of each particular PC. These variances λ_i are proportional to the squares of the diagonal elements in matrix $\boldsymbol{D}$:

$$\lambda_i = d_i^2/(n-1)$$

The fraction of variance explained by PC i can therefore be expressed as

$$FV(i) = \lambda_i / \sum_{j=1}^{a} \lambda_j \tag{4.5}$$

One main problem in the application of PCA is the decision on how many PCs to retain; we will come back to this in Section 4.3.

One final remark needs to be made about the unique solution given by the SVD algorithm: it is only unique up to the sign. As is clear from, e.g., Equation 4.4, one can obtain exactly the same solution by reversing the sign of *both* scores and loadings simultaneously. There are no conventions, so one should always keep in mind that this possibility exists. For the interpretation of the data, it make no difference whatsoever.

Although SVD is a fast algorithm in some cases it can be efficient not to apply it to the data matrix directly, especially in cases where there is a large difference in the numbers of rows and columns. In such a case, it is faster

[1] One alternative for SVD is the application of the Eigen decomposition on the covariance or correlation matrix of the data. SVD is numerically more stable and is therefore preferred in most cases.

to apply SVD to either $\boldsymbol{X}^T\boldsymbol{X}$ or $\boldsymbol{X}\boldsymbol{X}^T$, whichever is the smaller one. If the number of columns is much smaller than the number of rows, one would obtain

$$\boldsymbol{X}^T\boldsymbol{X} = (\boldsymbol{U}\boldsymbol{D}\boldsymbol{V}^T)^T\boldsymbol{U}\boldsymbol{D}\boldsymbol{V}^T = \boldsymbol{V}\boldsymbol{D}^2\boldsymbol{V}^T$$

Applying `svd`[2] directly yields loadings and sums of squares. Matrix $\boldsymbol{T}$, the score matrix, is easily found by right-multiplying both sides of Equation 4.4 with $\boldsymbol{P}$:

$$\boldsymbol{X}\boldsymbol{P} = \boldsymbol{T}\boldsymbol{P}^T\boldsymbol{P} = \boldsymbol{T} \tag{4.6}$$

because of the orthonormality of $\boldsymbol{P}$. Similarly, we can find the left singular vectors and singular values when applying SVD to $\boldsymbol{X}\boldsymbol{X}^T$ – see the example in the next section.

4.2 Doing It Yourself

Calculating scores and loadings is easy: consider the wine data first. We perform PCA on the autoscaled data to remove the effects of the different scales of the variables using the `svd` function provided by R

```
> wines.svd <- svd(wines.sc)
> wines.scores <- wines.svd$u %*% diag(wines.svd$d)
> wines.loadings <- wines.svd$v
```

The first two PCs represent the plane that contains most of the variance; how much exactly is given by the squares of the values on the diagonal of $\boldsymbol{D}$. The importance of individual PCs is usually given by the percentage of the overall variance that is explained:

```
> wines.vars <- wines.svd$d^2 / (nrow(wines) - 1)
> wines.totalvar <- sum(wines.vars)
> wines.relvars <- wines.vars / wines.totalvar
> variances <- 100 * round(wines.relvars, digits = 3)
> variances[1:5]

[1] 36.0 19.2 11.2  7.1  6.6
```

The first PC covers more than one third of the total variance; for the fifth PC this amount is down to one fifteenth.

The scores show the positions of the individual wine samples in the coordinate system of the PCs. A score plot can be produced as follows:

```
> plot(wines.scores[,1:2], type = "n",
+      xlab = paste("PC 1 (", variances[1], "%)", sep = ""),
+      ylab = paste("PC 2 (", variances[2], "%)", sep = ""))
> abline(h = 0, v = 0, col = "gray")
> points(wines.scores[,1:2], pch = wine.classes)
```

[2] Or `eigen`, which returns eigenvectors and eigenvalues.

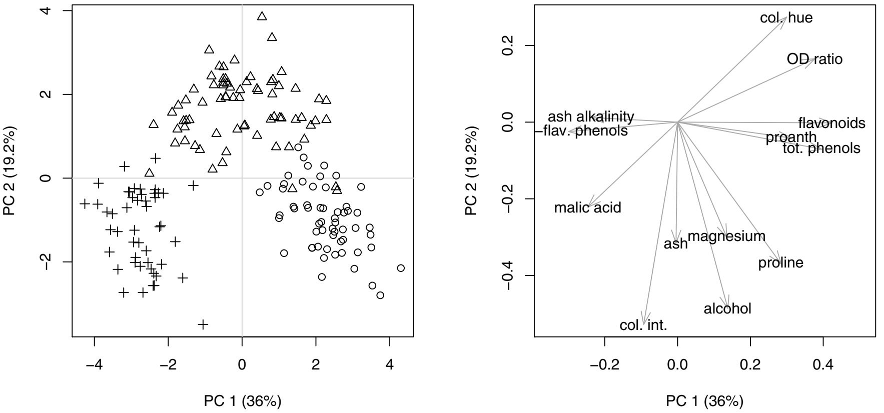

Fig. 4.1. Left plot: scores on PCs 1 and 2 for the autoscaled wine data. Different symbols correspond to the three cultivars. Right plot: loadings on PCs 1 and 2.

The result is depicted in the left plot of Figure 4.1. It is good practice to indicate the amount of variance associated with each PC on the axis labels. The three cultivars can be clearly distinguished: class 1, Barbera, indicated with open circles, has the smallest scores on PC 1 and class 2 (Barolo – plusses in the figure) the largest. PC 2, corresponding to 19% of the variance, improves the separation by separating the Grignolinos, the middle class on PC 1, from the other two. Note that this is a happy coincidence: PCA does not explicitly look to discriminate between classes. In this case, the three cultivars clearly have different characteristics. What characteristics these are can be seen in the loading plot, shown on the right in Figure4.1. It shows the contribution of the original variables to the PCs. Loadings are traditionally shown as arrows from the origin:

```
> plot(wines.loadings[,1] * 1.2, wines.loadings[,2], type = "n",
+      xlab = paste("PC 1 (", variances[1], "%)", sep = ""),
+      ylab = paste("PC 2 (", variances[2], "%)", sep = ""))
> arrows(0, 0, wines.loadings[,1], wines.loadings[,2],
+      col = "darkgray", length = .15, angle = 20)
> text(wines.loadings[,1:2], labels = colnames(wines))
```

The factor of 1.2 in the plot command is used to create space for the text labels. Clearly, the wines of class 3 are distinguished by lower values of alcohol and a lower color intensity. Wines of class 1 have high flavonoid and phenol content and are low in non-flavonoid phenols; the reverse is true for wines of class 2. All of these conclusions could probably have been drawn also by looking at class-specific boxplots for all variables – however, the combination of one score plot and one loading plot shows this in a much simpler way, and even presents direct information on correlations between variables and objects. We will come back on this point later, when treating biplots.

As an example of the kind of speed improvement one can expect when applying SVD on the crossproduct matrices rather than the original data, consider the `prostate` data. Timings can be obtained by wrapping the code within a `system.time` call:

```
> system.time({
+   prost.svd <- svd(prostate)
+   prost.scores <- prost.svd$u %*% diag(prost.svd$d)
+   prost.variances <- prost.svd$d^2 / (nrow(prostate) - 1)
+   prost.loadings <- prost.svd$v
+   })

   user  system elapsed
 30.706   0.264  30.988
```

Here, the number of variables is much larger than the number of objects (which, by the way, is not extremely small either), so we perform SVD on the matrix $\boldsymbol{X}\boldsymbol{X}^T$:

```
> system.time({
+   prost.tcp <- tcrossprod(prostate)
+   prost.svd <- svd(prost.tcp)
+   prost.scores <- prost.svd$u %*% diag(sqrt(prost.svd$d))
+   prost.variances <- prost.svd$d / (nrow(prostate) - 1)
+   prost.loadings <- solve(prost.scores, prostate)
+   })

   user  system elapsed
 16.085   0.132  16.216
```

The second option is twice as fast.

4.3 Choosing the Number of PCs

The question how many PCs to consider, or put differently: where the information stops and the noise begins, is difficult to answer. Many methods consider the amount of variance explained, and use statistical tests or graphical methods to define which PCs to include. In this section we briefly review some of the more popular methods.

The amount of variance per PC is usually depicted in a scree plot: either the variances themselves or the logarithms of the variances are shown as bars. Often, one also considers the fraction of the total variance explained by every single PC. The last few PCs usually contain no information and, especially on a log scale, tend to make the scree plot less interpretable, so they are usually not taken into account in the plot.

```
> par(mfrow = c(2,2))
> barplot(wines.vars[1:10], main = "Variances",
>         names.arg = paste("PC", 1:10))
> barplot(log(wines.variances[1:10]), main = "log(Variances)",
>         names.arg = paste("PC", 1:10))
> barplot(wines.relvars[1:10], main = "Relative variances",
>         names.arg = paste("PC", 1:10))
> barplot(cumsum(100 * wines.relvars[1:10]),
>         main = "Cumulative variances (%)",
>         names.arg = paste("PC", 1:10), ylim = c(0, 100))
```

This leads to the plots in Figure 4.2. Clearly, PCs 1 and 2 explain much more variance than the others: together they cover 55% of the variance. The scree plots show no clear cut-off, which in real life is the rule rather than the exception. Depending on the goal of the investigation, for these data one could consider three or five PCs. Choosing four PCs would not make much sense in this case, since the fifth PC would explain almost the same amount of variance: if the fourth is included, the fifth should be, too.

4.3.1 Statistical Tests

One can show that the explained variance for the i-th component, $\lambda_i = d_i^2/(n-1)$ is asymptotically normally distributed [26, 27], which leads to confidence intervals of the form

$$\ln(\lambda_i) \pm z_\alpha \sqrt{\frac{2}{n-1}}$$

For the wine example, 95% confidence intervals can therefore be obtained as

```
> llambdas <- log(wines.vars)
> CIwidth <- qnorm(.975) * sqrt(2 / (nrow(wines) - 1))
> CIs <- cbind(exp(llambdas - CIwidth),
+              wines.vars,
+              exp(llambdas + CIwidth))
> colnames(CIs) <- c("CI 0.025", "  Estimate", "  CI 0.975")
> CIs[1:5,]
```

```
      CI (0.025)   Estimate   CI (0.975)
PC 1      3.7958      4.678        5.765
PC 2      2.0297      2.501        3.083
PC 3      1.1793      1.453        1.791
PC 4      0.7501      0.924        1.139
PC 5      0.6993      0.862        1.062
```

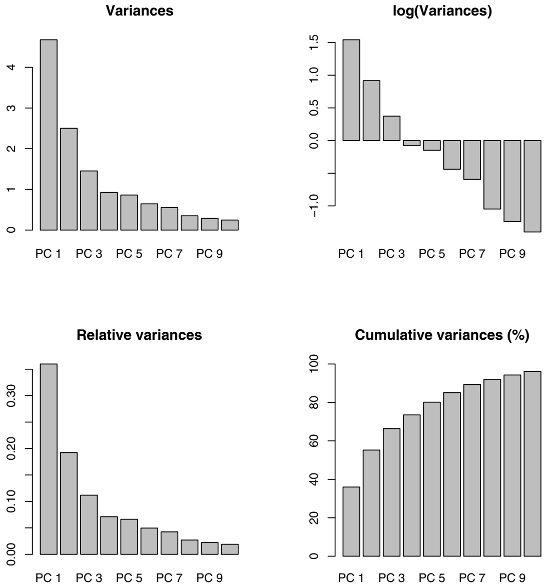

Fig. 4.2. Scree plots for the assessment of the amount of variance explained by each PC. From left to right, top to bottom: variances, logarithms of variances, fractions of the total variance and cumulative percentage of total variance.

Mardia *et al.* present an approach testing the equality of variances for individual PCs [26]. For the autoscaled wine data, one could test whether the last $p-k$ PCs are equally important, i.e. have equal values of λ. The quantity

$$(n-1)(p-k)\log(a_0/g_0)$$

is distributed approximately as a χ^2-statistic with $(p-k+2)(p-k-1)/2$ degrees of freedom [26, p. 236]. In this formula, a_0 and g_0 indicate arithmetic and geometric means of the $p-k$ smallest variances, respectively. We can use this test to assess whether the last three PCs are useful or not:

```
> small.ones <- wines.vars[11:13]
> n <- nrow(wines)
> nsmall <- length(small.ones)
> geo.mean <- prod(small.ones)^{1/nsmall}
> mychisq <- (n - 1) * nsmall * log(mean(small.ones) / geo.mean)
> ndf <- (nsmall + 2) * (nsmall - 1) / 2
> 1 - pchisq(mychisq, ndf)

[1] 9.91e-05
```

This test finds that after PC 10 there is still a difference in the variances. In fact, the test finds a difference after any other cutoff, too: apparently all PCs are significant.

The use of statistical tests for determining the optimal number of PCs has never really caught on. Most scientists are prepared to accept a certain loss of information (variance) provided that the results, the score plots and loading plots, help to answer scientific questions. Most often, one uses informal graphical methods: if an elbow shows up in the scree plot that can be used to decide on the number of PCs. In other applications, notably with spectral data, one can sometimes check the loadings for structure. If there is no more structure – in the form of peak-like shapes – present in the loadings of higher PCs, they can safely be discarded.

4.4 Projections

Once we have our low-dimensional view of the original high-dimensional data, we may be interested how other data are positioned. This may be data from new samples, measured a different instrument, or at a different day. The key point is that the low-dimensional representation allows us to look at the data and in one glance assess whether there are patterns. Obtaining scores for new data is pretty easy. Given a new data matrix $\boldsymbol{X}$, the projections in the space defined by loadings $\boldsymbol{P}$ can be obtained by simple right-multiplication:

$$\boldsymbol{X}\boldsymbol{P} = \boldsymbol{T}\boldsymbol{P}^T\boldsymbol{P} = \boldsymbol{T}$$

The wine data, for example, are more or less ordered according to vintage. If we would perform PCA on the first half of the data, the third class, the Grignolino wines, would not play a part in defining the PCs, and the first class, Barbera, would dominate. To see the effect of this, we can project the second half of the data matrix into the PCA space defined by the first half. We start by constructing scores and loadings:

```
> X1 <- scale(wines[1:88,])
> X1.svd <- svd(X1)
> X1.pca <- list(scores = X1.svd$u %*% diag(X1.svd$d),
+                loadings = X1.svd$v)
```

Then, we scale the second half of the data using the means and standard deviations of the first half. This is a very important detail that sometimes is missed – obviously, both halves should have the same point of origin. Using the scaled second half of the data, we calculated the corresponding scores, and show them in a plot, together with the scores of the first half:

```
> X2 <- scale(wines[89:177,],
+             center = attr(X1, "scaled:center"),
+             scale = attr(X1, "scaled:scale"))
```

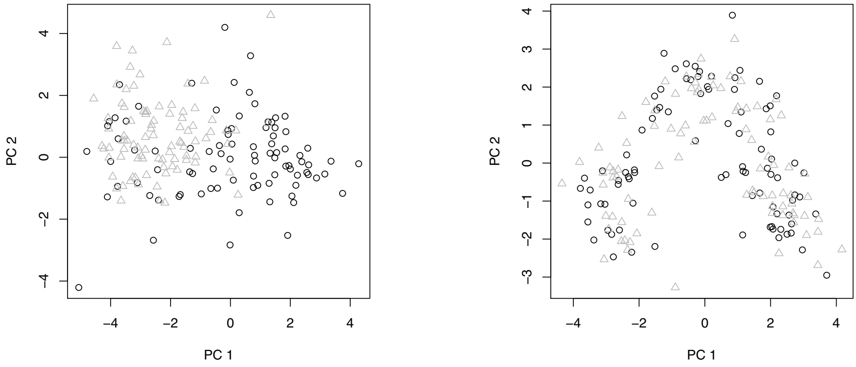

Fig. 4.3. Projections in PCA space. Left plot: second half of the wine data (triangles) projected into the PCA space defined by the first half (circles). Right plot: PCA model based on the odd rows (circles). The even rows (triangles) are projected in this space. The result is very similar to a PCA on the complete data matrix.

```
> X2.scores <- X2 %*% X1.pca$loadings
> plot(rbind(X1.pca$scores, X2.scores),
+      pch = rep(c(1,2), c(88, 89)),
+      col = rep(c(1,2), c(88, 89)),
+      xlab = "PC 1", ylab = "PC 2")
```

This leads to the left plot in Figure 4.3. Clearly, the familiar shape of the PC 1 vs. PC 2 plot has been destroyed, and what is more: the triangles, corresponding to the second half of the data, are generally in a different location than the first half (circles). Since Grignolino wines are only present in the second half, and Barbera wines only in the first half, this comes as no surprise. The right plot in the same figure shows a completely different picture. It has been obtained by defining `X1` and `X2` as follows:

```
> odd <- seq(1, nrow(wines), by = 2)
> even <- seq(2, nrow(wines), by = 2)
> X1 <- scale(wines[odd,])
> X2 <- scale(wines[even,],
+             center = attr(X1, "scaled:center"),
+             scale = attr(X1, "scaled:scale"))
```

Now both halves have similar compositions, and the data clouds neatly overlap. In fact, this is a very important way to check whether a division in training and test set, a topic that we will talk about extensively in later chapters, is a good one.

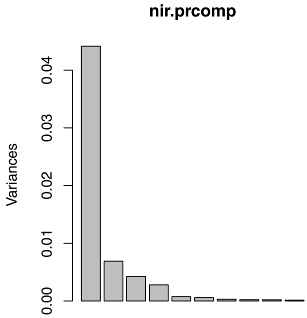

Fig. 4.4. Scree plot for the NIR data. By default, the scree plot shows not more than ten PCs.

4.5 R Functions for PCA

The standard R function for PCA is `prcomp`. By default, the data are mean-centered (but not scaled!). We will show its use on the gasoline data:

```
> nir.prcomp <- prcomp(gasoline$NIR)
> summary(nir.prcomp, digits = 2)
```

```
Importance of components:
                         PC1   PC2    PC3    PC4    PC5     PC6
Standard deviation     0.210 0.083 0.0651 0.0529 0.0275 0.02426
Proportion of Variance 0.726 0.113 0.0695 0.0460 0.0124 0.00967
Cumulative Proportion  0.726 0.839 0.9086 0.9546 0.9670 0.97664
```

The output of the `summary` command is limited here to only six PCs. One can see that they explain almost 98% of the variance. The default `plot` command is to show a scree plot (in fact, the `screeplot` function is used):

```
> plot(nir.prcomp)
```

The result is depicted in Figure 4.4. Probably, most people would select four components to be included: although the first is much larger than the others, components two to four still contribute a substantial amount, whereas higher components are much less important.

Plotting the loadings of the first four PCs shows some interesting structure. The are available as the `rotation` element of the `prcomp` object:

```
> nir.loadings <- nir.prcomp$rotation[,1:4]
```

The original variables are, of course, highly correlated, and therefore connecting subsequent variables in the loading space forms a trajectory. In the following code, producing the plots in Figure 4.5, the variables with the most extreme loadings have been indicated – these can easily be found using the `identify` function (see the manual page for details).

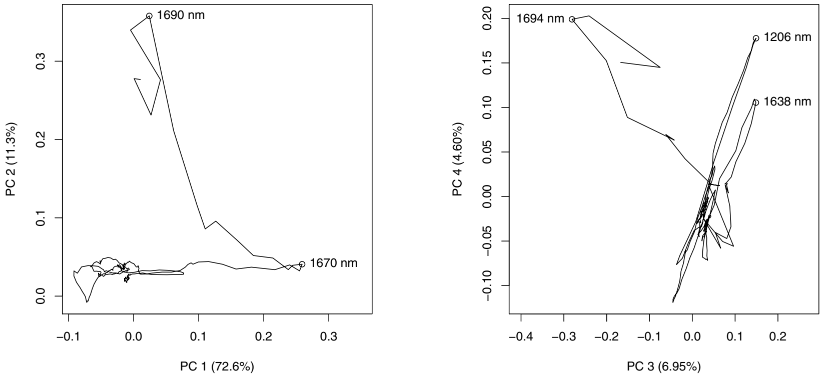

Fig. 4.5. Loading plots for the mean-centered gasoline data; the left plot shows PC 1 versus PC 2, and the right plot PC 3 versus PC 4. Some extreme variable weights are indicated with the corresponding wavelengths.

```
> par(mfrow = c(1,2), pty = "s")
> offset <- c(0, 0.09) # to create space for labels
> plot(nir.loadings[,1:2], type = "l",
+      xlim = range(nir.loadings[,1]) + offset,
+      xlab = "PC 1 (72.6%)", ylab = "PC 2 (11.3%)")
> points(nir.loadings[c(386, 396), 1:2])
> text(nir.loadings[c(386, 396), 1:2], pos = 4,
+      labels = paste(c(1670, 1690), "nm"))
```

The procedure for PCs three and four is completely analogous:

```
> offset <- c(-0.12, 0.12) # to create space for labels
> plot(nir.loadings[,3:4], type = "l",
+      xlim = range(nir.loadings[,3]) + offset,
+      xlab = "PC 3 (6.95%)", ylab = "PC 4 (4.60%)")
> points(nir.loadings[c(154, 370, 398), 3:4])
> text(nir.loadings[c(154, 370), 3:4], pos = 4,
+      labels = paste(c(1206, 1638), "nm"))
> text(nir.loadings[398, 3:4, drop = FALSE],
+      labels = "1694 nm", pos = 2)
```

We can see that the variation on PC 1 is mainly attributable to the intensities around 1670 nm, whereas the wavelengths around 1690 nm are contributing most to PC 2. PC 3 shows the largest loadings at 1638, 1694 and 1206 nm; the latter two are also extreme loadings on PC 4. These wavelengths correspond with areas of significant variation (see Figure 2.1).

An even more interesting visualization is the *biplot* [28, 29]. This shows scores and loadings in one and the same plot, which can makes interpretation

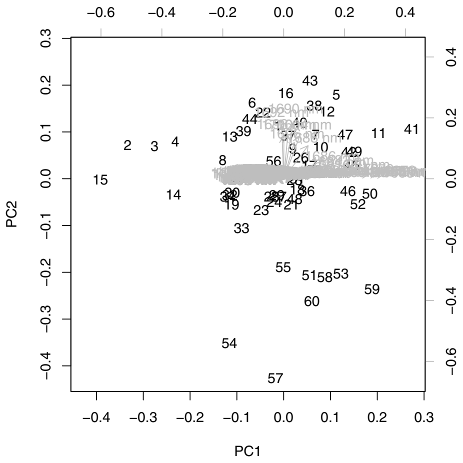

Fig. 4.6. Biplot for the mean-centered gasoline data showing PC 1 versus PC 2.

easier. The origins for the score and loading plots are overlayed, and the two sets of points are plotted in separate axis systems. An example is shown in Figure 4.6:

```
> biplot(nir.prcomp)
```

Again, scores are plotted as individual points, and loadings as arrows. The axes corresponding to the scores are shown on the bottom and the left of the picture; the axis to the right and no top are for the loadings. Since there are many correlated variables, the loading plot looks quite crowded.

We can see in the biplot that wavelength 1690 nm is one of the arrows pointing upward, in the direction of PC 2. Samples 15 and 41 are most different in PC 1, and samples 57 and 43 differ most in PC 2. Let us plot the mean-centered data for these four samples (since the mean-centered data are what is fed to PCA):

```
> extremes <- c(15,41,45,57)
> Xextr <- scale(gasoline$NIR, scale = FALSE)[extremes,])
> wavelengths <- seq(900, 1700, by = 2)
> matplot(wavelengths, t(Xextr),
+         type = "l", xlab = "Wavelength (nm)",
+         ylab = "Intensity (mean-scaled)", lty = c(1,1,2,2),
+         col = c(1,"gray",1,"gray"))
> abline(v = wavelengths[extreme.vars], col = "gray", lty = 3)
> legend("bottomleft", legend = paste("sample", extremes),
+        lty = c(1,1,2,2), col = c(1,2,1,2), bty = "n")
```

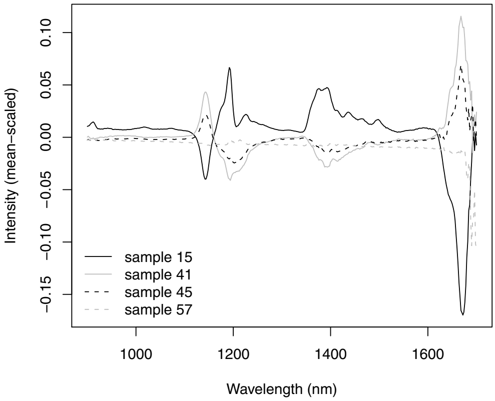

Fig. 4.7. Mean-centered spectra of samples that are extreme in either component.

This results in Figure 4.7. The largest difference in intensity, at 1670 nm, corresponds with the most important variables in PC 1 – and although it is not directly recognizable from the loadings in this figure, this is exactly the wavelength with the largest loading on PC 1. The largest difference in the samples that are extreme in PC 2 is just above that, at 1690 nm, in agreement with our earlier observation. Another major feature just below 1200 nm is represented in the bottom left corner of the loading plot of PC 1 vs PC 2.

Another R function, `princomp`, also does PCA, using the `eigen` decomposition on the covariance matrix instead of directly performing `svd` on the data themselves. Since the `svd`-based calculations are more stable, these are to be preferred for regular applications; `princomp` does allow you to provide a specific covariance matrix that will be used for the decomposition, which can be useful in some situations (see Section 11.1), but is retained mainly for compatibility reasons.

The package **ChemometricsWithR** comes with a set of PCA functions, too, based on the code presented in this chapter. The basic function is `PCA`, and the usual generic functions `print`, `plot`, and `summary` are available, as well as some auxiliary functions such as `screeplot`, `project`, and the extraction functions `variances`, `loadings` and `scores`. To produce, e.g., plots very similar to the ones shown in Figure 4.1, for example, one can issue:

```
> wines.PCA <- PCA(scale(wines))
> scoreplot(wines.PCA, pch = wine.classes, col = wine.classes)
> loadingplot(wines.PCA, show.names = TRUE)
```

One useful feature of these functions is that the percentage of explained variance is automatically shown at the axis labels.

4.6 Related Methods

PCA is not alone in its aim to find low-dimensional representations of high-dimensional data sets. Several other methods try to do the same thing, but rather than finding the projection that maximizes the explained variance, they choose other criteria. In Principal Coordinate Analysis and the related multidimensional scaling methods, the aim is to find a low-dimensional projection that reproduces the experimentally found *distances* between the data points. When these distances are Euclidean, the results are the same or very similar to PCA results; however, other distances can be used as well. Independent Component Analysis maximizes deviations from normality rather than variance, and Factor Analysis concentrates on reproducing covariances. We will briefly review both in the next paragraphs.

4.6.1 Multidimensional Scaling

In some cases, applying PCA to the raw data matrix is not appropriate, for example in situations where regular Euclidean distances do not apply – similarities between chemical structures, e.g., can be expressed easily in several different ways, but it is not at all clear how to represent molecules into fixed-length structure descriptors [30], something that is required by distance measures such as the Euclidean distance. Even when comparing spectra or chromatograms, the Euclidean distance can be inappropriate, for instance in the presence of peak shifts [20,9]. In other cases, raw data are simply not available and the only information one has consists of similarities. Based on the sample similarities, the goal of methods like Multidimensional Scaling (MDS, [31,32]) is to reconstruct a low-dimensional map of samples that leads to the same similarity matrix as the original data (or a very close approximation). Since visualization usually is one of the main aims, the number of dimensions usually is set to two, but in principle one could find an optimal configuration with other dimensionalities as well.

The problem is something like making a topographical map, given only the distances between the cities in the country. In this case, an exact solution is possible in two dimensions since the original distance matrix was calculated from two-dimensional coordinates. Note that although distances can be reproduced exactly, the map still has rotational and translational freedom – in practice this does not pose any problems, however. An amusing example is given by maps not based on kilometers but rather on travel time – the main cities will be moved to the center of the plot since they usually are connected by high-speed trains, whereas smaller villages will appear to be further away. In such a case, and in virtually all practical applications, a two-dimensional plot will not be able to reproduce all similarities exactly.

In MDS, there are several ways to indicate the agreement between the two distance matrices, and these lead to different methods. The simplest approach

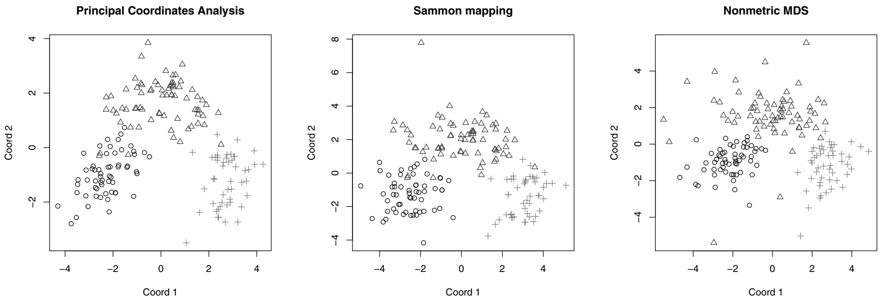

Fig. 4.8. Multidimensional scaling approaches on the wine data: classical MDS (left), Sammon mapping (middle) and non-metric MDS (right).

is to perform PCA on the double-centered distance matrix[3], an approach that is known as Principal Coordinate Analysis, or Classical MDS [33]. The criterion to be minimized is called the stress, and is given by

$$S = \sum_{j<i} (||x_i - x_j|| - e_{ij})^2 = \sum_{j<i} (d_{ij} - e_{ij})^2$$

where e_{ij} corresponds with the true, given, distances, and d_{ij} are the distances between objects x_i and x_j in the low-dimensional space.

In R, this is available as the function `cmdscale`:

```
> wines.dist <- dist(scale(wines))
> wines.cmdscale <- cmdscale(wines.dist)
> plot(wines.cmdscale$points,
+      pch = wine.classes, col = wine.classes,
+      main = "Principal Coordinate Analysis",
+      xlab = "Coord 1", ylab = "Coord 2")
```

This leads to the left plot in Figure 4.8, which up to the reversal of the sign in the first component is exactly equal to the scores plot of Figure 4.1.

Other approaches optimize slightly different criteria: two well-known examples are Sammon mapping and Kruskal-Wallis mapping [34] – both are available in the **MASS** package as functions `sammon` and isoMDS, respectively. Sammon mapping decreases the influence of large distances, which can dominate the map completely. It minimizes the following stress criterion:

$$S = \frac{1}{\sum_i \sum_{j<i} d_{ij}} \sum_i \sum_{j<i} \frac{d_{ij} - e_{ij}}{d_{ij}}$$

[3] Double centering is performed by mean-centering in both row and column dimensions, and subsequently adding the grand mean of the original matrix to center the data around the origin.

Since no analytical solution is available, gradient descent optimization is employed to find the optimum. The starting point usually is the classical solution, but one can also provide another configuration – indeed, one approach to try to avoid local optima is to repeat the mapping starting from many different starting sets, or to use different sets of search parameters. The wine data are a case in point: whereas the default initial step size of .2 generates exactly the same solution as the PCA, tinkering with this parameter leads to a substantial improvement:

```
> wines.sammon <- sammon(wines.dist)

Initial stress        : 0.14753
stress after   0 iters: 0.14753

> wines.sammon <- sammon(wines.dist, magic = .00003)

Initial stress        : 0.14753
stress after  10 iters: 0.14347, magic = 0.002
stress after  19 iters: 0.11250

> plot(wines.sammon$points, main = "Sammon mapping",
+      col = wine.classes, pch = wine.classes,
+      xlab = "Coord 1", ylab = "Coord 2")
```

Note that the function `sammon` returns a list rather than a coordinate matrix: the coordinates in low-dimensional space can be accessed in list element `points`.

The non-metric scaling implemented in `isoMDS` uses a two-step optimization that alternatively finds a good configuration in low-dimensional space, and an appropriate non-monotone transformation. In effect, one finds a set of points that leads to the same *order* of the distances in the low-dimensional approximation and in the real data, rather than resulting in approximately the same distances.

```
> wines.isoMDS <- isoMDS(wines.dist)

initial  value 25.980817
iter   5 value 19.977649
iter  10 value 17.798597
iter  15 value 17.327216
iter  20 value 17.128918
iter  20 value 17.116929
iter  20 value 17.111706
final  value 17.111706
converged

> plot(wines.isoMDS$points, main = "Non-metric MDS",
+      col = wine.classes, pch = wine.classes,
+      xlab = "Coord 1", ylab = "Coord 2")
```

The results of Sammon mapping and IsoMDS are shown in the middle and right plots of Figure 4.8, respectively. Here we again see the familiar horse-shoe shape, although it is somewhat different in nature in the non-metric version in the plot on the right. There, the Grignolino class is much more spread out than the other two.

MDS is popular in the social sciences, but much less so in the life sciences: maybe there its disadvantages are more important. The first drawback is that MDS requires a full distance matrix. For data sets with many thousands of samples this can be prohibitive in terms of computer memory. The other side of the coin is that the (nowadays more common) data sets with many more samples than variables do not present any problem; they can be analysed by MDS easily. The second and probably more important disadvantage is that MDS does not provide an explicit mapping operator. That means that new objects cannot be projected into the lower-dimensional point configuration as we did before with PCA; either one redoes the MDS mapping, or one positions the new samples as good as possible within the space of the mapped ones and takes several iterative steps to obtain a new, complete, set of low-dimensional coordinates. Finally, the fact that the techniques rely on optimization, rather than an analytical solution, is a disadvantage: not only does it take more time, especially with larger data sets, but also the optimization settings may need to be tweaked for optimal performance.

4.6.2 Independent Component Analysis and Projection Pursuit

Variation in many cases equals information, one of the reasons behind the widespread application of PCA. Or, to put it the other way around, a variable that has a constant value does not provide much information. However, there are many examples where the *relevant* information is hidden in small differences, and is easily overwhelmed by other sources of variation that are of no interest. The technique of *Projection Pursuit* [35,36,37] is a generalization of PCA where a number of different criteria can be optimized. One can for instance choose a viewpoint that maximises some grouping in the data. In general, however, there is no analytical solution for any of these criteria, except for the variance criterium used in PCA. A special case of Projection Pursuit is *Independent Component Analysis* (ICA) [38], where the view is taken to maximise deviation from multivariate normality, given by the negentropy J. This is the difference of the entropy of a normally distributed random variable $H(x_{\mathrm{G}})$ and the entropy of the variable under consideration $H(x)$

$$J(x) = H(x_{\mathrm{G}}) - H(x)$$

where the entropy itself is given by

$$H(x) = -\int f(x) \log f(x) dx$$

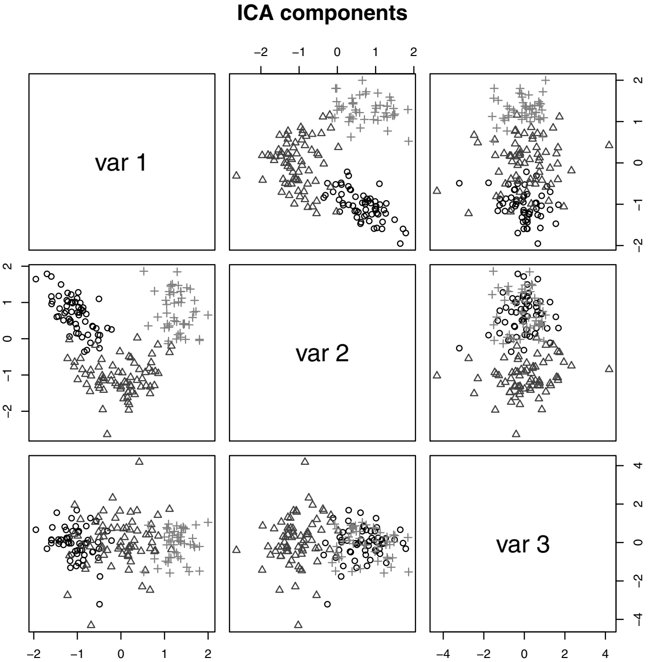

Fig. 4.9. Fast ICA applied to the wine data, based on a three-component model.

Since the entropy of a normally distributed variable is maximal, the negentropy is always positive [39]. Unfortunately, this quantity is hard to calculate, and in practice approximations, such as kurtosis and the fourth moment are used. Package **fastICA** provides the fastICA procedure of Hyvarinen and Oja [40], employing other approximations that are more robust and faster. The algorithm can be executed either fully in R, or using C code for greater speed.

The fastICA algorithm first mean-centers the data, and then performs a "whitening", *viz.* a principal component analysis. These principal components are then rotated in order to optimize the non-gaussianity criterion. Applied to the wine data, this gives the result shown in Figure 4.9:

```
> wines.ica <- fastICA(wines.sc, 3)
> pairs(wines.ica$S, main = "ICA components",
+       col = wine.classes, pch = wine.classes)
```

It is interesting to note that the first ICA component in this case is not related to the difference in variety. Instead, it is the plot of IC 2 versus IC 3 that shows most discrimination, and is most similar to the PCA score plot shown in Figure 4.1. One characteristic that should be noted is that ICA

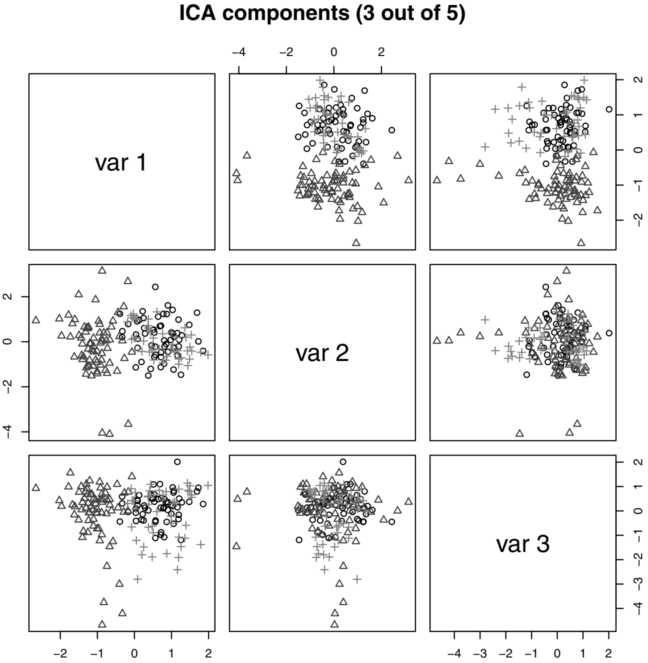

Fig. 4.10. Fast ICA applied to the wine data using five components; only the first three components are shown.

components can change, depending on their number: where in PCA the first component remains the same, no matter how many other components are included, in ICA this is not the case. The pairs plot will therefore be different when taking, e.g., a five-component ICA model:

```
> wines.ica5 <- fastICA(wines.sc, 5)
> pairs(wines.ica5$S[,1:3],
+       main = "ICA components (3 out of 5)",
+       col = wine.classes, pch = wine.classes)
```

In Figure 4.10 we can see that the scores are quite different from the ones in Figure 4.9: now, only component two shows discrimination between the three varieties. In fact, for this model the familiar horse shoe appears in the plot of IC 2 versus IC 5! One should be very cautious in the inspection of ICA score plots. Of course, class separation is not the only criterion we can apply – just because we happen to know in this case what the grouping is does not mean that projections not showing this grouping should be considered “not useful”.

4.6.3 Factor Analysis

Another procedure closely related to PCA is Factor Analysis (FA), developed some eighty years ago by the psychologist Charles Spearman, who hypothesized that a large number of abilities (mathematical, artistic, verbal) could be summarized in one underlying factor "intelligence" [41]. Although this view is no longer mainstream, the idea caught on, and FA can be summarized as trying to describe a set of observed variables with a small number of abstract latent factors.

The technique is very similar to PCA, but there is a fundamental difference. PCA aims at finding a series of rotations in such a way that the first axis corresponds with the direction of most variance, and each subsequent orthogonal axis explains the most of the remaining variance. In other words, PCA does not fit an explicit model. FA, on the other hand, does. For a mean-centered matrix $\boldsymbol{X}$, the FA model is

$$\boldsymbol{X} = \boldsymbol{L}\boldsymbol{F} + \boldsymbol{U}$$

where $\boldsymbol{L}$ is the matrix of loadings on the *common factors* $\boldsymbol{F}$, and $\boldsymbol{U}$ is a matrix of *specific factors*, also called *uniquenesses*. The common factors again are linear combinations of the original variables, and the scores present the positions of the samples in the new coordinate system. The result is a set of latent factors that capture as much variation, shared between variables, as possible. Variation that is unique to one specific variable will end up in the specific factors. Especially in fields like psychology it is customary to try and interpret the common factors like in the original approach by Spearman. Summarizing, it can be stated that PCA tries to represent as much as possible of the diagonal elements of the covariance matrix, whereas FA aims at reproducing the off-diagonal elements [24].

There is considerable confusion between PCA and FA, and many examples can be found where PCA models are actually called factor analysis models: one reason is that the simplest way (in a first approximation) to estimate the FA model of Equation 4.6.3 is to perform a PCA – this method of estimation is called Principal Factor Analysis. However, other methods exist, e.g., based on Maximum Likelihood, that provide more accurate models. The second source of confusion is that for spectroscopic data in particular, scientists are often trying to interpret the PCs of PCA. In that sense, they are more interested in the FA model than in the model-free transformation given by PCA.

Factor analysis is available in R as the function `factanal`. Application to the wine data is straightforward:

```
> wines.fa <- factanal(wines.sc, 3, scores = "regression")
> wines.fa

Call:
factanal(x = wines.sc, factors = 3)
```

```
Uniquenesses:
     alcohol     malic acid            ash    ash alkalinity
       0.393          0.729          0.524             0.068
   magnesium   tot. phenols     flavonoids non-flav. phenols
       0.843          0.198          0.070             0.660
     proanth      col. int.       col. hue          OD ratio
       0.559          0.243          0.503             0.248
     proline
       0.388

Loadings:
                  Factor1 Factor2 Factor3
alcohol                    0.776
malic acid        -0.467           0.211
ash                        0.287   0.626
ash alkalinity    -0.297  -0.313   0.864
magnesium          0.119   0.367
tot. phenols       0.825   0.346
flavonoids         0.928   0.262
non-flav. phenols -0.533  -0.140   0.192
proanth            0.621   0.226
col. int.         -0.412   0.751   0.153
col. hue           0.653  -0.206  -0.170
OD ratio           0.865
proline            0.355   0.684  -0.134

               Factor1 Factor2 Factor3
SS loadings      4.005   2.261   1.310
Proportion Var   0.308   0.174   0.101
Cumulative Var   0.308   0.482   0.583

Test of the hypothesis that 3 factors are sufficient.
The chi square statistic is 159.14 on 42 degrees of freedom.
The p-value is 1.57e-15
```

The default `print` method for a factor analysis object shows the uniquenesses, i.e., those parts that cannot be explained by linear combinations of other variables. The uniquenesses of the variables `ash alkalinity` and `flavonoids`, e.g., are very low, indicating that they may be explained well by the other variables. The loadings are printed in such a way as to draw attention to patterns: only three digits are shown after the decimal point, and smaller loadings are not printed.

Scores can be calculated in several different ways, indicated by the `scores` argument of the `factanal` function. Differences between the methods are usually not very large; consult the manual pages of `factanal` to get more infor-

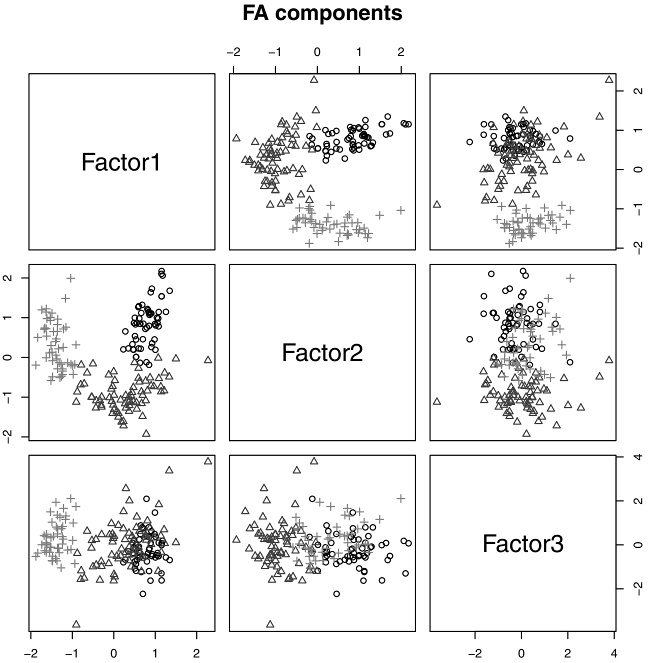

Fig. 4.11. Scores for the FA model of the wine data using three components.

mation about the exact implementation. The result for the regression scores, shown in Figure 4.11, is only slightly different from what we have seen earlier with PCA; the familiar horse shoe is again visible in the first two components.

In Factor Analysis it is usual to rotate the components in such a way that the interpretability is enhanced. One of the ways to do this is to require that as few loadings as possible have large values, something that can be achieved by a so-called *varimax* rotation. This rotation is applied by default in `factanal`, and is the reason why the horseshoe is rotated compared to the PCA score plot in Figure 4.1.

4.6.4 Discussion

Although there are some publications where ICA and FA are applied to data sets in the life sciences, their number is limited, and certainly much lower than the number of applications of PCA. There are several of reasons for this. Both ICA and FA do not have analytical solutions and require optimization to achieve their objectives, which takes more computing time, and can lead to different results, depending on the optimization settings. Moreover, several algorithms are available, each having slightly different definitions, which makes the results harder to compare and to interpret. PCA, on the other hand, always gives the same result (apart from the sign). In particular, PCA scores and loadings do not change when the number of component is altered. Since choosing the "right" number of components can be quite difficult, this is a real advantage over applying ICA. In a typical application, there are so many choices to make with respect to preprocessing, scaling, outlier detection and others, that there is a healthy tendency to choose methods that have as few tweaking possibilities as possible – if not, one can spend forever investigating the effects of small differences in analysis parameters. Nevertheless, there are cases where ICA, FA, or other dimension reduction methods can have definite advantages over PCA, and it can pay to check that.

5

Self-Organizing Maps

In PCA, the most outlying data points determine the direction of the PCs – these are the ones contributing most to the variance. This often results in score plots showing a large group of points close to the centre. As a result, any local structure is hard to recognize, even when zooming in: such points are not important in the determination of the PCs. One approach is to select the rows of the data matrix corresponding to these points, and to perform a separate PCA on them. Apart from the obvious difficulties in deciding which points to leave out and which to include, this leads to a cumbersome and hard to interpret two-step approach. It would be better if a projection can be found that *does* show structure, even within very similar groups of points.

Self-organizing maps (SOMs, [42]), sometimes also referred to as Kohonen maps after their inventor, Teuvo Kohonen, offer such a view. Rather than providing a continuous projection into $\mathbb{R}^2$, SOMs map all data to a set of discrete locations, organized in a regular grid. Associated with every location is a prototypical object, called a *codebook vector*. This usually does not correspond to any particular object, but rather represents part of the space of the data. The complete set of codebook vectors therefore can be viewed as a concise summary of the original data. Individual objects from the data set can be mapped to the set of positions, by assigning them to the unit with the most similar codebook vectors.

The effect is shown in Figure 5.1. A two-dimensional point cloud is simulated where most points are very close to the origin.[1] The codebook vectors of a 5-by-5 rectangular SOM are shown in black; neighbouring units in the horizontal and vertical directions are connected by lines. Clearly, the density of the codebook vectors is greatest in areas where the density of points is greatest. When the codebook vectors are shown at their SOM positions the plot on the right in Figure 5.1 emerges, where individual objects are shown at

[1] The point cloud is a superposition of two bivariate normal distributions, centered at the origin and with diagonal covariance matrices. The first has unit variance and contains 100 points; the other, containing 500 points, has variances of .025.

R. Wehrens, *Chemometrics with R: Multivariate Data Analysis in the Natural Sciences and Life Sciences*, Use R!, DOI 10.1007/978-3-642-17841-2_5,

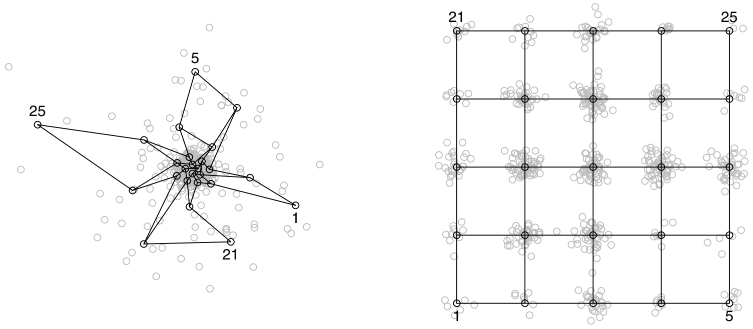

Fig. 5.1. Application of a 5-by-5 rectangular SOM to 600 bivariate normal data points. Left plot: location of codebook vectors in the original space. Right plot: location of data points in the SOM.

a random position close to "their" codebook vector. The codebook vectors in the middle of the map are the ones that cover the center of the data density, and one can see that these contain most data points. That is, relations within this densely populated area can be investigated in more detail.

5.1 Training SOMs

A SOM is trained by repeatedly presenting the individual samples to the map. At each iteration, the current sample is compared to the codebook vectors. The most similar codebook vector (the "winning unit") is then rotated slightly in the direction of the mapped object. This is achieved by replacing it with a weighted average of the old values of the codebook vector, cv_i, and the values of the new object obj:

$$cv_{i+1} = (1 - \alpha)\ cv_i + \alpha\ obj \tag{5.1}$$

The weight, also called the learning rate α, is a small value, typically in the order of 0.05, and decreases during training so that the final adjustments are very small.

As we shall see in Section 6.2.1, the algorithm is very similar in spirit to the one used in k-means clustering, where cluster centers and memberships are alternatingly estimated in an iterative fashion. The crucial difference is that not only the winning unit is updated, but also the other units in the "neighbourhood" of the winning unit. Initially, the neighbourhood is fairly large, but during training it decreases so that finally only the winning unit is

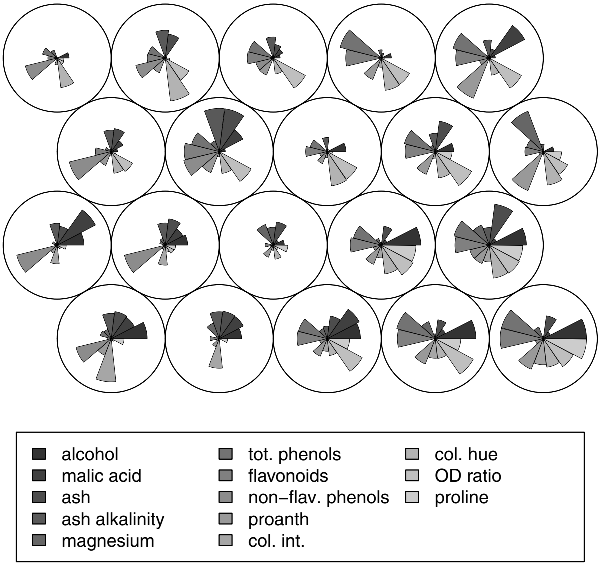

Fig. 5.2. Codebook vectors for a SOM mapping of the autoscaled wine data. The thirteen variables are shown counterclockwise, beginning in the first quadrant.

updated. The effect is that neighbouring units in general are more similar than units far away. Or, to put it differently, moving through the map by jumping from one unit to its neighbour would see gradual and more or less smooth transitions in the values of the codebook vectors. This is clearly visible in the mapping of the autoscaled wine data to a 5-by-4 SOM, using the **kohonen** package:

```
> wines.som <- som(wines.sc, somgrid(5, 4, "hexagonal"))
> plot(wines.som, type = "codes")
```

The result is shown in Figure 5.2. Units in this example are arranged in a hexagonal fashion and are numbered row-wise from left to right, starting from the bottom left. The first unit for instance, is characterized by relatively large values of `alcohol`, `flavonoids` and `proanth`; the second unit, to the right of the first, has lower values for these variables, but still is quite similar to unit number one.

The codebook vectors are usually initialized by a random set of objects from the data, but also random values in the range of the data can be employed. Sometimes a grid is used, based on the plane formed by the first two PCs. In practice, the initialization method will hardly ever matter; however, starting from other random initial values will lead to different maps. The conclusions drawn from the different maps, however, tend to be very similar.

The training algorithm for SOMs can be tweaked in many different ways. One can, e.g., update units using smaller changes for units that are further away from the winning unit, rather than using a constant learning rate within the neighbourhood. One can experiment with different rates of decreasing values for learning rate and neighbourhood size. One can use different distance measures. Regarding topology, hexagonal or rectangular ordering of the units is usually applied; in the first case, each unit has six equivalent neighbours, unless it is at the border of the map, in the second case, depending on the implementation, there are four or eight equivalent neighbours. The most important parameter, however, is the size of the map. Larger maps allow for more detail, but may contain more empty units as well. In addition, they take more time to be trained. Smaller maps are more easy to interpret; groups of units with similar characteristics are more easily identified. However, they may lack the flexibility to show specific groupings or structure in the data. Some experimentation usually is needed. As a rule of thumb, one can consider the object-to-unit ratio, which can lead to useful starting points. In image segmentation applications, for instance, where hundreds of thousands of (multivariate) pixels need to be mapped, one can choose a map size corresponding to an average of several hundreds of pixels per unit; in other applications where the number of samples is much lower, a useful object-to-unit ratio may be five. One more consideration may be the presence of class structure: for every class, several units should be allocated. This allows intra-class structure to be taken into account, and will lead to a better mapping.

Finally, there is the option to close the map, i.e., to connect the left and right sides of the map, as well as the bottom and top sides. This leads to a toroidal map, resembling the surface of a closed tube. In such a map, all differences between units have been eliminated: there are no more edge units, and they all have the same number of neighbours. Whereas this may seem a desirable property, there are a number of disadvantages. First, it will almost certainly be depicted as a regular map with edges, and when looking at the map one has to remember that the edges in reality do not exist. In such a case, similar objects may be found in seemingly different parts of the map that are, in fact, close together. Another pressing argument against toroidal maps is that in many cases the edges serve a useful purpose: they provide refuge for objects that are quite different from the others. Indeed, the corners of non-toroidal maps often contain the most distinct classes.

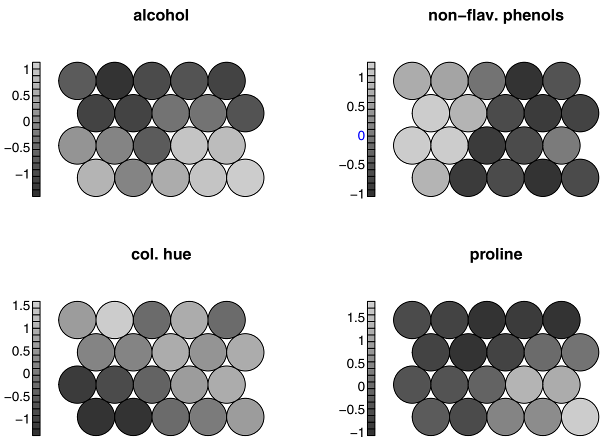

Fig. 5.3. Separate maps for the contributions of individual variables to the codebook vectors of the SOM shown in Figure 5.2.

5.2 Visualization

Several different visualization methods are provided in the **kohonen** package: one can look at the codebook vectors, the mapping of the samples, and one can also use SOMs for prediction. Here, only a few examples are shown. For more information, consult the manual pages of the `plot.kohonen` function, or the software description [43].

For multivariate data, the locations of the codebook vectors can not be visualized as was done for the two-dimensional data in Figure 5.1. In the **kohonen** package, the default is to show `segment` plots, such as in Figure 5.2 if the number of variables is smaller than 15, and a line plot otherwise. One can also zoom in and concentrate on the values of just one of the variables:

```
> par(mfrow = c(2,2))
> for (i in c(1, 8, 11, 13))
+   plot(wines.som, "property", property = wines.som$codes[,i],
+        main = colnames(wines)[i])
```

Clearly, in these plots, shown in Figure 5.3, there are regions in the map where specific variables have high values, and other regions where they are low. Areas of high values and low values are much more easily recognized than in Figure 5.2.

Perhaps the most important visualization is to show which objects map in which units. In the **kohonen** package, this is achieved by supplying the the `type = "mapping"` argument to the plotting function. It allows for using different plotting characters and colors (see Figure 5.4):

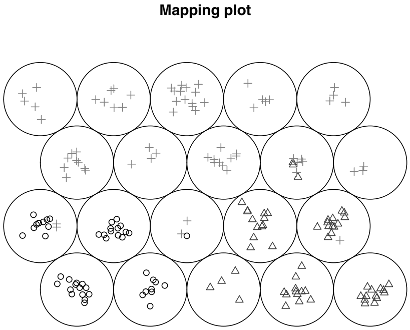

Fig. 5.4. Mapping of the 177 wine samples to the SOM from Figure 5.2. Circles correspond to Barbera, triangles to Barolo, and plusses to Grignolino wines.

```
> plot(wines.som, type = "mapping",
+      col = as.integer(vintages), pch = as.integer(vintages))
```

Again, one can see that the wines are well separated. Some class overlap remains with the Grignolinos in units 6, 8, 10 and 14. These plots can be used to make predictions for new data points: when the majority of the objects in a unit are, e.g., of the Barbera class, one can hypothesise that this is also the most probably class for future wines that end up in that unit. Such predictions can play a role in determining authenticity, an economically very important application.

Since SOMs are often used to detect grouping in the data, it makes sense to look at the codebook vectors more closely, and investigate if there are obvious class boundaries in the map – areas where the differences between neighbouring units are relatively large. Using a colour code based on the average distance to neighbours gives a quick and simple idea of where the class boundaries can be found. This idea is often referred to as the “U-matrix” [44], and can be employed by issuing:

```
> plot(wines.som, type = "dist.neighb")
```

The resulting plot is shown in Figure 5.5. The map is too small to really be able to see class boundaries, but one can see that the centers of the classes (the bottom left corner for Barbera, the bottom right corner for Barolo, and the top row for the Grignolino variety) correspond to areas of relatively small distances, i.e., high homogeneity.

Training progress, and an indication of the quality of the mapping, can be obtained using the following plotting commands:

```
> par(mfrow = c(1,2))
> plot(wines.som, "changes")
> plot(wines.som, "quality")
```

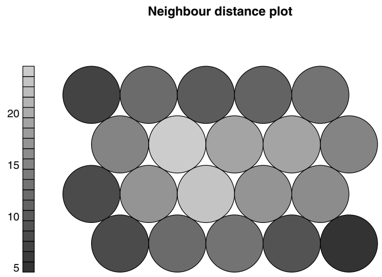

Fig. 5.5. Summed distances to direct neighbours: the U-matrix plot for the mapping of the wine data.

This leads to the plots in Figure 5.6. The left plot shows the average distance (expressed per variable) to the winning unit during the training iterations, and the right plot shows the average distance of the samples and their corresponding codebook vectors after training. Note that the latter plot concentrates on distances *within* the unit whereas the U-matrix plot in Figure 5.5 visualizes average distances *between* neighbouring units.

Finally, an indication of the quality of the map is given by the mean distances of objects to their units:

```
> summary(wines.som)
```

```
som map of size 5x4 with a hexagonal topology.
Training data included; dimension is 177 by 13.
Mean distance to the closest unit in the map: 3.580196
```

The `summary` function indicates that an object, on average, has a distance of 3.6 units to its closest codebook vector. The plot on the left in Figure 5.6 shows that the average distance drops during training: codebook vectors become more similar to the units that are mapped to them. The plot on the right, finally, shows that the distances within units can be quite different. Interestingly, some of the units with the largest spread only contain Grignolinos (units 16 and 20), so the variation can not be attributed to class overlap alone. For other visualization possibilities, consult the manual page for the function `plot.kohonen`.

5.3 Application

The main attraction of SOMs lies in the applicability to large data sets; even if the data are too large to be loaded in memory in one go, one can train the map sequentially on (random) subsets of the data. It is also possible to update the map when new data points become available. In this way, SOMs

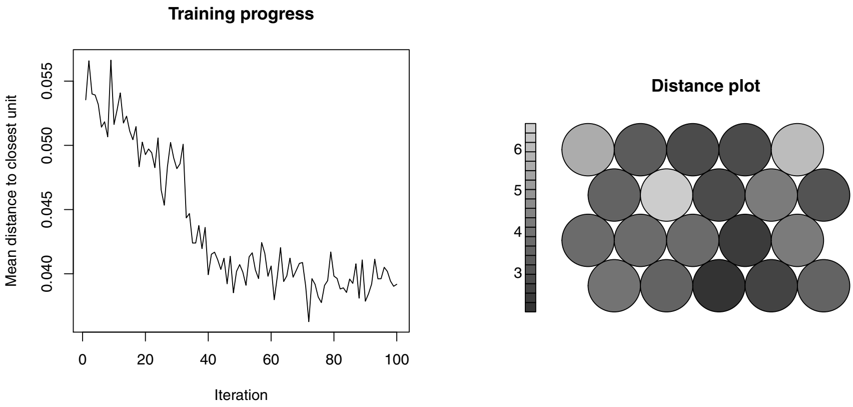

Fig. 5.6. Quality parameters for SOMs: the plot on the left shows the decrease in distance between objects and their closest codebook vectors during training. The plot on the right shows the mean distances between objects and codebook vectors per unit.

provide a intuitive and simple visualization of large data sets in a way that is complementary to PCA. An especially interesting feature is that these maps can show grouping of the data without explicitly performing a clustering. In large maps, sudden transitions between units, as visualized by e.g. a U-matrix plot, enable one to view the major structure at a glance. In smaller maps, this often does not show clear differences between groups – see Figure 5.5 for an example. One way to find groups is to perform a clustering of the individual codebook vectors. The advantage of clustering the codebook vectors rather than the original data is that the number of units is usually orders of magnitude smaller than the number of objects.

As a practical example, consider the mapping of the 654 samples from the `prostate` data using the complete, 10,523-dimensional mass spectra in a 7-by-5 map. This would on average lead to almost twenty samples per unit and, given the fact that there are three classes, leave enough flexibility to show within-class structure as well. The plot in Figure 5.7 is produced with the following code:

```
> X <- t(Prostate2000Raw$intensity)
> types <- Prostate2000Raw$type
> prostate.som <- som(X, somgrid(7, 5, "hexagonal"))
> plot(prostate.som, "mapping", col = as.integer(types),
+      pch =  as.integer(types),
+      main = "Prostate data")
> legend("bottom", legend = levels(types),
+        col = 1:3, pch = 1:3, ncol = 3)
```

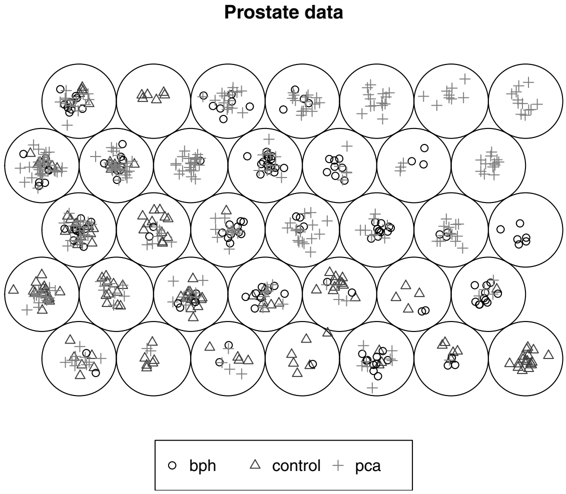

Fig. 5.7. Mapping of the prostate data. The cancer samples (pca) lie in the bottom left part of the map. Although there is considerable class overlap, one can clearly see that the control samples are at the opposite end of the map, and the benign enlargments are in between.

Clearly, there is considerable class overlap. However, some separation can be observed: the cancer and control samples are on opposite ends of the map, with the benign enlargments occupying middle ground. To investigate differences between the individual units, one can plot the codebook vectors of some of the units containing only objects from one class:

```
> cols <- c(2,1,3)
> units <- c(7,21,35)
> par(mfrow = c(3,1))
> for (i in 1:3)
+   plot(prostate.som$codes[units[i],], type = "l",
+        col = cols[i], main = paste("Unit", units[i]),
+        ylab = "codebook vector")
```

These codebook vectors, shown in Figure 5.8, display appreciable differences. The cancer samples, for instance, show peaks at m/z values of 6000 and 6200 that are absent in the controls and much lower in the `bph` samples. The large peaks at 3300, 4000 and 4300 m/z units on the other hand, are only present in the non-cancer samples.

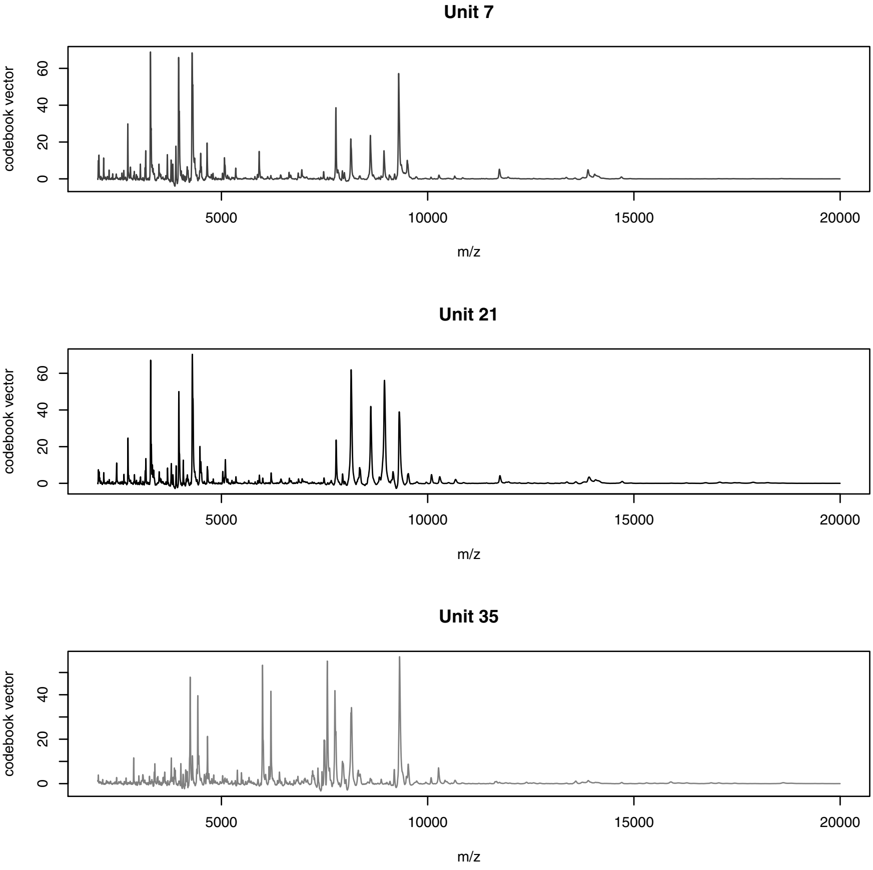

Fig. 5.8. Codebook vectors for three units from the far-right side of the map in Figure 5.7, containing only samples from one class: unit seven contains controls, unit 21 bph samples and unit 35 pca samples.

5.4 R Packages for SOMs

Apart from the **kohonen** package used in this chapter, other R packages are available implementing SOMs, most notably Ripley's **class** package [45], on which the **kohonen** package is based. Apart from the SOM training algorithm described earlier, the **class** package also implements another training strategy for SOMs, known as the batch algorithm. Rather than updating the map after every single object has been mapped, the update is performed after the complete data matrix has been presented; all codebook vectors are replaced by the mean of all objects that are mapped to them. Again, a neighbourhood is taken into account that gets smaller during training. The training loop contains only five (elegant) lines of R code. The advantage of the batch algorithm is that it dispenses with one of the parameters of the SOM: the learning rate α is no longer needed. The main disadvantage is that it is less stable and more likely to end up in a local optimum. Another package, often used in clustering of microarray data, is the **som** package. This package also implements the batch algorithm and provides extra flexibility in training parameters.

One advantage of SOMs is that other measures than the usual Euclidean distance can be used. An example can be found in the package **wccsom** [46], which applies a specially devised distance measure to compare X-ray powder diffractograms. In these patterns, the position rather than the intensity of features is the primary information, and therefore package **wccsom** (which is based on the **kohonen** package) utilises a cross-correlation-based distance, called wcc [20]. In cases where binary strings are compared, a measure like the Tanimoto distance can be used, which is just the fraction of bits that are not equal in the two strings. This is the default measure in the `distance` function in the **fingerprint** package, specifically desiged to handle binary descriptors.

5.5 Discussion

Conceptually, the SOM is most related to MDS, seen in Section 4.6.1. Both, in a way, aim to find a configuration in two-dimensional space that represents the distances between the samples in the data. Whereas (metric forms of) MDS focus on the preservation of the actual distances, SOMs provide a topological mapping, preserving the *order* of the distances, at least for the smallest ones. Because of this, an MDS mapping is often dominated by the larger distances, even when using methods like Sammon mapping, and the configuration of the finer structure in the data may not be well preserved. In SOMs, on the other hand, a big distance between the positions of two samples in the map does not mean that they are very dissimilar: if the map is too large, and not well trained, two regions in the map that are far apart may very well have quite similar codebook vectors. What one *can* say is that objects mapped to the same or to neighbouring units are likely to be similar.

Both MDS and SOMs operate using distances rather than the original data to determine the position in the low-dimensional representation of the data. This can be a considerable advantage when working with high-dimensional data: even when the number of variables is in the tens or hundreds of thousands, the distances between objects can be calculated fairly quickly. Obviously, MDS, in particular, runs into trouble when the number of samples gets large – SOMs can handle that more easily because of the iterative training procedure employed. It is not even necessary to have all the data in memory simultaneously.

Using SOMs is doubtful when the number of samples is low, although applications have been published with fewer than fifty objects. If the number of units in the map is much smaller than the number of objects in such cases, one loses the advantage of the spatial smoothness in the map, and one could just as well perform a clustering; if the number of units approaches the number of objects, it is more likely than not that the majority of the objects will occupy a unit by itself, which is not very informative either.

One should realise that in the case of correlated variables the distances that are calculated may be a bit biased: a group of highly correlated variables will have a major influence on the distance between objects. In areas like, e.g., quantitative structure-activity relationships (QSAR), it is usual to calculate as many chemical structure descriptors as possible in order to define the two- or three-dimensional structure of a set of compounds. Many of these descriptors are variations on a theme: some groups measure properties related to dipole, polarizability, surface area, etcetera. The influence of one single descriptor capturing information that is unrelated to the hundreds of other descriptors can easily be lost when calculating distances. For SOMs, one simple solution is to decorrelate the (scaled) data, e.g., using PCA, and to calculate distances using the scores.

6

Clustering

As we saw earlier in the visualizations provided by methods like PCA and SOM, it is often interesting to look for structure, or groupings, in the data. However, these methods do not explicitly define clusters; that is left to the pattern recognition capabilities of the scientist studying the plot. In many cases, however, it is useful to rely on somewhat more formal methods, and this is where clustering methods come in. They are usually based on object-wise similarities or distances, and since the late nineties have become hugely popular in the area of high-throughput measurement techniques in biology, such as DNA microarrays. There, the activities of tens of thousands of genes are measured, often as a function of a specific treatment, or as a time series. Of course, the question is which genes show the same activity pattern: if an unknown gene has much the same behaviour as another gene of which it is known that it is involved in a process like cell differentiation, one can hypothesise that the unknown gene is somehow related to this process as well.

With only a slight exaggeration one could say that there are about as many clustering algorithms as there are scientists and by no means do these methods always give the same results. Modern software packages have made many of these clustering methods available to a wide audience; unfortunately, this provides the temptation to try all methods in order to get the result one is looking for, rather than the result that is suggested by the data. There are no formal rules to help you decide which clustering method to use.

One of the reasons for this is that most clustering methods are heuristic in nature, rather than that they stem from solid statistical foundations. Moreover, assessing the quality of the clustering, or *validation*, is a problem: since the "real" clustering is by definition unknown (otherwise it would be more appropriate to use a supervised approach such as the classification methods described in Chapter 7) we can not say that one clustering is better than the other. Also cluster characteristics (sphericity, density, ...) can not be used for this, since different clustering methods "optimize" different criteria. It is often difficult for users to get a good idea of the behaviour of the separate methods, since our visualization abilities break down in more than three dimensions,

R. Wehrens, *Chemometrics with R: Multivariate Data Analysis in the Natural Sciences and Life Sciences*, Use R!, DOI 10.1007/978-3-642-17841-2_6,

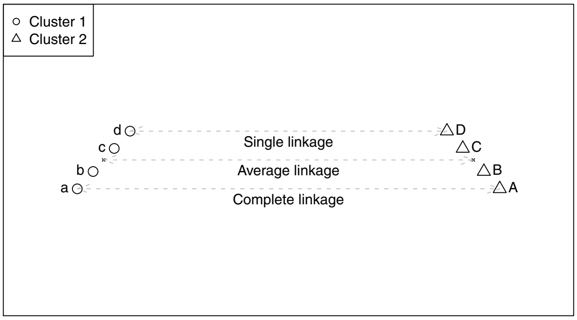

Fig. 6.1. Distances between clusters: single linkage, average linkage and complete linkage consider the closest points, the averages, and the farthest points, respectively.

and at the same time the assumptions behind the clustering methods are often unknown.

In this chapter, we concentrate on several popular classes of methods. Hierarchical methods are represented by *single*, *average* and *complete linkage*, respectively, while *k-means* is an example of partitional methods. Both yield "crisp" clusterings; objects belong to exactly one cluster. More sophisticated methods lead to a clustering where membership values are assigned to each object: the object can be assigned to the cluster with the highest membership value. An example is given by model-based clustering methods.

6.1 Hierarchical Clustering

Quite often, data have a hierarchical structure in the sense that groups consist of mutually exclusive sub-groups. This is often visualized in a tree-like structure, called a dendrogram. The dendrogram presents an intuitive and appealing way for visualising the hierarchical structure: the y-axis indicates the "distance" between different groups, whereas the connections show where successive splits (or joins) take place.

Hierarchical clustering starts with a square matrix containing distances or (dis)similarities; in the following we will assume we have the data in the form of distances. It is almost always performed in a bottom-up fashion. Starting with all objects in separate clusters, one looks for the two most similar clusters and joins them. Then, the distance matrix is updated. There are several possibilities to determine the distance between clusters. One option is to take the shortest distance between clusters. In Figure 6.1 this would correspond to the distance between objects `d` and `D`. This choice leads to the *single-linkage* algorithm. It joins two groups if any members of both groups are close together, a strategy that is sometimes also referred to as friends-of-friends: "any friend of yours is my friend, too!".

The opposite strategy is *complete linkage* clustering: there, the distance between clusters is determined by the objects in the respective clusters that are furthest apart – in Figure 6.1 objects `a` and `A` . In other words: to belong to the same cluster, the distances to *all* cluster members must be small[1]. This strategy leads to much more compact and equal-sized clusters. Of course, intermediate strategies are possible, too. Taking the distance between cluster means leads to *average linkage*. *Ward's method* explicitly takes into account the cluster size in calculating a weighted average, and in many cases gives very similar result to average linkage.

Let us see how this works by clustering a random subset of the wine data. In R hierarchical clustering is available through function `hclust`, which takes an object of class `dist` as its first argument:

```
> subset <- sample(nrow(wines), 20)
> wines.dist <- dist(wines.sc[subset,])
> wines.hcsingle <-  hclust(wines.dist, method = "single")
> plot(wines.hcsingle, labels = vintages[subset])
```

This leads to the dendrogram at the top in Figure 6.2. When we go down in distance, starting from the top, two Grignolino samples are split off from the main branch as singletons before the whole Barbera cluster is identified. Going down even further, the Grignolino samples are split off and finally we end up with the group of Barolos. This makes sense: also the PCA scoreplots show the Barolo (and Barbera) samples to be more homogeneous than the Grignolinos.

The bottom plot in Figure 6.2 shows the dendrogram from complete linkage, obtained by:

```
> wines.hccomplete <- hclust(wines.dist, method = "complete")
> plot(wines.hccomplete, labels = vintages[subset])
```

The structure is very different: immediately there is a split between Barbera and Barolo wines, with the main body of the Grignolinos at the Barbera side. Thus, even with a small and relatively simple data set and two related clustering methods, very different results can be obtained.

In principle, a dendrogram from a hierarchical clustering method in itself is not yet a clustering, since it does not give a grouping as such. However, these can be obtained by "cutting" the diagram at a certain height: all objects that are connected are supposed to be in one and the same cluster. For this, function `cutree` is available, which either takes the height at which to cut, or the number of clusters to obtain as an argument. In this case, let's cut at a height of 3.3:

```
> wines.cl.single <- cutree(wines.hcsingle, h = 3.3)
> table(wines.cl.single, vintages[subset])
```

[1] We can only be friends if all our friends are friends of *both* of us.

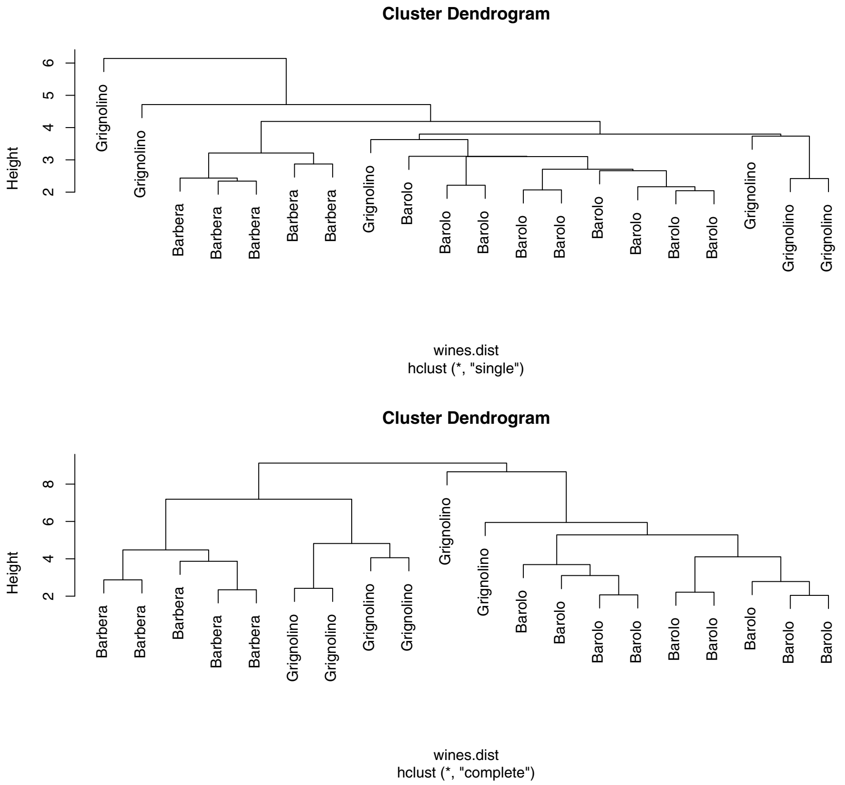

Fig. 6.2. Single linkage clustering (top) and complete linkage clustering (bottom) of 20 samples from the `wine` data.

```
wines.cl.single Barbera Barolo Grignolino
              1       5      0          0
              2       0      0          1
              3       0      9          0
              4       0      0          1
              5       0      0          1
              6       0      0          2
              7       0      0          1
```

The clustering is perfect in the sense that there are no mixed clusters; on the other hand, the Grignolino cluster is split over five different clusters. Clustering the complete wine data with single linkage does not lead to any useful result: even cutting the dendrogram at 40 clusters does not separate the Barolo wines from the Grignolinos.

Now we turn to the complete data set, and recalculate the clusterings. Single linkage, cut at a height to obtain three clusters, does not show anything useful:

```
> wines.dist <- dist(wines.sc)
> wines.hcsingle <- hclust(wines.dist, method = "single")
> table(vintages, cutree(wines.hcsingle, k = 3))

vintages      1  2  3
  Barbera    48  0  0
  Barolo     58  0  0
  Grignolino 67  3  1
```

Almost all samples are in cluster 1, and small bits of the data set are chipped off leading to clusters 2 and 3, each with only very few elements. On the other hand, the three-cluster solution from complete linkage is already quite good:

```
> wines.hccomplete <- hclust(wines.dist, method = "complete")
> table(vintages, cutree(wines.hccomplete, k = 3))

vintages      1  2  3
  Barbera     3  0 45
  Barolo     50  8  0
  Grignolino 14 52  5
```

Cluster 1 corresponds to mainly Barolo wines, cluster two to Grignolinos, and cluster three to the Barberas. Of course, there still is significant overlap between the clusters.

Hierarchical clustering methods enjoy great popularity: the intuitive visualization through dendrograms is one of the main reasons. These also provide the opportunity to see the effects of increasing the number of clusters, without actually recalculating the cluster structure. Obviously, hierarchical clustering will work best when the data actually have a hierarchical structure: that is, when clusters contain subclusters, or when some clusters are more similar than others. In practice, this is quite often the case.

A further advantage is that the clustering is unique: no random element is involved in creating the cluster model. For many other clustering methods, this is not the case. Note that the uniqueness property is present only in the case that there are no ties in the distances. If there are, one may obtain several different dendrograms, depending on the order of the data and the actual implementation of the software. Usually, the first available merge with the minimal distance is picked. When equal distances are present, one or more equivalent merges are possible, which may lead to different dendrograms. An easy way to investigate this is to repeat the clustering many times on distance matrices from data where the rows have been shuffled.

There are a number of drawbacks to hierarchical clustering, too. For data sets with many samples (more than ten thousand, say) these methods are less suitable. To start with, calculating the distance matrix may be very expensive, or even impossible. More importantly, interpreting the dendrograms quickly becomes cumbersome, and there is a real danger of over-interpretation. Examples where hierarchical methods are used with large data sets can be found in the field of DNA microarrays, where the ground-breaking paper of Eisen et al. [47] seems to have set a trend.

There are a number of cases where the results of hierarchical clustering can be misleading. The first is the case where in reality there is no class structure. Cutting a dendrogram will always give you clusters: unfortunately, there is no warning light flashing when you investigate a data set with no class structure. Furthermore, even when there are clusters, they may be too close to separate, or they may overlap. In these cases it is impossible to conclude anything about individual cases (although it can still be possible to infer characteristics of the clusters as a whole). The two keys to get out of this conundrum are formed by the use of prior information, and by visualization. If you know class structure is present, and you already have information about part of that structure, the clustering methods that fail to reproduce that knowledge obviously are not performing well, and you are more likely to trust the results of the methods that do find what you already know. Another idea is to visualise the (original) data, and give every cluster a different colour and plotting symbol. One can easily see if clusters are overlapping or are nicely separated. Note that the dendrogram can be visualized in a number of equivalent ways: the ordering of the groupings from left to right is arbitrary to some extent and may depend on your software package.

The **cluster** package in R also provides functions for hierarchical clustering: `agnes` implements single, average and complete linkage methods but also allows more control over the distance calculations using the `method = "flexible"` argument. In addition, it provides a coefficient measuring the amount of cluster structure, the “agglomerative coefficient”, ac:

$$ac = \frac{1}{n} \sum_i (1 - m_i)$$

where the summation is over all n objects, and m_i is the ratio of the dissimilarity of the first cluster an object is merged to and the dissimilarity level of the final merge (after which only one cluster remains). Compare these numbers for three hierarchical clusterings of the wine data:

```
> wines.agness <- agnes(wines.dist, method = "single")
> wines.agnesa <- agnes(wines.dist, method = "average")
> wines.agnesc <- agnes(wines.dist, method = "complete")
> cbind(wines.agness$ac, wines.agnesa$ac, wines.agnesc$ac)
```

```
[1,] 0.538022 0.6994485 0.8162538
```

Complete linkage is doing the best job for these data, according to the agglomerative coefficient.

6.2 Partitional Clustering

A completely different approach is taken by partitional clustering methods. Instead of starting with individual objects as clusters and progressively merging similar clusters, partitional methods choose a set of cluster centers in such a way that the overall distance of all objects to the closest cluster centers is minimised. Algorithms are iterative and usually start with random cluster centers, ending when no more changes in the cluster assignments of individual objects are observed. Again, many different flavours exist, each with its own characteristics. In general, however, these algorithms are very fast and are suited for large numbers of objects. The calculation of the complete distance matrix is unnecessary - only the distances to the cluster centers need to be calculated, where the number of clusters is much smaller than the number of objects - and this saves resources. Two examples will be treated here: k-means and k-medoids. The latter is a more robust version, where outlying observations do not influence the clustering to a large extent.

6.2.1 K-Means

The k-means algorithm is very simple and basically consists of two steps. It is initialized by a random choice of cluster centers, e.g. a random selection of objects in the data set or random values within the range for each variable. Then the following two steps are iterated:

1. Calculate the distance of an object to all cluster centers and assign the object to the closest center; do this for all objects.
2. Replace the cluster centers by the means of all objects assigned to them.

Note the similarity to the training of SOMs in Chapter 5, particular to the batch training algorithm. The main difference is that the SOM imposes a spatial ordering in the units so that also inter-unit relations are observed. However, the goals of the two methods are quite different: SOMs aim at providing a suitable mapping to two dimensions, and the units should not be seen as individual clusters, whereas k-means explicitly focusses on finding a specific number of groups. The R function is conveniently called `kmeans`. Application to the wine data leads to the following result:

```
> wines.km <- kmeans(wines.sc, centers = 3)
> wines.km

K-means clustering with 3 clusters of sizes 51, 61, 65

Cluster means:
  alcohol malic acid    ash ash alkalinity magnesium
1   0.174      0.864  0.187          0.517    -0.065
2   0.833     -0.301  0.366         -0.607     0.569
3  -0.918     -0.395 -0.491          0.164    -0.483
```

```
  tot. phenols flavonoids non-flav. phenols proanth
1      -0.9711    -1.2062            0.7192 -0.7717
2       0.8877     0.9802           -0.5617  0.5758
3      -0.0711     0.0266           -0.0371  0.0651
  col. int. col. hue OD ratio proline
1     0.938   -1.157   -1.287  -0.400
2     0.170    0.475    0.775   1.130
3    -0.896    0.461    0.282  -0.746

Clustering vector:
  [1] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 [28] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 [55] 2 2 2 2 3 3 1 3 3 3 3 3 3 3 3 3 3 3 2 3 3 3 3 3 3 3 3
 [82] 3 1 3 3 3 3 3 3 3 3 3 3 2 3 3 3 3 3 3 3 3 3 3 3 3 3 3
[109] 3 3 3 3 3 3 3 3 3 1 3 3 2 3 3 3 3 3 3 3 3 1 1 1 1 1 1
[136] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
[163] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

Within cluster sum of squares by cluster:
[1] 326 382 559

Available components:
[1] "cluster"  "centers"  "withinss" "size"
```

The algorithm not only returns the clustering of the individual objects, but also cluster-specific information such as the sum of squares, the cumulative distance of all cluster objects to the center of the cluster.

So the question now is: how good is the agreement with the vintages? Let's see:

```
> table(vintages, wines.km$cluster)
```

```
vintages      1  2  3
  Barbera     0 48  0
  Barolo      0  0 58
  Grignolino 65  3  3
```

Only six of the Grignolino samples are classified as Barbera and Barolo wines; a lot better than the complete-linkage solution.

The k-means algorithm enjoys great popularity through its simplicity, ease of interpretation, and speed. It does have a few drawbacks, however. We already mentioned the fact that one should pick the number of clusters in advance. In general, the correct number (if such a thing exists at all) is never known, and one will probably try several different clusterings with different numbers of clusters. Whereas hierarchical clustering delivers this in one go - the dendrogram only has to be cut at different positions - for k-means clustering (and partitional methods in general) one should repeat the whole clustering procedure. As already said, the results with four or five clusters may differ dramatically.

Worse, even a repeated clustering with the *same* number of clusters will give a different result, sometimes even a very different result. Remember that we start from a random initialization: an unlucky starting point may get the algorithm stuck in a local minimum. Repeated application, starting from different initial guesses, gives some idea of the variability. The `kmeans` function returns the within-cluster sums of squares for the separate clusters, which can be used as a quality criterion:

```
> wines.km <- kmeans(wines.sc, centers = 3)
> best <- wines.km
> for (i in 1:100) {
>   tmp <- kmeans(wines.sc, centers = 3)
>   if (sum(tmp$withinss) < sum(best$withinss))
>     best <- tmp
> }
```

One can then pick the one that leads to the smallest overall distance. In this particular case, the overall best solution is found every time – the wine data do not present that much of a problem. The `kmeans` function has a built-in argument for repeating the clustering and only returning the best solution. Thus, the loop in the previous example can be replaced by

```
> wines.km <- kmeans(wines.sc, centers = 3, nstart = 100)
```

Several minima with comparable overall distance measure may exist, so that different but equally good clustering solutions can be found by the algorithm.

6.2.2 K-Medoids

In k-means, cluster centers are given by the mean coordinates of the objects in that cluster. Since averages are very sensitive to outlying observations, the clustering may be dominated by a few objects, and the interpretation may be difficult. One way to resolve this is to assess clusterings with more groups than expected: the outliers may end up in a cluster of their own. A more practical alternative would be to use a more robust algorithm where the influence of outliers is diminshed. One example is the k-medoids algorithm [48], available in R through the function `pam` – Partitioning Around Medoids – in the **cluster** package. Rather than finding cluster centers at optimal positions, k-medoids aims at finding k representative objects within the data set. Typically, the sum of the distances is minimized rather than the sum of the squared distances, decreasing the importance of large distances.

Applied to the wine data, k-medoids gives the following result:

```
> wines.pam <- pam(wines.dist, k = 3)
> wines.pam
```

```
Medoids:
      ID
[1,]  35  35
[2,] 106 106
[3,] 148 148
Clustering vector:
  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 [28] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 [55] 1 1 1 1 2 2 2 2 1 2 1 2 2 2 1 2 1 2 1 1 2 2 2 1 1 2 2
 [82] 2 3 2 2 2 2 2 2 2 2 2 2 2 1 1 2 1 2 2 2 2 2 2 2 2 2 2
[109] 1 1 2 2 2 2 2 2 2 2 2 1 1 2 2 1 2 2 2 2 2 3 3 3 3 3 3
[136] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
[163] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
Objective function:
   build     swap
2.908525 2.808552

Available components:
[1] "medoids"    "id.med"     "clustering" "objective"
[5] "isolation"  "clusinfo"   "silinfo"    "diss"
[9] "call"
```

The result presents the medoids with their row numbers. The objective function, the sum of the distances of objects to the medoids, is reported in two stages: the first stage serves to find a good initial set of medoids, whereas the second stage performs a local search, trying all possible medoid swaps until no more improvement can be found. In this rather simple example, the average distance after the second stage has decreased by 0.1, compared to the distances to the initial set of medoids – not a huge decrease.

The implementation of `pam` in the **cluster** package comes with additional visualization methods. The first is the "silhouette" plot [48]. It shows a quality measure for individual clusterings: object with a high silhouette width (close to 1) are very well clustered, while objects with low values lie in between two or more clusters. Objects with a negative value may be even in the wrong cluster. The silhouette width s_i of object i is given by:

$$s_i = \frac{b_i - a_i}{\max(a_i, b_i)}$$

where a_i is the average distance of object i to all other objects in the same cluster, and b_i is the smallest distance of object i to another cluster. Thus, the maximal value will be obtained in those cases where the intra-cluster distance a is much smaller than the inter-cluster distance b.

For the wine data clustering, the silhouette plot shown in Figure 6.3 is obtained by:

```
> plot(wines.pam, main = "Silhouette plot")
```

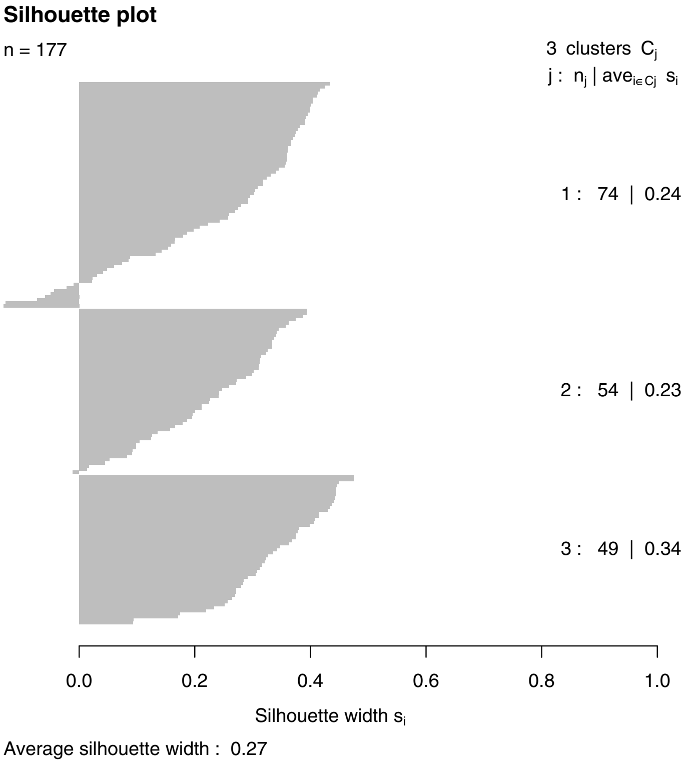

Fig. 6.3. Silhouette plot for the k-medoids clustering of the wine data. The three clusters contain 74, 54 and 49 objects, and have average silhouette widths of 0.24, 0.23 and 0.34, respectively.

An overall measure of clustering quality can be obtained by averaging all sihouette widths. This is an easy way to decide on the most appropriate number of clusters:

```
> best.pam <- pam(wines.dist, k = 2)
> for (i in 3:10) {
+   tmp.pam <- pam(wines.dist, k = i)
+   if (tmp.pam$silinfo$avg.width < best.pam$silinfo$avg.width)
+     best.pam <- tmp.pam
+ }
> best.pam$medoids
```

```
[1]  12  56  34  97  91 163 125 148
```

In this case, eight clusters seem to give the clustering with the least ambiguity. The agreement with the true class labels is quite good:

```
> table(vintages, best.pam$clustering)

vintages      1  2  3  4  5  6  7  8
  Barbera     0  0  0  0  0 18  0 30
  Barolo     21 17 20  0  0  0  0  0
  Grignolino  0  1  5 20 15  5 24  1
```

Clusters 1, 2 and 3 correspond to the Barolo wines, and clusters 6 and 8 to the Barbera. Again, the Grignolino wines are the most difficult to cluster, and 12 Grignolino samples end up in clusters dominated by other wines.

For large data sets, `pam` is too slow; in the **cluster** package, an alternative is provided in the function `clara` [48] which considers subsets of size `sampsize`. Each subset is partitioned using the same algorithm as in `pam`. The sets of medoids that result are used to cluster the complete data set, and the best set of medoids, i.e. the one for which the sum of the distances is minimal, is retained.

6.3 Probabilistic Clustering

In probabilistic clustering, sometimes also called fuzzy clustering, objects are not allocated to one cluster only. Rather, cluster memberships are used to indicate which of the clusters is more likely. If a "crisp" clustering result is needed, an object is assigned to the cluster with the highest membership value.

The most well-established methods are found in the area of mixture modelling, where individual clusters are represented by mixtures of parametric distributions, and the overall clustering is a weighted sum of the individual components [49,50]. Usually, multivariate normal distributions are applied. In that case, assuming G clusters, the likelihood is given by

$$L(\tau, \mu, \mathbf{\Sigma}|\mathbf{x}) = \prod_{i=1}^{n} \sum_{k=1}^{G} \tau_k \phi_k(\mathbf{x}_i|\mu_k, \mathbf{\Sigma}_k) \ ,$$

where τ_k is the fraction of objects in cluster k, μ_k and $\mathbf{\Sigma}_k$ correspond to the cluster means and covariance matrices of cluster k, respectively, and ϕ_k is the density of cluster k. If the cluster labels would be known, one could estimate the unknown parameters τ_k, μ_k and $\mathbf{\Sigma}_k$ by maximizing the likelihood (for example). Vice versa, when these parameters are known, it is easy to calculate the conditional probabilities of belonging to class k:

$$z_{ik} = \phi_k(\mathbf{x}_i|\theta_k) / \sum_{j=1}^{K} \phi_j(\mathbf{x}_i|\theta_j)$$

These two steps are the components in the Expectation-Maximization algorithm (EM) [51,52]: estimating the conditional probabilities is indicated with

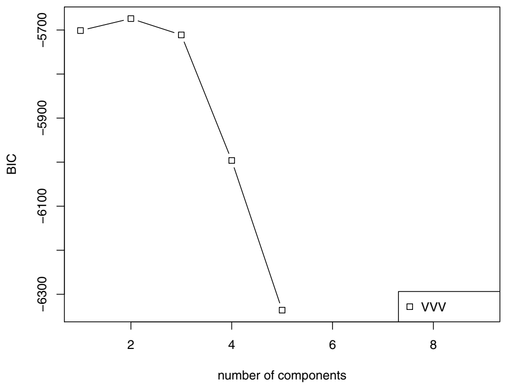

Fig. 6.4. BIC values for clustering the autoscaled wine data with **mclust**. The label "VVV" indicates a completely unconstrained model. The optimal model has two clusters.

the E-step, whereas estimating the parameters (class means and variances, and mixing proportions) is the M-step. The conditional probabilities z_{ik} can also be seen as indicators of uncertainty: the larger $z_{i,\max}$, the maximal value of all z_{ik} values for object i, the more certain the classification.

One can use the likelihood to determine what number of clusters is optimal. Of course, the likelihood will increase with the number of clusters, so one usually takes into account the number of parameters that are estimated in penalized versions. Two popular measures are Akaike's Information Criterion (AIC) and the Bayesian Information Criterion (BIC). The AIC criterion [53] is defined by

$$AIC = -2 \log L + 2p \tag{6.1}$$

where L is the likelihood and p the number of parameters in the model (here τ, μ, and $\mathbf{\Sigma}$). The closely related BIC criterion [54] uses a more heavy penalization:

$$BIC = -2 \log L + p \log n \tag{6.2}$$

The optimal model has a minimal value for AIC and/or BIC[2] – because of the more heavy penalty, BIC is likely to select slightly more parsimonious models than AIC. Several other criteria exist [49]. None of these is able to correctly identify the number of clusters in all cases, but in practice, differences are not very big and both AIC and BIC criteria are often used.

Several packages in R implement this form of clustering. In the **mclust** package [55], for example, one can calculate BIC values for different numbers of clusters easily:

[2] Especially for the BIC value, one often sees the negative form so that maximization will lead to an optimal model. This is also the definition by [54].

```
> wines.BIC <- mclustBIC(wines.sc, modelNames = "VVV")
> plot(wines.BIC)
```

This produces the plot in Figure 6.4. The BIC value, here given in its negative form, has a maximum at two clusters, which will be the model picked by the function `mclustModel` if no specific number of clusters is given. Alternatively, one can specifiy a specific number of clusters by providing a value for the `G` argument:

```
> wines.mclust2 <- mclustModel(wines.sc, wines.BIC)
> wines.mclust3 <- mclustModel(wines.sc, wines.BIC, G = 3)
```

One can make scatter plots at specific combinations of variables with the `coordProj` function, visualising the clustering in low-dimensional subspaces. Als the uncertainties, given by $1 - z_{i,\max}$, can be visualized. This provides an easy way to compare the two- and three-cluster solutions graphically:

```
> par(mfrow = c(2,2))
> coordProj(wines.sc, dimens = c(7, 13),
+           parameters = wines.mclust2$parameters,
+           z = wines.mclust2$z, what = "classification")
> title("2 clusters: classification")
> coordProj(wines.sc, dimens = c(7, 13),
+           parameters = wines.mclust3$parameters,
+           z = wines.mclust3$z, what = "classification")
> title("3 clusters: classification")
> coordProj(wines.sc, dimens = c(7, 13),
+           parameters = wines.mclust2$parameters,
+           z = wines.mclust2$z, what = "uncertainty")
> title("2 clusters: uncertainty")
> coordProj(wines.sc, dimens = c(7, 13),
+           parameters = wines.mclust3$parameters,
+           z = wines.mclust3$z, what = "uncertainty")
> title("3 clusters: uncertainty")
```

The result, here for the variables `flavonoids` and `proline`, is shown in Figure 6.5. The top row shows the classifications of the two- and three-cluster models, respectively. The bottom row shows the corresponding uncertainties.

Just like with k-means and k-medoids, the clustering using the EM algorithm needs to be kick-started with an initial guess. This may be a random initialization, but the EM algorithm has a reputation for being slow to converge, and an unlucky guess may lead into a local optimum. In `mclust`, the initialization is done by hierarchical clustering[3]. This has the advantage that initial models for many different numbers of clusters can be generated quickly.

[3] To be more precise, model-based hierarchical clustering [56].

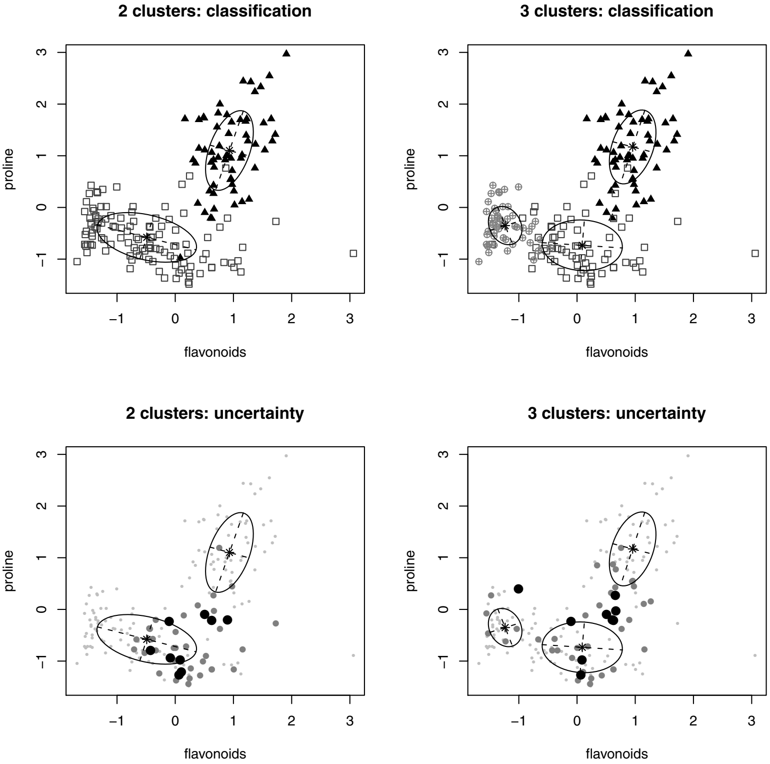

Fig. 6.5. Two- and three-cluster models for the `wines` data, obtained by **mclust**. The top row shows the classifications; the bottom row shows uncertainties at three levels, where the smallest dots have z-values over .95 and the largest, black, dots have z-values below .75. The others are in between.

Moreover, this initialization algorithm is stable in the sense that the same clustering is obtained upon repetition. A BIC table, such as the one depicted in Figure 6.4 is therefore easily obtained.

While mixtures of gaussians (or other distributions) have many attractive properties, they suffer from one big disadvantage: the number of parameters to estimate quickly becomes large. This is the reason why the BIC curve in Figure 6.4 does not run all the way to nine clusters, although that is the default in `mclust`: in high dimensions, clusters with only few members quickly lead to singular covariance matrices. In such cases, no BIC value is returned. Banfield and Raftery [57] suggested to impose restrictions on the covariance matrices of the clusters: one can, e.g., use spherical and equal-sized covariance matrices for all clusters. In this case, which is also the most restricted, the criterion that is optimized corresponds to the criterion used in k-means and in Ward's hierarchical clustering. For each cluster, Gp parameters need to be

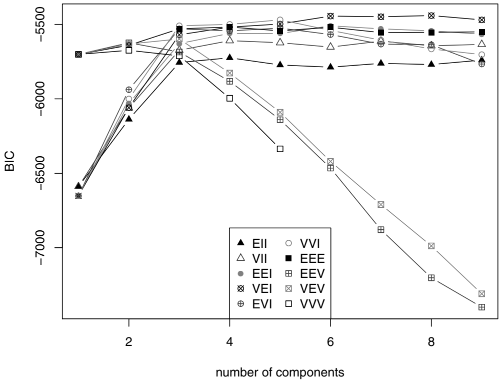

Fig. 6.6. BIC plots for all ten covariance models implemented in `mclust`: although the constrained models do not fit as well for the same numbers of clusters, they are penalized less and achieve higher BIC values for larger numbers of clusters.

estimated for the cluster centers, one parameter for the covariance matrices, and p mixing proportions, a total of $(G+1)p+1$. In contrast, for the completely free model such as the ones in Figures 6.4 and 6.5, indicated with “VVV” in `mclust`, every single covariance matrix requires $p(p+1)/2$ parameters. This leads to a grand total of $p(Gp+G+4)/2$ estimates. For low-dimensional data, this is still doable, but for higher-dimensional data the unrestricted models are no longer workable.

Consider the wine data again, but now consider all ten models implemented in `mclust`:

```
> wines.BIC <- mclustBIC(wines.sc)
> plot(wines.BIC, legendArgs = list(x = "bottom", ncol = 2))
```

This leads to the output in Figure 6.6. The three-letter codes in the legend stand for volume, shape and orientation, respectively. The “E” indicates equality for all clusters, the “V” indicates variability, and the “I” indicates identity. Thus, the “EEI” model stands for diagonal covariance matrices (the “I”) with equal volumes and shapes, and the “VEV” model indicates an ellipsoidal model with equal shapes for all clusters, but complete freedom in size and orientation. It is clear that the more constrained models achieve much higher BIC values for higher numbers of clusters: the unconstrained models are penalized more heavily for estimating so many parameters.

6.4 Comparing Clusterings

In many cases, one is interested in comparing the results of different clusterings. This may be to assess the behaviour of different methods on the same data set, but also to find out how variable the clusterings are that are obtained by randomly initialized methods like k-means. The difficulty here, of course, is that there is no golden standard; one cannot simply count the number of incorrect assignments and use that as a quality criterion. Moreover, the number of clusters may differ – still we may be interested in assessing the agreement between the partitions.

Several measures have been proposed in literature. Hubert and Arabie [58] compare several of these, and propose the adjusted Rand index, inspired by earlier work by Rand [59]. The original Rand index is based on the number of times two objects are classified in the same cluster, n. In the formulas below, $n_{i\cdot}$ indicates the number of object pairs classified in the same cluster in partition one, but not in partition two, $n_{\cdot j}$ the reverse, and n_{ij} the number of pairs classified in different clusters in both partitions. The index, comparing two partitions with I and J objects, respectively, is given by

$$R = \binom{n}{2} + 2\sum_{i=1}^{I}\sum_{j=1}^{J}\binom{n_{ij}}{2} - \left\{\sum_{i=1}^{I}\binom{n_{i\cdot}}{2} + \sum_{j=1}^{J}\binom{n_{\cdot j}}{2}\right\} \tag{6.3}$$

The adjusted Rand index "corrects for chance" by taking into account the expected value of the index under the null hypothesis of random partitions:

$$R_{\text{adj}} = \frac{R - E(R)}{\max(R) - E(R)} = \frac{a\binom{n}{2} - bc}{\frac{1}{2}\binom{n}{2}(b+c) - bc} \tag{6.4}$$

with

$$a = \sum_{i,j}\binom{n_{ij}}{2} \tag{6.5}$$

$$b = \sum_{i}\binom{n_{i\cdot}}{2} \tag{6.6}$$

$$c = \sum_{j}\binom{n_{\cdot j}}{2} \tag{6.7}$$

This measure is zero when the Rand index takes its expected value, and has a maximum of one.

The implementation in R takes only a few lines:

```
> AdjRkl <- function(part1, part2)
+ {
+   confusion <- table(part1, part2)
+
+   n <- sum(confusion)
+   a <- sum(choose(confusion[confusion>1], 2))
+   b <- apply(confusion, 1, sum)
+   b <- sum(choose(b[b>1], 2))
+   c <- apply(confusion, 2, sum)
+   c <- sum(choose(c[c>1], 2))
+
+   Rexp <- b*c/choose(n,2)
+   (a - Rexp) / (.5*(b+c) - Rexp )
+ }
```

The function takes two partitionings, i.e., class vectors, and returns the value of the adjusted Rand index. Note that the number of classes in both partitionings need not be the same. An alternative is function `adjustedRandIndex` in package **mclust**.

How this can be useful is easily illustrated. As already stated, repeated application of SOM mapping will, in general, lead to mappings that visually can appear very different. However, objects may find themselves very close to the same neighbours in repeated training runs, so that conclusions from the two maps will be very much the same. One way to investigate that is to quantify the similarities. Consider the SOM mapping of the wine data for two initializations:

```
> X <- scale(wines)
> set.seed(7)
> som.wines <- som(X, grid = somgrid(6, 4, "hexagonal"))
> set.seed(17)
> som.wines2 <- som(X, grid = somgrid(6, 4, "hexagonal"))
```

Assessing the similarities of the maps should not be done on the level of the individual units, since these are not relevant entities in themselves. Rather, the units should be aggregated into larger clusters. This can be achieved by looking at plots like Figure 5.5; an alternative is to explicitly cluster the codebook vectors (see, e.g., [60]). If hierarchical clustering is used, the dendrograms can be cut at the desired level, immediately providing cluster memberships for the individual samples.

```
> som.hc <- cutree(hclust(dist(som.wines$codes)), k = 3)
> som.hc2 <- cutree(hclust(dist(som.wines2$codes)), k = 3)
> plot(som.wines, "mapping", bgcol = terrain.colors(3)[som.hc],
+      pch = as.integer(vintages), main = "Seed 7")
> plot(som.wines2, "mapping", bgcol = terrain.colors(3)[som.hc2],
+      pch = as.integer(vintages), main = "Seed 17")
```

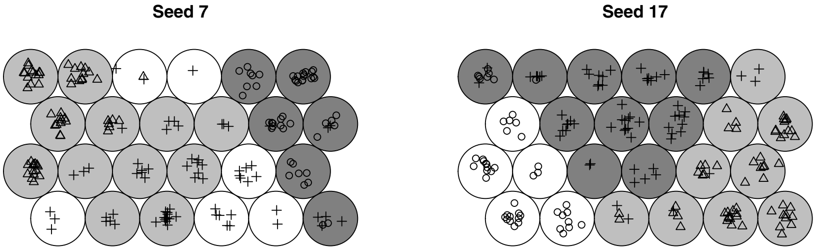

Fig. 6.7. Clustering of the codebook vectors of two mappings of the wine data, indicated by background colors. Symbols indicate vintages.

This leads to the plots in Figure 6.7. The mappings seem very different. Is this really the case, or is it just a visual artefact? Let's find out:

```
> som.clust <- som.hc[som.wines$unit.classif]
> som.clust2 <- som.hc2[som.wines2$unit.classif]
> AdjRkl(som.clust, som.clust2)
```

```
[1] 0.3312089
```

This rather low value suggests that both mappings are quite different. Note that this analysis does not take into account the vintages and is applicable also in cases where "true" class labels are unknown. Of course, one can also use the adjusted Rand index to compare clusterings with a set of "true" labels:

```
> AdjRkl(som.clust, som.clust)
```

```
[1] 0.4321626
```

```
> AdjRkl(som.clust, som.clust2)
```

```
[1] 0.7598247
```

Clearly, the second random seed gives a mapping that is more in agreement with the class labels, something that is also clear when looking at the agreement between plotting symbols and background color in Figure 6.7.

Other indices to measure correspondence between two partitionings include Fowlkes' and Mallows' B_k [61], Goodmans and Kruskals γ [62], and Meila's Variation of Information criterion [63], also available in **mclust**. The latter is a difference measure, rather than a similarity measure.

6.5 Discussion

Hierarchical clustering methods have many attractive features. They are suitable in cases where there is a hierarchical structure, i.e., subclusters, which very often is the case. A large number of variables does not pose a problem: the rate-limiting step is the calculation of the distance matrix, the size of which does not depend on the dimensionality of the data, but only on the number of samples. And last but not least, the dendrogram provides an appealing presentation of the cluster structure, which can be used to assess clusterings with different numbers of clusters very quickly. Partitional methods, on the other hand, are more general. In hierarchical clustering a split cannot be undone – once a sample is in one branch of the tree, there is no way it can move to the other branch. This can lead, in some cases, to suboptimal clusterings. Partitional methods do not know such restrictions: a sample can always be classified into a different class in the next iteration. Some of the less complicated partitional methods, such as k-means clustering, can also be applied with huge data sets, containing tens of thousands of samples, that cannot be tackled with hierarchical clustering.

Both types of clustering have their share of difficulties, too. In cases relying on distance calculations (all hierarchical methods, and some of the partitional methods, too), the choice of a distance function can dramatically influence the result. The importance of this cannot be overstated. On the one hand, this is good, since it allows one to choose the most relevant distance function available – it even allows one to tackle data that do not consist of real numbers but are binary or have a more complex nature. As long as there is a distance function that adequately represents dissimilarities between objects, the regular clustering methods can be applied. On the other hand, it is bad: it opens up the possibility of a wrong choice. Furthermore, one should realize that when correlated groups of variables are present, as often is the case in life science data, these variables may receive a disproportionally large weight in a regular distance measure such as Euclidean distance, and smaller groups of variables, or uncorrelated variables, may fail to be recognized as important.

Partitional methods force one to decide on the number of clusters beforehand, or perform multiple clusterings with different numbers of clusters. Moreover, there can be considerable differences upon repeated clustering, something that is less prominent in hierarchical clustering (only with ties in the distance data). The main problem with hierarchical clustering is that the bottom-up joining procedure may be too strict: once an object is placed in a certain category, it will stay there, whatever happens further on in the algorithm. Of course, there are many examples where this leads to a sub-optimal clustering. More generally, there may not be a hierarchical structure to begin with.

Both partitional and hierarchical clustering yield “crisp” clusters, that is, objects are assigned to exactly one cluster, without any doubt. For partitional methods, there are alternatives where each object gets a membership value for each of the clusters. If a crisp clustering is required, at the end of the algorithm the object is assigned to the cluster for which it has the highest membership. We have seen one example in the model-based clustering methods.

Finally, one should take care not to over-interpret the results. If you ask for five clusters, that is exactly what you get. Suppose one has a bananashaped cluster. Methods like k-means, but also complete linkage, will typically describe such a banana with three or four spherical clusters. The question is: are you interested in the peas or the pod[4]? It may very well be that several clusters in fact describe one and the same group, and that to find the other clusters one should actually instruct the clustering to look for more than five clusters.

Clustering is, because of the lack of "hard" criteria, more of an art than a science. Without additional knowledge about the data or the problem, it is hard to decide which one of several different clusterings is best. This, unfortunately, in some areas has led to a practice in which all available clustering routines are applied, and the one that seems most "logical" is selected and considered to describe "reality". One should always keep in mind that this may be a gross overestimation of the powers of clustering.

[4] Metaphor from Adrian Raftery.

Part III

Modelling

7

Classification

The goal of classification, also known as supervised pattern recognition, is to provide a model that yields the optimal discrimination between several classes in terms of predictive performance. It is closely related to clustering. The difference is that in classification it is clear what to look for: the number of classes is known, and the classes themselves are well-defined, usually by means of a set of examples, the training set. Labels of objects in the training set are generally taken to be error-free, and are typically obtained from information other than the data we are going to use in the model. For instance, one may have data – say, concentration levels of several hundreds of proteins in blood – from two groups of people, healthy, and not-so-healthy, and the aim is to obtain a classification model that distinguishes between the two states on the basis of the protein levels. The diagnosis may have been based on symptoms, medical tests, family history and subjective reasoning of the doctor treating the patient. It may not be possible to distinguish patients from healthy controls on the basis of protein levels, but if one would be able to, it would lead to a simple and objective test.

Apart from having good predictive abilities, an ideal classification method also provides insight in what distinguishes different classes from each other. Especially in the natural sciences, this has become an important objective: a gene, protein or metabolite, characteristic for one or several classes, is often called a *biomarker*. Such a biomarker, or more often, set of biomarkers, can be used as an easy and reliable diagnostic tool, but also can provide insight in the underlying biological processes. Unfortunately, in most cases biomarker identification is difficult because the number of variables far exceeds the number of cases, so that there is a real risk of false positives because of chance correlations.

What is needed as well is a reliable estimate of the success rate of the classifier. In particular, one would like to know how the classifier will perform in the future, on new samples, of course comparable to the ones used in setting up the model. This error estimate is obtained in a validation step – Chapter 9 provides an overview of several different methods. These are all the

R. Wehrens, *Chemometrics with R: Multivariate Data Analysis in the Natural Sciences and Life Sciences*, Use R!, DOI 10.1007/978-3-642-17841-2_7,

more important when the classifier of interest has tuneable parameters. These parameters are usually optimized on the basis of estimated prediction errors, but as a result the error estimates are positively biased, and a second validation layer is needed to obtain an unbiased error estimate. In this chapter, we will take a simple approach and divide the data in a representative part that is used for building the model (the training set), and an independent part used for testing (the test set). The phrase "independent" is of utmost importance: if, e.g., autoscaling is applied, one should use the column means and standard deviations of the training set to scale the test set. First scaling the complete data set and then dividing the data in training and test sets is, in a way, cheating: one has used information from the test data in the scaling. This usually leads to underestimates of prediction error.

That the training data should be representative seems almost trivial, but in some cases this is hard to achieve. Usually, a random division works well, but also other divisions may be used. In Chapter 4 we have seen that the odd rows of the `wine` data set are very similar to the even rows: in a classification context, we can therefore use the even rows as a training set and the odd rows as a test set:

```
> wines.trn <- wines[odd,]
> wines.tst <- wines[even,]
```

Note that classes are represented proportional to their frequency in the original data in both the training set and the test set.

There are many different ways of using the training data to predict class labels for future data. Discriminant analysis methods use a parametric description of means and covariances. Essentially, observations are assigned to the class having the highest probability density. Nearest-neighbour methods, on the other hand, focus on similarities with individual objects and assign objects to the class that is prevalent in the neighbourhood; another way to look at it is to see nearest-neighbour methods as local density estimators. Similarities between objects can also be used directly, e.g., in kernel methods; the most well-known representative of this type of methods is Support Vector Machines (SVMs). A completely different category of classifiers is formed by tree-based approaches. These create a model consisting of a series of binary decisions. Finally, neural-network based classification will be discussed.

7.1 Discriminant Analysis

In discriminant analysis, one assumes normal distributions for the individual classes: $N_p(\mu_k, \Sigma_k)$, where the subscript p indicates that the data are p-dimensional [64]. One can then classify a new object, which can be seen as a point in p-dimensional space, to the class that has the highest probability density ("likelihood") at that point – this type of discriminant analysis is therefore indicated with the term “Maximum-Likelihood” (ML) discriminant analysis.

Consider the following univariate example with two groups [26]: group one is $N(0,5)$ and group 2 is $N(1,1)$. The likelihoods of classes i are given by

$$L_i\left(x; \mu_i, \sigma_i\right) = \frac{1}{\sigma_i \sqrt{2\pi}} \exp\left[-\frac{\left(x-\mu_i\right)^2}{2\sigma_i^2}\right] \tag{7.1}$$

It is not too difficult to show that $L_1 > L_2$ if

$$\frac{12}{25}x^2 - x + 1/2 - \ln 5 > 0$$

which in this case corresponds to the regions outside the interval $[-0.9, 2.9]$. In more general terms, one can show [26] that for one-dimensional data $L_1 > L_2$ when

$$x^2\left(\frac{1}{\sigma_1^2} - \frac{1}{\sigma_2^2}\right) - 2x\left(\frac{\mu_1}{\sigma_1^2} - \frac{\mu_2}{\sigma_2^2}\right) + \left(\frac{\mu_1^2}{\sigma_1^2} - \frac{\mu_2^2}{\sigma_2^2}\right) < 2\ln\frac{\sigma_2}{\sigma_1} \tag{7.2}$$

This unrestricted form, where every class is individually described with a mean vector and covariance matrix, leads to quadratic class boundaries, and is called “Quadratic Discriminant Analysis” (QDA). Obviously, when $\sigma_1 = \sigma_2$ the quadratic term disappears, and we are left with a linear class boundary – “Linear Discriminant Analysis” (LDA). Both techniques will be treated in more detail below.

Another way of describing the same classification rules is to make use of the Mahalanobis distance:

$$d(\boldsymbol{x}, i) = \left(\boldsymbol{x} - \boldsymbol{\mu}_i\right)^T \boldsymbol{\Sigma}_i^{-1}\left(\boldsymbol{x} - \boldsymbol{\mu}_i\right) \tag{7.3}$$

Loosely speaking, this expresses the distance of an object to a class center in terms of the standard deviation in that particular direction. Thus, a sample $\boldsymbol{x}$ is simply assigned to the closest class, using the Mahalanobis metric $d(\boldsymbol{x}, i)$. In LDA, all classes are assumed to have the same covariance matrix $\boldsymbol{\Sigma}$, whereas in QDA every class is represented by its own covariance matrix $\boldsymbol{\Sigma}_i$.

7.1.1 Linear Discriminant Analysis

It is easy to show that Equation 7.2 in the case of two groups with equal variances reduces to

$$|\boldsymbol{x} - \boldsymbol{\mu}_2| > |\boldsymbol{x} - \boldsymbol{\mu}_1| \tag{7.4}$$

Each observation $\boldsymbol{x}$ will be assigned to class 1 when it is closer to the mean of class 1 than of class 2, something that makes sense intuitively as well. Another way to write this is

$$\boldsymbol{\alpha}^T\left(\boldsymbol{x} - \boldsymbol{\mu}\right) > 0 \tag{7.5}$$

with

$$\boldsymbol{\alpha} = \boldsymbol{\Sigma}^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2) \tag{7.6}$$

$$\boldsymbol{\mu} = (\boldsymbol{\mu}_1 + \boldsymbol{\mu}_2)/2 \tag{7.7}$$

This formulation clearly shows the linearity of the class boundaries. The separating hyperplane passes through the midpoint between the cluster centers, but is not necessarily perpendicular to the segment connecting the two centers.

In reality, of course, one does not know the true means $\boldsymbol{\mu}_i$ and the true covariance matrix $\boldsymbol{\Sigma}$. One then uses the plugin estimate $\boldsymbol{S}$, the sample covariance matrix, estimated from the data. In LDA, it is obtained by pooling the individual sample covariance matrices $\boldsymbol{S}_i$:

$$\boldsymbol{S} = \frac{1}{n-G} \sum_{i=1}^{G} n_i \boldsymbol{S}_i \tag{7.8}$$

where there are G groups, n_i is the number of objects in group i, and the total number of objects is n.

For the wine data, this can be achieved as follows:

```
> wines.counts <- table(vintages[odd])
> ngroups <- length(wines.counts)
> wines.groups <- split(as.data.frame(wines.trn),
+                       vintages[odd])
> wines.covmats <- lapply(wines.groups, cov)
> wines.wcovmats <- lapply(1:ngroups,
+                          function(i, x, y) x[[i]]*y[i],
+                          wines.covmats, wines.counts)
> wines.pooledcov <- Reduce("+", wines.wcovmats) /
+   (nrow(wines.trn) - ngroups)
```

This piece of code illustrates a convenient feature of the `lapply` function: when the first argument is a vector, it can be used as an index for a function taking also other arguments – here, a list and a vector. Each of the three covariance matrices is multiplied by a weight corresponding to the number of objects in that class. In the final step, the `Reduce` function adds the three weighted covariance matrices. An alternative is to use a plain and simple loop:

```
> wines.pooledcov2 <- matrix(0, ncol(wines), ncol(wines))
> for (i in 1:3) {
+   wines.pooledcov2 <- wines.pooledcov2 +
+     cov(wines.groups[[i]]) * nrow(wines.groups[[i]])
+ }
> wines.pooledcov2 <-
+   wines.pooledcov2 / (nrow(wines.trn) - ngroups)
```

The number of parameters that must be estimated in LDA is relatively small: the pooled covariance matrix contains $p(p+1)/2$ numbers, and each cluster center p parameters. For G groups this leads to a total of $Gp + p(p+1)/2$

estimates – for the wine data, with three groups and thirteen variables, this implies 130 estimates.

The LDA classification itself is now easily performed: first we calculate the Mahalanobis distances (using the `mahalanobis` function) to the three class centers using the pooled covariance matrix, and then we determine which of these three is closest for every sample in the training set:

```
> distances <-
+   sapply(1:ngroups,
+          function(i, samples, means, covs)
+            mahalanobis(samples, colMeans(means[[i]]), covs),
+          wines.trn, wines.groups, wines.pooledcov)
> trn.pred <- apply(distances, 1, which.min)
```

Let's see how good the predictions are:

```
> table(vintages[odd], trn.pred)
```

```
             trn.pred
               1  2  3
  Barbera     24  0  0
  Barolo       0 29  0
  Grignolino   0  0 36
```

The reproduction of the training data is perfect, much better than we have seen with clustering, which is not surprising since the LDA builds the model (in this case the pooled covariance matrix) using the information from the training set with the explicit aim of discriminating between the classes. However, we should not think that future observations are predicted with equal success. The test data should give an indication of what to expect:

```
> distances <-
+   sapply(1:ngroups,
+          function(i, samples, means, covs)
+            mahalanobis(samples, colMeans(means[[i]]), covs),
+          wines.tst, wines.groups, wines.pooledcov)
> tst.pred <- apply(distances, 1, which.min)
> table(vintages[even], tst.pred)
```

```
            tst.pred
               1  2  3
  Barbera     24  0  0
  Barolo       0 29  0
  Grignolino   1  0 34
```

One Grignolino sample has been classified as a Barbera – a very good result, confirming that the problem is not very difficult. Nevertheless, the difference with the unsupervised clustering approaches is obvious.

Of course, R already contains an `lda` function (in package **MASS**):

```
> wines.ldamod <- lda(wines.trn, grouping = vintages[odd],
+                     prior = rep(1,3)/3)
> wines.lda.testpred <- predict(wines.ldamod, new = wines.tst)
> table(vintages[even], wines.lda.testpred$class)

             Barbera Barolo Grignolino
  Barbera         24      0          0
  Barolo           0     29          0
  Grignolino       1      0         34
```

The `prior = rep(1,3)/3` argument in the `lda` function is used to indicate that all three classes are equally likely *a priori*. In many cases it makes sense to incorporate information about prior probabilities. Some classes may be more common than others, for example. This is usually reflected in the class sizes in the training set and therefore is taken into account when calculating the pooled covariance matrix, but it is not explicitly used in the discrimination rule. However, it is relatively simple to do so: instead of maximising L_i one now maximises $\pi_i L_i$, where π_i is an estimate of the prior probability of class i. In the two-group case, this has the effect of shifting the critical value of the discriminant function with an amount of $\log(\pi_2/\pi_1)$ in Equation 7.5. This approach is sometimes referred to as the Bayesian discriminant rule, and is the default behaviour of the `lda` function. Obviously, when all prior probabilities are equal, the Bayesian and ML discriminant rules coincide. Also in the example above, using the relative frequencies as prior probabilities would not have made any difference to the predictions – the three vintages have approximately equal class sizes.

The `lda` function comes with the usual supporting functions for printing and plotting. An example of what the plotting function provides is shown in Figure 7.1:

```
> plot(wines.ldamod, xlim = c(-7, 6))
```

The `xlim` argument provides some extra horizontal space to show the labels. The training samples are projected in the space of two new variables, the Linear Discriminants (LDs). In comparison to the PCA scoreplot from Figure 4.1, class separation has clearly increased. Again, this is the result of the way in which the LDs have been chosen: whereas the PCs in PCA account for as much variance as possible, in LDA the LDs maximize separation. This will be even more clear when we view LDA in the formulation by Fisher (see Section 7.1.3).

One particularly attractive feature of the `lda` function as it is implemented in **MASS** is the possibility to choose different estimators of means and covariances. In particular, the arguments `method = "mve"` and `method = "t"` are interesting as they provide robust estimates.

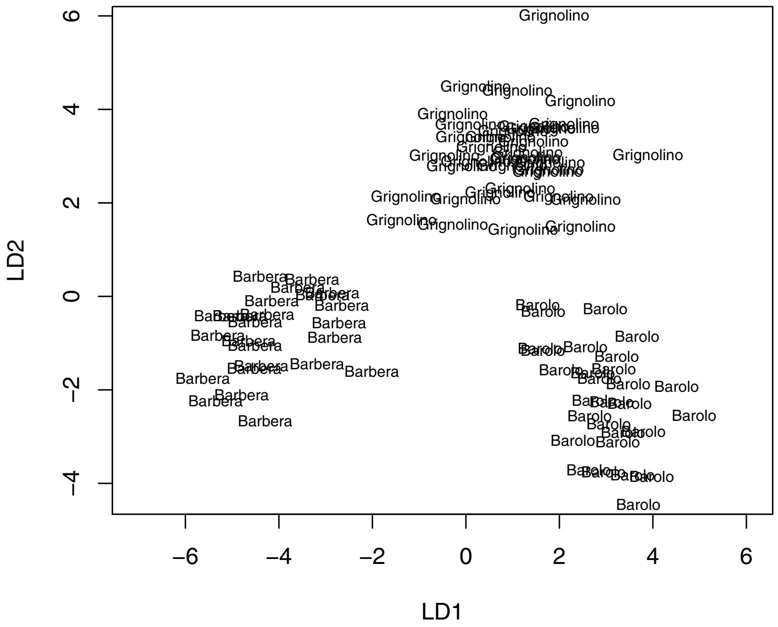

Fig. 7.1. Projection of the training data from the wine data set in the linear discriminant space. It is easy to see that linear class boundaries can be drawn so that all training objects are classified correctly.

7.1.2 Crossvalidation

If the number of samples is low, there are two important disadvantages of dividing a data set into two parts, one for training and one for testing. The first is that with small test sets, the error estimates are on a very rough scale: if there are ten samples in the test set, the errors are always multiples of ten percent. Secondly, the quality of the model will be lower than it can be: when building the classification model one needs all information one can get, and leaving out a significant portion of the data in general is not helpful. Only with large sets, consisting of, say, tens or hundreds of objects per class, it is possible to create training and test sets in such a way that modelling power will suffer very little while giving a reasonably precise error estimate. Even then, there is another argument against a division into training and test sets: such a division is random, and different divisions will lead to different estimates of prediction error. The differences may not be large, but in some cases they

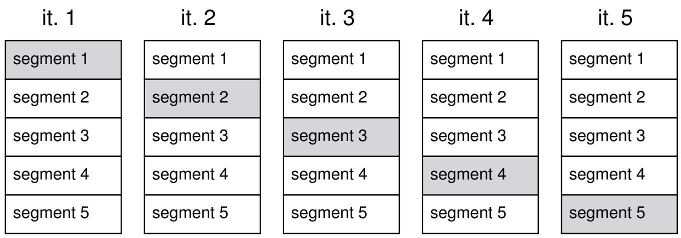

Fig. 7.2. Illustration of crossvalidation; in the first iteration, segment 1 of the data is left out during training and used as a test set. Every segment in turn is left out. From the prediction errors of the left-out samples the overall crossvalidated error estimate is obtained.

can be important, especially in the case of outliers and/or extremely unlucky divisions.

One solution would be to try a large number of random divisions and to average the resulting estimates. This is indeed a valid strategy – we will come back to this in Chapter 9. A very popular formalization of this principle is called *crossvalidation* [65]. The general procedure is as follows: one leaves out a certain part of the data, trains the classifier on the remainder, and uses the left-out bit – sometimes called the out-of-bag, or OOB, samples – to estimate the error. Next, the two data sets are joined again, and a new test set is split off. This continues until all objects have been left out exactly once. The crossvalidation error in classification is simply the number of misclassified objects divided by the total number of objects in the training set. If the size of the test set equals one, every sample is left out in turn – the procedure has received the name Leave-One-Out (LOO) crossvalidation. It is shown to be unbiased but can have appreciable variance: on average, the estimate is correct, but individual components may deviate considerably.

More stable results are usually obtained by leaving out a larger fraction, e.g. 10% of the data; such a crossvalidation is known as ten-fold crossvalidation. The largest errors cancel out (to some extent) so that the variance decreases; however, one pays the price of a small bias because of the size difference of the real training set and the training set used in the crossvalidation [66]. In general, the pros outweigh the cons, so that this procedure is quite often applied. It also leads to significant speed improvements for larger data sets, although for the simple techniques presented in this chapter it is not likely to be very important. The whole crossvalidation procedure is illustrated in Figure 7.2.

For LDA (and also QDA), it is possible to obtain the LOO crossvalidation result without complete refitting – upon leaving out one object, one can update the Mahalanobis distances of objects to class means and derive the classifications of the left-out samples quickly, without doing expensive matrix operations [34]. The `lda` function returns crossvalidated predictions in the list element `class` when given the argument `CV = TRUE`:

```
> wines.ldamod <- lda(wines.trn, grouping = vintages[odd],
+                     prior = rep(1,3)/3, CV = TRUE)
> table(vintages[odd],wines.ldamod$class)

             Barbera Barolo Grignolino
  Barbera         24      0          0
  Barolo           0     28          1
  Grignolino       1      0         35
```

So, where the training set can be predicted without any errors, LOO cross-validation on the training set leads to an estimated error percentage of $2/89 = 2.25\%$, twice the error on the test set. This difference in itself is not very alarming – error estimates also have variance.

7.1.3 Fisher LDA

A seemingly different approach to discriminant analysis is taken in Fisher LDA, named after its inventor, Sir Ronald Aylmer Fisher. Rather than assuming a particular distribution for individual clusters, Fisher devised a way to find a sensible rule to discriminate between classes by looking for a linear combination of variables $\boldsymbol{a}$ maximising the ratio of the between-groups sums of squares $\boldsymbol{B}$ and the within-groups sums of squares $\boldsymbol{W}$ [67]:

$$\boldsymbol{a}^T \boldsymbol{B}\boldsymbol{a} / \boldsymbol{a}^T \boldsymbol{W}\boldsymbol{a} \tag{7.9}$$

These sums of squares are calculated by

$$\boldsymbol{W} = \sum_{i=1}^{G} \tilde{\boldsymbol{X}}_i^T \tilde{\boldsymbol{X}}_i \tag{7.10}$$

$$\boldsymbol{B} = \sum_{i=1}^{G} n_i (\bar{\boldsymbol{x}}_i - \bar{\boldsymbol{x}})(\bar{\boldsymbol{x}}_i - \bar{\boldsymbol{x}})^T \tag{7.11}$$

where $\tilde{\boldsymbol{X}}_i$ is the mean-centered part of the data matrix containing objects of class i, and $\bar{\boldsymbol{x}}_i$ and $\bar{\boldsymbol{x}}$ are the mean vectors for class i and the whole data matrix, respectively. Put differently: $\boldsymbol{W}$ is the variation *around* the class centers, and $\boldsymbol{B}$ is the variation of the class centers around the global mean. It also holds that the total variance $\boldsymbol{T}$ is the sum of the betweens and within-groups variances:

$$\boldsymbol{T} = \boldsymbol{B} + \boldsymbol{W} \tag{7.12}$$

Fisher's criterion is equivalent to finding a linear combination of variables $\boldsymbol{a}$ corresponding to the subspace in which distances between classes are large and distances within classes are small – compact classes with a large separation. It can be shown that maximizing Equation 7.9 leads to an eigenvalue problem, and that the solution $\boldsymbol{a}$ is given by the eigenvector of $\boldsymbol{B}\boldsymbol{W}^{-1}$ corresponding with the largest eigenvalue. An object $\boldsymbol{x}$ is then assigned to the closest class, i, which means that for all classes $i \neq j$ the following inequality holds:

$$|\boldsymbol{a}^T\boldsymbol{x} - \boldsymbol{a}^T\bar{\boldsymbol{x}}_i| < |\boldsymbol{a}^T\boldsymbol{x} - \boldsymbol{a}^T\bar{\boldsymbol{x}}_j| \tag{7.13}$$

Interestingly, although Fisher took a completely different starting point and did not explicitly assume normality or equal covariances, in the two-group case Fisher LDA leads to exactly the same solution as ML-LDA. Consider the discrimination between Barbera and Grignolino wines. To enable easy visualization, we will restrict ourselves to only two variables, `flavonoids` and `proline`. Fisher LDA is performed by the following code:

```
> X <- wines[vintages != "Barolo", c(7, 13)]
> vint <- factor(vintages[vintages != "Barolo"])
>
> wines.counts <- table(vint)
> wines.groups <- split(as.data.frame(X), vint)
> WSS <-
+   Reduce("+", lapply(wines.groups,
+                      function(x) {
+                        crossprod(scale(x, scale = FALSE))}))
> BSS <-
+   Reduce("+", lapply(wines.groups,
+                      function(x, y) {
+                        nrow(x) * tcrossprod(colMeans(x) - y)},
+                      colMeans(X)))
> FLDA <- eigen(solve(WSS, BSS))$vectors[,1]
> FLDA / FLDA[1]
```

```
[1]  1.000000 -0.000876
```

Application of ML-LDA, Equation 7.5, leads to

```
> wines.covmats <- lapply(wines.groups, cov)
> wines.wcovmats <- lapply(1:length(wines.groups),
+                          function(i, x, y) x[[i]]*y[i],
+                          wines.covmats, wines.counts)
> wines.pcov12 <- Reduce("+", wines.wcovmats) / (length(vint) - 2)
> MLLDA <-
+   solve(wines.pcov12,
+         apply(sapply(wines.groups, colMeans), 1, diff))
> MLLDA / MLLDA[1]
```

```
flavonoids    proline
  1.000000  -0.000875
```

Setting the first element of the discrimination functions equal to 1 makes the comparison easier: the vector $\boldsymbol{a}$ in Equation 7.9 can be rescaled without

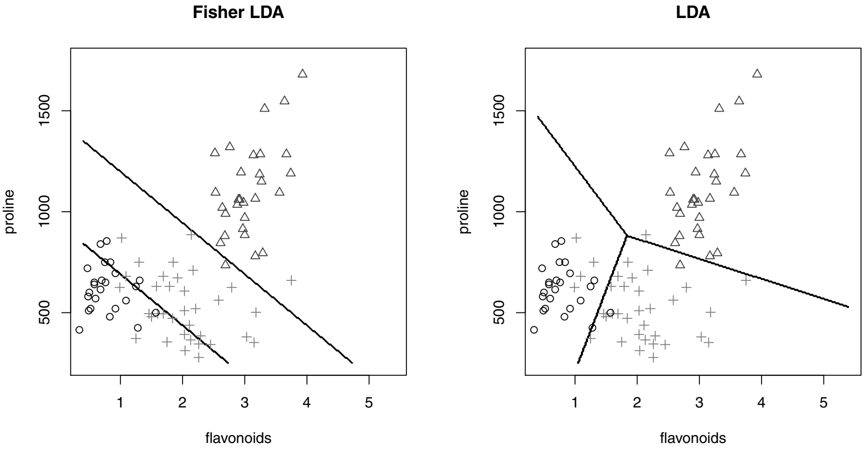

Fig. 7.3. Class boundaries for the wine data (proline and flavonoids only) for Fisher LDA (left) and ML-LDA (right). Models are created using the odd rows of the wine data; the plotting symbols indicate the even rows (test data), as mentioned in the text.

any effect on both allocation rules 7.13 and 7.5. In the two-group case, both ML-LDA and Fisher-LDA lead to the same discrimination function.

For problems with more than two groups, the results are different unless the sample means are collinear: Fisher LDA aims at finding *one* direction discriminating between the classes. An example is shown in Figure 7.3, where the boundaries between the three classes in the two-dimensional subset of the wine data are shown for Fisher LDA and ML-LDA.

The Fisher LDA boundaries are produced by code very similar to the two-group case on page 112: one should replace the line

```
> X <- wines[vintages != "Barolo", c(7, 13)]
```

by

```
> X <- wines.sc[odd, c(7, 13)]
```

and use `vintages[odd]` rather than `vint`. Then, after calculating the discriminant function `FLDA`, predictions are made at positions in a regular grid:

```
> scores <- gridXY %*% FLDA
> meanscores <- t(sapply(wines.groups, colMeans)) %*% FLDA
> Fdistance <- outer(scores, meanscores,
+                    FUN = function(x, y) abs(x - y))
> Fclassif <- apply(Fdistance, 1, which.min)
```

The distances of the scores of all gridpoints to the scores of the class means are calculated using the `outer` function – this leads to a three-column matrix.

The classification, corresponding to the class with the smallest distance, is obtained using the function which.min.

Finally, the class boundaries are visualized using the function contour; the points of the test set are added afterwards.

```
> contour(x, y,
+         matrix(Fclassif, nrow = length(x), byrow = TRUE),
+         main = "Fisher LDA", drawlabels = FALSE,
+         xlab = "flavonoids", ylab = "proline")
> points(wines.tst[, c(7, 13)],
+        col = as.integer(vintages[even]),
+        pch = as.integer(vintages[even]))
```

The result is shown in the left plot of Figure 7.3. The right plot is produced analogously:

```
> x <- seq(.4, 5.4, length = 251)
> y <- seq(250, 1750, length = 251)
> gridXY <- cbind(rep(x, each = length(y)), rep(y, length(x)))
> wines.ldamod <- lda(wines.trn[,c(7,13)],
+                     grouping = vintages[odd],
+                     prior = rep(1,3)/3)
> lda.2Dclassif <- predict(wines.ldamod, newdata = gridXY)$class
> contour(x, y,
+         matrix(as.integer(lda.2Dclassif),
+                nrow = length(x), byrow = TRUE),
+         main = "LDA", drawlabels = FALSE,
+         xlab = "flavonoids", ylab = "proline")
> points(wines.tst[,c(7,13)], col = as.integer(vintages[even]),
+        pch = as.integer(vintages[even]))
```

Immediately it is obvious that although the error rates of the two classifications are quite similar for the test set, large differences will occurr when data points are further away from the class centers. The class means are reasonably close to a straight line, so that Fisher LDA does not fail completely; however, for multi-class problems it is not a good idea to impose parallel class boundaries, as is done by Fisher LDA using only one eigenvector. It is better to utilise the information in the second and higher eigenvectors of $\boldsymbol{W}^{-1}\boldsymbol{B}$ as well [26]; these are sometimes called *canonical variates*, and the corresponding form of discriminant analysis is known as *canonical discriminant analysis*. The maximum number of canonical variates that can be extracted is one less than the number of groups.

7.1.4 Quadratic Discriminant Analysis

Quadratic discriminant analysis (QDA) takes the same route as LDA, with the important distinction that every class is described by its own covariance matrix, rather than one identical (pooled) covariance matrix for all classes. Given our exposé on LDA, the algorithm for QDA is pretty simple: one calculates the Mahalanobis distances of all points to the class centers, and assigns each point to the closest class. Let us see what this looks like in two dimensions:

```
> wines.trn <- wines[odd, c(7, 13)]
> wines.tst <- wines[even,  c(7, 13)]
> wines.groups <- split(as.data.frame(wines.trn), vintages[odd])
> wines.covmats <- lapply(wines.groups, cov)
> ngroups <- length(wines.groups)
> distances <- sapply(1:ngroups,
+                     function(i, samples, means, covs) {
+                       mahalanobis(samples,
+                                   colMeans(means[[i]]),
+                                   covs[[i]]) },
+                     wines.tst, wines.groups, wines.covmats)
> test.pred <- apply(distances, 1, which.min)
> table(vintages[even], test.pred)
```

```
            test.pred
              1  2  3
  Barbera    19  0  5
  Barolo      0 28  1
  Grignolino  3  1 31
```

Ten samples are misclassified in the test set. To see the class boundaries in two-dimensional space, we use the same visualization as seen in the previous section:

```
> qda.mahal.dists <- sapply(1:ngroups,
+                           function(i, samples, means, covs) {
+                             mahalanobis(samples,
+                                         colMeans(means[[i]]),
+                                         covs[[i]]) },
+                           gridXY, wines.groups, wines.covmats)
> qda.2Dclassif <- apply(qda.mahal.dists, 1, which.min)
> contour(x, y,
+         matrix(qda.2Dclassif, nrow = length(x), byrow = TRUE),
+         main = "QDA", drawlabels = FALSE,
+         xlab = "flavonoids", ylab = "proline")
> points(wines.tst, col = as.integer(vintages[even]),
+        pch = as.integer(vintages[even]))
```

The result is shown in the left plot of Figure 7.4. The quadratic form of the class boundaries is clearly visible. Again, only the test set objects are shown.

Using the `qda` function from the **MASS** package, modelling the odd rows and predicting the even rows is done just like with `lda`. Let's build a model using all thirteen variables:

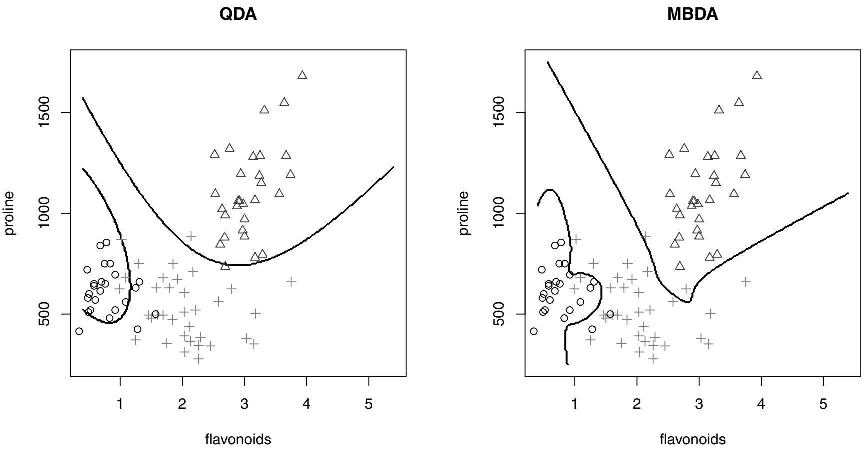

Fig. 7.4. Class boundaries for the wine data (proline and flavonoids only) for QDA (left) and MBDA (right). Models are created using the odd rows of the wine data; the plotting symbols indicate the even rows (test data), as mentioned in the text.

```
> wines.qda <- qda(wines[odd,], vintages[odd],
+                  prior = rep(1, 3)/3)
> test.qdapred <- predict(wines.qda, newdata = wines[even,])
> table(vintages[even], test.qdapred$class)

             test.qdapred
              Barbera Barolo Grignolino
  Barbera          24      0          0
  Barolo            0     29          0
  Grignolino        0      0         35
```

In this case, all test set predictions are correct.

The optional correction for unequal class sizes (or account for prior probabilities) is done in exactly the same way as in LDA. Several other arguments are shared between the two functions: both `lda` and `qda` can be called with the `method` argument to obtain different estimates of means and variances: the standard plug-in estimators, maximum likelihood estimates, or two forms of robust estimates. The `CV` argument enables fast LOO crossvalidation.

7.1.5 Model-Based Discriminant Analysis

Although QDA uses more class-specific information, it still is possible that the data are not well described by the individual covariance matrices, e.g., in case of non-normally distributed data. In such a case one can employ more greedy forms of discriminant analysis, utilizing very detailed descriptions of class densities. In particular, one can describe each class with a mixture of normal distributions, just like in model-based clustering, and then assign an object to the class for which the overall mixture density is maximal. Thus, for every class one estimates *several* means and covariance matrices – one describes the pod by a set of peas. Obviously, this technique can only be used when the ratio of objects to variables is very large.

Package **mclust** contains several functions for doing *model-based discriminant analysis* (MBDA), or *mixture discriminant analysis*, as it is sometimes called as well. The easiest one is `mclustDA`:

```
> wines.mclustDA <- mclustDA(train = list(data = wines[odd,],
+                                labels = vintages[odd]),
+                             test = list(data = wines[even,],
+                                labels = vintages[even]),
+                             G = 1:5)

EEI VEI VEI
  3   2   3
```

In this case, we have restricted the number of Gaussians, to be used for each individual class, to be at most five. Again, the BIC value is employed to select the optimal model complexity. For the Barolo class, a mixture of three Gaussians seems optimal; these all have the same diagonal covariance matrix (indicated with model EEI). The two other classes can be described by mixtures of two and three diagonal covariance matrices, respectively, with varying volume – see Section 6.3 for more information on model definition in **mclust**. The `print` method for the fitted object gives more information:

```
> wines.mclustDA

Modeling Summary:
           trainClass mclustModel numGroups
Barolo         Barolo         EEI         3
Grignolino Grignolino         VEI         2
Barbera       Barbera         VEI         3

Test Classification Summary:
   Barbera     Barolo Grignolino
        24         29         35

Training Classification Summary:
   Barbera     Barolo Grignolino
        24         28         37

Training Error: 0.01123596

Test Error: 0
```

Classification errors for both training and test sets are very low: the training set has one misclassification, and in the test set all objects are predicted correctly.

If more control is needed over the training process, functions `mclustDAtrain` and `mclustDAtest` are available in the **mclust** package. To visualize the class boundaries in the two dimensions of the wine data set employed earlier for the other forms of discriminant analysis, we can use

```
> wines.mclust2D <- mclustDAtrain(wines[odd, c(7, 13)],
+                                 vintages[odd], G = 1:5)
```

```
XXI VEV EVI
  1   3   2
```

The model is simpler than the model employed for the full, 13-dimensional data, which seems logical. Prediction and visualization is done by

```
> wines.mclust2Dpred <- mclustDAtest(gridXY, wines.mclust2D)
> contour(x, y,
+         matrix(apply(wines.mclust2Dpred, 1, which.max),
+                nrow = length(x), byrow = TRUE),
+         main = "MBDA", drawlabels = FALSE,
+         xlab = "flavonoids", ylab = "proline")
> points(wines[even, c(7, 13)],
+        col = as.integer(vintages[even]),
+        pch = as.integer(vintages[even]))
```

The class boundaries, shown in the right plot of Figure 7.4, are clearly much more adapted to the densities of the individual classes, compared to the other forms of discriminant analysis we have seen.

7.1.6 Regularized Forms of Discriminant Analysis

At the other end of the scale we find methods that are suitable in cases where we cannot afford to use very complicated descriptions of class density. One form of regularized DA (RDA) strikes a balance between linear and quadratic forms [68]: the idea is to apply QDA using covariance matrices $\widetilde{\Sigma}_k$ that are shrunk towards the pooled covariance matrix Σ:

$$\widetilde{\Sigma}_k = \alpha \widehat{\Sigma}_k + (1 - \alpha)\Sigma \tag{7.14}$$

where $\widehat{\Sigma}_k$ is the empirical covariance matrix of class k. In this way, characteristics of the individual classes are taken into account, but they are stabilized by the pooled variance estimate. The parameter α needs to be optimized, e.g., by using crossvalidation.

In cases where the number of variables exceeds the number of samples, more extreme regularization is necessary. One way to achieve this is shrinkage towards the unity matrix [3]:

$$\tilde{\Sigma} = \alpha\Sigma + (1 - \alpha)I \tag{7.15}$$

Equivalent formulations are given by:

$$\tilde{\Sigma} = \kappa\Sigma + I \tag{7.16}$$

and

$$\tilde{\Sigma} = \Sigma + \kappa I \tag{7.17}$$

with $\kappa \geq 0$. In this form of RDA, again the regularized form $\tilde{\Sigma}$ of the covariance is used, rather than the empirical pooled estimate Σ. Matrix $\tilde{\Sigma}$ is not singular so that the matrix inversions in Equations 7.3 or 7.6 no longer present a problem. In the extreme case, one can use a diagonal covariance matrix (with the individual variances on the diagonal) leading to diagonal LDA [69], also known as Idiot's Bayes [70]. Effectively, all dependencies between variables are completely ignored. For so-called "fat" matrices, containing many more variables than objects, often encountered in microarray research and other fields in the life sciences, such simple methods often give surprisingly good results.

Diagonal Discriminant Analysis

As an example, consider the odd rows of the prostate data, limited to the first 1000 variables. We are concentrating on the separation between the control samples and the cancer samples:

```
> prost <- prostate[prostate.type != "bph",1:1000]
> prost.type <- factor(prostate.type[prostate.type != "bph"])
> odd <- seq(1, length(prost.type), by = 2)
> even <- seq(2, length(prost.type), by = 2)
```

Although it is easy to re-use the code given in Sections 7.1.1 and 7.1.4, plugging in diagonal covariance matrices, we will use the `dDA` function from the **sfsmisc** package.

```
> prost.dlda <-
+   dDA(prost[odd,], as.integer(prost.type)[odd], pool = TRUE)
```

The `pool = TRUE` argument (the default) indicates that for all classes the same covariance matrix is to be used (LDA). The result for the predictions on the even samples is not too bad:

```
> prost.dldapred <- predict(prost.dlda, prost[even,])
> table(prost.type[even], prost.dldapred)

         prost.dldapred
           1  2
  control 32  8
  pca      7 77
```

Almost 88% of the test samples are predicted correctly. Allowing for different covariance matrices per class, we arrive at diagonal QDA, which does slightly worse for these data:

```
> prost.dqda <-
+   dDA(prost[odd,], as.integer(prost.type)[odd], pool = FALSE)
> prost.dqdapred <- predict(prost.dqda, prost[even,])
> table(prost.type[even], prost.dqdapred)
```

```
         prost.dqdapred
           1  2
  control 38  2
  pca     16 68
```

Shrunken Centroid Discriminant Analysis

In the context of microarray analysis, it has been suggested to combine RDA with the concept of "shrunken centroids" [71] – the resulting method is indicated as SCRDA [72] and is available in the R package **rda**. As the name suggests, class means are shrunk towards the overall mean. The effect is that the points defining the class boundaries (the centers) are closer, which may lead to a better description of local structure. These shrunken class means are then used in Equation 7.3, together with the diagonal covariance matrix also employed in DLDA. For a more complete description, see, e.g., [3].

Let us see how SCRDA does on the prostate example. Application of the `rda` function is straightforward[1]. The function takes two arguments, α and δ, where α again indicates the amount of unity matrix in the covariance estimate, and δ is a soft threshold, indicating the minimal coefficient size for variables to be taken into account in the classification:

```
> prost.rda <-
+   rda(t(prost[odd,]), as.integer(prost.type)[odd],
+       delta = seq(0, .4, length = 5),
+       alpha = seq(0, .4, length = 5))
```

Printing the fitted object shows some interesting results:

```
> prost.rda
```

```
Call:
rda(x = t(prost[odd, ]),
    y = as.integer(prost.type)[odd],
    xnew = t(prost[even, ]),
    ynew = as.integer(prost.type)[even],
    alpha = seq(0, 0.4, length = 5),
    delta = seq(0, 0.4, length = 5))
```

[1] Note that in this function the variables are in the *rows* of the data matrix and not, as usual, in the columns – hence the use of the transpose function.

```
$nonzero
     delta
alpha     0 0.1 0.2 0.3 0.4
  0    1000 433 193 121  92
  0.1 1000 220  34   3   0
  0.2 1000 192  19   4   0
  0.3 1000 179  18   4   0
  0.4 1000 195  24   4   0

$errors
     delta
alpha  0 0.1 0.2 0.3 0.4
  0   36  38  39  39  39
  0.1 10  16  32  41  41
  0.2  7  21  43  41  41
  0.3  4  23  44  41  41
  0.4  2  20  44  41  41
```

Increasing values of δ lead to a rapid decrease in the number of non-zero coefficients; however, these sparse models do not lead to very good predictions, and the lowest value for the training error is found at $\alpha = .4$ and $\delta = 0$. Obviously, the training error is not the right criterion to decide on the optimal values for these parameters. This we can do using the `rda.cv` crossvalidation function, and subsequently we can use the test data as a means to estimate the expected prediction error:

```
> prost.rdacv <-
+   rda.cv(prost.rda, t(prost[odd,]),
+          as.integer(prost.type)[odd])
```

Inspection of the result (not shown) reveals that the optimal value of α would be .2, with no thresholding ($delta = 0$). Predictions with these values lead to the following result:

```
> prost.rdapred <-
+   predict(prost.rda,
+           t(prost[odd,]), as.integer(prost.type)[odd],
+           t(prost[even,]), alpha = .2, delta = 0)
> table(prost.type[even],prost.rdapred)

         prost.rdapred
           1  2
  control 30 10
  pca      4 80
```

Overall, fourteen samples are misclassified, only slightly better than the DLDA model. This sort of behaviour is more general than one might think: for fat data, the simplest models are often among the top performers.

7.2 Nearest-Neighbour Approaches

A completely different approach, not relying on any distributional assumptions whatsoever, is formed by techniques focussing on distances between objects, and in particular on the closest objects. These techniques are known under the name of k-nearest-neighbours (KNN), where k is a number to be determined. If $k = 1$, only the closest neighbour is taken into account, and any new object will be assigned to the class of its closest neighbour in the training set. If $k > 1$, the classification is straightforward in cases where the k nearest neighbours are all of the same class. If not, a majority vote is usually performed. Class areas can be much more fragmented than with LDA or QDA; in extreme cases one can even find patch-work-like patterns. The smaller the number k, the more irregular the areas can become: only one object is needed to assign its immediate surroundings to a particular class.

As an example, consider the KNN classification for the first sample in the test set of the wine data (sample number two), based on the training set given by the odd samples. One starts by calculating the distance to all samples in the training set. Usually, the Euclidean distance is used – in that case, one should scale the data appropriately to avoid large numbers to dominate the results. For the wine data, autoscaling is advisable. The `mahalanobis` function has a useful feature that allows one to calculate the distance of one object to a set of others. The covariance matrix is given as the third argument[2]. Thus, the Euclidean distance can be calculated in two ways, either from the autoscaled data using a unit covariance matrix, or from the unscaled data using the estimated column standard deviations:

```
> wines.sc <- scale(wines, scale = sd(wines[odd,]),
+                   center = colMeans(wines[odd,]))
> dist2sample2a <- mahalanobis(wines.sc[odd,], wines.sc[2,],
                               diag(13))
> dist2sample2b <- mahalanobis(wines[odd,], wines[2,],
+                              diag(sd(wines[odd,])^2))

> range(dist2sample2a - dist2sample2b)

[1] -7.105427e-15  7.105427e-15
```

Next, we order the training samples according to their distance to sample two:

```
> nearest.classes <- vintages[odd][order(dist2sample2a)]
> nearest.classes[1:10]

 [1] Barolo Barolo Barolo Barolo Barolo Barolo Barolo Barolo
 [9] Barolo Barolo
Levels: Barbera Barolo Grignolino
```

[2] Note that function `sd` returns a vector of per-column standard deviations, and the `diag` functionis used to convert this into a matrix. An alternative would be to use `diag(diag(var(...)))`.

The closest ten objects are all of the Barolo class – apparently, there is little doubt that object 2 also should be a Barolo.

Rather than using a diagonal of the covariance matrix, one could also use the complete estimated covariance matrix of the training set. This would lead to the Mahalanobis distance:

```
> dist2sample2 <- mahalanobis(wines[odd,], wines[2,],
+                             cov(wines[odd,]))
> nearest.classes <- vintages[odd][order(dist2sample2)]
> nearest.classes[1:10]
```

```
 [1] Barolo     Barolo     Barolo     Grignolino Grignolino
 [6] Grignolino Barolo     Barolo     Grignolino Barolo
```

Note that autoscaling of the data is not necessary. Clearly, the results depend on the distance measure employed. Although the closest three samples are Barolo wines, the next three are Grignolinos; values of k between 5 and 9 would lead to a close call or even a tie. Several different strategies to deal with such cases can be employed. The simplest is to require a significant majority for any classification – in a 5-NN classification one may require at least four of the five closest neighbours to belong to the same class. If this is not the case, the classification category becomes "unknown". Although this may seem a weakness, in many applications it is regarded as a strong point if a method can indicate some kind of reliability – or lack thereof – for individual predictions.

The **class** package contains an implementation of the KNN classifier using Euclidean distances, `knn`. Its first argument is a matrix constituting the training set, and the second argument is the matrix for which class predictions are required. The class labels of the training set are given in the third argument. It provides great flexibility in handling ties: the default strategy is to choose randomly between the (tied) top candidates, so that repeated application can lead to different results:

```
> X <- scale(wines, scale = sd(wines[odd,]),
+            center = colMeans(wines[odd,]))
> knn(X[odd,], X[68,], cl = vintages[odd], k = 4)
```

```
[1] Barbera
Levels: Barbera Barolo Grignolino
```

```
> knn(X[odd,], X[68,], cl = vintages[odd], k = 4)
```

```
[1] Grignolino
Levels: Barbera Barolo Grignolino
```

Apparently, there is some doubt about the classification of sample 68 – it can be either a Barbera or Grignolino. Of course, this is caused by the fact that from the four closest neighbours, two are Barberas and two are Grignolinos.

Requiring at least three votes for an unambiguous classification (`l = 3`) leads to:

```
> knn(X[odd,], X[68,], cl = vintages[odd], k = 4, l = 3)
```

```
[1] <NA>
Levels: Barbera Barolo Grignolino
```

In many cases it is better not to have a prediction at all, rather than a highly uncertain one.

The value of k is crucial. Unfortunately, no rules of thumb can be given on the optimal choice, and this must be optimized for every data set separately. One simple strategy is to monitor the performance of the test set for several values of K and pick the one that leads to the smallest number of misclassifications. Alternatively, LOO crossvalidation can be employed:

```
> wines.knnresult <- rep(0, 10)
> for (i in 1:10) {
+   wines.knncv <- knn.cv(X[odd,], vintages[odd], k = i)
+   wines.knnresult[i] <-
+     sum(diag(table(vintages[odd], wines.knncv))) }
> 100 * wines.knnresult / length(odd)
```

```
 [1] 92.1 92.1 96.6 93.3 95.5 96.6 96.6 96.6 96.6 97.8
```

In this example, $k = 10$ shows the best prediction, although $k = 3$ and values six to nine also perform well.

An alternative is to use the convenience function `tune.knn` in package **e1071**. This function by default uses ten-fold crossvalidation for a range of values of k:

```
> knn.tuned <- tune.knn(X[odd,], vintages[odd], k = 1:10)
> knn.tuned
```

```
- sampling method: 10-fold cross validation

- best parameters:
 k
 6

- best performance: 0.01111111
```

```
> plot(knn.tuned)
```

The result is shown in the left plot of Figure 7.5 – the differences with the LOO results we saw earlier show what kind of variability is to be expected with crossvalidated error estimates. Indeed, repeated application of the tune function will – for these data – lead to quite different estimates for the optimal value of k:

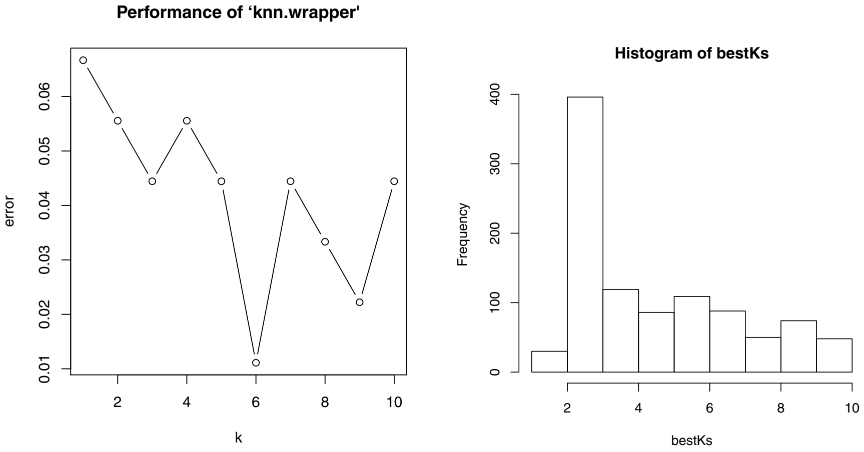

Fig. 7.5. Optimization of k for the wine data using the `tune` wrapper function. Left plot: one crossvalidation curve. Right plot: optimal values of k in 1000 crossvalidations.

```
> bestKs <- rep(0, 1000)
> for (i in 1:1000)
>   bestKs[i] <- tune.knn(X1, vintages[odd],
+                         k = 1:10)$best.parameters[1,1]
> hist(bestKs)
```

The result is shown in the right plot of Figure 7.5. In roughly half the cases, $k = 2$ is best. This is partly caused by a built-in preference for small values of k in the script: the smallest value of k that gives the optimal predictions is stored, even though larger values may lead to equally good predictions.

Although application of these simple strategies allow one to choose the optimal parameter settings, the optimal error associated with this setting (e.g., 97.8% in the LOO example) is not an estimation of the prediction error of future samples, because the test set is used in this procedure to fine-tune the method. Another layer of validation is necessary to find the estimated prediction error; see Chapter 9. The 1-nearest neighbour method enjoys great popularity, despite coming out worst in the above comparison – there, it is almost never selected. Nevertheless, it has been awarded a separate function in the **class** package: `knn1`. Most often, odd values of K smaller than ten are considered.

One potential disadvantage of the KNN method is that in principle, the whole training set – the training set in a sense *is* the model! – should be saved, which can be a nuisance for large data sets. Predictions for new objects can be slow, and storing really large data sets may present memory problems. However, things are not so bad as they seem, since in many cases one can safely prune away objects without sacrificing information. For the wine data, it is obvious that in large parts of the space there is no doubt: only objects from one class are present. Many of these objects can be removed and one then still will get exactly the same classification for all possible new objects.

7.3 Tree-Based Approaches

A wholly different approach to classification is formed by the tree-based approaches. These proceed in a way that is very similar to medical diagnosis: the data are "interrogated" and a series of questions are posed which finally lead to a classification. Modelling, in this metaphore, is to decide which questions are most informative. As a class, tree-based methods possess some unique advantages. They can be used for both classification and regression. Since the model is based on sequential decisions on individual variables, scaling is not important: every variable is treated at its own scale and no "overall" measure needs to be computed. Variable selection comes with the method – only those variables that contribute to the result are incorporated in the tree. Trees form one of the few methods that can accomodate variables of a very different nature, e.g., numerical, categorical and ordinal, in one single model. Their handling of missing values is simple and elegant. In short, trees can be used for almost any classification (and regression) problem.

Currently, tree-based modelling comes in two main branches: Breiman's Classification and Regression Trees (CART, [73]) and Quinlan's See5/C5.0 (and its predecessors, C4.5 [74] and ID3 [75]). Both are commercial and not open-source software, but R comes with two pretty faithful representation of CART in the form of the `rpart` and `tree` functions, from the packages with the same names. Since the approaches by Quinlan and Breiman have become more similar with every new release, and since no See5/C5.0 implementation is available in R, we will here only focus on `rpart` [76] – one of the recommended R packages – to describe the main ideas of tree-based classification. The differences with the implementation in the **tree** package are small; consult the manual pages and the technical report by Therneau *et al.* [76] for more information.

7.3.1 Recursive Partitioning and Regression Trees

Recursive Partitioning and Regression Trees, which is what the acronym `rpart` stands for, can be explained most easily by means of an example. Consider, again, the the two-dimensional subset of the wine data encountered earlier:

```
> wines.df <- data.frame(vint = vintages, wines[,c(7, 13)])
> wines.rpart <- rpart(vint ~ ., subset = odd,
+                            data = wines.df, method = "class")
```

In setting up the tree model, we explicitly indicate that we mean classification (`method = "class"`): the `rpart` function also provides methods for survival analysis and regression, and though it tries to be smart in guessing what exactly is required, it is better to explicitly provide the `method` argument. The result is an object of class `rpart`:

```
> wines.rpart

n= 89

node), split, n, loss, yval, (yprob)
      * denotes terminal node

1) root 89 53 Grignolino (0.2697 0.3258 0.4045)
  2) flavonoids< 1.23 23  1 Barbera (0.9565 0.0000 0.0435) *
  3) flavonoids>=1.23 66 31 Grignolino (0.0303 0.4394 0.5303)
    6) proline>=739 30  2 Barolo (0.0000 0.9333 0.0667) *
    7) proline< 739 36  3 Grignolino (0.0556 0.0278 0.9167) *
```

The top node, where no splits have been defined, is labelled as “Grignolino” since that is the most abundant variety – 36 out of 89 objects (a fraction of 0.4045) are Grignolinos. The first split is on the `flavonoids` variable. A value smaller than 1.235 leads to node 2, which is consisting almost completely of Barbera samples (more than 95 percent). This node is not split any further, and in tree terminology is indicated as a “leaf”. A flavonoid value larger than 1.235 leads to node three that is split further into separate Barolo and Grignolino leaves.

Of course, such a tree is much easier to interpret when depicted graphically:

```
> plot(X.rpart, margin = .12)
> text(X.rpart, use.n = TRUE)
> plot(X, pch = as.integer(vint)+1)
> segments(X.rpart$splits[1,4], par("usr")[3],
+          X.rpart$splits[1,4], par("usr")[4], lty = 2)
> segments(X.rpart$splits[1,4], X.rpart$splits[2,4],
+          par("usr")[2], X.rpart$splits[2,4], lty = 2)
```

This leads to the plots in Figure 7.6. The tree on the left shows the splits, corresponding to the tesselation of the surface in the right plot. The plot command sets up the coordinate system en plots the tree; the `margin` argument is necessary to reserve some space for the annotation added by the `text` command. At every split, the test, stored in the `splits` element in the `X.rpart` object, is shown. At the final nodes (the “leaves”), the results are summarized: two Barberas (triangles) are classified as Barolos (pluses), two Barolos are in the Grignolino area and one Grignolino is classified as a Barolo.

Application to multivariate data is equally simple. For the wine data, we will predict the classes of the even rows, again based on the odd rows:

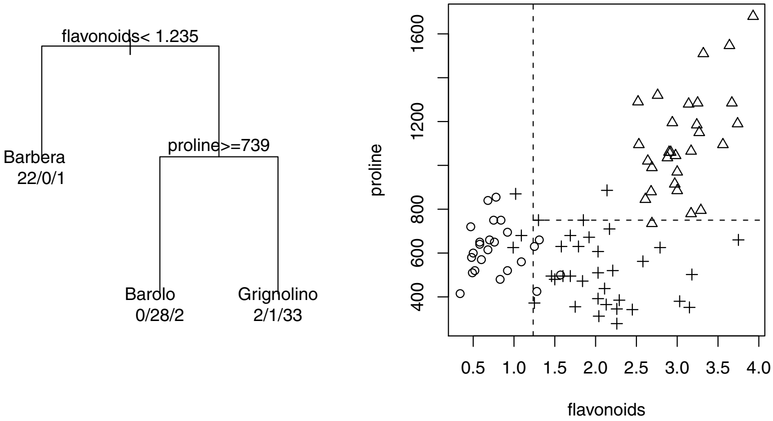

Fig. 7.6. Tree object from `rpart` using default settings (left). The two nodes lead to the class boundaries visualized in the right plot. Only points for the even rows, the test set, are shown.

```
> wines.df <- data.frame(wines, vint = vintages)
> wines.rpart <- rpart(vint ~ ., subset = odd,
+                      data = wines.df, method = "class")
> plot(wines.rpart, margin = .1)
> text(wines.rpart, use.n = TRUE)
```

The plot in Figure 7.7 shows that the flavonoids and proline variables are again important in the classification, now in addition to the colour intensity.

Prediction is done using the `predict.rpart` function, which returns a matrix of class probabilities, simply estimated from the composition of the training samples in the end leaves:

```
> wines.rpart.predict <- predict(wines.rpart,
+                                newdata = wines.df[even,])
> wines.rpart.predict[31:34,]
```

```
   Barbera Barolo Grignolino
62       0 0.0323      0.968
64       0 0.0323      0.968
66       0 0.1429      0.857
68       0 0.0323      0.968
```

In this rather simple problem, most of the probabilities are either 0 or 1, but here some Grignolinos are shown that have a slight chance of actually being Barolos, according to the tree model. The uncertainties are simply the misclassification rates of the training model: row 66 ends up in a lead containing

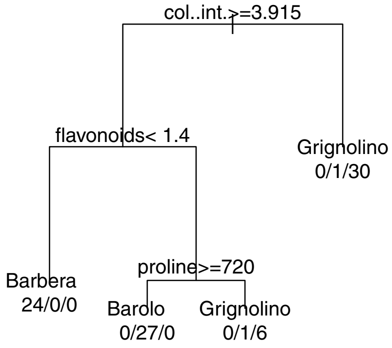

Fig. 7.7. Fitted tree using `rpart` on the odd rows of the `wine` data set (all thirteen variables).

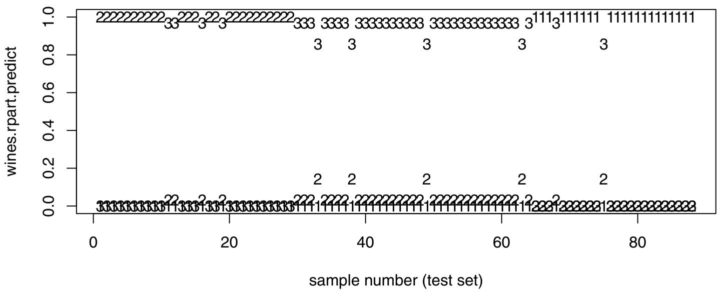

Fig. 7.8. Classification probabilities of the test set of the wine data for the tree shown in Figure 7.7. Within the first twenty samples we see four incorrect predictions: the true class is Barolo (indicated by "2") but there is some confusion with Grignolino ("3"). Similarly, the two other misclassifications show up clearly.

seven training samples, one of which is a Barolo and six are Grignolinos. The other rows end up in the large Grignolino group, containing also one Barolo sample. A global overview is more easily obtained by plotting the probabilities:

```
> matplot(wines.rpart.predict)
```

This leads to the plot in Figure 7.8. Clearly, most of the Barolos and Barberas are classified with complete confidence, corresponding with "pure" leaves. The Grignolinos on the other hand always end up in a leaf also containing one Barolo sample. When using the `type = "class"` argument to the prediction function, the result is immediately expressed in terms of classes, and can be used to assess the overall prediction quality:

```
> table(vintages[even],
+       predict(wines.rpart, newdata = wines.df[even,],
+               type = "class"))

             Barbera Barolo Grignolino
  Barbera         22      0          2
  Barolo           0     25          4
  Grignolino       0      0         35
```

This corresponds to the six misclassifications seen in Figure 7.8.

Constructing the Tree

The construction of the optimal tree is an NP-complete problem, and therefore one has to resort to simple approximations. The standard approach is the following. All possible splits – binary divisions of the data – in all predictor variables are considered; the one leading to the most "pure" branches is selected. The term "pure" in this case signifies that, in one leaf, only instances of one class are present. For categorical variables, tests for unique values are used; for continuous variables, all data points are considered as potential split values. This simple procedure is applied recursively until some stopping criterion is met.

The crucial point is the definition of "impurity": several different measures can be used. Two criteria are standing out [34]: the Gini index, and the entropy. The Gini index of a node is given by

$$I_G(p) = \sum_{i \neq j} p_i p_j = 1 - \sum_j p_j^2 \tag{7.18}$$

and is minimal (exactly zero) when the node contains only samples from one class – p_i is the fraction of samples from class i in the node. The entropy of a node is defined by

$$I_E(p) = -\sum_j p_j \log p_j$$

which again is minimal when the node is pure and contains only samples of one class (and we define $0 \log 0 = 0$).

The optimal split is the one that minimises the average impurity of the new left and right branches:

$$P_l I(p_l) + P_r I(p_r)$$

where P_l and P_r signify the sample fractions and $I(p_l)$ and $I(p_r)$ are the impurities of the left and right branches, respectively.

As an illustration, again consider the two-dimensional subset of the odd rows of the wine data, using variables `flavonoids` and `proline`. Since the data are continuous, we consider all values as potential splits, and calculate the Gini and entropy indices. We can, e.g., define a small function to calculate the impurity based on the Gini index:

```
> gini <- function(x, class, splitpoint)
+ {
+   left.ones <- class[x < splitpoint]
+   right.ones <- class[x >= splitpoint]
+   nleft <- length(left.ones)
+   nright <- length(right.ones)
+
+   if ((nleft == 0) | (nright == 0)) return (NA)
+
+   p.left <- table(left.ones) / nleft
+   p.right <- table(right.ones) / nright
+
+   (nleft * (1 - sum(p.left^2)) +
+    nright * (1 - sum(p.right^2))) /
+     (nleft + nright)
+ }
```

This function takes a vector `x`, for instance values for the `proline` variable in the `wines` data, a class vector and a split point, and returns the impurity value defined in Equation 7.18. To calculate which split is optimal for the two-dimensional wine data, we can use all possible values as split points, and plot the result:

```
> Ginis <- matrix(0, nrow(X), 2)
> splits.flav <- sort(X[,1])
> splits.prol <- sort(X[,2])
> for (i in 1:nrow(X)) {
+   Ginis[i,1] <- gini(X[,1], vint, splits.flav[i])
+   Ginis[i,2] <- gini(X[,2], vint, splits.prol[i])
+ }
> matplot(Ginis, pch = 1:2, col = 1:2)
> legend("topleft", legend = c("flavonoids", "proline"),
+        col = 1:2, pch = 1:2)
```

Figure 7.9 shows that the first split should be on the `flavonoids` column: it leads to a lower Gini value, i.e., more pure leaves. To identify the exact location of the minima, we can use function `which.min`:

```
> apply(Ginis, 2, which.min)
```

```
[1] 24 56
```

The 24th element in the sorted flavonoids concentrations is 1.25, but any value between the 23rd (1.22) and 24th elements would of course lead to the same Gini index.

The division obtained in this way can be split further for both the left and right parts of the data. Following the same strategy as above, this leads to the following suggestions for both variables:

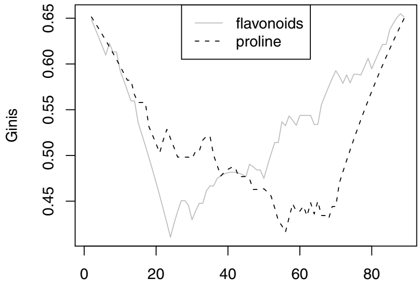

Fig. 7.9. Impurity values (Gini indices) for all possible split points in the two-dimensional subset of the wine data. The 24th value of the `flavonoids` variable lead to the best solution and is selected as the first split.

	Left part		Right part	
	Optimal value	Δ Gini index	Optimal value	Δ Gini index
Flavonoids	0.575	0.0107	2.30	0.1875
Proline	525	0.0180	739	0.3832

In this table, the third and fifth columns signify the change in purity compared to the unsplit parent node. Clearly, the best option is to choose the split in the right part on `proline` at a value of 739. This corresponds exactly to the result shown in Figure 7.6.

Obviously, one can keep on splitting nodes until every sample in the training set is a leaf in itself, or in any case until each single leaf contains only instances of one class. Such a tree is able to represent the training data perfectly, but whether the predictions of such a tree are reliable is quite another matter. In fact, these trees generally will not perform very well. By describing every single feature of the training set, the tree is not able to generalize. This is an example of *overfitting* (or overtraining, as it is sometimes called as well), something that is likely to occur in methods that have a large flexibility – in the case of trees, the freedom to keep on adding nodes.

The way this problem is tackled in constructing optimal trees is to use *pruning*, i.e., trimming useless branches. When exactly a branch is useless needs to be assessed by some form of validation – in **rpart**, tenfold cross-validation is used by default. One can therefore easily find out whether a particular branch leads to a decrease in prediction error or not.

More specifically, in pruning one minimizes the cost of a tree, expressed as

$$C(T) = R(T) + \alpha|T| \tag{7.19}$$

In this equation, T is a tree with $|T|$ leafs, $R(T)$ the “risk” of the tree – e.g., the proportion of misclassifications – and α a complexity penalty, chosen between 0 and ∞. One can see α as the cost of adding another node. It is not necessarily to build up the complete tree to calculate this measure: during the construction the cost of the current tree can be assessed and if it is above a certain value, the process stops. This cost is indicated with the complexity parameter (cp) in the rpart function, which is normalized so that the root node has a complexity value of one.

Once again looking at the first 1000 variables of the control and pca classes in the prostate data, one can issue the following commands to construct the full tree with no misclassifications in the training set:

```
> prost.df <- data.frame(type = prost.type, prost = prost)
> prost.rprt <-
+   rpart(type ~ ., data = prost.df, subset = odd,
+         control = rpart.control(cp = 0, minsplit = 0))
```

The two extra arguments tell the rpart function to keep on looking for splits even when the complexity parameter, cp, gets smaller than 0.1 and the minimal number of objects in a potentially split node, minsplit, is smaller than 20 (the default values). This leads to a tree with seven leaves. Printing the prost.rprt object would show that the training data are indeed predicted perfectly – however, four of the terminal nodes only three or fewer samples. Indeed, the test data are not predicted with the same level of accuracy:

```
> prost.rprtpred <-
+   predict(prost.rprt, newdata = prost.df[even,])
> table(prost.type[even], classmat2classvec(prost.rprtpred))

          control pca
  control      29  11
  pca          12  72
```

Pruning could decrease the complexity without sacrificing much accuracy in the description of the training set, and hopefully would increase the generalising abilities of the model. To see what level of pruning is necessary, the table of complexity values can be printed:

```
> printcp(prost.rprt)

Classification tree:
rpart(formula = type ~ ., data = prost.df,
      subset = odd, control = rpart.control(cp = 0,
      minsplit = 0))

Variables actually used in tree construction:
[1] prost.112 prost.209 prost.360 prost.588 prost.8 prost.965
```

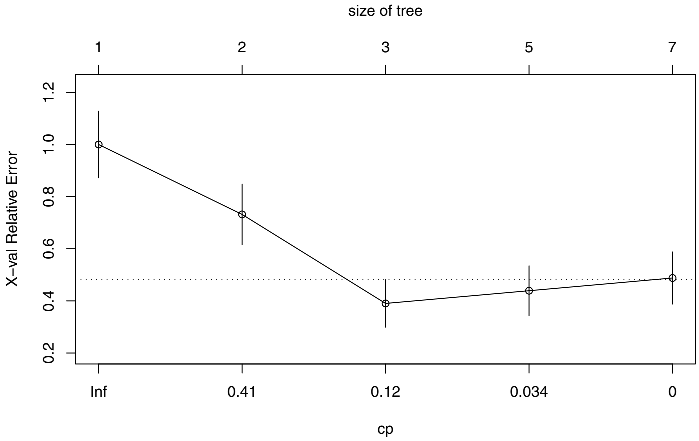

Fig. 7.10. Complexity pruning of a tree: in this case, three terminal nodes are optimal (lowest prediction error at lowest complexity).

```
Root node error: 41/125 = 0.328

n= 125

       CP nsplit rel error  xerror     xstd
1 0.56098      0   1.00000 1.00000 0.128024
2 0.29268      1   0.43902 0.73171 0.116462
3 0.04878      2   0.14634 0.39024 0.091103
4 0.02439      4   0.04878 0.43902 0.095739
5 0.00000      6   0.00000 0.48780 0.099970
```

Also a graphical representation is available:

```
> plotcp(prost.rpart)
```

This leads to Figure 7.10. Both from this figure and the complexity table shown above, it is clear that the tree with the lowest prediction error and the least number of nodes is obtained at a value of `cp` equal to 0.12. Usually, one chooses the complexity corresponding to the minimum of the predicted error plus one standard deviation, indicated by the dotted line in Figure 7.10. The tree created with a `cp` value of 0.12, containing only two leaves rather than the original seven, leads to a higher number of misclassifications (six rather than zero) in the training set, but unfortunately also to a slightly higher number of misclassifications in the test set:

```
> prost.rprt2 <-
+   rpart(type ~ ., data = prost.df, subset = odd,
+         control = rpart.control(cp = 0.12))
```

```
> prost.rprt2pred <-
+   predict(prost.rprt2, newdata = prost.df[even,])
> table(prost.type[even], classmat2classvec(prost.rprt2pred))

          control pca
  control      29  11
  pca          15  69
```

Either way, the result is quite a bit worse than what we have seen earlier with RDA (page 121).

Apart from the 0/1 loss function normally used in classification (a prediction is either right or wrong), **rpart** allows to specify other, more complicated loss functions as well – often, the cost of a false positive is very different from the cost of a false negative decision. Another useful feature in the **rpart** package is the possibility to provide prior probabilities for all classes.

7.3.2 Discussion

Trees offer a lot of advantages. Perhaps the biggest of them is the appeal of the model form: many scientists feel comfortable with a series of more and more specific questions leading to an unambiguous answer. The implicit variable selection makes model interpretation much easier, and alleviates many problems with missing values, and variables of mixed types (boolean, categorical, ordinal, numerical).

There are downsides too, of course. The number of parameters to adjust is large, and although the default settings quite often lead to reasonable solutions, there may be a temptation to keep fiddling until an even better result is obtained. This, however, can easily lead to overfitting: although the data are faithfully reproduced, the model is too specific and lacks generalization power. As a result, predictions for future data are generally of lower quality than expected. And as for the interpretability of the model: this is very much dependent on the composition of the training set. A small change in the data can lead to a completely different tree. As we will see, this is a disadvantage that can be turned into an advantage: combinations of tree-based classifiers often give stable and accurate predictions. These so-called Random Forests, taking away many of the disadvantages of simple tree-based classifiers while keeping the good characteristics, enjoy huge popularity and will be treated in Section 9.7.2.

7.4 More Complicated Techniques

When relatively simple models like LDA or KNN do not succeed in producing models with good predictive capabilities, one can ask the question: why do we fail? Is it because the data just do not contain enough information to build a useful model? Or are the models we have tried too simple? Do we need something more flexible, perhaps nonlinear? The distinction between information-poor data and complicated class boundaries is often hard to make.

In this section, we will treat two popular nonlinear techniques with complementary characteristics: whereas *Support Vector Machines* (SVMs) are very useful when the number of objects is not too large, *Artificial Neural Networks* (ANNs) should only be applied when there are ample training cases available. Conversely, SVMs are applicable in high-dimensional cases whereas ANNs are not: very often, a data reduction step like PCA is employed to bring the number of variables down to a manageable size. These two techniques do share one important property: they are very flexible indeed, and capable of modelling the most complex relationships. This puts a large responsibility on the researcher for thorough validation, especially since there are several parameters to tune. Because the theory behind the methods is rather extensive, we will only sketch the contours – interested readers are referred to the literature for more details.

7.4.1 Support Vector Machines

SVMs [77,78,79] in essence are binary classifiers, able to discriminate between two classes. They aim at finding a separating hyperplane maximizing the distance between the two classes. This distance is called the *margin* in SVM jargon; a synthetic example, present in almost all introductions to SVMs, is shown in Figure 7.11. Although both classifiers, indicated by the solid lines, perfectly separate the two classes, the classifier with slope 2/3 achieves a much bigger margin than the vertical line. The points that are closest to the hyperplane are said to lie on the margins, and are called *support vectors* – these are the only points that matter in the classification process itself. Note however that all other points have been used in setting up the model, i.e., in determining which points are support vectors in the first place. The fact that only a limited number of points is used in the predictions for new data is called the *sparseness* of the model, an attractive property in that it focuses attention to the region that matters, the boundary between the classes, and ignores the exact positions of points far from the battlefield.

More formally, a separating hyperplane can be written as

$$\boldsymbol{wx} - b = 0 \tag{7.20}$$

The margin is the distance between two parallel hyperplanes with equations

$$\boldsymbol{wx} - b = -1 \tag{7.21}$$

$$\boldsymbol{wx} - b = 1 \tag{7.22}$$

and is given by $2/||\boldsymbol{w}||$. Therefore, maximizing the margin comes down to minimizing $||\boldsymbol{w}||$, subject to the constraint that no data points fall within the margin:

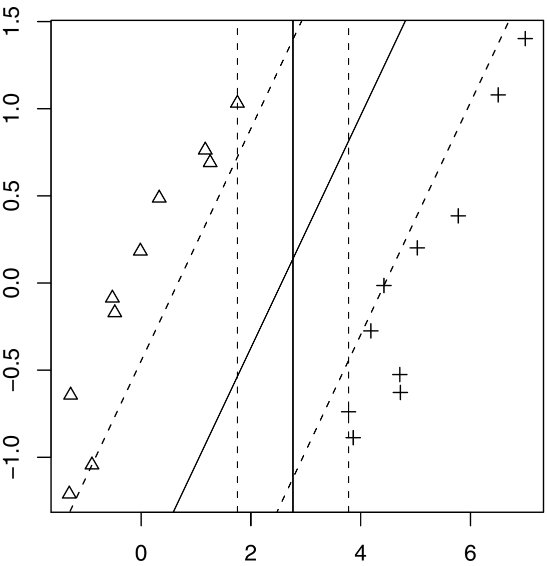

Fig. 7.11. The basic idea behind SVM classification: the separating hyperplane (here, in two dimensions, a line) is chosen in such a way that the margin is maximal. Points on the margins (the dashed lines) are called "support vectors". Clearly, the margins for the separating line with slope 2/3 are much further apart than for the vertical boundary.

$$c_i(\boldsymbol{w}\boldsymbol{x}_i) \leq 1 \tag{7.23}$$

where c_i is either -1 or 1, depending on the class label. This is a standard quadratic programming problem.

It can be shown that these equations can be rewritten completely in terms of inner products of the support vectors. This so-called *dual representation* has the big advantage that the original dimensionality of the data is no longer of interest: it does not really matter whether we are analysing a data matrix with two columns, or a data matrix with ten thousand columns. By applying suitable *kernel functions*, one can transform the data, effectively leading to a representation in higher-dimensional space. Often, a simple discrimination function can be obtained in this high-dimensional space, which translates into an often complex class boundary in the original space. Because of the dual representation, one does not need to know the exact transformation – it suffices to know that it exists, which is guaranteed by the use of kernel functions with specific properties. Examples of suitable kernels are the polynomial and gaussian kernels. More details can be found in the literature (e.g., [3]).

Package **e1071** provides an interface to the LIBSVM library[3] through the function svm. Application to the Barbera and Grignolino classes leads to the following results:

```
> wines.df <-
+   data.frame(vint = factor(vintages[vintages != "Barolo"]),
+              X = wines[vintages != "Barolo",])
> wines.svm <- svm(vint ~ ., data = wines.df, subset = odd)
> wines.svmpred <- predict(wines.svm, newdata = wines.df[even,])
> table(vint[even], wines.svmpred)
```

```
            wines.svmpred
             Barbera Grignolino
  Barbera         24          0
  Grignolino       0         35
```

These default settings lead to a perfect classification of the test set.

One attractive feature of SVMs is they are able to handle fat data matrices (where the number of features is much larger than the number of objects) without any problem. Let us see, for instance, how the standard SVM performs on the prostate data. We will separate the cancer samples from the other control class – again, we are considering only the first 1000 variables. Using the cross = 10 argument, we perform ten-fold crossvalidation, which should give us some idea of the performance on the test set:

```
> prost <- prostate[prostate.type != "bph",1:1000]
> prost.type <- factor(prostate.type[prostate.type != "bph"])
> prost.df <- data.frame(type = prost.type, prost = prost)
> prost.svm <- svm(type ~ ., data = prost.df, subset = odd,
+                  cross = 10)
> summary(prost.svm)
```

```
Call:
svm(formula = type ~ ., data = prost.df, cross = 10, subset = odd)


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  radial
       cost:  1
      gamma:  0.001

Number of Support Vectors:  88 ( 38 50 )

Number of Classes:  2
Levels: control pca
```

[3] See http://www.csie.ntu.edu.tw/~cjlin/libsvm/.

```
10-fold crossvalidation on training data:
Total Accuracy: 92
Single Accuracies:
 100 100 83.3 84.6 100 76.9 91.7 84.6 100 100
```

The summary (slightly edited to save space) shows us that rather than the complete training set of 125 samples, only 88 objects are seen as support vectors, which for SVMs is quite a large fraction. The prediction accuracies for the left out segments vary from 77 to 100%, with an overall error estimate of 8%. Let us see whether the test set can be predicted well:

```
> prost.svmpred <- predict(prost.svm, newdata = prost.df[even,])
> table(prost.type[even], prost.svmpred)
```

```
         prost.svmpred
          control pca
  control      33   7
  pca           1  83
```

Eight misclassifications out of 124 cases, nicely in line with the crossvalidation error estimate, is better than anything we have seen so far – not a bad result for default settings.

Extensions to More than Two Classes

The fact that only two-class situations can be tackled is a severe limitation: in reality, it often happens that we should discriminate between several classes. The standard approach is to turn one multi-class problem into several two-class problems. More specifically, one can perform one-against-one testing, where every combination of single classes is assessed, or one-against-all testing. In the latter case, the question is rephrased as: “to be class A or not to be class A” – the advantage is that, in the case of n classes, only n comparisons need to be made, whereas in the one-against-one case $.5n(n-1)$ models must be fitted. The disadvantage is that the class boundaries may be much more complicated: class “not A” may be very irregular in shape. The default in the function `svm` is to assess all one-against-one classifications, and use a voting scheme to pinpoint the final winning class.

```
> plot(wines.svm, wines.df[odd,], proline ~ flavonoids)
```

This leads to the left plot in Figure 7.12. The background colours indicate the predicted class for each region in the plot, obtained in a way very similar to the code used to produce the contour lines in Figure 7.3 and similar plots. Plotting symbols show the positions of the support vectors – these are shown as crosses, whereas regular data points, unimportant for this SVM model, are shown as open circles.

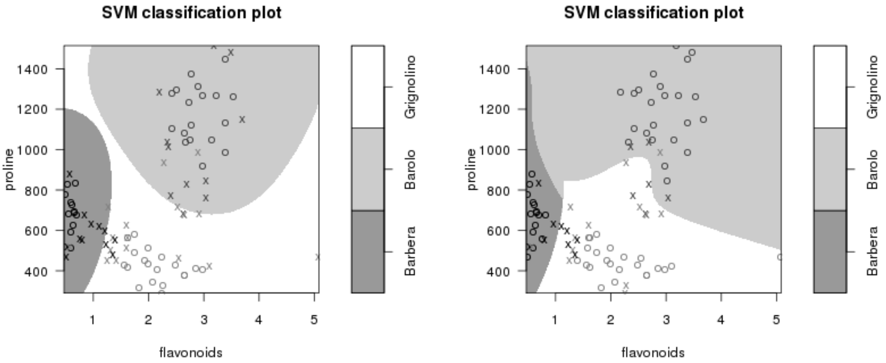

Fig. 7.12. SVM classification plots for the two-dimensional wine data (training data only). Support vectors are indicated by crosses; regular data points by open circles. Left plot: default settings of `svm`. Right plot: best SVM model with a polynomial kernel, obtained with `best.svm`.

Finding the Right Parameters

The biggest disadvantage of SVMs is the large number of tuning parameters. One should choose an appropriate kernel, and, depending on this kernel, values for two or three parameters. A special convenience function, `tune`, is available in the **e1071** package, which, given a choice of kernel, varies the settings over a grid, calculates validation values such as crossvalidated prediction errors, and returns an object of class `tune` containing all validation results. A related function is `best` which returns the model with the best validation performance. If we wanted to find the optimal settings for the three parameters `coef0`, `gamma` and `cost` using a polynomial kernel (the default kernel is a radial basis function), we could do it like this:

```
> wines.bestsvm <-
+   best.svm(vintages ~ ., data = wines.dfodd,
+              kernel = "polynomial",
+              coef0 = seq(-.5, .5, by = .1),
+              gamma = 2^(-1:1), cost = 2^(2:4))
> wines.bestsvmpred <-
+   predict(wines.bestsvm, newdata = wines.df[even,])
> sum(wines.bestsvmpred == vintages[even])
```

```
[1] 80
```

The number of correct classifications is exactly the same as with the default SVM parameters; however, the classification plot, shown in the right of Figure 7.12 looks completely different. Differences are mainly located in areas where no samples are present, but in some cases are also present in more relevant parts – consider, for example, the centers of the figures. This presents a

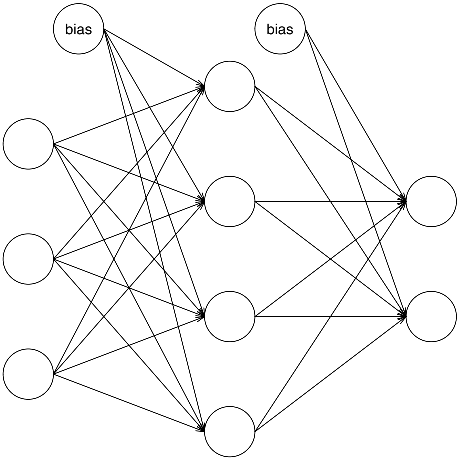

Fig. 7.13. The structure of a feedforward NN with three input units, four hidden units and two output units.

sobering illustration of the dangers we face when we trust complicated models trained with relatively few data points. Note that also the number and position of support vectors is radically different.

7.4.2 Artificial Neural Networks

Artificial Neural Networks (ANNs, also shortened to neural networks, NNs) form a family of extremely flexible modelling techniques, loosely based on the way neurons in human brains are thought to be connected – hence the name. Although the principles of NNs had already been defined in the fifties of the previous century with Rosenblatt's perceptron [80], the technique only really caught on some twenty years later with the publication of Rumelhart's and McClellands book [81]. Many different kinds of NNs have been proposed; here, we will only treat the flavour that has become known as *feed-forward neural networks*, *backpropagation networks*, after the name of the training rule (see below), or *multi-layer perceptrons*.

Such a network consists of a number of units, typically organized in three layers, as shown in Figure 7.13. When presented with input signals s_i, a unit will give an output signal s_o corresponding to a transformation of the sum of the inputs:

$$s_o = f(\sum_i s_i) \tag{7.24}$$

For the units in the input layer, the transformation is usually the identity function, but for the middle layer (the *hidden layer* in NN terminology) typically sigmoid transfer functions or threshold functions are used. For the hidden and output layers, special *bias units* are traditionally added, always having an output signal of +1 [34]. Network structure is very flexible. It is possible to use multiple hidden layers, remove links between specific units, to add connections skipping layers, or even to create feedback loops where output is again fed to special input units. However, the most common structure is to have a fully connected network such as the one depicted in Figure 7.13, consisting of one input layer, one hidden layer and one output layer. One can show that adding more hidden layers will not lead to better predictions (although in some cases it is reported to speed up training). Whereas the numbers of units in the input and output layers are determined by the data, the number of units in the hidden layer is a parameter that must be optimized by the user.

Connections between units are weighted: an output signal from a particular unit is sent to all connected units in the next layer, multiplied by the respective weights. These weights, in fact form the model for a particular network topology – training the network comes down to finding the set of weights that gives optimal predictions. The most popular training algorithm is called the *backpropagation* algorithm, and consists of a steepest-descent based adaption of the weights upon repeated presentation of training data. Many other training algorithms have been proposed as well.

In R, several packages are available providing general neural network capabilities, such as **AMORE** and **neuralnet** [82]. We will use the **nnet** package, one of the recommended R packages, featuring feed-forward networks with one hidden layer, several transfer functions and possibly skip-layer connections. It does not employ the usual backpropagation training rule but rather the optimization method provided by the R function `optim`.

For the autoscaled wine data, training a neural net with four units in the hidden layer is done as follows:

```
> X <- scale(wines, scale = sd(wines[odd,]),
+            center = colMeans(wines[odd,]))
> w.df <- data.frame(vintage = vintages, wines = X)
> wines.nnet <- nnet(vintage ~ ., data = w.df,
+                    size = 4, subset = odd)
```

```
# weights:  71
initial  value 109.648958
iter  10 value 0.812789
iter  20 value 0.002502
final  value 0.000093
converged
```

Although the autoscaling is not absolutely necessary (the same effect can be reached by using different weights for the connections of the input units to the hidden layer) it does make it easier for the network to reach a good solution – the optimization easily gets stuck in a local optimum. In practice, multiple training sessions should be performed, and the one with the smallest (crossvalidated) training error should be selected. An alternative is to use a (weighted) prediction using all trained networks.

As expected for such a flexibly fitting technique, the training data are reproduced perfectly:

```
> training.pred <- predict(wines.nnet, type = "class")
> sum(diag(table(vintages[odd],
+                training.pred))) / length(odd)

[1] 1
```

But also the test data in this case are predicted very well – only three errors are made:

```
> table(vintages[even],
+       classmat2classvec(predict(wines.nnet, X[even,])))

            Barbera Barolo Grignolino
  Barbera        23      0          1
  Barolo          0     29          0
  Grignolino      2      0         33
```

Several default choices have been made under the hood of the `nnet` function: the type of transfer functions in the hidden layer and in the output layer, the number of iterations, whether least-squares fitting or maximum likelihood fitting is done (default is least-squares), and several others. The only explicit setting in this example is the number of units in the hidden layer, and this immediately is the most important parameter, too. Choosing too many units will lead to a good fit of the training data but potentially bad generalization – overfitting. Too few hidden units will lead to a model that is not flexible enough.

A convenience function `tune.nnet` is available in package **e1071**, similar to `tune.svm`. Let us see whether our (arbitrary) choice of four hidden units can be improved upon:

```
> wines.nnetmodels <-
+   tune.nnet(vintage ~ ., data = w.df[odd,], size = 1:8)
> summary(wines.nnetmodels)

Parameter tuning of `nnet':
- sampling method: 10-fold cross validation

- best parameters:
 size
    2

- best performance: 0
- Detailed performance results:
  size   error dispersion
1    1 0.12361    0.09721
2    2 0.00000    0.00000
3    3 0.02222    0.04685
4    4 0.01111    0.03514
5    5 0.01111    0.03514
6    6 0.01111    0.03514
7    7 0.01111    0.03514
8    8 0.01111    0.03514
```

Clearly, one hidden unit is not enough; two hidden units apparently suffice. In addition to the `summary` function, a `plot` method is available as well. Instead of using `tune.nnet`, one can also apply `best.nnet` – this function directly returns the trained model with the optimal parameter settings:

```
> best.wines.nnet <-
+   best.nnet(vintage ~ ., data = w.df[odd,],
+             size = 1:8)
> table(vintages[even],
+       predict(best.wines.nnet, w.df[even,], type = "class"))
```

```
             Barbera Barolo Grignolino
  Barbera         23      0          1
  Barolo           0     29          0
  Grignolino       2      5         28
```

The result is quite a bit worse than the network with four hidden units shown above. It illustrates immediately the one major problem with applying neural networks to relatively small data sets: the variability of the training procedures is so large that one only has a faint idea of what optimal settings to apply. The golden rule of thumb is to use the smallest possible models. In this case, one would expect a network with only two hidden neurons to perform better than one with four, and perhaps, over a large number of training events, this may actually be the case. For individual training runs, however, it is not at all uncommon to see the kind of unexpected behaviour shown above.

In the example above we used the default stopping criterion of the `nnet` function, which is to perform 100 iterations of complete presentations of the training data. In several publications, scientists have advocated continuously monitoring prediction errors throughout the training iterations, in order to prevent the network from overfitting. In this approach, training should be stopped as soon as the error of the validation set starts increasing. Apart from the above-mentioned training parameters, this presents an extra level of difficulty which becomes all the more acute with small data sets. To keep these problems manageable, one should be very careful in applying neural networks in situations with few cases; the more examples, the better.

8

Multivariate Regression

In Chapters 6 and 7 we have concentrated on finding groups in data, or, given a grouping, creating a predictive model for new data. The last situation is "supervised" in the sense that we use a set of examples with known class labels, the training set, to build the model. In this chapter we will do something similar – now we are not predicting a discrete class property but rather a continuous variable. Put differently: given a set of independent real-valued variables (matrix $\boldsymbol{X}$), we want to build a model that allows prediction of $\boldsymbol{Y}$, consisting of one, or possibly more, real-valued dependent variables. As in almost all regression cases, we here assume that errors, normally distributed with constant variance, are only present in the dependent variables, or at least are so much larger in the dependent variables that errors in the independent variables can be ignored. Of course, we also would like to have an estimate of the expected error in predictions for future data.

8.1 Multiple Regression

The usual multiple least-squares regression (MLR), taught in almost all statistics courses, is modelling the relationship

$$\boldsymbol{Y} = \boldsymbol{X}\boldsymbol{B} + \mathcal{E} \tag{8.1}$$

where $\boldsymbol{B}$ is the matrix of regression coefficients and $\mathcal{E}$ contains the residuals. The regression coefficients are obtained by

$$\boldsymbol{B} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{Y} \tag{8.2}$$

with variance-covariance matrix

$$\mathrm{Var}(\boldsymbol{B}) = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\sigma^2 \tag{8.3}$$

R. Wehrens, *Chemometrics with R: Multivariate Data Analysis in the Natural Sciences and Life Sciences*, Use R!, DOI 10.1007/978-3-642-17841-2_8,

The residual variance σ^2 is typically estimated by

$$\hat{\sigma}^2 = \frac{1}{n-p-1} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \tag{8.4}$$

MLR has a number of attractive features, the most important of which is that it is the Best Linear Unbiased Estimator (BLUE) when the assumption of uncorrelated normally distributed noise with constant variance is met [26]. The standard deviations of the individual coefficients, given by the square roots of the diagonal elements of the variance-covariance matrix Var($\boldsymbol{B}$), can be used for statistical testing: variables whose coefficients are not significantly different from zero are sometimes removed from the model. Since removing one such variable will in general lead to different estimates for the remaining variables, this results in a stepwise variable selection approach (see Chapter 10).

For the `gasoline` data, a regression using four of the 401 wavelengths, evenly spaced over the entire range, would yield

```
> X <- gasoline$NIR[, 100*(1:4)]
> Y <- gasoline$octane
> odd <- seq(1, nrow(X), by = 2)
> even <- seq(2, nrow(X), by = 2)
> Xtr <- cbind(1, X[odd,])
> Ytr <- Y[odd]
> t(solve(crossprod(Xtr), t(Xtr)) %*% Ytr)

            1098 nm 1298 nm 1498 nm 1698 nm
[1,] 64.482  1312.9 -1607.5  229.94  26.244
```

Adding the column of ones using the `cbind` function on the fifth line in the example above causes an intercept to be fitted as well. The `solve` statement is the direct implemention of Equation 8.2. This also works when $\boldsymbol{Y}$ is multivariate – the regression matrix $\boldsymbol{B}$ will have one column for every variable to be predicted.

Rather than using this explicit matrix inversion, one would use the standard linear model function `lm`, which also provides the usual printing, plotting and summary functions:

```
> Xtr <- X[odd,]
> Blm <- lm(Ytr ~ Xtr)
> summary(Blm)

Call:
lm(formula = Y[odd] ~ X[odd, ])

Residuals:
     Min       1Q   Median       3Q      Max
-2.59870 -0.64860  0.07353  0.65509  1.73075
```

```
Coefficients:
                   Estimate Std. Error t value Pr(>|t|)
(Intercept)     64.5       15.6    4.12  0.00036 ***
Xtr1098 nm    1312.9      276.7    4.74  7.2e-05 ***
Xtr1298 nm   -1607.5      368.1   -4.37  0.00019 ***
Xtr1498 nm     229.9       82.1    2.80  0.00968 **
Xtr1698 nm      26.2       10.7    2.46  0.02100 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.116 on 25 degrees of freedom
Multiple R-squared: 0.4807,     Adjusted R-squared: 0.3976
F-statistic: 5.786 on 4 and 25 DF,  p-value: 0.001941
```

The `lm` function automatically fits an intercept; there is no need to explicitly add a column of ones to the matrix of independent variables. Under the usual assumption of normal *iid* residuals, the p-values for the coefficients are gathered in the last column: all coefficients are significant at the $\alpha = 0.05$ level.

8.1.1 Limits of Multiple Regression

Unfortunately, however, there are some drawbacks. In the context of the natural sciences, the most important perhaps is the sensitivity to correlation in the independent variables. This can be illustrated using the following example. Suppose we have a model that looks like this:

$$y = 2 + x_1 + 0.5x_2 - 2x_3$$

and further suppose that x_2 and x_3 are highly correlated ($r \approx 1.0$). This means that any of the following models will give more or less the same predictions:

$$\begin{aligned}
y &= 2 + x_1 - 1.5x_2 \\
y &= 2 + x_1 - 1.5x_3 \\
y &= 2 + x_1 + 5.5x_2 - 7x_3 \\
y &= 2 + x_1 + 1000.5x_2 - 1002x_3
\end{aligned}$$

So what is so bad about that? If all predictions would exactly be the same, not even that much, but in practice there will be differences, especially when new samples are far away from the space covered by the training set. The differences will be larger when coefficients are larger, such as in the last line; from a set of equivalent models, the one with the smallest regression coefficients is to be preferred. Furthermore, confidence intervals for the regression coefficients are based on the assumption of independence, which clearly is violated in this case: any coefficient value for x_2 can be compensated for by x_3, and variances for the x_2 and x_3 coefficients will be infinite. Also in cases where there is less than perfect correlation, we will see more unstable models, in the sense that the variances of the coefficient estimates will get large and predictions less reliable.

To be fair, ordinary multiple regression will not allow you to calculate the model in pathological cases like the above: matrix $\boldsymbol{X}^T\boldsymbol{X}$ will be singular, indicating that infinitely many inverse matrices, and, conversely, many different coefficient vectors, are possible – cf. drawing a straight line through only one point. Another case where the inverse cannot be calculated is the situation where there are more independent variables than samples:

```
> Xtr <- cbind(1, gasoline$NIR[odd,])
> solve(crossprod(Xtr), t(Xtr)) %*% Ytr
```

```
Error in solve.default(crossprod(Xtr), t(Xtr)) :
  system is computationally singular: reciprocal
  condition number = 3.49858e-24
```

This was the primary reason to select four of the variables in the gasoline example in the beginning of this section. Unfortunately, in almost all applications of spectroscopy the number of variables far exceeds the number of samples; the correlations between variables is often high, too.

One possibility to tackle the above problem is to calculate a *pseudoinverse* matrix $\boldsymbol{X}^+$ – such a pseudoinverse, or a generalized inverse, has the property that

$$\boldsymbol{X}\boldsymbol{X}^+ = \mathbf{1} \tag{8.5}$$

and can be applied to non-square matrices as well as square matrices. The most often used variant is the Moore-Penrose pseudoinverse, available in R as function `ginv` in package **MASS**:

```
> Blm <- ginv(Xtr) %*% Ytr
```

The Moore-Penrose inverse uses the singular value decomposition of the data matrix:

$$\boldsymbol{X}^{-1} = \left(\boldsymbol{U}\boldsymbol{D}\boldsymbol{V}^T\right)^{-1} = \boldsymbol{V}\boldsymbol{D}^{-1}\boldsymbol{U}^T \tag{8.6}$$

The trick is to ignore the singular values in $\boldsymbol{D}$ that are zero – in the inverse matrix $\boldsymbol{D}^{-1}$ these will still have a value of zero and the corresponding rows in $\boldsymbol{V}$ and $\boldsymbol{U}$ will be disregarded. In practice, of course, a threshold will have to be used which is usually taken to be dependent on the machine precision. Values smaller than the threshold will be set to zero in the inverse of $\boldsymbol{D}$. Singular values which are slightly larger, however, may exert a large influence on the result, and in many cases the generalized inverse is not very stable.

It is taking this idea one step further to restrict the number of singular values in Equation 8.6 to only the most important principal components. This is the basis of Principal Component Regression (PCR). PCR basically performs a regression on the *scores* of $\boldsymbol{X}$, where a suitable number of latent variables has to be chosen. An alternative, Partial Least Squares (PLS) regression employs the same basic idea, but takes the dependent variable into account when defining scores and loadings, whereas PCR concentrates on capturing variance in X only. In both techniques, the inversion of the covariance matrix is simple because of the orthogonality of the scores. The price we pay is threefold: vital information may be lost because of the data compression, we have to choose the degree of compression, i.e., the number of latent variables, and finally it is no longer possible to derive analytical expressions for the prediction error and the variances of individual regression coefficients. In practice, the latter is not too useful anyway because of the violated assumption of independence in virtually all examples from the natural sciences. To be able to say something about the optimal number of latent variables, and about the expected error of prediction, crossvalidation or similar techniques must be used (see Chapter 9).

We have already seen that in general models with small regression coefficients are to be preferred. It can be shown that PLS as well as PCR actually shrink the regression coefficients towards zero [3]. They are therefore biased methods: the coefficients on average will be smaller in absolute value than the unknown, "true", coefficients – however, this will be compensated for by a much lower variance. Other approaches, based on explicit penalization of the regression coefficients, can be used as well. If a quadratic (L_2) penalty is employed, the result is called *ridge regression*. A penalty in the form of absolute values (an L_1 penalty) leads to the *lasso*, whereas a combination of L_1 and L_2 penalties is known as the *elastic net*. Methods based on the L_1 norm have the advantage that many of the coefficients will have a value of zero, thereby implicitly performing variable selection. They will be treated, along with explicit variable selection methods, in Chapter 10.

8.2 PCR

The prime idea of PCR is to use scores rather than the original data for the regression step. This has two advantages: scores are orthogonal, so there are no problems with correlated variables, and secondly, the number of PCs taken into account usually is much lower than the number of original variables. This reduces the number of coefficients that must be estimated considerably, which in turn leads to more degrees of freedom for the estimation of errors. Of course, we have the added problem that we have to estimate how many PCs to retain.

8.2.1 The Algorithm

For the moment, let us select a PCs; matrices $\boldsymbol{T}$, $\boldsymbol{P}$, etcetera, will have a columns. The regression model is then built using the *reconstructed* matrix $\tilde{\boldsymbol{X}} = \boldsymbol{T}\boldsymbol{P}^T$ rather than the original matrix:

$$\boldsymbol{Y} = \tilde{\boldsymbol{X}}\boldsymbol{B} + \mathcal{E} = \boldsymbol{T}(\boldsymbol{P}^T\boldsymbol{B}) + \mathcal{E} = \boldsymbol{T}\boldsymbol{A} + \mathcal{E} \tag{8.7}$$

$$\boldsymbol{A} = (\boldsymbol{T}^T\boldsymbol{T})^{-1}\boldsymbol{T}^T\boldsymbol{Y} \tag{8.8}$$

where $\boldsymbol{A} = \boldsymbol{P}^T\boldsymbol{B}$ will contain the regression coefficients for the scores. The crossproduct matrix of $\boldsymbol{T}$ is diagonal (remember, $\boldsymbol{T}$ is orthogonal) so can be easily inverted. The regression coefficients for the scores can be back-transformed to coefficients for the original variables:

$$\begin{aligned}\boldsymbol{B} &= \boldsymbol{P}\boldsymbol{A} \\ &= \boldsymbol{P}(\boldsymbol{T}^T\boldsymbol{T})^{-1}\boldsymbol{T}^T\boldsymbol{Y}\end{aligned} \tag{8.9}$$

This can be simplified further by resubstituting $\boldsymbol{T} = \boldsymbol{U}\boldsymbol{D}$ (from the SVD routine):

$$\begin{aligned}\boldsymbol{B} &= \boldsymbol{P}(\boldsymbol{D}\boldsymbol{U}^T\boldsymbol{U}\boldsymbol{D})^{-1}\boldsymbol{D}\boldsymbol{U}^T\boldsymbol{Y} \\ &= \boldsymbol{P}\boldsymbol{D}^{-2}\boldsymbol{D}\boldsymbol{U}^T\boldsymbol{Y} \\ &= \boldsymbol{P}\boldsymbol{D}^{-1}\boldsymbol{U}^T\boldsymbol{Y}\end{aligned} \tag{8.10}$$

In practice, one always performs PCR on a mean-centered data matrix. In Chapter 4 we have seen that without mean-centering the first PC often dominates and is very close to the vector of column means, an undesirable situation. By mean-centering, we explicitly force a regression model without an intercept. The result is that the coefficient vector $\boldsymbol{B}$ does not contain an abscissa vector $\boldsymbol{b}_0$; it should be calculated explicitly by taking the difference between the mean y values and the mean *predicted* y-values.

$$\boldsymbol{b}_0 = \bar{\boldsymbol{y}} - \boldsymbol{X}\boldsymbol{B} \tag{8.11}$$

For every variable in $\boldsymbol{Y}$, we will find one number in $\boldsymbol{b}_0$.

Let's see how this works for the gasoline data. We will model the odd rows, now based on the complete NIR spectra. We start by mean-centering the data, based only on the odd rows:

```
> X <- scale(gasoline$NIR, scale = FALSE,
+              center = colMeans(gasoline$NIR[odd,]))
```

Note that we do not use autoscaling, since that would blow up noise in uninformative variables. Next, we calculate scores and use these as independent variables in a regression. For the moment, we choose five PCs.

```
> Xodd.svd <- svd(X[odd,])
> Xodd.scores <- Xodd.svd$u %*% diag(Xodd.svd$d)
> gasodd.pcr <-
+   lm(gasoline$octane[odd] ~ I(Xodd.scores[,1:5]) - 1)
```

The `- 1` in the formula definition prevents `lm` from fitting an intercept; the other coefficients are not affected, whether an intercept is fitted or not, but removing it makes the comparison below slightly easier. The regression coefficients of the wavelengths, the original variables, are obtained by multiplying the regression coefficients of the scores with the corresponding loadings:

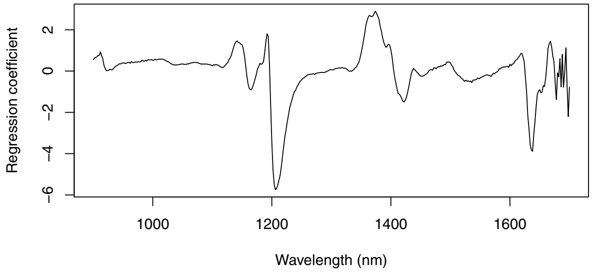

Fig. 8.1. Regression coefficients for the gasoline data (based on the odd rows), obtained by PCR using five PCs.

```
> gasodd.coefs <- coef(gasodd.pcr) %*% t(Xodd.svd$v[,1:5])
```

The same model can be produced by the `pcr` function from the **pls** package [83]:

```
> gasoline.pcr <- pcr(octane ~ ., data = gasoline,
+                     subset = odd, ncomp = 5)
> all.equal(c(coef(gasoline.pcr)), c(gasodd.coefs))

[1] TRUE

> plot(wavelengths, coef(gasoline.pcr), type = "l",
+      xlab = "Wavelength (nm)", ylab = "Regression coefficient")
```

The last line produces the plot of the regression coefficients, shown in Figure 8.1. As usual, the intercept is not visualized. One can clearly see features such as the peaks around 1200 and 1400 nm. Often, such important variables can be related to physical or chemical phenomena.

The model can be summarized by the generic function `summary.mvr`:

```
> summary(gasoline.pcr)

Data:   X dimension: 30 401
        Y dimension: 30 1
Fit method: svdpc
Number of components considered: 5
TRAINING: % variance explained
        1 comps  2 comps  3 comps  4 comps  5 comps
X        74.318    86.26    91.66    96.11    97.32
octane    9.343    11.32    16.98    97.22    97.26
```

Clearly, the first component focuses completely on explaining variation in $\boldsymbol{X}$; it is the fourth component that seems most useful in predicting $\boldsymbol{Y}$, the octance number.

8.2.2 Selecting the Optimal Number of Components

How much variance of $\boldsymbol{Y}$ is explained is one criterion one could use to determine the optimal number of PCs. More often, however, one monitors the (equivalent) root-mean-square error (RMS or RMSE):

$$\text{RMS} = \sqrt{\sum_{i}^{n} (\hat{y}_i - y_i)^2 / n} \tag{8.12}$$

where $\hat{y}_i - y_i$ is the difference between predicted and true value, and n is the number of predictions. A simple R function to find RMS values is the following:

```
> rms <- function(x, y) sqrt(mean((x-y)^2))
```

It is important to realise that both criteria assess the fit of the model to the training data, i.e., the quality of the reproduction rather than the predictive abilities of the model.

The **pls** package comes with an extractor function for RMS estimates:

```
> RMSEP(gasoline.pcr, estimate = "train", intercept = FALSE)

1 comps  2 comps  3 comps  4 comps  5 comps
 1.3467   1.3319   1.2887   0.2358   0.2343
```

The `intercept = FALSE` argument prevents the RMS error based on nothing but the average of the dependent variable to be printed. The error when using *only* the fourth PC in the regression is given by

```
> RMSEP(gasoline.pcr, estimate = "train", comp = 4)

[1]  0.6287
```

It is only half the size of the error of prediction using PCs one to three, again confirming that the fourth PC is the dominant one in the prediction model. Nevertheless, the combination of the fourth with the first three components leads to a significant improvement. Adding the fifth does not seem worthwhile. Compare these numbers with the MLR predictions based on only four wavelengths:

```
> rms(Ytr - fitted(Blm))

[1] 1.019
```

The first three PCs apparently capture less relevant information for predicting the octane numbers than four randomly selected wavelengths!

One should be very cautious in interpreting these numbers – reproduction of the training set is not a reliable way to assess prediction accuracy, which is what we really are after. For one thing, adding another component will

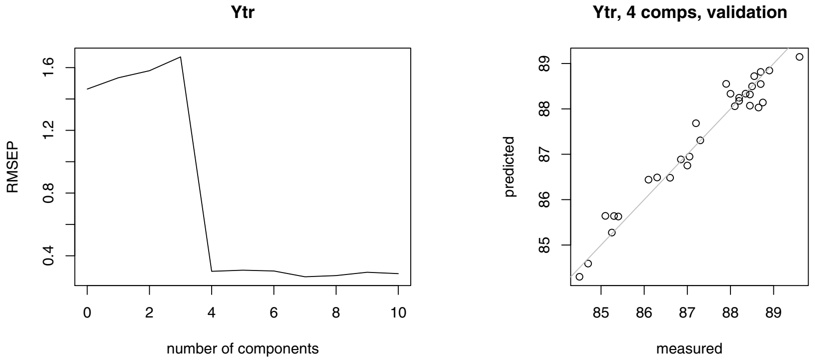

Fig. 8.2. Validation plot for PCR regression on the gasoline data (left) and prediction quality of the optimal model, containing four PCs (right).

(by definition) always decrease the error. Predictions for new data, however, may not be as good: the model often is too focussed on the training data, the phenomenon known as overfitting that we also saw in Section 7.3.1.

A major difficulty in PCR modelling is therefore the choice of the optimal number of PCs to retain. Usually, crossvalidation is employed to estimate this number: if the number of PCs is too small, we will incorporate too little information, and our model will not be very useful – the crossvalidation error will be high. If the number of PCs is too large, the training set will be reproduced very well, but again, the crossvalidation error will be large because the model is not able to provide good predictions for the left-out samples. So the strategy is simple: we perform the regression for several numbers of PCs, calculate the crossvalidation errors, put the results in a graph, *et voilà*, we can pick the number we like best.

In LOO crossvalidation, n different models are created, each time omitting another sample y_{-i} from the training data. Equation 8.12 then is used to calculate an RMS estimate. Alternatively, a group of samples is left out simultaneously. One often sees the name RMSCV or RMSECV, to indicate that it is the RMS value derived from a crossvalidation procedure. Of course, this procedure can be used in other contexts as well; the RMSEP usually is associated with prediction of unseen test data, RMSEV is an error estimate from some form of validation, and RMSEC is the calibration error, indicating how well the model fits the training data. The RMSEC value is in almost all cases the lowest; it measures how well the model represents the data on which it is based. The RMSEP is an estimate of the thing that really interests us, the error of prediction, but it can only reliably be calculated directly when large training and test sets are available and the training data are representative for the test data. In practice, RMSE(C)V estimates are the best we can do.

In the **pls** package, the function `RMSEP` is used to calculate all these quantities – it takes an argument `estimate` which can take the values `"train"` (used in the example above), `"CV"` and `"test"` for calibration, crossvalidation and test set estimates, respectively. The `pcr` function has an optional argument for crossvalidation: `validation` can be either `"none"`, `"LOO"`, or `"CV"`. In the latter case, 10 segments are used in the crossvalidation by default ("leave 10% out"). Application to the gasoline data leads to the following results:

```
> gasoline.pcr <- pcr(octane ~ ., data = gasoline, subset = odd,
+                     validation = "LOO", ncomp = 10)
> plot(gasoline.pcr, "validation", estimate = "CV")
```

This leads to the left plot in Figure 8.2, showing the RMSECV estimate[1] against the number of PCs. Not unexpectedly, four PCs clearly are a very good compromise between model complexity and predictive power. For the argument `estimate`, one can also choose `"adjCV"`, which is a bias-corrected error estimate [84].

Zooming in on the plot would show that the absolute minimum is at seven PCs, and perhaps taking more than ten PCs into consideration would lead to an lower global minimum. However, one should keep in mind that an RMS value is just an estimate with an associated variance, and differences are not always significant. Moreover, the chance of overfitting increases with a higher number of components. Numerical values are accessible in the `validation` list element of the fitted object. The RMS values plotted in Figure 8.2 can be assessed as follows:

```
> RMSEP(gasoline.pcr, estimate = "CV")
```

```
(Intercept)    1 comps    2 comps    3 comps    4 comps    5 comps
     1.4631     1.5351     1.5802     1.6682     0.3010     0.3082
    6 comps    7 comps    8 comps    9 comps   10 comps
     0.3031     0.2661     0.2738     0.2949     0.2861
```

which is the same as

```
> sqrt(gasoline.pcr$validation$PRESS / nrow(Xtr))
```

The quality of the four-component model can be assessed by visualization: a very common plot shows the true values of the dependent variable on the x-axis and the predictions on the y-axis. We use a square plot where both axes have the same range so that an ideal prediction would lie on a line with a slope of 45 degrees.

```
> par(pty = "s")
> plot(gasoline.pcr, "prediction", ncomp = 4)
```

[1] Note that the axis uses the term "RMSEP" which sometimes is used in a more general sense – whether it deals with crossvalidation or true predictions must be deduced from the context.

Each point in the right plot of Figure 8.2 is the prediction for that point when it was not part of the training set. Therefore, this plot gives some idea of what to expect for unseen data. Note, however, that this particular number of PCs was chosen with the explicit aim to minimise errors for these data points – the LOO crossvalidation was used to assess the optimal number of PCs. The corresponding error estimate is therefore optimistically biased, and we need another way of truly assessing the expected error for future observations.

This is given, for example, by the performance of the model on an unseen test set:

```
> gasoline.pcr.pred <- predict(gasoline.pcr, ncomp = 4,
+                                newdata = gasoline[even,])
> rms(gasoline$octane[even], gasoline.pcr.pred)
```

```
[1] 0.2101745
```

or, by using the RMSEP function again:

```
> RMSEP(gasoline.pcr, ncomp = 4, newdata = gasoline[even,],
+       intercept = FALSE)
```

```
[1]  0.2102
```

The error on the test set is 0.21, which is smaller than the crossvalidated error on the training set with four components (0.3010). Although this may seem surprising at first, it is again the result of the fact that LOO error estimates, although unbiased, have a large variance [66]. We will come back to this point in Chapter 9.

8.3 Partial Least Squares (PLS) Regression

In PCR, the information in the independent variables is summarized in a small number of principal components. However, there is no *a priori* reason why the PCs associated with the largest singular values should be most useful for regression. PC 1 covers the largest variance, but still may have only limited predictive power, as we have seen in the gasoline example. Since we routinely pick PCs starting from number 1 and going up, there is a real chance that we include variables that actually do not contribute to the regression model. Put differently: we compress information in $\boldsymbol{X}$ without regard to what is to be predicted, so we can never be sure that the essential part of the data is preserved. Although it has been claimed that *selecting* specific PCs (e.g. nrs 2, 5 and 6) on which to base the regression, rather than a sequence of PCs starting from one, leads to better models (see, e.g., [85]), this only increases the difficulties one faces: selection is a much more difficult process than determining a threshold.

PLS forms an alternative. Just like PCR, PLS defines orthogonal latent variables to compress the information and throw away irrelevant stuff. However, PLS explicitly aims to construct latent variables in such a way as to capture most variance in $\boldsymbol{X}$ and $\boldsymbol{Y}$, *and* to maximize the correlation between these matrices. Put differently: it maximizes the *covariance* between $\boldsymbol{X}$ and $\boldsymbol{Y}$. So it seems we keep all the advantages, and get rid of the less desirable aspects of PCR. The algorithm is a bit more complicated than PCR; in fact, there exist several almost equivalent algorithms to perform PLS. The differences are caused by either small variations in the criterion that is optimized, different implementations to obtain speed improvements in specific situations, or by different choices for scaling intermediate results.

8.3.1 The Algorithm(s)

Just as in PCR, in PLS it is customary to perform mean-centering of the data so that there is no need to estimate an intercept vector; this is obtained afterwards. The notation is a bit more complicated than with PCR, as already mentioned, since now both $\boldsymbol{X}$ and $\boldsymbol{Y}$ matrices have scores and loadings. Moreover, in many algorithms one employs additional weight matrices. One other difference with PCR is that the components of PLS are extracted sequentially whereas the PCs in PCR can be obtained in one SVD step. In each iteration in the PLS algorithm, the variation associated with the estimated component is removed from the data in a process called deflation, and the remainder (indicated with $\boldsymbol{E}$ for the "deflated" $\boldsymbol{X}$ matrix, and $\boldsymbol{F}$ for the deflated $\boldsymbol{Y}$) is used to estimate the next component. This continues until the user decides it has been enough, or until all components have been estimated.

The first component is obtained from an SVD of the crossproduct matrix $\boldsymbol{S} = \boldsymbol{X}^T\boldsymbol{Y}$, thereby including information on both variation in $\boldsymbol{X}$ and $\boldsymbol{Y}$, and on the correlation between both. The first left singular vector, $\boldsymbol{w}$, can be seen as the direction of maximal variance in the crossproduct matrix, and is usually indicated with the somewhat vague description of "weights". The projections of matrix $\boldsymbol{X}$ on this vector are called "X scores":

$$\boldsymbol{t} = \boldsymbol{X}\boldsymbol{w} = \boldsymbol{E}\boldsymbol{w} \tag{8.13}$$

Eventually, these scores t will be gathered in a matrix $\boldsymbol{T}$ that fulfills the same role as the score matrix in PCR; it is a low-dimensional, full-rank, estimate of the information in $\boldsymbol{X}$. Therefore, regressing $\boldsymbol{Y}$ on $\boldsymbol{T}$ is easy, and the coefficient vector for $\boldsymbol{T}$ can be converted to a coefficient vector for the original variables.

The next step in the algorithm is to obtain loadings for $\boldsymbol{X}$ and $\boldsymbol{Y}$ by regressing against the *same* score vector $\boldsymbol{t}$:

$$\boldsymbol{p} = \boldsymbol{E}^T\boldsymbol{t}/(\boldsymbol{t}^T\boldsymbol{t}) \tag{8.14}$$

$$\boldsymbol{q} = \boldsymbol{F}^T\boldsymbol{t}/(\boldsymbol{t}^T\boldsymbol{t}) \tag{8.15}$$

Notice that one divides by the sum of all squared elements in $\boldsymbol{t}$: this leads to "normalized" loadings. It is not essential that the scaling is done in this way. In fact, there are numerous possibilities to scale either loadings, weights, or scores – one can choose to have either the scores or the loadings orthogonal. Unfortunately, this can make it difficult to compare the scores and loadings of different PLS implementations. The current description is analogous to PCR where the loadings are taken to have unit variance.

Finally, the data matrices are deflated: the information related to this latent variable, in the form of the outer products $\boldsymbol{t}\boldsymbol{p}^T$ and $\boldsymbol{t}\boldsymbol{q}^T$, is subtracted from the (current) data matrices.

$$\boldsymbol{E}_{n+1} = \boldsymbol{E}_n - \boldsymbol{t}\boldsymbol{p}^T \tag{8.16}$$

$$\boldsymbol{F}_{n+1} = \boldsymbol{F}_n - \boldsymbol{t}\boldsymbol{q}^T \tag{8.17}$$

The estimation of the next component then can start from the SVD of the crossproduct matrix $\boldsymbol{E}_{n+1}^T \boldsymbol{F}_{n+1}$. After every iteration, vectors $\boldsymbol{w}$, $\boldsymbol{t}$, $\boldsymbol{p}$ and $\boldsymbol{q}$ are saved as columns in matrices $\boldsymbol{W}$, $\boldsymbol{T}$, $\boldsymbol{P}$ and $\boldsymbol{Q}$, respectively.

In words, the algorithm can be summarized as follows: the vectors $\boldsymbol{w}$ constitute the direction of most variation in the crossproduct matrix $\boldsymbol{X}^T\boldsymbol{Y}$. The scores t are the coordinates of the objects on this axis. Loadings for $\boldsymbol{X}$ and $\boldsymbol{Y}$ are obtained by regressing both matrices against the scores, and the products of the scores and loadings for $\boldsymbol{X}$ and $\boldsymbol{Y}$ are removed from data matrices $\boldsymbol{E}$ and $\boldsymbol{F}$.

One complication is that columns of matrix $\boldsymbol{W}$ can not be compared directly: they are derived from successively deflated matrices $\boldsymbol{E}$ and $\boldsymbol{F}$. An alternative way to represent the weights, in such a way that all columns relate to the original $\boldsymbol{X}$ matrix, is given by

$$\boldsymbol{R} = \boldsymbol{W}(\boldsymbol{P}^T\boldsymbol{W})^{-1} \tag{8.18}$$

Matrix $\boldsymbol{R}$ has some interesting properties, one of which is that it is a generalized inverse for $\boldsymbol{P}^T$. It also holds that $\boldsymbol{T} = \boldsymbol{X}\boldsymbol{R}$. For interpretation purposes, one sometimes also calculates so-called y-scores $\boldsymbol{U} = \boldsymbol{Y}\boldsymbol{Q}$. Alternatively, these y-scores can be obtained as the right singular vectors of $\boldsymbol{E}^T\boldsymbol{F}$.

Now, we are in the same position as in the PCR case: instead of regressing $\boldsymbol{Y}$ on $\boldsymbol{X}$, we use scores $\boldsymbol{T}$ to calculate the regression coefficients $\boldsymbol{A}$, and later convert these back to the realm of the original variables:

$$\boldsymbol{Y} = \tilde{\boldsymbol{X}}\boldsymbol{B} + \mathcal{E} = \boldsymbol{T}(\boldsymbol{P}^T\boldsymbol{B}) + \mathcal{E} = \boldsymbol{T}\boldsymbol{A} + \mathcal{E} \tag{8.19}$$

$$\boldsymbol{A} = (\boldsymbol{T}^T\boldsymbol{T})^{-1}\boldsymbol{T}^T\boldsymbol{Y} \tag{8.20}$$

$$\boldsymbol{B} = \boldsymbol{R}\boldsymbol{A} \tag{8.21}$$

These equations are almost identical with the PCR algorithm presented in Equations 8.7 and 8.8. The difference lies first and foremost in the calculation of $\boldsymbol{T}$, which now includes information on $\boldsymbol{Y}$, and in the calculation of the regression coefficients for the original variables, where PLS uses $\boldsymbol{R}$ rather than $\boldsymbol{P}$. Again, the singularity problem is solved by using a low-dimensional score matrix $\boldsymbol{T}$ of full rank. The coefficient for the abscissa is obtained in the same way as with PCR (Equation 8.11).

In the **pls** package, PLS regression is available as function `plsr`:

```
> gasoline.pls <- plsr(octane ~ ., data = gasoline,
+                      subset = odd, ncomp = 5)
> summary(gasoline.pls)

Data:   X dimension: 30 401
        Y dimension: 30 1
Fit method: kernelpls
Number of components considered: 5
TRAINING: % variance explained
     1 comps  2 comps  3 comps  4 comps  5 comps
X      71.71    79.70    90.71    95.70    96.59
Ytr    22.82    93.93    97.49    97.79    98.74
```

Clearly, the first components of the PLS model explain much more variation in $\boldsymbol{Y}$ than the corresponding PCR model: the first two PLS components already cover almost 94%, whereas the first two PCR components barely exceed ten percent. The price to be paid lies in the description of the $\boldsymbol{X}$ data: the two-component PCR model explains seven percent more than the corresponding PLS model.

To assess how many components are needed, the `validation` argument can be used, in the same way as with the `pcr` function:

```
> gasoline.pls <- plsr(octane ~ ., data = gasoline, subset = odd,
+                      validation = "LOO", ncomp = 10)
> par(mfrow = c(1,2))
> plot(gasoline.pls, "validation", estimate = "CV")
> par(pty = "s")
> plot(gasoline.pls, "prediction", ncomp = 3)
> abline(0, 1, col = "gray")
```

The resulting plots, shown in Figure 8.3, indicate that a PLS model comparable to the four-component PCR model from Figure 8.2 only needs three latent variables. This difference between PLS and PCR is often observed in practice: PLS models typically need one or two fewer components than PCR models to achieve similar CV error estimates. Let's check the predictions of the unseen data, the even rows of the data frame:

```
> RMSEP(gasoline.pls, ncomp = 3, newdata = gasoline[even,],
+       intercept = FALSE)

[1]  0.2093
```

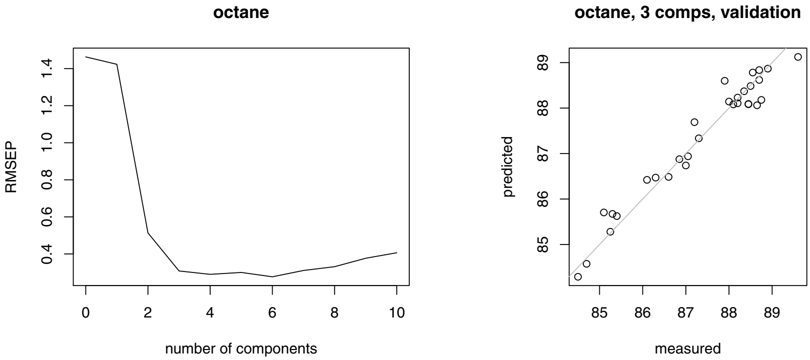

Fig. 8.3. Validation plot for PLS regression on the gasoline data (left) and prediction quality of the optimal model, containing three PLS components (right).

Indeed, with one component less the PLS model achieves a prediction quality that is as good as that of the PCR model.

The `plsr` function takes a `method` argument to specify which PLS algorithm is to be used. The default is `kernelpls` [86], a very fast and stable algorithm which gives results equal to the original NIPALS algorithm [87] which is available as `oscorespls`. The kernel algorithm performs SVD on crossproduct matrix $\boldsymbol{X}^T\boldsymbol{Y}\boldsymbol{Y}^T\boldsymbol{X}$ rather than $\boldsymbol{X}^T\boldsymbol{Y}$, and avoids deflation of $\boldsymbol{Y}$. In cases with large numbers of variables (tens of thousands), a variant called `widekernelpls` [88] is more appropriate – again, it operates by constructing a smaller kernel matrix, this time $\boldsymbol{X}\boldsymbol{X}^T\boldsymbol{Y}\boldsymbol{Y}^T$, on which to perform the SVD operations. However, it is numerically less stable than the default algorithm. Also the `widekernelpls` algorithm gives results that are identical (upon convergence) to the NIPALS results.

One popular alternative formulation, SIMPLS [89], deflates matrix $\boldsymbol{S}$ rather than matrices $\boldsymbol{E}$ and $\boldsymbol{F}$ individually. It can be shown that SIMPLS actually maximizes the covariance between $\boldsymbol{X}$ and $\boldsymbol{Y}$ (which is usually taken as "the" PLS criterion), whereas the other algorithms are good approximations; for univariate $\boldsymbol{Y}$, SIMPLS predictions are equal to the results from NIPALS and kernel algorithms. However, for multivariate $\boldsymbol{Y}$, there may be (minor) differences between the approaches. SIMPLS can be invoked by providing the `method = "simpls"` argument to the `plsr` function. In all these variants, the scores will be orthogonal, whereas the loadings are not:

```
> cor(gasoline.pls$loadings[,1:3])

         Comp 1   Comp 2   Comp 3
Comp 1  1.00000 -0.55329 -0.07528
Comp 2 -0.55329  1.00000 -0.06226
Comp 3 -0.07528 -0.06226  1.00000
```

```
> cor(gasoline.pls$scores[,1:3])
```

```
          Comp 1    Comp 2    Comp 3
Comp 1 1.000e+00 6.704e-17 2.113e-17
Comp 2 6.704e-17 1.000e+00 1.789e-16
Comp 3 2.113e-17 1.789e-16 1.000e+00
```

Given that one has a certain freedom to decide where exactly in the algorithm to normalize, the outcome of different implementations, and in particular, in different software packages, may seem to vary significantly. However, the regression coefficients, and therefore the predictions, of all these are usually virtually identical. For all practical purposes there is no reason to prefer the outcome of one algorithm over another.

8.3.2 Interpretation

PLS models give separate scores and loadings for $\boldsymbol{X}$ and $\boldsymbol{Y}$, and additionally, in most implementations, some form of a weight matrix. In most cases, one concentrates on the matrix of regression coefficients $\boldsymbol{B}$ which is independent of algorithmic details such as the exact way of normalization of weights, scores and loadings, and is directly comparable to regression coefficients from other methods like PCR. Sometimes, plots of weights, loadings or scores of individual components can be informative, too, although one should be careful not to overinterpret: there is no a priori reason to assume that the individual components directly correspond to chemically interpretable entities [90].

The interpretation of the scores and loadings is similar to PCA: a score indicates how much a particular object contributes to a latent variable, while a loading indicates the contribution of a particular *variable*. An example of a loading plot is obtained using the code below:

```
> plot(gasoline.pls, "loading", comps = 1:3, legendpos = "top",
+      lty = c(1, 2, 4), col = c(1, 2, 4))
```

This leads to Figure 8.4. The percentage shown in the legend corresponds with the variation explained of the $\boldsymbol{X}$ matrix for each latent variable. Note that the third component explains more variation of $\boldsymbol{X}$ than the second; in a PCR model this would be impossible[2]. Components one and two show spectrum-like shapes, with the largest values at the locations of the main features in the data, as expected – the third component is focussing very much on the last ten data points. This raises questions on the validity of the model: it is doubtful that these few (and noisy) wavelengths should play a major part. Perhaps a more prudent choice for the number of latent variables from Figure 8.3 would have been to use only two.

[2] The `plot.mvr` function can be applied to PCR models as well as PLS models, so the discussion in this paragraph pertains to both.

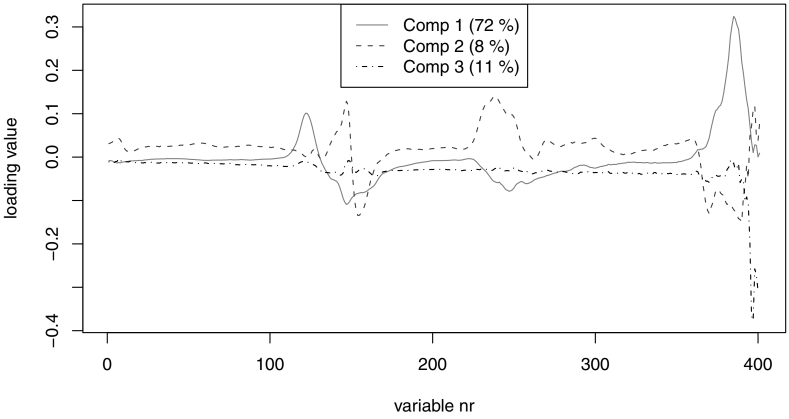

Fig. 8.4. PLS loadings for the first three latent variables for the gasoline data; the third component has large loadings only for the last ten variables.

Biplots, showing the relations between scores and loadings, can be made using the function `biplot.mvr`. One can in this way inspect the influence of individual PLS components, where the regression coefficient matrix $\boldsymbol{B}$ gives a more global view summarizing the influence of all PLS components. The argument `which`, taking the values `"x"`, `"y"`, `"scores"` and `"loadings"`, indicates what type of biplot is required. In the first case, the scores and loadings for x are shown in a biplot; the second case does the same for y. The other two options show combinations of x- and y- scores and loadings, respectively. In the current example, y-loadings are not very interesting since there is only one y variable, octane. An additional source of information is the relation between the X-scores, $\boldsymbol{T}$, and the Y-scores, $\boldsymbol{U}$. For the gasoline model these plots are shown for the first two latent variables in Figure 8.5 using the following code:

```
> plot(scores(gasoline.pls)[,1], Yscores(gasoline.pls)[,1],
+      xlab = "X scores", ylab = "Y scores", main = "LV 1")
> abline(h = 0, v = 0, col = "gray")
> plot(scores(gasoline.pls)[,2], Yscores(gasoline.pls)[,2],
+      xlab = "X scores", ylab = "Y scores", main = "LV 2")
> abline(h = 0, v = 0, col = "gray")
```

Usually one hopes to see a linear relation, as is the case for the second latent variable; the first LV shows a less linear behaviour. One could replace the linear regression in Eqs. 8.14 and 8.15 by a polynomial regression (using columns of powers of t), or even a non-linear regression. There are, however, not many reports where this has led to significant improvements; see for example [91,92].

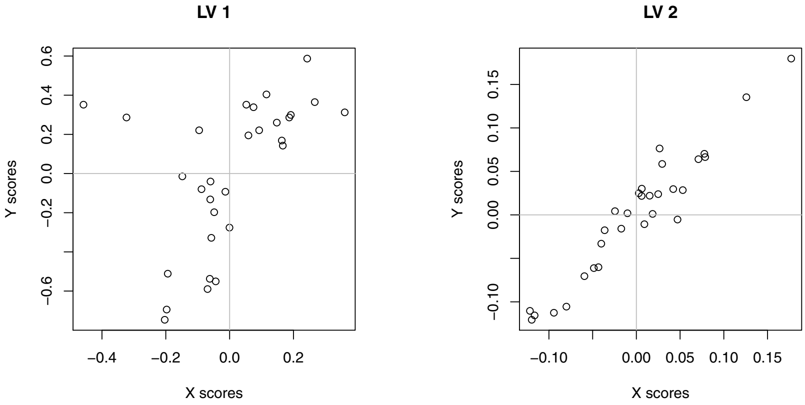

Fig. 8.5. Relation between X-scores and Y-scores, $\boldsymbol{T}$ and $\boldsymbol{U}$, respectively, for the first two latent variables (gasoline data).

For multivariate $\boldsymbol{Y}$, there is an additional difference between PLS and PCR. With PCR, separate models are fit for each $\boldsymbol{Y}$ variable: the algorithm does not try to make use of any correlation between the separate dependent variables. With PLS, this is different. Of course, one can fit separate PLS models for each $\boldsymbol{Y}$ variable (this is often indicated with the acronym PLS1), but one can also do it all in one go (PLS2). In that case, the same $\boldsymbol{X}$-scores $\boldsymbol{T}$ are used for *all* dependent variables; although a separate set of regression coefficients will be generated for every y-variable, implicitly information from the other dependent variables is taken into account. This can be an advantage, especially when there is appreciable correlation between the y-variables, but in practice there is often little difference between the two approaches. Most people prefer multiple PLS1 models (analogous to PCR regression) since they seem to give slightly better fits.

It is no coincidence that chemistry has been the first area in which PLS was really heavily used. Analytical chemists had been measuring spectra and trying to relate them to chemical properties for years, when this regression technique finally provided them with the tool to do so. Other disciplines such as statistics were slow to follow, but eventually PLS has found its place in a wide range of fields. Also the theoretical background has now been clarified: PLS started out as an algorithm that was really poorly understood. Nowadays, there are very few people who dispute that PLS is an extremely useful method, but it has been overhyped somewhat. In practical applications, its performance is very similar to techniques like PCR. Careful thinking of experimental design, perhaps variable selection, and appropriate preprocessing of the data is likely to be far more important than the exact choice of multivariate regression technique.

PLS Packages for R

Apart from the **pls** package, several other packages provide PLS functions, both for regression and classification. The list is large and rapidly expanding; examples include the packages **lspls** implementing Least-Squares PLS [93], and **gpls** [94] implementing generalized partial least squares, based on the Iteratively ReWeighted Least Squares (IRWLS) method [95]. Weighted-average PLS [96], often used in paleolimnology, can be found in package **paltran**. Package **plsgenomics** implements methods for classification with microarray data and prediction of transcription factor activities from combined ChIP-chip analysis, combining a.o. ridge regression with PLS. This package also provides function `pls-lda` performing LDA on PLS scores (see Section 11.3.2). Package **plspm** contains, in addition to the usual functions for PLS regression, also functions for Path Modelling [97]. Penalized PLS [98] is available in package **ppls**, and sparse PLS, forcing small loadings to become zero so that fewer variables are taking part in the model, in package **spls** [99]. Which of these packages is most suited depends on the application.

8.4 Ridge Regression

Ridge Regression (RR) [100,101] is another way to tackle regression problems with singular covariance matrices, usually for univariate $\boldsymbol{Y}$. From all possible regression models giving identical predictions for the data at hand, RR selects the one with the smallest coefficients. This loss function is implemented by posing a (quadratic) penalty on the size of the coefficients $\boldsymbol{B}$:

$$\underset{\boldsymbol{B}}{\arg\max}\, (\boldsymbol{Y} - \boldsymbol{X}\boldsymbol{B})^2 + \lambda \boldsymbol{B}^T \boldsymbol{B} \tag{8.22}$$

The solution is given by

$$\hat{\boldsymbol{B}} = \left(\boldsymbol{X}^T \boldsymbol{X} + \lambda \boldsymbol{I}\right)^{-1} \boldsymbol{X}^T \boldsymbol{Y} \tag{8.23}$$

Compared to Equation 8.2, a constant is added to the diagonal of the crossproduct matrix $\boldsymbol{X}^T\boldsymbol{X}$, which makes it non-singular. The size of λ is something that has to be determined (see below). This *shrinkage* property has obvious advantages in the case of collinearities: even if the RR model is not quite correct, it will not lead to wildly inaccurate predictions (which may happen when some coefficients are very large). Usually, the intercept is not included in the penalization: one would expect that adding a constant c to the y-values would lead to predictions that are exactly the same amount larger. If the intercept would be penalized as well, this would not be the case.

Optimal values for λ may be determined by crossvalidation (or variants thereof). Several other, direct, estimates of optimal values for λ have been proposed. Hoerl and Kennard [101] use the ratio of the residual variance s^2, estimated from the model, and the largest regression coefficient:

$$\hat{\lambda}_{HK} = \frac{s^2}{\max(\boldsymbol{B}_i^2)} \tag{8.24}$$

A better estimate is formed by using the harmonic means of the regression coefficients rather than the largest value. This is known as the Hoerl-Kennard-Baldwin estimate [102]:

$$\hat{\lambda}_{HKB} = \frac{ps^2}{\boldsymbol{B}^T\boldsymbol{B}} \tag{8.25}$$

where p, as usual, indicates the number of columns in $\boldsymbol{X}$. A variance-weighted version of the latter is given by the Lawless-Wang estimate [103]:

$$\hat{\lambda}_{LW} = \frac{ps^2}{\boldsymbol{B}^T\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{B}} \tag{8.26}$$

Since PCR and PLS can also be viewed as shrinkage methods [90,3], there are interesting links with ridge regression. All three shrink the regression coefficients away from directions of low variation. Ridge regression can be shown to be equivalent to PCR with shrunken eigenvalues for the principal components; PCR uses a hard threshold to select which PCs to take into account. PLS also shrinks – it usually takes the middle ground between PCR and RR. However, in some cases PLS coefficients may be inflated, which may lead to slightly worse performance [3]. In practice, all three methods lead to very similar results. Another common feature is that just like PCR and PLS, ridge regression is not affine equivariant: it is sensitive to different (linear) scalings of the input. In many cases, autoscaling is applied by default. It is also hard-coded in the `lm.ridge` function in package **MASS**. Unfortunately, for many types of spectroscopic data autoscaling is not very appropriate, as we have seen earlier. For the gasoline data, it does not work very well either:

```
> gasoline.ridge <-
+   lm.ridge(octane ~ NIR, data = gasoline, subset = odd,
+            lambda = seq(0.001, 0.1, by = 0.01))
> select(gasoline.ridge)

modified HKB estimator is -6.695607e-28
modified L-W estimator is -3.908617e-28
smallest value of GCV  at 0.001
```

Both the HKB estimate and the L-W estimate suggest a very small value of λ; the generalized crossvalidation (see Chapter 9) suggests the smallest value of λ is the best.

Apart from the links with PCR and PLS, ridge regression is also closely related to SVMs when seen in the context of regression – we will come back to this in Section 8.6.1. Related methods using L_1 penalization, such as the lasso and the elastic net, will be treated in more detail in Section 10.2.

8.5 Continuum Methods

In many ways, MLR and PCR form the opposite ends of a scale. In PCR, the stress is on summarizing the variance in $\boldsymbol{X}$; the correlation with the property that is to be predicted is not taken into account in defining the latent variables. With MLR the opposite is true: one does not care how much information in $\boldsymbol{X}$ is actually used as long as the predictions are OK. PLS takes a middle ground with the criterion that latent variables should explain as much of the covariance between $\boldsymbol{X}$ and $\boldsymbol{Y}$ as possible.

There have been attempts to create regression methods that offer other intermediate positions, most notably Continuum Regression (CR, [104]) and Principal Covariates Regression (PCovR, [105]). Althought they are interesting from a theoretical viewpoint, in practice they have never caught on. One possible explanation is that there is little gain in yet another form of multivariate regression, where methods like PCR, PLS and RR already exist and in most cases give very similar results. Moreover, these continuum methods provide additional crossvalidation problems because more often than not an extra parameter needs to be set.

8.6 Some Non-Linear Regression Techniques

Many non-linear techniques are available for regression. Here, we will very briefly focus on two classes of methods that we have already seen in Chapter 7 on classification, SVMs and neural networks. Both can be adapted to continuous output without much trouble.

8.6.1 SVMs for Regression

In order not to get lost in mathematical details that would be out of context in this book, only the rough contours of the use of SVMs in regression problems are sketched here. A more throrough treatment can be found in the literature (e.g., [3]). Typically, SVMs tackle linear regression problems by minimization of a loss function of the following form:

$$L_{\mathrm{SVM}} = \sum_i V(y_i - f(x_i)) + \lambda ||\beta||^2 \tag{8.27}$$

where the term $V(y_i - f(x_i))$ corresponds to an error function describing the differences between experimental and fitted values, and the second term is a regularization term, keeping the size of the coefficients small. If the error function V is taken to be the usual squared error then this formulation is equal to ridge regression, but more often other forms are used. Two typical examples are shown in Figure 8.6: the left panel shows a so-called ε-insensitive error function, where only errors larger than a cut-off ε are taken into account

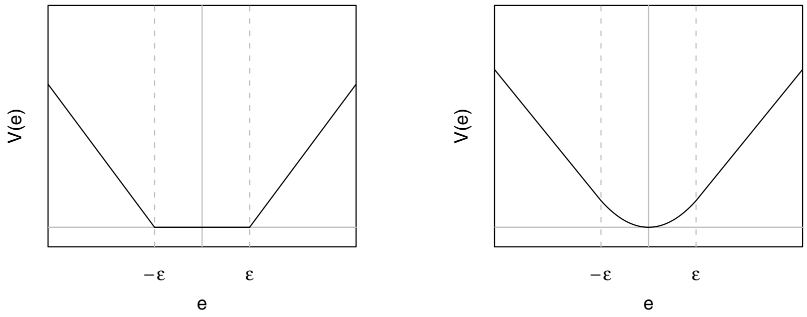

Fig. 8.6. Typical error functions for SVMs in a regression setting: left, the ε-insensitive error function; right, the Huber function.

(linearly), and the right plot shows the Huber loss function, which is quadratic up to a certain value c and linear above that value. The reason to use these functions is that the linear error function for larger errors leads to more robust behaviour. Moreover, the solution of the loss function in Equation 8.27 can be formulated in terms of inner products, just like in the classification case, and again only a subset of all coefficients (the support vectors) are non-zero. Moreover, kernels can be used to find simple linear relationships in high dimensions which after back-transformation represent much more complex patterns.

Let us concentrate on how to use the svm function, seen earlier, in regression problems. We again take the gasoline data, and fit a model, for the moment using the default settings:

```
> gasoline.svm <- svm(octane ~ ., data = gasoline,
+                      subset = odd, cross = 10)
```

Note that the svm function by default performs autoscaling on both X and y, which for the gasoline data is not optimal for reasons discussed earlier. The next piece of code shows the predictions for the training set (recognition, not the crossvalidated predictions), and the test set:

```
> plot(gasoline$octane[odd], predict(gasoline.svm),
+      main = "Training set", xlab = "Octane number (true)",
+      ylab = "Octane number (predicted)")
> abline(0, 1)
> plot(gasoline$octane[even],
+      predict(gasoline.svm, new = gasoline[even,]),
+      main = "Test set", xlab = "Octane number (true)",
+      ylab = "Octane number (predicted)")
> abline(0, 1)
```

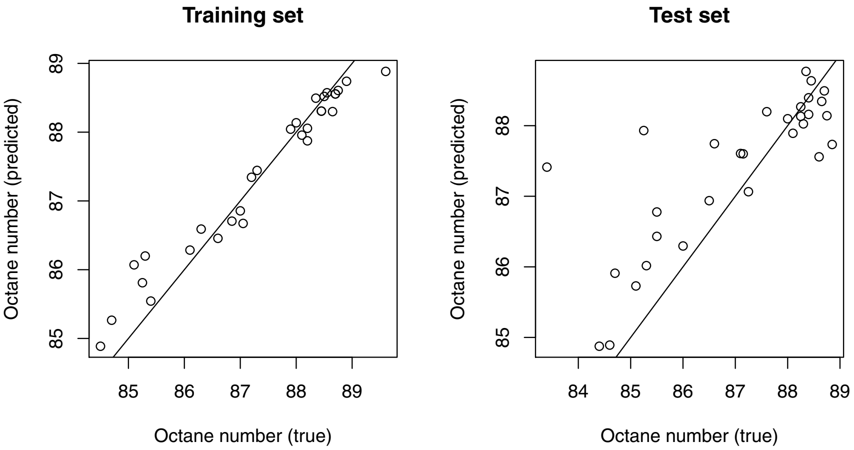

Fig. 8.7. Predictions for the training data (left) and the test data (right) using the default values of the `svm` function.

The result, shown in Figure 8.7, tells us that the training data are predicted quite well (although there seems to be a slight tendancy to predict too close to the mean), but the predictions for the test data show large errors. Clearly, the model is not able to generalize. Applying `summary` to the fitted `gasoline.svm` object shows that the mean squared error of crossvalidation equals 0.505, which corresponds to an RMS value of 0.711, quite a lot higher than we have seen with some other methods. This is a clear example of overfitting.

In this kind of situation, one should consider the parameters of the method. The default behaviour of the `svm` function is to use the ε-insensitive error function, with $\varepsilon = .1$, a value of 1 for the penalization factor in Equation 8.27, and a gaussian (“radial basis”) kernel, which also has some parameters to tune. Since in this case already many variables are available for the SVM, it makes sense to use a less flexible kernel. Indeed, swapping the radial basis kernel for a linear one makes a big difference for the prediction of the test data:

```
> gasoline.svm <- svm(octane ~ ., data = gasoline,
+                     subset = odd, kernel = "linear")
> rms(gasoline$octane[even],
+     predict(gasoline.svm, new = gasoline[even,]))
```

```
[1] 0.2642303
```

Optimization of the parameters, for instance using `tune.svm`, should lead to further improvements.

8.6.2 ANNs for Regression

The basic structure of a backpropagation network as shown in Figure 7.13 remains unchanged for regression applications: the numbers of input and output units equal the number of independent and dependent variables, respectively. Again, the number of hidden units is subject to optimization. In high-dimensional cases one practical difficulty needs to be solved – the data need to be compressed, otherwise the number of weights would be too high. In many practical applications, PCA is performed on the input matrix and the network is trained on the scores.

Let's see how that works out for the gasoline data. We will use the scores of the first five PCs, and fit a neural network model with five hidden units. To indicate that we want numerical output rather than class output, we set the argument `linout` to `TRUE` (the default is use logistic output units):

```
> X <- scale(gasoline$NIR, scale = FALSE,
+            center = colMeans(gasoline$NIR[odd,]))
> Xodd.svd <- svd(X[odd,])
> Xodd.scores <- Xodd.svd$u %*% diag(Xodd.svd$d)
> Xeven.scores <- X[even,] %*% Xodd.svd$v
> gas.nnet <- nnet(Xodd.scores[,1:5],
+                  matrix(gasoline$octane[odd], ncol = 1),
+                  size = 5, linout = TRUE)
```

```
# weights:  36
initial  value 230114.767351
iter  10 value 2.873687
iter  20 value 1.599885
iter  30 value 1.557083
iter  40 value 1.541940
iter  50 value 1.508819
iter  60 value 1.463200
iter  70 value 1.329733
iter  80 value 1.234174
iter  90 value 1.196288
iter 100 value 1.109134
```

The number of weights is 36, perhaps already a bit too high given the limited number of samples. Clearly, the decrease in error value for the training set has not yet slowed after the (default) maximum value of 100 training iterations – however, using more iterations may lead to overfitting. The usual approach is to divide the data in three sets: the training set, the validation set, of which the predictions are continuously monitored, and a test set. As soon as the errors in the predictions for the validation set start to increase, training is stopped. At that point, the network is considered to be trained, and its prediction errors can be assessed using the test set. Obviously, this can only be done

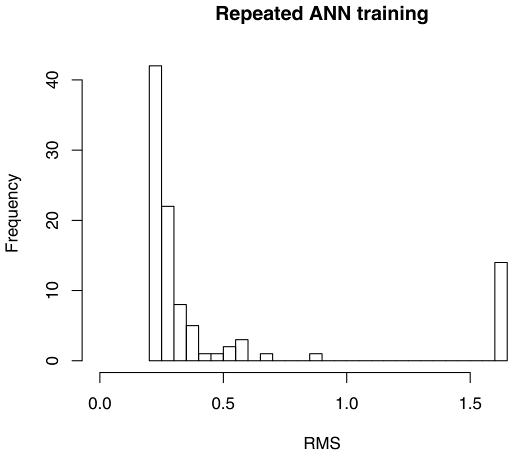

Fig. 8.8. Histogram of prediction errors (RMS values) for the even rows of the gasoline data upon repeated neural network training.

when the number of data points is not too small, a serious impediment in the application of neural nets in the life sciences...

For the moment, we will ignore the issue, and we will assess the predictive performance of our final network:

```
> rms(gas.nnet.pred, gasoline$octane[even])
```

```
[1] 0.2930
```

The prediction error is similar to the PCR result on page 155. Obviously, not only the number of training iterations, but also the number of hidden units needs to be optimized. Similar to the approach in classification, the convenience function `tune.nnet` can be helpful.

One further remark needs to be made: since the initialization of neural nets usually is done randomly, repeated training sessions rarely lead to comparable results. Figure 8.8 shows a histogram of RMS prediction errors (test set, even rows of the gasoline data) of 100 trained networks. Clearly, there is considerable spread. Even a local optimum can be discerned around 1.63 in which more than 10% of the networks end up. The bottom line is that although neural networks have great modelling power, one has to be very careful in using them – in particular, one should have enough data points to allow for rigid validation.

8.7 Classification as a Regression Problem

In many cases, classification can be tackled as a regression problem. The obvious example is logistic regression, one of the most often used classification methods in the medical and social sciences. In logistic regression, one models the log of the *odds ratio*, usually with a multiple linear regression:

$$\ln \frac{p}{1-p} = \boldsymbol{XB} \tag{8.28}$$

where p is the probability of belonging to class 1. This way of modelling has several advantages, the most important of which may be that the result can be directly interpreted as a probability – the p value will always be between 0 and 1. Moreover, the technique makes very few assumptions: the independent variables do not need to have constant variance, or even be normally distributed; they may even be categorical. The usual least-squares estimators for finding $\boldsymbol{B}$ are not used, but rather numerical optimization techniques maximizing the likelihood. However, a big disadvantage is that many data points are needed. Because of the high dimensionality of most data sets in the life sciences, logistic regression has not been widely adopted in this field and we will not treat it further here.

Although it may not seem immediately very useful to use regression for classification problems, it does open up a whole new field of elaborate, possibly non-linear regression techniques as classifiers. Another very important application is classification of fat data matrices, a situation that is covered quite well in a regression context as we have seen. This topic will be treated in more detail in Section 11.3. Here, we will show the general idea in the context of LDA.

8.7.1 Regression for LDA

In the case of a two-class problem (healthy/diseased, true/false, yes/no) the dependent variable is coded as 1 for the first class, and 0 for the other class. For prediction, any object whose predicted value is above 0.5 will be classified in the second class. In the field of machine learning, often a representation is chosen where one class is indicated with the label -1 and the other one with 1; the class boundary then is at 0. For problems involving more than two classes, a class matrix is used with one column for every class and at most one "1" per row; the position of the "1" indicates the class of that particular object.

Package **kohonen** contains a function to convert a class vector to a matrix:

```
> wine.indices <- seq(1, 175, by = 25)
> classvec2classmat(vintages[wine.indices])
```

```
     Barbera Barolo Grignolino
[1,]       0      1          0
[2,]       0      1          0
[3,]       0      1          0
[4,]       0      0          1
[5,]       0      0          1
[6,]       0      0          1
[7,]       1      0          0
```

In this example, there is only one Barbera sample, and three each of Barolo and Grignolino.

To illustrate the close connection between discriminant analysis and linear regression, consider a two-variable subset of the wine data with equal sizes of only classes Barolo and Grignolino:

```
> C <- classvec2classmat(vintages[c(1:25, 61:85)])
> X <- wines[c(1:25, 61:85), c(7, 13)]
```

The regression model can be written as

$$\boldsymbol{C} = \boldsymbol{X}\boldsymbol{B} + \mathcal{E} \tag{8.29}$$

where $\boldsymbol{C}$ is the two-column class matrix – note that because this is only a two-class problem, we could also have used one vector with the 0/1 or -1/1 coding. Solving this equation by least squares leads to:

```
> C <- classvec2classmat(vintages[c(1:25, 61:85)])
> X <- wines[c(1:25, 61:85), c(7, 13)]
> wines.lm <- lm(C ~ X)
> wines.lm.predict <- classmat2classvec(predict(wines.lm))
> table(vintages[c(1:25, 61:85)], wines.lm.predict)

            wines.lm.predict
             Barolo Grignolino
  Barbera         0          0
  Barolo         23          2
  Grignolino      1         24
```

This is exactly the same classification as the one obtained from LDA:

```
> wines.lda <- lda(factor(vintages[c(1:25, 61:85)]) ~ X)
> table(vintages[c(1:25, 61:85)], predict(wines.lda)$class)

             Barolo Grignolino
  Barbera         0          0
  Barolo         23          2
  Grignolino      1         24
```

The `factor` function in the `lda` line is used to convert the three-level factor `vintages` to a two-level factor containing only Barolo and Grignolino, making the comparison with the `lm` predictions easier. This means that instead of doing LDA, we could use linear regression with a binary dependent variable: any object for which the predicted value is larger than 0.5 will be classified in class 2, otherwise in class 1. The direct equality of the least-squares solution with LDA only holds for two groups with equal class sizes [34, 3]; for more classes, or classes with different sizes, the linear regression approach actually optimizes a slightly different criterion than LDA.

8.7.2 Discussion

Although the concept of using regression methods for classification seems appealing and certainly adds flexibility, there are some remarks that should be made. The question arises what exactly it is that we are predicting. Since the only two reasonable values are zero and one, what do other values signify? How can we compare predictions from different classifiers? If a prediction is far greater than one, does that mean that the classification is more reliable than a prediction that is close to the class cut-off point of 0.5?

More serious are the violations of the usual regression assumptions. In logistic regression, maximum likelihood methods are used to obtain the regression coefficients, but in most cases where a classification is disguised as a regression problem, ordinary least squares methods are used. The assumption of normally distributed errors with equal variance is certainly not fulfilled here. Still, as is so often the case, when a method performs in practice by achieving good quality predictions, people will not be afraid to use it.

Part IV

Model Inspection

9

Validation

Validation is the assessment of the quality of a predictive model, in accordance with the scientific paradigm in the natural sciences: a model that is able to make accurate predictions (the position of a planet in two weeks' time) is – in some sense – a "correct" description of reality. In many applications in the natural sciences, unfortunately, validation is hard to do: chemical and biological processes often exibit quite significant variation unrelated to the model parameters. An example is the circadian rhythm: metabolomic samples, be it from animals or plants, will show very different characteristics when taken at different time points. When the experimental meta-data on the exact time point of sampling are missing, it will be very hard to ascribe differences in metabolite levels to differences between patients and controls, or different varieties of the same plant. Only a rigorous and consistent experimental design will be able to prevent this kind of fluctuations. Moreover, biological variation between individuals often dominates measurement variation. The bigger the variation, the more important it is to have enough samples for validation. Only in this way, reliable error estimates can be obtained.

A second validation aspect is to assess the stability of the model coefficients, summarized here with the term *model stability*. In the example of multiple regression with a singular covariance matrix in Section 8.1.1, the variance of the coefficients effectively is infinite, indicating that the model is highly unstable. Regression methods like ridge regression and PLS yield coefficients with much lower variance, at the expense of introducing bias. If it is possible to derive confidence intervals for these, this not only provides an idea of the stability of the model, but it can also be useful in determining which variables actually are important in the model.

Finally, there is the possibility of making use of prior knowledge. Particularly in the natural sciences, one can often assess whether the features that seem important make sense. In a regression model for spectroscopic data, for instance, one would expect wavelengths with large regression coefficients to correspond to peaks in the spectra – large coefficients in areas where no peaks are present would indicate a not too reliable model. Since in most forms of

R. Wehrens, *Chemometrics with R: Multivariate Data Analysis in the Natural Sciences and Life Sciences*, Use R!, DOI 10.1007/978-3-642-17841-2_9,

spectroscopy it is possible to associate spectral features with physicochemical phenomena (specific vibrations, electron transitions, atoms, ...) one can often even say something about the expected sign and magnitude of the regression coefficients. Should these be very different than expected, one may be on to something big – but more likely, one should treat the predictions of such a model with caution, even when the model appears to fit the data well. Typically, more experiments are needed to determine which of the two situations applies. Because of the problem-specific character of this particular type of validation, we will not treat it any further, but will concentrate on the error estimation and model stability aspects..

9.1 Representativity and Independence

One key aspect is that both error estimates and confidence intervals for the model coefficients are derived from the available data (the training data), but that the model will only be relevant when these data are *representative* for the system under study. If there is any systematic difference between the data on which the model is based and the data for which predictions are required, these predictions will be suboptimal and in some cases even horribly wrong. These systematic differences can have several causes: a new machine operator, a new supplier of chemicals or equipment, new schedules of measurement time (“from now on, Saturdays can be used for measuring as well”) – all these things may cause new data to be slightly but consistently different from the training data, and as a result the predictive models are no longer optimal. In analytical laboratories, this is a situation that often occurs, and one approach dealing with this is treated in Section 11.4.

Especially with extremely large data sets, validation is sometimes based on only one division in a training set and a test set. If the number of samples is very large, the sheer size of the data will usually prevent overfitting and the corresponding error estimates can be quite good. However, it depends on how the training and test sets are constructed. A random division is to be preferred; to be even more sure, several random divisions may be considered. One can check whether the training data are really representative for the test data: pathological cases where this is not the case can usually be recognized by simple visualization (e.g. using PCA). However, one should be very careful not to reject a division too easily: as soon as one starts to use the test data, in this case, to assess whether the division between training and test date is satisfactory, there is the risk of biasing the results. The training set should not only be representative of the test set, but also completely *independent*. An example is the application of the Kennard-Stone algorithm [106] to make the division in training and test sets. The algorithm selects training samples from the complete data set to cover the complete space of the independent variables as good as possible. However, if the training samples are selected in such a way that they are completely surrounding the test samples, the prediction error on the test set will probably be lower than it should be – it is biased. Of course, when the algorithm is only used to decrease the number of samples in the training set, and the test set has been set aside before the Kennard-Stone algorithm is run, then there is no problem (provided the discarded training set samples are not added to the test set!) and we can still treat the error on the test set as an unbiased estimate of what we can expect for future samples.

If the available data can be assumed to be representative of the future data, we can use them in several ways to assess the quality of the predictions. The main point in all cases is the same: from the data at hand, we simulate a situation where unseen data have to be predicted. In *crossvalidation*, this is done by leaving out part of the data, and building the model on the remainder. In *bootstrapping*, the other main validation technique, the data are resampled with replacement, so that some data points are present several times in the training set, and others (the “out-of-bag”, or OOB, samples) are absent. The performance of the model(s) on the OOB samples is then an indication of the prediction quality of the model for future samples.

In estimating errors, one should take care not to use *any* information of the test set: if the independence of training and test sets is compromized error estimates become biased. An often-made error is to scale (autoscaling, mean-centering) the data before the split into training and test sets. Obviously, the information of the objects in the test set *is* being used: column means and standard deviations are influenced by data from the test set. This leads to biased error estimates – they are, in general, lower that they should be. In the crossvalidation routines of the **pls** package, for example, scaling of the data is done in the correct way: the OOB samples in a crossvalidation iteration are scaled using the means (and perhaps variances) of the in-bag samples. If, however, other forms of scaling are necessary, this can not be done automatically. The **pls** package provides an explicit `crossval` function, which makes it possible to include sample-specific scaling functions in the calling formula:

```
> gasoline.mscpcr <- pcr(octane ~ msc(NIR), data = gasoline,
+                               ncomp = 4)
> gasoline.mscpcr.cv <- crossval(gasoline.mscpcr, loo = TRUE)
> RMSEP(gasoline.mscpcr.cv, estimate = "CV")
```

```
(Intercept)  1 comps  2 comps  3 comps  4 comps
      1.543    1.452   0.8838   0.2647   0.2712
```

This particular piece of code applies multiplicative scatter correction (MSC, see Section 3.2) on all in-bag samples, and scales the OOB samples in the same way, as it should be done. Interestingly, this leads to a PCR model where three components would be optimal, one fewer component than without the MSC scaling.

9.2 Error Measures

A distinction has to be made between the prediction of a continuous variable (regression), and a categorical variable, as in classification. In regression, the root-mean-square error of validation (RMSEV) is given, analogously to Equation 8.12, by

$$\text{RMSEV} = \sqrt{\frac{\sum_i (\hat{y}_{(i)} - y_{(i)})^2}{n}} \tag{9.1}$$

where $y_{(i)}$ is the out-of-bag sample in a crossvalidation or bootstrap. That is, the predictions are made for samples that have not been used in building the model. A summary of these prediction errors can be used as an estimate for future performance. In this case, the average of the sum of squared errors is taken – sometimes there are better alternatives.

For classification, the simplest possibility is to look at the fraction of correctly classified observations. in R:

```
> err.rate <- function(x, y) sum(x != y)/length(x)
```

A more elaborate alternative is to assign each type of misclassification a *cost*, and to minimize a loss function consisting of the total costs associated with misclassifications. In a two-class situation, for example, this makes it possible to prevent false negatives at the expense of accepting more false positives; in a medical context, it may be the case that a specific test should recognize all patients with a specific disease, even if that means that a few people without the disease are also tagged. Missing a positive sample (a false negative outcome) in this example has much more radical consequences than the reverse, incorrectly calling a healthu person ill.

A related alternative is to focus on the two components of classification accuracy, *sensitivity* and *specificity.* Sensitivity, also known as the *recall rate* or the *true positive rate*, is the fraction of objects from a particular class k which are actually assigned to that class:

$$\text{sensitivity}_k = \frac{TP_k}{TP_k + FN_k} \tag{9.2}$$

where TP_k is the number of True Positives (i.e., objects correctly assigned to class k) and FN_k is the number of False Negatives (objects belonging to class k but classified otherwise). A sensitivity of one indicates that all objects of class k are assigned to the correct class – note that many other objects, not of class k, may be assigned to that class as well.

Specificity is related to the purity of class predictions, and summarizes the fraction of objects in class k that belong elsewhere:

$$\text{specificity}_k = \frac{TN_k}{FP_k + TN_k} \tag{9.3}$$

TN_k and FP_k indicate True Negatives and False Positives for class k, respectively. A specificity of one indicates that no objects have been classified as class k incorrectly. The measure $1 -$ specificity is sometimes referred to as the *false positive rate*.

9.3 Model Selection

In practice, one will have to compromise between specificity and sensitivity: usually, sensitivity can be increased at the expense of specificity and vice versa by changing parameters of the classification procedure. For two-class problems, a common visualization is the Receiver Operating Characteristic (ROC, [107]), which plots the true positive rate against the false positive rate for several values of the classifier threshold. Consider, e.g., the optimization of k, the number of neighbours in the KNN classification of the wine data. Let us focus on the distinction between Barbera and Grignolino, where we (arbitrarily) choose Barbera as the positive class, and Grignolino as negative.

```
> X <- wines[vintages != "Barolo", ]
> vint <- factor(vintages[vintages != "Barolo"])
> kvalues <- 1:12
> ktabs <- lapply(kvalues,
+                 function(i) {
+                   kpred <- knn.cv(X, vint, k = i)
+                   table(vint, kpred)
+                 })
```

For twelve different values of k we calculate the crossvalidated predictions and we save the crosstable. From the resulting list we can easily calculate true positive and false positive rates:

```
> TPrates <- sapply(ktabs, function(x) x[1,1]/sum(x[,1]))
> FPrates <- sapply(ktabs, function(x) 1 - x[2,2]/sum(x[2,]))
> plot(FPrates, TPrates, type = "b",
+      xlim = c(.15, .45), ylim = c(.5, .75),
+      xlab = "FP rate", ylab = "TP rate")
> text(FPrates, TPrates, 1:12, pos = 4)
```

In this case, the result, shown in Figure 9.1, leaves no doubt that $k = 1$ gives the best results: it shows the lowest fraction of false positives (i.e., Grignolinos predicted as Barberas) as well as the highest fraction of true positives. The closer a point is to the top left corner (perfect prediction), the better.

In the field of *model selection*, one aims at selecting the best model amongst a series of possibilities, usually based on some quality criterion such as an error estimate. What makes model selection slightly special is that we are not interested in the error estimates themselves, but rather in the order of the sizes of the errors: we would like to pick the model with the smallest error. Also biased error estimates are perfectly acceptable when the bias does not influence our choice, and in some cases biased estimates are even preferable

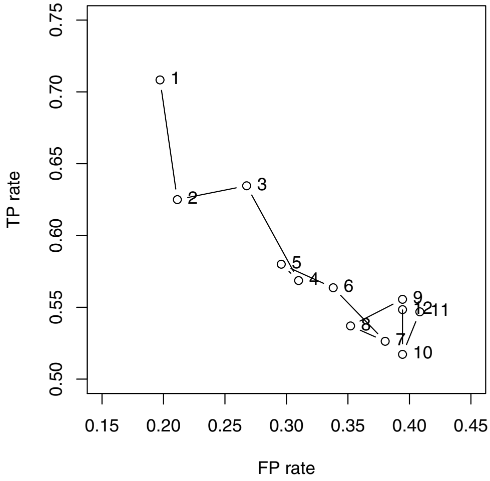

Fig. 9.1. ROC curve (zoomed in to display only the relative part) for the discrimination between Grignolino and Barbera wines using different values of k in KNN classification. Predictions are LOO-crossvalidated.

since they often have a lower variance. We will come back to this in the discussion of different crossvalidation estimates.

An alternative to resampling approaches such as the bootstrap and crossvalidation is provided by simple, direct estimates, usually consisting of a term indicating the agreement between empirical and predicted values, and a penalty for model complexity. Important examples are Mallows' C_p [108] and the AIC and BIC values [53,54], already encountered in Section 6.3. The C_p value is a special case of AIC for general models, adjusting the expected value in such a way that it is approximately equal to the prediction error. In a regression context, these two measures are given by

$$C_p = \text{MSE} + 2 \times p\ \hat{\sigma}^2/n \tag{9.4}$$

$$\text{BIC} = \text{MSE} + \log n \times p\ \hat{\sigma}^2/n \tag{9.5}$$

where n is the number of objects, p is the number of parameters in the model, MSE is the mean squared error of calibration, and $\hat{\sigma}^2$ is an estimate of the residual variance – an obvious choice would be MSE$/(n-p)$ [66]. It can be seen that, for any practical data size, BIC penalizes more heavily than C_p and AIC, and therefore will choose more parsimonious models. For model selection in the life sciences, these statistics have never really been very popular. A simple reason is that it is hard to assess the "true" value of p: how many degrees of freedom do you have in a PLS or PCR regression? Methods like crossvalidation are more simple to apply and interpret – and with computing power being cheap, scientists happily accept the extra computational effort associated with it.

9.4 Crossvalidation Revisited

Crossvalidation, as we already have seen, is a simple and trustworthy method to estimate prediction errors. There are two main disadvantages of LOO crossvalidation. The first is the time needed to perform the calculations. Especially for data sets with many objects and time-consuming modelling methods, LOO may be too expensive to be practical. There are two ways around this problem: the first is to use fast alternatives to direct calculations – in some cases analytical solutions exist, or fast and good approximations. A second possibility is to focus on leaving out larger segments at a time. This latter option also alleviates the second disadvantage of LOO crossvalidation – the relatively large variabiality of its error estimates.

9.4.1 LOO Crossvalidation

Let us once again look at the equation for the LOO crossvalidation error:

$$\varepsilon_{CV}^2 = \frac{1}{n} \sum_{i=1}^{n} \left(y_{(i)} - \hat{y}_{(i)} \right)^2 = \frac{1}{n} \sum_{i=1}^{n} \varepsilon_{(i)}^2 \tag{9.6}$$

where subscript (i) indicates that observation i is being predicted while not being part of the training data. Although the procedure is simple to understand and implement, it can take a lot of time to run for larger data sets. However, for many modelling methods it is not necessary to calculate the n different models explicitly. For ordinary least-squares regression, for example, one can show that the i-th residual of a LOO crossvalidation is given by

$$\varepsilon_{(i)}^2 = \varepsilon_i^2 / (1 - h_{ii}) \tag{9.7}$$

where ε_i^2 is the squared residual of sample i when it is *included* in the training set, and h_{ii} is the i-th diagonal element of the hat matrix $\boldsymbol{H}$, given by

$$\boldsymbol{H} = \boldsymbol{X} \left(\boldsymbol{X}^T \boldsymbol{X} \right)^{-1} \boldsymbol{X}^T \tag{9.8}$$

Therefore, the LOO error estimate can be obtained without explicit iteration by

$$\varepsilon_{CV}^2 = \frac{1}{n} \sum_{i=1}^{n} \left(\frac{y_i - \hat{y}_i}{1 - h_{ii}} \right)^2 \tag{9.9}$$

This shortcut is available in all cases where it is possible to write the predicted values as a product of a type of hat matrix $\boldsymbol{H}$, independent of y, and the measured y values:

$$\hat{y} = \boldsymbol{H} y \tag{9.10}$$

Generalized crossvalidation (GCV, [109]) goes one step further: instead of using the individual diagonal elements of the hat matrix h_{ii}, the average diagonal element is used:

$$\varepsilon^2_{GCV} = \frac{1}{n\left(1 - \sum_{j=1}^n h_{jj}\right)^2} \sum_{i=1}^n (y_i - \hat{y}_i)^2 \tag{9.11}$$

Applying these equations to PCR leads to small differences with the usual LOO estimates, since the principal components that are estimated when leaving out each sample in turn will deviate slightly. Consider the (bad) fit of the one-component PCR model for the gasoline data, calculated with explicit construction of n sets of size $n - 1$:

```
> gasoline.pcr <- pcr(octane ~ ., data = gasoline,
+                     validation = "LOO", ncomp = 1)
> RMSEP(gasoline.pcr, estimate = "CV")
```

```
(Intercept)      1 comps
      1.543        1.447
```

The estimate based on Equation 9.9 is obtained by

```
> gasoline.pcr2 <- pcr(octane ~ ., data = gasoline, ncomp = 1)
> X <- gasoline.pcr2$scores
> HatM <- X %*% solve(crossprod(X), t(X))
> sqrt(mean((gasoline.pcr2$residuals/(1 - diag(HatM)))^2))
```

```
[1] 1.419
```

The GCV estimate from Equation 9.11 deviates more from the LOO result:

```
> sqrt(mean((gasoline.pcr2$residuals/(1 - mean(diag(HatM))))^2))
```

```
[1] 1.389
```

If one is willing to ignore the variation in the PCs introduced by leaving out individual objects, as may be perfectly acceptable in the case of data sets with many objects, this provides a way to significantly speed up calculations. The example above was four times faster than the explicit loop, as is implemented in the `pcr` function with the `validation = "LOO"` argument. For PLS, it is a different story: there, the latent variables are estimated using y, and Equation 9.10 does not hold.

9.4.2 Leave-Multiple-Out Crossvalidation

Instead of leaving out one sample at a time, it is also possible to leave out a sizeable fraction, usually 10% of the data; the latter is also called “ten-fold crossvalidation”. This approach has become quite popular – not only is it roughly ten times faster, it also shows less variability in the error estimates [66]. Again, there is a bias-variance trade-off: the variance may be smaller, but a small bias occurs because the model is based on a data set that is appreciably smaller than the “real” data set, and therefore is slightly pessimistic by nature.

This “leave-multiple-out” (LMO) crossvalidation is usually implemented in a random way: the order of the rows of the data matrix is randomized, and consecutive chunks of roughly equal size are used as test sets. In case the data are structured, it is possible to use non-randomized chunks: the functions in the **pls** package have special provisions for this. The following lines of code lead, e.g., to interleaved sample selection:

```
> gasoline.pcr <- pcr(octane ~ ., data = gasoline,
+                     validation = "CV", ncomp = 4,
+                     segment.type = "interleaved")
> RMSEP(gasoline.pcr, estimate = "CV")
```

```
(Intercept)  1 comps  2 comps  3 comps  4 comps
      1.543    1.426    1.446    1.218   0.2468
```

An alternative is to use `segment.type = "consecutive"`. Also, it is possible to construct the segments (i.e., the crossvalidation sets) by hand or otherwise, and explicitly present them to the modelling function using the `segments` argument. See the manual pages for more information.

9.4.3 Double Crossvalidation

In all cases where crossvalidation is used to establish optimal values for modelling parameters, the resulting error estimates are not indicative of the performance of future observations. They are biased, in that they are used to pick the optimal model. Another round of validation is required. This leads to *double crossvalidation* [65], as visualized in Figure 9.2: the inner crossvalidation loop is used to determine the optimal model parameters, very often, in chemometrics, the optimal number of latent variables, and the outer crossvalidation loop assesses the corresponding prediction error. At the expense of more computing time, one is able to select optimal model parameters as well as estimate prediction error.

The problem is that usually one ends up selecting different parameter settings in different crossvalidation iterations: leaving out segment 1 may lead to a PLS model with two components, whereas segment two may seem to need four PLS components. Which do you choose? Averaging is no solution – again, one would be using information which is not supposed to be available, and the

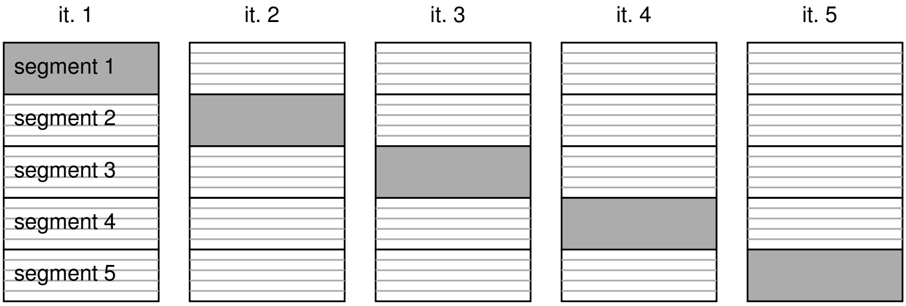

Fig. 9.2. Double crossvalidation: the inner CV loop, indicated by the grey horizontal lines, is used to estimate the optimal parameters for the modelling method. The outer loop, a five-fold crossvalidation, visualized by the gray rectangles, is used to estimate the prediction error.

resulting error estimates would be biased. One approach is to use all optimal models simultaneously, and average the predictions [110]. The disadvantage is that one loses the interpretation of one single model; however, this may be a reasonable price to pay. Other so-called ensemble methods will be treated in Sections 9.7.1 and 9.7.2.

9.5 The Jackknife

Jackknifing [66] is the application of crossvalidation to obtain statistics other than error estimates, usually pertaining to model coefficients. The jackknife can for instance be used to assess the bias and variance of regression coefficients. In general, the mean squared error of a model coefficient estimate b can be decomposed in a bias and a variance component:

$$MSE(b) = \text{Var}(b) + \text{Bias}(b)^2 \tag{9.12}$$

Biased regression methods like ridge regression and PLS achieve lower MSE values by decreasing the variance component, but pay a price by accepting bias. The jackknife estimate of bias, for example, is given by

$$\widehat{\text{Bias}}_{jck}(b) = (n-1)(\bar{b}_{(i)} - b) \tag{9.13}$$

where b is the regression coefficient[1] obtained with the full data, and $b_{(i)}$ is the coefficient from the data with sample i removed, just like in LOO crossvalidation. The bias estimate is simply the difference between the average of all these LOO estimates, and the full-sample estimate, multiplied by the factor $n-1$.

[1] In a multivariate setting we should use an index such as b_j – to avoid complicated notation we skip that for the moment.

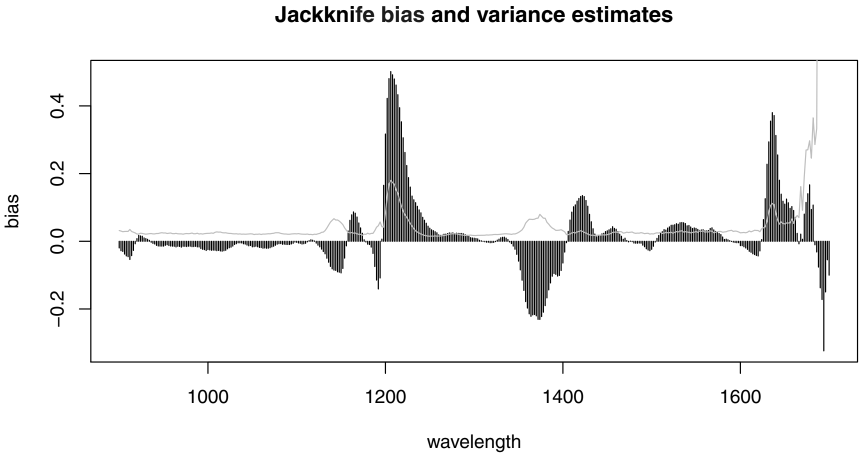

Fig. 9.3. Jackknife estimates of bias and variance (gray line) for a two-component PLS model on the gasoline data.

Let us check the bias of the PLS estimates on the gasoline data using two latent variables. The `plsr` function, when given the argument `jackknife = TRUE`,[2] is keeping all regression coefficients of a LOO crossvalidation in the `validation` element of the fitted object, so finding the bias estimates is not too difficult:

```
> gasoline.pls <- plsr(octane ~ ., data = gasoline,
+                      validation = "LOO", ncomp = 2,
+                      jackknife = TRUE)
> n <- length(gasoline$octane)
> b.oob <- gasoline.pls$validation$coefficients[,,2,]
> bias.est <- (n-1) * (rowMeans(b.oob) - coef(gasoline.pls))
> plot(wavelengths, bias.est, xlab = "wavelength", ylab = "bias",
+      type = "h", main = "Jackknife bias estimates")
```

The result is shown in Figure 9.3 – clearly, the bias for specific coefficients can be appreciable.

The jackknife estimate of variance is given by

$$\widehat{\text{Var}}_{jck}(b) = \frac{n-1}{n} \sum (b_{(i)} - \bar{b}_{(i)})^2 \tag{9.14}$$

and is implemented in `var.jack`. Again, an object of class `mvr` needs to be supplied that is fitted with `jackknife = TRUE`:

[2] Information on this functionality can be found in the manual page of function `mvrCv`.

```
> var.est <- var.jack(gasoline.pls)
> lines(wavelengths, var.est, col = "gray")
```

The result is shown as the gray line in Figure 9.3. In the most important regions, bias seems to dominate variance.

Several variants of the jackknife exist, including some where more than one sample is left out [66]. In practice, however, the jackknife has been replaced by the more versatile bootstrap.

9.6 The Bootstrap

The bootstrap [66,111] is a generalization of the ideas behind crossvalidation: again, the idea is to generate multiple data sets that, after analysis, shed light on the variability of the statistic of interest as a result of the different training set compositions. Rather than splitting up the data to obtain training and test sets, in *nonparametric bootstrapping* one generates a training set – a bootstrap sample – by sampling with replacement from the data. The idea is that where the measured data set is one possible realization of the underlying population, an individual bootstrap sample is, analogously, one realization from the complete set. Since we may have sufficient knowledge of difference between the complete set and the empirical realizations, simply by generating more bootstrap samples, we can study the distribution of the statistic of interest θ. In nonparametric bootstrapping applied to regression problems, there are two main approaches for generating a bootstrap sample, One is to sample (again, with replacement) from the *errors* of the initial model. Bootstrap samples are generated by adding the resampled errors to the original data. This strategy is appropriate when the $\boldsymbol{X}$ data can be regarded as fixed and the model is assumed to be correct. In other cases, one can sample complete cases, i.e., rows from the data matrix, to obtain a bootstrap sample. In such a bootstrap sample, some rows are present multiple times; others are absent.

In *parametric bootstrapping* on the other hand, one describes the data with a parametric distribution, from which then random bootstrap samples are generated. In the life sciences, high-dimensional data are the rule rather than the exception, and therefore any parametric description of a data set is apt to be based on very sparse data. Consequently, the parametric bootstrap has been less popular in this context.

Whether bootstrap samples are generated using parametric or non-parametric bootstrapping, the following analysis is identical. Typically, several hundreds to thousands bootstrap samples are analysed, and the variability of the statistic of interest is monitored. This enables one to make inferences, both with respect to estimating prediction errors and confidence intervals for model coefficients.

9.6.1 Error Estimation with the Bootstrap

Because a bootstrap sample will effectively never contain all samples in the data set, there are samples that have not been involved in building the model. These out-of-bag samples can conveniently be used in estimation of prediction errors. A popular estimator is the so-called .632 estimate $\hat{\varepsilon}_{.632}$, given by

$$\hat{\varepsilon}^2_{.632} = .368 \text{ MSEC} + .632 \bar{\varepsilon}^2_B \tag{9.15}$$

where $\bar{\varepsilon}^2_B$ is the average squared prediction error of the OOB samples in the B bootstrap samples, and MSEC is the mean squared training error (on the complete data set). The factor $.632 \approx (1 - e^{-1})$ is approximately the probability of a sample to end up in a bootstrap sample [66]. In practice, the .632 estimator is the most popular form for estimating prediction errors; a more sophisticated version, correcting possible bias, is known as the .632+ estimator [112] but in many cases the difference is small.

As an example, let us use bootstrapping rather than crossvalidation to determine the optimal number of latent variables in PCR fitting of the gasoline data. In this case, the independent variables are not fixed, and there is some uncertainty on whether the model is correct. This leads to the adoption of the resampling cases paradigm. We start by defining bootstrap sample indices – in this case we take 500 bootstrap samples.

```
> B <- 500
> ngas <- nrow(gasoline)
> boot.indices <-
+   matrix(sample(1:ngas, ngas * B, replace = TRUE), ncol = B)
> sort(boot.indices[,1])
```

```
 [1]  1  4  5  5  6  6  7 10 10 11 12 12 12 14 15 16 17 18
[19] 19 20 21 22 23 23 24 24 24 27 28 28 28 30 30 34 35 38
[37] 38 39 41 44 44 46 47 47 48 48 48 49 51 51 54 55 55 58
[55] 59 59 60 60 60 60
```

Objects 2 and 3 are absent from the first bootstrap sample, shown here as an example, but others occur multiple times – object 60 even four times. We now build PCR models with all these bootstrap samples and record the predictions of the out-of-bag objects. The following code is not particularly memory-efficient but easy to understand:

```
> npc <- 5
> predictions <- array(NA, c(ngas, npc, B))
> for (i in 1:B) {
+   gas.bootpcr <- pcr(octane ~ ., data = gasoline,
+                      ncomp = npc, subset = boot.indices[,i])
+   oobs <- (1:ngas)[-boot.indices[,i]]
+   predictions[oobs,,i] <-
+     predict(gas.bootpcr,
+             newdata = gasoline$NIR[oobs,])[,1,]
+ }
```

Next, the OOB errors for the individual objects are calculated, and summarized in one estimate:

```
> diffs <- sweep(predictions, 1, gasoline$octane)
> sqerrors <- apply(diffs^2, c(1,2), mean, na.rm = TRUE)
> sqrt(colMeans(sqerrors))

[1] 1.4759 1.4872 1.2316 0.2857 0.2765
```

Finally, the out-of-bag errors are combined with the calibration error to obtain the .632 estimate:

```
> gas.pcr <- pcr(octane ~ ., data = gasoline, ncomp = npc)
> RMSEP(gas.pcr, intercept = FALSE)

1 comps  2 comps  3 comps  4 comps  5 comps
 1.3656   1.3603   1.1097   0.2305   0.2260

> error.632 <- .368 * colMeans(gas.pcr$residuals^2) +
+   .632 * colMeans(sqerrors)
> sqrt(error.632)

       1 comps 2 comps 3 comps 4 comps 5 comps
octane   1.436   1.442   1.188  0.2667  0.2591
```

The result is an upward correction of the too optimistic training set errors. We can compare the .632 estimate with the LOO and ten-fold crossvalidation estimates:

```
> gas.pcr.cv <- pcr(octane ~ ., data = gasoline, ncomp = npc,
+                     validation = "CV")
> gas.pcr.loo <- pcr(octane ~ ., data = gasoline, ncomp = npc,
+                      validation = "LOO")
> bp <- barplot(sqrt(error.632),
+                 ylim = c(0, 1.6), col = "peachpuff")
> lines(bp, sqrt(c(gas.pcr.cv$validation$PRESS) / ngas),
+       col = 2)
> lines(bp, sqrt(c(gas.pcr.loo$validation$PRESS) / ngas),
+       col = 3, lty = 2)
> legend("topright", lty = 1:2, col = 2:3,
+        legend = c("CV", "LOO"))
```

The result is shown in Figure 9.4. The estimates in general agree very well – the differences that can be seen are the consequence of the stochastic nature of both ten-fold crossvalidation and bootstrapping: every time a slightly different result will be obtained.

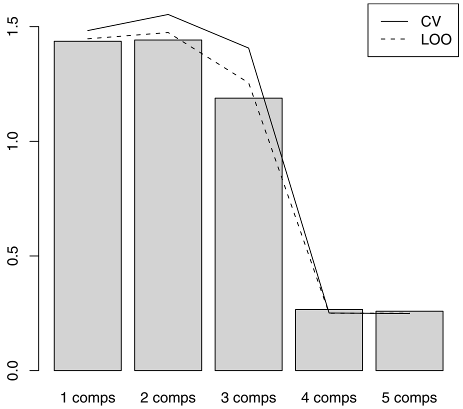

Fig. 9.4. Error estimates for PCR on the gasoline data: bars indicate the result of the .632 bootstrap, the solid line is the ten-fold crossvalidation, and the dashed line the LOO crossvalidation.

It now should be clear what is the philosophy behind the .632 estimator. What it estimates, in fact, is the amount of optimism associated with the RMSEC value, $\hat{\omega}_{.632}$:

$$\hat{\omega}_{.632} = .632(\text{MSEC} - \bar{\varepsilon}_B) \tag{9.16}$$

The original estimate is then corrected for this optimism:

$$\hat{\varepsilon}_{.632} = \text{MSEC} + \hat{\omega}_{.632} \tag{9.17}$$

which leads to Equation 9.15.

Several R packages are available that contain functions for bootstrapping. Perhaps the two best known ones are **bootstrap**, associated with the book by Efron and Tibshirani [66], and **boot**, written by Angelo Canty and implementing functions from Davison and Hinkley [111]. The former is a relatively simple package, maintained mostly to support the book [66] – **boot**, a recommended package, is the primary general implementation of bootstrapping in R. The implementation of the .632 estimator using **boot** is done in a couple of steps [111, page 324]. First, the bootstrap samples are generated, returning the statistic to be bootstrapped – in this case, the prediction errors:[3]

[3] In reference [111] and the **boot** package the number of bootstrap samples is typically a number like 499 or 999, whereas other implementations use 500 and 1000. The differences are not very important in practice.

```
> gas.pcr.boot632 <-
+   boot(gasoline,
+        function(x, ind) {
+          mod <- pcr(octane ~ ., data = x,
+                     subset = ind, ncomp = 4)
+          gasoline$octane -
+            predict(mod, newdata = gasoline$NIR, ncomp = 4)},
+        R = 499)
> dim(gas.pcr.boot632$t)
```

```
 [1] 499  60
```

The optimism is assessed by only considering the errors of the out-of-bag samples. For every bootstrap sample, we can find out which errors we should take into account with the `boot.array` function:

```
> in.bag <- boot.array(gas.pcr.boot632)
> in.bag[1:10,1]
```

```
 [1] 0 1 1 2 0 1 2 0 0 1
```

That is, when considering the occurrence of the first ten objects in the first bootstrap sample, we can see that first object has been left out of the training data in bootstrap samples 1, 5, 8 and 9, was present once in the training data in bootstrap samples 2, 3, 6 and 10, and was present twice in samples 4 and 7. Every bootstrap sample therefore takes a slightly different view of the data.

Averaging the squared errors of the OOB objects leads to the following .632 estimate:

```
> in.bag <- boot.array(gas.pcr.boot632)
> oob.error <- mean((gas.pcr.boot632$t^2)[in.bag == 0])
> app.error <- MSEP(pcr(octane ~ ., data = gasoline, ncomp = 4),
+                   ncomp = 4, intercept = FALSE)
> sqrt(.368 * c(app.error$val) + .632 * oob.error)
```

```
[1] 0.2684
```

This error estimate is slightly higher than the four-fold crossvalidation result on page 183. Note that it is not exactly equal to the .632 estimate on page 188 because different bootstrap samples have been selected. The difference this time is very small indeed.

9.6.2 Confidence Intervals for Regression Coefficients

The bootstrap may also be used to assess the variability of a statistic such as an error estimate. A particularly important application in chemometrics is the standard error of a regression coefficient from a PCR or PLS model. Alternatively, confidence intervals can be built for the regression coefficients.

No analytical solutions such as those for MLR exist in these cases; nevertheless, we would like to be able to say something about which coefficients are actually contributing to the regression model.

Typically, for an interval estimate such as a confidence interval, more bootstrap samples are needed than for a point estimate, such as an error estimate. Several hundred bootstrap samples are taken to be sufficient for point estimates; several thousand for confidence intervals. Taking smaller numbers may drastically increase the variability of the estimates, and with the current abundance of computing power there is rarely a case for being too economical.

The simplest possible approach is the *percentile* method: estimate the models for B bootstrap samples, and use the $B\alpha/2$ and $B(1-\alpha/2)$ values as the $(1-\alpha)$ confidence intervals. For the gasoline data, modelled with PCR using four PCs, these bootstrap regression coefficients are obtained by:

```
> B <- 1000
> ngas <- nrow(gasoline)
> boot.indices <-
+   matrix(sample(1:ngas, ngas * B, replace = TRUE), ncol = B)
> npc <- 4
> gas.pcr <- pcr(octane ~ ., data = gasoline, ncomp = npc)
> coefs <- matrix(0, ncol(gasoline$NIR), B)
> for (i in 1:B) {
+   gas.bootpcr <- pcr(octane ~ ., data = gasoline,
+                      ncomp = npc, subset = boot.indices[,i])
+   coefs[,i] <- c(coef(gas.bootpcr))
+ }
```

A plot of the regression coefficients of all bootstrap samples is shown in Figure 9.5:

```
> matplot(wavelengths, coefs, type = "l",
+         lty = 1, col = "gray",
+         ylab = "Coefficients", xlab = "Wavelength (nm)")
> abline(h = 0)
```

Some of the wavelengths show considerable variation in their regression coefficients, especially the longer wavelengths above 1650 nm.

In the percentile method using 1000 bootstrap samples, the 95% confidence intervals are given by the 25th and 975th ordered values of each coefficient:

```
> coef.stats <- cbind(apply(coefs, 1, quantile, .025),
+                     c(coef(gas.pcr)),
+                     apply(coefs, 1, quantile, .975))
> matplot(wavelengths, coef.stats, type = "n",
+         xlab = "Wavelength (nm)",
+         ylab = "Regression coefficient")
> abline(h = 0, col = "gray")
```

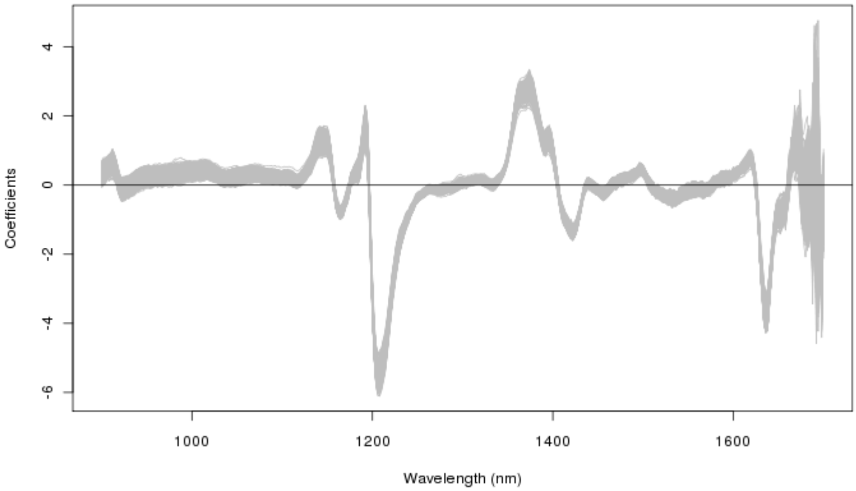

Fig. 9.5. Regression coefficients from all 1000 bootstrap samples for the gasoline data, using PCR with four latent variables.

```
> matlines(wavelengths, coef.stats,
+          lty = c(2,1,2), col = c(2,1,2))
```

The corresponding plot is shown in Figure 9.6. Clearly, for most coefficients, zero is not in the confidence interval. A clear exception is seen in the longer wavelengths: there, the confidence intervals are very wide, indicating that this region contains very little relevant information.

The percentile method was the first attempt at deriving confidence intervals from bootstrap samples [113] and has enjoyed huge popularity; however, one can show that the intervals are, in fact, incorrect. If the intervals are not symmetric (and it can be seen in Figure 9.6 that this is quite often the case – it is one of the big advantages of bootstrapping methods that they are able to define asymmetric intervals), it can be shown that the percentile method uses the skewness of the distribution the wrong way around [66]. Better results are obtained by so-called *studentized* confidence intervals, in which the statistic of interest is given by

$$t_b = \frac{\hat{\theta}_b - \hat{\theta}}{\hat{\sigma}_b} \tag{9.18}$$

where $\hat{\theta}_b$ is the estimate for the statistic of interest, obtained from the b-th bootstrap sample, $\hat{\sigma}_b$ is the standard deviation of that estimate, and $\hat{\theta}$ is the estimate obtained from the complete original data set. In the example of regression, $\hat{\theta}$ corresponds to the regression coefficient at a certain wavelength. Often, no analytical expression existst for $\hat{\sigma}_b$, and it should be obtained by

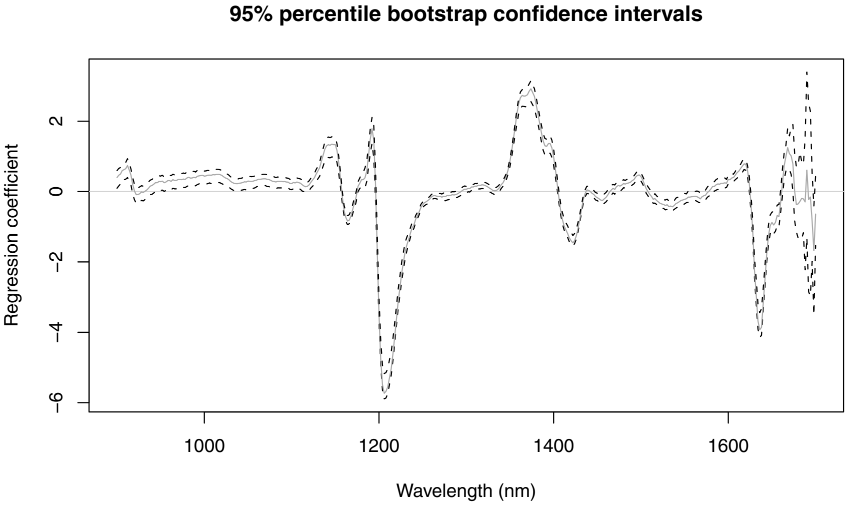

Fig. 9.6. Regression vector and 95% confidence intervals for the individual coefficients, for the PCR model of the gasoline data with four PCs. Confidence intervals are obtained with the bootstrap percentile method.

other means, e.g., crossvalidation, or an inner bootstrap loop. The studentized confidence intervals are then given by

$$\hat{\theta} - t_{B(1-\alpha/2)} \leq \theta \leq \hat{\theta} - t_{B\alpha/2} \tag{9.19}$$

Several other ways of estimating confidence intervals exist, most notably the bias-corrected and accelerated (BCα) interval [66, 111].

The **boot** package provides the function `boot.ci`, which calculates several confidence interval estimates in one go. Again, first the bootstrap sampling is done and the statistics of interest are calculated:

```
> gas.pcr.bootCI <-
+   boot(gasoline,
+        function(x, ind) {
+          c(coef(pcr(octane ~ ., data=x,
+                     subset = ind)))},
+        R = 999)
> dim(gas.pcr.bootCI$t)
```

```
[1] 999 401
```

Here we use `R = 999` to conform to the setup of the **boot** package. The regression coefficients are stored in the `gas.pcr.bootCI` object, which is of class `"boot"`, in the element named `t`. Plots of individual estimates can be made through the `index` argument:

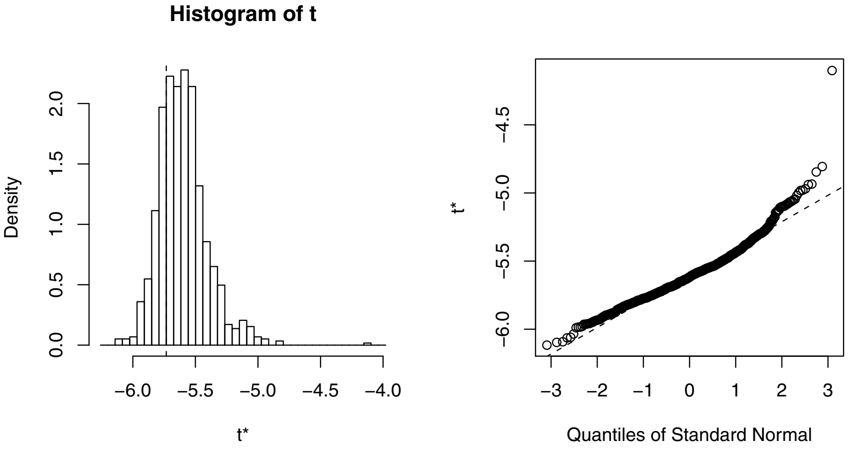

Fig. 9.7. Bootstrap plot for the regression coefficient at 1206 nm; in all bootstrap samples the coefficient is much smaller than zero.

```
> smallest <- which.min(gas.pcr.bootCI$t0)
> plot(gas.pcr.bootCI, index = smallest)
```

From the plot, shown in Figure 9.7, one can see the distribution of the values for this coefficient in all bootstrap samples – the corresponding confidence interval will definitely not contain zero. The dashed line indicates the estimate based on the full data; these estimates are stored in the list element `t0`.

Confidence intervals for individual coefficients can be obtained from the `gas.pcr.bootCI` object as follows:

```
> boot.ci(gas.pcr.bootCI, index = smallest,
+         type = c("perc", "bca"))

BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
Based on 999 bootstrap replicates

CALL :
boot.ci(boot.out = gas.pcr.bootCI, type = c("perc", "bca"),
        index = smallest)

Intervals :
Level     Percentile            BCa
95%   (-5.928, -5.104 )   (-6.117, -5.532 )
Calculations and Intervals on Original Scale
Warning : BCa Intervals used Extreme Quantiles
Some BCa intervals may be unstable
Warning message:
```

```
In norm.inter(t, adj.alpha) : extreme order statistics used
as endpoints
```

Despite the warning messages, one can see that the intervals agree reasonably well; the BCα intervals are slightly shifted downward compared to the percentile intervals. Neither contains zero, as expected.

9.6.3 Other R Packages for Bootstrapping

The bootstrap is such a versatile technique, that it has found application in many different areas of science. This has led to a large number of R packages implementing some form of the bootstrap – at the moment of writing, the package list of the CRAN repository contains already four other packages in between the packages **boot** and **bootstrap** already mentioned. To name just a couple of examples: package **FRB** contains functions for applying bootstrapping in robust statistics; **DAIM** provides functions for error estimation including the .632 and .632+ estimators. Using **EffectiveDose** it is possible to estimate the effects of a drug, and in particular to determine the effective dose level – bootstrapping is provided for the calculation of confidence intervals. Packages **meboot** and **BootPR** provide machinery for the application of bootstrapping in time series.

9.7 Integrated Modelling and Validation

Obtaining a good multivariate statistical model is hardly ever a matter of just loading the data and pushing a button: rather, it is a long and sometimes seemingly endless iteration of visualization, data treatment, modelling and validation. Since these aspects are so intertwined, it seems to make sense to develop methods that combine them in some way. In this section, we consider approaches that combine elements of model fitting with validation. The first case is bagging [114], where many models are fitted on bootstrap sets, and predictions are given by the average of the predictions of these models. At the same time, the out-of-bag samples can be used for obtaining an unbiased error estimate. Bagging is applicable to all classification and regression methods, but will give benefits only in certain cases; the classical example where it works well is given by trees [114] – see below. An extension of bagging, also applied to trees, is the technique of random forests [115]. Finally, we will look at boosting [116], an iterative method for binary classification giving progressively more weight to misclassified samples. Bagging and boosting can be seen as meta-algorithms, because they consist of strategies that, in principle at least, can be combined with any model-fitting algorithm.

9.7.1 Bagging

The central idea behind bagging is simple: if you have a classifier (or a method for predicting continuous variables) that on average gives good predictions but has a somewhat high variability, it makes sense to average the predictions over a large number of applications of this classifier. The problem is how to do this in a sensible way: just repeating the same fit on the same data will not help. Breiman proposed to use bootstrapping to generate the variability that is needed. Training a classifier on every single bootstrap sets leads to an ensemble of models; combining the predictions of these models would then, in principle, be closer to the true answer. This combination of bootstrapping and aggregating is called bagging [114].

The package **ipred** implements bagging for classification, regression and survival analysis using trees – the **rpart** implementation is employed. For classification applications, also the combination of bagging with kNN is implemented (in function `ipredknn`). We will focus here on bagging trees. The basic function is `ipredbag`, while the function `bagging` provides the same functionality using a formula interface. Making a model for predicting the octane number for the gasoline data is very easy:

```
> gasoline.bagging <- ipredbagg(gasoline$octane[odd],
+                               gasoline$NIR[odd,],
+                               coob = TRUE)
> gasoline.bagging
```

```
Bagging regression trees with 25 bootstrap replications
Out-of-bag estimate of root mean squared error:  1.0666
```

The OOB error is already quite high which does not forbode well... Let's see. Predictions for the even-numbered samples can be obtained by the usual `predict` function:

```
> gs.baggpreds <- predict(gasoline.bagging, gasoline$NIR[even,])
> resids <- gs.baggpreds - gasoline$octane[even]
> sqrt(mean(resids^2))
```

```
[1] 1.642050
```

This is not such a good result – even worse than the result of the prediction using four wavelengths, shown on page 152. Again, one should keep in mind that the default settings of the functions may not always be optimal and that in some cases substantial improvements are possible.

Doing classification with bagging is equally simple. Here, we show the example of discriminating between the `control` and `pca` classes of the prostate data, again using only the first 1000 variables as we did on page 119:

```
> prost.bagging <- bagging(type ~ ., data = prost.df,
+                          subset = odd)
```

```
> prost.baggingpred <- predict(prost.bagging,
+                              newdata = prost.df[even,])
> table(prost.type[even], prost.baggingpred)
```

```
         prost.baggingpred
          control pca
  control      29  11
  pca           4  80
```

which doubles the number of misclassifications compared to the SVM solution on page 138) but still is a lot better than the single-tree result.

Does bagging always improve things? Unfortunately not. Clearly, when a classification or regression procedure changes very little with different bootstrap samples, the result will be the same as the original predictions. It can be shown [114] that bagging is especially useful for predictors that are unstable, i.e., predictors that are highly adaptive to the composition of the data set. Examples are trees, neural networks [3] or variable selection methods. In these cases, bagging can improve performance, sometimes even quite drastically so.

9.7.2 Random Forests

The combination of bagging and tree-based methods is a good one, as we saw in the last section. However, Breiman and Cutler saw that more improvement could be obtained by injecting extra variability into the procedure, and they proposed a number of modifications leading to the technique called Random Forests [115]. Again, bootstrapping is used to generate data sets that are used to train an ensemble of trees. One key element is that the trees are constrained to be very simple – only few nodes are allowed, and no pruning is applied. Moreover, at every split, only a subset of all variables is considered for use. Both adaptations force diversity into the ensemble, which is the key to why improvements can be obtained with aggregating.

It can be shown [115] that an upper bound for the generalization error is given by

$$\hat{E} \leq \bar{\rho}(1 - q^2)/q^2$$

where $\bar{\rho}$ is the average correlation between predictions of individual trees, and q is a measure of prediction quality. This means that the optimal gain is obtained when many good yet diverse classifiers are combined, something that is intuitively logical – there is not much point in averaging the outcomes of identical models, and combining truly bad models is unlikely to lead to good results either.

The R package **randomForest** provides a convenient interface to the original Fortran code of Breiman and Cutler. The basic function is `randomForest`, which either takes a formula or the usual combination of a data matrix and an outcome vector:

```
> wines.df <- data.frame(vint = vintages, wines)
> wines.rf <- randomForest(vint ~ ., subset = odd,
+                          data = wines.df)
> wines.rf
```

```
Call:
 randomForest(formula = vint ~ ., data = wines.df, subset = odd)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 3

        OOB estimate of  error rate: 4.49%
Confusion matrix:
           Barbera Barolo Grignolino class.error
Barbera         24      0          0  0.00000000
Barolo           0     28          1  0.03448276
Grignolino       2      1         33  0.08333333
```

The `print` method shows the result of the fit in terms of the error rate of the out-of-bag samples, in this case less than 5%. Because the algorithm fits trees to many different bootstrap samples, this error estimate comes for free. Prediction is done in the usual way:

```
> wines.rf.predict <- predict(wines.rf,
+                             newdata = wines.df[even,])
> sum(wines.rf.predict == wines.df[even,"vint"]) / length(even)
```

```
[1] 1
```

In this case, prediction for the even rows in the data set is perfect. Note that repeated training may lead to small differences because of the randomness involved in selecting bootstrap samples and variables in the training process. Also in many other applications random forests have shown very good predictive abilities (see, e.g., reference [117] for an application in chemical modelling).

So it seems the most important disadvantage of tree-based methods, the generally low quality of the predictions, has been countered sufficiently. Does this come at a price? At first sight, yes. Not only does a random forest add complexity to the original algorithm in the form of tuning parameters, the interpretability suffers as well. Indeed, an ensemble of trees would seem more difficult to interpret than one simple sequence of yes/no questions. Yet in reality things are not so simple. The interpretability, one of the big advantages of trees, becomes less of an issue when one realises that a slight change in the data may lead to a completely different tree, and therefore a completely different interpretation. Such a small change may, e.g., be formed by the difference between successive crossvalidation or bootstrap iterations – thus, the resulting error estimate may be formed by predictions from trees using different variables in completely different ways.

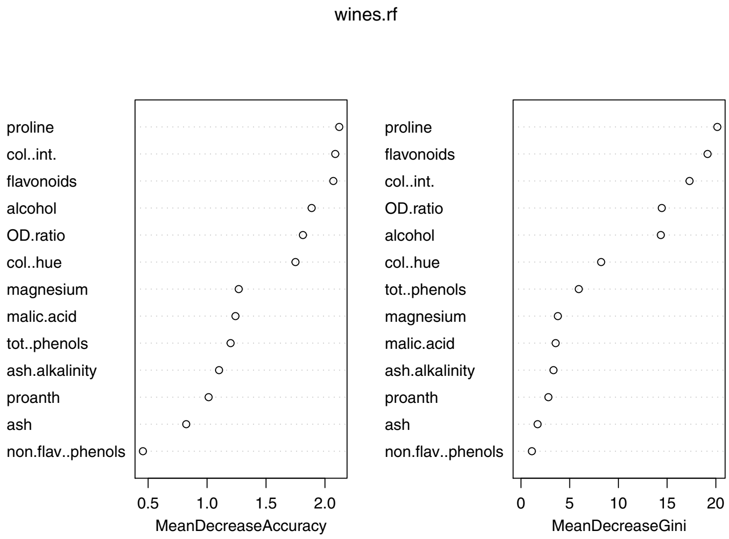

Fig. 9.8. Assessment of variable importance by random forests: the left plot shows the mean decrease in accuracy and the right the mean decrease in Gini index, both after permuting individual variable values.

The technique of random forests addresses these issues in the following ways. A measure of the importance of a particular variable is obtained by comparing the out-of-bag errors for the trees in the ensemble with the out-of-bag errors when the values for that variable are permuted randomly. Differences are averaged over all trees, and divided by the standard error. If one variable shows a big difference, this means that the variable, in general, is important for the classification: the scrambled values lead to models with decreased predictivity. This approach can be used for both classification (using, e.g., classification error rate as a measure) and regression (using a value like MSE). An alternative is to consider the total increase in node purity.

In package **randomForest** this is implemented in the following way. When setting the parameter `importance = TRUE` in the call to `randomForest`, the importances of all variables are calculated during the fit – these are available through the extractor function `importance`, and for visualization using the function `varImpPlot`:

```
> wines.rf <- randomForest(vint ~ ., data = wines.df,
+                          importance = TRUE)
> varImpPlot(wines.rf)
```

The result is shown in Figure 9.8. The left plot shows the importance measured using the mean decrease in accuracy; the right plot using the mean decrease in node impurity, as measured by the Gini index. Although there are small differences, the overall picture is the same using both indices.

The second disadvantage, the large number of parameters to set in using tree-based models, is implicitly taken care of in the definition of the algorithm: by requiring all trees in the forest to be small and simple, no elaborate pruning schemes are necessary, and the degrees of freedom of the fitting algorithm have been cut back drastically. Furthermore, it appears that in practice random forests are very robust to changes in settings: averaging many trees also takes away a lot of the dependence on the exact value of parameters. This has caused random forests to be called one of the most powerful off-the-shelf classifiers available.

Just like the classification and regression trees seen in Section 7.3, random forests can also be used in a regression setting. Take the gasoline data, for instance: training a model using the default settings can be achieved with the following command.

```
> gasoline.rf <- randomForest(gasoline$NIR[odd,],
+                              gasoline$octane[odd],
+                              importance = TRUE,
+                              xtest = gasoline$NIR[even,],
+                              ytest = gasoline$octane[even])
```

For interpretation purposes, we have used the `importance = TRUE` argument, and we have provided the test samples at the same time. The results, shown in Figure 9.9, are better than the ones from bagging:

```
> pl.range <- c(83,90)
> plot(gasoline$octane[odd], gasoline.rf$predicted,
+      main = "Training: OOB prediction", xlab = "True",
+      ylab = "Predicted", xlim = pl.range, ylim = pl.range)
> abline(0, 1)
> plot(gasoline$octane[even], gasoline.rf$test$predicted,
+      main = "Test set prediction", xlab = "True",
+      ylab = "Predicted", xlim = pl.range, ylim = pl.range)
> abline(0, 1)
```

However, there seems to be a bias towards the mean – the absolute values of the predictions at the extremes of the range are too small. Also the RMS values confirm that the test set predictions are much worse than the PLS and PCR estimates of .21:

```
> resids <- gasoline.rf$test$predicted - gasoline$octane[even]
> sqrt(mean(resids^2))
```

```
[1] 0.6167062
```

One of the reasons can be seen in the variable importance plot, shown in Figure 9.10:

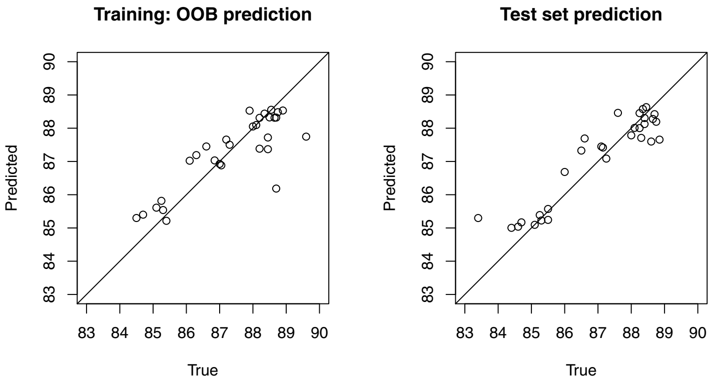

Fig. 9.9. Predictions for the gasoline data using random forests. Left plot: OOB predictions for the training data – right plot: test data.

```
> rf.imps <- importance(gasoline.rf)
> plot(wavelengths, rf.imps[,1] / max(rf.imps[,1]),
+      type = "l", xlab = "Wavelength (nm)",
+      ylab = "Importance")
> lines(wavelengths, rf.imps[,2] / max(rf.imps[,2]), col = 2)
> legend("topright", legend = c("Error decrease", "Gini index"),
+        col = 1:2, lty = 1)
```

Both criteria are dominated by the wavelengths just above 1200 nm. Especially the Gini index leads to a sparse model, whereas the error-based importance values clearly are much more noisy. Interestingly, when applying random forests to the first derivative spectra of the gasoline data set (not shown) the same feature around 1200 nm is important, but the response at 1430 nm comes up as an additional feature. Although the predictions improve somewhat, they are still nowhere near the PLS and PCR results shown in Chapter 8.

For comparison, we also show the results of random forests on the prediction of the even samples in the prostate data set:

```
> prost.rf <-
+   randomForest(prost[odd,], prost.type[odd],
+                x.test = prost[even,], y.test = prost.type[even])
> prost.rfpred <- predict(prost.rf, newdata = prost[even,])
> table(prost.type[even], prost.rfpred)
```

```
         prost.rfpred
          control pca
  control      30  10
  pca           3  81
```

Again, a slight improvement over bagging can be seen.

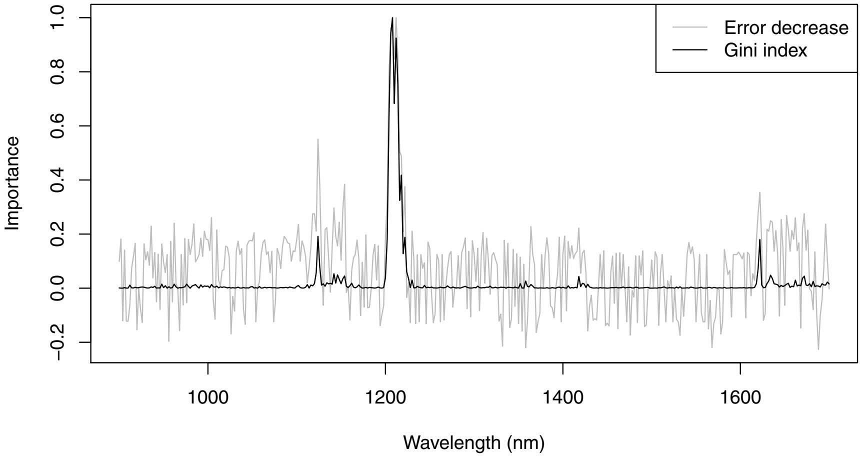

Fig. 9.10. Variable importance for modelling the gasoline data with random forests: basically, only the wavelengths just above 1200 nm seem to contribute.

9.7.3 Boosting

In boosting [116], validation and classification are combined in a different way. Boosting, and in particular in the adaBoost algorithm that we will be focusing on in this section, is an iterative algorithm that in each iteration focusses the attention to misclassified samples from the previous step. Just as in bagging, in principle any modelling approach can be used; also similar to bagging, not all combinations will show improvements.

The main idea is to use weights on the samples in the training set. Initially, these weights are all equal, but during the iterations the weights of incorrectly predicted samples increase. In adaBoost, which stands for adaptive boosting, the changes in the weight of object i is given by

$$D_{t+1}(i) = \frac{D_t(i)}{Z_t} \times \begin{cases} e^{-\alpha_t} & \text{if correct} \\ e^{\alpha_t} & \text{if incorrect} \end{cases} \tag{9.20}$$

where Z_t is a suitable normalization factor, and α_t is given by

$$\alpha_t = .5 \ln \left(\frac{1 - \epsilon_t}{\epsilon_t} \right) \tag{9.21}$$

with ϵ_t the error rate of the model at iteraton t. In prediction, the final classification result is given by the weighted average of the T predictions during the iterations, with the weights given by the α values.

The algorithm itself is very simple and easily implemented. The only parameter that needs to be set in an application of boosting is the maximal number of iterations. A number that is too large would potentially lead to overfitting, although in many cases it has been observed that overfitting does not occur (see, e.g., references in [116]).

Boosting trees in R is available in package **ada** [118], which directly follows the algorithms described in reference [119]. Let us revisit the prostate example, also tackled with SVMs (on page 138):

```
> prost.ada <- ada(type ~ ., data = prost.df, subset = odd)
> prost.adapred <- predict(prost.ada, newdata = prost.df[even,])
> table(prost.type[even], prost.adapred)

         prost.adapred
          control pca
  control      29  11
  pca           4  80
```

The result is equal to the one obtained with bagging. The development of the errors in training and test sets can be visualized using the default `plot` command. In this case, we should add the test set to the `ada` object first[4]:

```
> prost.ada <- addtest(prost.ada, prost[even,], prost.type[even])
> plot(prost.ada, test = TRUE)
```

This leads to the plot in Figure 9.11. The final error on the test set is less than half of the error at the beginning of the iterations. Clearly, both the training and testing errors have stabilized already after some twenty iterations.

The version of boosting employed in this example is also known as *Discrete adaboost* [119, 3], since it returns 0/1 class predictions. Several other variants have been proposed, returning membership probabilities rather than crisp classifications and employing different loss functions. In many cases they outperform the original algorithm [119].

Since boosting is in essence a binary classifier, special measures must be taken to apply it in a multi-class setting, similar to the possibilities mentioned in section 7.4.1. A further interesting connection with SVMs can be made [120]: although boosting does not explicitly maximize margins, as SVMs do, it does come very close. The differences are, firstly, that SVMs use the L_2 norm, the sum of the squared vector elements, whereas boosting uses L_1 (the sum of the absolute values) and L_∞ (the largest value) norms for the weight and instance vectors, respectively. Secondly, boosting employs greedy search rather than kernels to address the problem of finding discriminating directions in high-dimensional space. The result is that although there are intimate connections, in many cases the models of boosting and SVMs can be quite different.

[4] We could have added the test set data to the original call to `ada` as well – see the manual page.

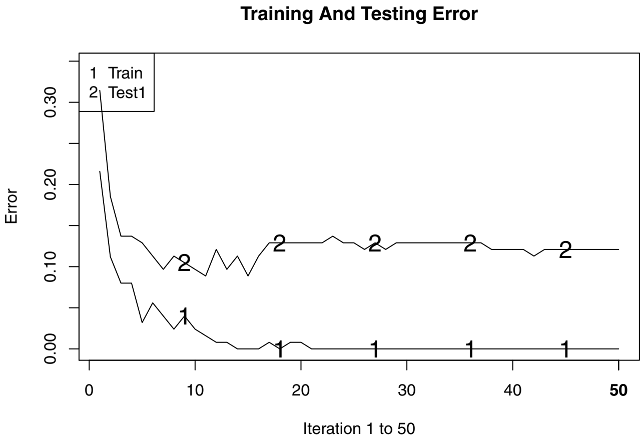

Fig. 9.11. Development of prediction errors for training and test sets of the prostate data (two classes, only 1000 variables) using `ada`.

The obvious drawback of focusing more and more on misclassifications is that these may be misclassifications with a reason: outlying observations, or samples with wrong labels, may disturb the modelling to a large extent. Indeed, boosting has been proposed as a way to detect outliers.

10

Variable Selection

Variable selection is an important topic in many types of modelling: the choice which variables to take into account to a large degree determines the result. This is true for every single technique discussed in this reader, be it PCA, clustering methods, classification methods, or regression. In the unsupervised approaches, uninformative variables can obscure the "real" picture, and distances between objects can become meaningless. In the supervised cases (both classification and regression), there is the danger of chance correlations with dependent variables, leading to models with low predictive power. This danger is all the more real given the very low sample-to-variable ratios of current data sets. The aim of variable selection then is to reduce the independent variables to those that contain relevant information, and thereby to improve statistical modelling. This should be seen both in terms of predictive performance (by decreasing the number of chance correlations) and in interpretability – often, models can tell us something about the system under study, and small sets of coefficients are usually easier to interpret than large sets.

In some cases, one is able to decrease the number of variables significantly by utilizing domain knowledge. A classical application is peak-picking in spectral data. In metabolomics, for instance, where biological fluids are analysed by, e.g., NMR spectroscopy, one can typically quantify hundreds of metabolites. The number of metabolites is usually orders of magnitude smaller than the number of variables (ppm values) that have been measured; moreover, the metabolite concentrations lend themselves for immediate interpretation, which is not the case for the raw NMR spectra. A similar idea can be found in the field of proteomics, where mass spectrometry is used to find the presence or absence of proteins, based on the presence or absence of protein fragments called peptides. Quantification is more problematic here, so typically one obtains a list of proteins that have been found, including the number of fragments that have been used in the identification. When this step is possible it is nearly always good to do so. The only danger is to find what is already known – in many cases, data bases are used in the interpretation of the complex spectra: an unexpected compound, or a compound that is not in the data base but is

R. Wehrens, *Chemometrics with R: Multivariate Data Analysis in the Natural Sciences and Life Sciences*, Use R!, DOI 10.1007/978-3-642-17841-2_10,

present in the sample, is likely to be missed. Moreover, incorrect assignments present additional difficulties. Even so, the list of metabolites or proteins may be too long for reliable modelling or useful interpretation, and one is interested in further reduction of the data.

Very often, this variable selection is achieved by looking at the coefficients themselves: the large ones are retained, and variables with smaller coefficients are removed. The model is then refitted with the smaller set, and this process may continue until the desired number of variables has been reached. Unfortunately, as shown in Section 8.1.1, model coefficients can have a huge variance when correlation is high, a situation that is the rule rather than the exception in the natural sciences nowadays. As a result, coefficient size is not always a good indicator of importance. A more sophisticated approach is the one we have seen in Random Forests, where the decrease in model quality upon permutation of the values in one variable is taken as an importance measure. Especially for systems with not too many variables, however, tests for coefficient significance remain popular.

An alternative way of tackling variable selection is to use modelling techniques that explicitly force as many coefficients as possible to be zero: all these are apparently not important for the model and can be removed without changing the fitted values or the predictions. It can be shown that a ridge-regression type of approach with a penalty on the size of the coefficients has this effect, if the penalty is suitably chosen [3] – a whole class of methods has descended from this principle, starting with Tibshirani's lasso [121].

One could say that the only reliable way of assessing the modelling power of a smaller set is to try it out – and if the result is disappointing, try out a different subset of variables. Given a suitable error estimate (and we will come back to that), one can employ optimization algorithms to find the subset that gives maximal modelling power. Two strategies can be followed: one is to fix the size of the subset, often dictated by practical considerations, and find the set that gives the best performance; the other is to impose some penalty on including extra variables and let the optimization algorithm determine the eventual size. In small problems it is possible, using clever algorithms, to find the globally optimal solution; in larger problems it very quickly becomes impossible to assess all possible solutions, and one is forced to accept that the global optimum may be missed.

10.1 Tests for Coefficient Significance

Testing whether coefficient sizes are significantly different from zero is especially useful in cases where the number of parameters is modest, less than fifty or so. Even if it does not always lead to the optimal subset, it can help to eliminate large numbers of variables that do not contribute to the predictive abilities of the model. Since this is a univariate approach – every variable is tested individually – the usual caveats about correlation apply. Rather than concentrating on the size and variability of individual coefficients, one can compare nested models with and without a particular variable. If the error decreases significantly upon inclusion of that variable, it can be said to be relevant. This is the basis of many stepwise approaches, especially in regression.

10.1.1 Confidence Intervals for Individual Coefficients

Using the wine data as an example, we can assess the confidence intervals for the model quite easily, when we formulate the problem in a regression sense:

```
> X <- wines[odd,]
> C <- classvec2classmat(vintages[odd])
> wines.lm <- lm(C ~ X)
> wines.lm.summ <- summary(wines.lm)
> wines.lm.summ[[3]]
```

```
Response Grignolino :
Coefficients:
                   Estimate Std. Error t value Pr(>|t|)
(Intercept)         2.77235    0.63633    4.36  4.1e-05 ***
Xalcohol           -0.12466    0.04918   -2.53   0.0133 *
Xmalic acid        -0.06631    0.02628   -2.52   0.0138 *
Xash               -0.56351    0.12824   -4.39  3.6e-05 ***
Xash alkalinity     0.03227    0.00975    3.31   0.0014 **
Xflavonoids         0.12497    0.05547    2.25   0.0272 *
Xcol. int.         -0.04748    0.01661   -2.86   0.0055 **
Xproline           -0.00064    0.00012   -5.33  1.0e-06 ***
---
Signif. codes:  0 `***' 0.001 `**' 0.01 `*' 0.05 `.' 0.1 ` ' 1
```

The (edited) result shows the regression coefficients for Grignolino, the third of the dependent variables. We have removed those variables that are not significantly different from zero at the 0.1 confidence level. Seven variables are, and so we could consider a model including only these. Comparing which variables are important for all three cultivars, we can look at the fourth column in the summary for each of the independent variables, which contains the p-value:

```
> sapply(wines.lm.summ,
+        function(x) which(x$coefficients[,4] < .1))
```

```
$`Response Barbera`
      Xmalic acid                Xash        Xflavonoids
                3                   4                  8
Xnon-flav. phenols         Xcol. int.          XOD ratio
                9                  11                 13
```

```
$`Response Barolo`
    (Intercept)        Xalcohol            Xash
              1               2               4
Xash alkalinity     Xflavonoids        Xproanth
              5               8              10
      XOD ratio        Xproline
             13              14


$`Response Grignolino`
    (Intercept)        Xalcohol     Xmalic acid
              1               2               3
           Xash Xash alkalinity     Xflavonoids
              4               5               8
     Xcol. int.        Xproline
             11              14
```

Variables ash and flavonoids occur as significant for all three cultivars; six others (not counting the intercept) for two cultivars.

In cases where no confidence intervals can be calculated analytically, such as in PCR or PLS, we can, e.g., use bootstrap confidence intervals. For the gasoline data, modelled with PCR using four latent variables, we have calculated bootstrap confidence intervals in Section 9.6.2. The percentile intervals, shown in Figure 9.6, already indicated that most regression coefficients are significantly different from zero. How does that look for the (better) BCα confidence intervals? Let's find out:

```
> gas.BCACI <-
+   sapply(1:ncol(gasoline$NIR),
+          function(i, x) {
+            boot.ci(x, index = i, type = "bca")$bca[,4:5]},
+          gas.pcr.bootCI)
```

A plot of the regression coefficients with these 95% confidence intervals immediately shows which variables are significantly different from zero:

```
> coefs <- gas.pcr.bootCI$t0
> matplot(wavelengths, t(gas.BCACI), type = "n",
+         xlab = "Wavelength (nm)", ylab = "Regression coefficient",
+         main = "Gasoline data: PCR (4 PCs)")
> abline(h = 0, col = "gray")
> lines(wavelengths, coefs, col = "gray")
> matlines(wavelengths, t(gas.BCACI), col = 2, lty = 1)
> insignif <- apply(gas.BCACI, 2, prod) < 0
> coefs[insignif] <- NA
> lines(wavelengths, coefs, lwd = 2)
```

In Figure 10.1 non-significant coefficients are shown in gray, significant coefficients in black.

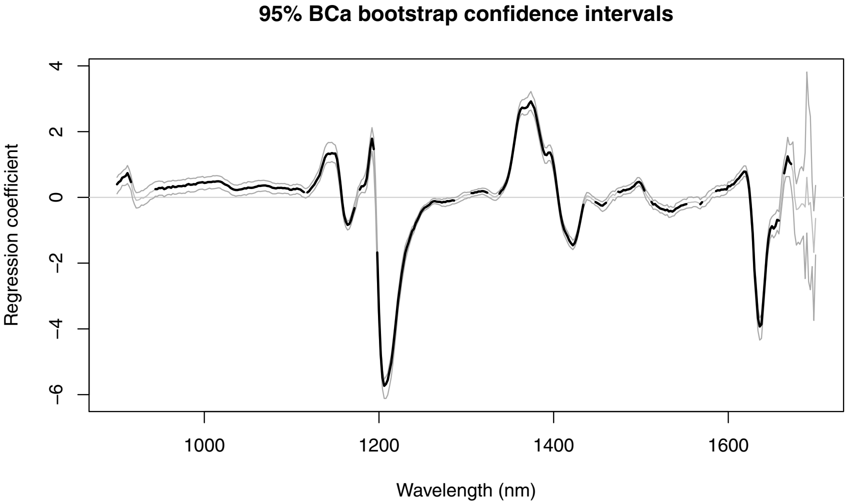

Fig. 10.1. Significance of regression coefficients for PCR using four PCs on the gasoline data; coefficients whose 95% confidence interval (calculated with the BCα bootstrap) does not contain zero are indicated in black, others in gray.

Re-fitting the model after removing the insignificant wavelengths (in total 74, nearly 20%) leads to

```
> smallmod <- pcr(octane ~ NIR[,!insignif], data = gasoline,
+                 ncomp = 4, validation = "LOO")
> RMSEP(smallmod, intercept = FALSE, estimate = "CV")
```

```
1 comps  2 comps  3 comps  4 comps
 1.5236   1.4964   0.2801   0.2945
```

The error estimate is slightly higher than with the full data set, and the optimal number of latent variables has decreased from four to three. This is quite common: after variable selection, refitting leads to more parsimonious models which are often more easy to interpret. Even if the predictions are not (much) better, the improved interpretability is often seen as reason enough to consider variable selection.

Although this kind of procedures has been proposed in the literature (e.g., [122]), it is essentially incorrect. For the spectrum-like data, the correlations between the wavelengths are so large that the confidence intervals of individual coefficients are not useful to determine which variables are significant – both errors of the first (false positives) and second kind (false negatives) are possible. In practice, however, the procedure at least gives some idea of where important information is located.

10.1.2 Tests Based on Overall Error Contributions

In regression problems for data sets with not too many variables, the standard approach is stepwise variable selection. This can be performed in two directions: either one starts with a model containing all possible variables and iteratively discards variables that contribute least. This is called *backward selection*. The other option, *forward selection*, is to start with an "empty" model, i.e., prediction with the mean of the independent variable, and to keep on adding variables until the contribution is no longer significant.

As a criterion for inclusion, values like AIC, BIC or Cp can be employed – these take into account both the improvement in the fit as well as a penalty for having more variables in the model. The default for the R functions `add1` and `drop1` is to use the AIC. Let us consider the regression form of LDA for the wine data, leaving out the Barolo class for the moment:

```
> C <- as.numeric(vintages[vintages != "Barolo"])
> X <- wines[vintages != "Barolo",]
>
> wines2.df <- data.frame(vintages = C, wines = X)
> wines2.lm0 <- lm(vintages ~ 1, data = wines2.df)
> add1(wines2.lm0, scope = names(wines2.df)[-1])
```

```
Single term additions

Model:
vintages ~ 1
                        Df Sum of Sq    RSS    AIC
<none>                               114.6   -2.5
wines.alcohol            1      45.3   69.2  -60.5
wines.malic.acid         1      35.0   79.6  -43.9
wines.ash                1      12.6  102.0  -14.4
wines.ash.alkalinity     1       4.3  110.3   -5.1
wines.magnesium          1       2.9  111.7   -3.6
wines.tot..phenols       1      30.3   84.3  -37.1
wines.flavonoids         1      63.5   51.1  -96.6
wines.non.flav..phenols  1      11.5  103.0  -13.1
wines.proanth            1      18.8   95.8  -21.8
wines.col..int.          1      72.3   42.3 -119.2
wines.col..hue           1      61.1   53.5  -91.2
wines.OD.ratio           1      71.8   42.8 -117.7
wines.proline            1      14.8   99.8  -17.0
```

According to this model, the first variable to enter should be `col..int` – this gives the largest effect in AIC. Since we are comparing equal-sized models, this also implies that the residual sum-of-squares of the model with only an intercept and `col..int` is the smallest.

Conversely, when starting with the full model, the `drop1` function would lead to elimination of the term that contributes least:

```
> wines2.lmfull <- lm(vintages ~ ., data = wines2.df)
> drop1(wines2.lmfull)
```

```
Single term deletions

Model:
vintages ~ wines.alcohol + wines.malic.acid +
    wines.ash + wines.ash.alkalinity +
    wines.magnesium + wines.tot..phenols + wines.flavonoids +
    wines.non.flav..phenols + wines.proanth + wines.col..int. +
    wines.col..hue + wines.OD.ratio + wines.proline
                        Df Sum of Sq    RSS    AIC
<none>                                 14.6 -221.6
wines.alcohol            1       0.1   14.7 -222.7
wines.malic.acid         1       1.3   15.9 -213.3
wines.ash                1       0.5   15.1 -219.5
wines.ash.alkalinity     1       0.1   14.7 -223.1
wines.magnesium          1   0.00016   14.6 -223.6
wines.tot..phenols       1       0.4   15.0 -220.4
wines.flavonoids         1       3.3   17.9 -199.5
wines.non.flav..phenols  1       0.7   15.3 -218.3
wines.proanth            1       0.1   14.7 -222.7
wines.col..int.          1       3.8   18.5 -195.8
wines.col..hue           1       0.6   15.3 -218.4
wines.OD.ratio           1       1.0   15.6 -215.6
wines.proline            1   0.01929   14.6 -223.4
```

In this case, `wines.magnesium` is the variable with the largest negative AIC value, and this is the first one to be removed.

Concentrating solely on forward or backward selection will in practice often lead to sub-optimal solutions: the order in which the variables are eliminated or included is of great importance and the chance of ending up in a local optimum is very real. Therefore, forward and backward steps are often alternated. This is the procedure implemented in the `step` function:

```
> step(wines2.lmfull)
```

```
Coefficients:
            (Intercept)         wines.malic.acid
                  2.444                   -0.114
              wines.ash       wines.tot..phenols
                 -0.472                   -0.167
       wines.flavonoids  wines.non.flav..phenols
                  0.483                    0.764
```

```
    wines.col..int.           wines.col..hue
             -0.129                    0.447
      wines.OD.ratio
              0.270
```

The output of the function has been edited to show those variables eventually ending up in the model, and their coefficients. From the thirteen original variables, only eight remain.

Several other functions can be used for the same purpose: the **MASS** package contains functions `stepAIC`, `addterm` and `dropterm` which allows more model classes to be considered. Package **leaps** contains function `regsubsets`[1] which is guaranteed to find the best subset, based on the branch-and-bounds algorithm. Another package implementing this algorithm is **subselect**, with the function `leaps`.

The branch-and-bounds algorithm was first proposed in 1960 in the area of linear programming [123], and was introduced in statistics by [124]. The title of the latter paper has led to the name of the R-package. This particular algorithm manages to avoid many regions in the search space that can be shown to be less good than the current solution, and thus is able to tackle larger problems than would have been feasible using an exhaustive search. Application of the `regsubsets` function leads to the same set of selected variables:

```
> wines2.leaps <- regsubsets(vintages ~ ., data = wines2.df)
> wines2.leaps.sum <- summary(wines2.leaps)
> names(which(wines2.leaps.sum$which[8,]))
```

```
[1] "(Intercept)"              "wines.malic.acid"
[3] "wines.ash"                "wines.tot..phenols"
[5] "wines.flavonoids"         "wines.non.flav..phenols"
[7] "wines.col..int."          "wines.col..hue"
[9] "wines.OD.ratio"
```

In some special cases, approximate distributions of model coefficients can be derived. For two-class linear discriminant analysis, a convenient test statistic is given by [26]:

$$F = \frac{a_i^2(m-p+1)c^2}{t_i m(m+c^2)D^2} \tag{10.1}$$

with $m = n_1 + n_2 - 2$, n_1 and n_2 signifying group sizes, p the number of variables, $c^2 = n_1 n_2/(n_1+n_2)$, and D^2 is the Mahalanobis distance between the class centers, based on all variables. The estimated coefficient in the discriminant function is a_i, and t_i is the i-th diagonal element in the inverse of the total variance matrix $\boldsymbol{T}$, given by

[1] It also contains the function `leaps` for compatibility reasons; `regsubsets` is the preferred function.

$$\boldsymbol{T} = \boldsymbol{W} + \boldsymbol{B} \tag{10.2}$$

This statistic has an F-distribution with 1 and $m - p + 1$ degrees of freedom. Let us see what that gives for the Grignolino and Barbera classes from the wine data, already considered on page 112, but now using all variables. We again use the code from that page, and subsequently calculate the elements for the test statistic:

```
> Ddist <- mahalanobis(colMeans(wines.groups[[1]]),
+                      colMeans(wines.groups[[2]]),
+                      wines.pcov12)
> m <- sum(sapply(wines.groups, nrow)) - 2
> p <- ncol(wines)
> c <- prod(sapply(wines.groups, nrow)) /
+   sum(sapply(wines.groups, nrow))
> Fcal <- (MLLDA^2 / diag(Tii)) *
+   (m - p + 1) * c^2 / (m * (m + c^2 * Ddist))
> Fcal
> which(Fcal > qf(.95, 1, m-p+1))
```

```
      malic acid                ash         flavonoids
               2                  3                  7
non-flav. phenols          col. int.           col. hue
               8                 10                 11
        OD ratio
              12
```

Using this method, seven variables are shown to be contributing to the separation between Grignolino and Barbera wines on the $\alpha = 0.05$ level. The only variable missing, when compared to the earlier selected set of eight, is `tot..phenols`, which has a p-value of .08.

10.2 Explicit Coefficient Penalization

In the chapter on multivariate regression we already saw that several methods use the concept of shrinkage to reduce the variance of the regression coefficients, at the cost of bias. Ridge regression achieves this by explicit coefficient penalization, as shown in Equation 8.22. Although it forces the coefficients to be closer to zero, the values almost never will be exactly zero. If that would be the case, the method would be performing variable selection: those variables with zero values for the regression coefficients can safely be removed from the data.

Interestingly enough, one can obtain the desired behaviour by replacing the quadratic penalty in Equation 8.22 by an absolute-value penalty:

$$\underset{\boldsymbol{B}}{\arg\max} \, (\boldsymbol{Y} - \boldsymbol{X}\boldsymbol{B})^2 + \lambda|\boldsymbol{B}| \tag{10.3}$$

The penalty, consisting of the sum of the absolute values of the regression coefficients, is an L_1-norm. As already stated before, ridge regression, focussing on squared coefficients, employs an L_2-norm, and measures like AIC or BIC are using the L_0-norm, taking into account only the number of non-zero regression coefficients. In Equation 10.3, with increasing values for parameter λ more and more regression coefficients will be exactly zero. This method has become known under the name *lasso* [121, 3]; an efficient method to solve this equation – and related approaches – has become known under the name of *least-angle regression*, or *LARS* [125]. Several R versions for the lasso are available; package **lars** is written by the inventors of the method, and will be used here as an example. Packages implementing similar routines are **glmnet**, a package for lasso-type generalized linear regression by the same authors, **lpc** for "lassoed principal components", **relaxo**, a generalization of the lasso using possibly different penalization coefficients for the variable selection and parameter estimation steps, and several others.

Rather than one set of coefficients for one given value of λ, the function `lars` from the package with the same name returns an entire sequence of fits, with corresponding regression coefficients. For the odd rows of the gasoline data, the model is simply obtained as follows:

```
> gas.lasso <- lars(gasoline$NIR[odd,], gasoline$octane[odd])
> plot(gas.lasso, xlim = c(0, .1))
> plot(gas.lasso, breaks = FALSE)
> plot(gas.lasso, breaks = FALSE,
+       xvar = "step", xlim = c(0, 20))
```

The result of the corresponding `plot` method is shown in the top left pane in Figure 10.2. It shows the (standardized) regression coefficients against the size of the L_1 norm of the coefficient vector. The x-axis has been truncated here to be able to see what is happening. For an infinitely large value of λ, the weight of the penalty, no variables are selected. Gradually decreasing the penalty leads to a fit using only one non-zero coefficient. Its size varies linearly with the penalty – until the next variable enters the fray. The right of the plot shows the position of the entrances of new non-zero coefficients. This piecewise linear behaviour is the key to the LARS algorithm, and makes it possible to calculate the whole trace in approximately the same amount of time as needed for a normal linear regression. The complete trace is shown in the top right plot in the same figure; for clarity the vertical lines have been left out (using `breaks = FALSE`). To the right, the variable numbers of some of the coefficients are shown at their "final" values, i.e. at the end point of the algorithm. This enables easy identification of what may be the most influential variables. The bottom left plot shows the same coefficients plotted against the number of variables in the model.

Of course, the value of the regularization parameter λ needs to be optimized. A function `cv.lars` is available for that, using by default ten-fold crossvalidation. It automatically plots the CV curve:

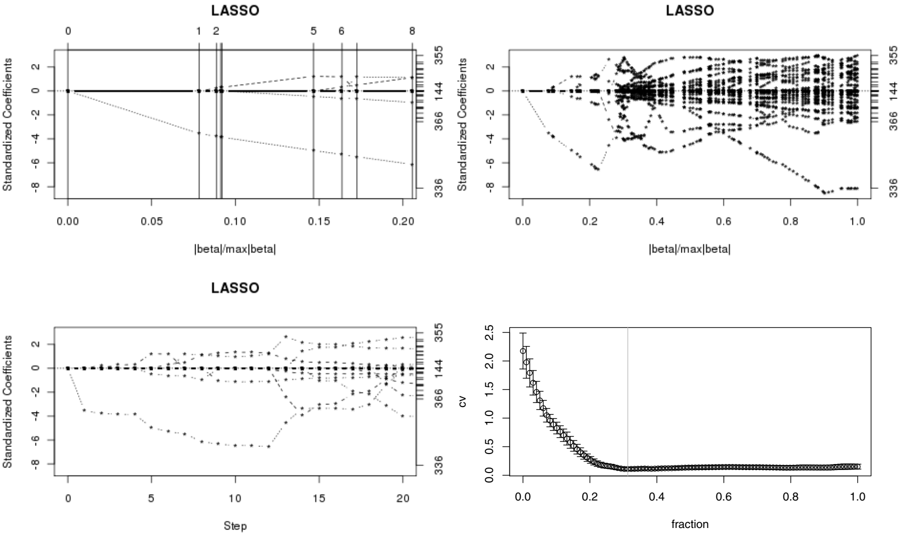

Fig. 10.2. Top row: coefficients plotted against the fraction of the coefficient vector lenght. Left plot: partial view showing the inclusion of the first eight variables. Right plot: complete view until the saturated model. Bottom row, left plot: same coefficients, now plotted against the number of included variables. Right plot: cross-validation curve.

```
> gas.lasso.cv <- cv.lars(gasoline$NIR[odd,],
+                         gasoline$octane[odd])
> best <- which.min(gas.lasso.cv$cv)
> abline(v = gas.lasso.cv$fraction[best], col = "red")
```

The result is shown in the bottom right plot in Figure 10.2. In this case, the optimal CV error is reached at approximately one-third of the maximal length of the coefficient vector. Thus, “optimal” predictions are obtained at this point:

```
> gas.lasso.pred <- predict(gas.lasso, gasoline$NIR[even,],
+                           s = best)
> resids <- gas.lasso.pred$fit - gasoline$octane[even]
> sqrt(mean(resids^2))
```

```
[1] 0.2263087
```

The prediction error for the test set is very close to the best values seen with PCR and PLS, but now only a small fraction of the 400 original variables is included in the model. The `predict.lars` function can also be used to assess which coefficients are non-zero:

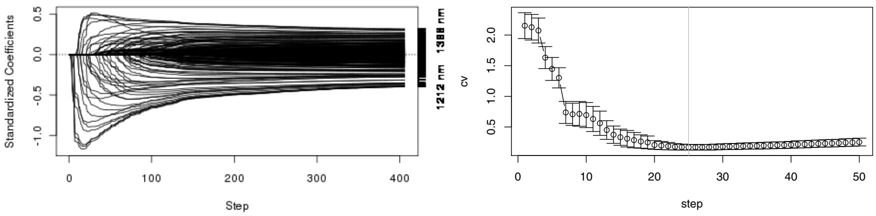

Fig. 10.3. Elastic net results for the gasoline data. The left plot shows the development of the regression coefficients upon relaxation of the penalty parameter. The right plot shows the ten-fold crossvalidation curve for the first fifty variables.

```
> gas.lasso.coefs <- predict(gas.lasso, gasoline$NIR[even,],
+                            s = best, type = "coef")
> gas.lasso.coefs$coefficients[gas.lasso.coefs$coeff != 0]
```

```
   914 nm    1194 nm    1206 nm    1212 nm    1226 nm    1360 nm
 15.22869    7.55376   -4.27597  -38.28831  -57.36533    0.28568
  1362 nm    1366 nm    1620 nm    1630 nm    1636 nm    1682 nm
 27.64500   30.18210   18.25265  -12.84600   -1.39421   -1.00424
  1692 nm    1696 nm    1698 nm    1700 nm
 -0.63252   -3.07971   -0.37270    0.00218
```

Again we see that also in this model the longer wavelengths at the right extreme are included, but that the main coefficients are around 1200 and 1360 nm.

A further development is mixing the L_1-norm of the lasso and related methods with the L_2-norm used in ridge regression. This is known as the *elastic net* [126]. The penalty term is given by

$$\sum_i \left(\alpha|\beta_i| + (1-\alpha)\beta_i^2 \right) \tag{10.4}$$

where the sum is over all variables. The result is that large coefficients are penalized heavily (because of the quadratic term) and that many of the coefficients are exactly zero, leading to a sparce solution.

The elastic net is available as function `enet` in package **elasticnet**:

```
> gas.enet <- enet(gasoline$NIR, gasoline$octane, lambda = .5)
> plot(gas.enet, "step")
```

The argument `lambda` indicates the weight of the quadratic penalty – using `lambda = 0` would lead to the LASSO solution seen earlier. The plot, shown in the left part of Figure 10.3, again displays the development of the coefficients upon inclusion of more and more variables. Crossvalidation is available, too. In the following example, we limit ourselves to the inclusion of a maximum of 50 variables:

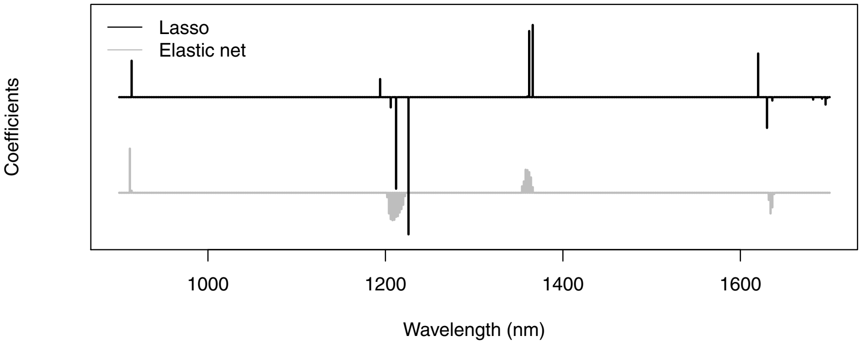

Fig. 10.4. Non-zero coefficients in the lasso and elastic net models. A small vertical offset has been added to facilitate the comparison.

```
> gas.enet.cv <- cv.enet(gasoline$NIR, gasoline$octane,
+                               lambda = .5, s=1:50, mode="step")
```

The result is shown in the right plot in Figure 10.3. Further inspection of the elastic net model, completely analogous to the code shown earlier for the lasso, shows that in this case an RMS value is obtained for the test data of 0.427, almost twice the error of the lasso solution, using 24 non-zero coefficients. However, the location of the important variables seems to make more sense than in the case of the lasso: from Figure 10.4 we can see that in the noisy area at the longer wavelengths nothing is selected by the elastic net. Moreover, the variables that are selected are placed in four small, contiguous regions, which makes sense chemically.

10.3 Global Optimization Methods

Given the speed of modern-day computing, it is possible to examine large numbers of different models and select the best one. However, as we already saw with leaps-and-bounds approaches, even in cases with a moderate number of variables it is practically impossible to assess the quality of all subsets. One must, therefore, limit the number of subsets that is going to be considered to a manageable size. The stepwise approach does this by performing a very local search around the current best solution before adding or removing one variable; it can be compared to a steepest-descent strategy. The obvious disadvantage is that many areas of the search space will never be visited. For regression or classification cases with many variables, almost surely the method will find a local optimum, very often of low quality.

An alternative is given by random search – just sampling randomly from all possible subsets until time is up. Of course, the chance of finding the global optimum in this way is smaller than the chance of winning the lottery... What is needed is a search strategy that combines random elements with "gradient" information; that is, a strategy that uses information, available in solutions of higher quality, with the ability to throw that information away if needed, in order to be able to escape from local optima. This type of approaches has become known under the heading of *global search* strategies. The two best-known ones in the area of chemometrics are *Simulated Annealing* and *Genetic Algorithms*. Both will be treated briefly below.

What is quality, in this respect, again depends on the application. In most cases, the property of interest will be the quality of prediction of unseen data, which for larger data sets can conveniently be estimated by crossvalidation approaches. For data sets with few samples, this will not work very well because of the coarse granularity of the criterion: many subsets will lead to an equal number of errors. Additional information should be used to distinguish between these.

10.3.1 Simulated Annealing

In Simulated Annealing (SA, [127, 128]), a sequence of candidate solutions is assessed, starting from a random initial point. A new solution, not too far away from the current one, is unconditionally accepted if it is better than the current one. If not, and this is the defining feature of SA, it is accepted with a probability

$$p_{\text{acc}} = \exp\left((E_{i+1} - E_i)/T_t\right) \tag{10.5}$$

where E_i is the quality of the current solution, E_{i+1} the quality of the new candidate solution, and T_i the state of the control parameter at the current time point i. Note that p_{acc}, defined in this way, is always between zero and one since $E_i > E_{i+1}$: we already saw that an improvement will always be accepted so in this case the quality of the current solution is higher than that of the new solution. This criterion is known as the *Metropolis* criterion [129]. Other criteria are possible, too, but are hardly ever used.

The name comes from an analogy to annealing in metallurgy, where crystals with fewer defects can be created by repeatedly heating and cooling a material: during the (slow) cooling, the atoms are able to find their energetically most favorable positions in a regular crystal lattice, whereas the heating allows atoms that have been caught in unfavourable positions (local optima) to "try again" in the next cooling stage. The analogy with the optimization task is clear: if an improvement is found (better atom positions) it is accepted; if not, then sometimes a deterioration in quality is accepted in order to be able to cross a ridge in the solution landscape and to find an solution that is better in the end. Very often, the control parameter is therefore indicated with T, to stress the analogy with temperature. During the optimization, it will slowly be decreasing in magnitude – the cooling – causing fewer and fewer solutions of lower quality to be accepted. In the end, only real improvements are allowed. It can be shown that SA leads to the global optimum if the cooling is slow enough [130]; unfortunately, the practical importance of this proof is limited since the cooling may have to be infinitely slow...

The naive implementation of an SA therefore can be very simple: one needs a function that generates a new solution in the neighbourhood of the current one, an evaluation function to assess the quality of the new solution, and the acceptance function, including a cooling schedule for the search parameter T. For the evaluation function, typically some form of crossvalidation is used – note that the evaluation function in this schedule probably is the most time-consuming step, and since it will be executed many times (typically tens of thousands or even millions of solutions are evaluated by global search methods) it should be very fast. If possible, analytical estimates for crossvalidation error should be used.

The step function should generate a new solution that is somewhat different from the current one, but not too much. In this way a trajectory through the solution space can be obtained – if the differences between old and new solutions would be too big, we would be performing random search. The following is an example of a simple step function for variable selection:

```
> SAstep <- function(curr.set, maxvar,
+                    fraction = .3, size.dev = 1)
+ {
+   if (is.null(curr.set)) { # create a new set
+     sample(1:maxvar, round(maxvar * fraction))
+   } else { # modify the existing one
+     new.size <- length(curr.set)
+     if (size.dev > 0) # perhaps change the size
+       new.size <- new.size + sample(-size.dev:size.dev, 1)
+     if (new.size < 2) new.size <- 2
+     if (new.size > maxvar - 2) new.size <- maxvar - 2
+
+     not.in <- which(!((1:maxvar) %in% curr.set))
+     superset <- c(curr.set,
+                   sample(not.in, max(size.dev, 1)))
+     newset <- sample(superset, new.size)
+     ## looks familiar? If yes, then try again
+     while (length(newset) == length(curr.set) &&
+            !any(is.na(match(newset, curr.set))))
+       newset <- sample(superset, new.size)
+
+     newset
+   }
+ }
```

If no current subset is given, a random selection is made, the size of a given fraction of the complete set. If a subset is presented as the first argument, the function first determines the length of the output: if `size.dev` equals zero, the output has the same length as the input. If not (the default), there is some margin for change. From the combination of the current set and a random set of the previously unselected variables a new subset is selected, again randomly. The last step is repeated until the new set differs from the current set, which is usually the case. Anticipating the application to a well-known data set, let us generate a random subset from thirteen variables as an example:

```
> sbst <- SAstep(NULL, 13)
> sbst

[1] 13  5  2  1

> (sbst <- SAstep(sbst, 13))

[1]  5  1 13

> (sbst <- SAstep(sbst, 13))

[1]  5 13  3
```

Of course, it is very well possible that an earlier selected subset is reached again during the SA iterations, and in fact, the next step in the example – that can be reproduced by issuing `set.seed(7)` before running the first assignment of `sbst` – would again lead to the second subset. More sophisticated step functions can take this into account; another possibility is to generate a number of close neighbours rather than one, and to accept the best (or the least bad) of these.

The evaluation is based on the fast built-in LOO classification estimates of the `lda` function; to force the algorithm to solutions with as few variables as possible one can add an explicit penalization for every variable included in the model. The following (rather naive) function takes the number of correct classifications minus the number of variables as the objective function:

```
> lda.loofun <- function(x, grouping, subset)
+ {
+   lda.obj <- lda(x[, subset, drop = FALSE],
+                  grouping, CV = TRUE)
+
+   sum(lda.obj$class == grouping) - length(subset)
+ }
```

Of course, many methods may be used to strike a balance between the quality of the fit and the number of variables, analogously to the AIC and BIC criteria. The SA function itself is now a simple loop with some bookkeeping:

```
> SAfun <- function(x, response, eval.fun, Tinit, niter = 100,
+                   cooling = .05, fraction = .3, ...)
+ { # preparations...
+   nvar <- ncol(x)
+   best <- curr <- SAstep(NULL, nvar, fraction = fraction)
+   best.q <- curr.q <- eval.fun(x, response, curr, ...)
+
+   Temp <- Tinit
+   for (i in 1:niter) { # Go!
+     new <- SAstep(curr, nvar)
+     new.q <- eval.fun(x, response, new, ...)
+     accept <- TRUE
+     if (new.q < curr.q) { # Metropolis criterion
+       p.accept <- exp((new.q - curr.q) / Temp)
+       if (runif(1) > p.accept) accept <- FALSE
+     }
+     if (accept) {
+       curr <- new
+       curr.q <- new.q
+       if (curr.q > best.q) { # store best until now
+         best <- curr
+         best.q <- curr.q
+       }
+     }
+     Temp <- Temp * (1 - cooling)
+   }
+   list(best = best, best.q = best.q)
+ }
```

This SA function takes the data matrix and the response as the first two arguments, and an evaluation function as its third; the initial value of the temperature control parameter is in position four. Extra arguments can be passed to the evaluation function using the ellipses (...). At every iteration, a new candidate subset is evaluated. If it is accepted, it replaces the old subset, and a check is performed to see whether the new solution is the best one so far. At the end of the iteration, the best subset and the associated quality value are returned.

Let us see how this works in the two-class wines example from Section 10.1.2, excluding the Barolo variety. This is a simple example for which it still is quite difficult to assess all possible solutions, especially since we do not force a model with a specific number of variables. An SA run with 100 iterations leads to the following result:

```
> C <- factor(vintages[vintages != "Barolo"])
> X <- wines[vintages != "Barolo",]
```

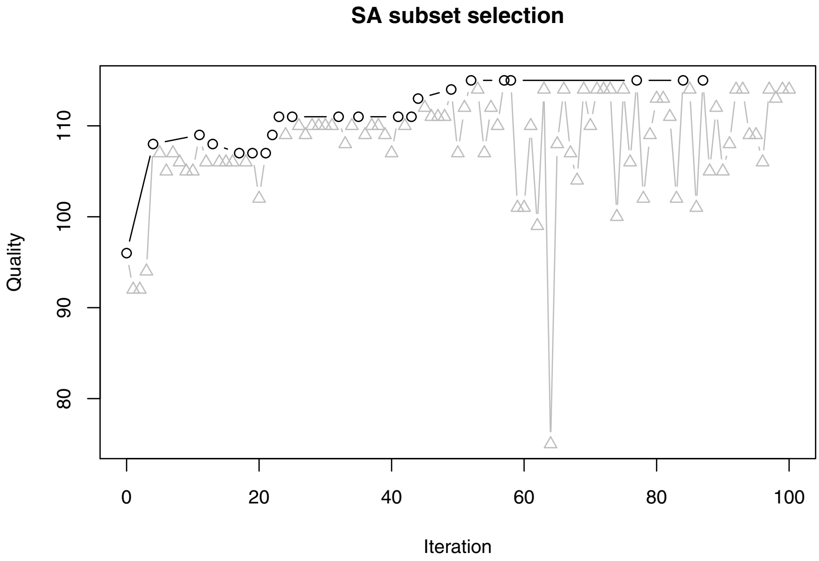

Fig. 10.5. Progression of the quality of the candidate subsets during an SA optimization. Black circles indicated accepted solutions; gray triangles indicate rejections.

```
> SAobj <- SAfun(X, C, lda.loofun, Tinit = 1)
> SAobj

$best
[1]  11 7 10

$best.qual
[1] 115
```

With only three variables, flavonoids, color intensity and color hue, the result is only one misclassification; in fact, this is the same error rate as obtained with the full set. Several other combinations may be found with the same error rate. It is illustrative to consider the quality values throughout the optimization, shown in Figure 10.5. Black circles indicate acceptance of a new solution; gray triangles indicate new solutions that have been rejected. In the beginning of the optimization, sometimes solutions are accepted that do not constitute an improvement (visible between iterations ten and twenty). Later in the optimization, with the lower temperature parameter, this no longer occurs.

A more ambitious example is to predict the octane number of the gasoline samples with only a subset of the NIR wavelengths. The only thing we have to change is the evaluation function:

```
> pls.cvfun <- function(x, response, subset, ...)
+ {
+   pls.obj <- plsr(response ~ x[,subset],
+                   validation = "CV", ...)
+   -MSEP(pls.obj, estimate = "CV")$val[length(subset) + 1]
+ }
```

In this case, we use the explicit crossvalidation provided by the `plsr` function. The number of components to take into account can be specified in the extra argument of the evaluation function; the error of the model with the largest number of latent variables is returned. Since the SA function, as implemented here, does maximization only, the negative value of the mean squared error is used as quality indicator. Let us try to find an optimal two-component PLS model (fewer variables often lead to less complicated models). We start with a very small model using only eight variables ($.02 \times 401$), do the optimization, and look at the RMSCV of the best solution:

```
> SAobj <- SAfun(gasoline$NIR, gasoline$octane,
+                eval.fun = pls.cvfun, Tinit = 3,
+                fraction = .02, niter = 1000, ncomp = 2)
> length(SAobj$best)

[1] 20

> sqrt(-SAobj$best.q)

[1] 0.1998400
```

Comparing with the model using two latent variables in Figure 8.3, the error of the best subset of twenty variables is almost three times smaller! In this optimization, we used 1000 iterations and a higher value of the search parameter `Tinit`; users will in most cases do some tweaking to find optimal results.

Although the above examples only take seconds on a modern desktop computer, real applications can be very computer-intensive; the implementation should be done in compilable code such as Fortran or C. This is the case for the `anneal` function in package **subselect**, which can be used for variable selection in situations like discriminant analysis, PCA, and linear regression, according to the criterion employed. For LDA, this function takes the between-groups covariance matrix, the minimal and maximal number of variables to be selected, the within-groups covariance matrix and its expected rank, and a criterion to be optimized (see below) as arguments. For the LDA example above, a solution to find the optimal three-variable subset would look like this:

```
> winesHmat <- ldaHmat(X, C)
> wines.anneal <-
+   anneal(winesHmat$mat, kmin = 3, kmax = 3,
+          H = winesHmat$H, criterion = "ccr12", r = 1)
```

```
> wines.anneal$bestsets

       Var.1 Var.2 Var.3
Card.3     2     7    10

> wines.anneal$bestvalues

   Card.3
0.8328148
```

Repeated application (using, e.g., `nsol = 10`) in this case leads to the same solution every time. Rather than the direct estimates of prediction error, the `anneal` function uses functions of the within- and between-groups covariance matrices [131]. In this case using the `ccr12` criterion, the first root of $\boldsymbol{B}\boldsymbol{W}^{-1}$ is optimized, analogous to Fisher's formulation of LDA on page 111. As an other example, Wilk's Λ is given by

$$\Lambda = \det(\boldsymbol{W}) / \det(\boldsymbol{T}) \tag{10.6}$$

and is (in a slightly modified form) available in the `tau2` criterion. For the current case where the dimensionality of the within-covariance matrices is estimated to be one, all criteria lead to the same result.

The new result differs from the subset from our own implementation in only one instance: variable 11, color hue, is swapped for the malic acid concentration. The reason, of course, is that both functions optimize different criteria. Let us see how the two solutions fare when evaluated with the criterion of the other algorithm. The value for the `ccr12` criterion of the solution using variables 7, 10 and 11, found with our own simplistic SA implementation, can be assessed easily:

```
> ccr12.coef((nrow(X) - 1) * var(X), winesHmat$H,
+            r = 1, c(7, 10, 11))

[1] 0.8229304
```

which, as expected, is slightly lower than that of the set consisting of variables 2, 7 and 10. Conversely, the prediction quality of the newer set is slightly worse:

```
> lda.loofun(X, C, c(2, 7, 10))

[1] 114
```

Obviously, there are probably many sets with the same or similar values for the quality criterion of interest, and to some extent it is a matter of chance which one is returned by the search algorithm. Moreover, the number of possible quality values can be limited, especially with criteria based on the number of misclassifications. This can make it more difficult to discriminate between two candidate subsets.

The `anneal` function for subset selection is also applicable in other types of problems than classification alone: e.g., for variable selection in PCA it uses a measure of similarity of the original data matrix and of the projections on the k-variable subspace – again, several different criteria are available. The speed and applicability in several domains are definite advantages of this particular implementation. However, there are some disadvantages, too: firstly, because of the formulation using covariance matrices it is hard to apply `anneal` to problems with large numbers of variables. Finding the most important discriminating variables in the prostate data set would stretch your computer to the limit... Secondly, in the case of LDA the quality criteria are focussing very much on the fit of the training data, rather than on prediction ability. The danger of overfitting is huge – we will come back to this point later. Finally, it can be important to monitor the progress of the optimization, or at least keep track of the speed with which improvements are found – especially when fine-tuning the SA parameters (temperature, cooling schedule) one would like to have the possibility to assess acceptance rates. Currently, no such functionality is provided in the **subselect** package.

10.3.2 Genetic Algorithms

Genetic Algorithms (GAs, [132]) manage a population of candidate solutions, rather than one single solution as is the case with most other optimization methods. Every solution in the population is represented as a string of values, and in a process mimicking sexual reproduction offspring is generated combining parts of the parent solutions. The quality of the offspring is measured in an evaluation phase – again in analogy with biology, this quality is often called "fitness". Strings with a low fitness will have no or only a low probability of reproduction, so that subsequent generations will generally consist of better and better solutions. This obvious imitation of the process of natural selection has led to the name of the technique. GAs have been applied to a wide range of problems in very diverse fields – an overview of applications within chemistry can be found in, e.g., [133].

Again, the best way to find out how GAs work is to write a simple version ourselves. The first choice we have to make is on the *representation* of the candidate solutions, i.e., the candidate subsets. Two obvious possibilities present themselves: either a vector of indices of the variables in the subset, or a string of zeros and ones. For other optimization problems, e.g. non-linear fitting, real numbers can also be used. In the example here, we will use vectors of indices to indicate subsets; a population of candidate solutions is then conveniently stored as a list of subsets. This is typically initialized randomly:

```
> GA.init.pop <- function(popsize, nvar, kmin, kmax)
+ {
+   lapply(1:popsize,
+          function(ii, x, min, max) {
+            if (min == max) {
+              sample(x, min)
+            } else {
+              sample(x, sample(min:max, 1))
+            }},
+          nvar, kmin, kmax)
+ }
> pop1 <- GA.init.pop(pops = 5, nvar = 13, kmin = 2, kmax = 4)
> pop1
```

```
[[1]]
[1] 6 2 1 3

[[2]]
[1]  5 12  2 13

[[3]]
[1]  4 10

[[4]]
[1] 6 2

[[5]]
[1]  1 12  4
```

In this example, five different subsets containing two to four variables are generated. Each of these solutions can be directly evaluated with the same function as we already used in the SA implementation:

```
> pop1.q <-
+   sapply(pop1, function(subset) lda.loofun(X, C, subset))
> pop1.q
```

```
[1] 102 106 101  99 107
```

The two two-variable solutions do not work well and the last one, the three-variable subset is the best.

The *selection* function determines which of these solutions is allowed to reproduce, and is the driving force behind the optimization – if all solutions would have the same probability the result would be a random search. Typical selection procedures are to use random sampling with equal probabilities for all solution above a quality cutoff, or to use random sampling with (scaled) quality indicators as probability weights. The following function combines these concepts, and returns the indices of the candidate solutions that may become parents:

```
> GA.select <- function(pop, number, qlts,
+                       min.qlt = .4, qlt.exp = 1)
+ {
+   n <- length(pop)
+   qlts <- qlts - min(qlts)
+   threshold <- quantile(qlts, min.qlt)
+
+   weights <- rep(0, n)
+   weights[qlts > threshold] <-
+     qlts[qlts > threshold] ^ qlt.exp
+
+   sample(n, number, replace = TRUE, prob = weights)
+ }
> GA.select(pop1, 2, pop1.q, qlt.exp = 0)
```

```
[1] 2 5
```

In this case we have applied rank-threshold selection: all members of the population above the quality cutoff have the same chance of being selected. Providing a value for the parameter qlt.exp (the default is 1) will lead to a further bias towards the better solutions:

```
> GA.select(pop1, 2, pop1.q)
```

```
[1] 5 5
```

In this case, the function selects the same object (the best one) twice – there is no provision to prevent this, in accordance with general practice. If too many identical samples are selected, this is seen as a product of a selection pressure that is too high, something that should be corrected by either lowering the values for either the min.qlt or qlt.exp parameters. If too many bad samples are selected, the reverse applies.

A crucial difference between GAs and SA is that in a GA no problem-specific step function is needed: this is taken care of by the GA machinery of reproduction, typically consisting of two components, a crossover and a mutation. Both are random processes, not influenced by the quality of the solutions involved. The aim of the crossover is to combine elements of the parent solutions to obtain better children. An example implementation could be the following:

```
> GA.XO <- function(subset1, subset2)
+ {
+   n1 <- length(subset1)
+   n2 <- length(subset2)
+   if (n1 == n2) {
```

```
+     length.out <- n1
+   } else {
+     length.out <- sample(n1:n2, 1)
+   }
+   sample(unique(c(subset1, subset2)), length.out)
+ }
> GA.XO(pop1[[1]], pop1[[2]])
```

```
[1]  5  2 13  3
```

The result of combining to four-variable solutions is one child, again a four-variable solution. By repeatedly calling the selection and crossover operators, the next generation of bright young candidate solutions is gradually obtained.

Finally, a process of random mutation is active with a low probability, to maintain diversity in the population of candidate solutions. This also ensures that variables that were not selected in the initial population have a chance of entering the picture. The following function may remove a selected variable, add a previously unselected variable, or do both. The length of the mutated solution can be equal to the original, or differ by one. As with all other GA components, its machinery is based on chance:

```
> GA.mut <- function(subset, maxvar, mut.prob = .01)
+ {
+   if (runif(1) < mut.prob) { # swap variable in or out
+     new <- sample((1:maxvar)[-subset], 1)
+     length.out <- sample(-1:1 + length(subset), 1)
+
+     sample(c(new, subset), length.out)
+   } else {                   # do nothing
+     subset
+   }
+ }
> GA.mut(pop1[[1]], 13, 1)
```

```
[1] 3 1 2
```

In this case, just to show the effect, we have forced a mutation by setting the mutation probability to one. Obviously, having a high mutation rate will reduce the whole algorithm to random search, so the default mutation probability is usually very low.

Putting it all together leads to the following GA function:

```
> GAfun <- function(X, C, eval.fun, kmin, kmax,
+                    popsize = 20, niter = 50,
+                    mut.prob = .05, ...)
+ {
+   nvar <- ncol(X) # preparations: the first generation
+   pop <- GA.init.pop(popsize, nvar, kmin, kmax)
+   pop.q <- sapply(pop,
+                   function(subset) eval.fun(X, C, subset, ...))
+   best.q <- max(pop.q)
+   best <- pop[[which.max(pop.q)]]
+
+   for (i in 1:niter) { # Go!
+     new.pop <-
+       lapply(1:popsize, function(j) {
+         GA.mut(GA.XO(pop[[GA.select(pop, 1, pop.q)]],
+                      pop[[GA.select(pop, 1, pop.q)]]),
+                maxvar = nvar, mut.prob = mut.prob)}
+              ) # do crossover, and later perhaps mutation
+     pop <- new.pop
+     pop.q <- sapply(pop,
+                     function(subset) eval.fun(X, C, subset, ...))
+     if (max(pop.q) > best.q) { # bookkeeping of best solutions
+       best.q <- max(pop.q)
+       best <- pop[[which.max(pop.q)]]
+     }
+     if (length(unique(pop.q)) == 1) break # total convergence
+   }
+   list(best = best, best.q = best.q, n.iter = i)
+ }
```

In this function, the default number of evaluations, the number of generations times the population size, equals 1000, the same as in our proposed SA function. If the population has completely converged to solutions with the same fitness, the iteration stops.

So, how does this function perform? Let's find out using the wine data:

```
> GAobj <- GAfun(X, C, lda.loofun, kmin = 3, kmax = 3)
> GAobj

$best
[1] 7 10 12

$best.q
[1] 115

$n.iter
[1] 8
```

We see that the same quality value as the optimal SA solution is obtained with a slightly different subset. This problem is perhaps not the best example of the use of a GA: the number of variables is fairly low, and only after eight

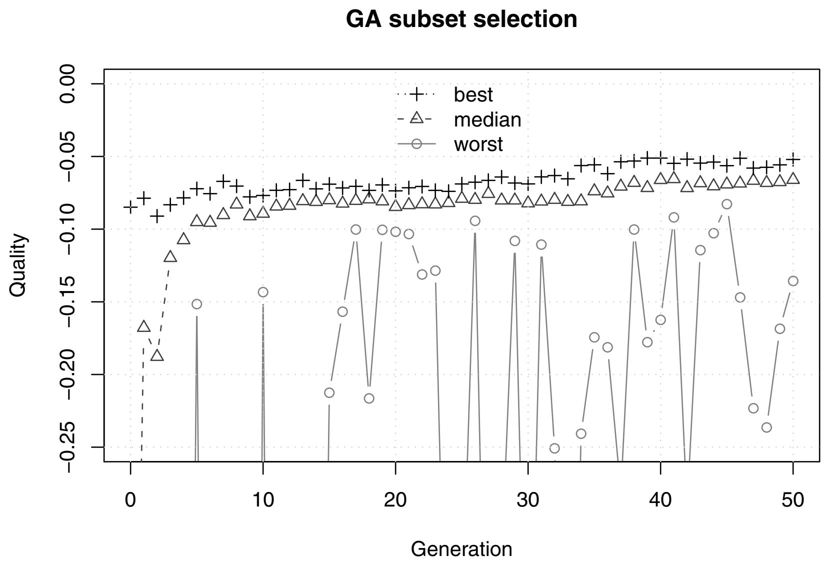

Fig. 10.6. Progression of the quality of the best, median and worst solution in the population during the GA-based subset selection for PLS-modelling of the gasoline data. The y-axis is limited to the interval $[-0.2, 0]$.

generations there is convergence, in the sense that all solutions have the same fitness.

A better example is the variable selection for the gasoline data. Again, the only thing we have to do is to plug in the appropriate evaluation function and perhaps tweak some of the settings:

```
> GAobj <- GAfun(gasoline$NIR, gasoline$octane, ncomp = 2,
+                  eval.fun = pls.cvfun, kmin = 3, kmax = 25)
> GAobj$best

 [1] 227 103 221 398  72 189  75 148  48 306 241 342
[13] 154  43  33 162   4   7  89 147  81 142  97 135

> sqrt(-GAobj2$best.q)

[1] 0.2260878
```

The result, using 24 variables, is slightly worse than obtained with the SA optimization but still much better than with the full set.

As with the SA example, it is sometimes useful to visualize the progress of the optimization. A plot of the best, median and worst members of the population is shown in Figure 10.6. Such a plot immediately shows the level of diversity in the population, which decreases during the optimization but is still appreciable in the last generation; much larger, in this case, than the difference between the overall best solution and the best solution of the first, random, generation. The plot shows a slightly upward trend, also in the later generations, so that better results may be obtained by continuing for more generations. Typically, population sizes of 50–100 candidate solutions are used, and the number of generations is usually several hundred to one thousand.

In more complicated problems, speed is a big issue. Some simple tricks can be employed to speed up the optimization. Typically, several candidate solutions will survive unchanged during a couple of generations. Rigorous bookkeeping may prevent unnecessary quality assessments, which in almost all cases is the most computer-intensive part of a GA. An implementational trick that is also very often applied is to let the best few solutions enter the next generation unchanged; this process, called *elitism*, makes sure that no valuable information is thrown away and takes away the need to keep track of the best solution. Provisions can be taken to prevent premature convergence: if the population is too homogeneous the power of the crossover operator decreases drastically, and the optimization usually will not lead to a useful answer. One strategy is to disallow offspring that is equal to other candidate solutions; a second strategy is to penalize the fitness of candidate solutions that are too similar; the latter strategy is sometimes called *sharing*.

The `genetic` function in the **subselect** package provides a fast Fortran-based GA. The details of the crossover and mutation functions are slightly different from the description above – indeed, there are probably very few implementations that share the exact same crossover and mutation operators, testimony to the flexibility and power of the evolutionary paradigm. Having seen the working of the `anneal` function, most input parameters will speak for themselves:

```
> wines.genetic <-
+   genetic(winesHmat$mat, kmin = 3, kmax = 5, nger = 20,
+           popsize = 50, maxclone = 0,
+           H = winesHmat$H, criterion = "ccr12", r = 1)
> wines.genetic$bestvalues
```

```
   Card.3    Card.4    Card.5
0.8328148 0.8436784 0.8524806
```

```
> wines.genetic$bestsets
```

```
       Var.1 Var.2 Var.3 Var.4 Var.5
Card.3     2     7    10     0     0
Card.4     2     3     7    10     0
Card.5     2     3     7    10    12
```

And indeed, the same three-variable solution is found as the optimal one. This time, also four- and five-variable solutions are returned (because of the values of the `kmin` and `kmax` arguments).

The `maxclone` argument tries to enforce diversity by replacing duplicate offspring by random solutions (which are not checked for duplicity, however). Leaving out this argument would, in this simple example, lead to a premature end of the optimization because of the complete homogeneity of the population. Both `anneal` and `genetic` provide the possibility of a further local optimization of the final best solution.

10.3.3 Discussion

Variable selection is a difficult process. Simple stepwise methods only work with a small number of variables, whereas the largest gains can be made in the nowadays typical situation of hundreds or even thousands of variables. More complicated methods containing elements of random search, such as SA or GA approaches, can have a high variability, especially in cases where correlations between variables are high. One approach is to repeat the variable selection multiple times, and to use those variables that are consistently selected. Although this strategy is intuitively appealing, it does have one flaw: suppose that variables a and b are highly correlated, and that a combination of either a or b with a third variable c leads to a good model. In repeated selection runs, c will typically be selected twice as often as a or b – if the overall selection threshold is chosen to include c but neither of a and b, the model will not work well.

In addition, the optimization criterion is important. It has been shown that LOO crossvalidation as a criterion for variable selection is inconsistent, in the sense that even with an infinitely large data set it will not choose the correct model [134]. Baumann *et al.* advocate the use of leave-multiple-out crossvalidation for this purpose [135,136], even though the computational burden is high. In this approach, the data are repeatedly split, randomly, in training and test sets, where the number of repetitions needs to be greater than the number of variables, and for every split a separate crossvalidation is performed to optimize the parameters of the modelling method such as the number of latent variables in PCR or PLS. A workable alternative is to fix the number of latent variables to a “reasonable” number, and to find the subset of variables that with this particular setting leads to the best results. This takes away the nested crossvalidation but may lead to subsets that are suboptimal. In general, one should accept the fact that there is no guarantee that the optimal subset will be found, and it is wise to accept a subset that is “good enough”.

Part V

Applications

11

Chemometric Applications

This chapter highlights some typical examples of research themes in the chemometrics community. Up to now we have concentrated on fairly general techniques, found in many textbooks and applicable in a wide range of fields. Several other topics, some of them more specific to the field of chemometrics, combine elements from the previous chapters and warrant a separate section. We start with two examples dealing with outliers and noise, respectively. *Robust PCA* is an attractive method to identify outliers in multivariate space, at a modest computational cost. *Orthogonal Signal Correction* and its combination with PLS, *OPLS*, is a way to remove variation in the data that is irrelevant for predicting the dependent variable. In some cases this leads to simpler models that are easier to interpret. Discriminant analysis with PLS, *PLSDA*, is an important technique for doing classification in high-dimensional data sets. However, there are certain risks involved. In analytical laboratories, there is often a need to develop calibration models that can be transferred across a range of instruments. One example is to develop a model using a laboratory, high-quality setup, and then to apply the model for in-line measurements of a much lower quality. The approach to achieve this has become known as *calibration transfer*. Finally, we take a look at a decomposition of a matrix $\boldsymbol{X}$ where the individual components are directly interpretable, e.g. as concentration profiles or spectra of pure compounds: *Multivariate Curve Resolution.*

11.1 Outlier Detection with Robust PCA

Identifying outliers is always an extremely difficult and dangerous task, prone to subjective judgements. The danger of introducing bias by rejecting samples that do not fit in with the expected pattern is always present. At the same time, however, outlying samples *will* occur in practice: whole microarrays with expressions of tens of thousands of genes can be useless because of some experimental artifact, and including them could be detrimental to the results. One

R. Wehrens, *Chemometrics with R: Multivariate Data Analysis in the Natural Sciences and Life Sciences*, Use R!, DOI 10.1007/978-3-642-17841-2_11,

of the problems obviously is that in high-dimensional space, every object of a small-to-medium-sized data set can be seen as an outlier. Only if the samples are occupying a restricted subspace can we have any hope of performing a meaningful outlier detection.

11.1.1 Robust PCA

Of course, we know a tool to find such a subspace: PCA. What would be easier than to apply PCA to the data, and see the outliers far away from the bulk of the data? Although this sometimes does happen, and PCA in these cases is a valuable outlier detector, in other cases the outliers are harder to spot. The point is that PCA is not a robust method: since it is based on the concept of variance, outliers will greatly influence scores and loadings, sometimes even to the extent that they will dominate the first PCs. What is needed in such cases is a *robust* form of PCA [137]. Many different approaches exist, each characterized by their own *breakdown point*: the fraction of outliers that can be present without influencing the covariance estimates.

The simplest form is to perform the SVD on a robust estimate of the covariance or correlation matrix [138]. One such estimate is given by the Minimum Covariance Determinant (MCD, [139]), which has a breakdown point of up to .5, meaning that half of the data can be "wrong" without affecting the estimate. Higher breakdown points than .5 obviously do not make too much sense. As the name already implies, the MCD estimator basically samples subsets of the data of a specific size, in search of the subset that leads to a covariance matrix with a minimal determinant, i.e. covering the smallest hypervolume. The assumption is that the outlying observations are far away from the other data points, increasing the volume of the covariance ellipsoid. The size of the subset, to be chosen by the user, determines the breakdown point, given by $(n - h + 1)/n$, with n the number of observations and h the size of the subset. Unless one really expects a large fraction of the data to be contaminated, it is recommended to choose $h \approx .75n$. The resampling approach can take a lot of time, and although fast algorithms are available [140], matrices with more than a couple of hundred variables remain hard to tackle.

The MCD covariance estimator is available in several R packages. One example is `cov.mcd` in package **MASS**. If we use this in combination with the `princomp` function, we can see the difference between robust and classical covariance estimation. Let's focus on the Grignolino samples from the wine data:

```
> X <- wines[vintages == "Grignolino",]
> X.sc <- scale(X)
> X.clPCA <- princomp(X.sc)
> X.robPCA <- princomp(X.sc, covmat = cov.mcd(X.sc))
> biplot(X.clPCA, main = "Classical PCA")
> biplot(X.robPCA, main = "MCD-based PCA")
```

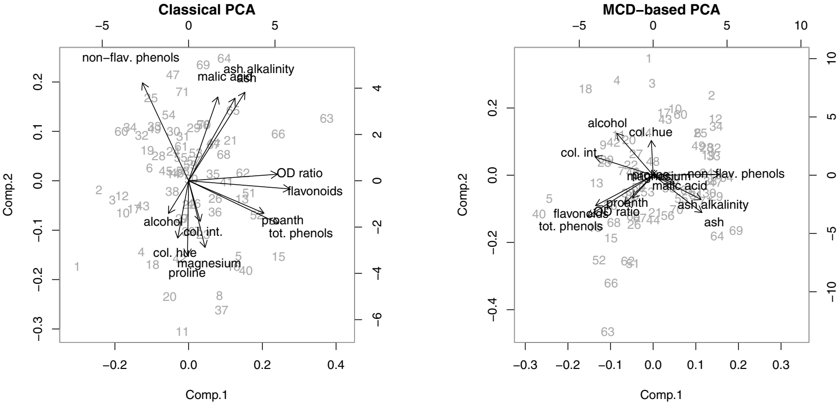

Fig. 11.1. Biplots for the Grignolino samples: the classical PCA solution is shown on the left, whereas the right plot is based on the MCD covariance estimate.

This leads to the biplots in Figure 11.1. There are clear differences in the first two PCs: in the classical case PC 1 is dominated by the variables `OD ratio`, `flavonoids`, `proanth` and `tot. phenols`, leading to samples 63, 66, 15 and 1, 2, and 3 to having extreme coordinates. In the robust version, on the other hand, these samples have very relatively small PC 1 scores. Rather, they are extremes of the second component, the result of increased influence of variables (inversely) correlated with `ash` on the first component. Although many of the relations in the plots are similar (the main effect seems to be a rotation), the example shows that even in cases where one would not expect it applying (more) robust methods can lead to appreciable differences.

An important impediment for the application of the MCD estimator is that it can only be calculated for non-fat data matrices, i.e. matrices where the number of samples is larger than the number of variables – in other cases, the covariance matrix is singuar, with a determinant of zero. In such cases another approach is necessary. One example is ROBPCA [141], combining Projection Pursuit and robust covariance estimation: PP is employed to find a subspace of lower dimension in which the MCD estimator can be applied. ROBPCA has one property that we also saw in ICA (Section 4.6.2): if we increase the number of PCs there is no guarantee that the first PCs will remain the same – in fact, they usually are not. Obviously, this can make interpretation somewhat difficult, especially since the method to choose the "correct" number of PCs is less obvious in robust PCA than in classical PCA [137].

Since the details of the ROBPCA algorithm are a lot more complicated than can be treated here, we just illustrate its use. ROBPCA, as well as several other robust versions of PCA, is available in package **rrcov** as the function `PcaHubert`. Application to the Grignolino samples using five PCs leads to the following result:

```
> X.HubPCA5 <- PcaHubert(X.sc, k = 5)
> summary(X.HubPCA5)

Call:
PcaHubert(x = X.sc, k = 5)
Importance of components:
                        [,1]  [,2]  [,3]  [,4]  [,5]
Standard deviation      1.59 1.317 1.087 0.891 0.871
Proportion of Variance 0.36 0.248 0.169 0.114 0.109
Cumulative Proportion  0.36 0.608 0.778 0.891 1.000
```

Note that the final line gives the cumulative proportion of variance as a fraction of the variance captured in the robust PCA model, and not as the fraction of the total variance, usual in classical PCA. If we do not provide an explicit number of components (the default, `k = 0`) the algorithm chooses the optimal number itself:

```
> X.HubPCA <- PcaHubert(X.sc)
> summary(X.HubPCA)

Call:
PcaHubert(x = X.sc)
Importance of components:
                         [,1]  [,2]  [,3]  [,4]  [,5]  [,6]  [,7]
Standard deviation     1.541 1.295 1.124 0.958 0.871 0.759 0.531
Proportion of Variance 0.303 0.214 0.161 0.117 0.097 0.073 0.036
Cumulative Proportion  0.303 0.516 0.677 0.794 0.891 0.964 1.000
```

Apparently this optimal number equals seven in this case. The rule-of-thumb to calculate the "optimal" number of components is based on the desire to explain a significant portion of the variance explained by the model (a fraction of .8 is used as the default) while not taking into account components with very small standard deviations – the last component of the model should have an eigenvalue at least .1% of the largest one. If the number of variables is small enough, the MCD algorithm is used directly; if not, the ROBPCA algorithm is used. One can force the use of ROBPCA by setting `mcd = FALSE`. Note that the standard deviations of the first components are not the same as the ones calculated for the five-component model.

The default plotting method is different from the classical plot: it shows an outlier map, or distance-distance map, rather than scores or loadings. The main idea of this plot is to characterise every sample by two different distances:

- the Orthogonal Distance (OD), indicating the distance between the true position of every data point and its projection in the space of the first few PCs;
- the Score Distance (SD), or the distance of the sample projection to the center of all sample projections.

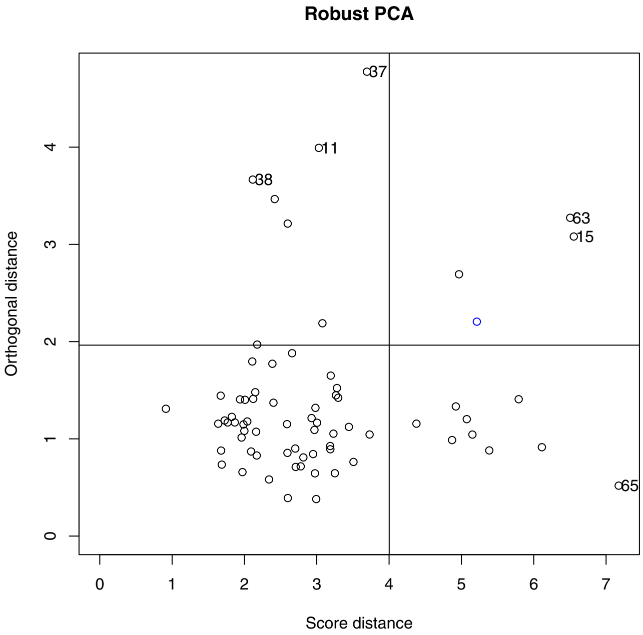

Fig. 11.2. Outlier map for the Grignolino data, based on a seven-component ROBPCA model.

Or, put differently, the SD of a sample is the distance to the center, measured in the hyperplane of the PCA projection, and the OD is the distance to this hyperplane. Obviously, both SD and OD depend on the number of PCs. When a sample is above the horizontal threshold it is too far away from the PCA subspace; when it is to the right from the vertical threshold it is too far from the other samples *within* the PCA subspace. The horizontal and vertical thresholds are derived from χ^2 approximations [142].

For the Grignolino data, this leads to the plot in Figure 11.2:

```
> plot(X.HubPCA)
```

Several of the most outlying samples are indicated with their indices, so that they can be inspected further. Also a biplot method is available, which shows a plot that is very similar to the right plot in Figure 11.1. Inspection of the data shows that objects 63 and 15 do contain some extreme values in some of the variables – indeed, object 63 is also the object with the smallest score on PC 1 in a classical PCA. However, it would probably be too much to remove them from the data completely.

11.1.2 Discussion

A robust approach can be extremely important in cases where one suspects that some of the data are outliers. Classical estimates can be very sensitive to extreme values, and it frequently occurs that only one or very few samples dominate the rest of the data. This need not be an error, because influential observations may be correct, but in general one would put more trust in a model that is based on many observations rather than a few. This is not in contradiction with the desire to build sparse models, as seen in the section on SVMs, for example: there, the sparseness was obtained by selecting only those objects in the relevant part of the space, using all other objects in the selection process.

The robust methods in this section have a wider applicability than just outlier detection: they can be used as robust plugin estimators in classification and regression methods. Robust LDA can be obtained, for example, by using robust estimate of the pooled covariance matrix; robust QDA by robust covariances for all classes. PCR can be robustified in several ways, e.g., by applying SVD to a robust covariance matrix estimate; an alternative is formed by regressing on robust scores, for instance from the ROBPCA algorithm. One can even replace the least squares regression by robust regression methods such as least trimmed squares [139]. Also robust versions of PLS regression exist [143,144]. These robust versions of classification and regression methods share the big advantage that one can safely leave in all objects, even though some of them may be suspected outliers: the analysis will not be influenced by only a couple atypical observations. And to turn the question of outliers around: if robust and classical analyses give the same or similar results, then one can conclude that there are no (influential) outliers in the data.

R contains many packages with facilities for robust statistics, the most important one probably being **robustbase**. According to the taskview on CRAN, plans exist to further streamline the available packages, using **robustbase** as the basic package for robust statistics, and several more specialized packages building on that, such as is the case already for packages like **rrcov**.

11.2 Orthogonal Signal Correction and OPLS

Orthogonal Signal Correction (OSC) was first proposed by Wold and coworkers [145] with the aim to remove information from $\boldsymbol{X}$ that is orthogonal to $\boldsymbol{Y}$. Several different algorithms have been proposed in literature – a concise comparison of several of them has appeared in [146]. The conclusion of that paper is that OSC in essence does not improve prediction quality per se but rather leads to more parsimonious models, that are easier to interpret. Moreover, the part of $\boldsymbol{X}$ that has been removed before modelling can be inspected as well and may provide information on how to improve measurement quality.

As an example, we will show one form of OSC, called Orthogonal Projection to Latent Structures (OPLS, [147]), as summarized in [146]. Using the weights w and loadings p from an initial PLS model, the variation orthogonal to the dependent variable is obtained and subtracted from the original data matrix:

$$w_\perp = p - \frac{w^T p}{w^T w} w \tag{11.1}$$

$$t_\perp = \boldsymbol{X} w_\perp \tag{11.2}$$

$$p_\perp = \frac{\boldsymbol{X}^T t_\perp}{t_\perp^T t_\perp} \tag{11.3}$$

$$\boldsymbol{X}_c = \boldsymbol{X} - t_\perp^T p_\perp \tag{11.4}$$

The corrected matrix $\boldsymbol{X}_c$ is then used in a regular PLS model. If desired, more orthogonal components can be extracted – it is claimed that for univariate $\boldsymbol{Y}$ only one PLS component is required in the final model.

Let us see how this works out for the gasoline data. From Figure 8.3 we have concluded that, based on a training set consisting of the odd rows of the `gasoline` data frame, three PLS components are needed. To make things easier, we start by mean-centering the spectra, based on the mean of the training data only. The OSC-corrected matrix is then obtained as follows:

```
> gasoline$NIR <- scale(gasoline$NIR, scale = FALSE,
+                       center = colMeans(gasoline$NIR[odd,]))
> gasoline.pls <- plsr(octane ~ ., data = gasoline,
+                      ncomp = 5, subset = odd,
+                      validation = "LOO")
> ww <- gasoline.pls$loading.weights[,1]
> pp <- gasoline.pls$loadings[,1]
> w.ortho <- pp - crossprod(ww, pp)/crossprod(ww) * ww
> t.ortho <- Xtr %*% w.ortho
> p.ortho <- crossprod(Xtr, t.ortho) / c(crossprod(t.ortho))
> Xcorr <- Xtr - tcrossprod(t.ortho, p.ortho)
```

Next, a new PLS model is created using the corrected data matrix:

```
> gasoline.osc1 <- data.frame(octane = gasoline$octane[odd],
+                             NIR = Xcorr)
> gasoline.opls1 <- plsr(octane ~ ., data = gasoline.osc1,
+                        ncomp = 5, validation = "LOO")
```

Removal of a second OSC component proceeds along the same lines:

```
> pp2 <- gasoline.opls1$loadings[,1]
> w.ortho2 <- pp2 - crossprod(ww, pp2)/crossprod(ww) * ww
> t.ortho2 <- Xcorr %*% w.ortho2
> p.ortho2 <- crossprod(Xcorr, t.ortho2) / c(crossprod(t.ortho2))
> Xcorr2 <- Xcorr - tcrossprod(t.ortho2, p.ortho2)
```

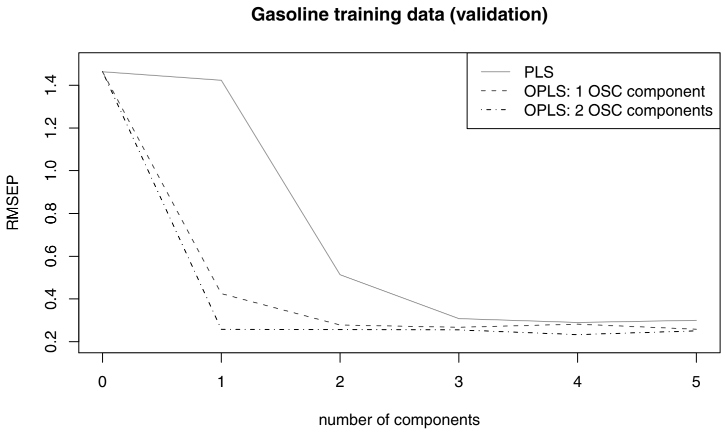

Fig. 11.3. Crossvalidation results for the gasoline data (training set only): removal of one or two orthogonal components leads to more parsimonious PLS models.

```
> gasoline.osc2 <- data.frame(octane = gasoline$octane[odd],
+                             NIR = Xcorr2)
> gasoline.opls2 <- plsr(octane ~ ., data = gasoline.osc2,
+                        ncomp = 5, validation = "LOO")
```

Note that the ww vector is the same for every component that is removed [147]. We now can compare the validation curves of the regular PLS model, the PLS model having one orthogonal component removed, and the PLS model with two components removed:

```
> plot(gasoline.pls, "validation", estimate = "CV",
+      ylim = c(0.2, 1.5),
+      main = "Gasoline training data (validation)")
> lines(0:5, c(RMSEP(gasoline.opls1, estimate = "CV"))$val,
+       col = 2, lty = 2)
> lines(0:5, c(RMSEP(gasoline.opls2, estimate = "CV"))$val,
+       col = 4, lty = 4)
> legend("topright", lty = c(1,2,4), col = c(1,2,4),
+        legend = c("PLS", "OPLS: 1 OSC component",
+                   "OPLS: 2 OSC components"))
```

The result is shown in Figure 11.3. Clearly, the best prediction errors for the two OPLS models are comparable (even slightly better) to the error in the original model using three components, and the optimal values are reached with fewer latent variables.

To do prediction, one has to deflate the new data in the same way as the training data; i.e., one has to subtract the orthogonal components before presenting the data to the final PLS model.

```
> Xtst <- gasoline$NIR[even,]
> t.tst <- Xtst %*% w.ortho
> p.tst <- crossprod(Xtst, t.tst) / c(crossprod(t.tst))
> Xtst.osc1 <- Xtst - tcrossprod(t.tst, p.tst)
> gasoline.opls1.pred <- predict(gasoline.opls1,
+                                newdata = Xtst.osc1,
+                                ncomp = 2)
```

Predictions for the OPLS model with two OSC components removed are obtained in the same way:

```
> t.tst2 <- Xtst.osc1 %*% w.ortho2
> p.tst2 <- crossprod(Xtst.osc1, t.tst2) / c(crossprod(t.tst2))
> Xtst.osc2 <- Xtst.osc1 - tcrossprod(t.tst2, p.tst2)
> gasoline.opls2.pred <- predict(gasoline.opls2,
+                                newdata = Xtst.osc2,
+                                ncomp = 1)
```

We can now compare the RMSEP values for the different PLS models:

```
> RMSEP(gasoline.pls, newdata = gasoline[even,],
+       ncomp = 3, intercept = FALSE)

[1]  0.2093

> rms(gasoline$octane[even], gasoline.opls1.pred)

[1] 0.3790201

> rms(gasoline$octane[even], gasoline.opls2.pred)

[1] 0.4488759
```

Although the crossvalidation errors do not increase, the prediction of the unseen test data deteriorates quite a bit.

11.3 Discrimination with Fat Data Matrices

“Fat” data matrices, or data sets with many more variables than objects, are becoming the rule rather than the exception in the natural sciences. Although this means that a lot of information is available for each sample, it also means in practice that a lot of numbers are available that do not say anything particularly interesting about the sample – these can be pure noise, but also genuine signals, unrelated to the research question at hand. Another problem is the correlation that is often present between variables. Finding relevant differences between classes of samples in such a situation is difficult: the number of parameters to estimate in regular forms of discriminant analysis far exceeds the number of independent samples available. An example is the `prostate` data set, containing more than 10,000 variables and only 327 samples. We could eliminate several without losing information, and also removing variables that are “clearly” not related to the dependent variable (in as far as we would be able to recognise these) would help, the idea that is formalized in the OPLS approach from the previous section. An alternative is formed by variable selection techniques, such as the ones described in Chapter 10, but these usually rely on accurate error estimates that are hard to get with low numbers of samples.

The same low number of samples also forces the statistical models to have very few parameters: fat matrices can be seen as describing sparsely – very sparsely – populated high-dimensional spaces, and only the simplest possible models have any chance of gaining predictive power. The simplest possible case is that of linear discriminant analysis, but direct calculation of the coefficients is impossible because of the matrix inversion in Equation 7.6 – the covariance matrix is singular. Regularization approaches like RDA are one solution; the extreme form of regularization, diagonal LDA, enjoys great popularity in the realm of microarray analysis. Another often-used strategy is to compress the information in a much smaller number of variables, usually linear combinations of the original set, and perform simple methods like LDA on the new, small, data matrix. Two approaches are popular: PCDA and PLSDA.

11.3.1 PCDA

One way to compress the information in a fat data matrix into something that is more easy to analyse is PCA. Subsequently, LDA is performed on the scores – the result is often referred to as PCDA or PCLDA. The prostate data provide a nice example: the number of variables far exceeds the number of samples, even though that number is not low in absolute terms. Again, we are trying to discriminate between control samples and cancer samples, so we consider only two of the three classes. Now, however, we use all variables – in the cases of SVMs and boosting this would have led to large memory demands, but the current procedure is much more efficient. Using the SVD on the crossproduct matrix of the non-bph samples of the prostate data, similar to the procedure shown on page 48, we obtain scores and loadings. The first sixteen PCs cover just over 70% of the variance of the $\boldsymbol{X}$ matrix, not too surprising given the number of variables. We should have a look at the scores, for clarity limiting ourselves to the first five components:

```
> pairs(prost.scores, pch = as.integer(prost.type),
+       col = as.integer(prost.type),
+       labels = paste("PC", 1:5))
```

The result is shown in Figure 11.4. Although some interesting structure is visible, there is no obvious separation between the classes in any of the plots. This five-dimensional representation of the data can be used in any form of discriminant analysis; we will stick to LDA, and just to get a feeling for what

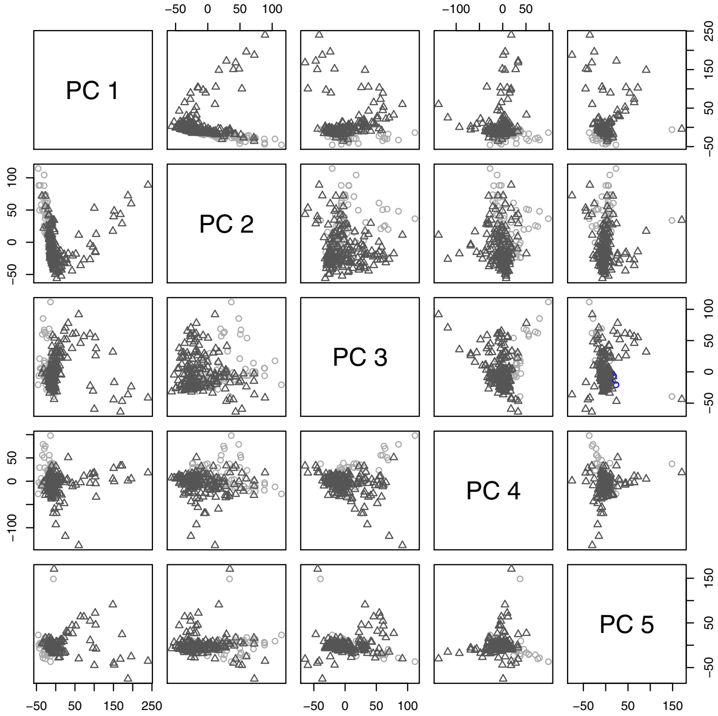

Fig. 11.4. Pairs plot of the scores of the prostate data in the first five PCs.

we can hope to expect, we will use five PCs. The naive, and as we shall see later, incorrect approach would be the following:

```
## INCORRECT
> prost.pcda5 <- lda(prost.type ~ prost.scores[,1:5], CV = TRUE)
> table(prost.type, prost.pcda5$class)
```

```
prost.type control pca
   control      44  37
   pca          12 156
```

Leave-one-out crossvalidation leads to a correct prediction in over 80% of the cases, better than we might have expected on the basis of Figure 11.4. One should not forget, however, that the cancer class is more than twice the size of the control class, so that already a not-too-clever random classification of cancer for all samples would lead to a success rate of over 65%. Note that the prediction errors are slightly unbalanced: more control samples are predicted to be cancer than vice versa. This is the result of the default prior of the `lda` function, which is proportional to the class representation in the training set.

As already stated, the above procedure is incorrect: the error estimate is optimistically biased because the PCA step (including mean-centering and scaling) has not been incorporated in the crossvalidation. As it is now, the left-out sample still exerts influence on the classification model through its contribution to the PCs, whereas in the correct way, the crossvalidation should include the PCA. This can be done by using an explicit crossvalidation loop, leaving out part of the samples, performing PCA and building the LDA model, but a more easy approach is to see the classification as a regression problem and use the `pcr` function, with its built-in crossvalidation facilities. While we are at it, we should also separate training data from test data, in order to get some kind of estimate for the prediction error, as well as the optimal number of latent variables. Here goes:

```
> odd <- seq(1, length(prost.type), by = 2)
> even <- seq(2, length(prost.type), by = 2)
> prost.df <- data.frame(class = as.integer(prost.type),
+                        msdata = I(prost))
> prost.pcr <- pcr(class ~ msdata, ncomp = 16,
+                  data = prost.df, subset = odd,
+                  validation = "CV", scale = TRUE)
> validationplot(prost.pcr, estimate = "CV")
```

In this case the validation (by default a ten-fold crossvalidation) is done correctly: the scaling and the PCA are done only *after* the out-of-bag samples are removed from the training data. This leads to the left plot in Figure 11.5 – a cautious person would perhaps select six PCs here, but since the number of samples is quite large, one might even consider twelve PCs. Your results may differ because of the randomness involved in choosing the crossvalidation segments.

Note that the RMSEP measure shown here is not quite what we are interested in: rather than the devation from the ideal values of 0 and 1 in the classification matrix, we should look at the number of correct classifications.

```
> prost.trn <- predict(prost.pcr)
> prost.trn.cl <- round(prost.trn[,1,])
> prost.trn.err <- apply(prost.trn.cl, 2,
+                        err.rate, prost.df$class[odd])
> plot(1 - prost.trn.err, col = "gray", type = "l",
+      xlab = "# PCs"ylab = "Misclassif. rate")
```

The result is shown in the right plot of Figure 11.5. Although the RMS values in the left plot go up with higher number of PCs, the number of misclassifications does not The training data show a reasonable performance, with a crossvalidated error just below 15% for a model taking into account six PCs. Of course we can do the same for the test set:

```
> prost.tst <- predict(prost.pcr, newdata = prost.df[even,])
> prost.tst.cl <- round(prost.tst[,1,])
```

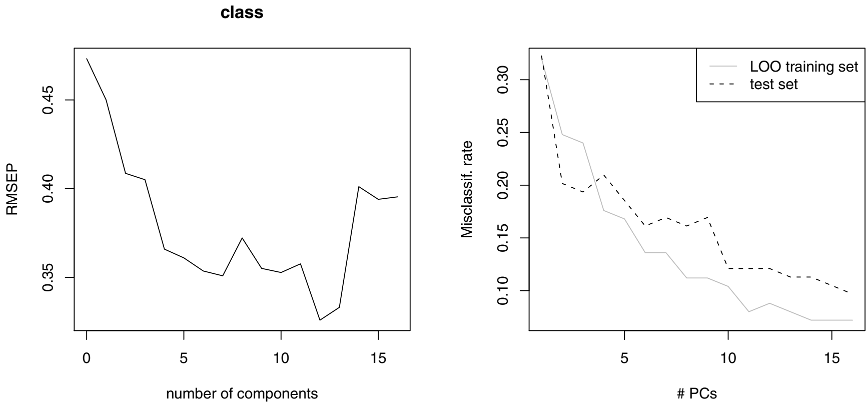

Fig. 11.5. Validation plots for the PCR regression plot of the prostate MS data: discrimination between `control` and `pca` samples. Left plot: RMSEP values for the class codes. Right plot: fraction of misclassifications.

```
> prost.tst.err <- apply(prost.tst.cl, 2,
+                        err.rate, prost.df$class[even])
> lines(prost.tst.err, lty = 2)
```

Given this crossvalidation curve, the optimal value seems to be around eleven PCs. It is good to see that the crossvalidation estimates are reasonably close to the test set results: this is an indication that there is no overfitting. Obviously, one should not look at the results of the test data in choosing the optimal number of PCs! The test data show that with eleven PCs the expected classification error is around twelve percent.

A big advantage of this approach is that it can be applied to multiclass problems directly. In the PCR-based implementation, one just needs to convert the class vector into a class-membership matrix, with one column per class, and run the algorithm:

```
> prostate.clmat <- classvec2classmat(prostate.type)
> prostate.df <- data.frame(class = I(prostate.clmat),
+                           msdata = I(prostate))
> prostate.pcr <- pcr(class ~ msdata, ncomp = 16,
+                     data = prostate.df, subset = odd,
+                     validation = "CV", scale = TRUE)
```

Again, we should convert the predicted values in the PCR crossvalidation to classes, and plot the number of misclassifications so that we can pick the optimal number of components:

```
> predictions.loo <-
+   sapply(1:16, function(i, arr) classmat2classvec(arr[,,i]),
+          prostate.pcr$validation$pred)
> loo.err <- apply(predictions.loo, 2, err.rate,
+                  prostate.type[odd])
> plot(loo.err, type = "l", main = "PCDA", col = "gray",
+      ylim = c(.2, max(loo.err)),
+      xlab = "# PCs", ylab = "Misclassif. rate")
```

This results in the gray solid line in Figure 11.6 – the classification error is quite high for all numbers of components. Perhaps eight components can be picked as the least bad model. The test set results can be obtained in a completely analogous way, and are shown as the black dashed lines in the same plot:

```
> prostate.pcrpred <-
+   predict(prostate.pcr, new = prostate.df[even,])
> predictions.pcrtest <-
+   sapply(1:16, function(i, arr) classmat2classvec(arr[,,i]),
+          prostate.pcrpred)
> lines(apply(predictions.pcrtest, 2, err.rate,
+             prostate.type[even]),
+       type = "l", lty = 2)
```

We could ask ourselves what is going wrong:

```
> table(prostate.type[even], predictions.pcrtest[,8])
```

```
          bph control pca
  bph       4       3  32
  control   0      30  10
  pca       1       8  75
```

There is considerably confusion between the `pca` and `bph` classes: almost all `bph` objects are classified as `pca`. The controls are separated relatively well.

11.3.2 PLSDA

Although the above approach often is reported to work well in practice, it has a (familiar) flaw: there is no reason to assume that the information relevant for the class discrimination is captured in the *first* PCs. Since PLS takes into account the dependent variable when defining latent variables, this is a logical alternative. In literature, this form of discriminant analysis, usually done in the form of a direct regression on coded class variables, is called PLSDA [148]. For the prostate data, this leads to:

```
> prostate.pls <- pls(prostate.classmat ~ prostate,
+                      ncomp = 16, validation = "CV")
```

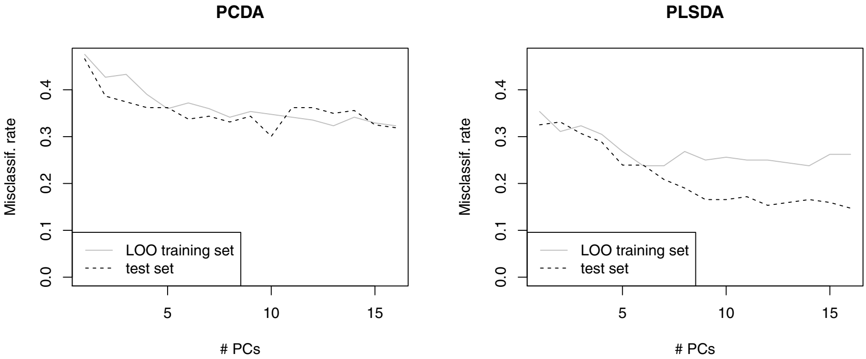

Fig. 11.6. PCDA (left) and PLSDA (right) classification results for the three-class prostate data.

Using code that is completely analogous to the PCDA case on the previous pages, we arrive at the right plot in Figure 11.6. As expected, fewer components are needed for optimal results – six would be selected in the case of PLSDA, whereas PCR would need eight. The PLSDA error values are considerably lower than with PCR, both for the LOO crossvalidation and for the test data: already with one PLS component the prediction of the test data is better than the PCDA model achieves with eight. With six components, PLSDA prediction of the test set looks like this:

```
> table(prostate.type[even], predictions.plstest[,6])

          bph control pca
  bph      25       1  13
  control   1      33   6
  pca      13       5  66
```

Clearly, the confusion between the bph and pca classes has decreased significantly.

An alternative is to perform a classical LDA on the PLS scores – in several papers (e.g., [148, 149, 150]) this is claimed to be superior in quality. Where performing LDA on the first couple of PCs is completely analogous to doing PCR on the class matrix (for equal class sizes, at least), this is not the case in PLS, since class knowledge is used in defining latent variables. One therefore may expect some differences. First we should calculate the scores of the test set, which is done by post-multiplying the (scaled) test data with the projection matrix in the PLS object:

```
> Xtst <- scale(prostate[even,],
+               center = colMeans(prostate[odd,]),
+               scale = sd(prostate[odd,]))
> tst.scores <- Xtst %*% prostate.pls$projection
```

Next, we can build an LDA model on the scores of the training set, directly available with the `scores` extractor function, and use this model to make predictions for the test set:

```
> prostate.ldapls <- lda(scores(prostate.pls)[,1:6],
+                        prostate.type[odd])
> table(prostate.type[even],
+       predict(prostate.ldapls, new = tst.scores[,1:6])$class)

          bph control pca
  bph      23       0  16
  control   1      34   5
  pca      15       4  65
```

Compared to the direct PLSDA method, two more misclassifications are obtained. For this case at least, the differences are small.

A Word of Warning

Because it is more focused on information in the dependent variable, PLS can be called a more greedy algorithm than PCR. In many cases this leads to a better fit for PLS (with the same number of components as the PCR model, that is), but it also presents a bigger risk of overfitting. The following example will make this clear: suppose we generate random data from a normal distribution, and allocate every sample randomly to one of two possible classes:

```
> nvar <- 2000
> nobj <- 40
> RandX <- matrix(rnorm(nobj*nvar), nrow = nobj)
> RandY <- sample(c(0, 1), nobj, replace = TRUE)
```

Next, we compress the information in variable `RandX` into two latent variables, so that LDA can be applied. We use both PCA[1] and PLS. The results are quite interesting:

```
> Rand.pcr <- pcr(RandY ~ RandX, ncomp = 2)
> Rand.ldapcr <- lda(RandY ~ scores(Rand.pcr), CV = TRUE)
> table(RandY, Rand.ldapcr$class)

RandY  0  1
    0  8 12
    1 13  7
```

[1] For simplicity, we employ the `pcr` function from the **pls** package so that the results can be directly compared with the results from the `plsr` function.

```
> Rand.pls <- plsr(RandY ~ RandX, ncomp = 2)
> Rand.ldapls <- lda(RandY ~ scores(Rand.pls), CV = TRUE)
> table(RandY, Rand.ldapls$class)
```

```
RandY  0  1
    0 20  0
    1  0 20
```

Where the PCA compression leads to results that are reasonably close to the expected 50-50 prediction, PLS-LDA leads to perfect predictions. For random data, that is not exactly what we would want! Note that we did not perform any optimization of the number of latent variables employed – in this case, choosing two latent variables is already enough to be in deep trouble. Validation plots for the regression models *would* have shown that there is trouble ahead – in both the PCR and PLS case, zero latent variables would appear optimal. The moral of the story should by now sound familiar: especially in cases with low ratios of numbers of objects to numbers of variables, one should be very, very careful...

11.4 Calibration Transfer

Imagine a company with high-quality expensive spectrometers in the central lab facilities, and cheap simple equipment on the production sites, perhaps in locations all over the world – you can easily see why a calibration model set up with data from the better instruments (the "master" instruments) may do a better job in capturing the essentials from the data. In general, however, it is not possible to directly use the "good" model for the inferior ("slave") instruments: there will be systematic differences, such as different wavelength ranges and resolutions, as well as some less-clear ones; every measuring device has its own characteristics. Predictions directly using the model from the master instrument, if possible at all, will not always be of a high quality.

Constructing individual calibration models for every instrument separately is of course the best strategy to avoid incompatibilities. However, this may be difficult because calibration samples may not be available, or very expensive. Ideally then, one would want to use the model of the master instrument for the slave spectrometers, too. Several ways of achieving this have been proposed in the literature [151]. For one thing, one may try to iron out the differences between the slaves and the master by careful preprocessing, usually including variable selection. In this way, parts of the data that are not too dependent on the instrument – but are still relevant for calibration purposes – can be utilized. This is sometimes called *robust calibration* in chemometrics literature, and is not to be confused with robust regression as is known in statistics.

Another approach has become known under the phrase *calibration transfer* or *calibration standardization* [152]. The goal is to use a limited number of common calibration samples on the slave instruments to adapt the model developed on the master. This strategy is also useful in cases where instrument response changes over time, or when batch-to-batch differences occur. Although a complete recalibration is always better, it has been reported that an approach using standardization leads to an increase in errors by only 20–60%, which in practice may still be perfectly adequate.

Several variants of calibration transfer have been published (see, e.g., [153, 154]). The simplest is *direct standardization* (DS, [152]). For a set of samples measured on both instruments, one constructs a transformation $\boldsymbol{F}$ relating the measurements on the master instrument, $\boldsymbol{X}_1$, to the measurements on the slave $\boldsymbol{X}_2$:

$$\boldsymbol{X}_1 = \boldsymbol{X}_2 \boldsymbol{F} \tag{11.5}$$

The transformation can be easily obtained by a generalized inverse:

$$\boldsymbol{F} = \boldsymbol{X}_2^+ \boldsymbol{X}_1 \tag{11.6}$$

Alternatives are to use PCR or PLS regression, which are less prone to overfitting. Thus, responses measured on the slave instrument can then be transformed to what they would look like on the primary instrument. Consequently, the calibration model of the primary instrument applies – this approach is also known as backward calibration transfer. The reverse (forward transfer) is also possible: spectra of the high-resolution master instrument can be transformed to be similar to the data measured on the lower-quality slaves. The latter option is usually preferred in on-line situations, where speed is an issue.

To illustrate this, we will use data presented at the 2002 Chambersburg meeting,[2] available from the **ChemometricsWithR** package. It consists of NIR data of 654 pharmaceutical tablets, measured at two Multitab spectrometers. Each tablet should contain 200 mg of (undisclosed) active ingredient. The data have already been divided into training, validation and test sets. The complete wavelength range is from 600 to 1898 nm; we will use the first-derivatives of the area between 1100 and 1700 nm (`calibrate.1`). Let us first build a PLS model for the training data, using the **pls** package:

```
> data(shootout)
> wl <- seq(600, 1898, by = 2)
> indices <- which(wl >= 1100 & wl <= 1700)
> nir.training1 <-
+   data.frame(X = I(shootout$calibrate.1[,indices]),
+              y = shootout$calibrate.Y[,3])
> mod1 <- plsr(y ~ X, data = nir.training1,
+              ncomp = 5, validation = "LOO")
> RMSEP(mod1, estimate = "CV")

(Intercept)    1 comps    2 comps    3 comps    4 comps    5 comps
      22.05      17.95       5.83       4.95       4.90       4.72
```

[2] http://www.idrc-chambersburg.org/shootout2002.html

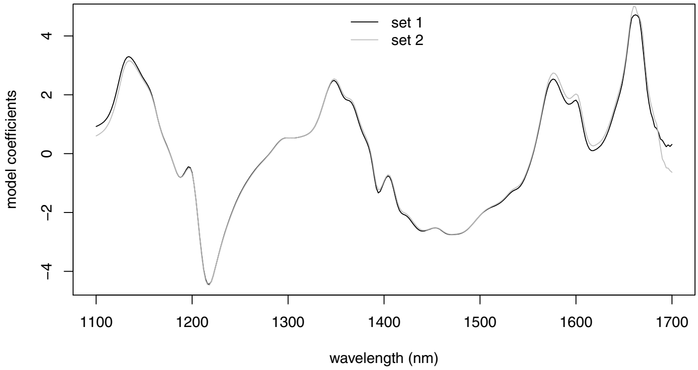

Fig. 11.7. Regression coefficients for the 3-LV PLS models of the NIR shootout data; the two instruments are indicated in black and gray, respectively.

Three components should be enough. The model based on the spectra measured at the second instrument (`mod2`) is made in the same way, and also requires three latent variables. Figure 11.7 shows the regression vectors of the two models. Some small differences are visible, in particular in the areas around 1100, 1600 and 1700 nm. As a consequence, `mod1` fares very well in predictions based on spectra measured on instrument 1:

```
> RMSEP(mod1, estimate = "test", ncomp = 3, intercept = FALSE,
+       newdata = data.frame(y = test.Y[,3],
+         X = I(test.1[,indices])))
```

```
[1] 4.974
```

Then again, predictions for data from instrument 2 are quite a bit off:

```
> RMSEP(mod1, estimate = "test", ncomp = 3, intercept = FALSE,
+       newdata = data.frame(y = test.Y[,3],
+         X = I(test.2[,indices])))
```

```
[1] 9.983
```

The average error for predictions based on data from instrument 2 is twice as large – maybe surprising, because the model coefficients do not seem to be very different. Keep in mind, though, that the intercept is usually not shown in these plots – in this case, a difference of more than twenty units is found between the intercepts of the two models.

Now suppose that five samples (numbers 10, 20, ... 50) are available for standardization purposes: these have been measured at both instruments. Let us transform the data measured on instrument 2, so that the model of instrument 1 can be applied. Now we can use an estimate of transformation matrix $\boldsymbol{F}$ to make the data from the two instruments comparable:

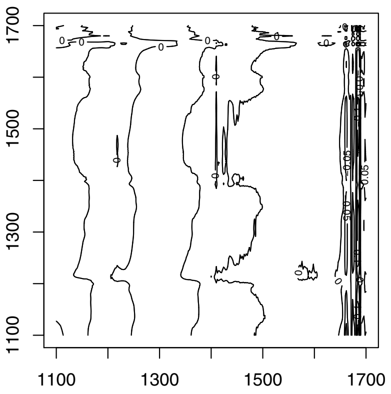

Fig. 11.8. Contour lines of the transformation matrix `F1`, mapping spectral data of one NIR instrument to another.

```
> recal.indices <- 1:5 * 10
> F1 <- ginv(shootout$calibrate.2[recal.indices, indices]) %*%
+   shootout$calibrate.1[recal.indices, indices]
> RMSEP(mod1, estimate = "test", ncomp = 3, intercept = FALSE,
+       newdata = data.frame(y = test.Y[,3],
+         X = I(test.2[,indices] %*% F1)))
```

```
[1]  4.485
```

Immediately, we are in the region where we expect prediction errors of `mod1` to be. Matrix `F1` can be visualized using the `contour` function, which sheds some light on the corrections that are made:

```
> contour(wl[indices], wl[indices], F1)
```

The result is shown in Figure 11.8. Horizontal lines correspond with columns in `F1`, containing the multiplication factors for the spectra of the second instrument $\boldsymbol{X}_2$. The largest changes can be found in the area between 1600 and 1700 nm, not surprisingly the area where the largest differences in regression coefficients are found as well.

Because spectral variations are often limited to a small range, it does not necessarily make sense to use a complete spectrum of one instrument to estimate the response at a particular wavelength at the other instrument. Wavelengths in the area of interest are much more likely to have predictive power. This realization has led to *piecewise direct standardization* (PDS, [152]), where only measurements $x_{i-k}, ..., x_i, ..., x_{i+k}$ of one spectrometer are used to predict x_i at the other spectrometer. This usually is done with PLS or PCR.

The row-wise concatenation of these regression vectors, appropriately filled with zeros to obtain the correct overall dimensions, will then lead to $\boldsymbol{F}$, in the general case a banded diagonal matrix.

An obvious disadvantage of the method is that it takes many separate multivariate regression steps. Moreover, one needs to determine the window size, and information is lost at the first and last k spectral points. Nevertheless, some good results have been obtained with this strategy [152].

Methods that directly transform the model coefficients have been described as well, but results were disappointing [155]. Apart from this, a distinct disadvantage of transforming the model coefficients is that the response variable y is needed. The DS and PDS approaches described above, on the other hand, can be applied even when no response information is available: in Equation 11.6 only the spectral data are used.

11.5 Multivariate Curve Resolution

In Multivariate Curve Resolution (MCR, sometimes indicated with Alternating Least Squares regression, ALS, or even MCR-ALS), one aims at decomposing a data matrix in such a way that the individual components are corresponding directly to chemically relevant characteristics, such as spectra and concentration profiles. In contrast to PCA, no orthogonality is imposed. This comes at a high price, however: in PCA, the orthogonality constraint ensures that only one linear combination of original variables gives the optimal approximation of the data (up to the sign ambiguity). In MCR, usually a band of feasible solutions is obtained. On the other hand, the orthogonality constraint in PCA prevents one from direct interpretation of the PCs: real, “pure” spectra will almost never be orthogonal. This direct interpretation is the goal of MCR.

Monitoring a reaction by some form of spectroscopy is a classical example: during the reaction products are formed which may, in turn, react to form other compounds. The concentrations of the starting compounds go down over time, those of the end products go up, and those of the intermediates first go up and finally go down again. Some of the components may be known to be present, others may be unexpected and their spectra unknown. Another example, often encountered, is given by data from HPLC-UV analyses: the sample is separated over a column and at certain points in time the UV-Vis spectra are measured. Again, this results in concentration profiles over time for all compounds in the sample, as well as the associated pure spectra.

An example of such a data set is `bdata`. It consists of UV measurements at 73 wavelengths, measured at 40 time points. Two data matrices are available; at this moment we will concentrate on the first. The sample is a mixture of three compounds, two of which are diazinon and parathion-ethyl, both organophosphorus pesticides [156]. A perspective plot of the data is shown in Figure 11.9.

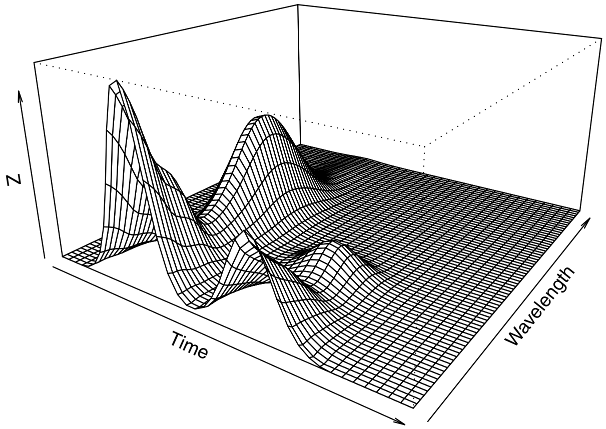

Fig. 11.9. Perspective plot of a HPLC-UV data set. At several points in time, spectra are measured from the eluate passing the detector; the goal is to estimate the pure spectra and the concentration profiles of the compounds in the mixture.

11.5.1 Theory

The basis of the method is laid in the a seminal paper by Lawton and Sylvestre [157], but the real popularity within the chemometrics community came two decades later, especially because of the work by Tauler, De Juan and Maeder (e.g., [158, 159]). In the ideal case, only the number of components and suitable initial estimates for either concentration profiles or pure spectra should have to be provided to the algorithm. The data matrix typically contains the spectra of the mixture at several time points, and can be decomposed as

$$\boldsymbol{X} = \boldsymbol{C}\boldsymbol{S}^T + \boldsymbol{E} \tag{11.7}$$

where $\boldsymbol{C}$ is the matrix containing the "pure" concentration profiles, $\boldsymbol{S}$ contains the "pure" spectra and $\boldsymbol{E}$ is an error matrix. The word "pure" is quoted since there is rotational ambiguity: $\boldsymbol{C}\boldsymbol{R}^{-1}$ and $\boldsymbol{R}\boldsymbol{S}^T$ will, for any rotation matrix $\boldsymbol{R}$, lead to the same approximation of the data:

$$\boldsymbol{X} = \boldsymbol{C}\boldsymbol{S}^T = \boldsymbol{C}\boldsymbol{R}^{-1}\boldsymbol{R}\boldsymbol{S}^T \tag{11.8}$$

Whether it is possible to identify one particular set of $\boldsymbol{C}\boldsymbol{R}^{-1}$ and $\boldsymbol{R}\boldsymbol{S}^T$ as better than others depends on the presence of additional information. This often takes the form of constraints. If C indeed corresponds to a concentration matrix, it can only contain non-negative elements. Such a non-negativity constraint is applicable for many forms of spectroscopy as well, like a number of other constraints that will be briefly treated below.

The decomposition of Equation 11.7 is usually performed by repeated application of multiple least squares regression – hence the name ALS. Given a starting estimate of, e.g., $\boldsymbol{C}$, one can calculate the matrix $\boldsymbol{S}$ that minimizes the residuals $\boldsymbol{E}$ in Equation 11.7, which in turn can be used to improve the estimate of $\boldsymbol{C}$, etcetera.

$$\hat{\boldsymbol{S}} = \boldsymbol{X}^T\boldsymbol{C}(\boldsymbol{C}^T\boldsymbol{C})^{-1} = \boldsymbol{X}^T\boldsymbol{C}^+ \tag{11.9}$$

$$\hat{\boldsymbol{C}} = \boldsymbol{X}\boldsymbol{S}(\boldsymbol{S}^T\boldsymbol{S})^{-1} = \boldsymbol{X}\left(\boldsymbol{S}^T\right)^+ \tag{11.10}$$

Equations 11.9 and 11.10 alternate until no more improvement is found or until the desired number of iterations is reached.

11.5.2 Finding Suitable Initial Estimates

The better the initial estimates, the better the quality of the final results – moreover, the MCR algorithm will usually need fewer iterations to converge. Therefore it pays to invest in a good initialization. Several strategies have been proposed. Perhaps the most simple, conceptually, is to calculate the ranks of submatrices, a procedure that has become known in chemometrics as Evolving Factor Analysis (EFA, [160]). The original proposal was to start with a small submatrix, and monitor the size of the eigenvalues upon adding more and more columns (or rows) to the data matrix; later approaches apply a moving window, e.g. Evolving Windowed Factor Analysis (EWFA, [161]). Several other methods such as the Orthogonal Projection Approach (OPA, [162]) and SIMPLISMA [163] are not based on SVD, but on dissimilarities between the spectra. These typically return the indices of the "purest" variables or objects. Here, we focus on EFA and OPA.

Evolving Factor Analysis

EFA basically keeps track the number of independent components encountered in the data matrix upon sequentially adding columns (often corresponding to time points). This is especially useful in situations where there is a development over time, such as occurs when monitoring a chemical reaction – the reaction proceeds in a number of steps and it can be interesting to see when certain intermediates are formed, and what the order of the formation is. In this context, the result of an MCR consists of the pure spectra of all species, and their concentration profiles over time.

The most basic implementation of EFA is takes a data matrix, and the desired number of components. In the forward pass, starting from a submatrix containing only three rows, the singular values of iteratively growing matrices are stored. The backward pass does the same, but now the growth starts at

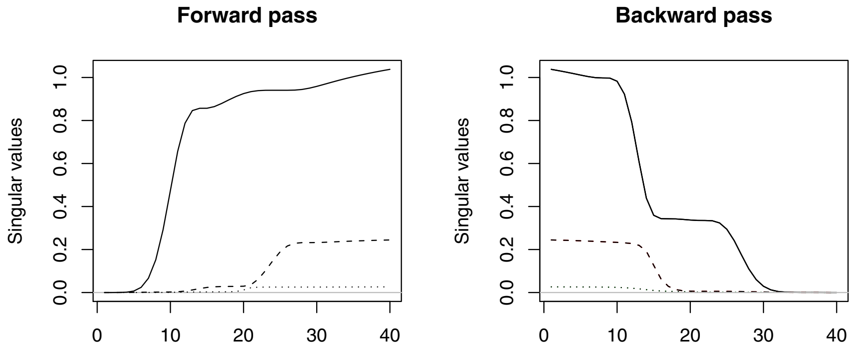

Fig. 11.10. Forward (left plot) and backward (right) traces of EFA on the HPLC-UV data.

the end and is in the backward direction. The "pure" profiles then are obtained by combining the forward and backward traces, where it is assumed that the first compound to come up is also the first one to disappear. Should one be interested in initial estimates of the other dimension, it suffices to present a transpose matrix as the first argument. In R code this would be:

```
> efa <- function(x, ncomp)
+ {
+   nx <- nrow(x)
+   Tos <- Fros <- matrix(0, nx, ncomp)
+   for (i in 3:nx)
+     Tos[i,] <- svd(scale(x[1:i,], scale = FALSE))$d[1:ncomp]
+   for (i in (nx-2):1)
+     Fros[i,] <- svd(scale(x[i:nx,], scale = FALSE))$d[1:ncomp]
+
+   Combos <- array(c(Tos, Fros[,ncomp:1]), c(nx, ncomp, 2))
+   list(forward = Tos, backward = Fros,
+        pure.comp = apply(Combos, c(1,2), min))
+ }
```

For the HPLC-UV data mentioned above, the forward and backward passes give the following result:

```
> X <- bdata$d1
> X.efa <- efa(X, 3)
> matplot(X.efa$forward, type = "l", ylab = "Singular values")
> matplot(X.efa$backward, type = "l", ylab = "Singular values")
```

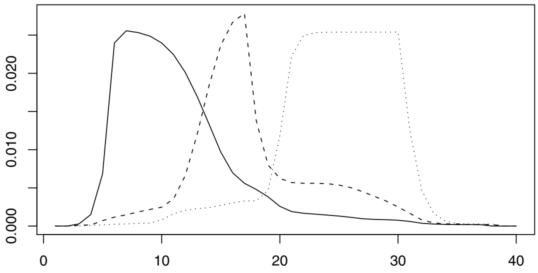

Fig. 11.11. Estimated pure traces using EFA for the Bdata data.

The result is shown in Figure 11.10. In the left plot, showing the forward pass of the EFA algorithm, the first compounds starts to come up at approximately the fifth time point. The second and third come in at about the thirteenth and nineteenth time points. In the backward trace we see similar behaviour. Estimates of the “pure” traces are obtained by simply taking the minimal values – we combine the first component in the forward trace with the last component in the backward trace, etcetera.

```
> matplot(X.efa$pure.comp, type = "l", ylab = "", col = 1)
```

The result is shown in Figure 11.11. Although the elution profiles do not yet resemble neat chromatographic peaks, they are good enough to serve as initial guesses and start the MCR iterations. In some cases, logarithms are plotted rather than the singular values themselves – this can make it easier to see when a compound starts to go up.

OPA – the Orthogonal Projection Approach

Rather than estimating concentration profiles, where the implicit assumption is that a compound that comes up first is also the first to disappear, one can also try to find wavelengths that only lead to absorbtion for one particular compound in the mixture. Methods like the Orthogonal Projection Approach (OPA, [162]) and SIMPLISMA [163] have been developed exactly for that situation. Put differently, they focus on finding time points in the chromatograms in which the spectra are most dissimilar. Several related methods are published as well – an overview is available in [164]. Here, we will treat OPA only.

The key idea is to calculate dissimilarities of all spectra with a set of reference spectra. Initially, the reference set contains only one spectrum, usually taken to be the average spectrum. In every iteration except the first, the reference set is extended with the spectrum that is most dissimilar – only in the first iteration, the reference is replaced rather than extended by the most dissimilar spectrum. As a dissimilarity measure, the determinant of the crossproduct matrix of $\boldsymbol{Y}_i$ is used:

$$d_i = \det(\boldsymbol{Y}_i^T \boldsymbol{Y}_i) \tag{11.11}$$

where $\boldsymbol{Y}_i$ is the reference set, augmented with spectrum i:

$$\boldsymbol{Y}_i = [\boldsymbol{Y}_{\text{ref}} \; \boldsymbol{y}_i] \tag{11.12}$$

Several scaling issues should be addressed; usually the spectra in the reference set are scaled to unit length [164], and that is the convention we will also use.

This is easily implemented in R:

```
> opa <- function(x, ncomp)
+ {
+   Xref <- colMeans(x)
+   Xref <- Xref / sqrt(sum(crossprod(Xref))) # scaling
+
+   selected <- rep(0, ncomp)
+   for (i in 1:ncomp) {
+     Xs <- lapply(1:nrow(x),
+                  function(ii, xx, xref) rbind(xref, xx[ii,]),
+                  x, Xref)
+     dissims <- sapply(Xs, function(xx) det(tcrossprod(xx)))
+     selected[i] <- which.max(dissims)
+     newX <- x[selected[i],]
+
+     if (i == 1) {
+       Xref <- newX / sqrt(crossprod(newX))
+     } else {
+       Xref <- rbind(Xref, newX / sqrt(sum(crossprod(newX))))
+     }
+   }
+   dimnames(Xref) <- NULL
+
+   list(pure.comp = t(Xref), selected = selected)
+ }
```

Again, the function takes a data matrix and the desired number of components as arguments. For the HPLC-UV data, this leads to

```
> X.opa <- opa(X, 3)
> matplot(X.opa$pure.comp, type = "l", col = 1,
+         ylab = "response", xlab = "wavelength number")
```

In this case, the pure spectra rather than the pure elution profiles are obtained. The result is shown in Figure 11.12. Clearly, the first and third component

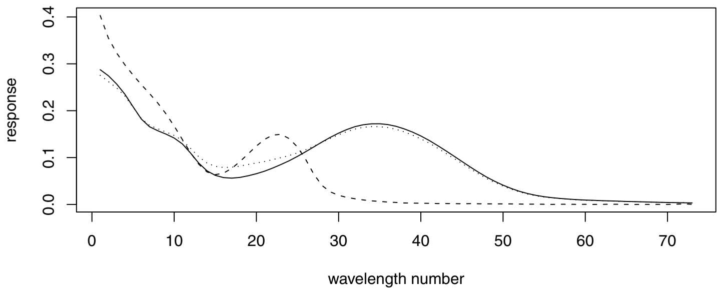

Fig. 11.12. Pure spectra obtained using OPA for the HPLC-UV data set.

are very similar. Adding another component would lead to an estimate of a pure spectrum that is very similar to what we already have – it seems two components is about optimal.

11.5.3 Applying MCR

The initial estimates of either elution profiles or pure spectra can be used to initiate the MCR iterations of Eqs. 11.9 and 11.10. The simplest possible form of MCR could look something like the following:

```
> mcr <- function(x, init, what = c("row", "col"),
+                  convergence = 1e-8, maxit = 50)
+ {
+   what <- match.arg(what)
+   if (what == "col") {
+     CX <- init
+     SX <- ginv(CX) %*% x
+   } else {
+     SX <- init
+     CX <- x %*% ginv(SX)
+   }
+
+   rms <- rep(NA, maxit + 1)
+   rms[1] <- sqrt(mean((x - CX %*% SX)^2))
+
+   for (i in 1:maxit) {
+     CX <- x %*% ginv(SX)
+     SX <- ginv(CX) %*% x
+
+     resids <- x - CX %*% SX
```

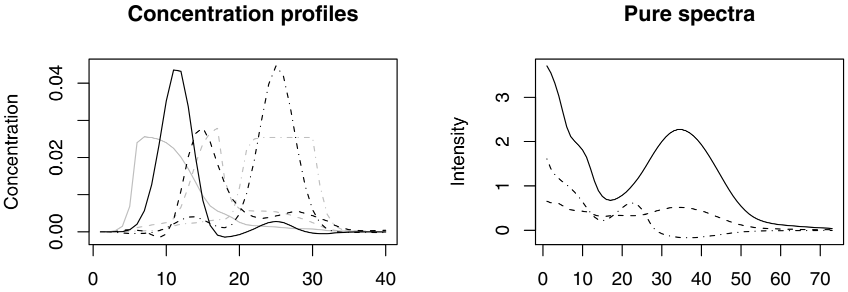

Fig. 11.13. Estimates of concentration profiles (left) and spectra of pure compounds (right) for the HPLC-UV data with MCR. The gray profiles in the left plot indicate the initialization values from EFA.

```
+       rms[i+1] <- sqrt(mean(resids^2))
+       if ((rms[i] - rms[i+1]) < convergence) break;
+     }
+
+     list(C = CX, S = SX, resids = resids, rms = rms[!is.na(rms)])
+ }
```

Depending on the nature of the initialization, the algorithm starts by estimating pure spectra (input parameter `what == "col"`) or elution profiles (`what == "row"`). For the Moore-Penrose inverse we again use function `ginv` from the **MASS** package. The RMS error for the initial estimate is calculated and the iterations are started. The algorithm stops when the improvement is too small. Alternatively, the algorithm stops when a predetermined number of iterations has been reached.

The result of applying this algorithm is visualized in Figure 11.13:

```
> X.mcr.efa <- mcr(X, X.efa$pure.comp, what = "col")
> matplot(X.mcr.efa$C, col = 1, type = "n",
+         main = "Concentration profiles",
+         ylab = "Concentration")
> matlines(X.efa$pure.comp, type = "l", lty = c(1,2,4),
+          col = "gray")
> matlines(X.mcr.efa$C, type = "l", lty = c(1,2,4), col = 1)
> matplot(t(X.mcr.efa$S), col = 1, type = "l", lty = c(1,2,4),
+         main = "Pure spectra", ylab = "Intensity")
```

The original concentration profile estimates from EFA have been indicated in gray in the left plot – clearly, they have improved dramatically. The peak shapes now are close to what one should expect from a chromatographic separations. The corresponding estimates of the pure spectra are shown on the right. The quality of the model can be assessed by looking at the RMS values:

```
> X.mcr.efa$rms
```

```
[1] 0.0134851 0.0001368 0.0001366 0.0001366
```

```
> X.mcr.opa$rms
```

```
[1] 0.0001474 0.0001367 0.0001366 0.0001366
```

Both the EFA- and OPA-based models end up with the same error value; note that the initial guess provided by OPA is already very close to the final result. Although both models achieve the same RMS error, they are not identical: this is a result of the rotational ambiguity, where there is a band of solutions with similar or identical quality.

11.5.4 Constraints

One remedy for the rotational ambiguity is to use constraints. From the set of equivalent solutions, only those are considered relevant for which certain conditions hold. In this particular case two constraints are immediately obvious: a non-negativity constraint can be applied to both concentration profiles and spectra, and in addition the concentration profiles can be thought to be unimodal – compounds show a unimodal distribution across the chromatographic column. A non-negativity constraint can be crudely implemented by inserting lines like

```
SX[SX < 0] <- 0
```

and

```
CX[CX < 0] <- 0
```

in the `mcr` function, but a better way to do this is to use non-negative least squares. This (as well as several other constraints) has been implemented in the `als` function of package **ALS**. Again, the function requires initial estimates of either C or S. Let us see what this leads to:

```
> X.als.efa <- als(CList = list(X.efa$pure.comp),
+                  PsiList = list(X), S = matrix(0, 73, 3),
+                  nonnegS = TRUE, nonnegC = TRUE,
+                  optS1st = TRUE, uniC = TRUE)
```

```
Initial RSS 1.648
Iteration (opt. S): 1, RSS: 0.5329, RD: 0.6766
Iteration (opt. C): 2, RSS: 0.000716, RD: 0.9987
...
Iteration (opt. S): 9, RSS: 0.0001455, RD: 0.0547
Iteration (opt. C): 10, RSS: 0.0001499, RD: -0.02999
Initial RSS / Final RSS = 1.648 / 0.0001499 = 10995
```

The output shows the initial RSS, which in this case – since we specify zeros as the initial estimate of `S` – equals the sum of squares in `X`. After estimating `S`, the RSS value has decreased to 0.533. The "RD" in the output signifies the improvement in the corresponding step: using the first estimate for the pure spectra to estimate concentration profiles virtually eliminates the fitting error. After ten iterations, the algorithm stops because there is no further improvement.

The non-negativity constraints for both spectra and diffusion profiles are given by the `nonnegS` and `nonnegC` arguments; `optS1st = TRUE` indicates that the first equation to be solved is Equation 11.9 – giving `S0` as an argument is therefore not necessary, although it is necessary to provide a (dummy) matrix of the correct size. The unimodality constraint is indicated with `uniC = TRUE`. The results are shown in Figure 11.14. All spectra and concentration profiles in this plot have been scaled to unit length to allow for easier comparison. Basically, the two initializations lead to the same models, although the effect of the unimodality constraint is much clearer in the EFA-based concentration profiles: the second and third peaks show artificial shoulders. This is much less the case with the OPA-based fit. On the other hand, the RSS value of the EFA-based model is slightly better: 0.00015 versys 0.00019.

As already stated, constraints are a way to bring in prior knowledge and to limit the number of possible solutions to the chemically relevant ones. Apart from the non-negativity and unimodality constraints encountered in the previous section, several others can be applied. An important example is *selectivity*: in some cases one knows that certain regions in a particular spectrum do not contain peaks from one compound. This forces the algorithm to assign any signal in that region to spectra of other components. Knowledge of mass balances in chemical reactions can lead to *closure* constraints, indicating that the sum of certain concentrations is constant.

The most stringent constraint is to impose an explicit model for one or even both of the data dimensions. One example can be found in the area of diffusion-ordered spectroscopy (DOSY), a form of NMR in which proton patterns of compounds in a mixture are separated on the basis of diffusion coefficients. Theoretical considerations lead to the assumption that the diffusion profiles follow an exponential curve [165, 166]. Indeed, so-called single-channel algorithms for interpreting DOSY data concentrate on fitting mono- or bi-exponentials to individual variables [167]. Such data may conveniently be tackled with MCR, where the diffusion profiles are fit using exponential curves. This is reported to lead to more robust results than less stringent constraints [168]. An extension of the **ALS** package, **TIMP**, allows to do this in R as well. A well-documented example of the use of **TIMP** in the realm of GC-MS data, where individual peaks are represented by generalized normal distributions, can be found in [169].

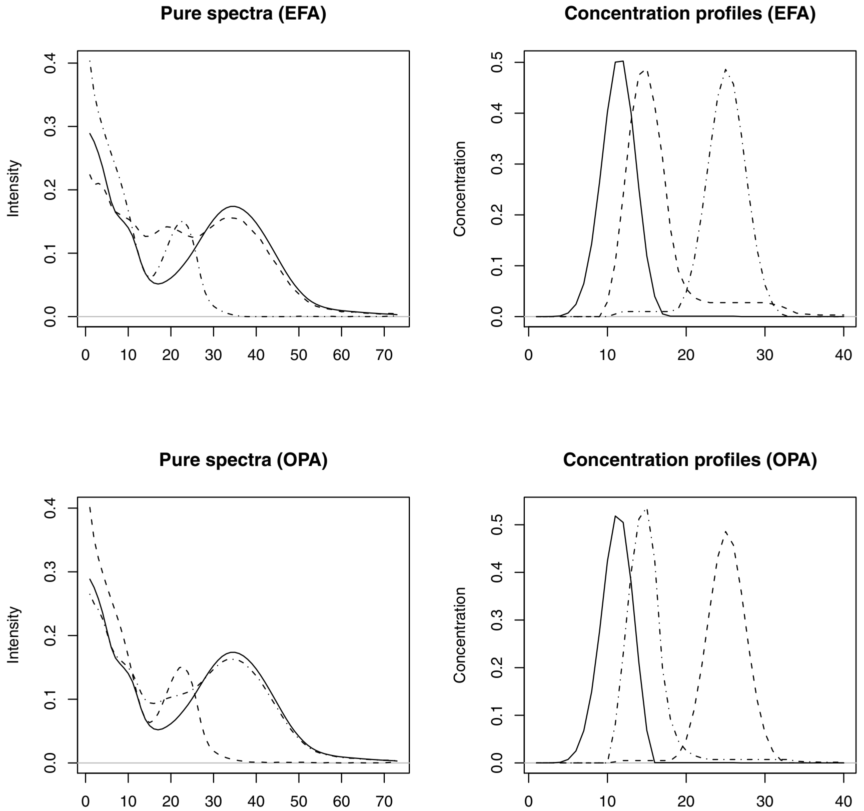

Fig. 11.14. Results for the HPLC-UV data with non-negativity and unimodality constraints: the top row shows the fit after initialization with EFA, the bottom line with OPA.

11.5.5 Combining Data Sets

In the classical application, MCR-ALS leads to estimates of pure spectra and pure concentration profiles, given a matrix of several measurements of the mixture. Extensions are possible in cases where either one mixture is studied with different measurement methods, or where several mixtures containing the same components are studied [170]. In the first case, the concentration profiles of the individual components $\boldsymbol{C}$ are the same, but it is possible to estimate the pure spectra for two or more spectroscopic techniques:

$$[\boldsymbol{X}_1|\boldsymbol{X}_2|...|\boldsymbol{X}_n] = \boldsymbol{C}\left[\boldsymbol{S}_1^T|\boldsymbol{S}_2^T|...|\boldsymbol{S}_n^T\right] \tag{11.13}$$

In the other situation one assumes that the constituents of the different samples are the same, but the concentrations are not:

$$\left[\begin{array}{c} \boldsymbol{X}_1 \\ \hline \boldsymbol{X}_2 \\ \hline \cdots \\ \hline \boldsymbol{X}_n \end{array}\right] = \left[\begin{array}{c} \boldsymbol{C}_1 \\ \hline \boldsymbol{C}_2 \\ \hline \cdots \\ \hline \boldsymbol{C}_n \end{array}\right] \boldsymbol{S}^T \tag{11.14}$$

An example of an application where common spectra and distinct concentration profiles are estimated is available from the `als` manual page. This particular form of MCR allows one to quantify compounds in the presence of unknown interferents: one of the additional data matrices is then the outcome of a measurement of a known quantity of the compound of interest. Because of the linearity of the response in most forms of spectroscopy, it is then possible to relate the concentration in the mixture to that of the standard.

The HPLC-UV data provide a way to see how this works: a second data matrix is present, containing the same three compounds. Since the variability of the chromatographic separation is much larger than the variability in spectral response, we assume common spectra, and will estimate two sets of concentration profiles. This can be done with the following code:

```
> C0 <- matrix(0, 40, 3)
> X2.als.opa <- als(CList = list(C0, C0),
+                   PsiList = list(bdata$d1, bdata$d2),
+                   S = X.opa$pure.comp,
+                   nonnegS = TRUE, nonnegC = TRUE,
+                   optS1st = FALSE, uniC = TRUE)

Initial RSS 2.135595
Iteration (opt. C): 1, RSS: 0.001065275, RD: 0.9995012
Iteration (opt. S): 2, RSS: 0.000796382, RD: 0.2524164
Iteration (opt. C): 3, RSS: 0.000729763, RD: 0.08365207
...
Iteration (opt. C): 35, RSS: 0.0004115873, RD: 0.0005339897
Initial RSS / Final RSS = 2.135595 / 0.0004115873 = 5188.681

> resids2 <- X2.als.opa$S[,1:2] - cbind(c(bdata$sp1), c(bdata$sp2))
> apply(resids2, 2, function(x) sum(x^2))

[1] 0.0026367 0.0004589
```

The residuals of the original model can be calculated in a similar way:

```
> resids <- X.als.opa$S[,1:2] - cbind(c(bdata$sp1), c(bdata$sp2))
> apply(resids, 2, function(x) sum(x^2))

[1] 0.0007835 0.0006311
```

Clearly, in this case the gain is very limited: although the second component is slightly closer to the spectrum of the pure compound, the first is actually estimated with less accuracy. A more interesting effect is the behaviour of the third compound, however. It is present in a larger quantity in the second data matrix, and as a result the estimated spectrum contains more details.

This way of combining data matrices also provides an opportunity for quantitation: when some of the components of the mixture are known, one can measure samples in which these components are present in known concentrations. This immediately enables the analyst to convert the area under the curve in the concentration profiles to true concentrations.

Part VI

Appendices

R. Wehrens, *Chemometrics with R: Multivariate Data Analysis in the Natural Sciences and Life Sciences*, Use R!, DOI 10.1007/978-3-642-17841-2,

A

R Packages Used in this Book

The table below contains an overview of R packages mentioned (but not necessarily treated in the same detail) in the book.

Table A.1. R packages used in this book

ada
ALS
AMORE
boost
boot
BootPR
bootstrap
caMassClass
ChemometricsWithR
class
cluster
DAIM
dtw
e1071
EffectiveDose
elasticnet
fastICA
fingerprint
FRB
glmnet
gpls
gtools
ipred
kohonen
lars
lasso2
leaps
lpc
lspls
MASS
mclust
meboot
msProstate
neuralnet
nnet
paltran
VPdtw
pls
plsgenomics
plspm
ppls
PROcess
ptw
randomForest
rda
relaxo
robustbase
rpart
rrcov
sfsmisc
som
spls
stats
subselect
TIMP
tree
wccsom
xcms

References

1. R Development Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2010. ISBN 3-900051-07-0.
2. W.N. Venables, D.M. Smith, and the R development Core Team. An introduction to R, December 2009. Version 2.10.1.
3. T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning.* Springer Series in Statistics. Springer, New York, 2001.
4. K. Varmuza and P. Filzmoser. *Introduction to Multivariate Statistical Analysis in Chemometrics.* Taylor & Francis - CRC Press, Boca Raton, FL, USA, 2009.
5. J. Kalivas. Two data sets of near infrared spectra. *Chemom. Intell. Lab. Syst*, 37:255–259, 1997.
6. M. Forina, C. Armanino, M. Castino, and M. Ubigli. Multivariate data analysis as a discriminating method of the origin of wines. *Vitis*, 25:189–201, 1986.
7. B.-L. Adam, Y. Qu, J.W. Davis, M.D. Ward, M.A. Clements, L.H. Cazares, O.J. Semmes, P.F. Schellhammer, Y. Yasui, Z. Feng, and G.L. Wright. Serum protein fingerprinting coupled with a pattern-matching algorithm distinguishes prostate cancer from benign prostate hyperplasia and healthy men. *Cancer Res.*, 62(13):3609–3614, July 2002.
8. Y. Qu, B.-L. Adam, Y. Yasui, M.D. Ward, L.H. Cazares, P.F. Schellhammer, Z. Feng, O.J. Semmes, and G.L. Wright. Boosted decision tree analysis of surface-enhanced laser desorption/ionization mass spectral serum profiles discriminates prostate cancer from noncancer patients. *Clin Chem*, 48(10):1835–43, October 2002.
9. T.G. Bloemberg, J. Gerretzen, H.J.P. Wouters, J. Gloerich, M. van Dael, H.J.C.T. Wessels, L.P. van den Heuvel, P.H.C. Eilers, L.M.C. Buydens, and R. Wehrens. Improved parametric time warping for proteomics. *Chemom. Intell. Lab. Systems*, 2010.
10. A. Savitsky and M.J.E. Golay. Smoothing and differentiation of data by simplified least squares procedures. *Anal. Chem.*, 36:1627–1639, 1964.
11. W.S. Cleveland. Robust locally weighted regression and smoothing scatterplots. *J. Am. Stat. Assoc.*, 74:829–836, 1979.
12. G.P. Nason. *Wavelet methods in statistics with R.* Springer, New York, 2008.
13. P. Geladi, D. MacDougall, and H. Martens. Linearization and scatter-correction for NIR reflectance spectra of meat. *Appl. Spectr.*, 39:491–500, 1985.

14. T. Næs, T. Isaksson, and B.R. Kowalski. Locally weighted regression and scatter correction for near-infrared reflectance data. *Anal. Chem.*, 62:664–673, 1990.
15. H. Sakoe and S. Chiba. Dynamic programming algorithm optimization for spoken word recognition. *IEEE Trans. Acoust., Speech, Signal Process.*, 26:43–49, 1978.
16. L.R. Rabiner, A.E. Rosenberg, and S.E. Levinson. Considerations in dynamic time warping algorithms for discrete word recognition. *IEEE Trans. Acoust., Speech, Signal Process.*, 26:575–582, 1978.
17. C.P. Wang and T.L. Isenhour. Time-warping algorithm applied to chromatographic peak matching gas chromatography / Fourier Transform infrared / Mass Spectrometry. *Anal. Chem.*, 59:649–654, 1987.
18. N.P.V. Nielsen, J.M. Carstensen, and J. Smedsgaard. Aligning of single and multiple wavelength chromatographic profiles for chemometric data analysis using correlation optimized warping. *J. Chrom. A*, 805:17–35, 1998.
19. P.H.C. Eilers. Parametric time warping. *Anal. Chem.*, 76:404–411, 2004.
20. R. de Gelder, R. Wehrens, and J.A. Hageman. A generalized expression for the similarity spectra: application to powder diffraction pattern classification. *J. Comput. Chem.*, 22(3):273–289, 2001.
21. D. Clifford, G. Stone, I. Montoliu, S. Rezzi, F.-P. Martin, P. Guy, S. Bruce, and S. Kochhar. Alignment using variable penalty dynamic time warping. *Anal. Chem.*, 81:1000–1007, 2009.
22. W. Windig, J. Phalp, and A. Payna. A noise and background reduction method for component detection in liquid chromatography/mass spectrometry. *Anal. Chem.*, 68:3602–3606, 1996.
23. T. Giorgino. Computing and visualizing dynamic time warping alignments in R: the dtw package. *J. Stat. Softw.*, 31(7), 2009.
24. J.E. Jackson. *A User's Guide to Principal Components.* Wiley, Chichester, 1991.
25. I.T. Jolliffe. *Principal Component Analysis.* Springer, New York, 1986.
26. K. Mardia, J. Kent, and J. Bibby. *Multivariate Analysis.* Academic Press, 1979.
27. W. Härdle and L. Simar. *Applied Multivariate Statistical Analysis.* Springer, Berlin, 2nd edition, 2007.
28. K.R. Gabriel. The biplot graphic display of matrices with application to principal component analysis. *Biometrika*, 58:453–467, 1971.
29. J.C. Gower and D.J. Hand. *Biplots.* Number 54 in Monographs on Statistics and Applied Probability. Chapman and Hall, London, UK, 1996.
30. K. Baumann. Uniform-length molecular descriptors for quantitative structure-property relationships (QSPR) and quantitative structure-activity relationships (QSAR): classification studies and similarity searching. *Trends Anal. Chem.*, 18(1):36–46, 1999.
31. T.F. Cox and M.A.A. Cox. *Multidimensional Scaling.* Chapman and Hall, 2001.
32. I. Borg and P.J.F. Groenen. *Modern Multidimensional Scaling.* Springer, 2nd edition, 2005.
33. J.C. Gower. Some distance properties of latent root and vector methods used in multivariate analysis. *Biometrika*, 53:325–328, 1966.
34. B.D. Ripley. *Pattern recognition and neural networks.* Cambridge University Press, 1996.

35. J.H. Friedman and J.W. Tukey. A projection pursuit algorithm for exploratory data analysis. *IEEE Trans. Comput.*, C23:881–889, 1974.
36. P.J. Huber. Projection pursuit. *The Annals of Statistics*, 13:435–475, 1985.
37. J.H. Friedman. Exploratory projection pursuit. *Journal of the American Statistical Association*, 82:249–266, 1987.
38. A. Hyvärinen, J. Karhunen, and E. Oja. *Independent Component Analysis.* Wiley, Chichester, 2001.
39. T.M. Cover and J.A. Thomas. *Elements of Information Theory.* Wiley, Chichester, 1991.
40. A. Hyvärinen and E. Oja. Independent component analysis: algorithms and applications. *Neural Networks*, 13:411–430, 2000.
41. C. Spearman. “General intelligence”, objectively determined and measured. *Am. J. Psychol.*, 15:201–293, 1904.
42. T. Kohonen. *Self-Organizing Maps.* Number 30 in Springer Series in Information Sciences. Springer, Berlin, 3 edition, 2001.
43. R. Wehrens and L.M.C. Buydens. Self- and super-organising maps in R: the kohonen package. *Journal of Statistical Software*, 21(5), 9 2007.
44. A. Ultsch. Self-organizing neural networks for visualization and classification. In O. Opitz, B. Lausen, and R. Klar, editors, *Information and Classification – Concepts, Methods and Applications*, pages 307–313. Springer Verlag, 1993.
45. W. N. Venables and B. D. Ripley. *Modern Applied Statistics with S.* Springer, New York, fourth edition, 2002. ISBN 0-387-95457-0.
46. R. Wehrens and E. Willighagen. Mapping databases of x-ray powder patterns. *R News*, 6(3):24–28, August 2006.
47. M. Eisen, P.T. Spellman, P.O. Brown, and D. Botstein. Cluster analysis and display of genome-wide expression patterns. *Proc. Natl. Acad. Sci. USA*, 95:14863–14868, December 1998.
48. L. Kaufman and P.J. Rousseeuw. *Finding Groups in Data – An Introduction to Cluster Analysis.* John Wiley & Sons, New York, 1990.
49. G.J. McLachlan and D. Peel. *Finite Mixture Models.* John Wiley & Sons, New York, 2000.
50. C. Fraley and A.E. Raftery. Model-based clustering, discriminant analysis, and density estimation. *J. Am. Stat. Assoc.*, 97:611–631, 2002.
51. A.P. Dempster, N.M. Laird, and D.B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *J. R. Statist. Soc. B*, 39(1):1–38, 1977.
52. G.J. McLachlan and T. Krishnan. *The EM Algorithm and Extensions.* John Wiley & Sons, 1997.
53. H. Akaike. A new look at the statistical model identification. *IEEE Trans. Automatic Control*, 19:716–723, 1974.
54. G. Schwarz. Estimating the dimension of a model. *Ann. Statist.*, 6:461–464, 1978.
55. C. Fraley and A.E. Raftery. Enhanced software for model-based clustering, discriminant analysis, and density estimation: MCLUST. *J. Classif.*, 20:263–286, 2003.
56. C. Fraley. Algorithms for model-based gaussian hierarchical clustering. *SIAM J. Scient. Comput.*, 20:270–281, 1998.
57. J.D. Banfield and A.E. Raftery. Model-based Gaussian and non-Gaussian clustering. *Biometrics*, 49:803–821, 1993.
58. L. Hubert. Comparing partitions. *J. Classif.*, 2:193–218, 1985.

59. W.M. Rand. Objective criteria for the evaluation of clustering methods. *J. Am. Stat. Assoc.*, 66:846–850, 1971.
60. J. Vesanto and E. Alhoniemi. Clustering of the self-organising map. *IEEE Trans. Neural Netw.*, 11:586–600, 2000.
61. E.B. Fowlkes and C.L. Mallows. A method for comparing two hierarchical clusterings. *J. Am. Stat. Assoc.*, 78:553–584, 1983. Including discussion.
62. L.A. Goodman and W.H. Kruskal. Measures of association for cross classifications. *J. Am. Statist. Assoc.*, 49:732–764, 1954.
63. M. Meila. Comparing clusterings – an information-based distance. *J. Multivar. Anal.*, 98(5):873–895, 2007.
64. G.J. McLachlan. *Discriminant Analysis and Statistical Pattern Recognition.* Wiley-Interscience, 2004.
65. M. Stone. Cross-validatory choice and assessment of statistical predictions. *J. R. Statist. Soc. B*, 36:111–147, 1974. Including discussion.
66. B. Efron and R.J. Tibshirani. *An Introduction to the Bootstrap.* Chapman and Hall, New York, 1993.
67. R.A. Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7:179–188, 1936.
68. J. Friedman. Regularized discriminant analysis. *J. Am. Stat. Assoc.*, 84:165–175, 1989.
69. S. Dudoit, J. Fridlyand, and T.P. Speed. Comparison of discrimination methods for the classification of tumors using gene expression data. *J. Am. Stat. Assoc.*, 97:77–87, 2002.
70. D. Hand and K. Yu. Idiot's Bayes – not so stupid after all? *Int. Statist. Rev.*, 69:385–398, 2001.
71. R. Tibshirani, T. Hastie, B. Narashimhan, and G. Chu. Class prediction by nearest shrunken centroids with applications to dna microarrays. *Statistical Science*, 18:104–117, 2003.
72. Y. Guo, T. Hastie, and R. Tibshirani. Regularized linear discriminant analysis and its application in microarrays. *Biostatistics*, 8(1):86–100, 2007.
73. L. Breiman, J.H. Friedman, R.A. Olshen, and C.J. Stone. *Classification and Regression Trees.* Wadsworth, 1984.
74. J.R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, 1993.
75. J.R. Quinlan. The induction of decision trees. *Mach. Learn.*, 1(1):81–106, 1986.
76. T.M. Therneau and E.J. Atkinson. An introduction to recursive partitioning using the RPART routines. Technical Report 61, Mayo Foundation, September 1997.
77. V. Vapnik. *The Nature of Statistical Learning Theory.* Springer-Verlag, 1995.
78. N. Cristianini and J. Shawe-Taylor. *An Introduction to Support Vector Machines and other kernel-based Learning Methods.* Cambridge University Press, 2000.
79. B. Schölkopf and A.J. Smola. *Learning with kernels.* MIT Press, Cambridge, MA, 2002.
80. F. Rosenblatt. *Principles of neurodynamics.* Spartan Books, Washington DC, 1962.
81. D.E. Rumelhard and J.L. McClelland, editors. *Parallel distributed processing: explorations in the microstructure of cognition. Volume 1: Foundations.* MIT Press, Cambridge MA, 1986.
82. Frauke Günther and Stefan Fritsch. neuralnet: Training of neural networks. *The R Journal*, 2(1):30–38, June 2010.

83. B.-H. Mevik and R. Wehrens. The pls package: principal component and partial least squares regression in R. *J. Stat. Soft.*, 18(2), 2007.
84. B.-H. Mevik and H.R. Cederkvist. Mean squared error of prediction (MSEP) estimates for principal component regression (PCR) and partial least squares regression (PLSR). *J. Chemom.*, 18:422–429, 2004.
85. A.S. Barros and D.N. Rutledge. Genetic algorithms applied to the selection of principal components. *Chemom. Intell. Lab. Syst.*, 40:65–81, 1998.
86. B.S. Dayal and J.F. MacGregor. Improved PLS algorithms. *J. Chemom.*, 11:73–85, 1997.
87. H. Martens and T. Næs. *Multivariate Calibration.* Wiley, Chichester, 1989.
88. S. Rännar, F. Lindgren, P. Geladi, and S. Wold. The PLS Kernel algorithm for data sets with many variables and fewer objects. Part 1: Theory and algorithm. *J. Chemom.*, 8:111, 1994.
89. S. de Jong. SIMPLS: an alternative approach to partial least squares regression. *Chemom. Intell. Lab. Syst.*, 18:251–263, 1993.
90. I.E. Frank and J.H. Friedman. A statistical view of some chemometrics regression tools. *Technometrics*, 35:109–135, 1993.
91. S. Wold, N. Kettaneh-Wold, and B. Skagerberg. Nonlinear PLS modeling. *Chemom. Intell. Lab. Syst.*, 7:53–65, 1989.
92. K. Hasegawa, T. Kimura, Y. Miyashita, and K. Funatsu. Nonlinear partial least squares modeling of phenyl alkylamines with the monoamine oxidase inhibitory activities. *J. Chem. Inf. Comput. Sci-*, 36:1025–1029, 1996.
93. K. Jorgensen, V. H. Segtnan, K. Thyholt, and T. Næs. A comparison of methods for analysing regression models with both spectral and designed variables. *J. Chemometr.*, 18:451–464, 2004.
94. B. Ding and R. Gentleman. Classification using penalized partial least squares. *J. Comput. Graph. Stat.*, 14:280–298, 2005.
95. B.D. Marx. Iteratively reweighted partial least squares estimation for generalized linear regression. *Technometrics*, 38:374–381, 1996.
96. C.J.F. ter Braak and S. Juggins. Weighted averaging partial least squares regression WAPLS: an improved method for reconstructing environmental variables from species assemblages. *Hydrobiologia*, 269:485–502, 1993.
97. M. Tenenhaus, V. Esposito Vinzi, Y.M. Chatelin, and C. Lauro. PLS path modelling. *Comput. Stat. Data Anal.*, 48:159–2005, 2005.
98. N. Krämer, A.-L. Boulesteix, and G. Tutz. Penalized partial least squares with applications to B-spline transformations and functional data. *Chemom. Intell. Lab. Syst.*, 94:60–69, 2008.
99. H. Chun and S. Keles. Sparse partial least squares for simultaneous dimension reduction and variable selection. *J. Royal Stat. Soc. – Series B*, 72:3–25, 2010.
100. A.E. Hoerl. Application of ridge analysis to regression problems. *Chemical Engineering Progress*, 58:54–59, 1962.
101. A.E. Hoerl and R.W. Kennard. Ridge regression: biased estimation for nonorthogonal problems. *Technometrics*, 8:27–51, 1970.
102. A.E. Hoerl, R.W. Kennard, and K.F. Baldwin. Ridge regression: some simulations. *Commun. Stat. – Simul. Comput.*, 4:105–123, 1975.
103. J.F. Lawless and P. Wang. A simulation study of ridge and other regression estimators. *Commun. Stat. – Theory and Methods*, 5:303–323, 1976.
104. M. Stone and R.J. Brooks. Continuum regression: cross-validated sequentially constructed prediction embracing ordinary least squares, partial least squares

and principal components regression (with discussion). *J. R. Statist. Soc.*, 52:237–269, 1990.

105. S. de Jong and H.A.L. Kiers. Principal covariates regression: Part I. Theory. *Chemom. Intell. Lab. Syst.*, 14:155–164, 1992.
106. R.W. Kennard and L. Stone. Computer aided design of experiments. *Technometrics*, 11:137–148, 1969.
107. C.D. Brown and H.T. Davis. Receiver operating characteristic curves and related decision measures: a tutorial. *Chemom. Intell. Lab. Syst.*, 80:24–38, 2006.
108. C.L. Mallows. Some comments on Cp. *Technometrics*, 15:661–675, 1973.
109. P. Craven and G. Wahba. Smoothing noisy data with spline functions. *Numer. Math.*, 31:377–403, 1979.
110. S. Smit, M.J. van Breemen, H.C.J. Hoefsloot, A.K. Smilde, J.M.F.G. Aerts, and C.G. de Koster. Assessing the statistical validity of proteomics based biomarkers. *Anal. Chim. Acta*, 592:210–217, 2007.
111. A.C. Davison and D.V. Hinkley. *Bootstrap Methods and their Applications.* Cambridge University Press, Cambridge, 1997.
112. B. Efron and R. Tibshirani. Improvements on cross-validation: the .632+ bootstrap method. *J. Am. Stat. Assoc.*, 92:548–560, 1997.
113. B. Efron. Bootstrap methods: another look at the jackknife. *Ann. Stat.*, 7:1–26, 1979.
114. L. Breiman. Bagging predictors. *Machine Learning*, 24:123–140, 1996.
115. L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.
116. Y. Freund and R.E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *J. Comput. Syst. Sci.*, 55(1):119–139, 1997.
117. V. Svetnik, A. Liaw, C. Tong, J.C. Culberson, R.P. Sheridan, and B.P. Feuston. Random forest: a classification and regression tool for compound classification and qsar modeling. *J. Chem. Inf. Comput. Sci.*, 43(6):1947–58, 2003.
118. G. Michailides, K. Johnson, and M. Culp. ada: an R package for stochastic boosting. *J. Stat. Softw.*, 17(2), 2006.
119. J.H. Friedman, T. Hastie, and R. Tibshirani. Additive logistic regression: a statistical view of boosting. *Ann. Stat.*, 28:337–374, 2000.
120. R.E. Schapire, Y. Freund, P. Bartlett, and W.S. Lee. Boosting the margin: a new explanation for the effectiveness of voting methods. *Ann. Stat.*, 26:1651–1686, 1998.
121. R. Tibshirani. Regression shrinkage and selection via the lasso. *J. Royal. Statist. Soc B*, 58:267–288, 1996.
122. R. Wehrens and W.E. van der Linden. Bootstrapping principal-component regression models. *J. Chemom.*, 11(2):157–171, 1997.
123. A.H. Land and A.G. Doig. An automatic method for solving discrete programming problems. *Econometrica*, 28:497–520, 1960.
124. G.M. Furnival and G.M. Wilson. Regression by leaps and bounds. *Technometrics*, 16:499–511, 1974.
125. B. Efron, T. Hastie, I. Johnstone, and R. Tibshirani. Least angle regression. *Annals of Statistics*, 32:407–499, 2004.
126. H. Zou and T. Hastie. Regularization and variable selection via the elastic net. *J. Royal. Stat. Soc. B*, 67:301–320, 2005.
127. S. Kirkpatrick, C.D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

128. V. Cerny. A thermodynamical approach to the travelling salesman problem: an efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45:41–51, 1985.
129. N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, and E. Teller. Equations of state calculations by fast computing machines. *J. Chem. Phys.*, 21:1087–1092, 1953.
130. V. Granville, M. Krivanek, and J.-P. Rasson. Simulated annealing: a proof of convergence. *IEEE Trans. Patt. Anal. Machine Intell.*, 16:652–656, 1994.
131. A.P. Duarte Silva. Efficient variable screening for multivariate analysis. *J. Mult. Anal.*, 76:35–62, 2001.
132. D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Kluwer Academic Publishers, Boston, MA., 1989.
133. R. Leardi. Genetic algorithms in chemometrics and chemistry: a review. *J. Chemom.*, 15:559–569, 2001.
134. J. Shao. Linear model selection by cross-validation. *J. Am. Statist. Assoc.*, 88:486–494, 2003.
135. K. Baumann, H. Albert, and M. von Korff. A systematic evaluation of the benefits and hazards of variable selection in latent variable regression. Part I. Search algorithm, theory and simulations. *J. Chemometr.*, 16:339–350, 2002.
136. K. Baumann, H. Albert, and M. von Korff. A systematic evaluation of the benefits and hazards of variable selection in latent variable regression. Part II. Practical applications. *J. Chemometr.*, 16:351–360, 2002.
137. M. Hubert. Robust calibration. In S.D. Brown, R. Tauler, and B. Walczak, editors, *Comprehensive Chemometrics – Chemical and Biochemical Data Analysis*, chapter 3.07, pages 315–343. Elsevier, 2009.
138. C. Croux and G. Haesbroeck. Principal components analysis based on robust estimators of the covariance or correlation matrix. *Biometrika*, 87:603–618, 2000.
139. P. Rousseeuw. Least median of squares regression. *J. Am. Stat. Assoc.*, 79:871–880, 1984.
140. P.J. Rousseeuw and K. van Driessen. A fast algorithm for the minimum covariance determinant estimator. *Technometrics*, 41:212–223, 1999.
141. M. Hubert, P.J. Rousseeuw, and K. Vanden Branden. ROBPCA: a new approach to robust principal component analysis. *Technometrics*, 47:64–79, 2005.
142. V. Todorov and P. Filzmoser. An object oriented framework for robust multivariate analysis. *J. Stat. Softw.*, 32(3):1–47, 2009.
143. M. Hubert and K. Vanden Branden. Robust methods for partial least squares regression. *J. Chemom.*, 17:537–549, 2003.
144. B. Liebmann, P. Filzmoser, and K. Varmuza. Robust and classical PLS regression compared. *J. Chemom.*, 24:111–120, 2009.
145. S. Wold, H. Antti, F. Lindgren, and J. Ohman. Orthogonal signal correction of near-infrared spectra. *Chemom. Intell. Lab. Syst.*, 44:175–185, 1998.
146. O. Svensson, T. Kourti, and J.F. MacGregor. A comparison of orthogonal signal correction algorithms and characteristics. *J. Chemom.*, 16:176–188, 2002.
147. J. Trygg and S. Wold. Orthogonal projections to latent structures (O-PLS). *J. Chemom.*, 16:119–128, 2002.
148. M. Barker and W. Rayens. Partial least squares for discrimination. *J. Chemom.*, 17:166–173, 2003.
149. D.V. Nguyen and D. Rocke. Tumor classification by partial least squares using microarray gene expression data. *Bioinformatics*, 18:39–50, 2002.

150. A.L. Boulesteix. PLS dimension reduction for classification with high-dimensional microarray data. *Stat. Appl. Genet. Mol. Biol.*, 3, 2004. Article 33.
151. O.E. de Noord. Multivariate calibration standardization. *Chemom. Intell. Lab. Syst.*, 25:85–97, 1994.
152. Y. Wang, D.J. Veltkamp, and B.R. Kowalski. Multivariate instrument standardization. *Anal. Chem.*, 63:2750–2756, 1994.
153. Z.Y. Wang, T. Dean, and B.R. Kowalski. Additive background correction in multivariate instrument standardization. *Anal. Chem.*, 67:2379–2385, 1995.
154. E. Bouveresse, D.L. Massart, and P. Dardenne. Modified algorithm for standardization of near-infrared spectrometric instruments. *Anal. Chem.*, 67:1381–1389, 1995.
155. Y. Wang, M.J. Lysaght, and B.R. Kowalski. Improvement of multivariate calibration through instrument standardization. *Anal. Chem.*, 64:764–771, 1992.
156. R. Tauler, S. Lacorte, and D. Barceló. Application of multivariate self-modeling curve resolution to the quantitation of trace levels of organophosphorus pesticides in natural waters from interlaboratory studies. *J. Chromatogr. A*, 730:177–183, 1996.
157. W.H. Lawton and E.A. Sylvestre. Self-modeling curve resolution. *Technometrics*, 13:617–633, 1971.
158. R. Tauler. Multivariate curve resolution applied to second-order data. *Chemom. Intell. Lab. Syst.*, 30:133–146, 1995.
159. A. de Juan, M. Maeder, M. Martinez, and R. Tauler. Combining hard- and soft-modelling techniques to solve kinetic problems. *Chemom. Intell. Lab. Syst.*, 54:49–67, 2000.
160. M. Maeder. Evolving factor analysis for the resolution of overlapping chromatographic peaks. *Anal. Chem.*, 59:527–530, 1987.
161. H.R. Keller and D.L. Massart. Peak purity control in liquid chromatography with photodiode-array detection by a fixed size moving window evolving factor analysis. *Anal. Chim. Acta*, 246:379–390, 1991.
162. F. Questa Sanchez, M.S. Khots, D.L. Massart, and J.O. de Beer. Algorithm for the assessment of peak purity in liquid chromatography with photodiode-array detection. *Anal. Chim. Acta*, 285:181–192, 1994.
163. W. Windig and J. Guilment. Interactive self-modeling mixture analysis. *Anal. Chem.*, 63:1425–1432, 1991.
164. F. Cuesta Sanchez, B. van de Bogaert, S.C. Rutan, and D.L. Massart. Multivariate peak purity approaches. *Chemom. Intell. Lab. Syst.*, 36:153–164, 1996.
165. P. Stilbs. Molecular self-diffusion coefficients in Fourier transform nuclear magnetic resonance spectrometric analysis of complex mixtures. *Anal. Chem.*, 53:2135–2137, 1981.
166. P. Stilbs. Fourier-transform pulsed-gradient spin-echo studies of molecular diffusion. *Progr. NMR Spectrosc.*, 19:1–45, 1987.
167. R. Huo, R. Wehrens, J. van Duynhoven, and L.M.C. Buydens. Assessment of techniques for DOSY NMR data processing. *Anal. Chim. Acta*, 490:231–251, 2003.
168. R. Huo, R. Wehrens, and L.M.C. Buydens. Improved DOSY NMR data processing by data enhancement and combination of multivariate curve resolution with non-linear least squares fitting. *J. Magn. Reson.*, 169:257–269, 2004.

169. K.M. Mullen, I.H.M. van Stokkum, and V.V. Mihaleva. Global analysis of multiple gas chromatography-mass spectrometry (GC/MS) data sets: a method for resolution of co-eluting components with comparison to MCR-ALS. *Chemom. Intell. Lab. Syst.*, 95:150–163, 2009.

170. G. Munoz and A. de Juan. pH- and time-dependent hemoglobin transitions: a case study for process modelling. *Anal. Chim. Acta*, 595:198–208, 2007.

Index