Multivariate Statistical Analysis using the R package **chemometrics**

Heide Garcia and Peter Filzmoser

Department of Statistics and Probability Theory
Vienna University of Technology, Austria
P.Filzmoser@tuwien.ac.at

November 5, 2011

Abstract

In multivariate data analysis we observe not only a single variable or the relation between two variables but we consider several characteristics simultaneously. For a statistical analysis of chemical data (also called chemometrics) we have to take into account the special structure of this type of data. Classic model assumptions might not be fulfilled by chemical data, for instance there will be a large number of variables and only few observations, or correlations between the variables occur. To avoid problems arising from this fact, for chemometrics classical methods have to be adapted and new ones developed.

The statistical environment R is a powerful tool for data analysis and graphical representation. It is an open source software with the possibility for many individuals to assist in improving the code and adding functions. One of those contributed function packages - **chemometrics** implemented by Kurt Varmuza and Peter Filzmoser - is designed especially for the multivariate analysis of chemical data and contains functions mostly for regression, classification and model evaluation.

The work at hand is a vignette for this package and can be understood as a manual for its functionalities. The aim of this vignette is to explain the relevant methods and to demonstrate and compare them based on practical examples.

Contents

1 Introduction

1.1 The R package chemometrics

Multivariate data analysis is the simultaneous observation of more than one characteristic. In contrast to the analysis of univariate data, in this approach not only a single variable or the relation between two variables can be investigated, but the relations between many attributes can be considered. For the statistical analysis of chemical data one has to take into account the special structure of this type of data. Very often, classic model assumptions are not fulfilled by chemical data, for instance there will be less observations than variables, or correlations between the variables occur. Those problems motivated the adaption and development of several methods for chemometrics.

The statistical environment R (R Development Core Team 2006) is a powerful tool for data analysis and graphical representation. The programming language behind it is object oriented. As part of the GNU project (Galassi et al. 2009), it is an open source software and freely available on `http://cran.r-project.org`. The fact that anyone can assist in improving the code and adding functions makes R a very flexible tool with a constantly growing number of add-on packages. An introduction to R can be found in Venables and Smith (2002).

Varmuza and Filzmoser (2009) wrote a book for multivariate data analysis in chemometrics, and contributed to the R framework with a function package for corresponding applications. The package contains about 30 functions, mostly for regression, classification and model evaluation and includes some data sets used in the R help examples.

The work at hand is a vignette for this R package **chemometrics** and can be understood as a manual for its functionalities. It is written with the help of Sweave (Leisch 2002), a reporting tool which allows for LaTeX as well as R code and output to be presented within one document. For details on R vignettes and how to extract the R code from the document see Leisch (2003). The aim of this vignette is to explain the relevant methods and to demonstrate them based on practical examples.

To get started, the package **chemometrics** and other internally required packages can be loaded simultaneously by the command

```
> library(chemometrics)
```

1.2 Overview

In chemometrics very often we have to deal with multicollinear data containing a large number of variables (and only few observations) which makes the use of some classical statistical tools impossible.

Therefore, in chapter 2 we start with a very important method to analyze this type of data. The idea of *principal component analysis* (PCA) is to transform the original variables to a smaller set of latent variables (principal components) with the aim to maximize the explained variance of the data. Since the new variables are uncorrelated we get rid of the multicollinearity this way. The usage of the respective **chemometrics** functions is demonstrated with the help of a practical data example.

In chapter 3 we explain several regression methods that are suitable for chemical data. The process of finding the optimal variables to use for the model or an adequate transformation of the data and the optimal values for the model coefficients is called *calibration* of a regression model. In order to apply ordinary least squares regression properly we need at least to apply some method for the selection of the most important variables. An algorithm for *stepwise* variable selection is implemented in **chemometrics** which - starting with the empty regression model - adds and removes variables in order to reduce a certain criterion in each step. The resulting subset of the original variables is then used for OLS regression.

A method that unlike OLS does not require uncorrelated variables or normal distribution of the residuals is *principal component regression* (PCR) where not the original variables are used for the explanation of the dependent variable but the principal components. The components are chosen in a way to maximize the prediction performance of the regression model. In contrast to PCR, *Partial least squares* (PLS) regression uses so-called PLS components similarly. Those components are calculated in order to maximize the covariance between the independent and the dependent variables. For the case of outliers in the data, there is a *robust* version of PLS implemented in the **chemometrics** package.

Last but not least there are two very similar methods which are based on a modified OLS objective function: *Ridge* and *Lasso regression*. The modification consists in adding a penalty term to the objective function which causes a shrinkage of the regression coefficients. For Ridge regression, this term depends on the squared coefficients and for Lasso regression on the absolute coefficients. At the end of this section there is a section dedicated to the *comparison* of the results gained by the above mentioned methods applied to a data example.

Chapter 4 finally takes us to the optimal usage of the *classification* methods implemented in the R package **chemometrics**. The task here is to classify new objects based on a model that was trained using data with known class membership. Two conceptually relatively simple methods are *k nearest neighbors* and *classification trees*. The former classifies a new object according to the class that appears mostly among its neighbors where the number k of neighbors to consider has to be chosen optimally. Classification trees divide the object space along an optimally chosen coordinate into two regions which are divided again and again until some optimal tree size is reached, and classify a new object according to the majority class of the region it belongs to.

Artificial neural networks are motivated by the neurons in the human brain and use functions of linear combinations of the original variables - called hidden units - to model the class membership by regression. We obtain a probability for each object to belong to a certain class and assign it to the class with the highest probability. The number of hidden units and a

shrinkage parameter used in the objective function have to be optimized when building the model. Another more complex classification method are *support vector machines*. Here, the original data is transformed to a space with higher dimension in order to make the groups linearly separable by a hyperplane which is chosen in such a way that the margin between the classes is maximized. Often the groups are not perfectly separable, and so we allow for some objects to be on the wrong side of the hyperplane but not without constraining the sum of their distances from their respective class. The optimization of a parameter for this restriction is an important task here. Again the methods are demonstrated on a data example and the results are compared in the end.

The lecture of appendix A about *cross validation* is highly recommended as this method is used throughout the vignette to obtain a large number of predictions which is important for the optimization of model parameters and the estimation of the prediction performance of a model. In appendix B we explain some functions for the transformation of compositional data which appear frequently in chemistry.

2 Principal Component Analysis

Functions discussed in this section:

```
pcaCV
nipals
pcaVarexpl
pcaDiagplot
```

2.1 The Method

Principal component analysis (PCA) is a method for the visualization of complex data by dimension reduction. Besides exploratory data analysis also prediction models can be created using PCA. The method was introduced by Pearson (1901); Hotelling (1933) made further developments.

The high number of variables in multivariate data leads to three main problems:

1. Graphical representation of the data is not possible for more than three variables.
2. High correlation between the variables makes it impossible to apply many statistical methods.
3. Many variables contain only very few information.

PCA is able to avoid all of these problems by transforming the original variables into a smaller set of latent variables which are uncorrelated. Data transformed in such a way can then be

used by other methods. The latent variables with the highest concentration of information form lower dimensional data which can be visualized graphically. Noise is separated from important information.

Principal Component Transformation

Definition 2.1. *Considering a mean-centered $n \times m$-dimensional data matrix $\mathbf{X}$, we define the* principal component transformation *as*

$$\mathbf{T} = \mathbf{X} \cdot \mathbf{P},$$

$\mathbf{P}$ *being the so-called* loadings matrix *of dimension $m \times m$ and $\mathbf{T}$ the transformed $n \times m$-dimensional* matrix of scores*, the values of the* principal components *(PCs).*

We require $\mathbf{T}$ to have maximum variance, $\mathbf{P}$ to be an orthogonal matrix and its columns $\mathbf{p}_i$ to be unit vectors. This transformation is a classical eigenvalue problem and nothing else than an orthogonal rotation of the coordinate system. The loadings are the directions of the new axes (the new variables); the scores represent the coordinates of data points with respect to the new system (the new objects).

The mean vector of $\mathbf{T}$ is $\mathbf{0}$, and if the covariance matrix of $\mathbf{X}$ is $\mathbf{S}$, the covariance of $\mathbf{T}$ is $\mathbf{PSP}^{\mathsf{T}}$. The elements of $\mathbf{P}$, p_{ij}, represent the influence of the i^{th} variable on the j^{th} component, and the squared correlation of $\mathbf{x}_i$ and $\mathbf{t}_j$ can be interpreted as the variation of $\mathbf{x}_i$ explained by $\mathbf{t}_j$.

The PCA Model

The PCs resulting from the principal component transformation are ordered descendingly by their level of information but still have the same dimension as the original data. To reach the goal of dimension reduction we use only the first a principal components, resulting in the matrices ${}_a\mathbf{T}$ $(n \times a)$ and ${}_a\mathbf{P}$ $(m \times a)$, respectively, and in a model of the form

$$\mathbf{X} = {}_a\mathbf{T} \cdot {}_a\mathbf{P}^{\mathrm{T}} + {}_a\mathbf{E}.$$

That means we separate the noise ${}_a\mathbf{E}$ $(n \times m)$ from the relevant information. The model complexity a can be reasonably chosen by observing the percentage of the data's total variance that is explained by each model belonging to a fixed number of PCs.

Finally, the fitting of the model is evaluated with regard to the representation of the original variables in the new space and regarding potential outliers that can severely distort the results.

2.2 An R Session

The data set at hand results from the chemical analysis of wines grown in the same region in Italy but derived from three different cultivars. The analysis determined the quantities of 13

constituents found in each of the three types of wines. We want to apply PCA in order to reduce the dimension of the data by eliminating the noise.

First of all, since the principal component transformation is not scale invariant we scale the data before we do the PCA, leaving apart the first variable which contains the class information and is hence not relevant for PCA.

```
>   library(gclus)                # contains the data set
>   data(wine)
>   X <- scale(wine[,2:14])       # first column: wine type
```

The optimal number of components

Before we actually calculate the PCs it is necessary to determine the optimal model complexity. The **chemometrics** function `pcaCV` uses *repeated cross validation* (see appendix A) to calculate a large number of explained variances for different model complexities.

```
>   res <- pcaCV(X)
```

For an increasing number of components the output (Figure 1, left) shows the distribution of the explained variances obtained by 50 times repeated 4-fold CV. We choose the lowest possible model complexity which still leads to a reasonable high percentage of explained variance. Here, a threshold of 80% makes us compute 5 PCs.

NIPALS Algorithm

In most chemical applications it will be necessary to compute only a few principal components. For this purpose **chemometrics** offers an implementation of the so-called *nonlinear iterative partial least-squares* algorithm (NIPALS, Algorithm 2.1), developed by H. Wold (1966).

Assuming an initial score vector $\mathbf{u}$ which can be arbitrarily chosen from the variables in $\mathbf{X}$, the corresponding loading vector $\mathbf{b}$ is calculated by $\mathbf{X}^\mathsf{T}\mathbf{u}$ and normalized to length 1. This approximation can be improved by calculating $\mathbf{X} \cdot \mathbf{b}$. Until the improvement does not exceed a small threshold ϵ, the improved new vector is used to repeat this procedure.

If convergence is reached, the first principal component, $\mathbf{u} \cdot \mathbf{b}^\mathsf{T}$, is subtracted from the currently used $\mathbf{X}$ matrix, and the resulting residual matrix $\mathbf{X}_{res}$ (that contains the remaining information) is used to find the second PC. This procedure is repeated until the desired number of PCs is reached or until the residual matrix contains very small values.

Passing the argument $a = 5$, as determined before, `nipals` works as follows:

```
>   X_nipals <- nipals(X, a=5)
```

`X` is the data matrix; `a` the number of computed principal components. `nipals` gives us back two matrices: one containing the scores for each component and one with the loadings for each component. Furthermore, the improvement of the score vector in each step is displayed. In our case, warning messages occur:

```
WARNING! Iteration stop in h= 2  without convergence!
```

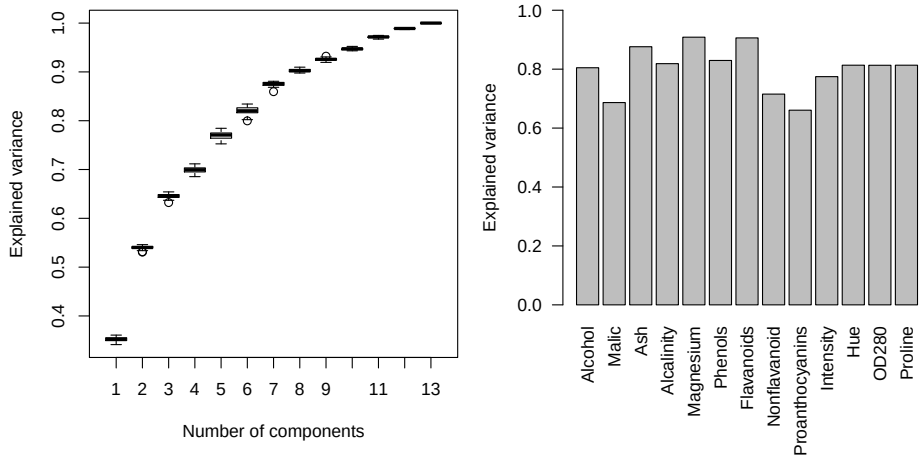

Figure 1: Left: output of `pcaCV`. Determination of the optimal number of PCs. The data for the boxplots was obtained by 4-fold CV, repeated 50 times. Right: output of `pcaVarexpl`. Control the explained variance of each variable.

Algorithm 2.1. The following scheme illustrates the algorithm used by the R function `nipals` (see Varmuza and Filzmoser 2009).

1.	$\mathbf{X}$	$\mathbf{X}$ is a mean-centered data matrix of dimension $n \times m$.
2.	$\mathbf{u} = \mathbf{x}_j$	Arbitrarily choose a variable of $\mathbf{X}$ as initial score vector.
3.	$\mathbf{b} = \mathbf{X}^\mathsf{T}\mathbf{u}/\mathbf{u}^\mathsf{T}\mathbf{u}$	Calculate the corresponding loading vector and
	$\mathbf{b} = \mathbf{b}/\|\|\mathbf{b}\|\|$	normalize it to length 1.
4.	$\mathbf{u}^* = \mathbf{X}\mathbf{b}$	Calculate an improved score vector.
5.	$\mathbf{u}_\Delta = \mathbf{u}^* - \mathbf{u}$	Calculate the summed squared differences
	$\Delta\mathbf{u} = \mathbf{u}_\Delta^\mathsf{T}\mathbf{u}_\Delta$	between the previous scores and the improved scores.
	if $\Delta\mathbf{u} > \epsilon$	Continue with step 6.
	if $\Delta\mathbf{u} < \epsilon$	Convergence is reached. Continue with step 7.
6.	$\mathbf{u} = \mathbf{u}^*$	Use the improved score vector to continue with step 3.
7.	$\mathbf{t} = \mathbf{u}^*$	Calculation of a PC is finished.
		Store score vector $\mathbf{t}$ in score matrix $\mathbf{T}$;
		store loading vector $\mathbf{p}$ in loading matrix $\mathbf{P}$.
	$\mathbf{p} = \mathbf{b}$	Stop if no further components are needed.
8.	$\mathbf{X}_{res} = \mathbf{X} - \mathbf{u}\mathbf{b}^\mathsf{T}$	Calculate the residual matrix of $\mathbf{X}$.
		Stop if the elements of $\mathbf{X}_{res}$ are very small.
		No further components are reasonable in this case.
9.	$\mathbf{X} = \mathbf{X}_{res}$	Replace $\mathbf{X}$ with $\mathbf{X}_{res}$.
		For calculation of the next PC continue with step 2.

By raising the maximum number of iterations for the approximation of scores and loadings (argument `it` which defaults to 10) to 160 we get a better result. If this measure still did not lead to convergence one would have the possibility to lower the tolerance limit for convergence (argument `tol`) which is preset to 10^{-4}. Another way would be to rethink the number of computed components.

We finally decide on the following option:

```
>   X_nipals <- nipals(X, a=5, it=160)
```

In the new orthogonal coordinate system the variables (components) are not correlated anymore, and we are able to use the transformed data with classical methods. However, this should not be started without evaluating the model's quality before.

Evaluation

The numerical accuracy of the NIPALS algorithm makes it very attractive for PC computation. However, since the calculation happens stepwise it is not the fastest algorithm. Singular value decomposition or eigenvalue decomposition are hence more recommendable if a higher number of components is required and for evaluation procedures in which PCA is carried out many

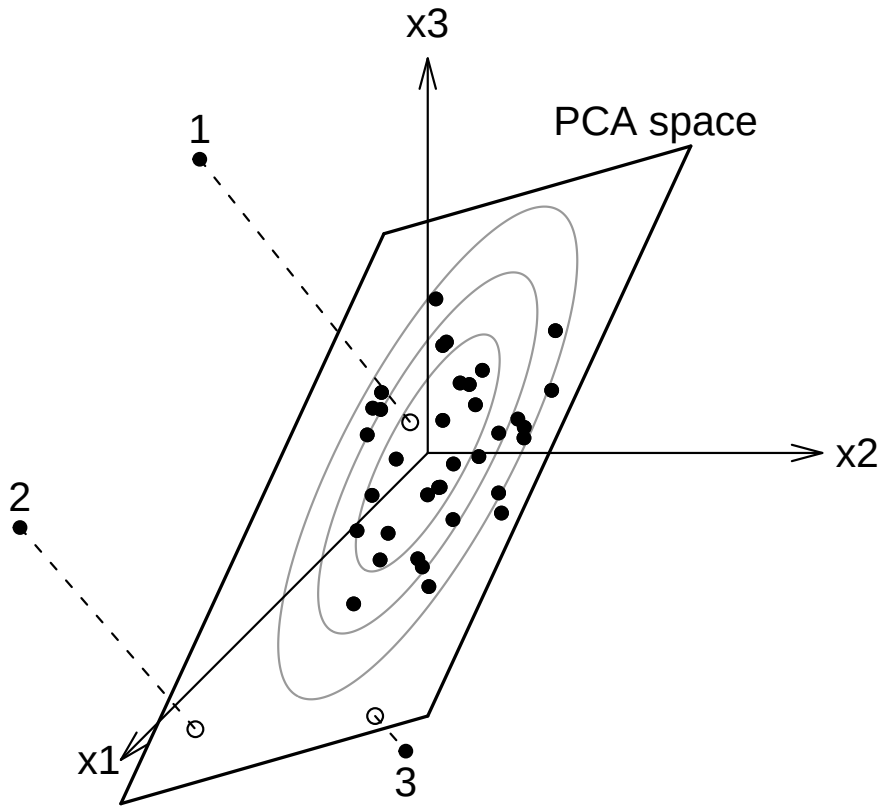

Figure 2: Types of outliers in principal component analysis.

times. The earlier described function `pcaCV` as well as the following `pcaVarexpl` work in this sense.

We want to evaluate the quality of our model taking a look at the representation of the original variables in the rotated coordinate system. Therefore, the function `pcaVarexpl` generates a plot (Figure 1, right) that shows how much of each variable's variance is explained by our model with 5 PCs.

```
>   res <- pcaVarexpl(X, a=5)
```

In some cases (for example if we consider the model with 5 or only 4 PCs) some variables may be under-represented. In order to avoid this we can add more PCs to the model.

Another important thing to do is to examine the existence of outliers in the data because they are able to severely influence the results of the PCA. For this purpose we need to calculate the *score distance* (SD) and the *orthogonal distance* (OD) of each object.

We understand the SD as the distance of an object in the PCA space from the center of the transformed data. Objects with an SD beyond a certain threshold are called *leverage points*. For the critical value of this threshold we can take the root of the 97.5% quantile of the chi-square distribution with a degrees of freedom, in our case

$$\sqrt{\chi^2_{5;0.975}} = 3.8.$$

On the other hand, the OD is calculated in the original space as the orthogonal distance of an object to the PCA subspace or, in other words, the distance between the object and its orthogonal projection on the PCA subspace. A robust cutoff value to distinguish between *orthogonal outliers* and non-outliers can be taken as

$$[\text{median}(\text{OD}^{2/3}) + \text{MAD}(\text{OD}^{2/3}) \cdot z_{0.975}]^{3/2}.$$

With the help of those two distance measures we can subdivide the data points in four classes: Regular points, orthogonal outliers as well as good and bad leverage points. Figure 2 taken from Varmuza and Filzmoser (2009) shows three dimensional data the majority of which lies in a two dimensional subspace spanned by the first two PCs. Both SD and OD are lower than the respective critical values; those objects form the regular part. If, instead, the OD exceeds its critical value (with an SD that is still in a low range), like for object 1, we have an orthogonal outlier. That means the original object lies far from the PCA space but its projection on the PCA space is within the range of the regular data. A point of that type can cause a shift of the PCA space, away from the majority of the data. A similar effect occurs in the case of bad leverage points (object 2), when both SD and OD are higher than the critical values. Points of this type lever the PCA space by pulling it in their direction and are hence able to completely distort the results. Object 3, on the other hand, is a good leverage point with large SD but low OD, causing a stabilization of the PCA space.

The function `pcaDiagplot` gives us a nice graphical possibility to detect outliers. It requires at least `X` (the original data), `X.pca` (the PCA transformed data) and `a` (the number of principal components computed). Figure 3 shows the output of `pcaDiagplot` with the classic NIPALS PCA.

```
>   X_nipals <- list(scores=X_nipals$T, loadings=X_nipals$P,
      sdev=apply(X_nipals$T, 2, sd))
```

```
>   res <- pcaDiagplot(X, X.pca=X_nipals, a=5)
```

However, for `pcaDiagplot` to work properly (i.e. to detect outliers correctly), robust PCA should be used (see Figure 4).

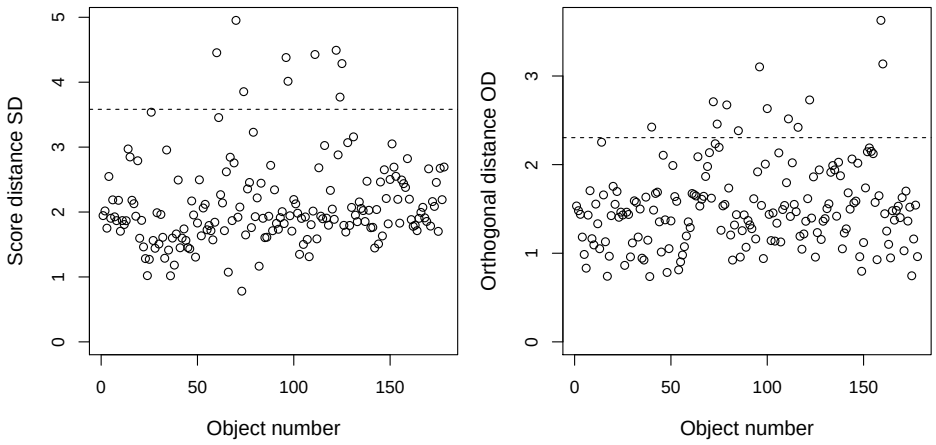

Figure 3: Output of `pcaDiagplot` with NIPALS PCA.

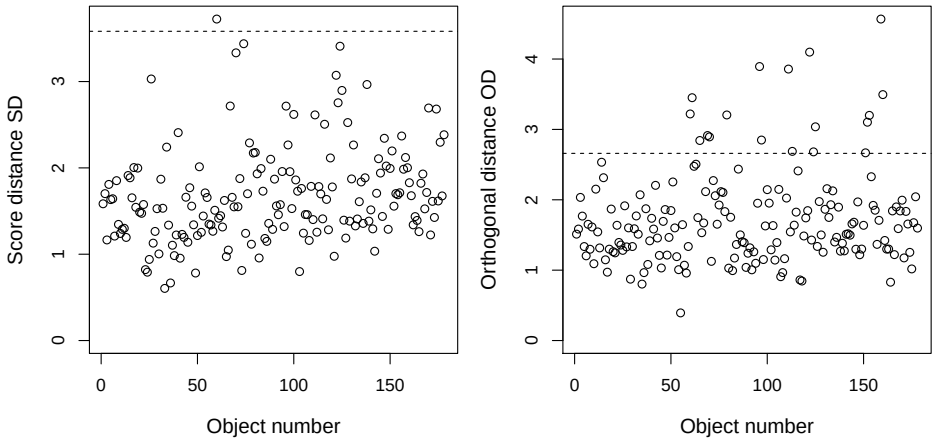

Figure 4: Output of `pcaDiagplot` with robust PCA.

```
>   X.rob <- scale(wine[,2:14], center = apply(wine[,2:14], 2, median),
      scale = apply(wine[,2:14], 2, mad))
>   library(pcaPP)                  # robust PCA based on projection pursuit
>   X.grid <- PCAgrid(X.rob,k=5,scale=mad)
>   res1 <- pcaDiagplot(X.rob,X.grid,a=5)
```

The difference is hard not to notice. Extracting the relevant object numbers we see for instance that if we do not use robust PCA `pcaDiagplot` detects only one of the two bad leverage points correctly.

```
DIAGNOSTICS WITH NIPALS:
 orthogonal outliers:  40 72 79 85 100 116 159 160
```

```
 good leverage points: 60 70 97 124 125
 bad leverage points:  74 96 111 122

DIAGNOSTICS WITH ROBUST PCA:
 orthogonal outliers:  61 65 69 70 79 96 97 111 113 122 124 125 151 152 153 159 160
 good leverage points:
 bad leverage points:  60
```

If we treat outliers like regular data we risk to determine a wrong PCA space; just to omit them is not a wise choice either. Those objects should rather be examined more closely regarding their origin and meaning.

3 Calibration

In data analysis very often we are interested in possible (linear) relations between the variables which might allow us to explain one or more dependent variables by several regressor variables. Such a *linear regression model* can be written as

$$\mathbf{y} = \mathbf{X} \cdot \mathbf{b} + \mathbf{e} \tag{1}$$

in case of a single dependent variable (*univariate* model). $\mathbf{y} = (y_1, \ldots, y_n)$ is the variable to be modelled, the $n \times (m+1)$ matrix $\mathbf{X}$ contains the *predictor variables* $\mathbf{x}_j = (x_{1j}, \ldots, x_{nj})^\mathsf{T}$, $j = 1, \ldots, m$, and a column of ones in the first place. The according *regression coefficients* are collected in the vector $\mathbf{b} = (b_0, b_1, \ldots, b_m)^\mathsf{T}$ where b_0, the coefficient belonging to the vector of ones, is called *intercept*. $\mathbf{e} = (e_1, \ldots, e_n)^\mathsf{T}$ is the *residual vector*.

If we want to predict several variables simultaneously, the *multivariate* model is

$$\mathbf{Y} = \mathbf{X} \cdot \mathbf{B} + \mathbf{E} \tag{2}$$

with p dependent variables collected in the $n \times p$ matrix $\mathbf{Y}$, the $n \times (m+1)$ predictor matrix $\mathbf{X}$ and the residual matrix $\mathbf{E}$. There is a coefficient vector with intercept for each dependent variable which results in an $(m+1) \times p$ matrix $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_p)$.

Given the basic form of a predictive model (1) or (2) we want to estimate the model parameters in such a way that the prediction performance (see below) is maximized. This estimation process is called *calibration* of the model and is carried out applying some regression method to known data. In the end, we examine the performance of the final model applying some validation method with data that was not used to find the model. This gives more realistic results.

In regression we encounter the same problems with multivariate data like listed in chapter 2: Ordinary least squares (OLS) regression can not be applied to data with more predictor variables than objects because the data matrix will be singular and the inverse can no longer be calculated. Furthermore, dependencies between the predictor variables are not allowed,

which is not a very realistic assumption in many practical cases. In the following, multiple as well as multivariate regression methods are presented alternatively to OLS.

In order to depict the various methods in this section we use a data example prepared by Liebmann et al. (2009) and enclosed in the R package **chemometrics**. For $n = 166$ alcoholic fermentation mashes of different feedstock (rye, wheat and corn) we have the matrix $\mathbf{Y}$ containing the concentration of glucose and ethanol (in g/L), respectively, as the two dependent variables. The $m = 235$ variables in $\mathbf{X}$ contain the first derivatives of near infrared spectroscopy (NIR) absorbance values at 1115-2285 nm.

```
>   data(NIR)
>   X <- NIR$xNIR
>   Y <- NIR$yGlcEtOH
>   namesX <- names(X)
>   attach(X)
```

Prediction Performance

In the following we give a short overview over measures for prediction performance assuming a univariate model (1). Using the predicted values $\hat{y}_i = \mathbf{x}_i^T \widehat{\mathbf{b}}$, where $\widehat{\mathbf{b}}$ are the estimated regression coefficients, we can write the residuals $e_i = y_i - \hat{y}_i$, $i = 1, \ldots, z$. Since we always aim to obtain a large number of predictions to get more reliable results, very often z (the number of predictions) will be larger than n.

An estimate for the spread of the error distribution is the *standard error of prediction* (SEP), the standard deviation of the residuals:

$$\text{SEP} = \sqrt{\frac{1}{z-1}\sum_{i=1}^{z}(e_i - \bar{\mathbf{e}})^2}$$

where $\bar{\mathbf{e}}$ is the arithmetric mean of the residuals (or *bias*). The fact that the SEP is measured in units of $\mathbf{y}$ is an advantage for practical applications. It can be used to compare the performance of different methods, which we will do later.

Robust measures for the standard deviation are the *interquartile range* (s_{IQR}) or the *median absolute deviation* (s_{MAD}). s_{IQR} is calculated as the difference between the empirical 3$^{\text{rd}}$ and 1$^{\text{st}}$ quartile and states the range of the middle 50% of the residuals. s_{MAD} is the median of the absolute deviations from the residuals' median.

$$\begin{aligned} s_{\text{IQR}} &= Q_3 - Q_1 \\ s_{\text{MAD}} &= \text{median}_i \, |e_i - \text{median}_j(e_j)| \end{aligned}$$

Being the mean of the squared residuals, the *mean squared error of prediction* (MSEP)

$$\text{MSEP} = \frac{1}{z}\sum_{i=1}^{z} e_i^2$$

is not measured in units of $\mathbf{y}$ and mostly used for model selection (determination of coefficients and other model parameters, variable selection, etc.).

With the relation $\text{SEP}^2 = \text{MSEP} - \bar{\mathbf{e}}^2$ we see that if the bias is close to zero, SEP^2 and MSEP are almost identical. In the case of (almost) zero bias, the square root of the MSEP

$$\text{RMSEP} = \sqrt{\frac{1}{z}\sum_{i=1}^{z} e_i^2}$$

is almost identical to the SEP.

Another important measure is the PRESS (*predicted residual error sum of squares*),

$$\text{PRESS} = \sum_{i=1}^{z} e_i^2 = z \cdot \text{MSEP}$$

the sum of the squared residuals, which is minimized for the determination of regression coefficients and often applied in CV.

In the best case (if the method we apply is fast enough), we split the data into a calibration set and a test set. The calibration set is then used for model selection by CV and with the test set we estimate the prediction performance for new data (model assessment). The test error we obtain from this procedure is the most realistic measure we can get (SEP_{test}, $\text{MSEP}_{\text{test}}$, $\text{RMSEP}_{\text{test}}$).

Since some algorithms are too slow for this costly procedure though, we do CV with the whole data set (without splitting off a test set) and obtain measures that are still acceptable (SEP_{CV}, MSEP_{CV}, RMSEP_{CV}).

Calculating "predictions" from the data that was used to develop the model, in general leads to too optimistic estimations and is not recommended (SEC, MSEC, RMSEC - the C stands for calibration).

3.1 Stepwise Regression

Functions discussed in this section:

```
stepwise
lmCV
```

Stepwise regression is a common method for *variable selection* (compare Hocking 1976). The number of predictors is reduced in order to find the most parsimonious (simplest) possible

model that still guarantees a good prediction performance. Our goal is a good fit of the data at a relatively low model complexity.

The **chemometrics** function `stepwise` does stepwise variable selection starting with the empty univariate model where the dependent variable **y** is explained only by the intercept and adding or removing in each step one variable until no more improvement can be done or until a certain number of steps is reached. The criterion used for the decision which variable to add is the *Bayesian information criterion* (Schwarz 1978):

Definition 3.1. *For a linear model with normally distributed residuals, the Bayesian information criterion (BIC) is a function of the sum of squares of the model residuals and given as*

$$\mathrm{BIC} = n \cdot \ln \frac{\mathrm{RSS}}{n} + a \cdot \ln n.$$

Since the RSS decreases with increasing a (number of predictor variables used), the BIC contains a penalty term depending on that number.

Note that the absolute BIC value has no meaning because it describes the loss of information compared to an exact model. Thus, the lower the BIC value the closer to reality is the model and the better is its performance. That is why we choose the variable that causes the model with the lowest BIC value.

For our NIR data, let us predict only the glucose concentration from the given x-variables.

```
>   y <- Y[,1]
>   NIR.Glc <- data.frame(X=X, y=y)
```

```
>   res.step <- stepwise(y~., data=NIR.Glc)
```

From the result of stepwise regression we can extract the number of steps and seconds needed for the algorithm and the number and names of the variables used for the final model:

```
>   steps    <- length(res.step$usedTime)
>   seconds  <- res.step$usedTime[steps]
>   varnbr   <- sum(finalvars <- res.step$models[steps,])
>   varnames <- namesX[as.logical(finalvars)]
```

```
steps needed:         22
seconds needed:       15
number of variables:  16
```

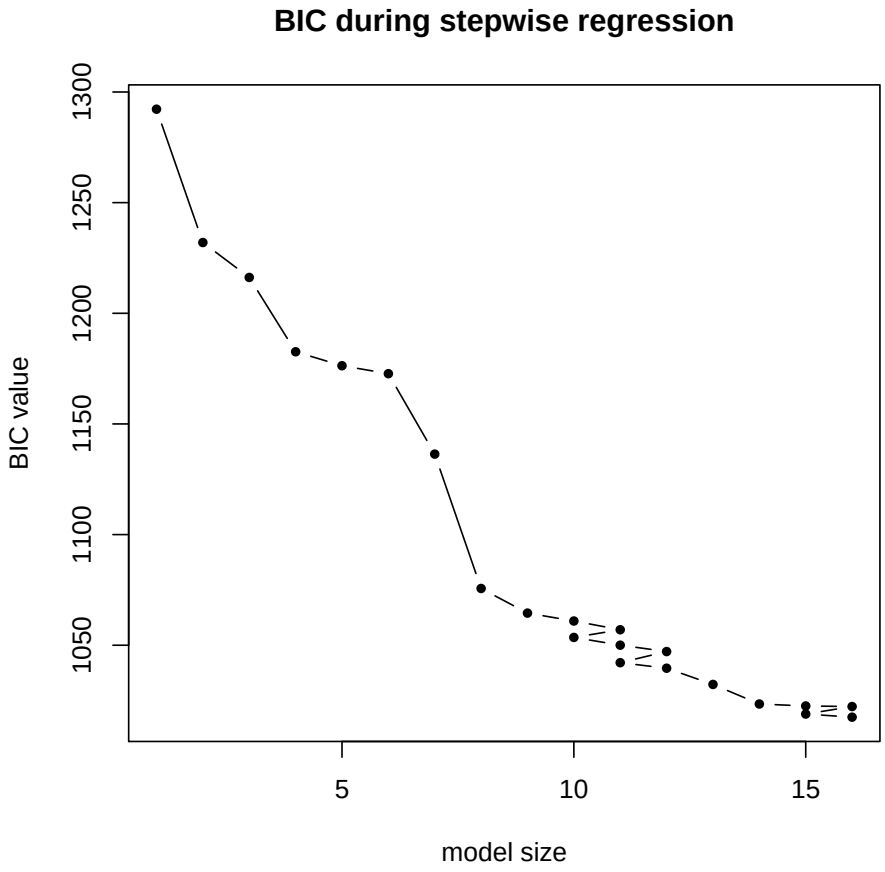

Figure 5: Change of the BIC value during stepwise regression. In steps 12, 15 and 21 a variable is dropped. After 22 steps the final model with 16 variables is reached. This result highly depends on the choice of the initial model.

In Figure 5 the change of the BIC value throughout the algorithm is tracked. We can see that the number of predictors decreases from time to time. In each of the first 11 steps one variable was added, whereas in step 12 one variable (the one that was added in step 5) is dropped again, and so on until in step 22 we reach the final model with 16 variables because no adding or removing of variables can achieve a further reduction of the BIC.

```
>   # produce Figure 5
>   modelsize <- apply(res.step$models, 1, sum)
>   plot(modelsize, res.step$bic, type="b", pch=20,
      main="BIC during stepwise regression",
      xlab="model size", ylab="BIC value")
```

Validation of the final model

We measure the prediction performance of the final model resulting from stepwise regression by *repeated cross validation* (see appendix A).

```
>   finalmodel <- as.formula(paste("y~", paste(varnames, collapse = "+"),
      sep = ""))
>   res.stepcv <- lmCV(finalmodel, data=NIR.Glc, repl=100, segments=4)
```

Here, `lmCV` carries out 100 times repeated 4-fold CV and computes the predicted values and residuals for each repetition as well as the performance measures SEP and RMSEP and their respective means. We analyze the performance of stepwise regression by graphical representation of those values (Figure 6).

```
>   par(mfrow=c(1,2))
>   plot(rep(y,100), res.stepcv$predicted, pch=".",
      main="Predictions from repeated CV",
      xlab="Measured y", ylab="Predicted y")
>   abline(coef=c(0,1))
>   points(y, apply(res.stepcv$predicted, 1, mean), pch=20)
>   plot(res.stepcv$SEP, main="Standard Error of Prediction",
      xlab="Number of repetitions", ylab="SEP")
>   abline(h=res.stepcv$SEPm)
>   abline(h=median(res.stepcv$SEP), lty=2)
```

The left hand side shows 100 small dots for each measured glucose concentration and their means by a bigger point. We see some noise in the data for small y-values which might result from roundoff errors and one object (at a glucose concentration of around 45) with suspiciously disperse predictions. However, on the right hand side we see the SEP for each CV repetition with

```
SEP mean:                4.626
SEP median:              4.617
SEP standard deviation:  0.143
```

If the cross validation did not give as stable SEP values as it is the case here one should consider other validation methods. For instance to carry out repeated double CV to obtain PLS models based on the regressor variables of the final model (compare Varmuza and Filzmoser 2009, section 4.9.1.6).

Note that the regression model found by stepwise variable selection highly depends on the choice of the starting model. In order to find the most parsimonious model it is recommended to start with the empty model though.

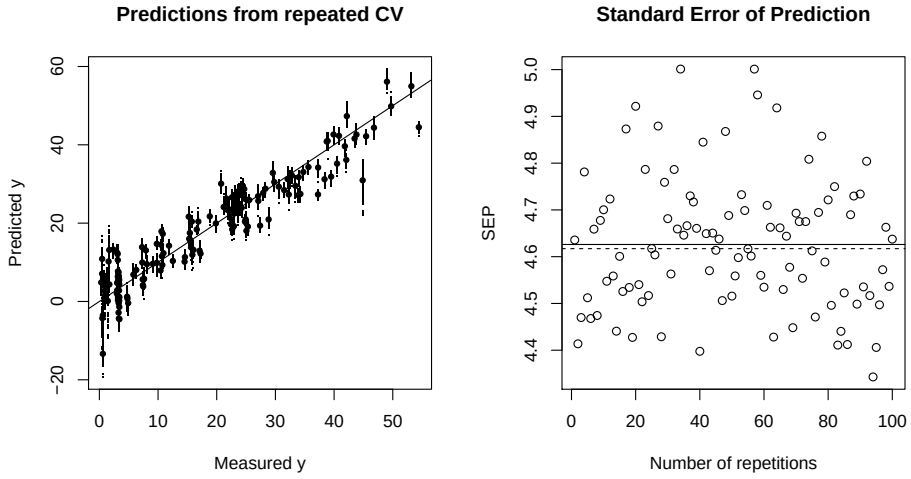

Figure 6: Left side: measured versus predicted glucose concentrations. Right side: standard error of prediction for each CV repetition. The horizontal solid and dashed line represent the mean and the median of the errors, respectively. The standard deviation of those 100 SEP values is relatively low.

3.2 Principal Component Regression

Functions discussed in this section:

```
mvr_dcv
plotcompmvr
plotpredmvr
plotresmvr
plotSEPmvr
```

Another way to face the multicollinearity problem is *principal component regression* (Jolliffe 1982). This method consists in replacing the matrix $\mathbf{X}$ by its principal components to explain the dependent variable $\mathbf{y}$. As discussed in chapter 2, these PCs are uncorrelated.

The Method

Considering an $n \times m$ matrix $\mathbf{X}$ that contains the predictor variables (but no intercept!), and additionally a dependent variable $\mathbf{y}$ with n observations, our univariate regression model is

$$\mathbf{y} = \mathbf{X} \cdot \mathbf{b} + \mathbf{e} \tag{3}$$

Here we assume that $\mathbf{X}$ is centered.

The first step of PCR is now to determine the optimal number a of principal components of $\mathbf{X}$. However, contrary to PCA, our goal here is not the maximization of the total explained

variance of $\mathbf{X}$ but to find the regression model with the *best prediction performance*. An important measure therefore is the *mean squared error of prediction* (MSEP).

Once the predictors are approximated by the chosen $n \times a$ scores matrix $\mathbf{T}$ and the $m \times a$ loadings $\mathbf{P}$ with $1 \leq a < m$, we replace $\mathbf{X}$ in (3) according to

$$\mathbf{X} \approx \mathbf{T}\mathbf{P}^{\mathsf{T}}$$

and we obtain the new regression model with uncorrelated regressors

$$\mathbf{y} = \mathbf{T}\mathbf{g} + \mathbf{e}_T \tag{4}$$

where $\mathbf{g} = \mathbf{P}^{\mathsf{T}}\mathbf{b}$ and $\mathbf{e}_T$ the new residual vector. The regression coefficients for the original model (3) can then be estimated by $\hat{\mathbf{b}}_{PCR} = \mathbf{P}\hat{\mathbf{g}}$, where $\hat{\mathbf{g}}$ are the OLS coefficients of (4).

PCR with R

The crucial step in PCR is the calculation of a, the optimum number of PCs. The **chemometrics** function `mvr_dcv` is designed especially for this purpose and carefully determines the optimal value for a by *repeated double cross validation* (see appendix A). Furthermore, predicted values, residuals and some performance measures are calculated. Take into account that the function `mvr_dcv` is made exclusively for univariate PLS models.

```
>   res.pcr <- mvr_dcv(y~., data=NIR.Glc, ncomp=40, method = "svdpc",
      repl = 100)
```

Note that the maximum value of `ncomp` is $\min(n-1, m)$ and that only the `method = "svdpc"` leads to the fitting of PCR models (other methods exist that fit PLS models, see section 3.3). This method uses singular value decomposition of the $\mathbf{X}$ matrix to calculate its principal components (see chapter 2).

For the actual selection of an optimum number of principal components one of three strategies can be chosen: `selstrat = c("diffnext", "hastie", "relchange")`. To clarify some terms we have to be aware of the results we get from RDCV: 100 repetitions yield 100 models for each number of components and thus 100 MSEP values. If we say "MSEP mean" we are talking about the mean over the CV replications for one model complexity. Analogously for "MSEP standard error". For all of those three selection strategies the maximum model complexity that can be chosen is the one that yields the lowest MSEP mean. Let us call this model *starting model*.

- `diffnext`: This strategy adds `stdfact` MSEP standard errors to each MSEP mean and observes the difference of this value and the MSEP mean of the model with one number of components less. If this change is negative a model is among the possible optima. The winner is the most complex of these candidates that still has less components than the starting model.

- `hastie` (default): The so-called *standard-error-rule* described by Hastie et al. (2001) adds `sdfact` MSEP standard errors to the lowest MSEP mean and decides in favour of the least complex model with an MSEP mean under this boundary. (See also appendix A.)
- `relchange`: Taking into consideration only the models with lower or the same complexity like the starting model, we use the largest MSEP mean to relativize the difference of each MSEP mean and the smallest MSEP mean. Among all models with a sufficiently high relative change (decrease larger than 0.001), we take the one with the highest complexity. Again among all models with lower or the same complexity the model with the lowest MSEP mean is chosen. From this one, the `sdfact`-standard-error-rule is applied.

The selection of the principal components happens in the inner CV loop after the calculation of the principal components which is done by the function `mvr` from package **pls** applying *singular value decomposition*. This function can be used later as well to calculate the loadings, scores and other details for the optimal model. That is to say that the output of `mvr_dcv` does not include those things but, in fact, predicted values, residuals and various (also robust) performance measures for each CV replication and the optimum number of PCs and the SEP for the final model. This output can be visualized by some ready-made plots.

```
> plotcompmvr(res.pcr)
```

Figure 7 shows the optimum number of PCs to use by plotting the frequencies of their occurance throughout the CV (`segments0` × `repl` values, in our case 400) against the number of

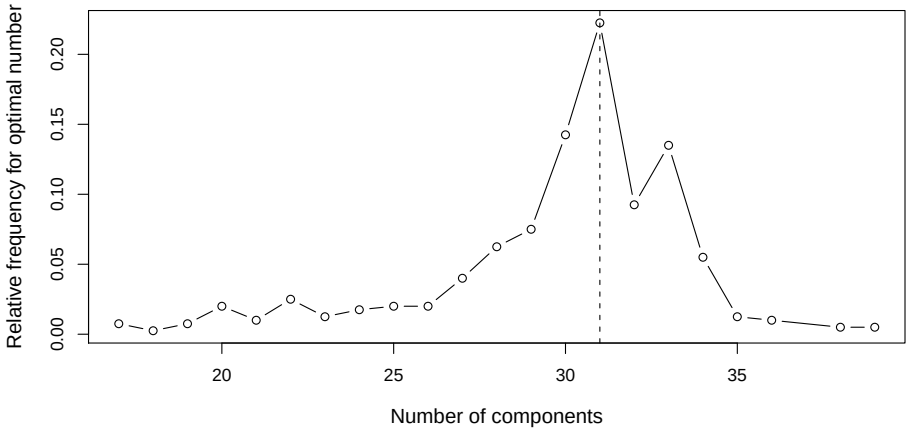

Figure 7: Output of function `plotcompmvr` for PCR. The relative frequency for the optimal number of PCR components throughout the CV. The maximum is marked by a dashed vertical line. For a better visualization, frequencies that are zero on the edges are left out.

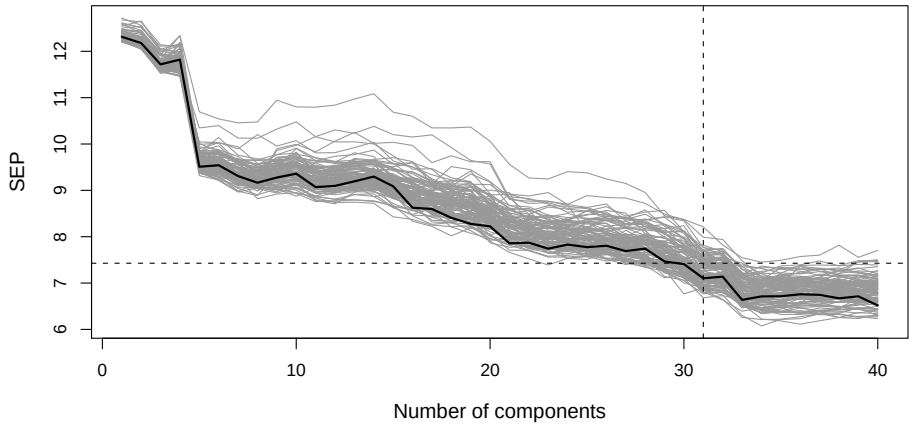

Figure 8: Output of `plotSEPmvr` for PCR. SEP values resulting from RDCV for each number of PCR components. One single CV (black line) is rather optimistic, especially for higher numbers of components.

components. A dashed vertical line at the number corresponding to the maximum frequency marks the optimum, here at 31.

The randomness of cross validation is depicted in Figure 8 where the standard errors of prediction (SEP) of the repeated double CV for each number of components are plotted. We can see 100 grey lines (one for each CV replication) and one black line resulting from one single CV carried out additionally that shows very well how optimistic it would be to use only a single CV. There is a dashed horizontal line at the SEP (7.5) of the model with the optimum number of components and the dashed vertical line indicates the optimum number of components.

```
>   optpcr <- res.pcr$afinal
>   plotSEPmvr(res.pcr, optcomp=optpcr, y=y, X=X, method="svdpc")
```

For the optimal model, two very similar plot functions, `plotpredmvr` and `plotresmvr`, present the distribution of the predicted values (Figure 9) and the residuals (Figure 10), respectively. Both demonstrate the situation for a single cross validation on the one hand and for repeated double CV on the other hand. The latter plot contains grey crosses for all CV replications and black crosses for the average of all replications. A straight line indicates where the exact prediction would be. The linear structure for lower glucose concentrations in Figure 10 derive from the data structure and probable rounding effects.

```
>   plotpredmvr(res.pcr, optcomp=optpcr, y=y, X=X, method="svdpc")
```

```
>   plotresmvr(res.pcr, optcomp=optpcr, y=y, X=X, method="svdpc")
```

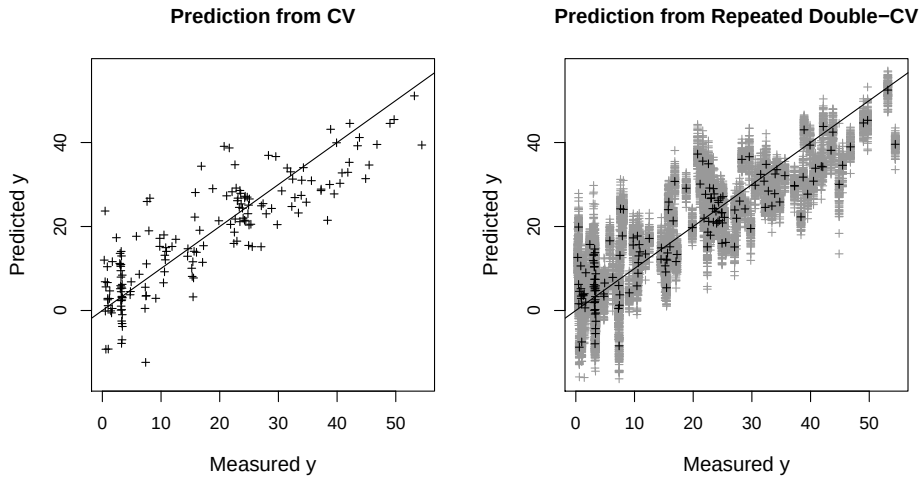

Figure 9: Output of `plotpredmvr` for PCR. Predicted versus measured glucose concentrations. The right plot shows the variation of the predictions resulting from RDCV.

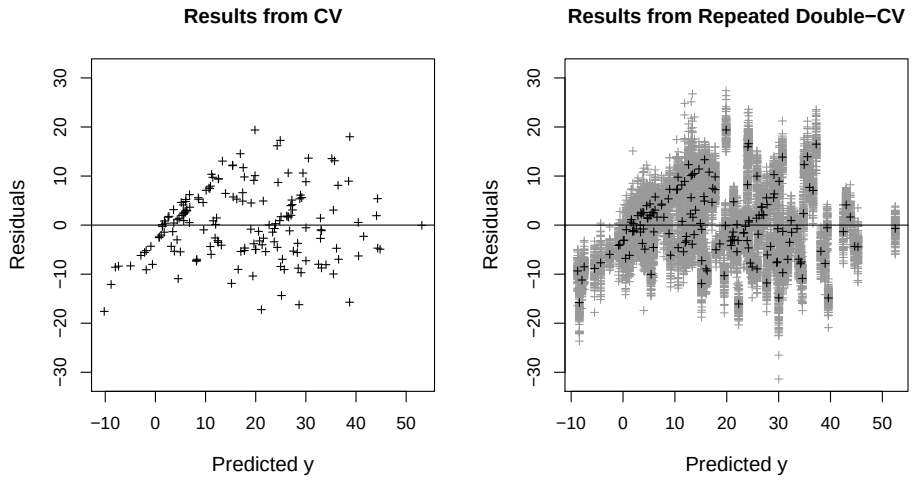

Figure 10: Output of `plotresmvr` for PCR. Residuals versus predicted values. The right plot shows again the variation of the residuals resulting from RDCV.

The SEP at the optimal number of components is indeed higher than the one we get from stepwise regression in section 3.1. On the other hand, the cross validation of PCR leads to a higher number of used principal components than the number of variables used in stepwise regression. Let us see which results *partial least squares regression* yields for the same example.

```
optimal number of PCs:   31
SEP mean:               7.433
SEP median:             7.449
SEP standard deviation: 0.373
```

3.3 Partial Least Squares Regression

Besides the functions based on the **pls** function `mvr` that were discussed in the previous section, and that fit not only PCR but also PLS models, this section shall be dedicated to the following functions:

```
pls1_nipals
pls2_nipals
pls_eigen
```

Partial least squares regression is another multivariate linear regression method, i.e. our goal is to predict several characteristics simultaneously. The stucture is basically the same as in PCR. However, while in the previous section we used the principal components of $\mathbf{X}$ to predict $\mathbf{Y}$ we shall now use the so-called *PLS components*. Instead of maximizing only the explained variance of the predictor variables, PLS components take into account the dependent variables by maximizing, for example, the covariance between the scores of $\mathbf{X}$ and $\mathbf{Y}$. That means that PLS components are relevant for the prediction of $\mathbf{Y}$, not for the modelling of $\mathbf{X}$.

Since covariance is the product of variance and correlation, PLS regression incorporates PCR (that maximizes the variance of $\mathbf{X}$) as well as OLS regression (that maximizes the correlation of $\mathbf{X}$ and $\mathbf{Y}$), and thus models the structural relation between dependent variables and predictors. Similar to PCR, PLS regression is able to process collinear data with more variables than objects by using a relatively small number of PLS components. The optimal number of components can be found once again by cross validation.

First developed by H. Wold (1966), the partial least squares regression got established in the field of chemometrics later, for example by Hoeskuldsson (1988). A review of PLS regression in chemometrics is given by S. Wold (2001), who also proposed the alternative name *projection to latent structures*.

The Method

In general, we consider a multivariate linear regression model (2) with an $n \times m$ matrix $\mathbf{X}$ of predictors and an $n \times p$ matrix $\mathbf{Y}$ of dependent variables; both matrices are mean-centered. Similar to PCR, we approximate

$$\mathbf{X} \approx \mathbf{T}\mathbf{P}^\mathsf{T}$$
$$\mathbf{Y} \approx \mathbf{U}\mathbf{Q}^\mathsf{T}$$

where $\mathbf{T}$ and $\mathbf{U}$ (both of dimension $n \times a$) are the respective score matrices that consist of linear combinations of the x- and y-variables and $\mathbf{P}$ ($m \times a$) and $\mathbf{Q}$ ($p \times a$) are the respective loadings matrices of $\mathbf{X}$ and $\mathbf{Y}$. Note that, unlike in PCR, in PLS in general not the loadings are orthogonal, but the scores. $a \leq \min(n, m)$ is the number of PLS components (chosen by means of CV).

Additionally, the x- and y-scores are related by the so-called *inner relation*

$$\mathbf{U} = \mathbf{T}\mathbf{D} + \mathbf{H},$$

a linear regression model maximizing the covariance between the x- and y-scores. The regression coefficients are stored in a diagonal matrix $\mathbf{D} = \mathrm{diag}(d_1, \ldots, d_a)$ and the residuals in the matrix $\mathbf{H}$.

The OLS estimator for $\mathbf{D}$ gives us an estimate $\widehat{\mathbf{U}} = \mathbf{T}\widehat{\mathbf{D}}$ and thus $\widehat{\mathbf{Y}} = \mathbf{T}\widehat{\mathbf{D}}\mathbf{Q}^\mathsf{T}$. Using $\widehat{\mathbf{Y}} = \mathbf{X}\widehat{\mathbf{B}}$ we obtain

$$\widehat{\mathbf{B}}_{PLS} = \mathbf{P}\widehat{\mathbf{D}}\mathbf{Q}^\mathsf{T} \tag{5}$$

The inner relation can destroy the uniqueness of the decomposition of the data matrices $\mathbf{X}$ and $\mathbf{Y}$. Normalization constraints on the score vectors $\mathbf{t}$ and $\mathbf{u}$ avoid this problem. To fulfill these restrictions we need to introduce (orthogonal) weight vectors $\mathbf{w}$ and $\mathbf{c}$ such that

$$\mathbf{t} = \mathbf{X}\mathbf{w} \text{ and } \mathbf{u} = \mathbf{Y}\mathbf{c} \text{ with}$$
$$\|\mathbf{t}\| = \|\mathbf{X}\mathbf{w}\| = 1 \text{ and } \|\mathbf{u}\| = \|\mathbf{Y}\mathbf{c}\| = 1 \tag{6}$$

Consequently, here the score vectors do not result from projection of $\mathbf{X}$ on loading vectors but on weights.

The objective function of PLS regression can then be written as

$$\max \mathrm{Cov}(\mathbf{X}\mathbf{w}, \mathbf{Y}\mathbf{c}) \tag{7}$$
$$\textit{or} \max \mathbf{t}^\mathsf{T}\mathbf{u} = (\mathbf{X}\mathbf{w})^\mathsf{T}\mathbf{Y}\mathbf{c} = \mathbf{w}^\mathsf{T}\mathbf{X}^\mathsf{T}\mathbf{Y}\mathbf{c} \tag{8}$$

with the constraints (6). The maximization problems (7) and (8) are equivalent. From (8) we see that the first weight vectors $\mathbf{w}_1$ and $\mathbf{c}_1$ are the left and right eigenvectors of the SVD of $\mathbf{X}^\mathsf{T}\mathbf{Y}$ corresponding to the largest singular value.

In the univariate case $\mathbf{y} = \mathbf{Xb} + \mathbf{e}$ (also referred to as PLS1) we approximate only $\mathbf{X} \approx \mathbf{TP}^\mathsf{T}$ and the inner relation reduces to

$$\mathbf{y} = \mathbf{Td} + \mathbf{h} \tag{9}$$

and the PLS estimator for $\mathbf{b}$ to

$$\widehat{\mathbf{b}}_{PLS} = \mathbf{P}\widehat{\mathbf{d}} \tag{10}$$

Algorithms

The algorithms provided by the R package **chemometrics** differ slightly in the results they give and also in computation time and numerical accuracy.

The *kernel algorithm* (Algorithm 3.1) proposed by Hoeskuldsson (1988) is based on the calculation of the so-called kernel matrices $\mathbf{X}^\mathsf{T}\mathbf{Y}\mathbf{Y}^\mathsf{T}\mathbf{X}$ and $\mathbf{Y}^\mathsf{T}\mathbf{X}\mathbf{X}^\mathsf{T}\mathbf{Y}$ and computes weights, scores and loadings step by step, in each step removing the extracted information from the data matrix. An efficiency brake might be the eigenvalue decomposition of the kernel matrices which is fast if the number of x- and y- variables does not get too high (the number of objects has no impact on the dimensions of the kernel matrices).

The *NIPALS algorithm* (Algorithm 3.3) already used for PCA (section 2) yields the same results as the kernel method by calculating the weights, scores and loadings in a different, numerically more accurate way. Remember the NIPALS algorithm as an iterative "improvement" of y-scores.

Hoeskuldsson (1988) describes a simpler version of the kernel algorithm. The *eigenvalue algorithm* uses the eigenvectors corresponding to the largest a eigenvalues of the kernel matrices to obtain directly the x- and y-loadings. Thus no weights have to be computed. The scores can then be calculated as usual by projecting the loadings to the x- and y-space respectively. Since no deflation of the data is made the scores are not uncorrelated which also means that the maximization problem (8) is not solved. However, the loadings are orthogonal which is an advantage for mapping.

The main difference of the *SIMPLS algorithm* (Algorithm 3.2 de Jong 1993) compared to the former ones is that the deflation is not made for the data $\mathbf{X}$ but the cross-product matrix $\mathbf{X}^\mathsf{T}\mathbf{Y}$ and that the weights are directly related to the original instead of the deflated data. An advantage is that no matrix inversion is needed for the computation of the regression coefficients.

Orthogonal projections to latent structures (O-PLS) by Trygg and Wold (2002) is a modification of NIPALS that extracts the variation from $\mathbf{X}$ that is orthogonal to $\mathbf{Y}$ and uses this filtered data for PLS:

$$\mathbf{X} - \mathbf{T}_o\mathbf{P}_o^\mathsf{T} = \mathbf{TP}^\mathsf{T} + \mathbf{E}$$

This way, not only the correlation between the x- and y-scores is maximized but also the covariance, which helps to interpret the result.

Algorithm 3.1. Kernel algorithm.

1a. calculate $\mathbf{w}_1$ as the first eigenvector of the kernel matrix $\mathbf{X}^\mathsf{T}\mathbf{Y}\mathbf{Y}^\mathsf{T}\mathbf{X}$
1b. calculate $\mathbf{c}_1$ as the first eigenvector of the kernel matrix $\mathbf{Y}^\mathsf{T}\mathbf{X}\mathbf{X}^\mathsf{T}\mathbf{Y}$
1c. normalize both vectors such that $\|\mathbf{X}\mathbf{w}_1\| = \|\mathbf{Y}\mathbf{c}_1\| = 1$
2a. project the x-data on the x-weights to calculate the x-scores $\mathbf{t}_1 = \mathbf{X}\mathbf{w}_1$
2b. project the y-data on the y-weights to calculate the y-scores $\mathbf{u}_1 = \mathbf{Y}\mathbf{c}_1$
3. calculate x-loadings by OLS regression $\mathbf{p}_1^\mathsf{T} = (\mathbf{t}_1^\mathsf{T}\mathbf{t}_1)^{-1}\mathbf{t}_1^\mathsf{T}\mathbf{X} = \mathbf{t}_1^\mathsf{T}\mathbf{X} = \mathbf{w}_1^\mathsf{T}\mathbf{X}^\mathsf{T}\mathbf{X}$
4. deflate the data matrix $\mathbf{X}_1 = \mathbf{X} - \mathbf{t}_1\mathbf{p}_1^\mathsf{T}$

Follow the steps 1-4 a times using the respective deflated data, until all a PLS components are determined. No deflation of $\mathbf{Y}$ is necessary. The regression coefficients can be estimated by

$$\widehat{\mathbf{B}} = \mathbf{W}(\mathbf{P}^\mathsf{T}\mathbf{W})^{-1}\mathbf{C}^\mathsf{T}$$

Note that the y-loadings are not needed for this purpose.
In the case of PLS1 there exists only one positive eigenvalue of $\mathbf{X}^\mathsf{T}\mathbf{y}\mathbf{y}^\mathsf{T}\mathbf{X}$.

Algorithm 3.2. SIMPLS algorithm.

0.	initialize the cross-product matrix of $\mathbf{X}$ and $\mathbf{Y}$	$\mathbf{S}_1 = \mathbf{X}^\mathsf{T}\mathbf{Y}$
1a.	calculate $\mathbf{w}_j$ as the first left singular vector of $\mathbf{S}_j = \mathbf{S}_{j-1} - \mathbf{P}_{j-1}(\mathbf{P}_{j-1}^\mathsf{T}\mathbf{P}_{j-1})^{-1}\mathbf{P}_{j-1}^\mathsf{T}\mathbf{S}_{j-1}$	
1b.	and normalize it	$\mathbf{w}_1 = \mathbf{w}_1/\|\mathbf{w}_1\|$
2a.	project the x-data on the x-weights to calculate the x-scores	$\mathbf{t}_j = \mathbf{X}\mathbf{w}_j$
2b.	and normalize them	$\mathbf{t}_1 = \mathbf{t}_1/\|\mathbf{t}_1\|$
3a.	calculate the x-loadings by OLS regression	$\mathbf{p}_j = \mathbf{X}^\mathsf{T}\mathbf{t}_j$
3b.	and store them in an accumulated x-loadings matrix	$\mathbf{P}_j = [\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_j]$

Follow the steps 1-3 a times, until all a PLS components are determined. Note that here the weights are directly related to $\mathbf{X}$ and not to a deflated matrix (step 2). The deflated cross-product matrix lies in the orthogonal complement of $\mathbf{S}_{j-1}$. The regression coefficients can be calculated by

$$\hat{\mathbf{B}} = \mathbf{W}\mathbf{T}^\mathsf{T}\mathbf{Y}$$

which explains why no y-scores or -loadings are calculated in the algorithm. If we compare the SIMPLS result to the kernel result we see that no matrix inversion is needed here.
In the PLS1 case the deflation step becomes simpler because $\mathbf{P}_j = \mathbf{p}_j$.

Algorithm 3.3. NIPALS algorithm.

0.	initialize the first y-score vector as a column of the y-data	$\mathbf{u}_1 = \mathbf{y}_k$
1a.	calculate the x-weights by OLS regression	$\mathbf{w}_1 = (\mathbf{X}^\mathsf{T}\mathbf{u}_1)/(\mathbf{u}_1^\mathsf{T}\mathbf{u}_1)$
1b.	and normalize them	$\mathbf{w}_1 = \mathbf{w}_1/\|\mathbf{w}_1\|$
2.	project the x-data on the x-weights to calculate the x-scores	$\mathbf{t}_1 = \mathbf{X}\mathbf{w}_1$
3a.	calculate the y-weights by OLS regression	$\mathbf{c}_1 = (\mathbf{Y}^\mathsf{T}\mathbf{t}_1)/(\mathbf{t}_1^\mathsf{T}\mathbf{t}_1)$
3b.	and normalize them	$\mathbf{c}_1 = \mathbf{c}_1/\|\mathbf{c}_1\|$
4a.	project the y-data on the y-weights to calculate the y-scores	$\mathbf{u}_1^* = \mathbf{Y}\mathbf{c}_1$
4b.	and determine the y-score improvement	$\Delta\mathbf{u} = \mathbf{u}_\Delta^\mathsf{T}\mathbf{u}_\Delta$
	where	$\mathbf{u}_\Delta = \mathbf{u}_1^* - \mathbf{u}_1$

If $\Delta\mathbf{u} > \epsilon$ go to step 1 using $\mathbf{u}_1^*$. If $\Delta\mathbf{u} < \epsilon$ the first component is found. Proceed with step 5a using the last $\mathbf{u}_1^*$.

5a.	find the x-loadings by OLS regression	$\mathbf{p}_1 = (\mathbf{X}^\mathsf{T}\mathbf{t}_1)/(\mathbf{t}_1^\mathsf{T}\mathbf{t}_1)$
5b.	find the inner relation parameter by OLS regression	$d_1 = (\mathbf{u}_1^\mathsf{T}\mathbf{t}_1)/(\mathbf{t}_1^\mathsf{T}\mathbf{t}_1)$
6a.	deflate the x-data	$\mathbf{X}_1 = \mathbf{X} - \mathbf{t}_1\mathbf{p}_1^\mathsf{T}$
6b.	deflate the y-data	$\mathbf{Y}_1 = \mathbf{Y} - d_1\mathbf{t}_1\mathbf{c}_1^\mathsf{T}$

Follow the steps 0-6 a times using the respective deflated data matrices, until all a PLS components are determined. The regression coefficients can be estimated by

$$\widehat{\mathbf{B}} = \mathbf{W}(\mathbf{P}^\mathsf{T}\mathbf{W})^{-1}\mathbf{C}^\mathsf{T}$$

which is the same result as for the kernel algorithm. Again the y-loadings are not needed for the regression coefficients.

For PLS1 the algorithm simplifies because no iterations are necessary to find an optimal y-score vector $\mathbf{u}^*$.

PLS with R

In section 3.2 we used the **chemometrics** function `mvr_dcv` to find the optimal number of principal components for a univariate model by repeated double CV. Several plot functions helped us to evaluate the results graphically. We applied the `method="svdpc"` for PCR. For PLS regression, we simply have to change the method to one of `simpls`, `kernelpls` or `oscorespls` in our R code and can use the same functions `mvr_dcv`, `plotcompmvr`, `plotSEPmvr`, `plotpredmvr` and `plotresmvr`.

```
>   res.pls <- mvr_dcv(y~., data=NIR.Glc, ncomp=40, method = "simpls",
      repl = 100)
```

```
> plotcompmvr(res.pls)
```

```
> optpls <- res.pls$afinal
> plotSEPmvr(res.pls, optcomp=optpls, y=y, X=X, method="simpls")
```

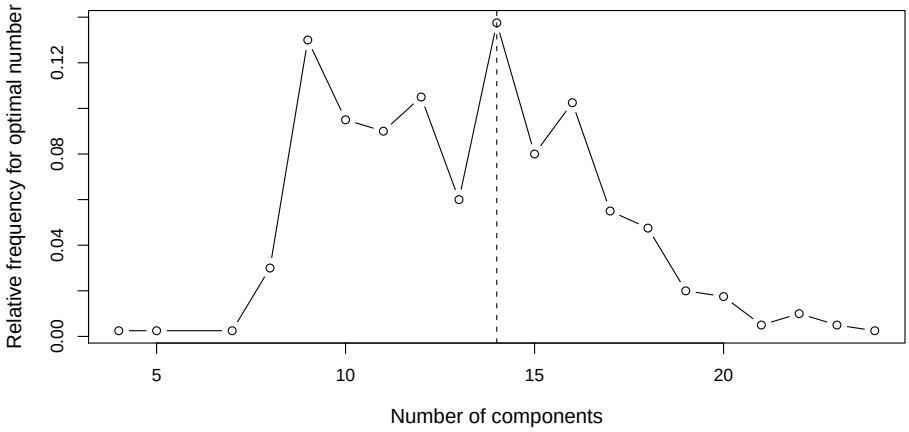

Figure 11: Output of `plotcompmvr` for PLS. Compare Figure 7. The relative frequency for the optimal number of PLS components throughout the CV. The maximum is marked by a dashed vertical line. For a better visualization, frequencies that are zero on the edges are left out.

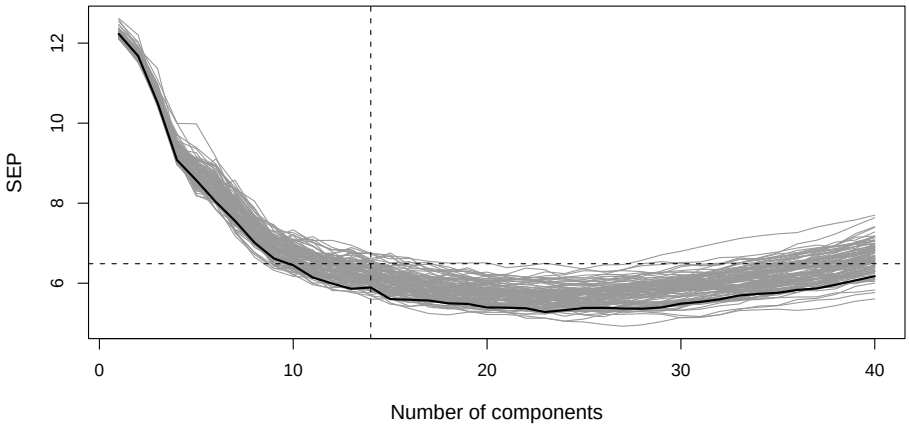

Figure 12: Output of `plotSEPmvr` for PLS. Compare Figure 8. SEP values resulting from RDCV for each number of PLS components. The black line shows how optimistic one single CV would be.

```
> plotpredmvr(res.pls, optcomp=optpls, y=y, X=X , method="simpls")
```

```
> plotresmvr(res.pls, optcomp=optpls, y=y, X=X, method="simpls")
```

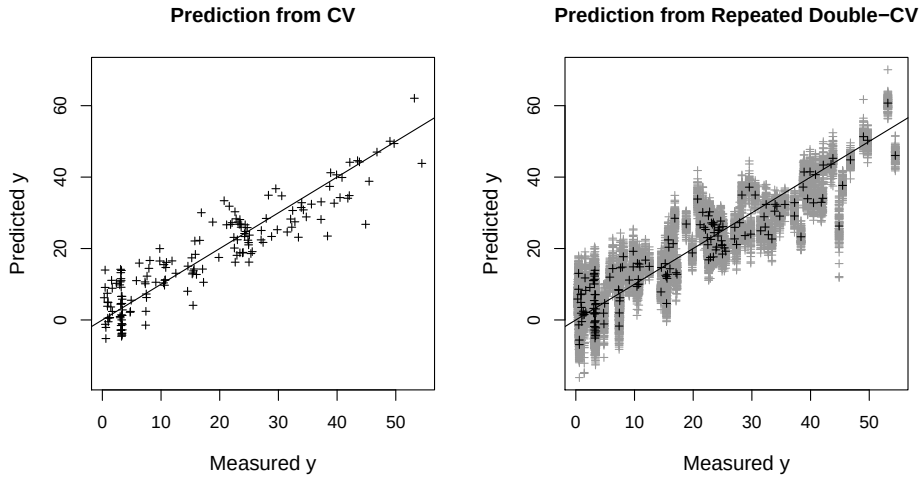

Figure 13: Output of `plotpredmvr` for PLS. Compare Figure 9. Predicted versus measured values. Grey crosses on the right: variation of predictions from RDCV.

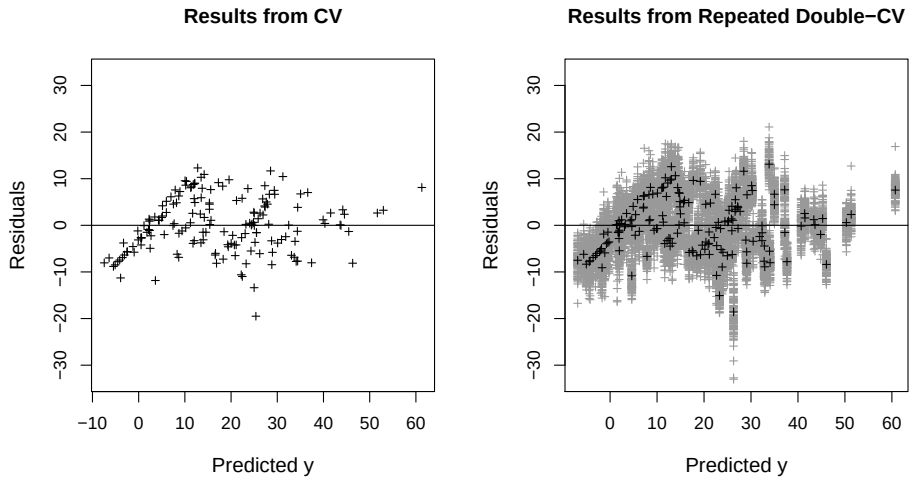

Figure 14: Output of `plotresmvr` for PLS. Compare Figure 10. Residuals versus predicted values. Grey crosses on the right: variation of predictions from RDCV.

For our example, the functions yield the following results: The optimal number of PLS components (Figure 11) is 14, which is rather low compared to the former results. Note that there is a second peak at 9 components which could be selected in order to obtain a possibly small model. We can also observe that the values for the number of used PLS components which occur during the CV are throughout lower and less spread than in the case of PCR. Not only the number of components is lower; also the standard error of prediction, 6.4, decreases compared to PCR. If we track the development of the SEP for an increasing number of components (Figure 12), we realize that it falls faster to a minimum than for PCR. The plots of predicted values (Figure 13) and residuals (Figure 14) show a very similar picture in both cases, though.

```
optimal number of PCs:   14
SEP mean:                6.489
SEP median:              6.491
SEP standard deviation:  0.408
```

To specifically calculate the loadings, scores and weights for a PLS model created by `mvr` using the algorithms SIMPLS, kernel or O-PLS the package **pls** includes useful functions (e.g. `scores`, `loadings`, `loading.weights`).

Additionally to those algorithms, in the **chemometrics** package there are also functions that use the NIPALS and the eigenvalue algorithm. The algorithms supported by `mvr_dcv` are usually sufficient for a careful analysis though.

If we have a univariate model the function `pls1_nipals` will calculate the scores and loadings for $\mathbf{X}$, weights for $\mathbf{X}$ and $\mathbf{y}$ as well as the final regression coefficients. The optimal number of components can be determined consistently by `mvr_dcv` with the `method=kernelpls`.

```
>   res.pls1nipals <- pls1_nipals(X, y, a = res.pls$afinal)
```

For a PLS2 model it is important to scale the $\mathbf{Y}$ data in order to achieve an equal treatment of all y-variables. After the optimal number of components has been determined separately, the scores, loadings and weights for $\mathbf{X}$ and $\mathbf{Y}$, the coefficients for the inner relation and the final regression coeffcients are then calculated by the function `pls2_nipals`.

```
>   Ysc <- scale(Y)
>   res.pls2nipals <- pls2_nipals(X, Ysc, a = 9)
```

Another option for PLS2 is the eigenvalue algorithm. The function `pls_eigen` returns the scores and loadings for $\mathbf{X}$ and $\mathbf{Y}$. For the number of components $a \leq \min(n, m, p)$ holds.

```
>   a <- min(dim(X)[1], dim(X)[2], dim(Y)[2])
>   res.plseigen <- pls_eigen(X, Ysc, a = a)
```

3.4 Partial Robust M-Regression

Functions discussed in this section:

```
prm
prm_cv
prm_dcv
plotprm
plotcompprm
plotSEPprm
plotpredprm
plotresprm
```

The methods PCR and PLS are not robust to outliers in the data. In order to obtain a better prediction performance we should apply a regression method that is non-sensitive to deviations from the normal model. However, robust regression methods like M-regression or better robust M-regression will fail due to the multicollinearity which occurs in our data.

Contrary to methods that try to robustify PLS (Wakeling and Macfie 1992, Cummins and Andrews 1995, Hubert and Vanden Branden 2003), Serneels et al. (2005) proposed a powerful partial version of robust M-regression for a univariate model (1).

Based on robust M-regression on the one hand and PLS regression on the other hand, partial robust M-regression combines all advantages of those two methods: it is robust against vertical outliers as well as outliers in the direction of the regressor variables and it is able to deal with more regressor variables than observations. Additionally, the used algorithm converges rather fast and exhibits a very good performance compared to other robust PLS approaches.

The Method

The idea of M-regression by Huber (1981) is to minimize not the RSS but a *function* of the sum of squares of the *standardized* residuals using a robust estimator for the standard deviation of the residuals. So, instead of the OLS estimator

$$\widehat{\mathbf{b}}_{OLS} = \arg\min_{\mathbf{b}} \sum_{i=1}^{n} (y_i - \mathbf{x}_i\mathbf{b})^2 \tag{11}$$

with the i^{th} observation $\mathbf{x}_i = (x_{i1}, \ldots, x_{im})$ we obtain the M-estimator

$$\widehat{\mathbf{b}}_{M} = \arg\min_{\mathbf{b}} \sum_{i=1}^{n} \rho(y_i - \mathbf{x}_i\mathbf{b}) \tag{12}$$

The function ρ is symmetric around zero and non-decreasing; i.e. there is a strong penalty for large residuals. If we denote the residuals in the objective function of M-regression by

$r_i = y_i - \mathbf{x}_i\mathbf{b}$, (12) can be rewritten with weights for the residuals:

$$w_i^r = \frac{\rho(r_i)}{r_i^2}$$
$$\widehat{\mathbf{b}}_M = \arg\min_{\mathbf{b}} \sum_{i=1}^{n} w_i^r (y_i - \mathbf{x}_i\mathbf{b})^2$$

The main criticism of M-regression is that it is only robust against vertical outliers. Outliers in the direction of the regressor variables can severely influence the estimation. By adding weights w_i^x for the x-variables to the estimator, Serneels et al. (2005) obtained a robust M-estimator.

$$\widehat{\mathbf{b}}_{RM} = \arg\min_{\mathbf{b}} \sum_{i=1}^{n} w_i (y_i - \mathbf{x}_i\mathbf{b})^2 \tag{13}$$
$$\text{with } w_i = w_i^r \cdot w_i^x. \tag{14}$$

This is equivalent to OLS on the data $\mathbf{X}$ and $\mathbf{y}$ multiplied with $\sqrt{w_i}$ rowwise.

The multicollinear data forces us to apply the concept of partial regression to robust M-regression. Like in PLS (equations (9) and (10)), the objective is to maximize the covariance between the dependent variable and the scores. The difference here is that we modify the covariance function to obtain loadings and scores according to a robust M-estimator.

Using the weighted covariance

$$\mathrm{Cov}_w(\mathbf{t}, \mathbf{y}) = \frac{1}{n} \sum_{i=1}^{n} (w_i t_i y_i)$$

we determine the loading vectors $\mathbf{p}_1, \ldots, \mathbf{p}_a$ in a sequential way by

$$\mathbf{p}_k = \arg\max_{\mathbf{p}} \mathrm{Cov}_w(\mathbf{X}\mathbf{p}, \mathbf{y})$$
$$\text{s.t. } \|\mathbf{p}\| = 1$$
$$\text{and } \mathrm{Cov}_w(\mathbf{p}_j, \mathbf{p}) = 0 \text{ for all previously calculated } \mathbf{p}_j.$$

The loadings are $\mathbf{t} = \mathbf{X}\mathbf{p}$.

In practice, this is done by an iterative algorithm (see algorithm 3.4), starting with appropriate starting weights and estimating PLS models until the new regression coefficient vector $\widehat{\mathbf{d}}$ converges. As usual, the optimal number of PLS components is selected by CV.

Algorithm 3.4. Modified IRPLS algorithm.

The algorithm used by Serneels et al. (2005) to accomplish PRM-regression is a slight modification of the *Iterative Reweighted Partial Least Squares* (IRPLS) algorithm proposed by Cummins and Andrews (1995). The modification constists in using robust starting values and making the weights depend not only on the residuals but also on the scores.

Equipped with the "Fair" weight function $f(z,c) = \frac{1}{(1+|\frac{z}{c}|)^2}$ where c is a tuning constant, the L_1-median med_{L1} and the mean absolute deviation of a residual vector $\mathbf{r} = (r_1, \ldots, r_n)$, $\hat{\sigma} = \text{MAD}(\mathbf{r}) = \text{median}_i\, |r_i - \text{median}_j\, r_j|$, we start with the

0.	initial residuals	$r_i = y_i - \text{median}_j\, y_j$
	and initial weights	$w_i = w_i^r \cdot w_i^x$
	where	$w_i^r = f\left(\frac{r_i}{\hat{\sigma}}, c\right)$
	and	$w_i^x = f\left(\frac{\|\mathbf{x}_i - \text{med}_{\text{L1}}(\mathbf{X})\|}{\text{median}_i \|\mathbf{x}_i - \text{med}_{\text{L1}}(\mathbf{X})\|}, c\right)$
1.	transform the x- and	$\tilde{\mathbf{x}}_i = \sqrt{w_i}\mathbf{x}_i$
	y-data rowwise	$\tilde{y}_i = \sqrt{w_i} y_i$
2.	SIMPLS on $\tilde{\mathbf{X}}$ and $\tilde{\mathbf{y}}$ yields the	
	inner relation coefficients and loadings	$\widehat{\mathbf{d}}$ and $\mathbf{P}$

If the relative difference in norm between the old and the new $\widehat{\mathbf{d}}$ is larger than some small threshold, e.g. 10^{-3}, the regression coefficients can be computed as $\widehat{\mathbf{b}}_{PRM} = \mathbf{P}\widehat{\mathbf{d}}$. Otherwise go to step 3.

3.	correct resulting scores rowwise	$\mathbf{t}_i = \mathbf{t}_i / \sqrt{w_i}$
	calculate the new residuals	$r_i = y_i - \mathbf{t}_i \widehat{\mathbf{d}}$
	and the new weights with	$w_i^x = f\left(\frac{\|\mathbf{t}_i - \text{med}_{\text{L1}}(\mathbf{T})\|}{\text{median}_i \|\mathbf{t}_i - \text{med}_{\text{L1}}(\mathbf{T})\|}, c\right)$
	and to to step 1.	

PRM with R

The most important thing we have to keep in mind is to use the original, unscaled data for partial robust M-regression with the package **chemometrics**. The scaling is done robustly within the functions which allow us to choose between the $L1$-median and the coordinatewise median for mean-centering the data. The latter is the faster calculation but the $L1$-median yields *orthogonally equivariant* estimators (Serneels et al. 2005). That means, if $\mathbf{X}$ and $\mathbf{y}$ are transformed with an orthogonal matrix $\mathbf{\Gamma}$ and a non-zero scalar c, respectively, the following property holds:

$$\widehat{\mathbf{b}}(\mathbf{X}\mathbf{\Gamma}, c\mathbf{y}) = \mathbf{\Gamma}^{\mathsf{T}}\widehat{\mathbf{b}}(\mathbf{X}, \mathbf{y})c$$

The optimal number of components can be determined by repeated double CV using the function `prm_dcv`. However, since RDCV is computationally rather expensive in this case, we first start with the function `prm_cv` which accomplishes a single CV:

```
>   res.prmcv <- prm_cv(X, y, a = 40, opt = "median")
```

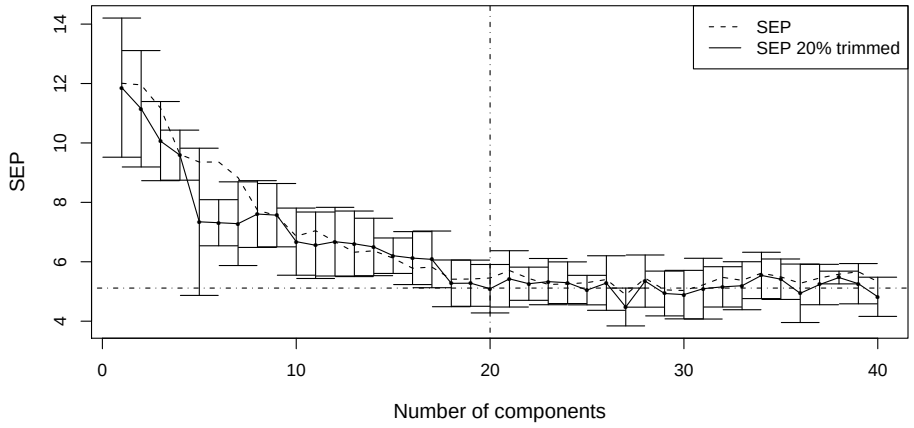

Figure 15: Output of function `prm_cv`. PRM-regression models with 1 to 40 components. Dashed line: mean of SEP values from CV. Solid part: mean and standard deviation of 20% trimmed SEP values from CV. Vertical and horizontal line correspond to the optimal number of components (after standard-error-rule) and the according 20% trimmed SEP mean, respectively.

As a basis for decision-making the function does not use the SEP but a trimmed version of the SEP (20% by default). That means that for the calculation of the SEP the 20% highest absolute residuals are discarded. The difference can be seen in Figure 15 that is produced by `prm_cv`. For each number of components the mean of the trimmed SEP values is plotted and their standard deviation is added. The optimal number of components is the lowest number that yields a trimmed SEP under the dashed horizontal line which is 2 standard errors above the minimum SEP. In our case it is the model with 20 components have a 20% trimmed SEP of 4.95.

```
optimal number of PCs: 20
                         classic   20% trimmed
SEP mean:                5.454      5.094
SEP median:              5.488      5.070
SEP standard deviation:  0.627      1.289
```

The effect of PRM on the prediction can be observed if we plot the predicted versus the measured **y** and the residuals against the predicted values:

```
>   plotprm(res.prmcv, y)
```

If we compare Figure 16 with the Figures 13 and 14 we do not see big differences, and comparing the 20% trimmed SEP values (see section 3.7, it becomes clear that there are no large outliers in our data. In absence of outliers, the robust method usually not better than the

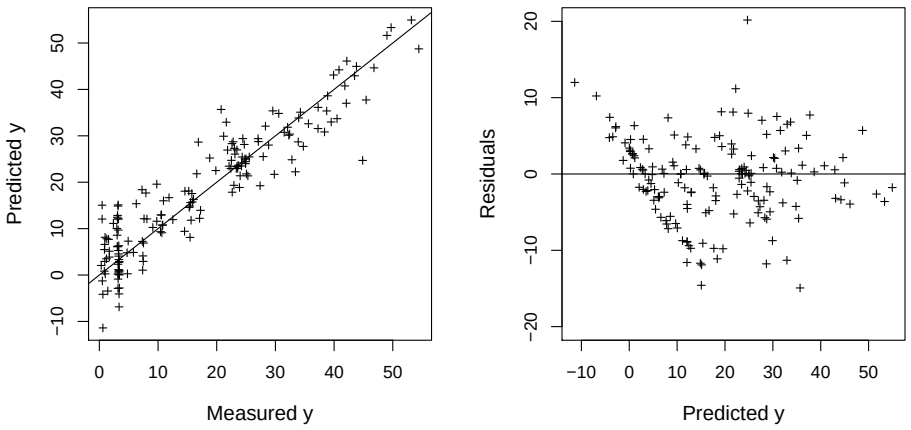

Figure 16: Output of function `plotprm`. Compare Figures 13 and 14. Left: Predicted versus measured glucose concentrations. Right: Residuals versus predicted values.

non-robust version. We can, however, observe artifacts in the residual plots which may be due to rounding errors.

Now that we have the optimal number of PLS components we can use it to get the estimates for the regression coefficients, the weights, scores, loadings and so on.

```
>   prm(X, y, a = res.prmcv$optcomp, opt = "l1m", usesvd = TRUE)
```

The argument `usesvd` is set to `TRUE` if the number of predictors exceeds the number of observations. The use of SVD and a slight modification of the algorithm make the computation faster in this case.

Repeated double CV may give a more precise answer on the optimal number of components than a single CV. Robust estimation on the other hand is usually more time consuming, and therefore repreated double CV will require considerably more time than single CV. Nevertheless, we will compare the results here. Repeated double CV for PRM can be executed as follows:

```
>   res.prmdcv <- prm_dcv(X, y, a = 40, opt = "median",repl=20)
```

We compute 40 components (which in practice may be too much). In total, 20 replications of the double CV scheme are performed. While the single CV for PRM needed about 4 minutes, repeated double CV requires more than 4 hours. Nevertheless, the plots of the results are interesting. We have the same diagnostic plots as for repeated double CV of the classical counterpart, and also the commands are similar.

The frequencies of the optimal numbers of components can be seen by:

```
> plotcompprm(res.prmdcv)
```

Figure 17 shows the resulting frequency distribution. There is a clear peak at 20 components. Note that here we obtain the same result as for single CV.

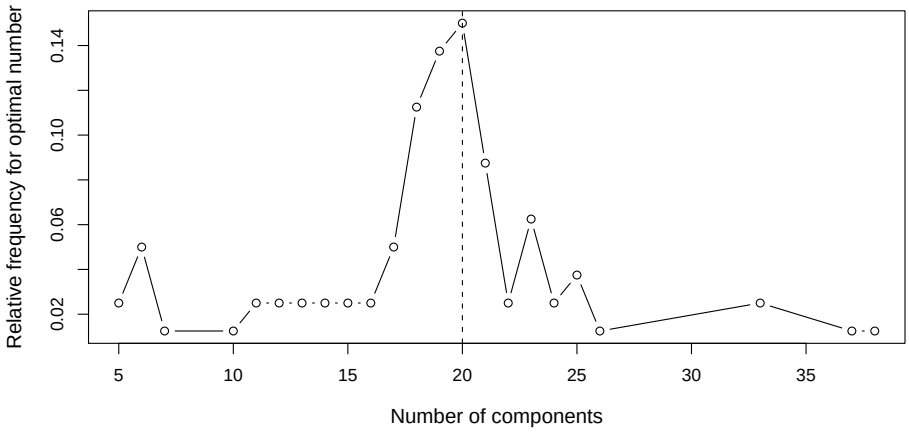

Figure 17: Output of `plotcompprm` for PRM-DCV. The optimal number of components is indicated by the vertical dashed line.

In a next plot the prediction performance measure is shown. Note that for robust methods a trimmed version of the SEP needs to be used in order to reduce the effect of outliers. Here we used a trimming of 20%.

```
> plotSEPprm(res.prmdcv,res.prmdcv$afinal,y,X)
```

The result of executing the above command is shown in Figure 18. The gray lines correspond to the results of the 20 repetitions of the double CV scheme, while the black line represents the single CV result. Obviously, single CV is much more optimistic than repeated double CV. Note that the single CV result is computed within this plot function, and thus this takes some time.

Similar as for repeated double CV for classical PLS, there are functions for PRM showing diagnostic plots:

```
> plotpredprm(res.prmdcv,res.prmdcv$afinal,y,X)
```

The predicted versus measured response values are shown in Figure 19. The left picture is the prediction from a single CV, while in the right picture the resulting predictions from repeated double CV are shown. The latter plot gives a clearer picture of the prediction uncertainty.

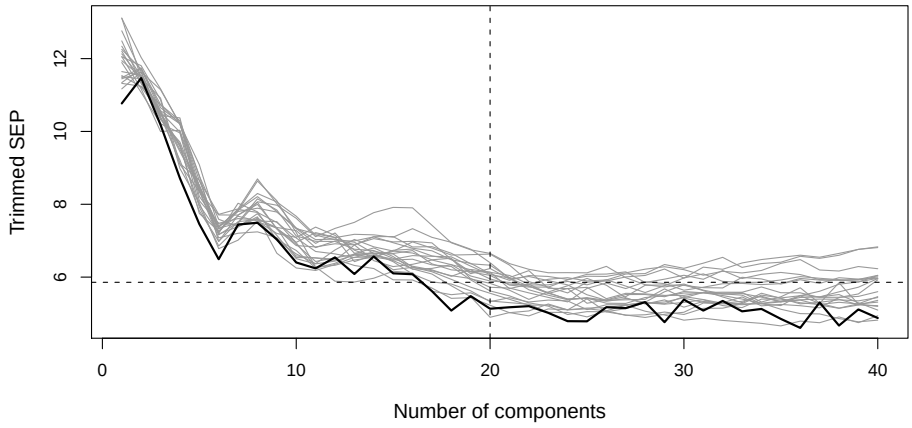

Figure 18: Output of `plotSEPprmdcv` for PRM. The gray lines result from repeated double CV, the black line from single CV.

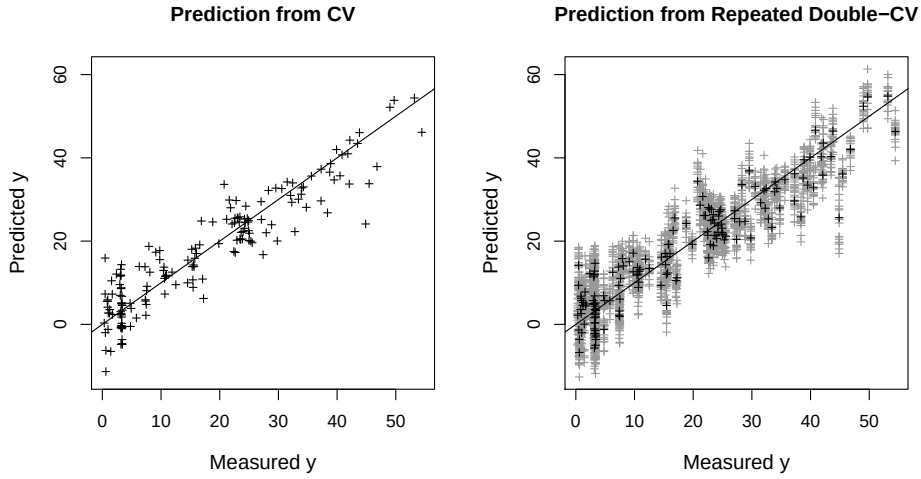

Figure 19: Predicted versus measured response values as output of `predprmdcv` for PRM. The left picture shows the results from single CV, the right picture visualizes the results from repeated double CV.

The residuals versus predicted values are visualized by:

```
> plotresprm(res.prmdcv,res.prmdcv$afinal,y,X)
```

The results are shown in Figure 20. Again, the left picture for single CV shows artifacts of the data, but the right picture for repeated double CV makes the uncertainty visible.

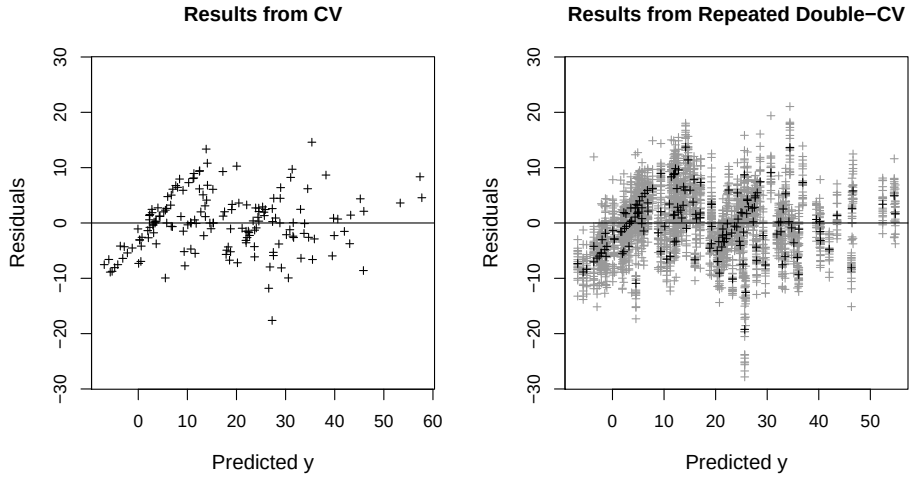

Figure 20: Output of `plotresprm` for PRM. The left picture shows the resisuals from single CV, the right picture visualizes the results from repeated double CV.

3.5 Ridge Regression

A method mainly used for multiple linear regression models like Equation (2) with much more predictors than objects and, consequently, multicollinear x-variables is Ridge regression. OLS estimation may lead to unbounded estimators in this case because an unreasonably large (positive or negative) coefficient can be offset by the appropriate coefficient of a correlated variable. To avoid this discomfort, Hoerl and Kennard (1970) suggested to penalize the objective function of OLS by a term depending on the coefficients.

For a univariate linear regression model (1)

$$\mathbf{y} = \mathbf{X} \cdot \mathbf{b} + \mathbf{e}$$

instead of minimizing the residual sum of squares (RSS) only, they estimate the coefficients by

$$\widehat{\mathbf{b}}_R = \underset{\mathbf{b}=(b_0,\ldots,b_m)}{\arg\min} \left\{ \sum_{i=1}^{n} (y_i - b_0 - \sum_{j=1}^{m} b_j x_{ij})^2 + \lambda_R \sum_{j=1}^{m} b_j^2 \right\} \tag{15}$$

where the penalty term does not contain the coefficient for the intercept. Thus the data origin remains untouched. λ_R, the so-called *shrinkage parameter*, is positive. If it was zero we would have the OLS model. Its name is motivated by the fact that the modified objective function causes a shrinkage of the estimated parameters.

The solution of the new optimization problem (15) in matrix form is

$$\widehat{\mathbf{b}}_R = (\mathbf{X}^\mathsf{T}\mathbf{X} + \lambda_R \mathbb{1})^{-1}\mathbf{X}^\mathsf{T}\mathbf{y}$$

and by adding a constant to the diagonal elements of $\mathbf{X}^\mathsf{T}\mathbf{X}$, its covariance matrix, $\mathbf{X}^\mathsf{T}\mathbf{X} + \lambda_R \mathbb{1}$,

becomes non-singular for appropriate λ_R. This *regularization* allows us to calculate the inverse in a numerically stable way. Note that the Ridge coefficients are a linear function of $\mathbf{y}$.

If $\widehat{\mathbf{b}}_{OLS} = (\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}\mathbf{X}^\mathsf{T}\mathbf{y}$ is the OLS estimator, the Ridge estimator can be written as

$$\widehat{\mathbf{b}}_R = (\mathbb{1} + \lambda_R(\mathbf{X}^\mathsf{T}\mathbf{X})^{-1})^{-1}\widehat{\mathbf{b}}_{OLS}.$$

Let us assume a regression model with i.i.d. residuals following a normal distribution

$$\mathbf{e} \sim N(\mathbf{0}, \sigma^2\mathbb{1}).$$

While the OLS estimator is unbiased with variance $\sigma^2(\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}$, for an increasing shrinkage parameter the Ridge estimator turns out to be more and more biased, but with a lower variance than in the OLS case:

$$\begin{aligned}\mathbb{E}[\widehat{\mathbf{b}}_R] &= (\mathbb{1} + \lambda_R(\mathbf{X}^\mathsf{T}\mathbf{X})^{-1})^{-1}\mathbf{b} \\ \mathrm{Var}[\widehat{\mathbf{b}}_R] &= \sigma^2(\mathbb{1} + \lambda_R(\mathbf{X}^\mathsf{T}\mathbf{X})^{-1})^{-1}(\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}(\mathbb{1} + \lambda_R(\mathbf{X}^\mathsf{T}\mathbf{X})^{-1})^{-1}\end{aligned}$$

The optimal shrinkage parameter is chosen by cross validation and finds the optimal tradeoff between bias and variance. Note that because of the occurring bias, only asymptotic confidence intervals and tests are available for Ridge regression (Firinguetti and Bobadilla 2009).

Ridge regression and PCR

Ridge regression can be seen as an alternative to principal component regression. In PCR, we usually use the first k principal components, which are ordered by decreasing eigenvalues. Remember that a large eigenvalue indicates a PC that aims to be very important for the prediction of $\mathbf{y}$ because it explains a big part of the total variance of the regressors. Here, k is usually chosen by cross validation and the critical boundary for the choice is relatively subjective. The remaining PCs are not considered at all in the final model, i.e. they have zero weight.

In Ridge regression, on the other hand, all variables are used with varying weights. The difference between important and less important variables is smoother than in PCR. Hastie et al. (2001) show that Ridge regression gives most weight along the directions of the first PCs and downweights directions related to PCs with small variance. Thus, the shrinkage in Ridge regression is directly related to the variance of the PCs.

Ridge regression in R

For Ridge regression, the **chemometrics** package provides the two functions `ridgeCV` and `plotRidge`. The most important result of the latter is the optimal value for the shrinkage

parameter λ_R. For a number of different parameter values, `plotRidge` carries out Ridge regression using the function `lm.ridge` from package **MASS**. By generalized cross validation (GCV, see appendix A), a fast evaluation scheme, the model that minimizes the prediction error MSEP is determined. Two plots visualize the results (see Figure 21).

```
>   res.plotridge <- plotRidge(y~., data=NIR.Glc, lambda=seq(0.5,10,by=0.05))
```

When the optimal parameter value is determined (1.25 in this case), the optimal model should be carefully evaluated by repeated cross validation (for our example we do 10-fold CV with 100 repetitions), using `ridgeCV` which also yields two plots (see Figure 22) and, additionally, some measures for prediction performance.

```
>   res.ridge <- ridgeCV(y~., data=NIR.Glc, lambdaopt=res.plotridge$lambdaopt,
      repl=100)
```

Ridge regression leads to an average SEP of 5.37 which is already a better result than achieved by PCR and PLS but stepwise regression is still doing better. The following method is only a slight modification of Ridge regression but with significantly different results.

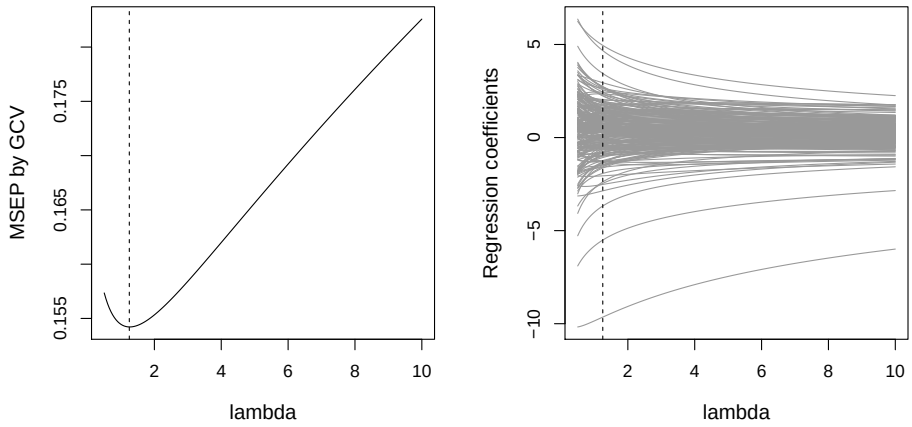

Figure 21: Output of function `plotRidge`. Left: The MSEP gained by GCV for each shrinkage parameter. The dashed vertical line at the minimum MSEP indicates the optimal λ_R. Right: For each λ_R, the regression coefficients are plotted. This shows very well the grade of shrinkage. The optimal parameter value from the left plot is shown by a vertical dashed line as well.

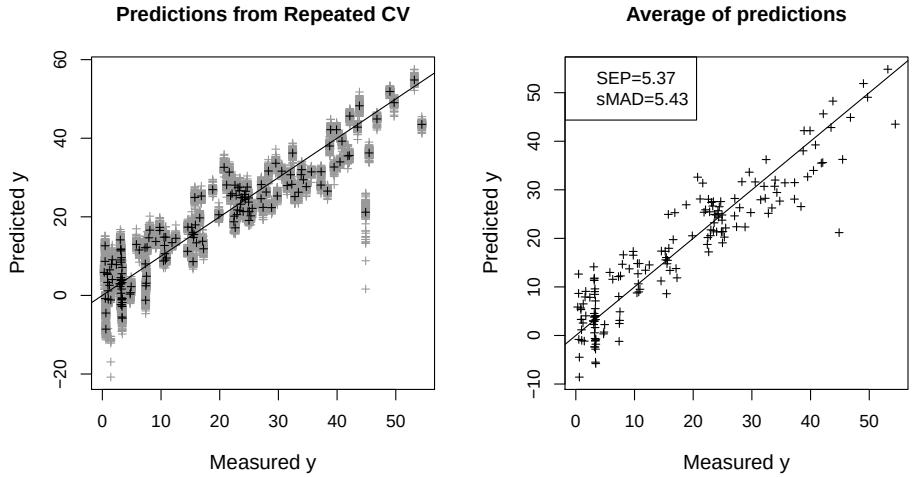

Figure 22: Output of function `ridgeCV`. The left plot opposes predicted and measured glucose concentrations, using a grey cross for each CV replication. The black crosses indicate the mean prediction. On the right side, we see only those black crosses of the mean predictions. Additionally the mean SEP and sMAD are displayed in the legendbox.

3.6 Lasso Regression

Just as in Ridge regression, in Lasso regression introduced by Tibshirani (1996) the OLS objective function is penalized by an additional term. While Ridge regression penalized the sum of squares of the regression coefficients (L_2-norm), Lasso regression uses the sum of absolute regression coefficients (L_1-norm):

$$\widehat{\mathbf{b}}_L = \underset{\mathbf{b}=(b_0,\ldots,b_m)}{\arg\min} \left\{ \sum_{i=1}^{n} (y_i - b_0 - \sum_{j=1}^{m} b_j x_{ij})^2 + \lambda_L \sum_{j=1}^{m} |b_j| \right\} \tag{16}$$

Here as well, the shrinkage parameter λ_L is positive, and the penalty term does not contain the intercept coefficient in order to keep the data origin.

There is the same variance-bias tradeoff as in Ridge regression but the Lasso bias is bounded for a fixed tuning parameter by a constant that does not depend on the true parameter values and thus it is more controllable (as commented in Knight's discussion of Efron et al. 2004). The optimal shrinkage parameter can be chosen as before by cross validation.

The new objective function comes down to a quadratic programming problem and its solution can no longer be written explicitly as a function of $\mathbf{y}$. A rather fast algorithm that accomplishes a stepwise orthogonal search was described by Efron et al. (2004). They show that a slight modification of their *Least Angle Regression* (LAR) algorithm implements the Lasso.

Lasso Regression and Variable Selection

To understand the R functions described below we write the objective function as a constrained optimization problem

$$\begin{aligned} &\min \sum_{i=1}^{n} (y_i - b_0 - \sum_{j=1}^{m} b_j x_{ij})^2 \\ &s.t. \sum_{j=1}^{m} |b_j| \leq s \end{aligned} \tag{17}$$

If λ_L in (16) is zero we obviously obtain the OLS solution and the problem is unconstrained which is equivalent to some maximum value of s; with increasing λ_L the coefficients are shrunk until in the extreme case the shrinkage parameter gets so high that all coefficients turn exactly zero. In the meantime the constraint value s in (17) decreases continuously to zero.

Here we see a major difference to Ridge regression: while the Ridge coefficients are in general different from zero Lasso forces some coefficients to be exactly zero and thus uses only a subset of the original x-variables in the final model. That means Lasso regression can be seen as an alternative *variable selection method*.

Lasso regression in R

For Lasso regression, the **chemometrics** package provides the two functions `lassoCV` and `lassocoef` the former of which is used first and provides the optimum constraint value. The latter calculates the Lasso coefficients of the final model.

```
>   res.lasso <- lassoCV(y~., data = NIR.Glc, fraction = seq(0, 1, by = 0.05),
      legpos="top")
```

The argument `fraction` expresses the absolute sum of Lasso coefficients for the current constraint value in terms of the absolute sum of OLS coefficients (which correspond to the unconstrained case):

$$\frac{\sum_{j=1}^{m} |b_j^L|}{\sum_{j=1}^{m} |b_j^{OLS}|} \tag{18}$$

Since Lasso regression is computationally more expensive than Ridge regression, we perform only a single 10-fold CV to determine the optimal model. In Figure 23 we see the mean MSEP values for each fraction (18) and their respective standard errors (more precisely, the standard errors multiplied by the argument `sdfact` which is 2 by default). The dashed horizontal line indicates the standard error above the minimum MSEP which serves as a bound to find the optimum fraction: the lowest fraction below that bound. Accordingly, the dashed vertical line at 0.05 shows the optimum fraction. The plot provides the mean SEP and MSEP values for the optimal model in the legend on top. We see here that for our example Lasso is not very competitive among the presented methods.

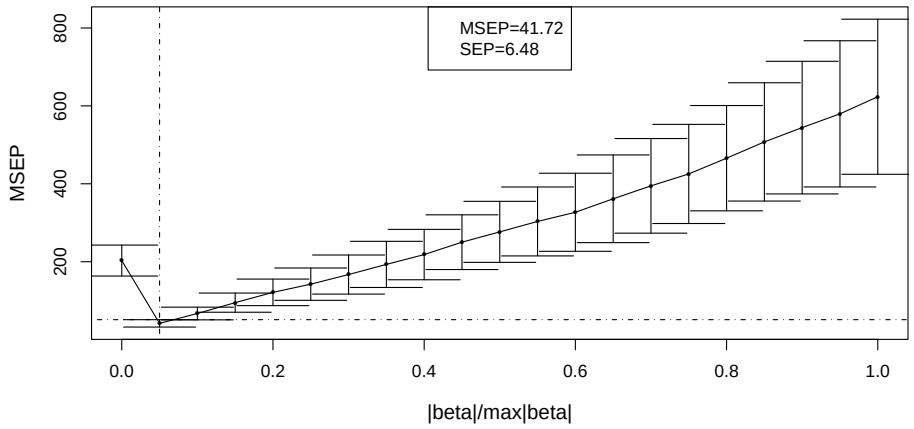

Figure 23: Output of function `lassoCV`. Similar to Figure 15, the means and standard deviations of the MSEP values calculated during CV are plotted. The dashed vertical and horizontal line indicate the optimal fraction value and the corresponding MSEP.

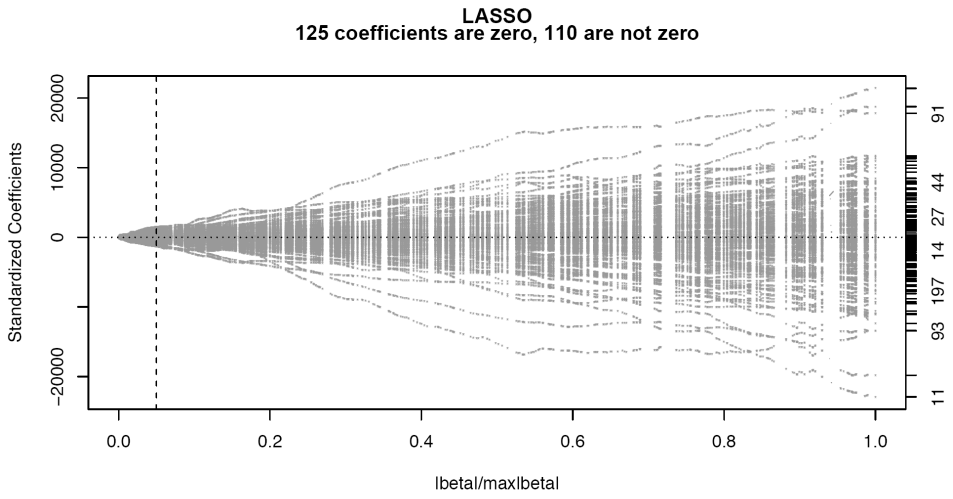

Figure 24: Output of function `lassocoef`. Development of the Lasso coefficients with increasing fraction. Dashed vertical line: optimal fraction determined before.

```
>   res.lassocoef <- lassocoef(y~., data=NIR.Glc, sopt=res.lasso$sopt)
```

The function `lassocoef` produces a plot (Figure 24) that shows the development of the Lasso coefficients as the fraction increases. Each line corresponds to one coefficient. Again, the vertical line indicates the optimal fraction as determined before. Besides the plot, the function provides the coefficients for the optimal fraction model as well as the number of coefficients that are zero and that are non-zero (this information can also be seen in the plot).

3.7 Comparison of Calibration Methods

The table below combines the results of this section. For each method it shows the classic SEP, the robust 20% trimmed SEP and the number of used variables or components.

We see that stepwise variable selection combined with OLS regression actually yields the best results. This makes us suspect that the data set does not contain outliers which disturb the analysis. In fact, a look at the plots illustrating the measured versus predicted values for each method (stepwise regression: Figure 6, PCR: Figure 9, PLS: Figure 13, Ridge regression: Figure 22) substantiates this suspicion because we cannot spot real outliers but only some sort of artefacts that occur due to the data structure. Anyway, none of the methods used here gives better results than stepwise OLS regression. Since in case of fulfilled prerequisites the OLS estimator is the *best linear unbiased estimator* (BLUE), there cannot be another method giving better results indeed. On the other hand we see that robust PLS (i.e. PRM) is performing somewhat better than classical PLS.

```
PREDICTION PERFORMANCE
Method      SEP     SEP20%     Nr. of Variables / Components

stepwise    4.61    4.4         16 variables
PCR         7.43    7.37        31 components
PLS         6.49    6.52        14 components
PRM-CV      5.4     4.95        20 components
PRM-DCV     5.95    5.86        20 components
Ridge       5.37    5.32       235 variables
Lasso       6.48    5.89       110 variables
```

Comparing the computation times (second table) and considering the type of algorithm used for the evaluation we can say that Ridge regression is a rather fast algorithm with a relatively good performance at the same time.

```
COMPUTATION TIMES
Method      Algorithm     Time needed

stepwise    100 x RCV       0min 44sec
PCR         100 x RDCV      2min 56sec
PLS         100 x RDCV      2min 27sec
PRM         single CV       4min 15sec
PRM         20 x RDCV     241min 15sec
Ridge       100 x RCV       1min 40sec
Lasso       single CV       0min 33sec
```

At this point it is important to underline that the conclusions made above of course hold only for the data we used here. Another data set that for example contains outliers may lead to a completely different rating of the regression methods.

4 Classification

Functions discussed in this section:

```
knnEval
nnetEval
svmEval
treeEval
```

Classification methods can be seen as a special case of prediction where each observation belongs to a specific known group of objects (for instance samples from different types of glass as used in Varmuza and Filzmoser 2009). Knowing about the class membership of all given

observations we can establish a *classification rule* that can be used to predict the class membership of new data. Hence, classification methods do not predict continuous characteristics but nominal scale variables. We call this way of establishing classification rules *supervised learning* because the used algorithm learns to classify new data by training with known data.

Like in regression, it is not enough to establish a prediction or classification rule with all the available data but we have to validate the resulting model for new data - for instance by dividing the data into training and test set (cross validation). As a performance measure that has to be minimized we use a misclassification error, i.e. a measure that tells us how many objects were not classified correctly. This can be the fraction of misclassified objects on the total number of objects:

$$\text{misclassification rate} = \frac{\text{number of misclassified objects}}{\text{number of objects}}$$

Concrete classification methods frequently used in chemometrics are for example linear discriminant analysis, PLS discriminant analysis, logistic regression, Gaussian mixture models, k nearest neighbor methods, classification trees, artificial neural networks or support vector machines. Since the R package **chemometrics** provides useful functions for the latter four methods we limit ourselves here to them. For other methods the interested reader may refer to Hastie et al. (2001) or Varmuza and Filzmoser (2009).

The provided functions are implemented to optimize necessary parameters of those methods with the big advantage that they all work according to the same scheme. They are made in a way that they require similar input and create similar and thus easily comparable output. The core of the functions `knnEval`, `nnetEval`, `svmEval` and `treeEval` is the evaluation of models with different parameter values in order to compare them and to choose the model with the optimal parameter.

This is done via *cross validation* (see appendix A): we provide the functions with the data with known class membership, a vector of indices for training data (this can be for instance two thirds of the whole data that are used to establish the classification rule) and a selection of parameter values to compare. The functions use the part of the data that is not in the training set as test data and basically compute three results for each parameter value:

- *Training error* = the misclassification rate that results if we apply the rule to the training data itself. Since the training data was used to find the rule, this is of course the most optimistic measure.
- *Test error* = the misclassification rate that results if we apply the rule to the test data.
- A number of s *CV errors* that are practically test errors that result from s-fold CV on the training set. The mean and standard deviation are used to depict the result. The CV error can be seen as the most realistic error measure for parameter optimization.

Those errors are plotted for each parameter value and the optimal value can be chosen according to the *one-standard-error-rule* described in section 4.1.

The data we shall use in this section result from a chemical analysis of $n = 178$ wines grown in the same region in Italy but derived from three different cultivars. Of each wine, $m = 13$ components were measured: alcohol, malic acid, ash, alcalinity of ash, magnesium, total phenols, flavanoids, nonflavanoid phenols, proanthocyanins, color intensity, hue, OD280/OD315 of diluted wines and proline (Frank and Asuncion 2010).

This is how we load and scale the data:

```
>   library(gclus)
>   data(wine)
>   X <- data.frame(scale(wine[,2:14]))   # data without class information
>   grp <- as.factor(wine[,1])            # class information
>   wine <- data.frame(X=X, grp=grp)
>   train <- sample(1:length(grp), round(2/3*length(grp)))
```

Scaling is necessary for `knnEval`, `nnetEval` and `svmEval`. We do not need it for `treeEval` but it is not a problem if the method is done with scaled data. The result will not change.

4.1 k Nearest Neighbors

Probably the simplest concept of classification is to conform to the neighbors of an object with unknown class membership. If the majority of the observations that are close to the object of interest is of class j, then we assign this object to class j as well.

In more detail, we assume n objects $\mathbf{x}_i = (x_{i1}, \ldots, x_{im})$ with the class information $y_i \in \{1, \ldots, p\}$ for p groups. If our task is to classify the new object $\tilde{\mathbf{x}}$, we calculate the (Euclidean) distance to all objects $\mathbf{x}_i$ and use the k objects with the smallest distance. The group that is most frequent among those k objects is assumed for $\tilde{\mathbf{x}}$. The parameter k has to be chosen optimally, i.e. in order to minimize the misclassification rate. This is once again done by CV.

Since for each object to be classified the distance to all other objects has to be calculated, be aware about possibly long computing times (especially for large data sets).

k-NN in R

The parameter k denoting the number of neighbors to be considered within the algorithm is optimized by (10-fold) cross validation by the following function:

```
>   knneval <- knnEval(X, grp, train, knnvec=seq(1,30), legpos="topright")
```

We input the whole wine data with its group memberships, the indices of training set objects and a selection of parameter values to be tried out. The algorithm calculates training, test and CV errors as described above and plots them as in Figure 25.

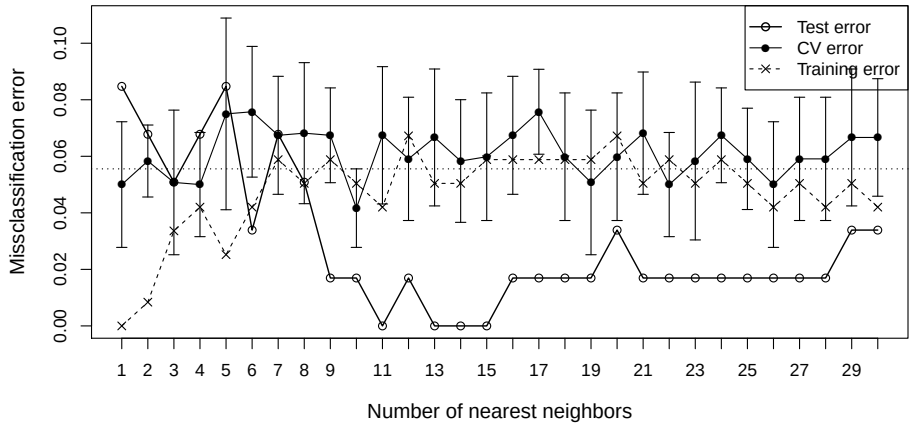

Figure 25: Output of `knnEval`. For each parameter value, the training and test error as well as the CV error mean and standard deviations are plotted. The dashed horizontal line corresponds to one standard error above the minimum CV error mean and is used to find the optimal number of neighbors according to the one-standard-error-rule.

For each parameter value, the solid points represent the means of the misclassification rates over all CV segments. Coming from these points, the standard deviations are visualized. The dashed horizontal line indicates the value one standard error above the minimal CV misclassification error mean, which is at $k = 10$. The lowest prediction error mean that lies below this line belongs to the model with $k = 1$ nearest neighbors. According to the *one-standard-error-rule* (Hastie et al. 2001), this is the optimal parameter.

```
optimal number of nearest neighbors: k = 1
test error at optimum:  0.0847
CV error threshold:     0.0556
```

Trying this several times, we see that the optimal parameter depends on the choice of the training set and the CV done within the evaluation scheme, so we repeat it 100 times and plot the frequencies of the optimal parameter values (Figure 26):

```
>   res <- array(dim=c(100,6))
>   colnames(res) <- c("k","trainerr","testerr","cvmean","cvse","threshold")
>   tt <- proc.time()
>   for (i in 1:100) {
      train <- sample(1:length(grp), round(2/3*length(grp)))
      knneval <- knnEval(X, grp, train, knnvec=seq(1,30), plotit=FALSE)
      indmin <- which.min(knneval$cvMean)
      res[i,6] <- knneval$cvMean[indmin] + knneval$cvSe[indmin]
      fvec <- (knneval$cvMean < res[i,6])
```

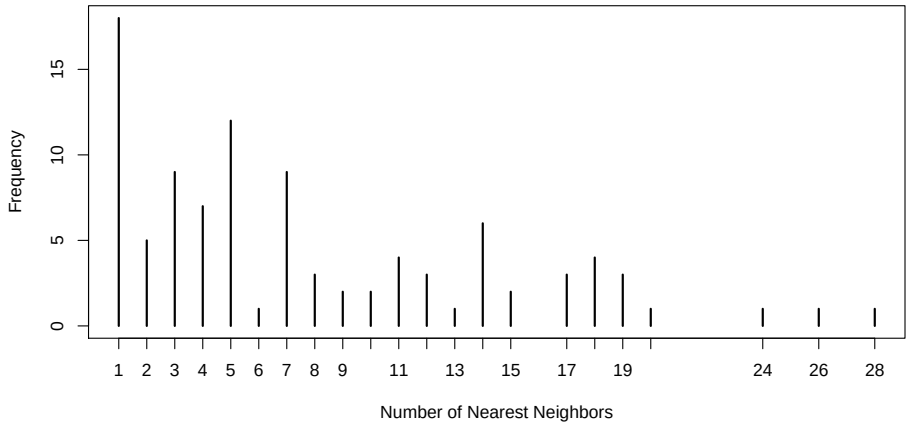

Figure 26: Frequencies of the optimal parameter values as they appear throughout the repetition of `knnEval`.

```
      indopt <- min((1:indmin)[fvec[1:indmin]])
      res[i,1] <- knneval$knnvec[indopt]
      res[i,2] <- knneval$trainerr[indopt]
      res[i,3] <- knneval$testerr[indopt]
      res[i,4] <- knneval$cvMean[indopt]
      res[i,5] <- knneval$cvSe[indopt]
    }
>   tt <- proc.time() - tt
>   plot(table(res[,1]), xlab="Number of Nearest Neighbors", ylab="Frequency")
```

The most frequent number of neighbors is $k = 1$, but also k = 5 and other appear rather often. Although this is not a very distinct result, if we observe the resulting errors we see that we obtain a very low misclassification error in any case, so it does not matter so much how many neighbors we choose to use.

```
                 median    sd
  training error   0.0169   0.0138
  test error       0.05     0.0228
  CV error mean    0.0258   0.0136
  (computing time for 100 repetitions: 4 min 24 sec)
```

After the optimal parameter is chosen, the actual classification rule (estimation of class memberships) can be determined by the function `knn` of package **class** which is also used by `knnEval` internally.

```
>   pred <- knn(X[train,], X[-train,], cl=grp[train], k = 1)
```

4.2 Classification Trees

The idea of classification trees (Breiman et al. 1984) is a very simple but powerful one. The space of objects with known class membership is partitioned by a hyperplane in a split point along one coordinate (*split variable*). The split variable and point have to be chosen in such a way that some misclassification measure is minimized. The resulting two regions are then partitioned again and again, until each region contains only objects from one class. The tree can be called *complete* or *full* then.

Step by step, as the regions become smaller also the misclassification error decreases until for the full tree the error measure for the training data is zero. Due to overfitting, the misclassification error for new cases can be expected to be far too high though, so we have to find the *optimal tree size* (number of regions) that minimizes not only the training error but also the test error and prune the complete tree down to it. This search is done by cross validation.

Given n objects $\mathbf{x}_i = (x_1, \ldots, x_m)$ with known class membership $y_i \in \{1, \ldots, p\}$, the optimal partition of the object space can be found by minimizing the sum of misclassification errors for each region, weighted by the number of objects in the respective region, and penalized with a term depending on the tree size which is weighted itself by a given complexity parameter - see Equation (19).

We define regions $R_1, \ldots, R_{|T|}$ where the number of regions, $|T|$, constitutes the tree size, and n_l, the number of objects in region l which obviously sum up to the total number of objects, n. Then, with the index function $I(y_i = j)$ that is 1 if object $\mathbf{x}_i$ belongs to group j and 0 otherwise,

$$p_{lj} = \frac{1}{n_l} \cdot \sum_{\mathbf{x}_i \in R_l} I(y_i = j)$$

is the relative frequency of group j objects that are located in region l.

A measure for misclassification that is more sensitive to changes in the relative frequencies than the misclassification rate we used before is the *Gini index* for the region l of a tree T:

$$Q_l(T) = \sum_{j=1,\ldots,p} p_{lj}(1 - p_{lj})$$

The Gini index is used to grow the full tree; for the optimization of the tree complexity we use the misclassification rate again.

Then we can define the objective function

$$\min_T \left\{ \sum_{l=1,\ldots,|T|} n_l Q_l(T) + \alpha |T| \right\} \tag{19}$$

The complexity parameter $\alpha \geq 0$ controls the tree size; $\alpha = 0$ yields the full tree. The higher α, the more the tree is pruned. With the help of cross validation with different values for the tree complexity, we reach our goal to find the smallest tree (i.e. the highest α) that still yields a sufficiently low misclassification rate.

When the optimal tree is found, new objects that fall into region l are assigned to the group $j(l)$ with the largest relative frequency in region l:

$$j(l) = \arg\max_j \{p_{lj}\}$$

Classification trees in R

The full tree can be grown and plotted using the function `rpart`:

```
>   tree <- rpart(grp~., data=wine, method="class")
>   plot(tree, main="Full Tree")
>   text(tree)
```

Figure 27 (left) shows that the first split is done where variable 14 has value 0.02574, the two subsequent regions are split along the variables 8 and 13, respectively. In the end the tree has five regions (terminal nodes of the diagram) and we have to find the optimal tree complexity α (alias `cp`) by (10-fold) cross validation. Note that the selected values for the tree complexity we input to `treeEval` are in descending order. This is reasonable because we want to use again the one-standard-error-rule to search for the *smallest* possible tree which corresponds to a *high* value of α.

```
>   treeeval <- treeEval(X, grp, train, cp=seq(0.45,0.05,by=-0.05))
```

We used the scaled data here although it is not necessary. As mentioned before, the result is the same as for the unscaled data except for changes in the splitting values. The split variables and the tree size stay the same. The results of a single execution of this function are presented in Figure 27 (right) and the following

```
optimal tree complexity: cp = 0.3
test error at optimum:  0.1525
CV error threshold:     0.1889
```

Again, in order to compensate instable results due to changes in the training set and the CV, we execute `treeEval` 100 times. The resulting errors are shown below and Figure 28 (left) gives us the frequencies of the optimal parameters.

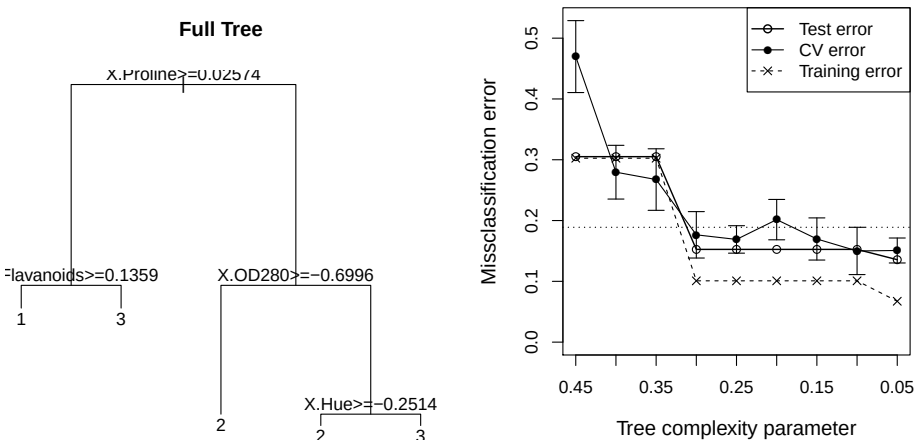

Figure 27: Left: Full classification tree for wine data showing the split variables and points that lead to 5 regions. Right: Output of function `treeEval`. The one-standard-error-rule can be applied.

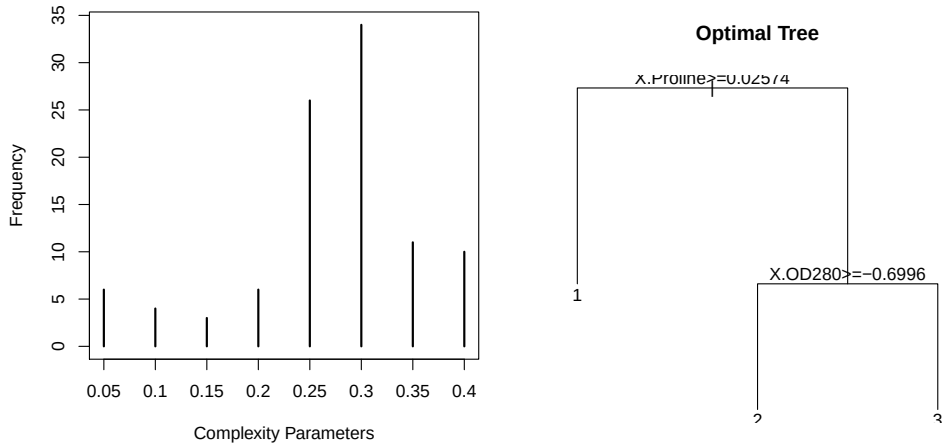

Figure 28: Left: Frequencies of optimal parameter values as they occur throughout the repetition of the evaluation. Right: Classification tree pruned to 3 regions after optimization of tree complexity.

```
                  median     sd
training error    0.0763   0.0325
test error        0.15     0.045
CV error mean     0.1432   0.04
(computing time for 100 repetitions: 7 min 31 sec)
```

We see that $\alpha = 0.3$ occurs most frequently, so we take it as the optimal complexity parameter. Pruning the full tree with this parameter, we obtain the optimal tree with 3 regions (see Figure 28, right). We use one more function from the package **rpart** to do this:

```
>   opttree <- prune(tree, cp=0.3)
>   plot(opttree, main="Optimal Tree")
>   text(opttree)
```

4.3 Artificial Neural Networks

In the style of neurons in the human brain, artificial neurons are modelled as devices with many inputs and one output. Artificial neural networks (ANNs - see Schalkoff 1997) are algorithms that "learn" by example - like humans or animals. Otto (2007) contains an overview over ANNs for chemometricians. More detailed information can be found in Cheng and Titterington (1994) or Jansson (1991).

Practically, an ANN comes down to a regression method where the response $\mathbf{y}$ is a binary variable in the case of two groups: its value is 0 for an object that belongs to the first group and 1 for the second group. For the general case of p groups we have to use p binary variables $\mathbf{y}_j$ being 1 if an object belongs to group j and 0 otherwise.

Neural networks do not model the $n \times p$ matrix $\mathbf{Y}$ directly by the $n \times m$ predictor matrix $\mathbf{X}$ but by a so-called *hidden layer* of variables between them. A nonlinear function (often the *sigmoid function*) of a linear combination of the x-variables builds r hidden layer variables $\mathbf{z}_k$:

$$\begin{aligned} &\mathbf{z}_k = \sigma(\mathbf{v}_k) = \frac{1}{1 - \exp(-\mathbf{v}_k)}, \\ &\text{where } \mathbf{v}_k = a_{0k} + a_{1k}\mathbf{x}_1 + \ldots + a_{mk}\mathbf{x}_m \\ &\text{and } k = 1, \ldots, r. \end{aligned}$$

Those *hidden units* can be used to form an additional hidden layer or to model $\mathbf{Y}$ directly by linear or nonlinear regression. The use of more than one hidden layer or nonlinear regression harbours a big risk of overfitting. That is why we limit ourselves here to one hidden layer that models the y-variables linearly:

$$\mathbf{y}_j = b_{0j} + b_{1j}\mathbf{z}_1 + \ldots + b_{rj}\mathbf{z}_r \text{ with } j = 1, \ldots, p.$$

The objective function of a neural network is the minimization not of the RSS but of the *cross entropy* (also called *deviance*):

$$\min \left\{ -\sum_{i=1}^{n} \sum_{j=1}^{p} \hat{y}_{ij} \log \hat{y}_{ij} \right\}$$

To avoid the danger of overfitting we introduce a regularization term into the objective. The idea is known from Ridge or Lasso regression: we add a penalty term called *weight decay* and obtain

$$\min \left\{ -\sum_{i=1}^{n} \sum_{j=1}^{p} \hat{y}_{ij} \log \hat{y}_{ij} + \lambda \sum (\text{parameters})^2 \right\} \tag{20}$$

with $\lambda \geq 0$ and where "parameters" means the values of all parameters used in the neural network. The optimal values for the parameters r and λ are obtained by cross validation. Unfortunately, we have to try out different combinations of weights and numbers of units. This is a rather time consuming and complicated procedure.

The resulting estimate $\widehat{\mathbf{Y}}$ contains values $\hat{y}_{ij} \in [0, 1]$ which describe the probability for the assignment of the object $\mathbf{x}_i$ to group j. The class with the highest probability is assumed for the respective object.

ANNs in R

In this case the function that we use for parameter tuning is `nnetEval` which requires a data matrix, a group vector and a selection of indices for training data. In contrast to the functions described before there are two parameters to be optimized. The structure of the function is the same though and so there are two versions of `nnetEval`: one with varying regularization parameters and one with selected numbers of hidden units.

```
>   weightsel <- c(0,0.01,0.1,0.15,0.2,0.3,0.5,1)
>   nneteval <- nnetEval(X, grp, train, decay=weightsel, size=5)
>   nneteval <- nnetEval(X, grp, train, decay=0.2, size=seq(5,30,by=5))
```

In order to find the optimal combination (r, λ) we have to try out several possibilities. Again, the result depends on the choice of the training set and the cross validation, so we will have to repeat the procedure again.

Since after some examination it turns out that the number of hidden units does not matter much in the sense of misclassification error, we will simply use $r = 5$ for the `size` parameter. This has the advantage of saving computing time which gets very high for large numbers of hidden units. 100 repetitions yield the following errors and the frequencies of optimal weight decay shown in Figure 29.

```
                  median     sd
training error    0        0.0013
test error        0.0169   0.0167
CV error mean     0.0167   0.0091
(computing time for 100 repetitions: 10 min 15 sec)
```

If we use 5 hidden units and a weight decay of 0.01 (see Figure 29), the plots produced by the function `nnetEval` (Figure 30) show slightly different errors for this parameter combination. This difference results from the cross validation.

```
>   par(mfrow=c(1,2))
>   nneteval <- nnetEval(wine, grp, train, size=5, legpos="topright",
      decay =  c(0,0.01,0.1,0.15,0.2,0.3,0.5,1))
>   nneteval <- nnetEval(wine, grp, train, decay=0.01, legpos="topright",
      size = c(5,10,15,20,25,30,40))
```

A concrete classification rule can be determined by `nnet` (package **class**) combined with `predict` which are also used by `nnetEval` internally.

```
>   rule <- nnet(X[train,], class.ind(grp[train]), size=5, entropy=TRUE,
      decay=0.01)
>   pred <- predict(rule, X[-train,])  # predicted probabilities for test set
>   pred <- apply(pred,1,which.max)    # predicted groups for test set
```

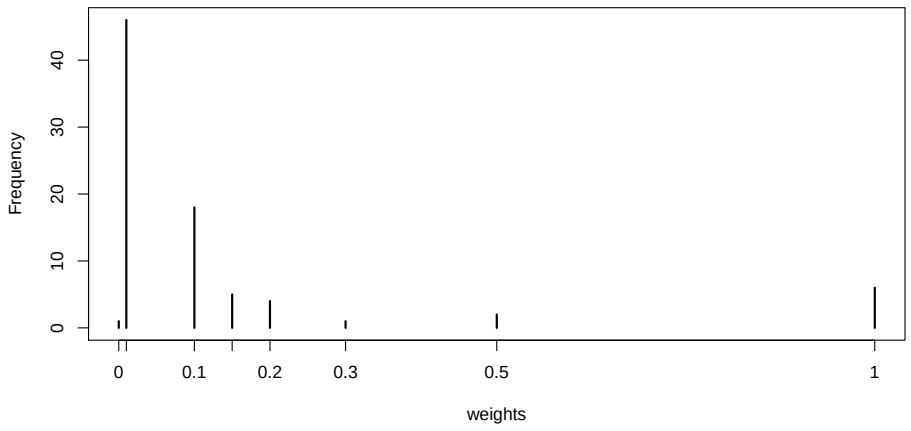

Figure 29: Frequencies of optimal weight decay values as they occur throughout the repetition.

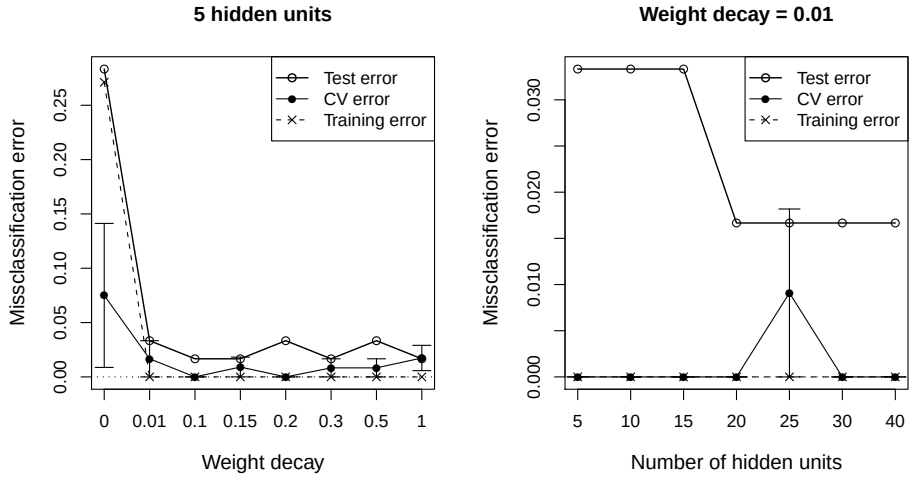

Figure 30: Output of function `nnetEval`. The left plot shows the variation of weight decay for 5 hidden unis, on the right side we see the errors for variation of hidden units at a weight decay of 0.01.

4.4 Support Vector Machines

Another method of supervised learning are support vector machines (Christianini and Shawe-Taylor 2000). For applications of SVM in chemistry see Ivanciuc (2007). The idea here is to transform the data into a space with higher dimension. This way the classes can become linearly separable, i.e. a single hyperplane dividing the new space separates the groups. The backtransformed hyperplane generally gives a nonlinear class separation. It can happen that the groups do not become perfectly separable by transforming the data into a higher dimensional space. However, there are ways to deal with that.

We aim to find the best linear division of the new space. We define it as the one that yields the biggest margin between the groups. In the case of non-separable groups we allow for objects lying on the wrong side of the hyperplane and constrain the sum of their distances from the class. If, for instance, the transformed space has two dimensions, one margin line is "supported" by two objects of one group and the other one is a parallel line through one point of the other group. The best straight line to separate the groups is halfway between the margin lines. This motivates the name *support vector machine*. In the case of more dimensions of the transformed space we have to use accordingly more points to support the hyperplane.

The transformation of the original data space is made by basis expansion: each m-dimensional object $\mathbf{x}_i$ is transformed into the new space by r basis functions ($r > m$):

$$\mathbf{h}(\mathbf{x}_i) = (h_1(\mathbf{x}_i), \ldots, h_r(\mathbf{x}_i)) \text{ for } i = 1, \ldots, n.$$

Thanks to the *kernel trick* (Boser et al. 1992) we do not have to specify the basis functions

explicitly but we can replace products $\mathbf{h}(\mathbf{x}_i)^\mathsf{T}\mathbf{h}(\mathbf{x}_j)$ by a *kernel function*

$$K(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{h}(\mathbf{x}_i)^\mathsf{T}\mathbf{h}(\mathbf{x}_j)$$

The objective function of our optimization problem (21) below can be expressed in a way that it contains $\mathbf{x}_i$ and $\mathbf{x}_j$ only as the product $\mathbf{x}_i^\mathsf{T}\mathbf{x}_j$, or rather $\mathbf{h}(\mathbf{x}_i)^\mathsf{T}\mathbf{h}(\mathbf{x}_j)$ if we consider the transformed data, and this fact allows us to apply this kernel trick. Common choices for kernel functions are:

$$K(\mathbf{x}_i, \mathbf{x}_j) = \begin{cases} (1 + \mathbf{x}_i^\mathsf{T}\mathbf{x}_j)^d & d^{\text{th}} \text{ degree polynomial} \\ \exp(-c\|\mathbf{x}_i - \mathbf{x}_j\|^2) & \text{radial basis function (RBF) with } c > 0 \\ \tanh(c_1\mathbf{x}_i^\mathsf{T}\mathbf{x}_j + c_2) & \text{neural network with } c_1 > 0 \text{ and } c_2 < 0 \end{cases}$$

If after this transformation the classes are linearly separable and if we assume two classes of objects and a response vector $\mathbf{y}$ that is -1 for the first group and +1 for the second, we can find a hyperplane

$$b_0 + \mathbf{b}^\mathsf{T}\mathbf{x} = 0 \text{ with } \mathbf{b}^\mathsf{T}\mathbf{b} = 1$$

that assigns an object $\mathbf{x}_i$ to the first group if $b_0 + \mathbf{b}^\mathsf{T}\mathbf{x}_i < 0$ and to the second group otherwise. Perfect group separation is possible if

$$y_i(b_0 + \mathbf{b}^\mathsf{T}\mathbf{x}_i) > 0 \text{ for } i = 1, \ldots, n.$$

The left hand side of Figure 31 (compare Varmuza and Filzmoser 2009) shows two groups of objects that are *linearly separable* in the two-dimensional space. The dashed lines with distance M illustrate the largest margin between the classes, one of them supported by two points of one group and the other one by a point of the other group. The separating hyperplane, a straight line (solid) in this case, lies exactly in the middle of the margin, at a distance $M/2$ between the dashed lines.

For the maximization of the margin M between the classes we search a scalar b_0 and a unit vector $\mathbf{b}$ such that all objects are more than $M/2$ away from the hyperplane.

$$\max M \text{ with respect to } b_0 \text{ and } \mathbf{b} \text{ (where } \mathbf{b}^\mathsf{T}\mathbf{b} = 1) \tag{21}$$

$$\text{s.t. } y_i(b_0 + \mathbf{b}^\mathsf{T}x_i) \geq M/2 \text{ for } i = 1, \ldots, n \tag{22}$$

For the linearly non-separable case which we can see in Figure 31 (right), we have to modify the condition (22) by introducing *slack variables* ξ_i that are 0 for objects on the correct side of the margin and a positive distance otherwise. We restrict the sum of those distances with a constant and the optimization problem is the following:

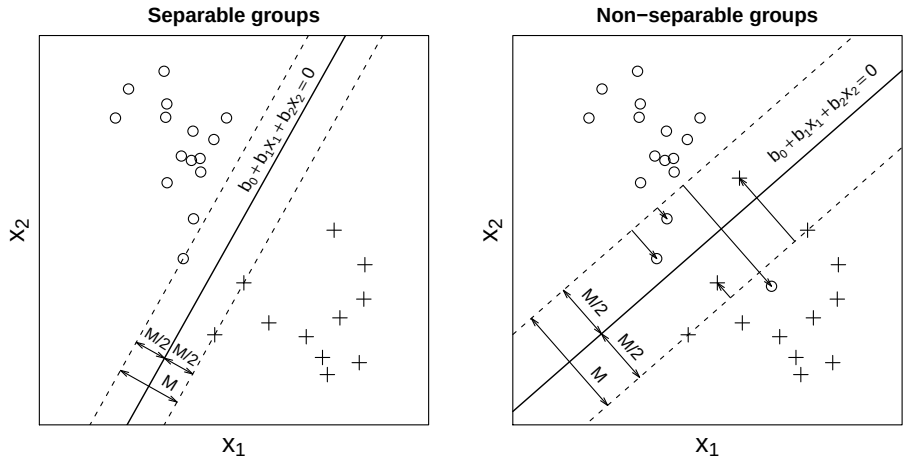

Figure 31: Left: Decision boundaries for separable groups supported by three points. Right: Margin for non-separable groups. The distances of objects that are on the wrong side are marked.

$$\begin{aligned}
&\max M \text{ with respect to } b_0 \text{ and } \mathbf{b} \text{ (where } \mathbf{b}^\mathsf{T}\mathbf{b} = 1) \\
&\text{s.t. } y_i(b_0 + \mathbf{b}^\mathsf{T} x_i) \geq {}^{M}/_{2}(1 - \xi_i), \\
&\xi_i \geq 0, \sum_i \xi_i \leq \text{const. for } i = 1, \ldots, n
\end{aligned}$$

This quadratic programming problem (quadratic objective function with linear inequality conditions) can be solved if the constant in the restriction for the slack variables is specified. To constrain the sum of the ξ_i we introduce a parameter γ that forces the sum to be small and thus allows only few objects on the wrong side of the margin. This may cause very complex boundaries in the original space and work well for the training data but can easily lead to problems with new data. A larger value of γ, on the other hand, avoids overfitting and yields smoother boundaries. The parameter can be optimized via CV.

SVM in R

For selected values of γ, we execute `svmEval` with RBFs and obtain Figure 32 (left).

```
>   gamsel <- c(0.001,0.01,0.02,0.05,0.1,0.15,0.2,0.5,1)
>   svmeval <- svmEval(X, grp, train, gamvec=gamsel, kernel="radial",
      legpos="top")
```

If we repeat the procedure for 100 different training sets the most frequent optimal γ turns out to be 0.01, as shown in Figure 32 (right). The medians of the resulting errors are as follows:

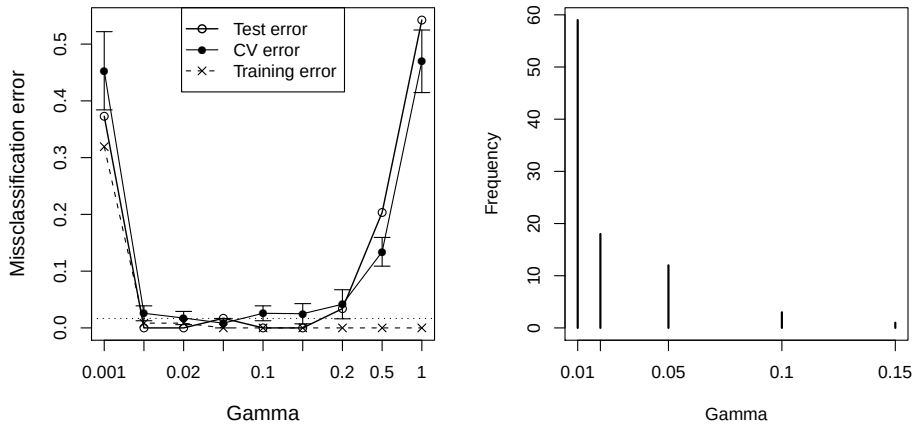

Figure 32: Left hand side: Output of function `svmEval`. Right hand side: The frequencies how often a parameter value turned out to be optimal throughout the repetition of `svmEval`.

```
                median     sd
training error  0.0085   0.0061
test error      0.0167   0.0145
CV error mean   0.0167   0.0081
(computing time for 100 repetitions: 7 min 16 sec)
```

For the optimal parameter $\gamma = 0.01$ we can establish the classification rule using the training set by `svm` from package **e1071** and obtain predictions for the test set by `predict`:

```
>   rule <- svm(X[train,],grp[train], kernel="radial", gamma=0.01)
>   pred <- predict(svmres, X[-train,])
```

4.5 Comparison of Classification Methods

Putting the results of the four methods we used in this section to classify our wine data together we can compare the test errors. We chose this error measure because it is the most realistic one for new data. Since we repeated each evaluation scheme 100 times we have test errors available that belong to the respective optimal parameter choice of each repetition. Figure 33 illustrates the distribution of those test errors for each method by a boxplot.

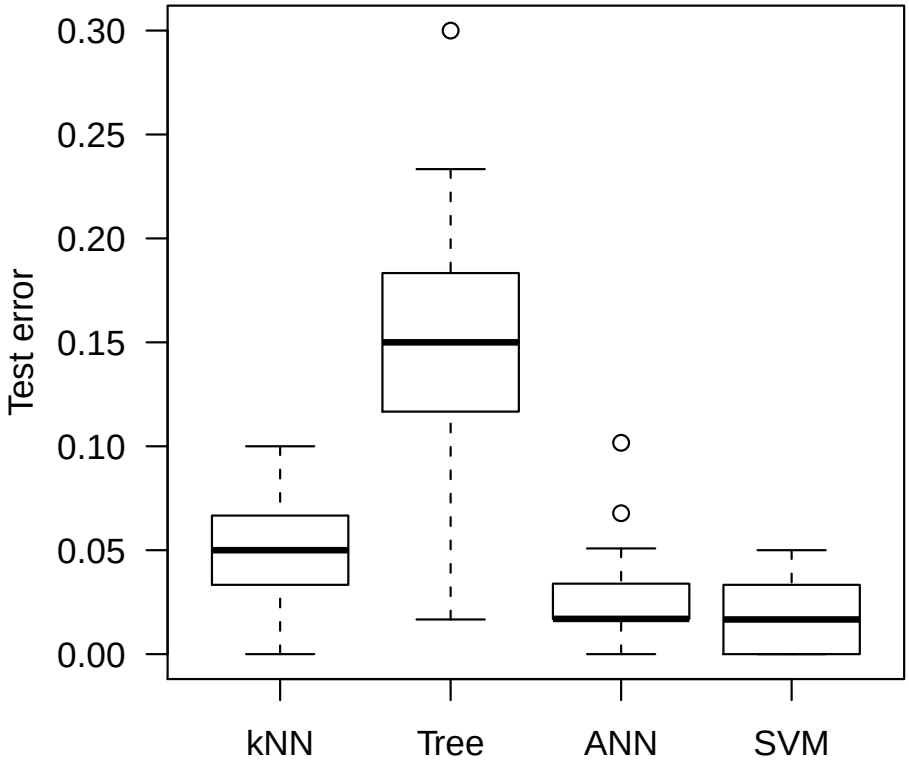

Figure 33: Comparison of test errors resulting from 100 times repeated evaluation schemes of the methods k nearest neighbors, classification tree, artificial neural networks and support vector machines.

Method	Median test error	Computing time
kNN	0.05	4 min
Tree	0.15	8 min
ANN	0.017	10 min
SVM	0.017	7 min

We can see how much classification trees depend on the choice of the training set. They are relatively instable and lead to a high test error compared to the other methods. Neural networks and support vector machines yield extremely good results for this data set, yet SVM seems to be the faster and more practical choice. Remember that the computation of ANNs becomes extremely slow for a higher number of hidden units than we use here, and the simultaneous optimization of the two parameters is a quite complex task. kNN gives us a slightly higher test error which is still in an acceptable range though. This method is a good alternative if we want to save computation time.

A Cross Validation

For model generation and model testing it is important to obtain realistic performance estimates for new cases. To optimize the performance of a model only for the data that was used for its creation may lead to *overfitting*, that means the model fits the sample data perfectly but does not predict new data well. Therefore, the data at hand is usually split into three subsets: *training*, *validation* and *test set*. The first two are then used for model selection and the latter for testing the model on new data.

If the original data set contains a large number of objects this is enough, however, very often in chemometrics only a relatively small number of objects is available. Useful tools in this case are resampling strategies like *cross validation*. In calibration, for instance, it is desireable to work with a reasonable high number of predictions when optimizing the model complexity on the one hand and estimating the prediction performance for new objects on the other hand. Analogously, this holds for classification.

Cross Validation (CV)

For the optimization of the model complexity, like the number of PLS components or PCs, or the number of nearest neighbors, the data is split into s segments, $s-1$ of which are used as training set and the remaining one as validation set. Cross validation with four segments for example is called *4-fold CV*; CV with n segments (and hence all subsets of size $n-1$ as training sets) is known as *leave-one-out* CV (LOO-CV).

With the training set we estimate models with different complexities $a = 1, \ldots, A$ and with each of these models we calculate the predicted values for the validation set objects. This is repeated $s-1$ times, each time with a different segment as validation set, resulting in an $n \times A$ matrix of predicted values for all n objects and each of the A models. From this matrix, we calculate the prediction error for each model and choose, for instance, the model with the lowest prediction error as the optimal one.

Repeated Cross Validation (RCV)

Not only in order to obtain a larger number of predictions but also to avoid the dependence of the predictions on the data split, it is recommended to repeat the above described procedure k times. That means we split the data k times into training and validation set and, accordingly, obtain k $(n \times A)$ prediction matrices and k prediction errors for each model. Their mean and standard deviation make a more careful choice of the optimal model complexity possible. Hastie et al. (2001) describe a useful *one-standard-error-rule* that consists in finding the lowest model complexity whose prediction error mean is below the minimal prediction error mean plus one standard error (see evaluation figures of chapter 4).

Double Cross Validation (DCV)

The issue of obtaining reliable estimates of the prediction performance for new cases can be handled by splitting the data into calibration set and test set first, then applying CV to the calibration set to optimize the model complexity and finally using the test set for the estimation of realistic prediction errors.

In detail, this is done as follows: in an outer CV loop, split the data into s_0 segments, $s_0 - 1$ of which are used as calibration set and the remaining one as test set. CV on the calibration set (s-fold, in an inner loop) yields a model of optimal complexity which is applied to the test set. Repeating this procedure $s_0 - 1$ times, until each segment was test set once, we obtain s_0 optimal models (one for each segment) and $n \times p \times A$ test-set-predicted values (for each segment and each model complexity calculated with the according model). The final model complexity can be taken as the median of the s_0 optimal values.

Repeated Double Cross Validation (RDCV)

In order to obtain a higher number of predictions and thus a more reliable estimation, it is recommendable to repeat the above described procedure k times (Filzmoser et al. 2009). This results in $s_0 \times k$ optimal models and $n \times p \times A \times k$ test-set-predictions. The final model complexity can be for instance chosen as the one that appears with the highest frequency. More selection rules for this case are described in chapter 3 concerning the **chemometrics** function `mvr_dcv`. Interesting insight can be achieved by analyzing the histogram or density of the predictions.

Generalized Cross Validation (GCV)

The MSE of the computationally relatively expensive LOO-CV can be approximated by a function depending on the RSS and the Hat-Matrix resulting from (OLS, Ridge, ...) regression with all objects. This approximation is known as GCV and saves a lot of computation time.

In OLS regression, the estimation of $\mathbf{y}$ can be described by the *Hat-Matrix* $\mathbf{H}$:

$$\begin{aligned} \widehat{\mathbf{y}} &= \mathbf{X}\widehat{\mathbf{b}} = \mathbf{X}(\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}\mathbf{X}^\mathsf{T}\mathbf{y} = \mathbf{H}\mathbf{y} \\ \mathbf{H} &= \mathbf{X}(\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}\mathbf{X}^\mathsf{T} \end{aligned} \tag{23}$$

If h_{ii}, $i = 1, \ldots, n$ denotes the diagonal elements of the matrix $\mathbf{H}$ which results from OLS regression with all n objects, the relation

$$\mathrm{MSE}_{LOO} = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{y_i - \hat{y}_i}{1 - h_{ii}}\right)^2 \tag{24}$$

holds for the MSE of LOO-CV without actually executing LOO-CV. The diagonal elements h_{ii} of the Hat-Matrix can be approximated by their average, $h_{ii} \approx \frac{\mathrm{tr}\,\mathbf{H}}{n}$, where the trace of $\mathbf{H}$

is the sum of its diagonal elements. This leads to the approximation of (24):

$$\text{MSE}_{LOO} \approx \frac{1}{\left(1 - \frac{\text{tr}\,\mathbf{H}}{n}\right)^2} \cdot \frac{RSS}{n}$$

For Ridge regression where the Hat-Matrix is $\mathbf{H} = \mathbf{X}(\mathbf{X}^\mathsf{T}\mathbf{X} + \lambda_R \mathbb{1})^{-1}\mathbf{X}^\mathsf{T}$ or for any other regression method where the estimation is done like in Equation (23) with a Hat-Matrix that depends only on the x-variables, GCV can be used analogously (Golub et al. 1979).

B Logratio Transformation

Functions discussed in this section:

```
alr
clr
ilr
```

B.1 Compositional Data

In chemistry, we often deal with data drawn from a more restricted space than for instance the set of real numbers. An example are concentrations of chemical compounds in a mixture. Aitchison (1986) defined:

Definition B.1. *"In statistics,* compositional data *are quantitative descriptions of the parts of some whole, conveying exclusively relative information." That means we are working with data (also called* closed data*) that can be represented as percentages, resulting in a constant row sum.*

The rows of a compositional data matrix $\mathbf{X} \in \mathbb{R}^{n \times m}$*,* $\mathbf{x}_i = (x_{i1}, x_{i2}, \ldots, x_{im})$*,* $i = 1, \ldots, n$*, satisfying* $x_{ij} > 0$ *and* $\sum_{j=1}^{m} x_{ij} = \kappa$ *are called* composition*, each element* x_{ij} *of a composition is a* component*.* κ *is a non-negative real constant specifying the row sum.*

The restriction of a constant row sum constitutes the major difficulty we encounter when dealing with compositional data.

As the total row sum is known, one component of a compositional data vector can be omitted. That means the data matrix does not have full rank but its rows are located in an $m - 1$ dimensional subspace of $\mathbb{R}^m$, which can be described by a so-called *simplex* as explained in the following.

Since a composition contains only relative information, multiplication by a non-negative constant leads to an equivalent vector. Each equivalence class in this spirit is represented by one

of its elements. The set of all representatives finally spans the sample space of compositional data:

$$S^m = \{\mathbf{x}_i = (x_{i1}, \ldots, x_{im}) \in \mathbb{R}^m : x_{ij} > 0, \sum_{j=1}^{m} x_{ij} = \kappa\}$$

Three phenomenons mainly represent the challenge of statistical inference carried out on such data:

1. If we try to apply classic methods (e.g. calculation of confidence ellipsoids) within the simplex without considering the special structure of the data we risk to leave the restricted area and reach, for example, negative values which is of course a rather meaningless result.
2. *Spurious correlations* between the components are a result of the fact that all variable values have to change as soon as one variable's value is modified, if we want the row sum to stay constant.
3. The arithmetric mean of compositional data has no physical meaning. In the simplex geometry, it is more recommendable to use the geometric mean instead.

B.2 Logratio Transformation

As a solution for the problems with compositional data, Aitchison (1986) suggested to transform the data from the simplex S^m to the unrestricted set of real numbers using the logratio transformation, to analyze the transformed data in the traditional way and, if needed, to transform the results back to the simplex for interpretation.

Figure 34 shows an artificial dataset of trivariate compositional data in a ternary diagram. The data has been alr-transformed to $\mathbb{R}^2$ using component number 2 as divisor (see below for details on alr) and tolerance ellipsoids that cover in case of normal distribution a certain percentage of the data points have been calculated. The back-transformed tolerance ellipses are deformed in the simplex, and the closer the lines approach to the boundary, the larger is the deformation. This gives an impression of the geometry of the simplex. Elliptically symmetric distributions in the unrestricted space thus show a different distribution in the simplex, and close to the boundary the difference gets more and more pronounced. A statistical analysis directly of the compositional data in the simplex will thus in general be misleading.

For a given $n \times m$ data matrix $\mathbf{X}$ containing n compositions in S^m three types of logratio transformations are implemented in **chemometrics**.

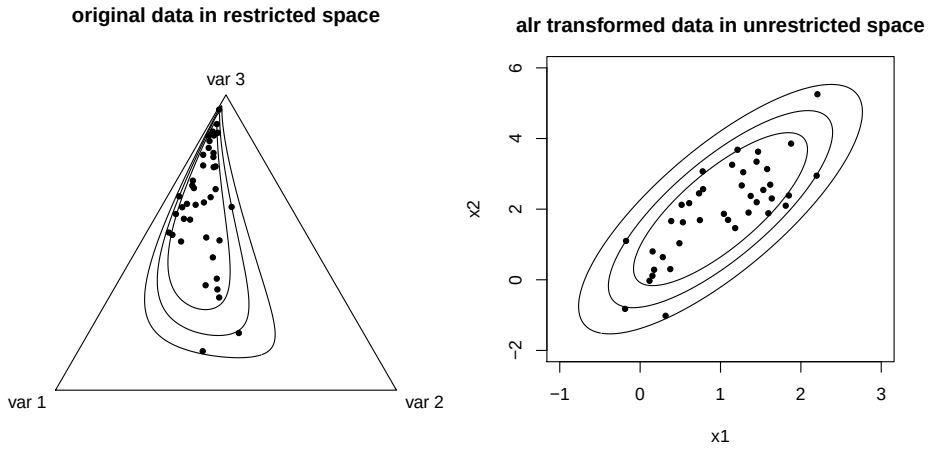

Figure 34: Left: Compositional data in its original space with tolerance regions that result from the ellipses on the right side. Right: alr-transformed data in $\mathbb{R}^2$ and tolerance ellipses calculated with classic methods.

Additive Logratio Transformation

It seems obvious to transform data from the $m-1$ dimensional restricted space S^m to the open $m-1$ dimensional real space $\mathbb{R}^{m-1}$. Using one priorily specified component, e.g. number k, as common divisor for all the others, the *additive logratio transformation* for the i^{th} row is defined as:

$$x_{ij}^{\text{alr}} = \log \frac{x_{ij}}{x_{ik}},$$

where $j = 1, \ldots, m$ and $j \neq k$.

Since the resulting vector still contains the relation to $m-1$ of the original components, it can be interpreted easily. The choice of the divisor component is rather subjective, though, and causes the transformation to be asymmetric in the components. However, the more problematic defect of the alr transformation is the fact that it is not isometric. So, if your analysis needs distances and lengths of vectors to be preserved in terms of the respective inner product, alr is not suitable.

The corresponding R command, `alr`, requires (besides the data matrix) the variable that should be used as a divisor:

```
>   X_alr <- alr(X, divisorvar=2)
```

Centered Logratio Transformation

To avoid the problems of the above described transformation, Aitchison also specifies the *centered logratio transformation*. As it uses the geometric mean $\bar{x}_{G,i}$ of each composition as a divisor instead of a single component, it is symmetric, and the transformation is defined as

$$x_{ij}^{\text{clr}} = \log \frac{x_{ij}}{\bar{x}_{G,i}}$$

for $j = 1, \ldots, m$.

Obviously, this transformation maps from the $m-1$ dimensional S^m to the m dimensional $\mathbb{R}^m$ and thus is not injective, resulting in singular covariance matrices of the transformed data. However, if this characteristic is not important for your analysis, clr is a good choice as it is an isometric transformation. Furthermore, just as in the alr case, the interpretation of clr transformed data is still simple because the relation to the original components is preserved.

```
>   X_clr <- clr(X)
```

Isometric Logratio Transformation

Although, as an isometric transformation, clr is already a widely useable tool, it still does not preserve the angles between compositions and, as mentioned, the covariance matrices do not have full rank. So, to overcome these last problems, Egozcue et al. (2003) developed the *isometric logratio transformation*. As the name says, it transformes data from the restricted $m-1$ dimensional simplex S^m isometrically to a $m-1$ dimensional subspace of $\mathbb{R}^m$. Thus, covariance matrices of the resulting data have full rank.

The isometric logratio transformation is based on an orthogonal basis of the clr space. Since of course the choice of this basis is not unique, indeed we are talking about a family of transformations that preserves not only lengths and distances but also the angles between compositions.

The ilr transformation for a composition $\mathbf{x}_i \in S^m$ is defined as

$$\mathbf{x}_i^{\text{ilr}} = \mathbf{V}^T \cdot \mathbf{x}_i^{\text{clr}}.$$

$\mathbf{V}$ denotes the $m \times (m-1)$ matrix containing in its columns the vectors of the orthogonal basis.

The basis used for the **chemometrics** function `ilr` is orthonormal, and the i^{th} basis vector is specified by

$$\mathbf{v}_i = \exp\left(\sqrt{\frac{1}{i(i+1)}} \cdot \Big(\underbrace{1, \ldots, 1}_{i \text{ times}}, -1, \underbrace{0, \ldots, 0}_{m-(i+1) \text{ times}}\Big)^{\mathsf{T}}\right).$$

```
>   X_ilr <- ilr(X)
```

After applying ilr, one can analyze the data with the standard tools. However, in order to interpret the result, it is recommended to transform the data back into the clr space because in the ilr space the relation to the original components is not preserved and interpretation is not that obvious.

References

J. Aitchison. *The Statistical Analysis of Compositional Data*. Chapman & Hall, London, United Kingdom, 1986.

B. E. Boser, I. Guyon, and V. Vapnik. A training algorithm for optimal margin classifiers. In *Computational Learing Theory*, pages 144–152, Pittsburgh, PA, 1992. ACM Press.

L. Breiman, J. H. Friedman, R. H. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, Belmont, CA, 1984.

B. Cheng and D. M. Titterington. Neural networks: A review from a statistical perspective. *Stat. Sci.*, 9:2–54, 1994.

N. Christianini and J. Shawe-Taylor. *An Introduction to Support Vektor Machines and Other Kernel-Based Learning Methods*. Cambridge University Press, Cambridge, NY, 2000.

D. J. Cummins and C. W. Andrews. Iteratively reweighted partial least squares: A performance analysis by Monte Carlo simulations. *Journal of Chemometrics*, 9:489–507, 1995.

S. de Jong. SIMPLS: An alternative approach to partial least squares regression. *Chemom. Intell. Lab. Syst.*, 18:251–263, 1993.

B. Efron, T. Hastie, I. Johnstone, and R. Tibshirani. Least angle regression (with discussion). *Annals of Statistics*, 32(2):407–499, 2004.

J. J. Egozcue, V. Pawlowsky-Glahn, G. Mateu-Figueros, and C. Barcelo-Vidal. Isometric logratio transformation for compositional data analysis. *Mathematical Geology*, 35(3):279–300, April 2003.

P. Filzmoser, B. Liebmann, and K. Varmuza. Repeated double cross validation. *Journal of Chemometrics*, 23(4):160–171, 2009.

L. Firinguetti and G. Bobadilla. Asymptotic confidence intervals in ridge regression based on the edgeworth expansion. *Statistical Papers*, pages 1–21, 2009.

A. Frank and A. Asuncion. UCI machine learning repository, 2010. URL `http://archive.ics.uci.edu/ml`.

M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, P. Alken, M. Booth, and F. Rossi. *GNU Scientific Library Reference Manual (3rd. Edition)*, 2009. URL `http://www.gnu.org/software/gsl/`. ISBN 0-9546120-7-8.

G. H. Golub, M. Heath, and G. Wahba. Generalized cross-validation as a method for choosing a good ridge parameter. *Technometrics*, 21(2):215–223, 1979.

T. Hastie, R. J. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer, New York, 2001.

R. R. Hocking. A Biometrics invited paper. The analysis and selection of variables in linear regression. *Biometrics*, 32(1):1–49, March 1976.

A. E. Hoerl and R. W. Kennard. Ridge-regression: Biased estimation for nonorthogonal problems. *Technometrics*, 8:27–51, 1970.

A. Hoeskuldsson. PLS regression methods. *Journal of Chemometrics*, 2(3):211–228, 1988.

H. Hotelling. Analysis of a complex of statistical variables into principal components. *Journal of Educational Psychology*, 24(6):417–441, September 1933.

P. J. Huber. *Robust Statistics*. Wiley, New York, 1981.

M. Hubert and K. Vanden Branden. Robust methods for partial least squares regression. *Journal of Chemometrics*, 17:537–549, 2003.

O. Ivanciuc. Applications of support vector machines in chemistry. *Rev. Computat. Chem.*, 23:291–400, 2007.

P. A. Jansson. Neural networks: An overview. *Annals of Chemometrics*, 63:357–362, 1991.

I. T. Jolliffe. A note on the use of principal components in regression. *Journal of the Royal Statistical Society, Series C (Applied Statistics)*, 31(3):300–303, 1982.

Friedrich Leisch. Sweave: Dynamic generation of statistical reports using literate data analysis. In Wolfgang Härdle and Bernd Rönz, editors, *Compstat 2002 — Proceedings in Computational Statistics*, pages 575–580. Physica Verlag, Heidelberg, 2002. URL `http://www.stat.uni-muenchen.de/~leisch/Sweave`. ISBN 3-7908-1517-9.

Friedrich Leisch. Sweave, part II: Package vignettes. *R News*, 3(2):21–24, October 2003. URL `http://CRAN.R-project.org/doc/Rnews/`.

B. Liebmann, A. Friedl, and K. Varmuza. Determination of glucose and ethanol in bioethanol production by near infrared spectroscopy and chemometrics. *Anal. Chim. Acta*, 642:171–178, 2009.

M. Otto. *Chemometrics - Statistics and Computer Application in Analytical Chemistry*. Wiley-VCH, Weinheim, Germany, 2007.

K. Pearson. On lines and planes of closest fit to systems of points in space. *Philosophical Magazine*, 2(6):559–572, November 1901.

R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2006. URL `http://www.R-project.org`. ISBN 3-900051-07-0.

R. J. Schalkoff. *Artificial Neural Networks*. McGraw-Hill, New York, 1997.

G. E. Schwarz. Estimating the dimension of a model. *Annals of Statistics*, 6(2):461–464, 1978.

S. Serneels, C. Croux, P. Filzmoser, and P. J. Van Espen. Partial robust M-regression. *Chemometrics and Intelligent Laboratory Systems*, 79(1-2):55–64, 2005.

R. Tibshirani. Regression shrinkage and selection via the lasso. *J. Royal Statistical Soc. B.*, 58:267–288, 1996.

J. Trygg and H. Wold. Orthogonal projections to latent structures (O-PLS). *Journal of Chemometrics*, 16:119–128, 2002.

K. Varmuza and P. Filzmoser. *Introduction to Multivariate Statistical Analysis in Chemometrics*. CRC Press, Boca Raton, Florida, 2009.

W. N. Venables and D. M. Smith. *An Introduction to R*, 2002. URL `http://cran.rproject.org/doc/manuals/R-intro.pdf`.

I. N. Wakeling and H. J. H. Macfie. A robust PLS procedure. *Journal of Chemometrics*, 6: 189–198, 1992.

H. Wold. Estimation of principal components and related models by iterative least squares. In P.R. Krishnaiah, editor, *Multivariate Analysis*, pages 391–420. Academic Press, New York, 1966.

S. Wold. PLS-regression: a basic tool of chemometrics. *Chemometrics and Intelligent Laboratory Systems*, 58:109–130, 2001.