Getting started with mdatools for R

Sergey Kucheryavskiy

July 21, 2023

Contents

Introduction

This is a user guide for *mdatools* — R package for preprocessing, exploring and analysis of multivariate data. The package provides methods mostly common for Chemometrics. The general idea of the package is to collect the popular chemometric methods and give a similar "user interface" for applying the methods to different datasets. So, if a user knows how to make a model and visualize results for one method, they can easily do this for the other methods as well. Usually I update the tutorial when a new version of the package is released, you can track main changes here.

The current version of tutorial is compatible with version *0.14.1* of the package.

All methods implemented in the package were tested using well-known datasets. However, there still could be some bugs, in this case please report to svkucheryavski@gmail.com or use Issues tool at GitHub. You are also very welcome to share your comments and suggestions about the package functionality.

If you want to cite the package, please use the following:

Sergey Kucheryavskiy, *mdatools – R package for chemometrics*, Chemometrics and Intelligent Laboratory Systems, Volume 198, 2020 (DOI: 10.1016/j.chemolab.2020.103937).

What is new

21.07.2023

- Added information about new feature in 0.14.1 — `cv.scope` parameter. See Model and results part of PLS chapter, just scroll down to cross-validation section.
- Small improvements to various parts of this tutorial.

31.03.2023

- Added section about normalization to Preprocessing chapter.
- Added section about missing values replacement to Preprocessing chapter.
- Improved description of validation procedure in PCA, PLS and other methods, including the use of Procrustes cross-validation.
- Small improvements in selected parts of text.

09.11.2022

Improved description of iPLS section, including new features appeared in v. 0.13.1.

03.03.2022

- Added section explaining how to use `pcv.nareplace()` method
- Added separate section for Procrustes cross-validation.

14.07.2022

Added description of new features appeared in v. 0.13.0 of the package, including new plotting method `plotRMSERatio()` for PLS models and new cross-validation possibilities. Validation of PLS model text has been extended and made as separate section.

12.09.2021

Added description of new features appeared in v. 0.12.0 of the package, including:

- new preprocessing methods `prep.transform()`, `prep.varsel()`.
- improvements to `prep.savgol()` and `prep.norm()`.
- a possibility to combine several preprocessing method together and apply at once.

In addition to that, text for Datasets chapter has been revised to provide more clear guides while the part about preprocessing was detached and now is available as separate chapter where you can find all details. Description of ALS baseline correction was moved to the section Correction of baseline.

Overview

The first version of this package was created in 2012 for an introductory PhD course on Chemometrics given at Department of Chemistry and Bioscience, Aalborg University. Quickly I found out that using R for this course (with all advantages it gives) needs a lot of routine work from students, since most of them were also beginners in R. Of course it is very good for understanding when students get to know e.g. how to calculate explained variance or residuals in PCA manually or make corresponding plots and so on, but for the introductory course these things (as well as numerous typos and small mistakes in a code) take too much time, which can be spent for explaining methods and proper interpretation of results.

This is actually also true for everyday use of these methods, most of the routines can be written once and simply re-used with various options. So it was decided to write a package where most widely used chemometric methods for multivariate data analysis are implemented and which also gives a quick and easy-to-use access to results, produced by these methods. First of all via numerous plots.

Here how it works. Say, we need to make a PCA model for data matrix `x` with autoscaling. Then make an overview of most important plots and investigate scores and loadings for first three components. The `mdatools` solution will be:

```
# make a model for autoscaled data with maximum possible number of components
m = pca(x, scale = TRUE)

# show explained variance plot
plotVariance(m)

# select optimal number of components (say, 4) for correct calculation of residuals
m = selectCompNum(m, 4)

# show plots for model overview
plot(m)

# show scores plot for PC1 and PC3
plotScores(m, c(1, 3))

# show loadings plot for the same components
plotLoadings(m, c(1, 3))

# show the loadings as a set of bar plots
plotLoadings(m, c(1, 3), type = "h")
```

Fairly simple, is not it? The other "routine", which have been taken into account is validation — any model can be cross-validated or validated with a test set. The model object will contain the validation results, which will also appear on all model plots, etc. See the next chapters for details.

What mdatools can do?

The package includes classes and functions for analysis, preprocessing and plotting data and results. So far the following methods for analysis are implemented:

- Principal Component Analysis (PCA)
- Soft Independent Modelling of Class Analogy (SIMCA), including data driven approach (DD-SIMCA)
- Partial Least Squares regression (PLS) with calculation of VIP scores and Selectivity ratio
- Partial Least Squares Discriminant Analysis (PLS-DA)
- Randomization test for PLS regression models
- Interval PLS for variable selection
- Multivariate curve resolution using the purity approach
- Multivariate curve resolution using the constrained alternating least squares

Preprocessing methods include:

- Mean centering, standardization and autoscaling
- Savitzky-Golay filter for smoothing and derivatives
- Standard Normal Variate for removing scatter and global intensity effect from spectral data
- Mutliplicative Scatter Correction for the same issue
- Normalization of spectra to unit area, unit length, unit sum, unit area under given range.
- Baseline correction with asymmetric least squares
- Kubelka-Munk transformation
- Element wise transformations (`log`, `sqrt`, power, etc.)

Besides that, some extensions for the basic R plotting functionality have been also implemented and allow to do the following:

- Color grouping of objects with automatic color legend bar.
- Plot for several groups of objects with automatically calculated axes limits and plot legend.
- Three built-in color schemes — one is based on Colorbrewer and the other two are jet and grayscale.
- Very easy-to-use possibility to apply any user defined color scheme.
- Possibility to show horizontal and vertical lines on the plot with automatically adjusted axes limits.
- Possibility to extend plotting functionality by using some attributes for datasets.

See `?mdatools` and next chapters for more details.

How to install and update

The package is available on CRAN, to install it just use:

```
install.packages("mdatools")
```

This is the recommended way to install the package. If you have installed it already and just want to update to the newest version, use:

```
update.packages("mdatools")
```

If you want to install it directly from GitHub, the easiest way is to install the `devtools` package first and then run the following command in R:

```
devtools::install_github("svkucheryavski/mdatools")
```

Datasets and plots

The package uses standard representation of the data in R: data frames, matrices and vectors. However, there are several additional methods and attributes, which makes the use of the datasets a bit more more efficient. There is also a support for images. But if you are used to simple datasets and standard procedures and do not want any complications, you can simply skip this chapter.

The package also uses its own set of plotting tools, which is a sort of an add-on for the R basic plotting system, extending its possibilities. From this point of view, learning how these tools work will simplify understanding of model plots a lot. The main improvements comparing to the basic plotting system are:

1. Much easier way to make plots with groups of objects (points, lines, bars, etc.)
2. Much easier way of adding legend to the group plots.
3. Much easier way of adding labels to data points, bars, etc.
4. Automatic axis limits when a plot contains several groups of objects.
5. Possibility to color points and lines according to values of a specific numerical variable of a factor.
6. Several built in color pallettes and an easy way to use user specific set of colors.
7. Much more!

This chapter explains most of the details.

Attributes and factors

This section tells how to extend the functionality of the package by using attributes assigned to datasets and how methods implemented in the package deal with factors.

Attributes for plots

The plot attributes will be explained very briefly here, you can find much more details in the next two sections. The idea is to provide some names and values to the data, which can be used later e.g. for making labels and titles on the plots. When dataset is used to create a model (e.g. PCA) all results representing objects (e.g. scores, distances, etc.) will inherit the row specific attributes and all results related to variables (e.g. loadings) will inherit column specific attributes.

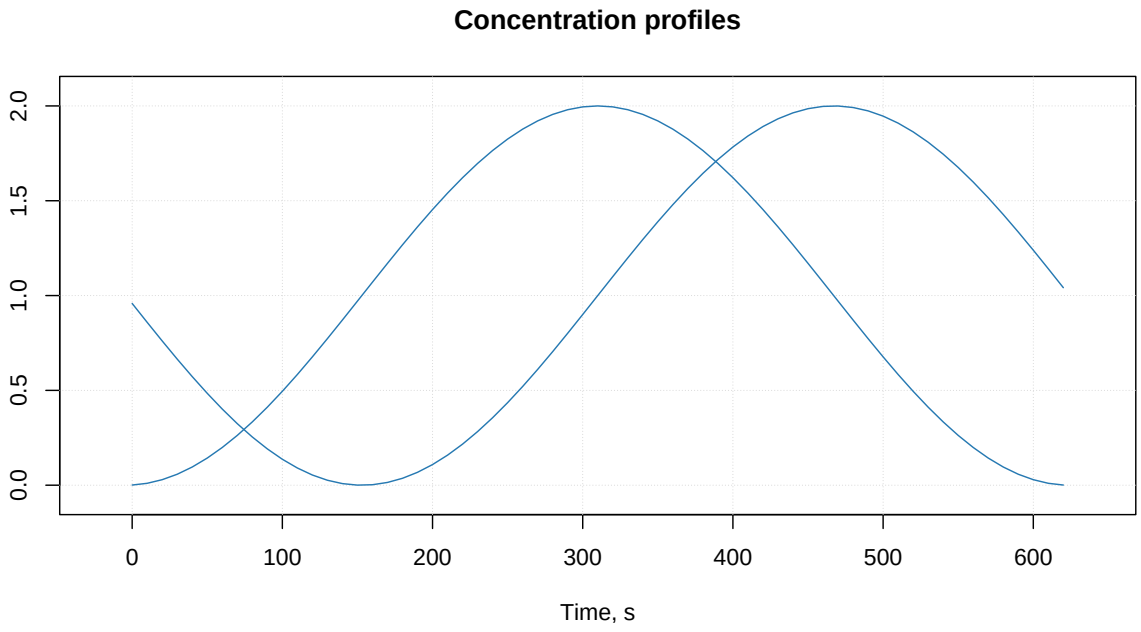
The main attributes for plots are following:

Attribute	Meaning
`name`	name of a dataset (used for plot parameter `main`).
`xaxis.name`	if variables/data columns represent the same property, e.g. wavelength, the property name can be defined by this attribute (used for plot axes labels).
`yaxis.name`	if objects/data rows represent the same property, e.g. reaction time, the property name can be defined by this attribute (used for plot axes labels).
`xaxis.values`	a vector of values, which correspond to the columns (e.g. for spectroscopic data it can be a vector with wavelength or wavenumbers).

Attribute	Meaning
`yaxis.values`	a vector of values, which correspond to the rows (e.g. for kinetic data it can be a vector with time or temperature values).

Here is a very simple example.

```
# generate data as matrix with two rows and 63 columns
t = -31:31
d = rbind(sin(t/10), cos(t/10)) + 1

# add name of the dataset
attr(d, "name") = "Concentration profiles"

# add name and values for a property representing the columns (time of reaction)
# we make values for x-axis to span from 0 to 620 seconds
attr(d, "xaxis.name") = "Time, s"
attr(d, "xaxis.values") = (t + 31) * 10

# make a line plot for the data
mdaplot(d, type = "l")
```



As you can notice, both the label and the ticks for x-axis correspond to the attributes we defined. As well as the main title of the plot. As we mentioned above these attributes will be inherited and e.g. PCA loadings will have the same x-axis if you decide to make a line plot for the loadings.

See more details in the section about plots.

Special methods for data transformations

Since data objects in R loose all user specified attributes when e.g. we transpose them or taking a subset, it was decided to write several methods, which would process attributes correctly. They also adjust indices of excluded rows and columns when user takes a subset or merge two data objects together. When data

matrix is transposed the corresponding method will swap the x- and y- attributes. All methods with a brief description are listed in the table below.

Method	Description
`mda.show(data)`	Show data object without excluded elements
`mda.t(data)`	Transpose data object
`mda.cbind(data1, data2, ...)`	Merge several datasets by columns
`mda.rbind(data1, data2, ...)`	Merge several datasets by rows
`mda.subset(data1, subset, select)`	Take a subset of data object (`subset` is numeric indices, names or logical values for rows, `select` — the same for columns)
`attrs = mda.getattr(data)`	Return all user specific attributes from an object
`data = mda.setattr(data, attrs)`	Assign user specific attributes to an object

To avoid any problems with arguments use these functions instead of the traditional ones, e.g. `mda.subset()` instead of `subset()`, when available.

Data frames with factors

All methods, implemented in the package, work with matrices, therefore, if a user provides data values as data frame, it is converted to matrix. It is also possible to provide data frames with one or several factor columns. In this case all factors will be converted to dummy variables with values 0 and 1. You can also do it manually, by using function `prep.df2mat()` as this is shown in an example below.

Let us first create a simple data with a factor column.

```
h = c(180, 175, 165, 190, 188)
c = as.factor(c("Gray", "Green", "Gray", "Green", "Blue"))
d = data.frame(Height = h, Eye.color = c)
show(d)
```

```
##   Height Eye.color
## 1    180      Gray
## 2    175     Green
## 3    165      Gray
## 4    190     Green
## 5    188      Blue
```

And this is the result of converting it to a matrix.

```
d.mat = mda.df2mat(d)
show(d.mat)
```

```
##      Height Blue Gray
## [1,]    180    0    1
## [2,]    175    0    0
## [3,]    165    0    1
## [4,]    190    0    0
## [5,]    188    1    0
```

The number of dummy variables by default is the number of levels minus one. You can change this by using argument `full = TRUE` is it is shown in the example below.

```
d.mat = mda.df2mat(d, full = TRUE)
show(d.mat)
```

```
##      Height Blue Gray Green
## [1,]    180    0    1     0
```

```
## [2,]    175    0    0     1
## [3,]    165    0    1     0
## [4,]    190    0    0     1
## [5,]    188    1    0     0
```

It is important to have level labels in all factor columns of the same data frame unique, as they are used for names of the dummy variables (e.g. you should not have two factors with the same level name). If a factor is hidden it will be just converted to numeric values and remain excluded from modelling.

Excluding rows and columns

Sometimes it is necessary to hide or to exclude a particular row or a particular column from a data, so they will not be shown on plots and will not be taken into account in modelling, but without removing them physically. So the excluded rows and columns are still there but are not treated in a usual way by the *mdatools* methods. In *mdatools* it is possible by using the following functions:

Function	Description
`mda.exclrows(x, ind)`	Exclude (hide) all rows specified by variable `ind`, which can be a vector with rows indices, names or a logical vector.
`mda.exclcols(x, ind)`	Exclude (hide) all columns specified by variable `ind`, which can be a vector with columns indices, names or a logical vector.

The mechanism is very simple, the indices of the rows or columns, which must be excluded, are saved into special attributes, which then is recognized by all methods implemented in `mdatools`. Standard R functions will ignore the attributes.

Here is a simple example. Let's create a dataset first (it can be either matrix or a data frame)

```
Height = c(180, 175, 165, 190, 188)
Weight = c(78, 79, 60, 99, 80)
Shoesize = c(44, 39, 35, 45, 44)
d = cbind(Height, Weight, Shoesize)
rownames(d) = paste0("O", 1:5)
show(d)
```

```
##    Height Weight Shoesize
## O1    180     78       44
## O2    175     79       39
## O3    165     60       35
## O4    190     99       45
## O5    188     80       44
```

Now let's exclude rows 3 and 4 and then column with name "Weight" from the data.

```
d = mda.exclrows(d, 3:4)
d = mda.exclcols(d, "Weight")
show(d)
```

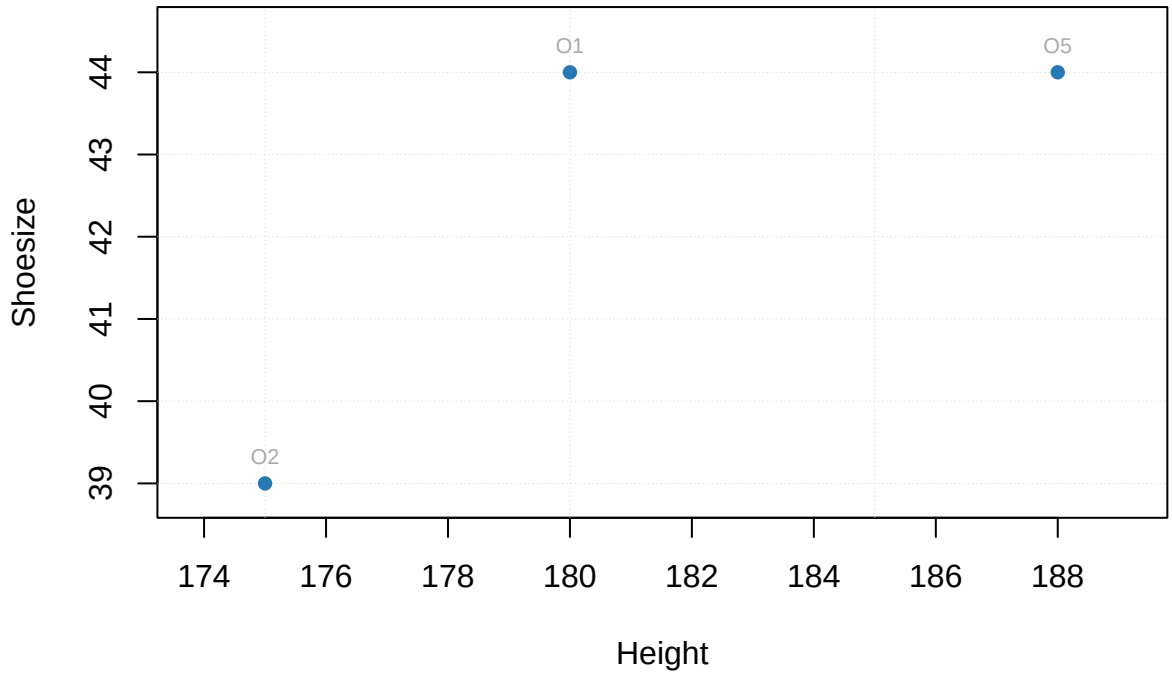
```
##    Height Weight Shoesize
## O1    180     78       44
## O2    175     79       39
## O3    165     60       35
## O4    190     99       45
## O5    188     80       44
## attr(,"exclrows")
```

```
## [1] 3 4
## attr(,"exclcols")
## [1] 2
```

As you can see, all the values are still there, but there are two new attributes, `exclcols` and `exclrows`. To avoid any issues do not change the values of the attributes manually, always use the functions above.

Now if you make a plot using function `mdaplot()` you will see only three points by default, because the other two a hidden. Also plot will be made for columns 1 and 3, because the second column (Weight) was also hidden. See the code and the result below:

```
mdaplot(d, show.labels = TRUE)
```



You can include the hidden columns and rows back by using `mda.inclcols()` and `mda.inclrows()`.

Simple plots

As it was already mentioned, *mdatools* has its own functions for plotting with several extra options not available in basic plot tools. These functions are used to make all plots in the models and results (e.g. scores, loadings, predictions, etc.) therefore it can be useful to spend some time and learn the new features (e.g. coloring data points with a vector of values or using manual ticks for axes). But if you are going to make all plots manually (e.g. using *ggplot2*) you can skip this and the next sections.

In this section we will look at how to make simple plots from your data objects. Simple plots are scatter (`type = "p"`), density-scatter (`type = "d"`), line (`type = "l"`), line-scatter (`type = "b"`), bar (`type = "h"`) or errorbar (`type = "e"`) plots made for a one set of objects. All plots can be created using the same method `mdaplot()` by providing a whole dataset as a main argument. Depending on a plot type, the method "treats" the data values differently.

This table below contains a list of parameters for `mdaplot()`, which are not available for traditional R plots. In this section we will describe most of the details using simple examples.

Parameter	Description
`cgroup`	a vector of values (same as number of rows in data) used to colorize plot objects with a color gradient.
`colmap`	color map for the color gradient (possible values are `'default'`, `'gray'` or a vector with colors).
`show.colorbar`	when color grouping is used, `mdaplot()` shows a color bar legend, this parameter allows to turn it off.
`show.labels`	logical parameter showing labels beside plot objects (points, lines, etc). Size and color of labels can be adjusted using parameters `lab.cex` and `lab.col`.
`labels`	parameter telling what to use as labels (by default row names, but can also be indices or manual values).
`lab.col`	color for the labels.
`lab.cex`	font size for the labels (as a scale factor).
`xticks`	vector with numeric values to show the x-axis ticks at.
`yticks`	vector with numeric values to show the y-axis ticks at.
`xticklabels`	vector with labels (numbers or text) for the x-ticks.
`yticklabels`	vector with labels (numbers or text) for the y-ticks.
`xlas`	an integer between 0 and 3 telling at which angle the x-tick labels have to be shown.
`ylas`	an integer between 0 and 3 telling at which angle the y-tick labels have to be shown.
`show.axes`	logical, if `TRUE`, function will make a new plot, if `FALSE`, add the plot objects to a previous one.
`show.lines`	a vector with two numbers — position of horizontal and vertical lines on a plot (e.g. coordinate axes).
`show.grid`	logical, show or not a grid. It places grid behind the plot object in contrast to conventional `grid()` method. Use `grid.lwd` and `grid.col` parameters to adjust the grid look.
`show.excluded`	logical, show or not points or lines corresponded to the excluded rows.
`opacity`	opacity of colors in range $0 \ldots 1$ (applied to all colors of current plot).

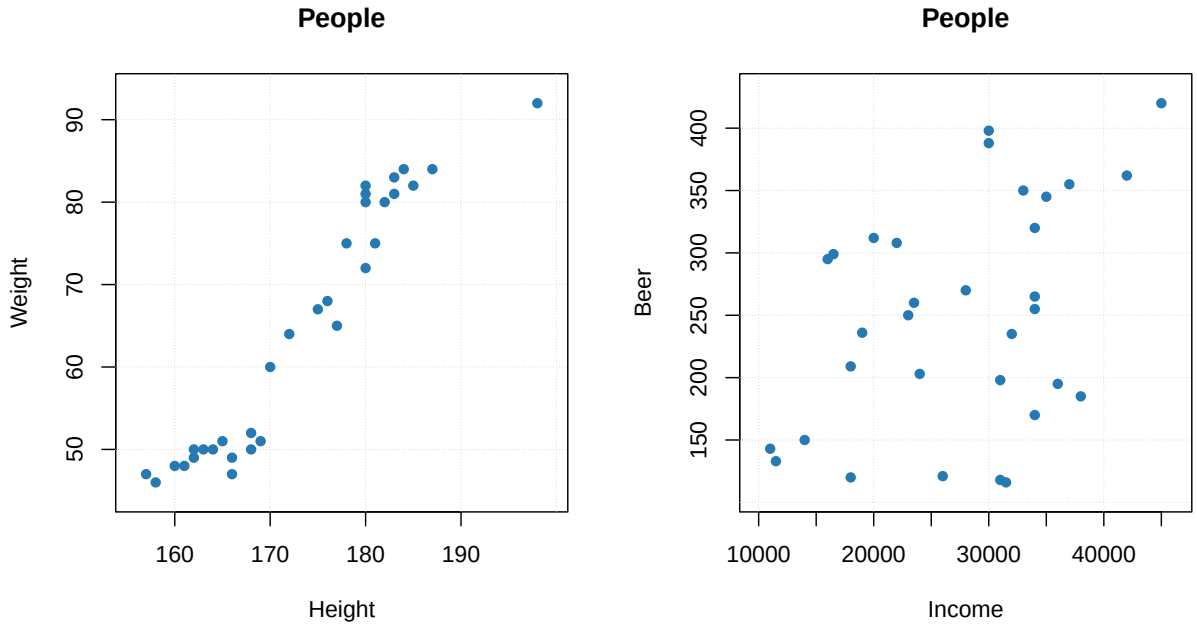
Scatter plots

We will use `people` dataset for illustration how scatter plots work (see `?people` for details).

```
data(people)
attr(people, "name") = "People"
attr(people, "xaxis.name") = "Variables"
attr(people, "yaxis.name") = "Persons"
```

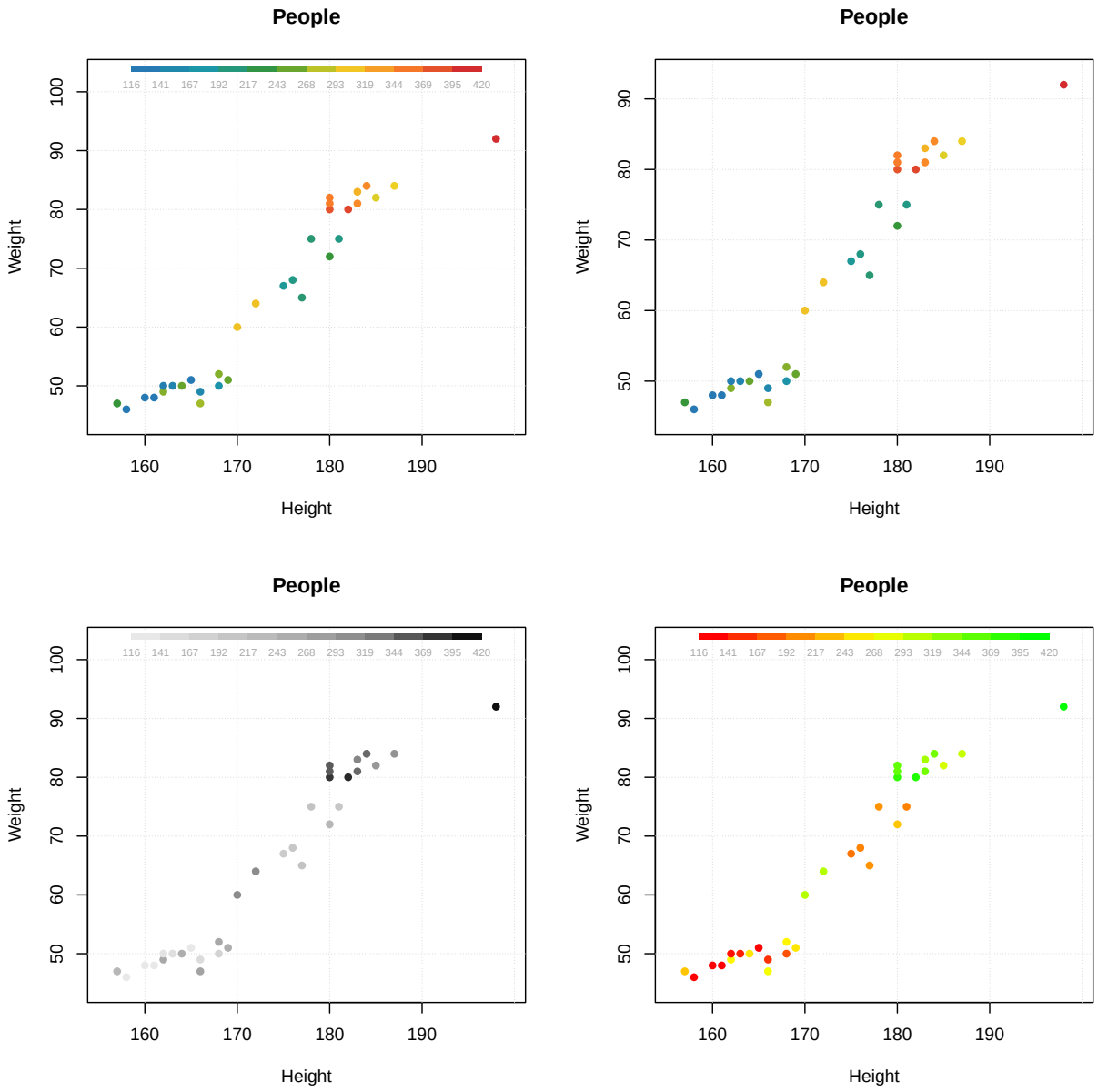
For scatter plots the method takes first two columns of a dataset as x and y vectors. If only one column is available `mdaplot()` uses it for y-values and generate x-values as an index for each value.

```
par(mfrow = c(1, 2))

# show plot for the whole dataset (columns 1 and 2 will be taken)
mdaplot(people, type = "p")

# subset the dataset and keep only columns 6 and 7 and then make a plot
mdaplot(mda.subset(people, select = c(6, 7)), type = "p")
```



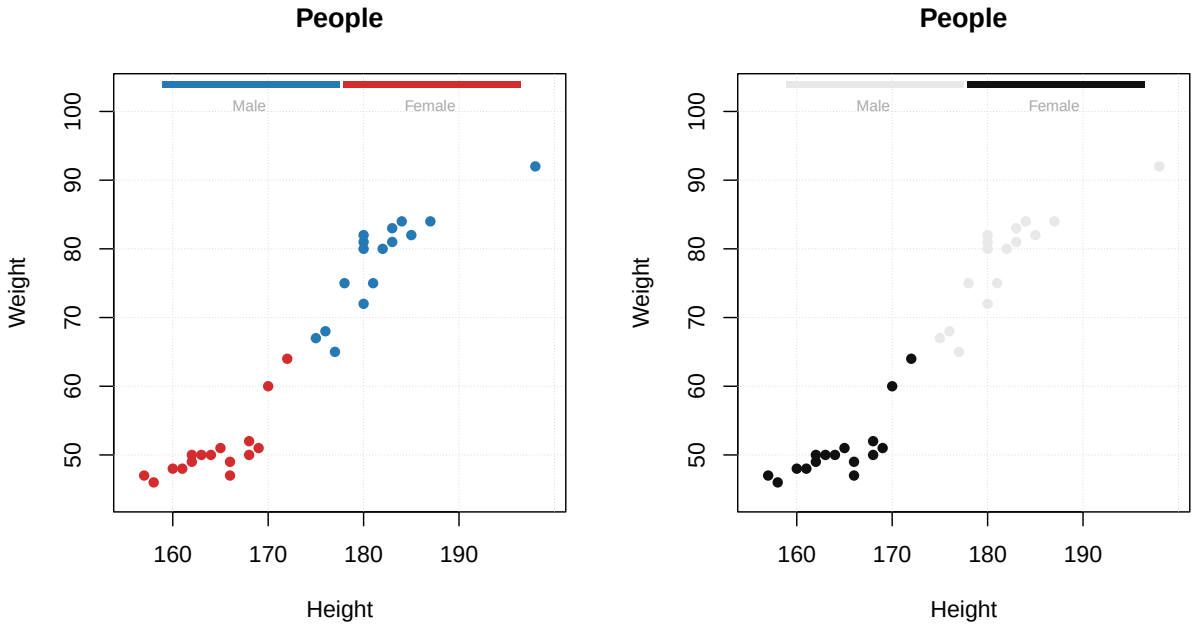
All parameters, available for the standard `points()` method will work with `mdaplot()` as well. Besides that, you can colorize points according to some values using a color gradient. By default, the gradient is generated using one of the diverging color schemes from colorbrewer2.org, but this can be changed using parameter `colmap` as it is shown below.

```
par(mfrow = c(2, 2))

# show Height vs Weight and color points by the Beer consumption
mdaplot(people, type = "p", cgroup = people[, "Beer"])

# do the same but do not show colorbar
mdaplot(people, type = "p", cgroup = people[, "Beer"], show.colorbar = FALSE)

# do the same but use grayscale color map
mdaplot(people, type = "p", cgroup = people[, "Beer"], colmap = "gray")

# do the same but using colormap with gradients between red, yellow and green colors
mdaplot(people, type = "p", cgroup = people[, "Beer"], colmap = c("red", "yellow", "green"))
```



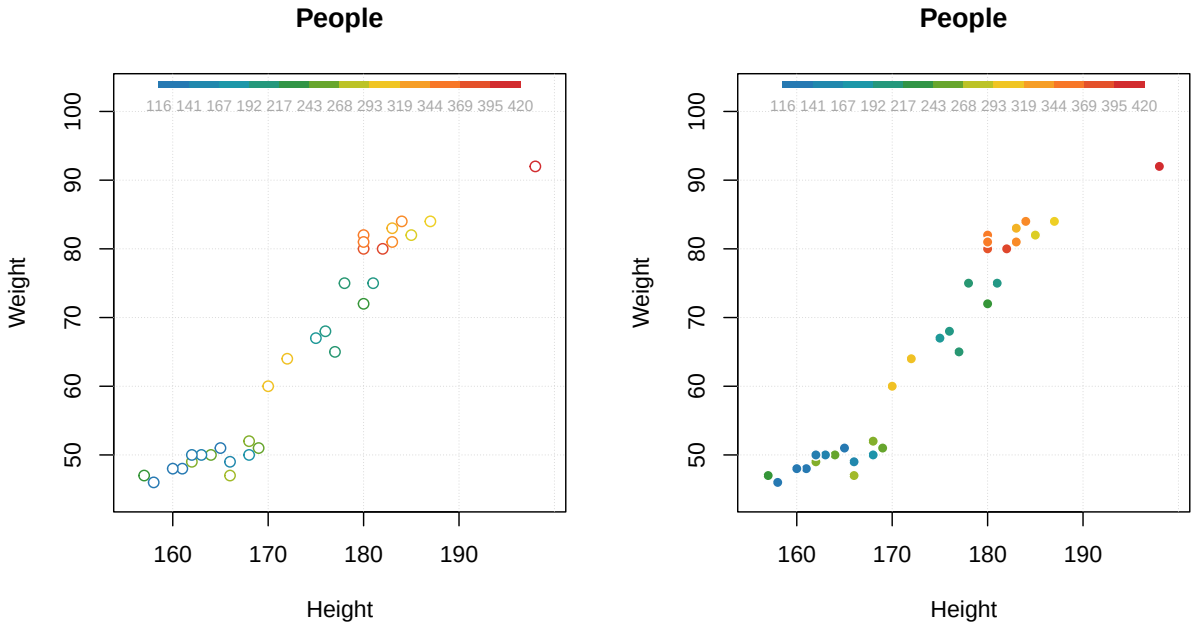
If the vector with values for color grouping is a factor, level labels will be shown on a colorbar legend and there will be a small margin between bars.

```
# make a factor using values of variable Sex and define labels for the factor levels
g = factor(people[, "Sex"], labels = c("Male", "Female"))

par(mfrow = c(1, 2))
mdaplot(people, type = "p", cgroup = g)
mdaplot(people, type = "p", cgroup = g, colmap = "gray")
```



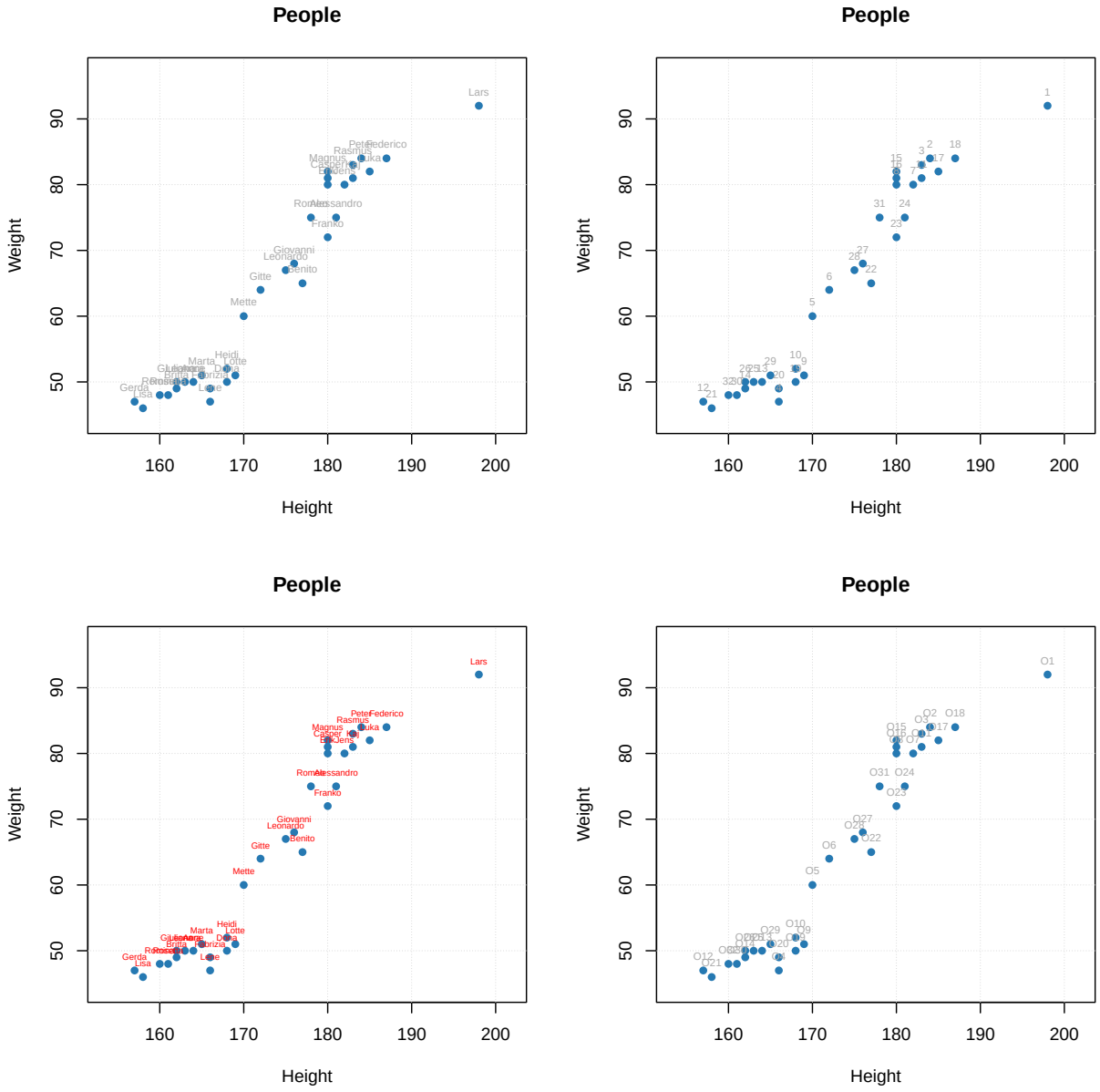
If you use point characters from 21 to 25 (the ones which allow to specify both color of border and background of the marker symbol), then the `cgroup` is applied to color of the borders of the symbols by default. If you want to apply it to background color, use logical parameter `pch.colinv` for that as shown below:

```
par(mfrow = c(1, 2))

# default way - color grouping is used for borders and "bg" for background
mdaplot(people, type = "p", cgroup = people[, "Beer"], pch = 21, bg = "white")

# inverse - color grouping is used for background and "bg" for border
mdaplot(people, type = "p", cgroup = people[, "Beer"], pch = 21, bg = "white", pch.colinv = TRUE)
```



Another useful option is adding labels to the data points. By default row names will be taken for the labels but you can specify a parameter `labels`, which can be either a text (`"names"` or `"indices"`) or a vector with values to show as labels. Color and size of the labels can be adjusted.

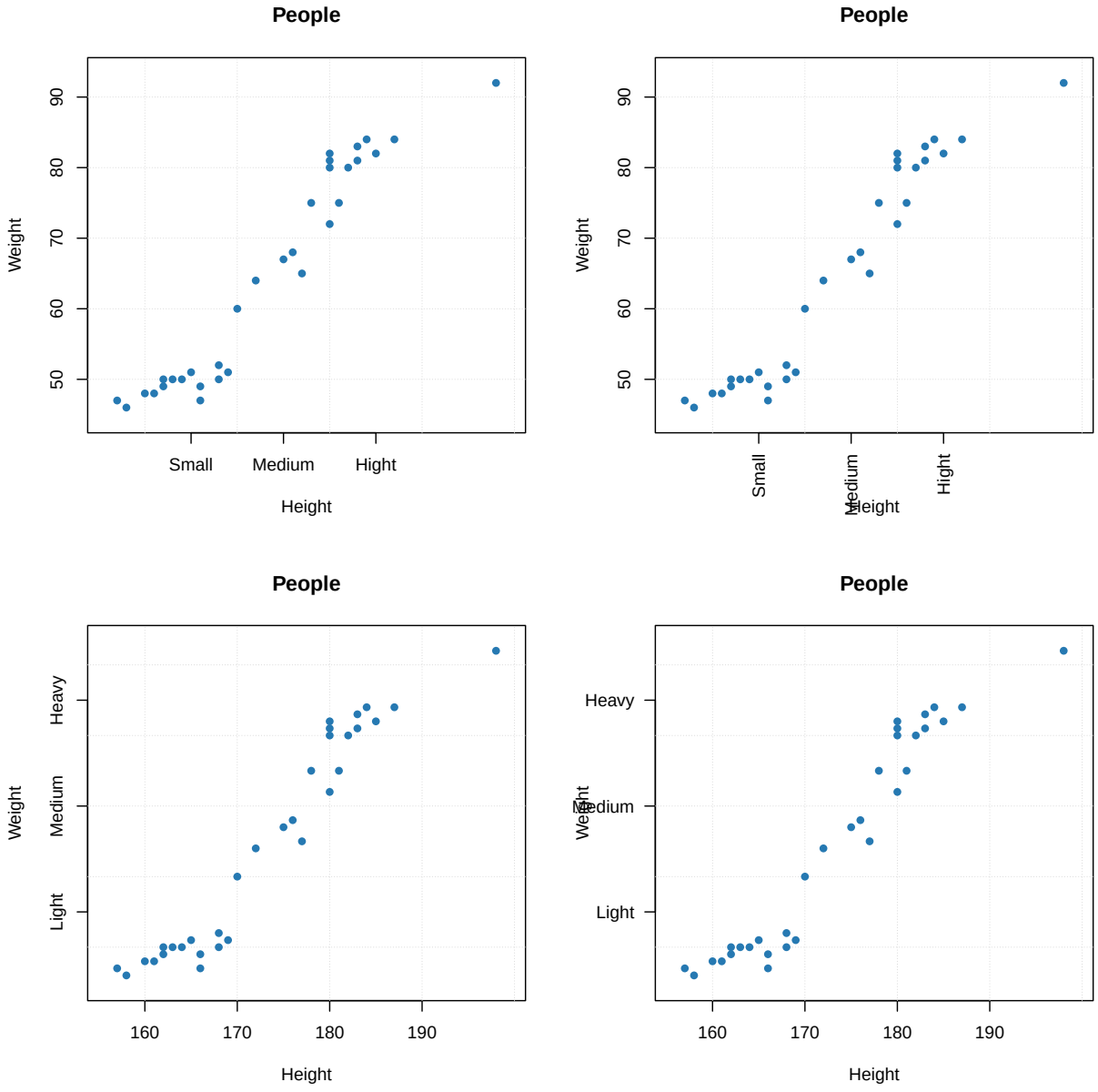
```
par(mfrow = c(2, 2))

# by default row names will be used as labels
mdaplot(people, type = "p", show.labels = TRUE)

# here we tell to use indices as labels instead
mdaplot(people, type = "p", show.labels = TRUE, labels = "indices")

# here we use names again but change color and size of the labels
mdaplot(people, type = "p", show.labels = TRUE, labels = "names", lab.col = "red", lab.cex = 0.5)

# finally we provide a vector with manual values to be used as the labels
mdaplot(people, type = "p", show.labels = TRUE, labels = paste0("O", seq_len(nrow(people))))
```



You can also manually specify axis ticks and tick labels. The labels can be rotated using parameters `xlas` and `ylas`, see the examples below. It is important though, that if you provide manual values for the tick labels you must also provide a vector of values with the positions the labels should be shown at. And, of course, the two vectors must have the same number of values.

```
par(mfrow = c(2, 2))

# manual values and tick labels for the x-axis
mdaplot(people, xticks = c(165, 175, 185), xticklabels = c("Small", "Medium", "Hight"))

# same but with rotation of the tick labels
mdaplot(people, xticks = c(165, 175, 185), xticklabels = c("Small", "Medium", "Hight"), xlas = 2)

# manual values and tick labels for the y-axis
```

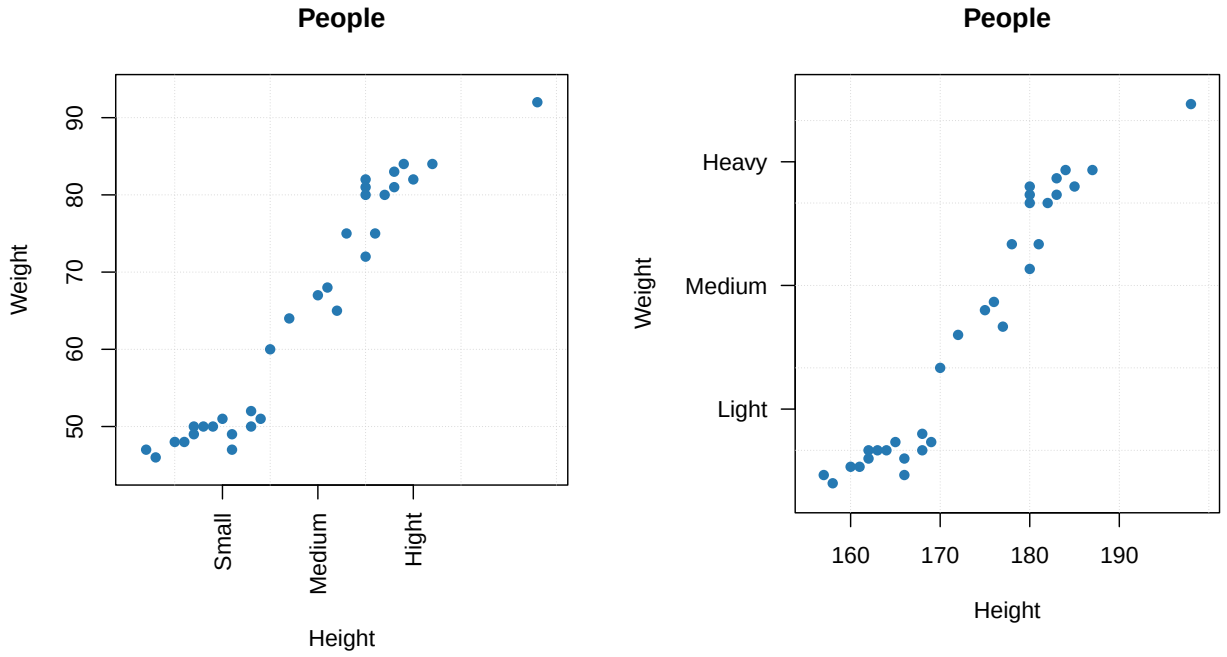
```
mdaplot(people, yticks = c(55, 70, 85), yticklabels = c("Light", "Medium", "Heavy"))

# same but with rotation of the tick labels
mdaplot(people, yticks = c(55, 70, 85), yticklabels = c("Light", "Medium", "Heavy"), ylas = 2)
```



If both axis labels and rotated axis ticks have to be shown, you can adjust plot margins and position of the label using `par()` function and `mtext()` for positioning axis label manually.

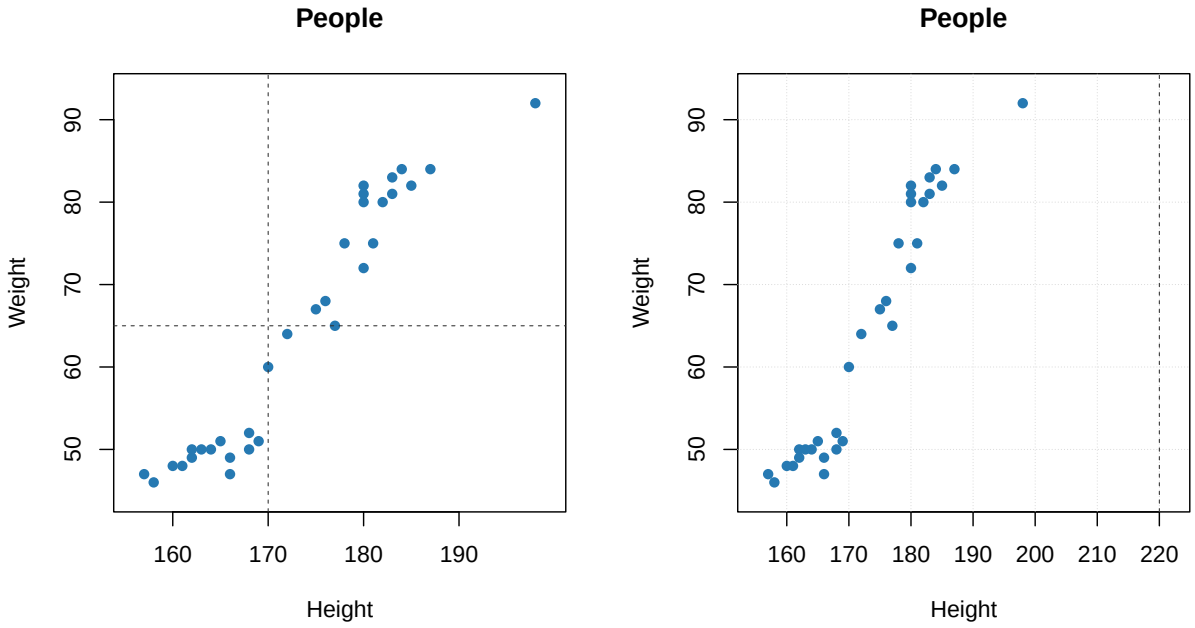
```
par(mfrow = c(1, 2))

# change margin for bottom part
par(mar = c(6, 4, 4, 2) + 0.1)
mdaplot(people, xticks = c(165, 175, 185), xticklabels = c("Small", "Medium", "Hight"),
        xlas = 2, xlab = "")
```

```
mtext("Height", side = 1, line = 5)

# change margin for left part
par(mar = c(5, 6, 4, 1) + 0.1)
mdaplot(people, yticks = c(55, 70, 85), yticklabels = c("Light", "Medium", "Heavy"),
        ylas = 2, ylab = "")
mtext("Weight", side = 2, line = 5)
```



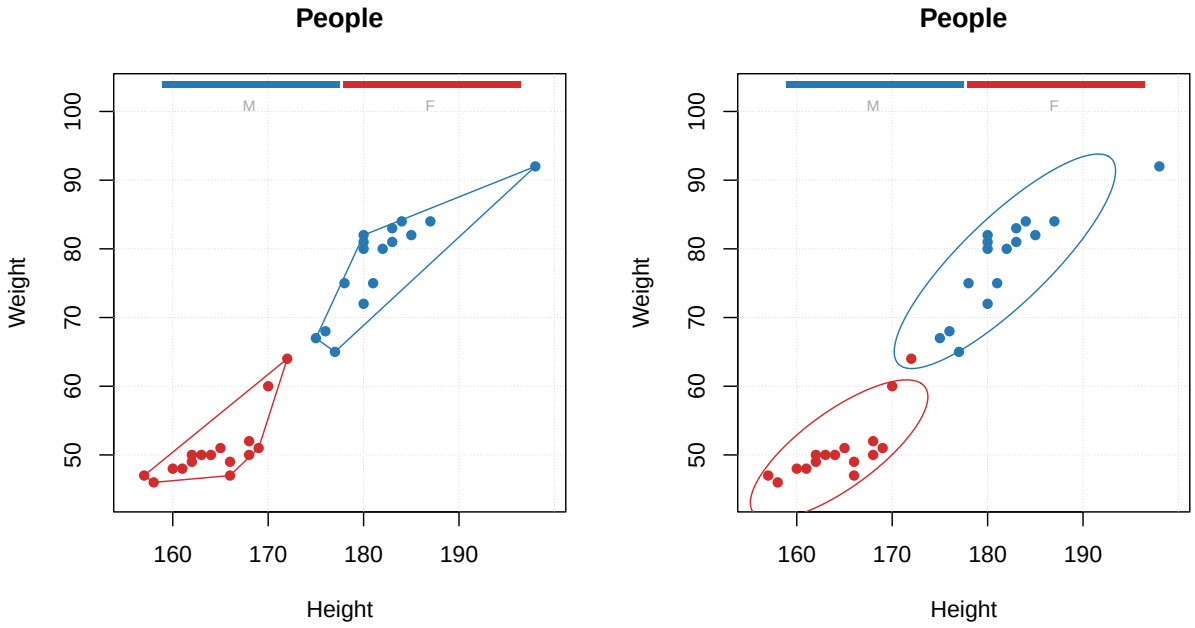
There is also a couple of other parameters, allowing to show/hide grid as well as show horizontal and vertical lines on the plot (axes limits will be adjusted correspondingly).

```
par(mfrow = c(1, 2))
mdaplot(people, show.grid = FALSE, show.lines = c(170, 65))
mdaplot(people, show.lines = c(220, NA))
```



Function `mdaplot()` can also return plot series data, which can be used for extra options. For example, in case of scatter plot you can add confidence ellipse or convex hull for data points. To do this, points must be color grouped by a factor as shown below. For confidence ellipse you can specify the confidence level (default 0.95).

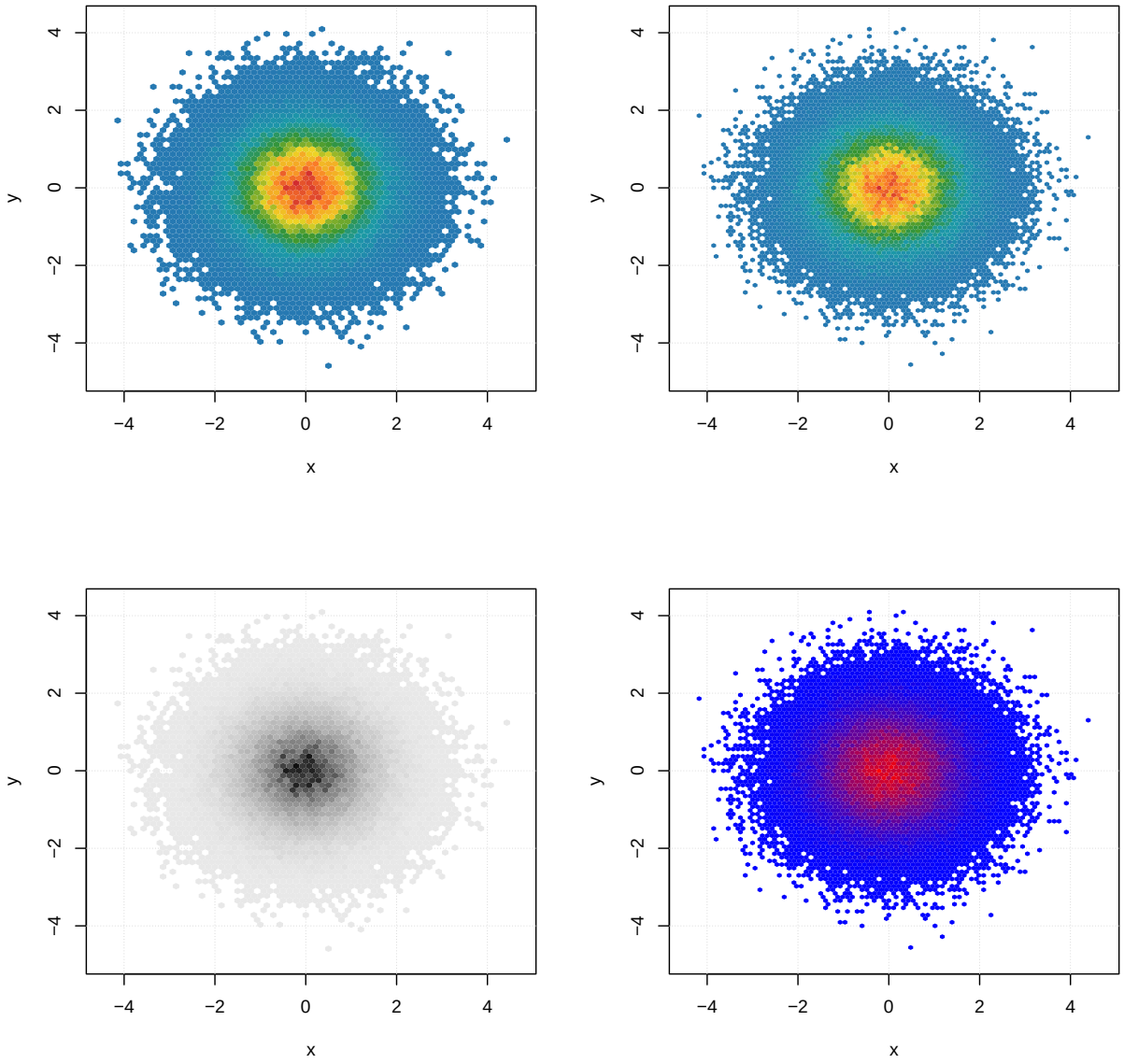
```
# define a factor using values of variable Sex and simple labels
g = factor(people[, "Sex"], labels = c("M", "F"))

par(mfrow = c(1, 2))

# make a scatter plot grouping points by the factor and then show convex hull for each group
p = mdaplot(people, cgroup = g)
plotConvexHull(p)

# make a scatter plot grouping points by the factor and then show 90% confidence intervals
p = mdaplot(people, cgroup = g)
plotConfidenceEllipse(p, conf.level = 0.90)
```



In case when number of data points is large (e.g. when dealing with images, where every pixel is a data point), using density plot is a good alternative to conventional scatter plots. The plot does not show all data points but instead split the whole plot space into small hexagonal regions and use color gradient for illustration a density of the points in each region. This approach is known as *hexagonal binning*. To create a density plot simply use `type="d"`. You can also specify color map and number of bins along each axes (`nbins`).

The code below show an example of using density plots for 100000 data points with x and y values taken from normally distributed population.

```
x = rnorm(100000)
y = rnorm(100000)
d = cbind(x, y)

par(mfrow = c(2, 2))
mdaplot(d, type = "d")
mdaplot(d, type = "d", nbins = 80)
mdaplot(d, type = "d", colmap = "gray")
mdaplot(d, type = "d", nbins = 80, colmap = c("blue", "red"))
```



Line plots

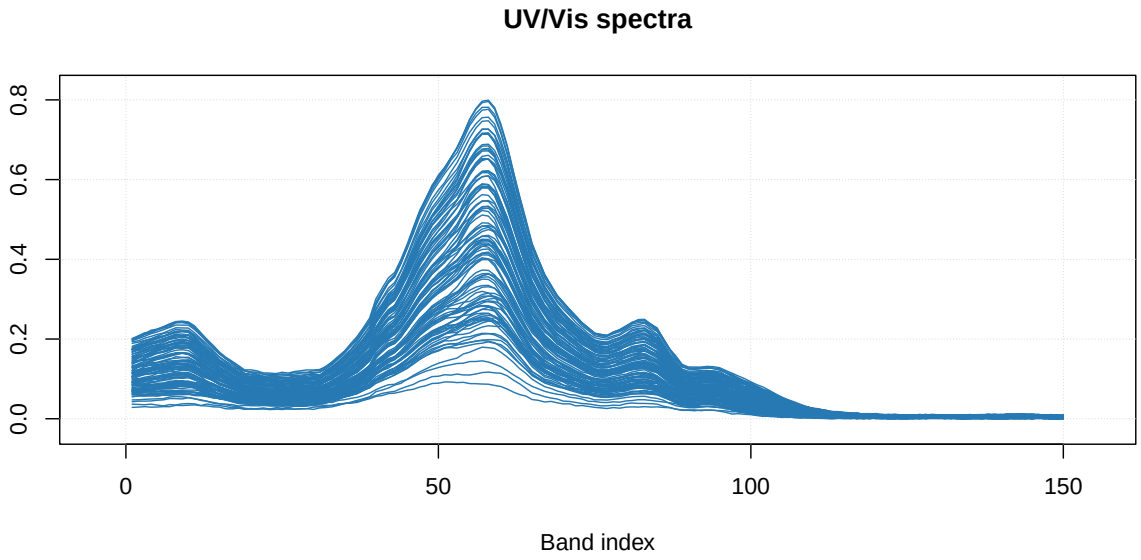
When line plot is created, the `mdatools()` shows a line plot for every row of the provided dataset. So if data set has more than one row, the plot will show a bunch of lines having same properties (color, type, etc). This is particularly useful when working with signals and spectroscopic data. In this subsection we will use simulated UV/Vis spectra from `simdata`. See `?simdata` for more details about this set.

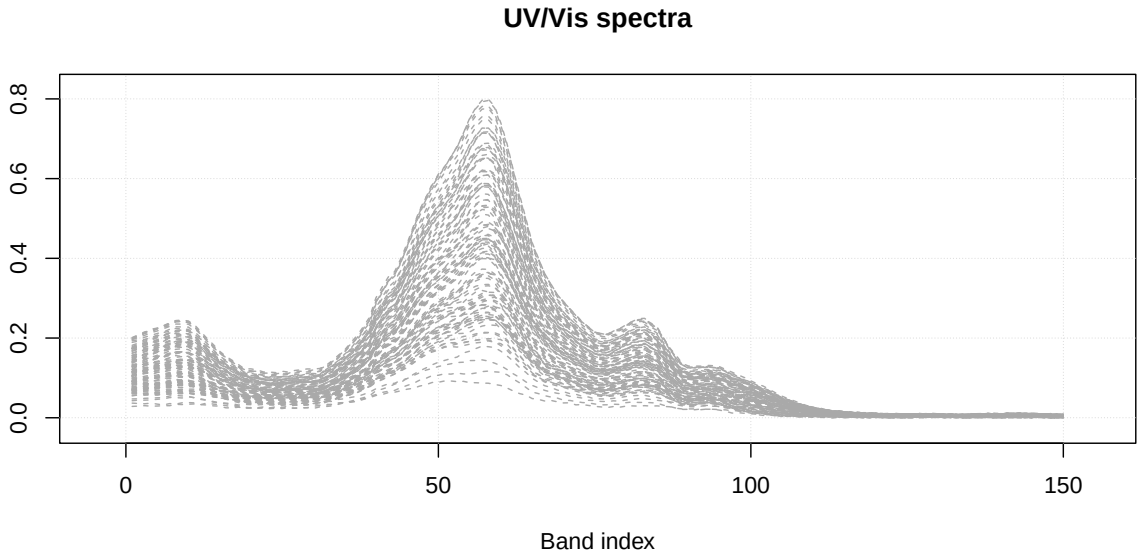
```
data(simdata)

# get the calibration spectra, wavelength and
# concentration of first component as separate variables
spectra = simdata$spectra.c
wavelength = simdata$wavelength
conc = simdata$conc.c[, 1]
```

```
# add names as attributes
attr(spectra, "name") = "UV/Vis spectra"
attr(spectra, "xaxis.name") = "Band index"
```

Here are simple examples of how to make the line plots.

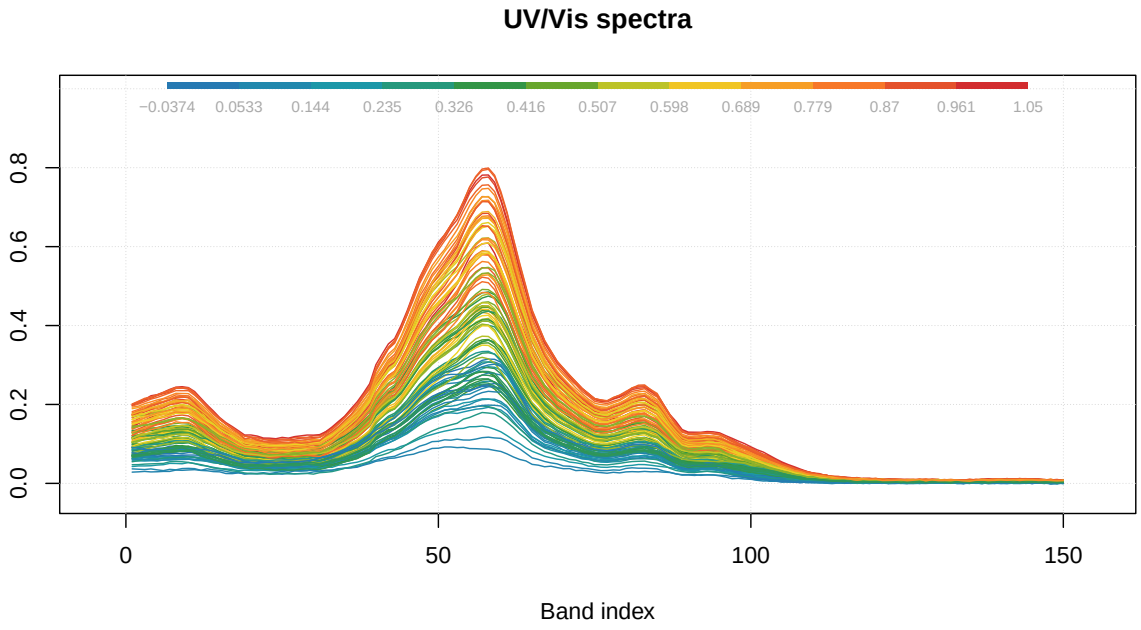
```
par(mfrow = c(2, 1))
mdaplot(spectra, type = "l")
mdaplot(spectra, type = "l", col = "darkgray", lty = 2)
```





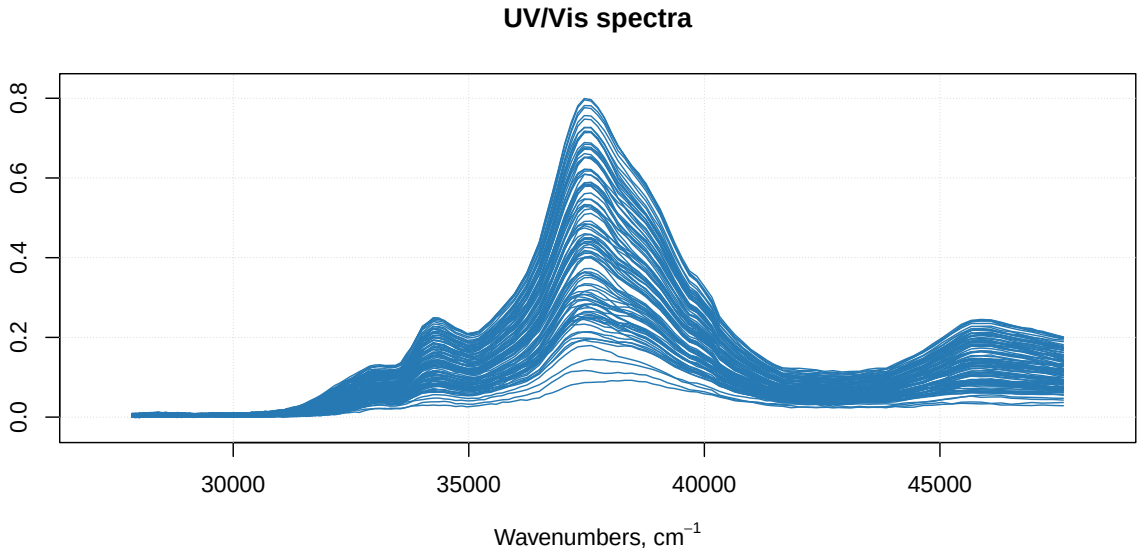
Most of the parameters described for scatter plots will work for the line plots as well. For example, you can colorise the lines by using a vector with some values (in the example below I use concentration of one of the chemical components).

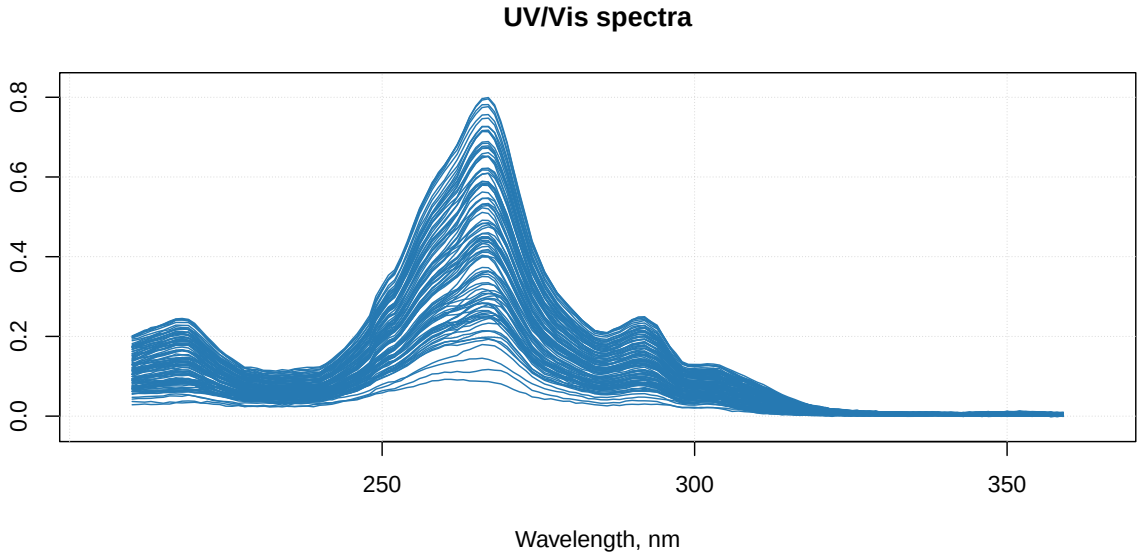
```
par(mfrow = c(1, 1))
mdaplot(spectra, type = "l", cgroup = conc)
```



And of course you can use the attributes, allowing to provide manual x-values — `'xaxis.values'` (similar parameter for y-values is `'yaxis.values'`). In the example below we show the spectra using wavelength in nm and wavenumbers in inverse cm.

```
par(mfrow = c(2, 1))

attr(spectra, "xaxis.name") = expression("Wavenumbers, cm"^-1)
attr(spectra, "xaxis.values") = 10^7/wavelength
mdaplot(spectra, type = "l")

attr(spectra, "xaxis.name") = "Wavelength, nm"
attr(spectra, "xaxis.values") = wavelength
mdaplot(spectra, type = "l")
```





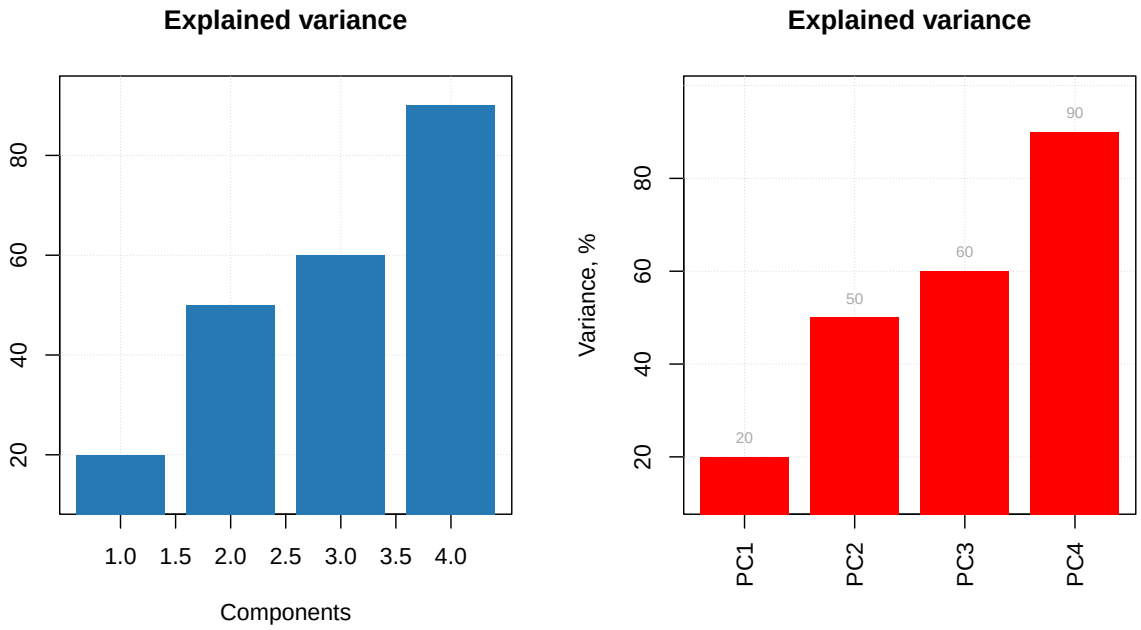
When you provide such data to any model methods (e.g. PCA, PLS, etc), then all variable related results (loadings, regression coefficients, etc.) will inherit this attribute and use it for making line plots.

Bar and errorbar plots

Bar plot is perhaps the simplest as it shows values for the first row of the data as bars. Let us start with simple dataset, where we have a matrix with explained variance of a data from e.g. PCA decomposition.

```
# make a simple two rows matrix with values
d = rbind(
   c(20, 50, 60, 90),
   c(14, 45, 59, 88)
)

# add some names and attributes
```

```
colnames(d) = paste0("PC", 1:4)
rownames(d) = c("Cal", "CV")
attr(d, "xaxis.name") = "Components"
attr(d, "name") = "Explained variance"

par(mfrow = c(1, 2))

# make a default bar plot
mdaplot(d, type = "h")

# make a bar plot with manual xtick labels, color and labels for data values
mdaplot(d, type = "h", xticks = seq_len(ncol(d)), xticklabels = colnames(d), col = "red",
   show.labels = TRUE, labels = "values", xlas = 2, xlab = "", ylab = "Variance, %")
```

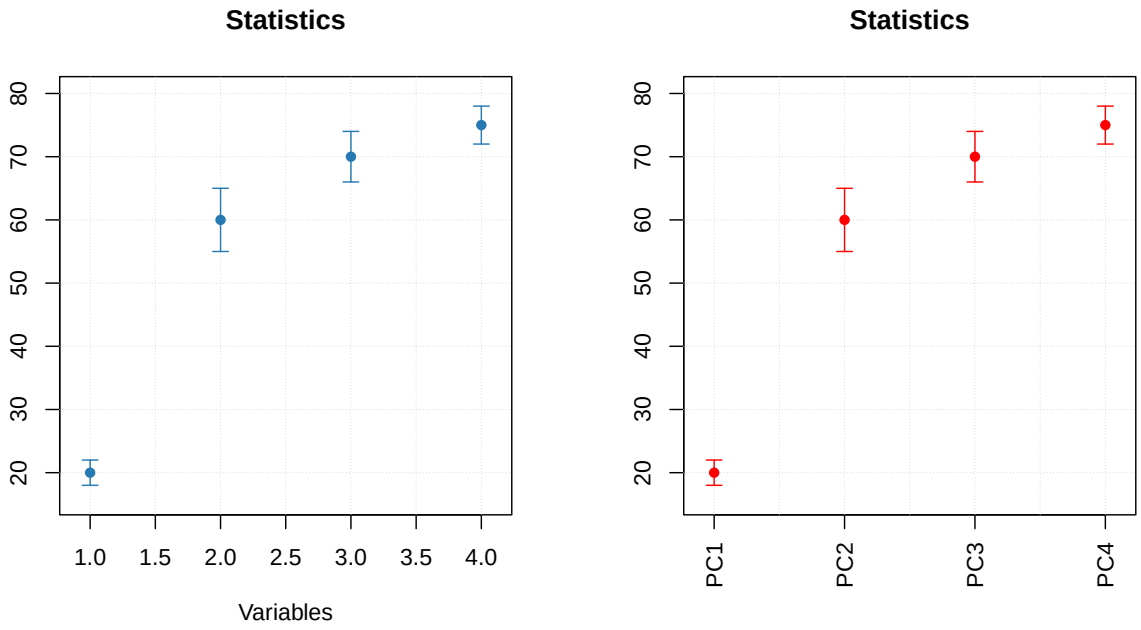


As you can notice, the values from the second rows were ignored, as bar plot always takes the first row.

Errorbar plot, in contrast, always expect data to have two or three rows. The first row is the origin points of the error bars, second row is the size of the bottom error bar and the third row is the size of the top error bar. If data has only two rows the both parts will be symmetric related to the origin.

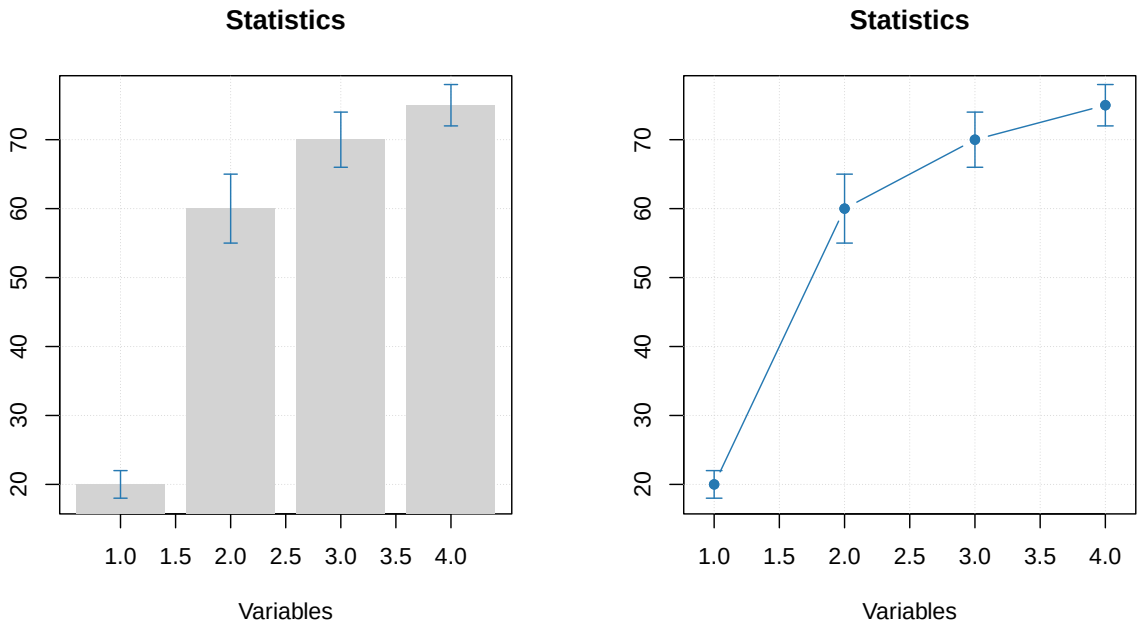
```
# generate some mean and standard deviation values and combine them as rows of a matrix
d = rbind(
   c(20, 60, 70, 75),
   c(2, 5, 4, 3)
)

# add names and attributes
rownames(d) = c("Mean", "Std")
colnames(d) = paste0("PC", 1:4)
attr(d, 'name') = "Statistics"
```

```
# show the plots
par(mfrow = c(1, 2))
mdaplot(d, type = "e")
mdaplot(d, type = "e", xticks = seq_len(ncol(d)),
   xticklabels = colnames(d), col = "red", xlas = 2, xlab = "")
```



All simple plots can be combined together on the same axes. In this case, first plot is created as usual and all other plots have to be created with option `show.axes = FALSE` as it is shown below. It must be noted that in this case axes limits have to be set manually when creating the first plot.

```
par(mfrow = c(1, 2))

mdaplot(mda.subset(d, 1), type = "h", col = "lightgray")
mdaplot(d, type = "e", show.axes = FALSE, pch = NA)

mdaplot(mda.subset(d, 1), type = "b")
mdaplot(d, type = "e", show.axes = FALSE)
```



In the next section we will discuss plots for several groups of objects (rows).

Plots for groups of objects

The package has another method for creating plots, `mdaplotg()`, which aims at making plots for groups of objects. It can be several groups of points, lines or bars, where every group has its own attributes, such as color, marker, line type and width, etc. There is a simple criterion to distinguish between the simple and group plots: group plots usually need a legend and simple plots — not. The `mdaplotg()` method allows to do a lot of things (e.g. split data into groups, add a legend and labels, etc) much easier and this section will show most of the details.

There are three ways to provide data sets for making the group plots. Let's discuss them first and then talk about some extra features.

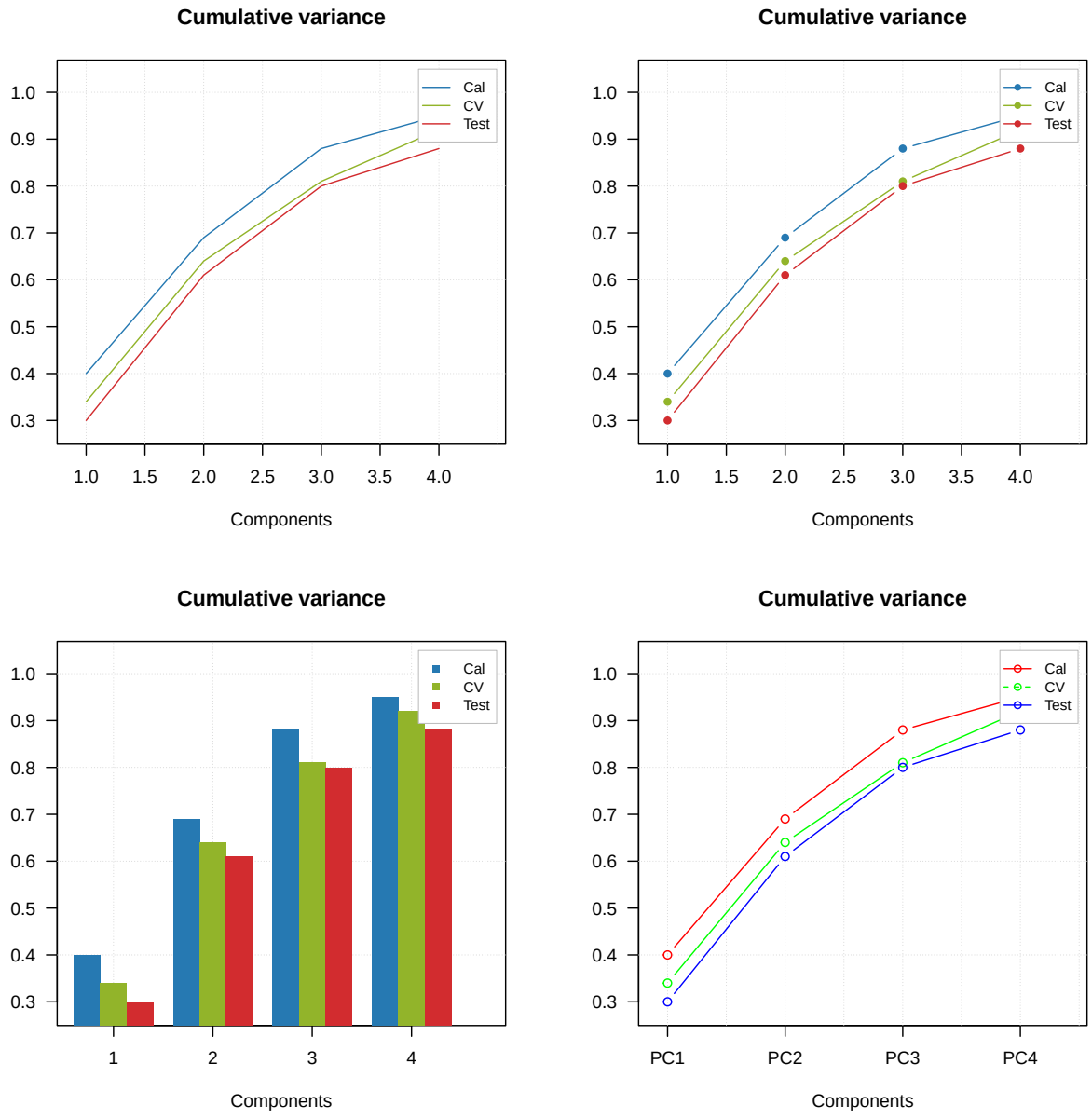
One matrix or data frame

If dataset is a matrix or a data frame, `mdaplotg()` will make a line, scatter-line or a bar plot, considering every row as a separate group. This can be useful, when, for example, you want to show how explained variance depends on a number of components for calibration and test set, or how loadings for the first two components look like.

If you want to change any parameters, like `pch`, `lty`, `lwd`, `col` or similar you need to provide either a vector with values for each group or one value for all groups. Axis limits, ticks, ticklabels, etc. can be defined similarly to the simple plots. Here are some examples.

```
# let's create a simple dataset with 3 rows
p = rbind(
   c(0.40, 0.69, 0.88, 0.95),
   c(0.34, 0.64, 0.81, 0.92),
   c(0.30, 0.61, 0.80, 0.88)
)

# add some names and attributes
```

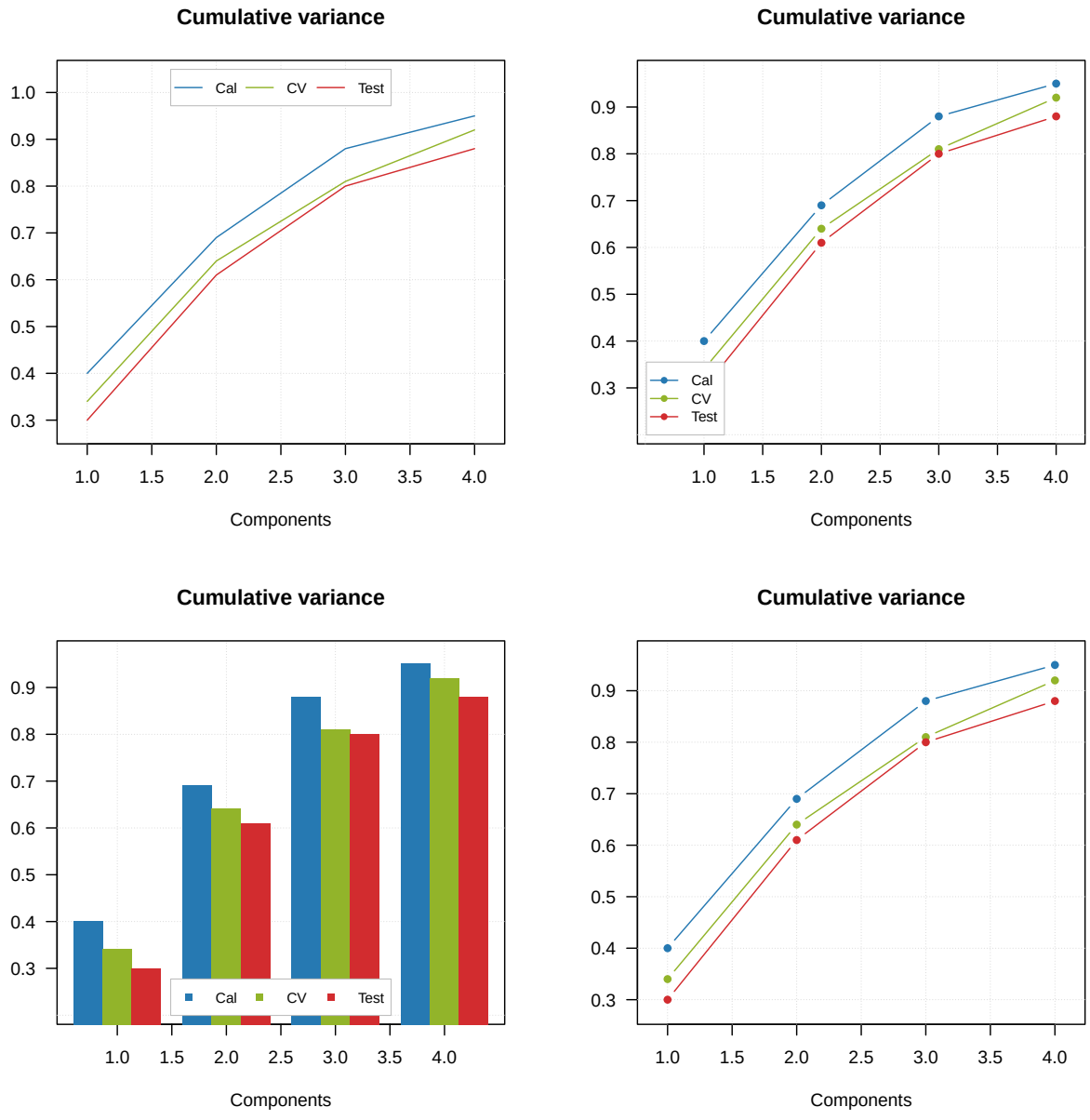
```
rownames(p) = c("Cal", "CV", "Test")
colnames(p) = paste0("PC", 1:4)
attr(p, "name") = "Cumulative variance"
attr(p, "xaxis.name") = "Components"

# and make group plots of different types
par(mfrow = c(2, 2))
mdaplotg(p, type = "l")
mdaplotg(p, type = "b")
mdaplotg(p, type = "h", xticks = 1:4)
mdaplotg(p, type = "b", lty = c(1, 2, 1), col = c("red", "green", "blue"), pch = 1,
         xticks = 1:4, xticklabels = colnames(p))
```



As you can see, `mdaplotg()` automatically created the legend and set colors, line parameters, etc. correctly.

You can change position of the legend using same names as for basic `legend()` command from R, or hide it using parameter `show.legend = FALSE`, as it is shown below.

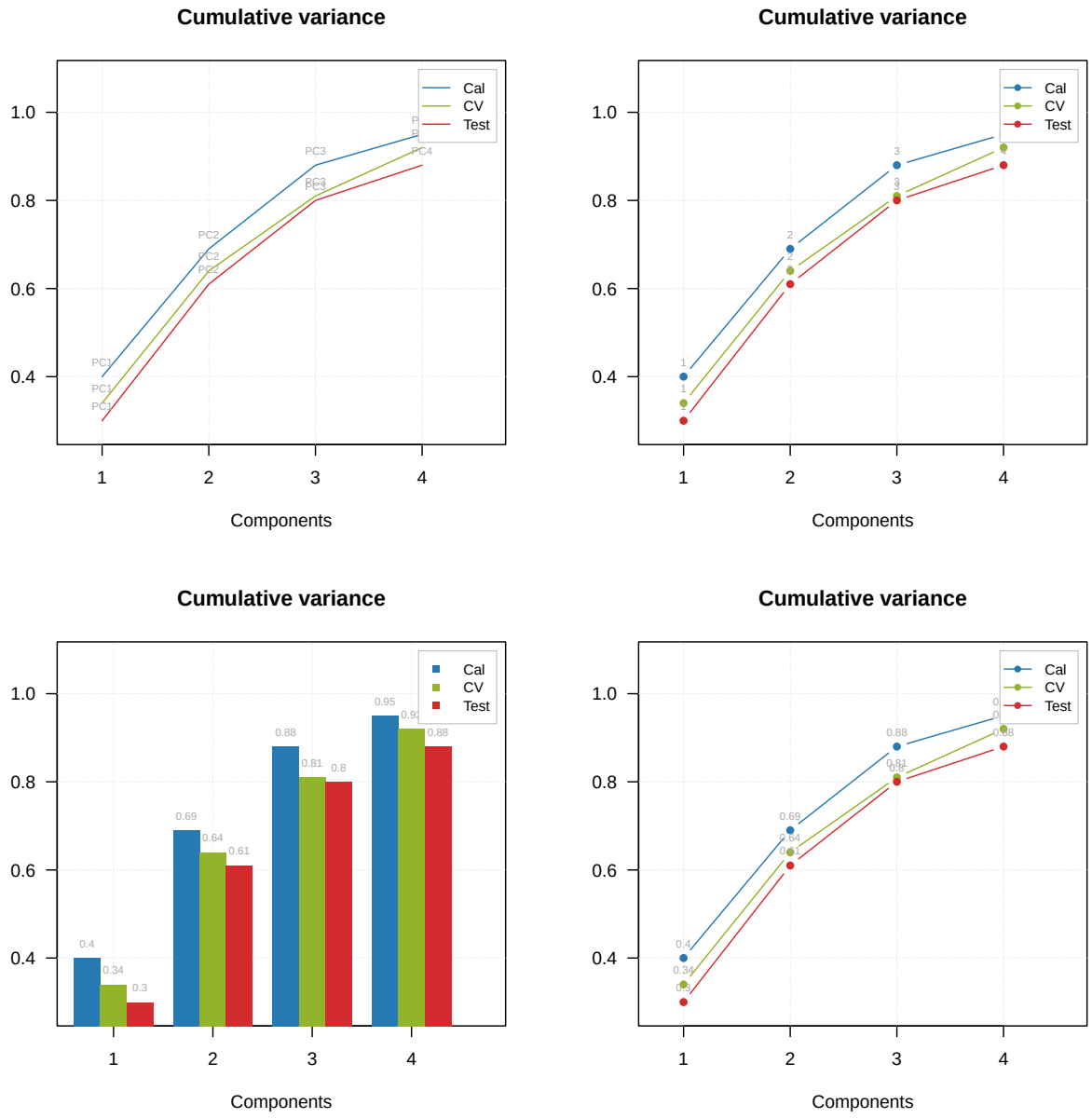
```
par(mfrow = c(2, 2))
mdaplotg(p, type = "l", legend.position = "top")
mdaplotg(p, type = "b", legend.position = "bottomleft")
mdaplotg(p, type = "h", legend.position = "bottom")
mdaplotg(p, type = "b", show.legend = FALSE)
```



Group plot also allows to show labels, in this case they can be either values, names or indices of the columns.

```
par(mfrow = c(2, 2))
mdaplotg(p, type = "l", show.labels = TRUE)
mdaplotg(p, type = "b", show.labels = TRUE, labels = "indices")
mdaplotg(p, type = "h", show.labels = TRUE, labels = "values")
```

```
mdaplotg(p, type = "b", show.labels = TRUE, labels = "values")
```



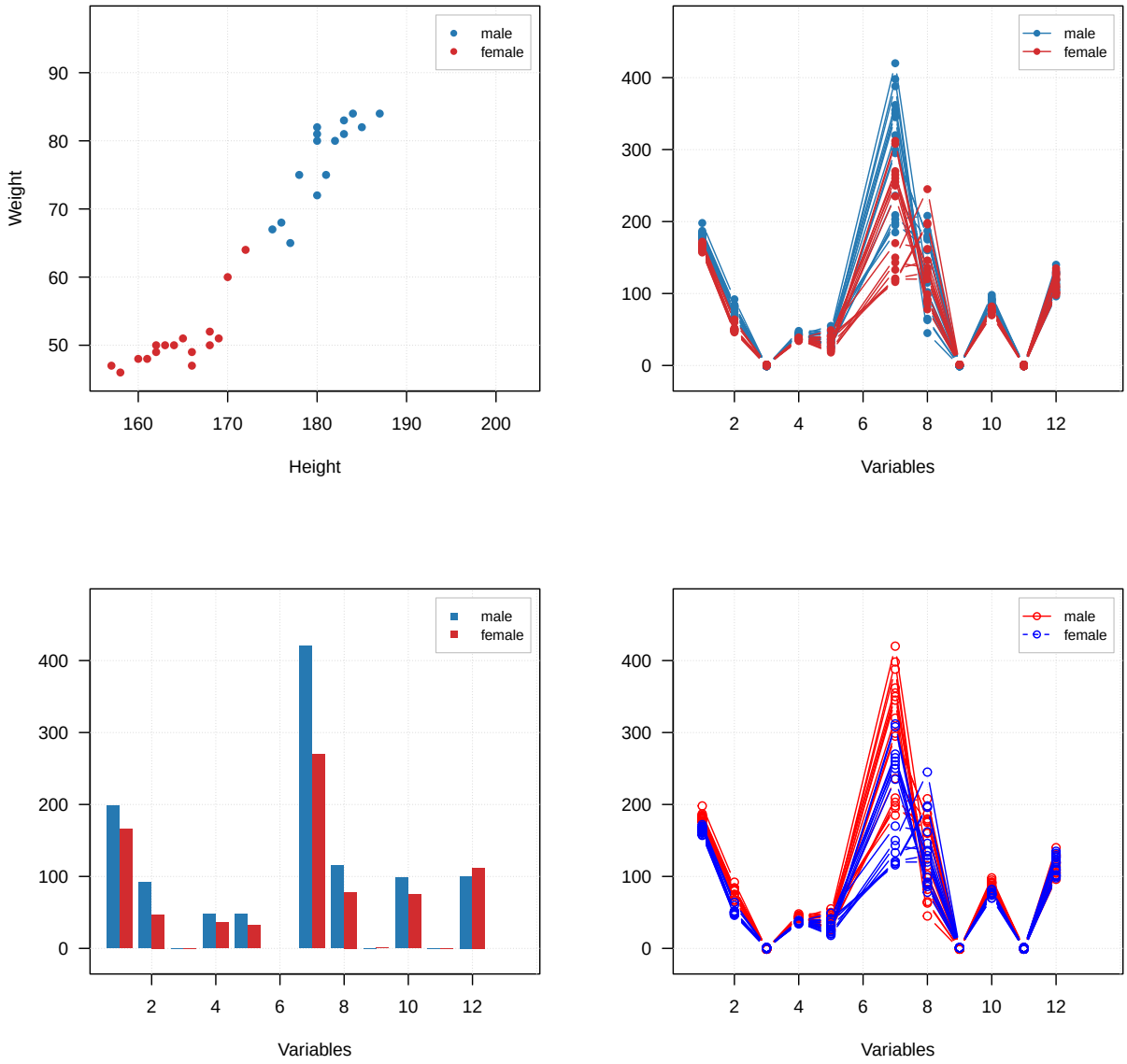
List with matrices or data frames

In this case every element of the list will be treated as a separate group. This way allows to make scatter plots as well and line plots with several lines in each group. Barplot can be also made but in this case first row from each datasets will be used. If you use names when create the list, the names will be taken as legend labels, otherwise method will look at attribute `"name"` for each data set.

In the example below we split *People* data to males and females and show the group plots.

```
# load data and exclude column with income
data(people)
people = mda.exclcols(people, "Income")
```

```
# use values of sex variable to split data into two subsets
sex = people[, "Sex"]
m = mda.subset(people, subset = sex == -1)
f = mda.subset(people, subset = sex ==  1)

# combine the two subsets into a named list
d = list(male = m, female = f)

# make plots for the list
par(mfrow = c(2, 2))
mdaplotg(d, type = "p")
mdaplotg(d, type = "b")
mdaplotg(d, type = "h")
mdaplotg(d, type = "b", lty = c(1, 2), col = c("red", "blue"), pch = 1)
```



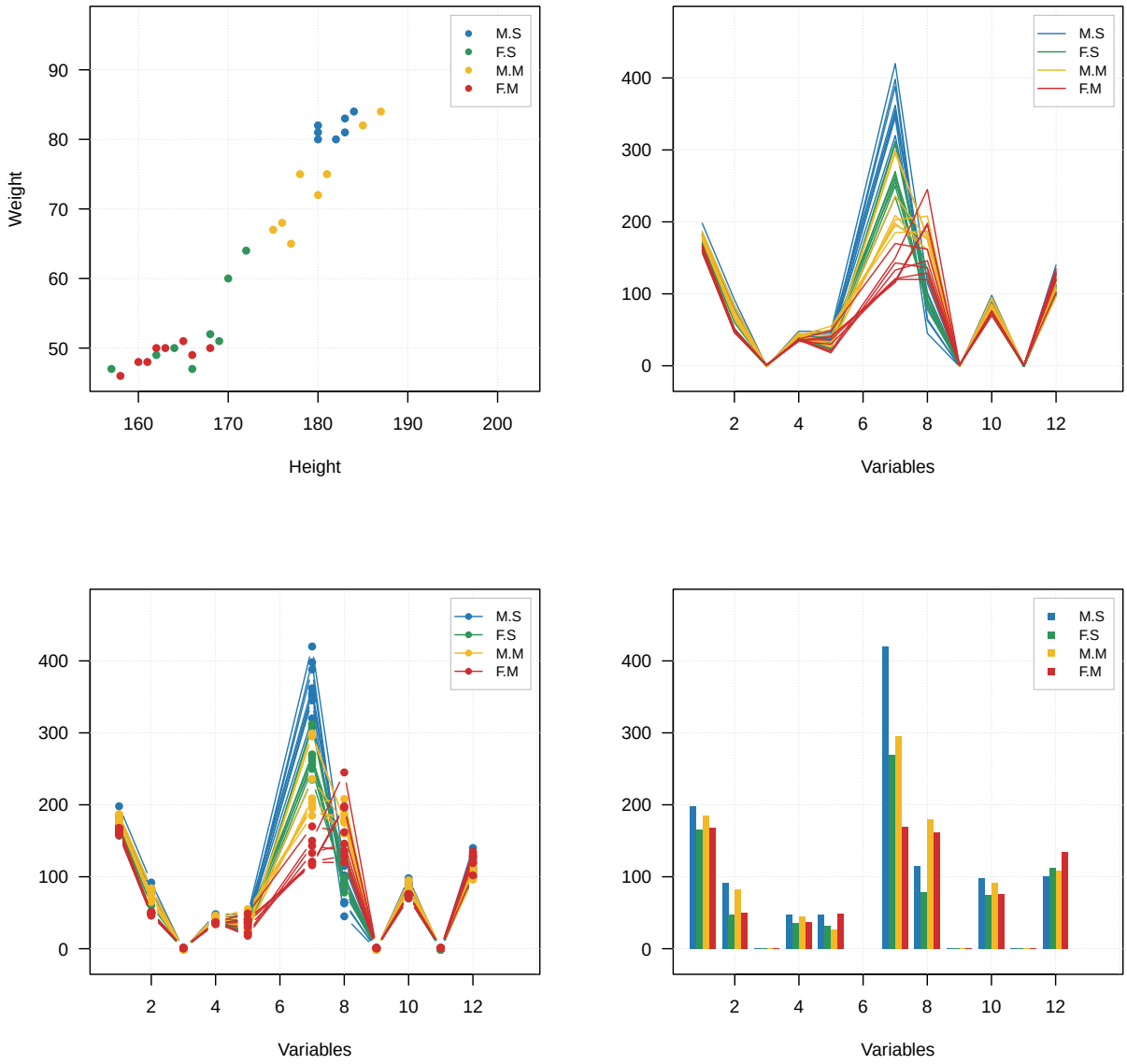
Most of the things described in the previous subsection will work similarly for this case. We will just add a bit more details on how labels and excluded rows are processed for the scatter plots. By default labels are row names or indices. In `mdaplotg()` you can not provide vector with manual values, so the best way to change them is to assign them as the row names. Indices are unique within each group, so if you have, e.g. three groups of points, there will be three points with index "1", three with "2", etc.

Use factors to split a dataset into groups

One more way to split data set into groups is to provide one or several factor columns using argument `groupby`. In this case `mdaplotg()` will find all combinations of the factor levels and split rows of dataset to the corresponding groups. In the example below we use variables *Region* and *Sex* to make plots for four groups.

It is assumed that you already loaded the Poeple data (from previous examples) and excluded the Income column.

```
sex = factor(people[, "Sex"], labels = c("M", "F"))
reg = factor(people[, "Region"], labels = c("S", "M"))
groups = data.frame(sex, reg)

par(mfrow = c(2, 2))
mdaplotg(people, type = "p", groupby = groups)
mdaplotg(people, type = "l", groupby = groups)
mdaplotg(people, type = "b", groupby = groups)
mdaplotg(people, type = "h", groupby = groups)
```



All parameters, described before, will work the same way in this case.

Working with images

The package also supports images, including hyperspectral images, however they have to be transformed into datasets. The idea is very simple, we keep information about image pixels in an unfolded form, as a matrix, and use attributes `width` and `height` to reshape the data when we need to show it as an image.

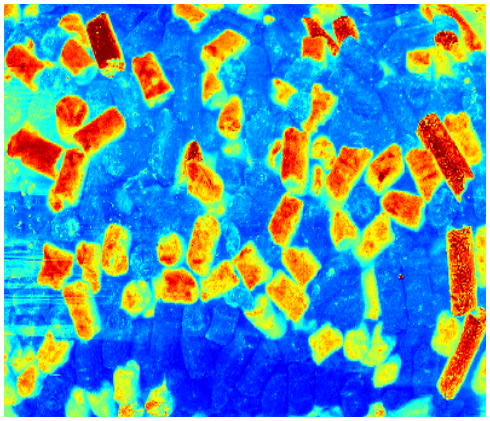
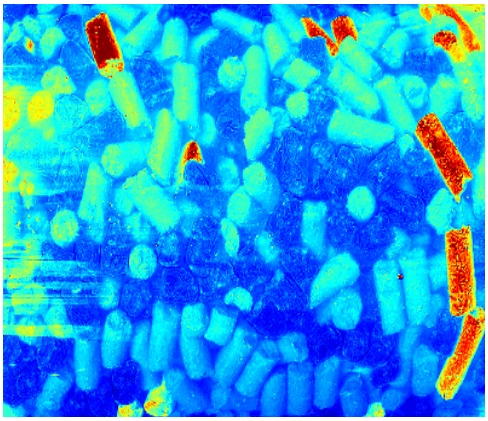
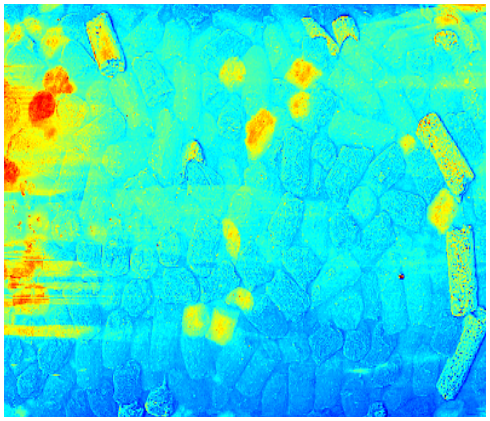
There are three methods that make this procedure easier: `mda.im2data()`, `mda.data2im()` and `imshow()`. The first converts an image (represented as 3-way array) to a data set, second does the opposite and the third takes dataset and shows it as an image. In the code chunk below you will see several examples how the methods work.

We will use a dataset `image` available in the package. It is a 3-way array of numbers, if you want to work with e.g. JPEG, PNG or other standard image files you can load them using specific packages (jpeg, png).

```
data(pellets)

# convert image to a data matrix and add some attributed
d = mda.im2data(pellets)
colnames(d) = c("Red", "Green", "Blue")
attr(d, "name") = "Image"

# show data values
mda.show(d, 10)
```

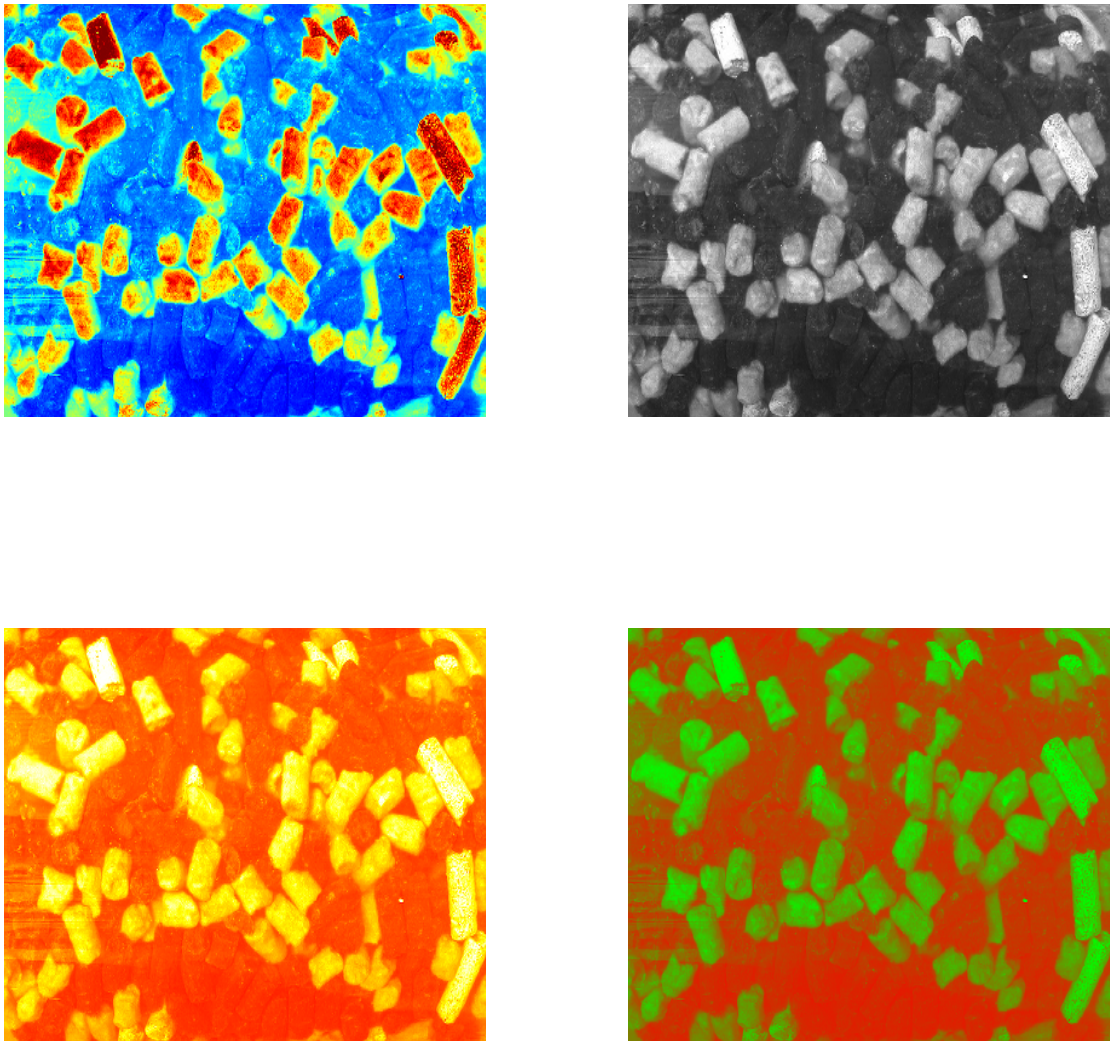
```
## Image
## -----
##       Red Green Blue
##  [1,] 183    80  101
##  [2,] 191    76  105
##  [3,] 187    73   99
##  [4,] 199    81  113
##  [5,] 198    81  110
##  [6,] 197    84  114
##  [7,] 191    83  109
##  [8,] 193    83  110
##  [9,] 188    83  114
## [10,] 172    86  115
```

```
# show separate channels and the whole image in plots
par(mfrow = c(2, 2))
imshow(d, 1)
imshow(d, 2)
imshow(d, 3)
imshow(d, 1:3)
```

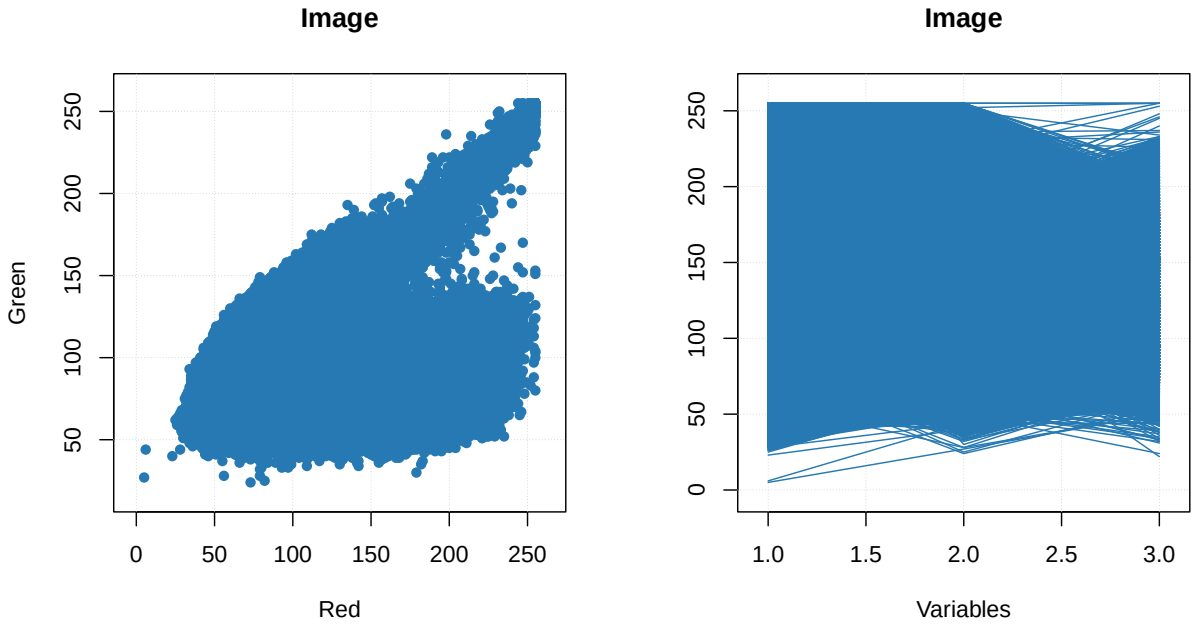

By default image for one channels is shown using jet color palette for intensities, but you can also use gray colors, palette from *colorbrewer2* as well as your own.

```
par(mfrow = c(2, 2))
imshow(d, 1)
imshow(d, 1, colmap = "gray")
imshow(d, 1, colmap = heat.colors(256))
imshow(d, 1, colmap = colorRampPalette(c("red", "green"))(256))
```



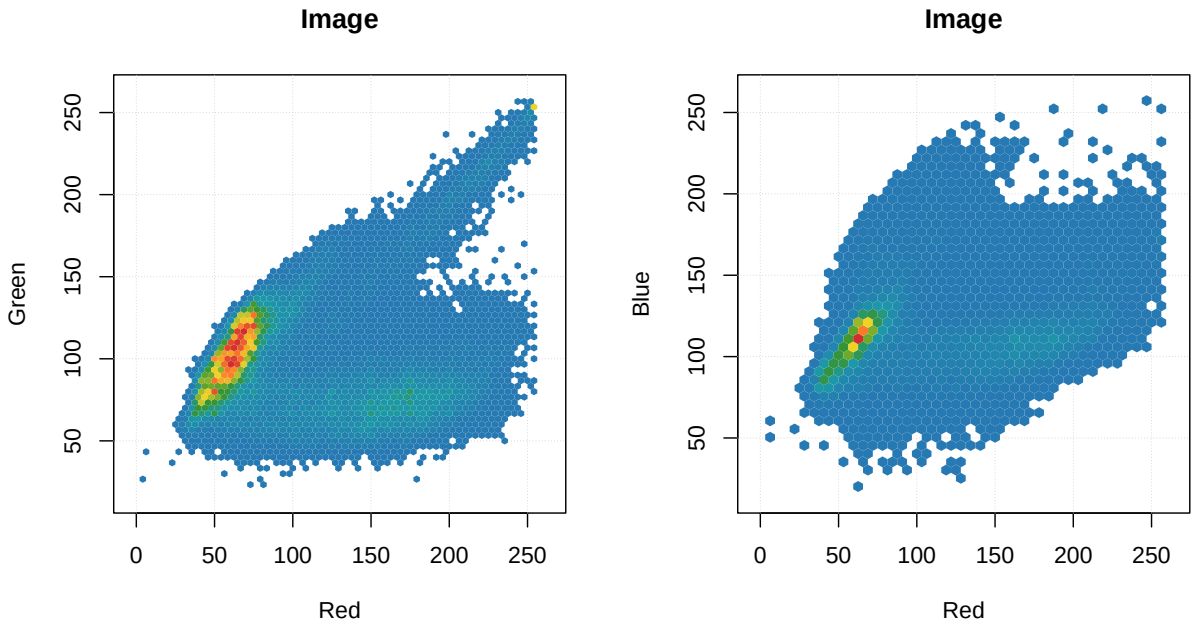
You can work with the image values as normal dataset and show scatter, line plots, calculate statistics, etc.

```
par(mfrow = c(1, 2))
mdaplot(d, type = "p")
mdaplot(d, type = "l")
```



However, it will take some time to show these plots as this image has several hundreds of thousands pixels, a faster alternative can be the use density plot based on hexagonal binning. Use `type = "d"` for this as shown below. Number of bins can be adjusted by using parameter `nbins`.

```
par(mfrow = c(1, 2))
mdaplot(d, type = "d")
mdaplot(mda.subset(d, select = c("Red", "Blue")), type = "d", nbins = 40)
```



Another useful thing is to set some of the pixels as background. The background pixels are removed from the image dataset physically, there is no way to get them back (in cotrast to excluded rows/pixels). It can

be particularly useful when working with e.g. geocorrected hyperspectral images, where, often, many pixels have NA values and there is no need to keep them in memory. To set pixels as background you need to use method `mda.setimbg()` with either pixel indices or vector with logical values as it is shown below.

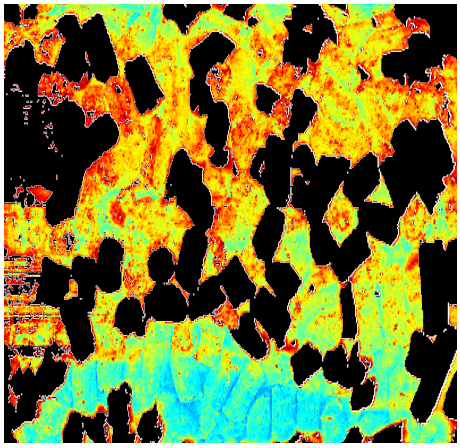
```
# original size
show(dim(d))
```

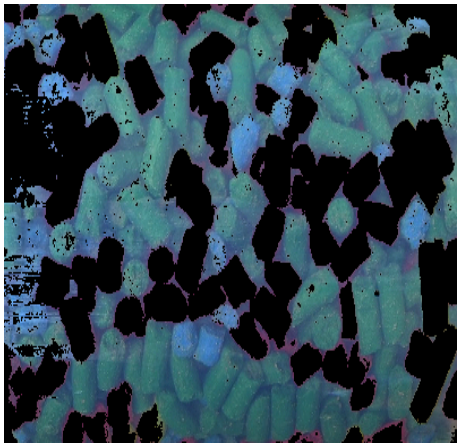
```
## [1] 114000      3
```

```
# set red epixels as background and show new size
d = mda.setimbg(d, d[, "Red"] > 100)
show(dim(d))
```

```
## [1] 66471     3
```

```
# show image with background pixels
par(mfrow = c(1, 2))
imshow(d, 1)
imshow(d, 1:3)
```





All image related attributes are inherited by all object/rows related results, e.g. scores, residuals, predicted values and classes, etc. This means if you provide an image to any modelling method, you can visualise the corresponding results also as an image. Some examples will be shown in chapter about PCA.

Preprocessing

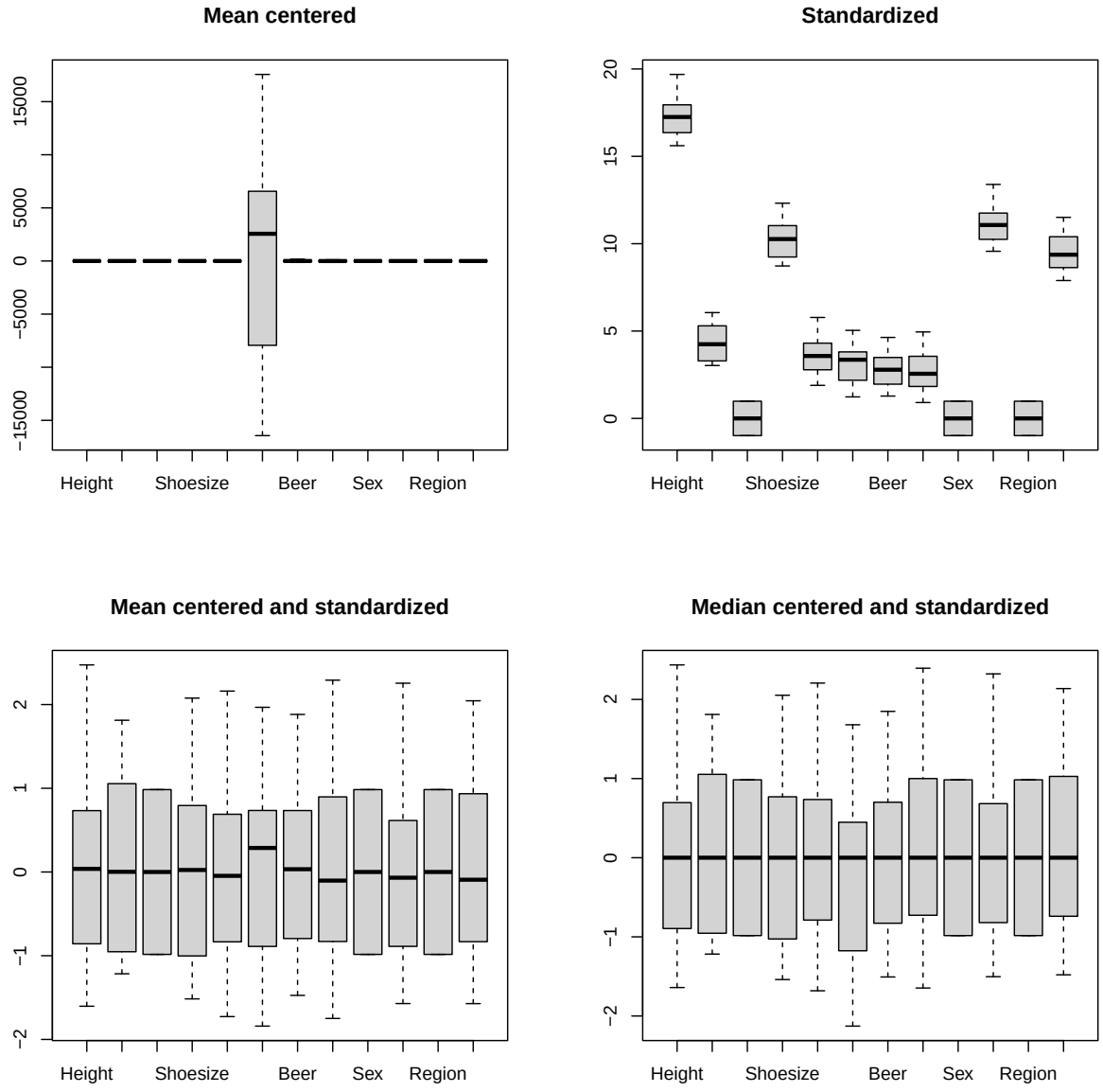
The package has several preprocessing methods implemented, mostly for different kinds of spectral data (but many of the methods will work with other datasets as well). All functions for preprocessing starts from prefix `prep.` which makes them easier to find by using code completion. In this chapter a brief description of the methods with several examples is shown.

Autoscaling

Autoscaling consists of two steps. First step is *centering* (or, more precise, *mean centering*) when center of a data cloud in variable space is moved to an origin. Mathematically it is done by subtracting mean from the data values separately for every column/variable. Second step is *scaling* og *standardization* when data values are divided to standard deviation so the variables have unit variance. This autoscaling procedure (both steps) is known in statistics simply as *standardization*. You can also use arbitrary values to center or/and scale the data, in this case use sequence or vector with these values should be provided as an argument for `center` or `scale`.

R has a built-in function for centering and scaling, `scale()`. The method `prep.autoscale()` is actually a wrapper for this function, which is mostly needed to set all user defined attributes to the result (all preprocessing methods will keep the attributes). Here are some examples how to use it:

```
library(mdatools)

# load People data
data(people)

# mean centering only
data1 = prep.autoscale(people, center = TRUE, scale = FALSE)

# scaling/standardization only
data2 = prep.autoscale(people, center = FALSE, scale = TRUE)

# autoscaling (mean centering and standardization)
data3 = prep.autoscale(people, center = TRUE, scale = TRUE)

# centering with median values and standardization
data4 = prep.autoscale(people, center = apply(people, 2, median), scale = TRUE)

par(mfrow = c(2, 2))
boxplot(data1, main = "Mean centered")
boxplot(data2, main = "Standardized")
boxplot(data3, main = "Mean centered and standardized")
boxplot(data4, main = "Median centered and standardized")
```

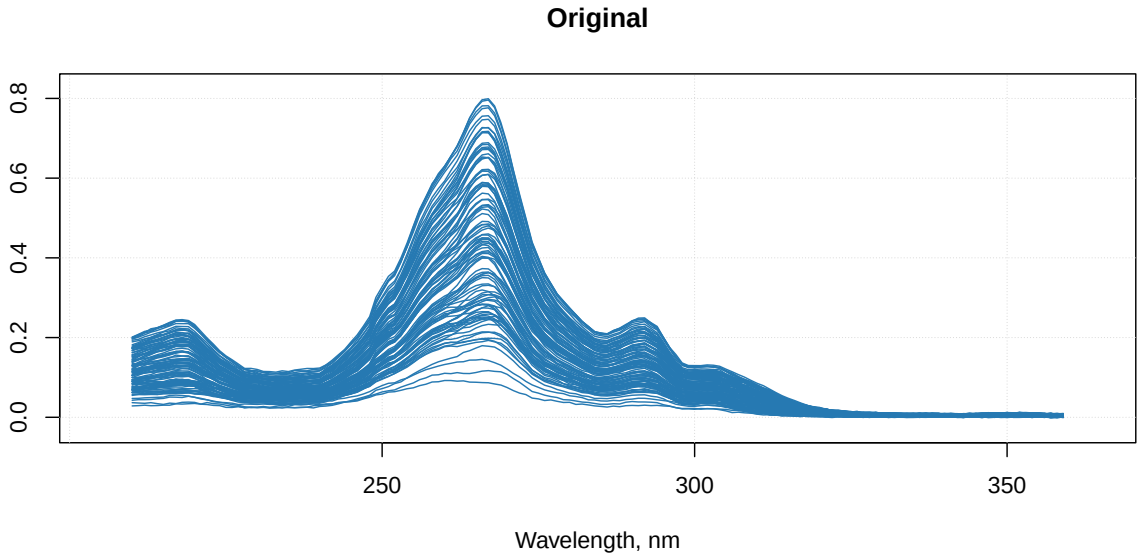


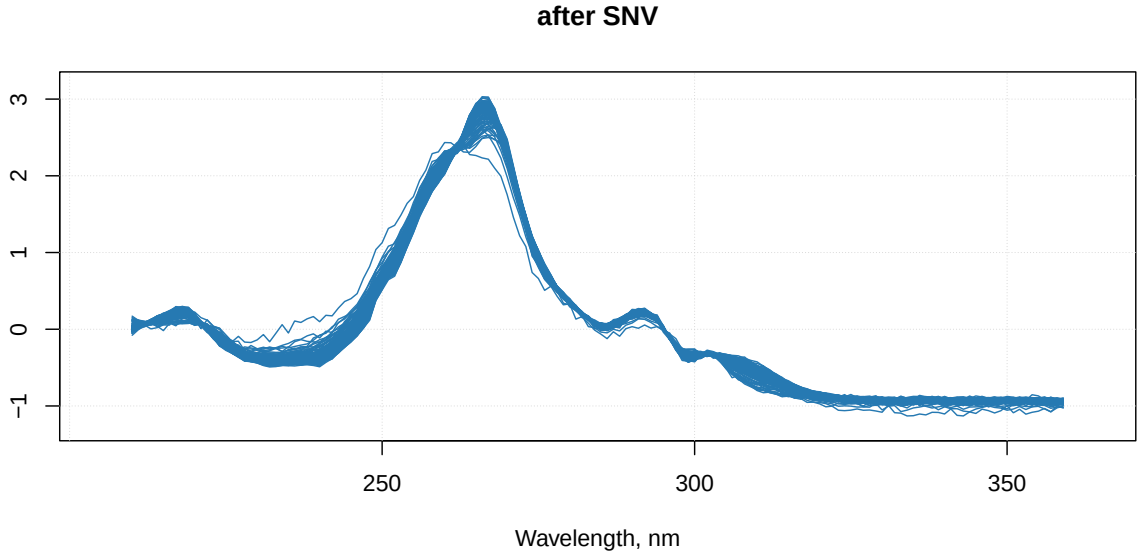
The method has also an additional parameter `max.cov` which helps to avoid scaling of variables with zero or very low variation. The parameter defines a limit for coefficient of variation in percent `sd(x) / m(x) * 100` and the method will not scale variables with coefficient of variation below this limit. Default value for the parameter is 0 which will prevent scaling of constant variables (which is leading to `Inf` values).

Correction of spectral baseline

Baseline correction methods include Standard Normal Variate (SNV), Multiplicative Scatter Correction (MSC) and correction of baseline with Asymmetric Least Squares (ALS). SNV is a very simple procedure aiming first of all at remove additive and multiplicative scatter effects from Vis/NIR spectra as well as correct the global intensity effect. It is applied to every individual spectrum by subtracting its average and dividing its standard deviation from all spectral values. Here is an example:

```
# load UV/Vis spectra from Simdata
data(simdata)
ospectra = simdata$spectra.c
attr(ospectra, "xaxis.values") = simdata$wavelength
attr(ospectra, "xaxis.name") = "Wavelength, nm"

# apply SNV and show the spectra
pspectra = prep.snv(ospectra)

par(mfrow = c(2, 1))
mdaplot(ospectra, type = "l", main = "Original")
mdaplot(pspectra, type = "l", main = "after SNV")
```



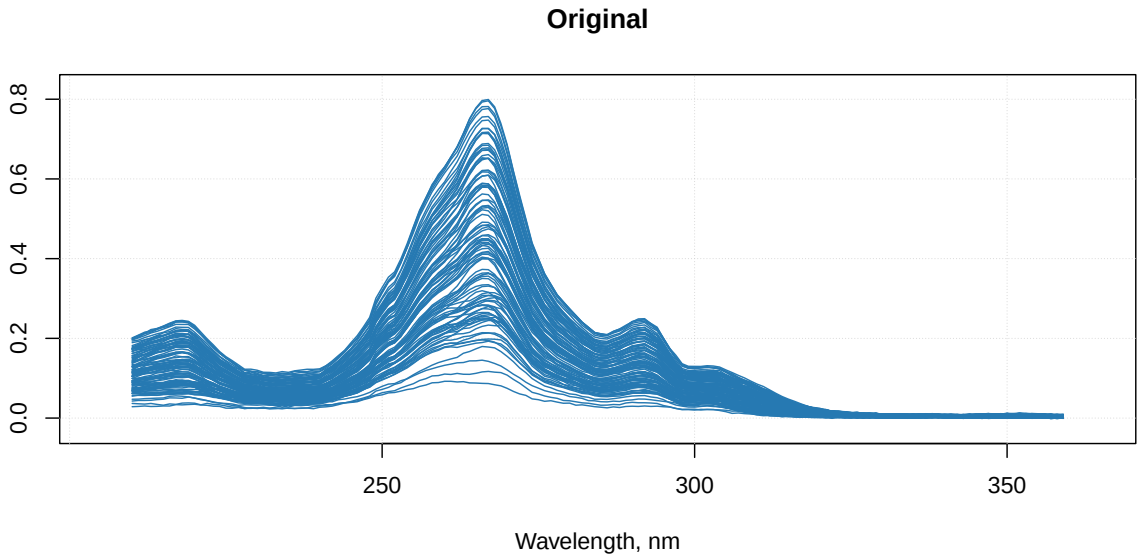


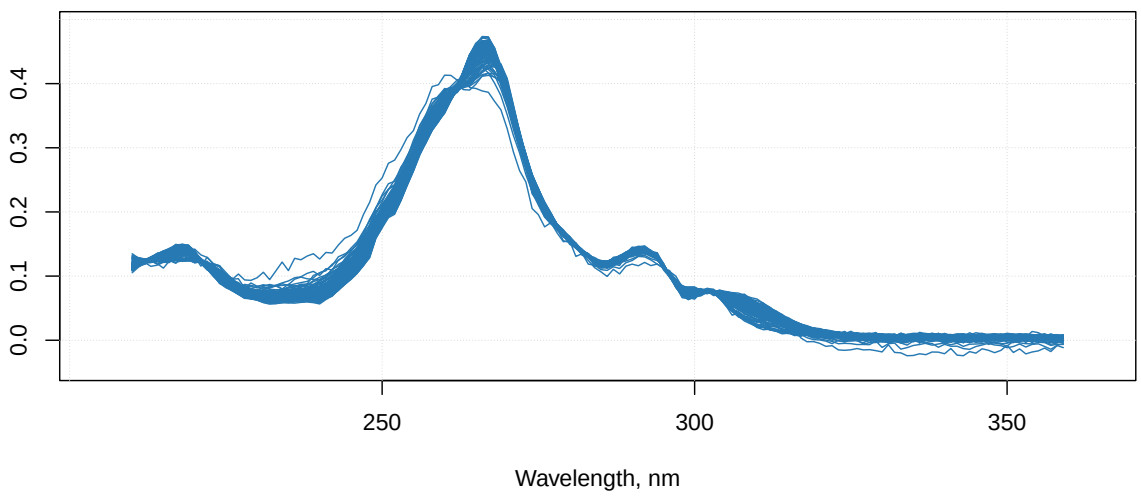
Multiplicative Scatter Correction does similar job but in a different way. First it calculates a mean spectrum

for the whole set (mean spectrum can be also provided as an extra argument). Then, for each individual spectrum, it makes a line fit for the spectral values and the mean spectrum. The coefficients of the line, intercept and slope, are used to correct the additive and multiplicative effects correspondingly.

The `prep.msc()` function adds the mean spectrum calculated for the original spectral data, to the attributes of the results, so it can be reused later.

```
# apply MSC and and get the preprocessed spectra
pspectra = prep.msc(ospectra)

# show the result
par(mfrow = c(2, 1))
mdaplot(ospectra, type = "l", main = "Original")
mdaplot(pspectra, type = "l", main = "after MSC")
```





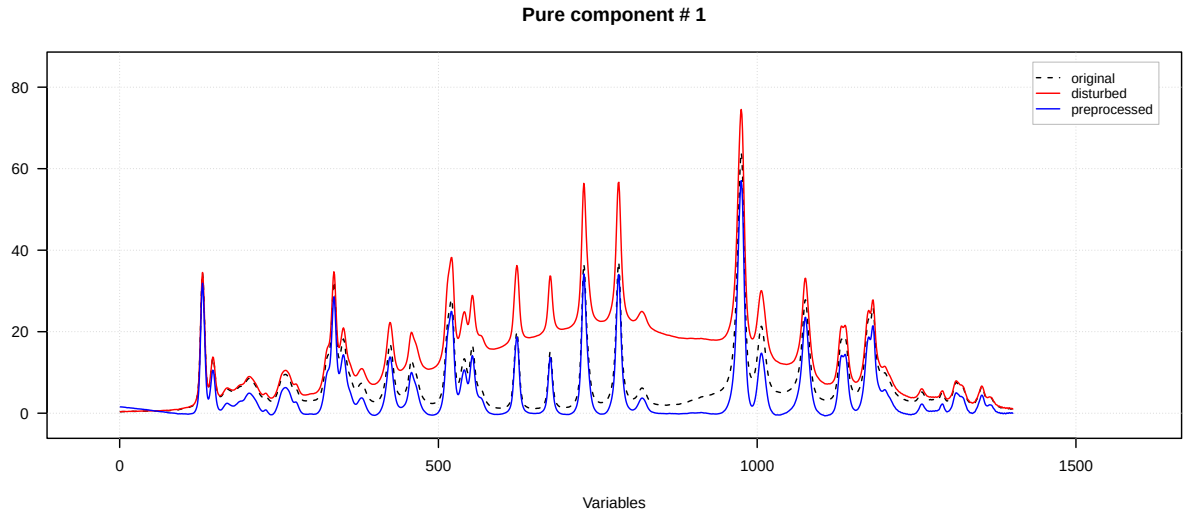
Baseline correction with asymmetric least squares

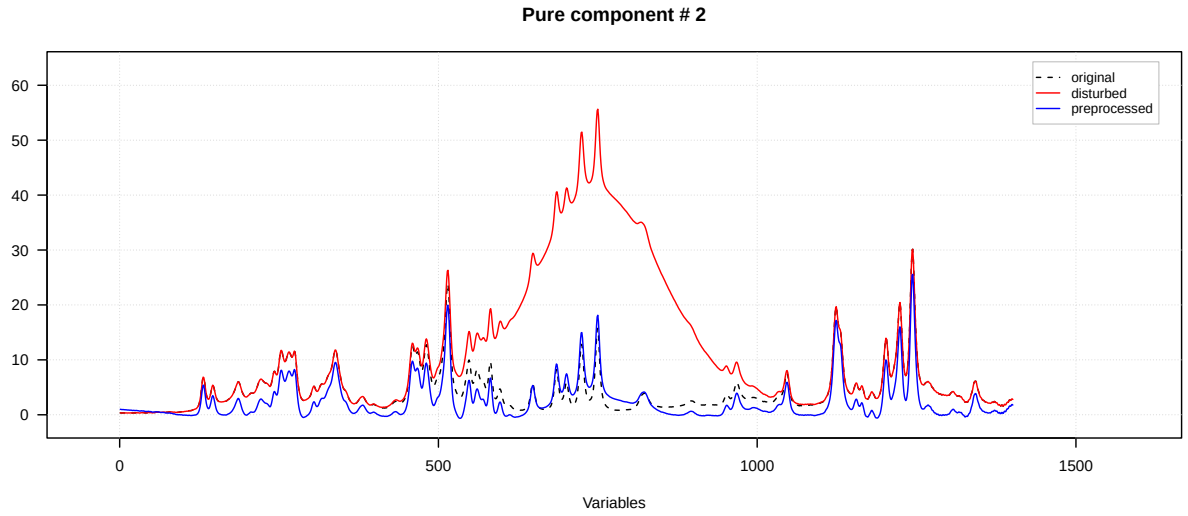
Asymmetric least squares (ALS) baseline correction allows you to correct baseline issues, which have wider shape comparing to the characteristic peaks. It can be used for example to correct the fluorescence effect in Raman spectra.

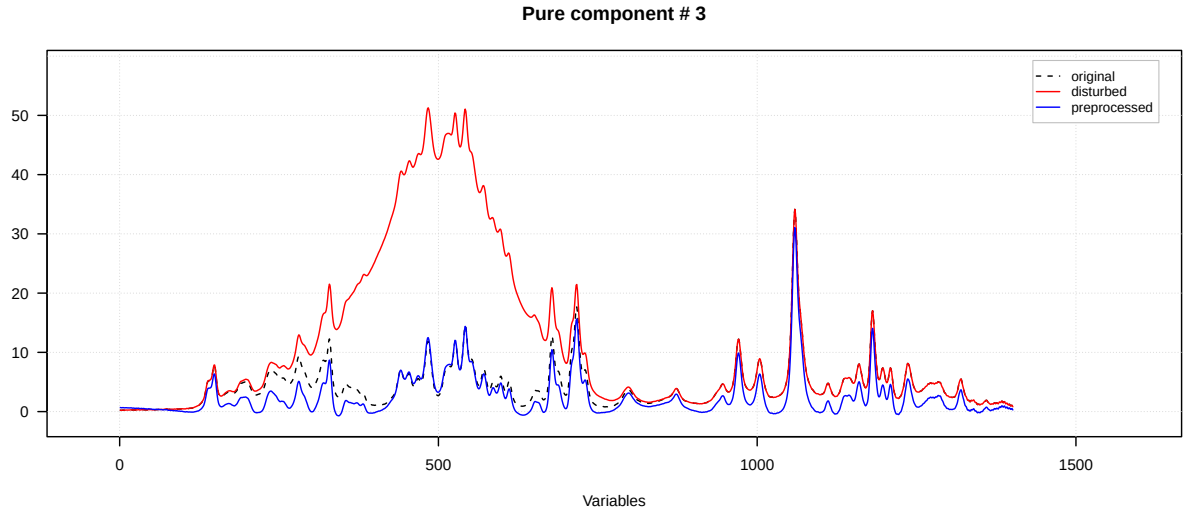
The method is based on Whittaker smoother and was proposed in this paper. It is implemented as a function `prep.alsbasecorr()`, which has two main parameters — power of a penalty parameter (`plambda`, usually varies betwen 2 and 9) and the ratio of asymmetry (`p`, usually between 0.1 and 0.001). For example, if `plambda = 5`, the penalty parameter λ, described in the paper will be equal to 10^5.

The choice of the parameters depends on how broad the disturbances of the baseline are and how narrow the original spectral peaks are. In the example below we took original spectra from the `carbs` dataset, add baseline disturbance using broad Gaussian peaks and then tried to remove the disturbance by applying the `prep.alsbasecorr()`. The result is shown in form of plots.

```
library(mdatools)
data(carbs)

# take original spectra from carbs dataset
x <- t(carbs$S)

# add disturbance to the baseline by using broad Gaussian peaks
y <- x + rbind(
   dnorm(1:ncol(x), 750, 200) * 10000,
   dnorm(1:ncol(x), 750, 100) * 10000,
   dnorm(1:ncol(x), 500, 100) * 10000
)

# preprocess the disturbed spectra using ALS baseline correction
y.new <- prep.alsbasecorr(y, plambda = 5, p = 0.01)

# show the original, disturbed and the preprocessed spectra separately for each component
par(mfrow = c(3, 1))
for (i in 1:3) {
   mdaplotg(list(
      original = x[i, , drop = FALSE],
      disturbed = y[i, , drop = FALSE],
      preprocessed = y.new[i, , drop = FALSE]
      ), type = "l", lty = c(2, 1, 1), col = c("black", "red", "blue"),
      main = paste("Pure component #", i)
   )
}
```







As one can notice, the blue curves with corrected spectra are pretty similar to the original spectra shown as

dashed black curves.

Normalization

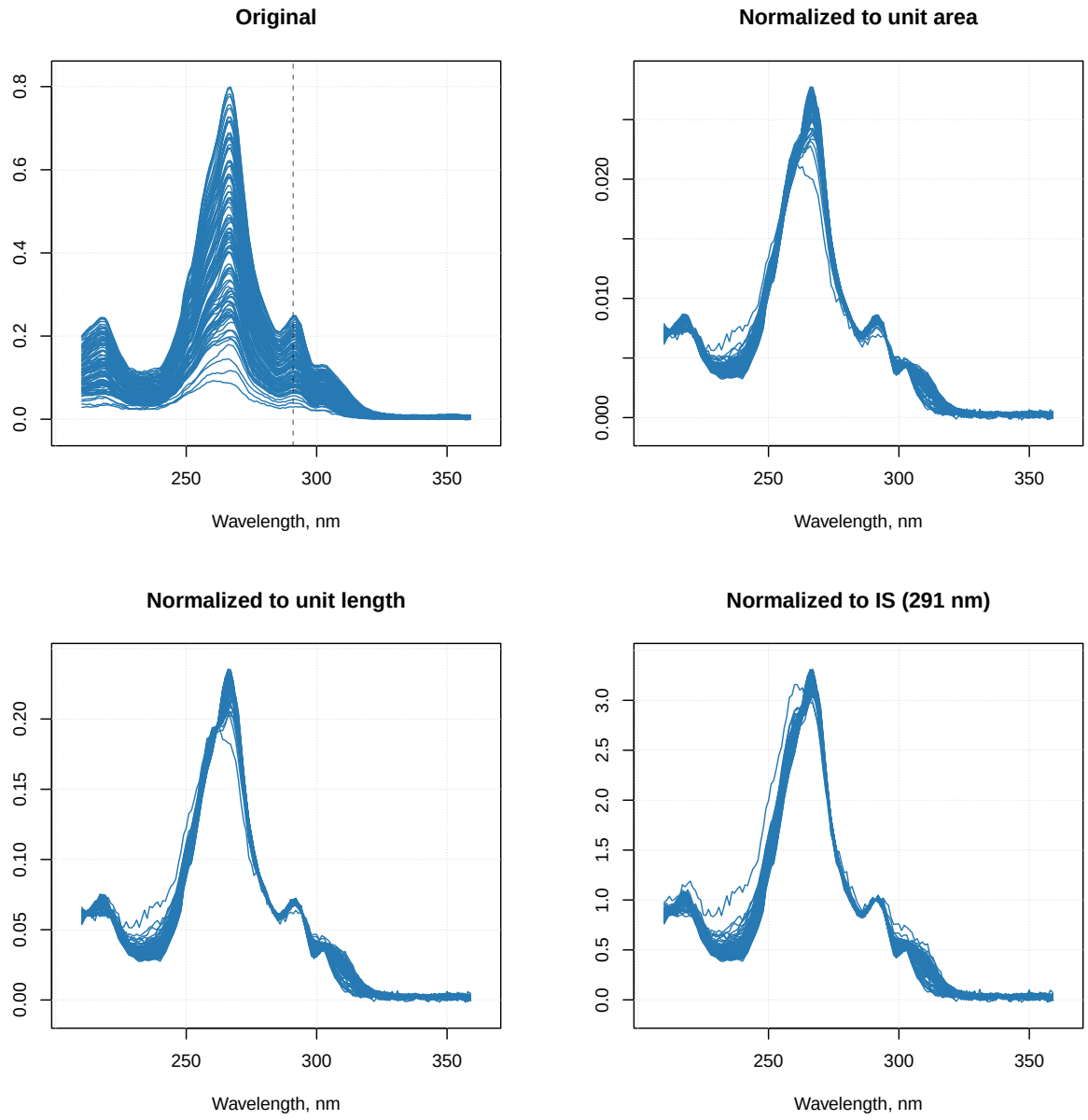
Normalization is a preprocessing which is applied to rows of the dataset (e.g. individual spectra or abundance values for individual measurements) in order to make all rows meet the same requirement. Depending on the requirement different normalization types exist. In *mdatools* the following methods are implemented:

- `"area"` — normalize every row to unit area under measurement points. The area is computed as a sum of absolute values from each row.
- `"sum"` — makes all values of each row sum up to one (similar to `"area"` but it takes sum of the original values, not the absolute ones).
- `"length"` — normalize every row, so if row is represented as a vector in variable space, this vector will have a unit Eucledian length.
- `"snv"` — Standard Normal Variate, makes all values from the same row to have zero mean and unit standard deviation (is described also in previous chapter as this normalization is often used for correction of baseline in spectral data).
- `"is"` — Internal standard normalization. This is common for spectroscopic data, the values from each row will be normalized so value for a given variable (or sum of values for several variables) are equal to one. Usually the variable corresponds to characteristic peak of an internal standard.
- `"pqn"` — Probabilistic Quotient Normalization, a method described in this paper.

All methods are implemented in function `prep.norm` you just need to provide the name of the method as a second argument (or named argument `type`). Internal standard normalization also requires an additional argument, which specifies index of column (or several columns), corresponding to the internal standard. The Probabilistic Quotient Normalization requires reference spectrum as an additional argument.

An example below shows how to make normalization for several selected types using the Simdata:

```
# get spectral data and wavelength
data(simdata)
w = simdata$wavelength

X1 = simdata$spectra.c
attr(X1, "xaxis.values") = w
attr(X1, "xaxis.name") = "Wavelength, nm"

# unit area normalization
X2 = prep.norm(X1, "area")
# unit length normalization
X3 = prep.norm(X1, "length")
# internal standard normalization for variable (column) with wavelength 291 nm
X4 = prep.norm(X1, "is", col.ind = match(291, w))

# show the original and preprocessed spectra
par(mfrow = c(2, 2))

mdaplot(X1, type = "l", main = "Original")
abline(v = 291, lty = 2, col = "#00000080")

mdaplot(X2, type = "l", main = "Normalized to unit area")
mdaplot(X3, type = "l", main = "Normalized to unit length")
mdaplot(X4, type = "l", main = "Normalized to IS (291 nm)")
```



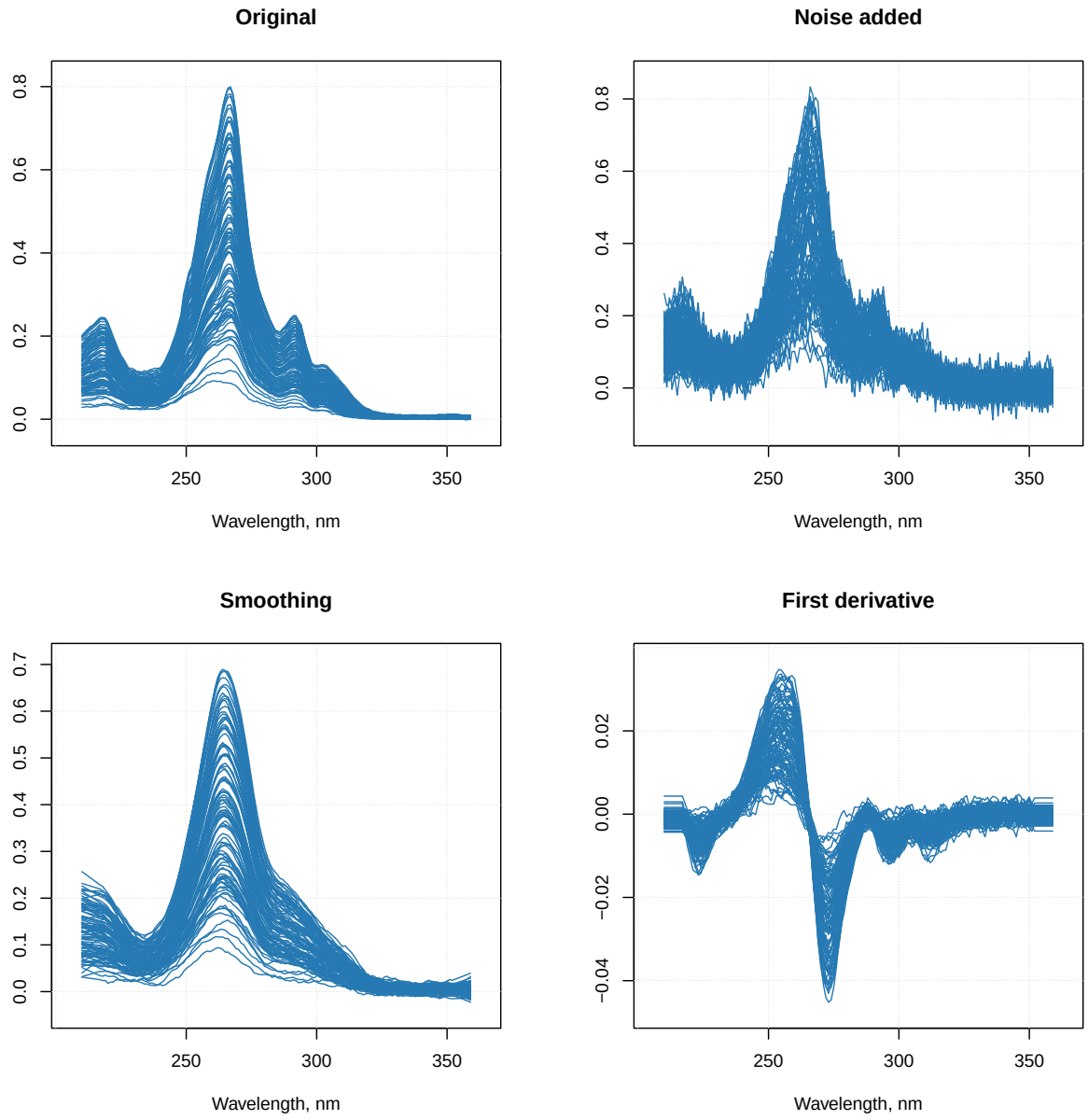
The vertical dashed line on the first plot shows position of the peak, which we use as internal standard for preprocessing shown on the last plot.

Smoothing and derivatives

Savitzky-Golay filter is used to smooth signals and calculate derivatives. The filter has three arguments: a width of the filter (`width`), a polynomial order (`porder`) and the derivative order (`dorder`). If the derivative order is zero (default value) only smoothing will be performed.

The next chunk of code takes the spectra from *Simdata*, adds additional random noise using random numbers generator for normal distribution and then applies the SG filter with different settings. The results are shown as plots under the chunk.

```r
# load UV/Vis spectra from Simdata
data(simdata)
ospectra = simdata$spectra.c
attr(ospectra, "xaxis.values") = simdata$wavelength
attr(ospectra, "xaxis.name") = "Wavelength, nm"

# add random noise to the spectra
nspectra = ospectra + 0.025 * matrix(rnorm(length(ospectra)), dim(ospectra))

# apply SG filter for smoothing
pspectra = prep.savgol(nspectra, width = 15, porder = 1)

# apply SG filter for smoothing and take a first derivative
dpspectra = prep.savgol(nspectra, width = 15, porder = 1, dorder = 1)

# show results
par(mfrow = c(2, 2))
mdaplot(ospectra, type = "l", main = "Original")
mdaplot(nspectra, type = "l", main = "Noise added")
mdaplot(pspectra, type = "l", main = "Smoothing")
mdaplot(dpspectra, type = "l",main = "First derivative")
```



Starting from *v.0.12.0* the algorithm has been modified in order to treat the end points better (for example, when a derivative is taken the end points may look a bit weird and have to be truncated). The implemented algorithm is based on the method described in this article.

Element wise transformations

Function `prep.transform()` allows you to apply element wise transformation — when the same transformation function is being applied to each element (each value) of the data matrix. This can be used, for example, in case of regression, when it is necessary to apply transformations which remove a non-linear relationship between predictors and responses.

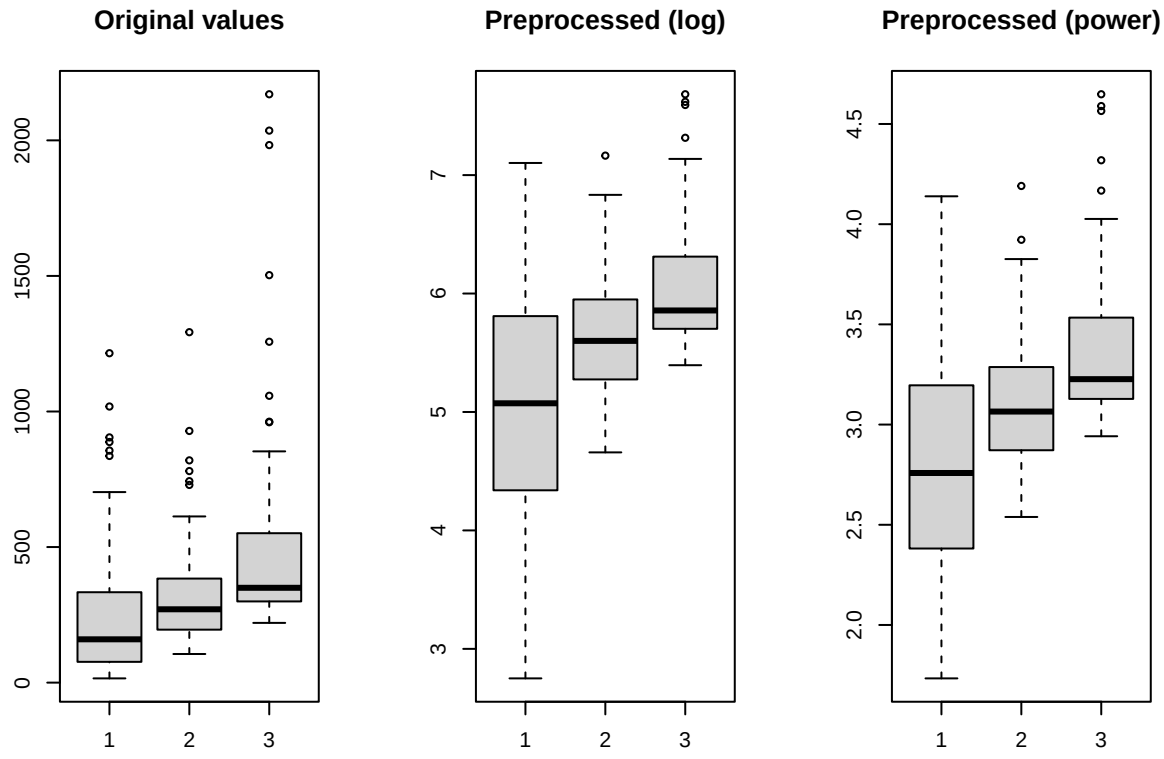
Often such transformation is either a logarithmic or a power. We can of course just apply a built-in R function e.g. `log()` or `sqrt()`, however in this case all additional attributes will be dropped in the preprocessed data. In order to tackle this and, also, to give a possibility for combining different preprocessing methods together,

you can use a function `prep.transform()` for this purpose.

The syntax of the function is following: `prep.transform(data, fun, ...)`, where `data` is a matrix with the original data values, you want to preprocess (transform), `fun` is a reference to transformation function and `...` are optional additional arguments for the function. You can provide either one of the R functions, which are element wise (meaning the function is being applied to each element of a matrix), such as `log`, `exp`, `sqrt`, etc. or define your own function.

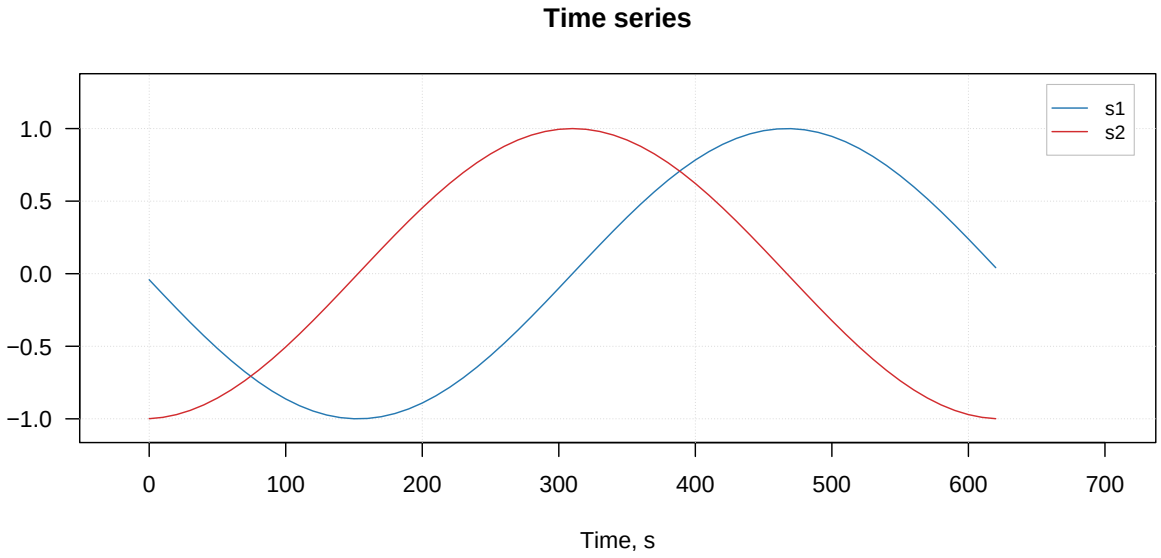
Here is an example:

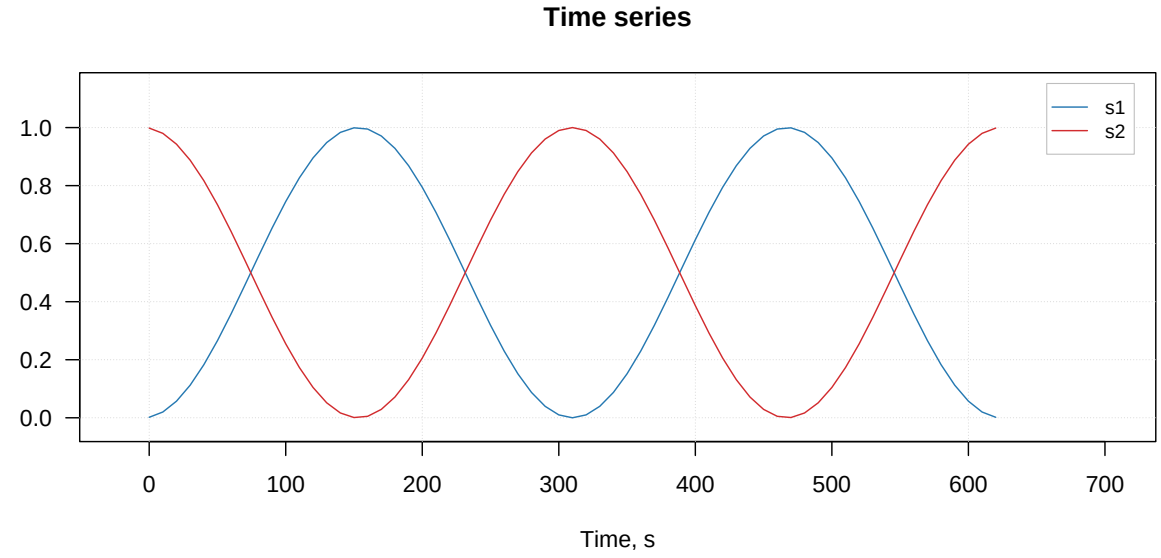
```
# create a matrix with 3 variables (skewed random values)
X <- cbind(
   exp(rnorm(100, 5, 1)),
   exp(rnorm(100, 5, 1)) + 100 ,
   exp(rnorm(100, 5, 1)) + 200
)

# apply log transformation using built in "log" function
Y1 <- prep.transform(X, log)

# apply power transformation using manual function with additional argument
Y2 <- prep.transform(X, function(x, p) x^p, p = 0.2)

# show boxplots for the original and the transformed data
par(mfrow = c(1,3))
boxplot(X, main = "Original values")
boxplot(Y1, main = "Preprocessed (log)")
boxplot(Y2, main = "Preprocessed (power)")
```



As already mentioned, the `prep.transform()` preserves all additional attributes, e.g. names and values for axes, excluded columns or rows, etc. Here is another example demonstrating this:

```
# generate two curves using sin() and cos() and add some attributes
t <- (-31:31)/10
X <- rbind(sin(t), cos(t))
rownames(X) <- c("s1", "s2")

# we make x-axis values as time, which span a range from 0 to 620 seconds
attr(X, "xaxis.name") <- "Time, s"
attr(X, "xaxis.values") <- (t * 10 + 31) * 10
attr(X, "name") <- "Time series"

# transform the dataset using squared transformation
Y <- prep.transform(X, function(x) x^2)

# show plots for the original and the transformed data
par(mfrow = c(2, 1))
mdaplotg(X, type = "l")
mdaplotg(Y, type = "l")
```





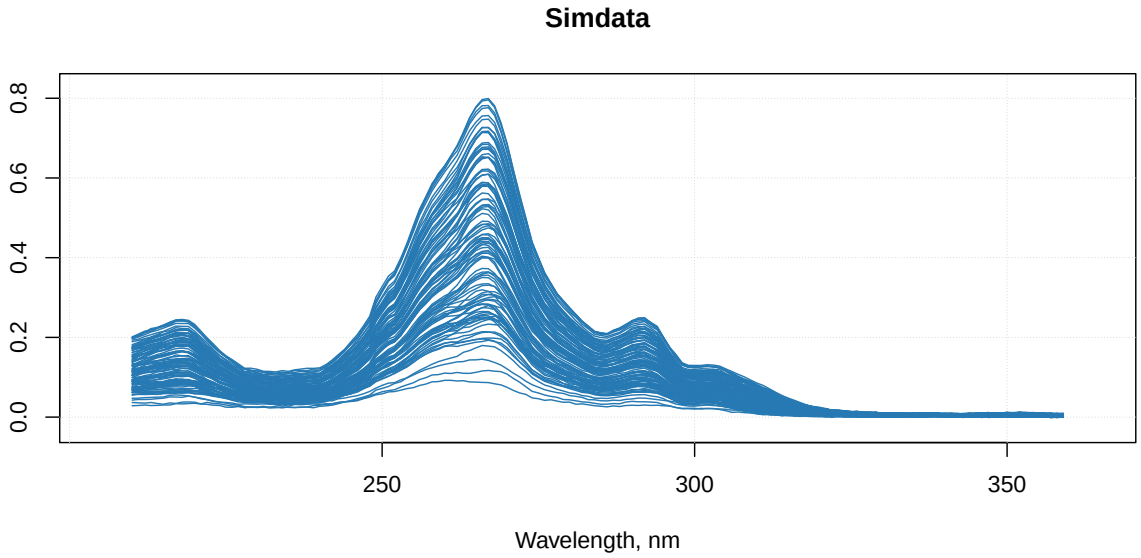
Notice, that the x-axis values for the original and the transformed data (which we defined using corresponding attribute) are the same.

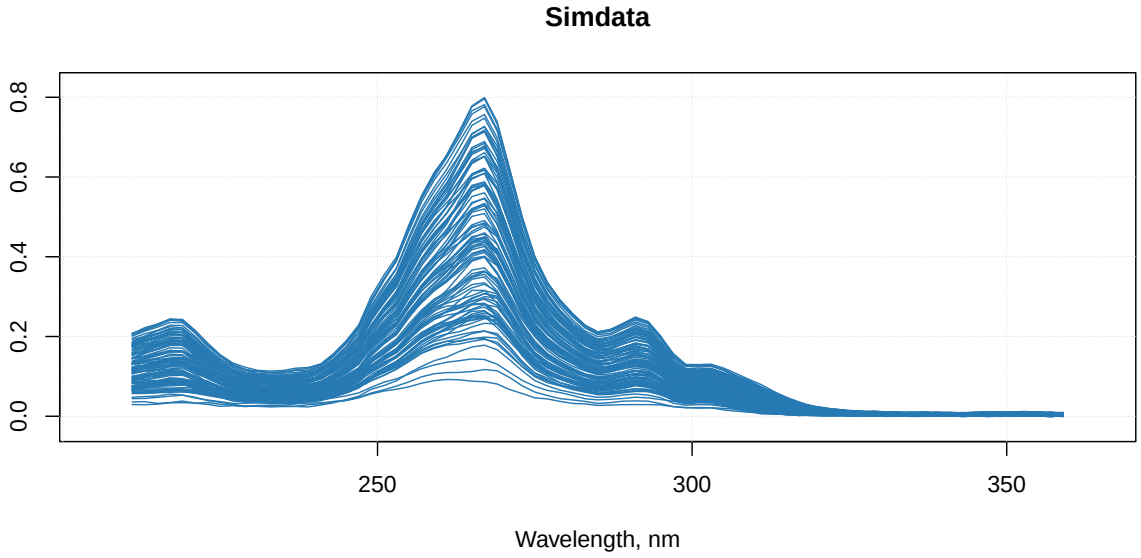
Variable selection as preprocessing method

Variable selection can be done by using `mda.exclcols()`, which simply hides variables/columns, which must not be taken into account in calculations, or by `mda.subset()` which selects only desired columns and remove the rest. Both methods preserve all additional attributes assigned to the data.

The method `prep.varsel()` is simply a wrapper, which allows selection of only desired variables (similar to `mda.subset()`) but can be also incorporated into preprocessing workflow (see next section for details). In the example below it is used to select only even columns from the data matrix.

```r
# load spectra from the Simdata and add some attributed
data(simdata)
X <- simdata$spectra.c
attr(X, "xaxis.values") <- simdata$wavelength
attr(X, "xaxis.name") <- "Wavelength, nm"
attr(X, "name") <- "Simdata"

# apply variable selection as preprocessing
Y <- prep.varsel(X, seq(2, ncol(X), by = 2))

# show both original and preprocessed spectra
par(mfrow = c(2, 1))
mdaplot(X, type = "l")
mdaplot(Y, type = "l")
```





You can notice that on the second plot the lines are not smooth anymore as the number of points is twice smaller.

Combining methods together

From *v.0.12.0* it is possible to combine several preprocessing methods and their parameters into one R object (list) and then apply them all at once in a correct order. This is particularly useful when you create a model based on a preprocessed calibration set and then want to apply the model to a new data. Which means the new data must be preprocessed in the same way as the data you used to create the model. The new functionality makes this easier.

First of all, you can see the list of preprocessing methods available for this feature as well as all necessary information about them if you run `prep.list()` as shown below:

```
prep.list()
```

```
##
##
## List of available preprocessing methods:
##
##
##  Standard normal variate normalization.
##  ---------------
##  name: 'snv'
##  no parameters required
##
##
##  Transform reflectance spectra to Kubelka-Munk units.
##  ---------------
##  name: 'ref2km'
##  no parameters required
##
##
##  Multiplicative scatter correction.
##  ---------------
##  name: 'msc'
##  parameters:
##   'mspectrum': mean spectrum (if NULL will be computed based on data).
##
##
##  Transformation of data values using math functions (log, sqrt, etc.).
##  ---------------
##  name: 'transform'
##  parameters:
##   'fun': function to transform the values (e.g. 'log')
##
##
##  Select user-defined variables (columns of dataset).
##  ---------------
##  name: 'varsel'
##  parameters:
##   'var.ind': indices of variables (columns) to select.
##
##
##  Normalization.
##  ---------------
##  name: 'norm'
##  parameters:
##   'type': type of normalization ('area', 'sum', 'length', 'is', 'snv', 'pqn').
##   'col.ind': indices of columns (variables) for normalization to internal standard peak.
##
##
##  Savitzky-Golay filter.
##  ---------------
##  name: 'savgol'
##  parameters:
##   'width': width of the filter.
##   'porder': polynomial order.
```

```
##   'dorder': derivative order.
##
##
##  Asymmetric least squares baseline correction.
##  ---------------
##  name: 'alsbasecorr'
##  parameters:
##   'plambda': power of the penalty parameter (e.g. if plambda = 5, lambda = 10^5)
##   'p': assymetry ratio (should be between 0 and 1)
##   'max.niter': maximum number of iterations
##
##
##  ---------------
##  name: 'autoscale'
##  parameters:
##   'center': a logical value or vector with numbers for centering.
##   'scale': a logical value or vector with numbers for weighting.
##   'max.cov': columns with coefficient of variation (in percent) below `max.cov` will not be scaled
```

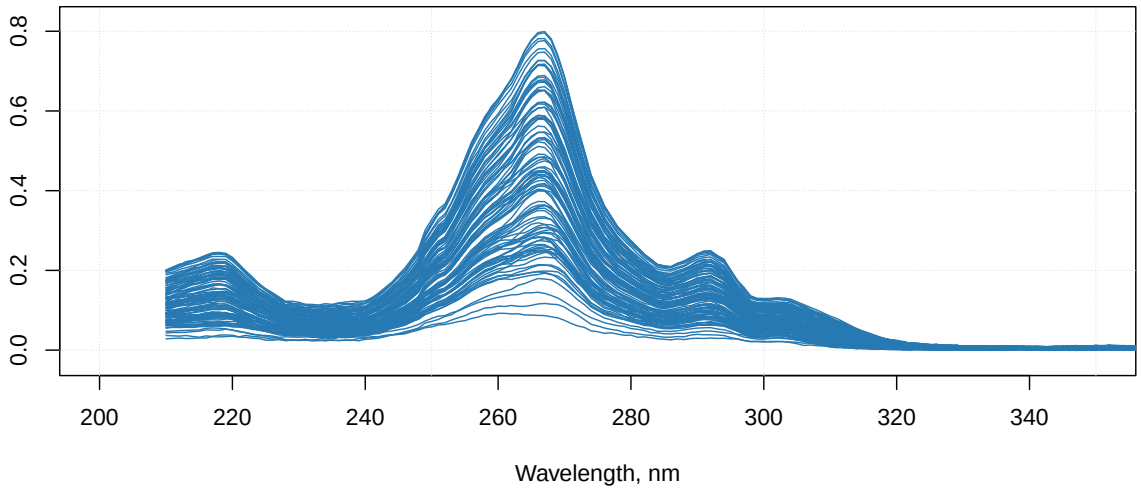
What you need to know is the name of the method and which parameters you can or want to provide (if you do not specify any parameters, default values will be used instead).

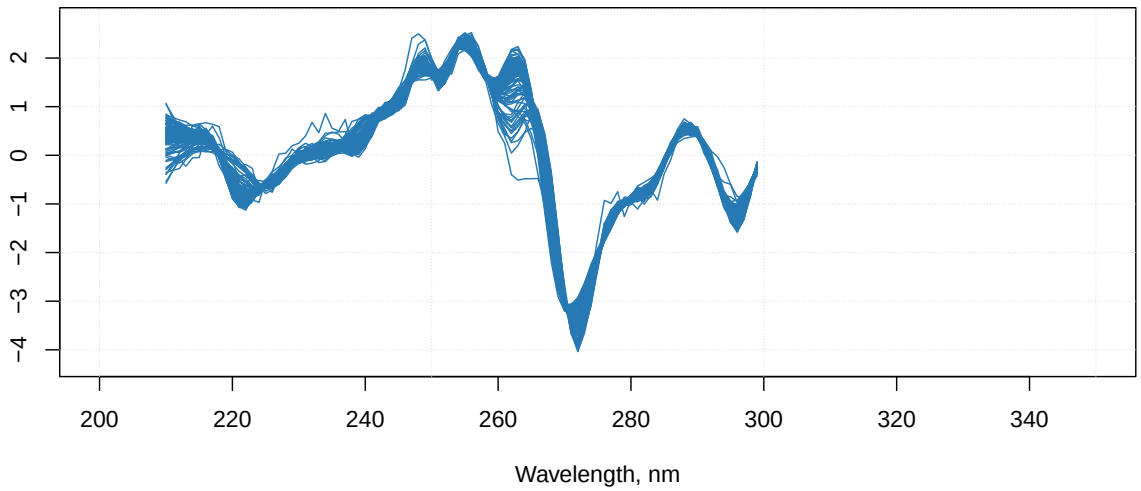
Let's start with a simple example, where we want to take spectra from *Simdata*, and, first of all, apply Savitzky-Golay with filter width = 5, polynomial degree = 2 and derivative order = 1. Then we want to normalize them using SNV and get rid of all spectral values from 300 nm and above. Here is how to make a preprocessing object for this sequence:

```
data(simdata)
w <- simdata$wavelength

myprep <- list(
   prep("savgol", list(width = 5, porder = 2, dorder = 1)),
   prep("snv"),
   prep("varsel", list(var.ind = w < 300))
)
```

Now you can apply the whole sequence to any spectral data by using function `employ.prep()`:

```
Xc <- simdata$spectra.c
attr(Xc, "xaxis.values") <- w
attr(Xc, "xaxis.name") <- "Wavelength, nm"

Xcp <- employ.prep(myprep, Xc)

par(mfrow = c(2, 1))
mdaplot(Xc, type = "l", xlim = c(200, 350))
mdaplot(Xcp, type = "l", xlim = c(200, 350))
```





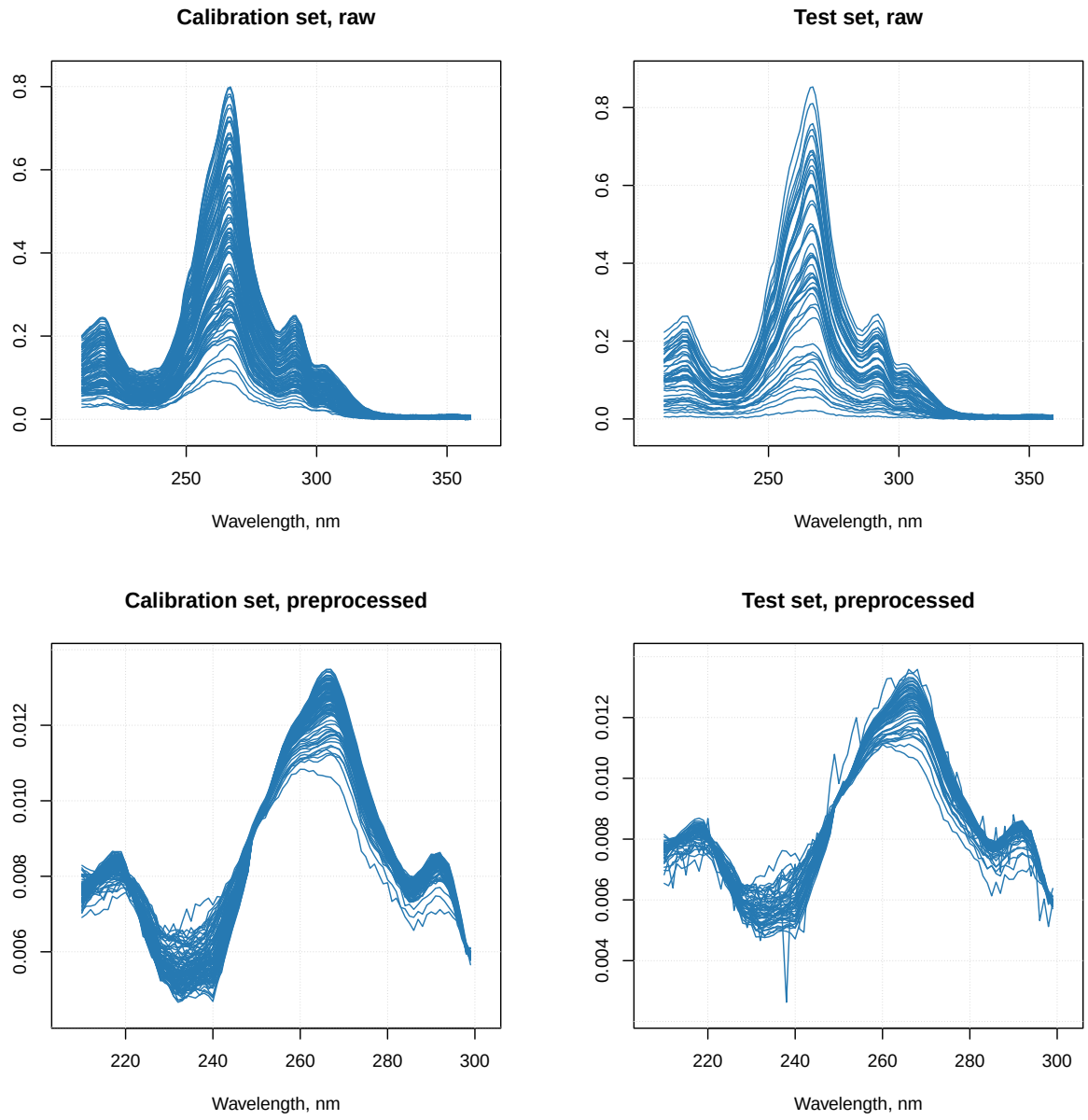
Now let's consider another example, where we have a calibration set and a test set. We need to create preprocessing sequence for the calibration set and, let's say, we ended up with the following sequence:

- Reflectance to absorbance transformation
- MSC correction
- Normalization to unit area
- Removing the part with wavelength > 300 nm

We have two issues here. First of all, there is no implemented method which allows you to convert reflectance spectra to absorbance (log(1/R)). Second is that MSC relies on a mean spectrum, it must be computed for the calibration set and then be re-used when apply the correction to a new data. Here is how to solve both:

```
# define calibration and test sets
Xc <- simdata$spectra.c
Xt <- simdata$spectra.t
w <- simdata$wavelength
```

```
attr(Xt, "xaxis.values") <- attr(Xc, "xaxis.values") <- w
attr(Xt, "xaxis.name") <- attr(Xc, "xaxis.name") <- "Wavelength, nm"


# create a function for converting R to A
r2a <- function(data) return(log(1/abs(data)))

# compute mean spectrum for calibration set
ms <- apply(Xc, 2, mean)

# create a sequence of preprocessing methods
myprep <- list(
   prep("r2a", method = r2a),
   prep("msc", list(mspectrum = ms)),
   prep("norm", list(type = "area")),
   prep("varsel", list(var.ind = w < 300))
)

Xcp <- employ.prep(myprep, Xc)
Xtp <- employ.prep(myprep, Xt)

par(mfrow = c(2, 2))
mdaplot(Xc, type = "l", main = "Calibration set, raw")
mdaplot(Xt, type = "l", main = "Test set, raw")
mdaplot(Xcp, type = "l", main = "Calibration set, preprocessed")
mdaplot(Xtp, type = "l", main = "Test set, preprocessed")
```



In this example I use `abs()` inside the function `r2a` to get rid of occasional negative values in the spectra caused by some noise.

As one can see, you can always define your own method by creating a function, whose first argument should always be `data` — like I did it with `r2a`. This function should treat the data as a matrix and return a matrix of the same dimension as the original data. Other parameters are optional. In my example `r2a` does not have any additional parameters therefore I skipped this argument when calling the `prep()` function.

And, as you probably notices, I just provided preliminary computed mean spectrum for the `"msc"` method. Now the object `myprep` can be saved together with a model to `RData` file and be reused when is needed.

Replacing missing values

Finally, there is one more useful method, which is not directly a preprocessing method, but can be considers as such one. This method allows to replace missing values in your dataset with the approximated ones.

The method uses PCA based approach described in this paper. The main idea is that we fit the dataset with a PCA model (e.g. PCA NIPALS algorithm can work even if data contains missing values) and then approximate the missing values as if they were lyining in the PC space.

The method has the same parameters as any PCA model. However, instead of specifying number of components you must specify another parameter, `expvarlim`, which tells how big the portion of variance the model must explain. The default value is `0.95` which corresponds to 95% of the explained variance. You can also specify if data must be centered (default `TRUE`) and scaled/standardized (default `FALSE`). See more details by running `?pca.mvreplace`.

The example below shows a trivial case. First we generate a simple dataset. Then we replace some of the numbers with missing values (`NA`) and then apply the method to approximate them.

```
library(mdatools)

# generate a matrix with correlated variables
s = 1:6
odata = cbind(s, 2*s, 4*s)

# add some noise and labels for columns and rows
set.seed(42)
odata = odata + matrix(rnorm(length(odata), 0, 0.1), dim(odata))
colnames(odata) = paste0("X", 1:3)
rownames(odata) = paste0("O", 1:6)

# make a matrix with missing values
mdata = odata
mdata[5, 2] = mdata[2, 3] = NA

# replace missing values with approximated
rdata = pca.mvreplace(mdata, scale = TRUE)

# show all matrices together
show(round(cbind(odata, mdata, round(rdata, 2)), 3))
```

```
##       X1     X2     X3    X1     X2     X3   X1    X2    X3
## O1 1.137  2.151  3.861 1.137  2.151  3.861 1.14  2.15  3.86
## O2 1.944  3.991  7.972 1.944  3.991     NA 1.94  3.99  7.51
## O3 3.036  6.202 11.987 3.036  6.202 11.987 3.04  6.20 11.99
## O4 4.063  7.994 16.064 4.063  7.994 16.064 4.06  7.99 16.06
## O5 5.040 10.130 19.972 5.040     NA 19.972 5.04 10.21 19.97
## O6 5.989 12.229 23.734 5.989 12.229 23.734 5.99 12.23 23.73
```

```
# show the difference between original and approximated values
show(round(odata - rdata, 3))
```

```
##    X1     X2    X3
## O1  0  0.000 0.000
## O2  0  0.000 0.465
## O3  0  0.000 0.000
## O4  0  0.000 0.000
## O5  0 -0.076 0.000
## O6  0  0.000 0.000
```

As you can see the method guess that the two missing values must be 7.51 and 10.21, while the original values were 7.97 and 10.13.

The method works if total number of missing values does not exceed 20% (10% if the dataset is small).

Principal component analysis

In this chapter we will discuss how to use PCA method implemented in the *mdatools*. Besides that, we will use PCA examples to introduce some principles, which are common for most of the other methods (e.g. PLS, SIMCA, PLS-DA, etc.) available in this package. This includes such things as model and result objects, getting and visualization of performance statistics, validation, use of different kinds of plots, and so on.

Principal component analysis is one of the methods that decompose a data matrix $\mathbf{X}$ into a combination of three matrices: $\mathbf{X} = \mathbf{T}\mathbf{P}^{\mathrm{T}} + \mathbf{E}$. Here $\mathbf{P}$ is a matrix with unit vectors, defined in the original variables space. The unit vectors, also known as *loadings*, form a new basis — *principal components*. The components are mutually orthogonal and oriented in variable space to capture direction of maximum variation of data points.

The data points are being projected to the principal components. Coordinates of these projections in principal component space, known as *scores*, forming matrix $\mathbf{T} = \mathbf{X}\mathbf{P}$. Product of scores and loadings, $\mathbf{T}\mathbf{P}^{\mathrm{T}}$ gives coordinates of the projections in the original variable space. Matrix $\mathbf{E}$ contains residuals — difference between position of projected data points and their original locations. This difference is a part of data variation, which PCA model does not capture, hence the name.

If original data matrix has I rows (observations, objects) and J variables and PCA decomposition is made with A components, then matrix $\mathbf{P}$ will have dimension $J \times A$, matrix $\mathbf{T}$ — $I \times A$, and $\mathbf{T}\mathbf{P}^{\mathrm{T}}$ and $\mathbf{E}$ will have the same dimension as the original data matrix, $\mathbf{X}$.

Relationship between data objects and principal component space (PCA model) can be described using two distances (in some literature they are also called residual distances) — *orthogonal distance*, OD, and *score distance*, SD. The orthogonal distance is a squared Euclidean distance between the position of an object and its projection in original variable space. It can be computed by taking sum of squared values from matrix $\mathbf{E} = \{e_{ij}\}$ along every row:

$$q_i = \sum_{j=1}^{J} e_{ij}^2$$

This distance usually is denoted as Q or q. It can be considered as a lack of fit measure for this particular object.

The score distance is a squared Mahalanobis distance between the projection of the objects and the origin. It is a measure of extremeness of object and usually denoted as h or T^2. The latter is used because Hotelling T^2 distribution is often used to describe the distribution of the score distance values, so in many software the distance is called as *Hotelling T^2 distance*. The distance can be computed using standardized scores (so score values for every component have unit variance).

$$h_i = \sum_{a=1}^{A} \frac{t_{ia}^2}{\lambda_a}$$

Here λ are eigenvalues for the principal components, which correspond to the variance of corresponding scores.

Both score and orthogonal distances are important statistics allowing to assess how well objects are described by PCA model. They can be assessed visually, using the *Distance plot*, — scatter plot where orthogonal distance is plotted against the score distance for particular number of components. In *mdatools* this plot is called as *Residuals plot* due to some historical reasons.

Both distances can be described using theoretical distributions. This helps to identify regular and extreme objects as well as outliers. All details will be shown later in this tutorial.

There are several other methods, which can be used for decomposition of data similar to PCA. Some of them are Projection Pursuit (PP), Independent Component Analysis (ICA) and many others, that work in a similar way. The main difference among the methods is the way they find the orientation of the unit-vectors. Thus, PCA finds them as directions of maximum variance of data points. In addition to that, all PCA loadings are orthogonal to each other. The PP and ICA use other criteria for the orientation of the basis vectors and e.g. for ICA the vectors are not orthogonal.

There are several methods to compute PCA loadings, including Singular Values Decomposition (SVD) and Non-linear Iterative Partial Least Squares (NIPALS). Both methods are implemented in this package and can be selected using `method` argument of main class `pca`. By default SVD is used. In addition, one can use randomized version of the two methods, which can be efficient if data matrix contains large amount of rows. This is explained in the last part of this chapter.

Models and results

In *mdatools*, any method for data analysis, such as PCA, PLS regression, SIMCA classification and so on, can create two types of objects — a model and a result. Every time you build a model you get a *model object*. Every time you apply the model to a dataset you get a *result object*. Thus for PCA, the objects have classes `pca` and `pcares` correspondingly.

Each object includes a list with object's properties (e.g. loadings for model, scores and explained variance for result) and provides a number of methods for visualization and exploration.

Model calibration

Let's see how this works using a simple example — *People* data. We already used this data when was playing with plots, it consists of 32 objects (persons from Scandinavian and Mediterranean countries, 16 male and 16 female) and 12 variables (height, weight, shoesize, annual income, beer and wine consumption and so on.). More information about the data can be found using `?people`. We will first load the data matrix:

```
library(mdatools)
data(people)
```

Now let's calibrate the model:

```
m = pca(people, 7, scale = TRUE, info = "People PCA model")
m = selectCompNum(m, 5)
```

Here `pca` is a function that builds (calibrates) a PCA model and returns the model object. In terms of programming we can call function `pca()` a constructor of pca model object. The constructor has one mandatory argument — matrix or data frame with data values. Other arguments are optional as they all have default values. In this case we use three additional: `7` is maximum number of components, `scale = TRUE` tells that data values should be standardized (centering is a default option) and `info` allows to add a short text with information about the model which can be shown later.

Function `selectCompNum()` allows to select an "optimal" number of components for the model. In our case we calibrate model with 7 principal components however, e.g. after investigation of explained variance, we found out that 5 components is optimal. In this case we have two choices. Either recalibrate the model using 5 components or use the model that is calibrated already but "tell" the model that 5 components is the

optimal number. In this case the model will keep all results calculated for 7 components but will use optimal number of components when necessary. For example, when showing distance plot for the model. Or when PCA model is used in SIMCA classification.

Function `print` prints the model object info:

```
print(m)
```

```
##
## PCA model (class pca)
##
##
## Call:
## pca(x = people, ncomp = 7, scale = TRUE, info = "People PCA model")
##
## Major model fields:
## $loadings - matrix with loadings
## $eigenvals - eigenvalues for components
## $ncomp - number of calculated components
## $ncomp.selected - number of selected components
## $center - values for centering data
## $scale - values for scaling data
## $alpha - significance level for critical limits
## $gamma - significance level for outlier limits
## $Qlim - critical values and parameters for orthogonal distances
## $T2lim - critical values and parameters for score distances
## $res - list with model results (calibration, test)
```

As you can see, there are no scores, explained variance values, distances and so on. Because they actually are not part of a PCA model, they are results of applying the model to a calibration set. But loadings, eigenvalues, number of calculated and selected principal components, vectors for centering and scaling the data, significance levels and corresponding critical limits for distances — all are parameters of the model.

Here is an example how to get values from loading matrix having the model object:

```
m$loadings[1:4, 1:4]
```

```
##              Comp 1     Comp 2      Comp 3      Comp 4
## Height   -0.3752858 -0.1354585 -0.07237117  0.07431853
## Weight   -0.3811357 -0.1114472 -0.06810860 -0.03313762
## Hairleng  0.3377735  0.1501634  0.07901752  0.11431601
## Shoesize -0.3776970 -0.1508061 -0.00127806  0.06569227
```

One can also notice that the model object has a particular field — `res`, which is in fact a list of PCA *result objects* containing results of applying the model to the calibration set (`cal`) and (if available) validation/test set (`test`). In case of regression and classification `res` can also contain object with cross-validation results (`cv`). For example here is the description of object with calibration results:

```
print(m$res$cal)
```

```
##
## Results for PCA decomposition (class pcares)
##
## Major fields:
## $scores - matrix with score values
## $T2 - matrix with T2 distances
## $Q - matrix with Q residuals
## $ncomp.selected - selected number of components
```

```
## $expvar - explained variance for each component
## $cumexpvar - cumulative explained variance
```

And if we want to look at scores, here is the way:

```
m$res$cal$scores[1:4, 1:4]
```

```
##           Comp 1    Comp 2     Comp 3     Comp 4
## Lars   -5.332528 0.6765698  1.0673076  1.1008056
## Peter  -3.114319 0.2934016 -0.6707091 -1.3099103
## Rasmus -2.996848 0.3604689 -0.2118207 -1.1169333
## Lene    1.084240 1.8449249 -0.4091936 -0.1228956
```

Both model and result objects also have related functions (methods), first of all for visualizing various values (e.g. scores plot, loadings plot, etc.). Some of the functions will be discussed later in this chapter, a full list can be found in help for a proper constructor (e.g. `?pca`).

The *result object* is also created every time you apply a model to a new data. Like in many built-in R methods, method `predict()` is used in this case. The first argument of the method is always a model object. Here is a PCA example (assuming we have already built the model):

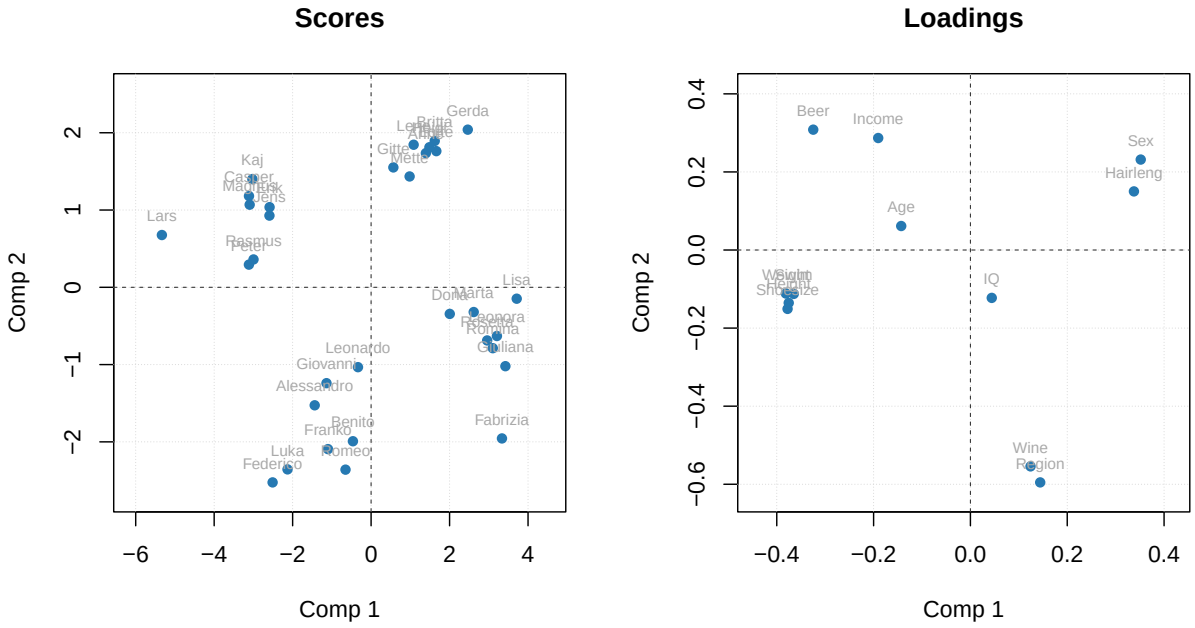
```
# create data matrix with three new measurements
Xt = rbind(
   c(180, 80, -1, 44, 35, 35000, 200, 100, -1, 96, -1, 105),
   c(170, 67,  1, 40, 41, 29500, 100, 200,  1, 94,  1, 105),
   c(175, 70,  1, 41, 28, 21500, 110, 120,  1, 97,  1, 115)
)

# project the new data to the previous PCA model and show the result object
res = predict(m, Xt)
print(res)
```

```
##
## Results for PCA decomposition (class pcares)
##
## Major fields:
## $scores - matrix with score values
## $T2 - matrix with T2 distances
## $Q - matrix with Q residuals
## $ncomp.selected - selected number of components
## $expvar - explained variance for each component
## $cumexpvar - cumulative explained variance
```

Plotting methods

First of all you can use the methods `mdaplot()` and `mdaplotg()` (or any others, e.g. `ggplot2`) for easy visualisation of the results as they are all available as matrices with proper names, attributes, etc. In the example below I create scores and loadings plots for PC1 vs PC2. Here I assume that the model from previous section is already created and available in your Global Environment. As you can see I take matrix with scores from the objects with calibration results (`m$res$cal`).

```
par(mfrow = c(1, 2))
mdaplot(m$res$cal$scores, type = "p", show.labels = TRUE, show.lines = c(0, 0))
mdaplot(m$loadings, type = "p", show.labels = TRUE, show.lines = c(0, 0))
```



To simplify this routine, every model and result class also has a number of functions for visualization. Thus for PCA the function list includes scores and loadings plots, explained variance and cumulative explained variance plots, distance/residuals plots, and many others.

A function that does the same for different models and results has always the same name. For example, `plotPredictions` will show predicted vs. measured plot for PLS model and PLS result, MLR model and MLR result, SIMCA model and SIMCA result, and so on, although the plot itself will be quite different. The first argument must always be either a model or a result object.

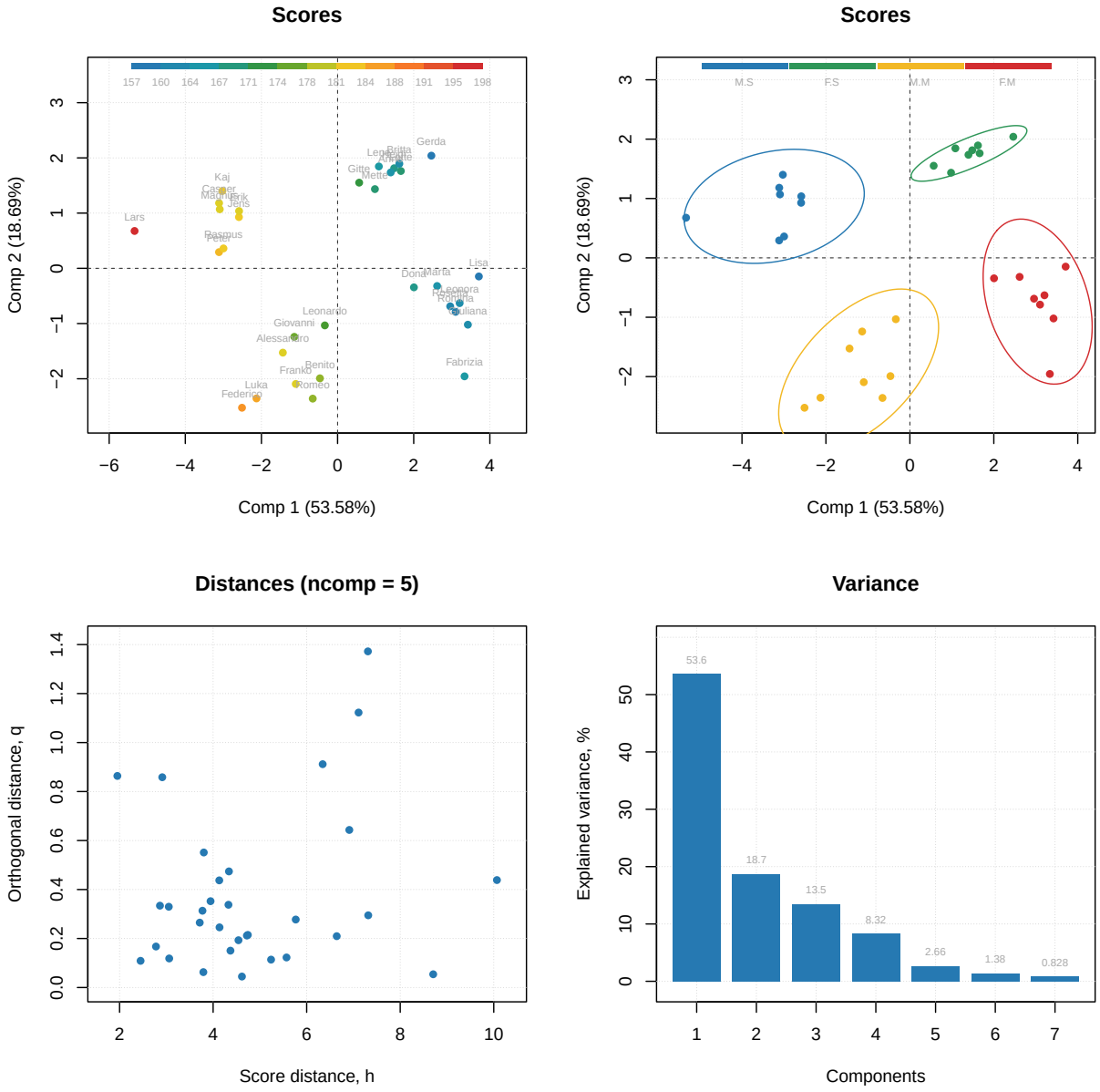
The major difference between plots for model and plots for result is following. A plot for result always shows one set of data objects — one set of points, lines or bars. For example, predicted vs. measured values for calibration set or scores values for test set and so on. For such plots method `mdaplot()` is used and you can provide any arguments, available for this method (e.g. color group scores for calibration results, add confidence ellipses for scores, etc.).

And a plot for a model in most cases shows several sets of data objects, e.g. predicted values for calibration and validation. In this case, a corresponding method uses `mdaplotg()` and, therefore, you can adjust the plot using arguments described for this method.

Here are some examples for results:

```
# create a factor for combination of Sex and Region values
g1 <- factor(people[, "Sex"], labels = c("M", "F"))
g2 <- factor(people[, "Region"], labels = c("S", "M"))
g <- interaction(g1, g2)

# scores plot for calibration results colored by Height
par(mfrow = c(2, 2))
plotScores(m$res$cal, show.labels = TRUE, cgroup = people[, "Height"])

# scores plot colored by the factor created above and confidence ellipses
p = plotScores(m$res$cal, c(1, 2), cgroup = g)
plotConfidenceEllipse(p)
```

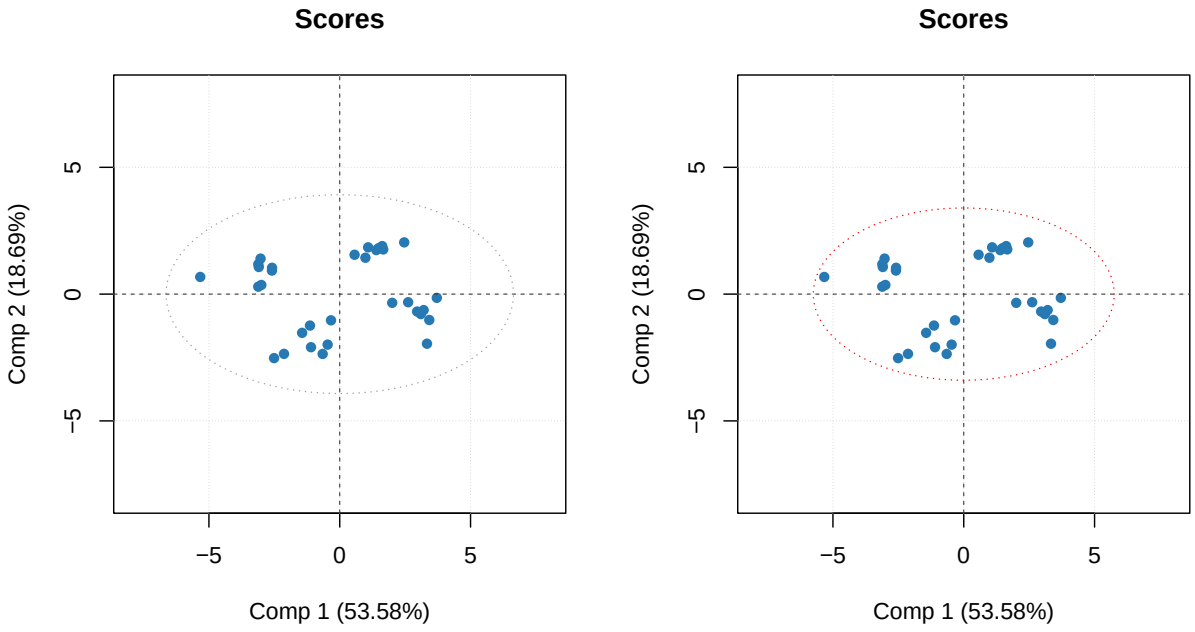
```
# distance plot for calibration results with labels
plotResiduals(m$res$cal, show.labels = TRUE)

# variance plot for calibration results with values as labels
plotVariance(m$res$cal, type = "h", show.labels = TRUE, labels = "values")
```



Scores plot can be also used together with `plotHotellingEllipse()` function. It works similar to `plotConfidenceEllipse()` or `plotConvexHull()` however does not require grouping of the values. You can add the ellipse both to score plot for results and score plot for model. In case of model, if you have e.g. both calibration and test set results you need to specify which one you want to use for the creating the ellipse. Code below shows several examples.

```
par(mfrow = c(1, 2))
```

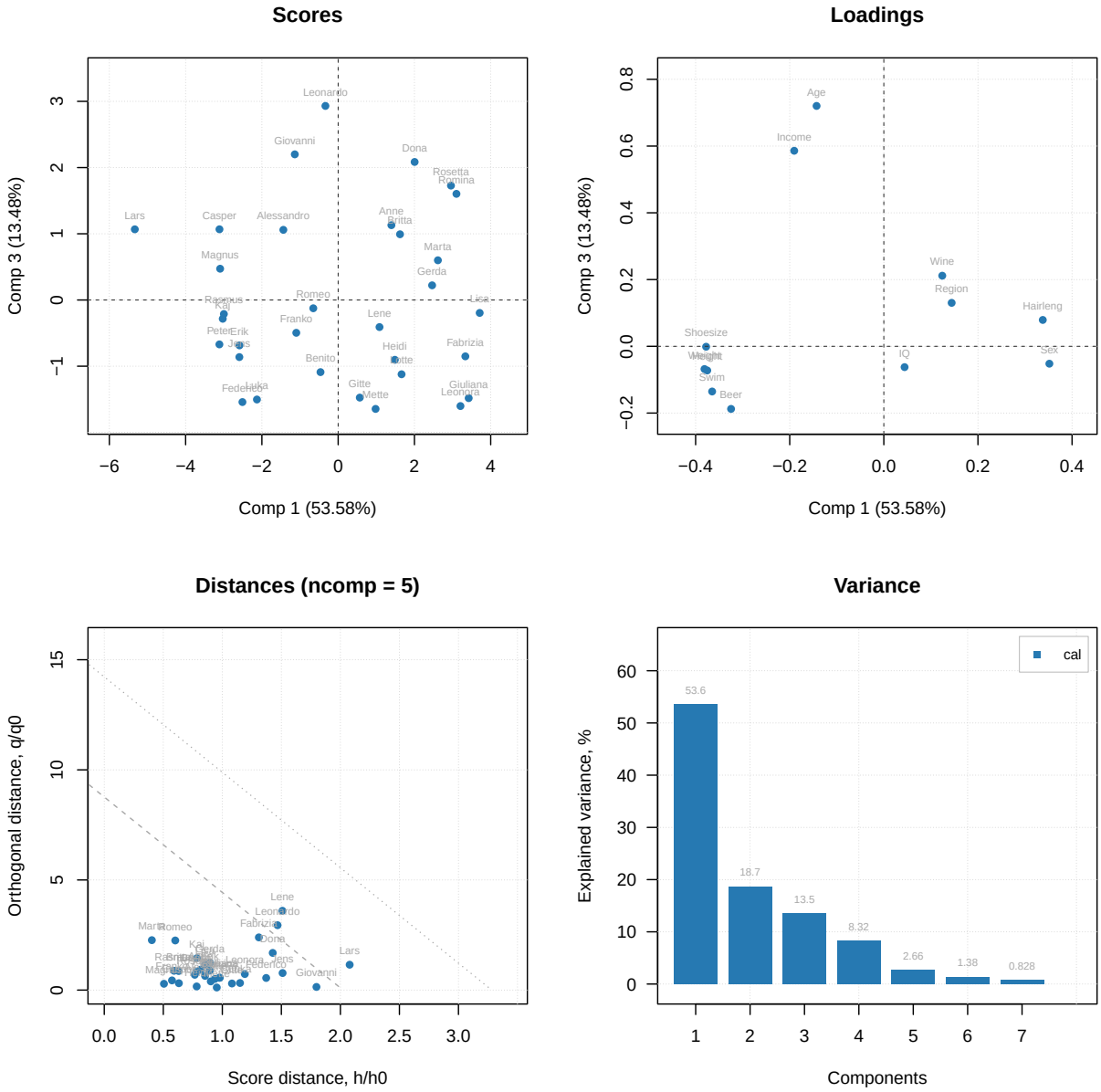
```
# default options for calibration results
p = plotScores(m$res$cal, xlim = c(-8, 8), ylim = c(-8, 8))
plotHotellingEllipse(p)

# also for calibration results but with specific significance limit and color
p = plotScores(m$res$cal, xlim = c(-8, 8), ylim = c(-8, 8))
plotHotellingEllipse(p, conf.lim = 0.9, col = "red")
```



The color grouping option is not available for the group (model) plots as colors are used there to underline the groups.

Now let's look at similar plots (plus loadings) for a model.

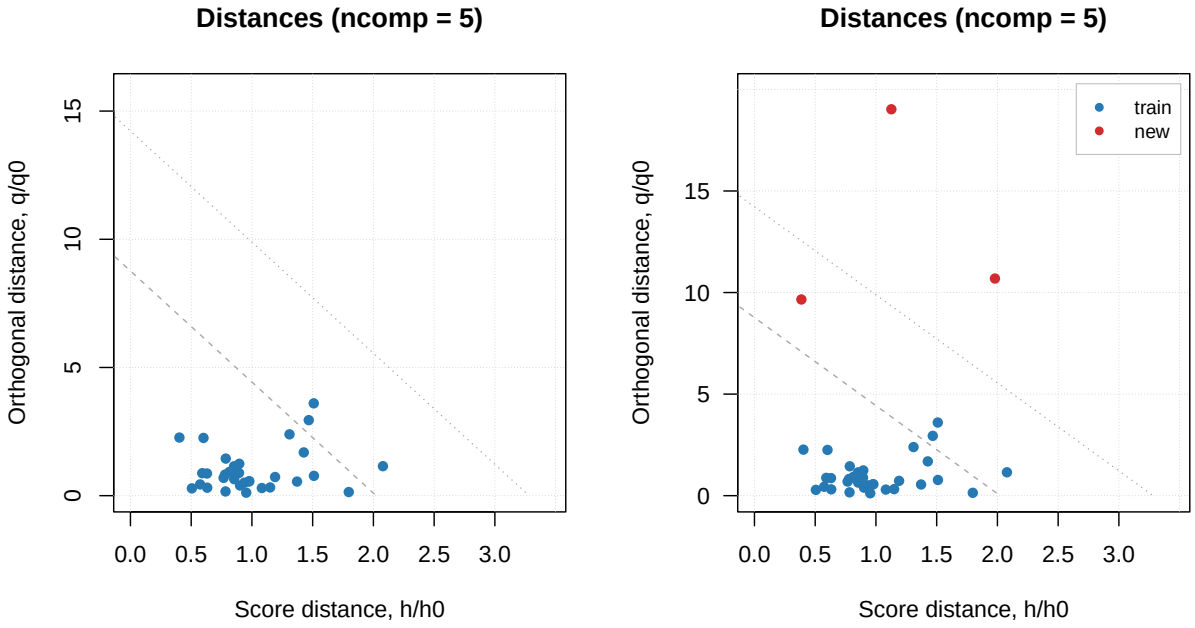
```
par(mfrow = c(2, 2))
plotScores(m, c(1, 3), show.labels = TRUE)
plotLoadings(m, c(1, 3), show.labels = TRUE)
plotResiduals(m, show.labels = TRUE)
plotVariance(m, type = "h", show.labels = TRUE, labels = "values")
```



As you can see, for in case of model, the distance plot also shows some lines. These are critical limits, all details about them as well as how to use the distance/residuals plot will be given in the next section.

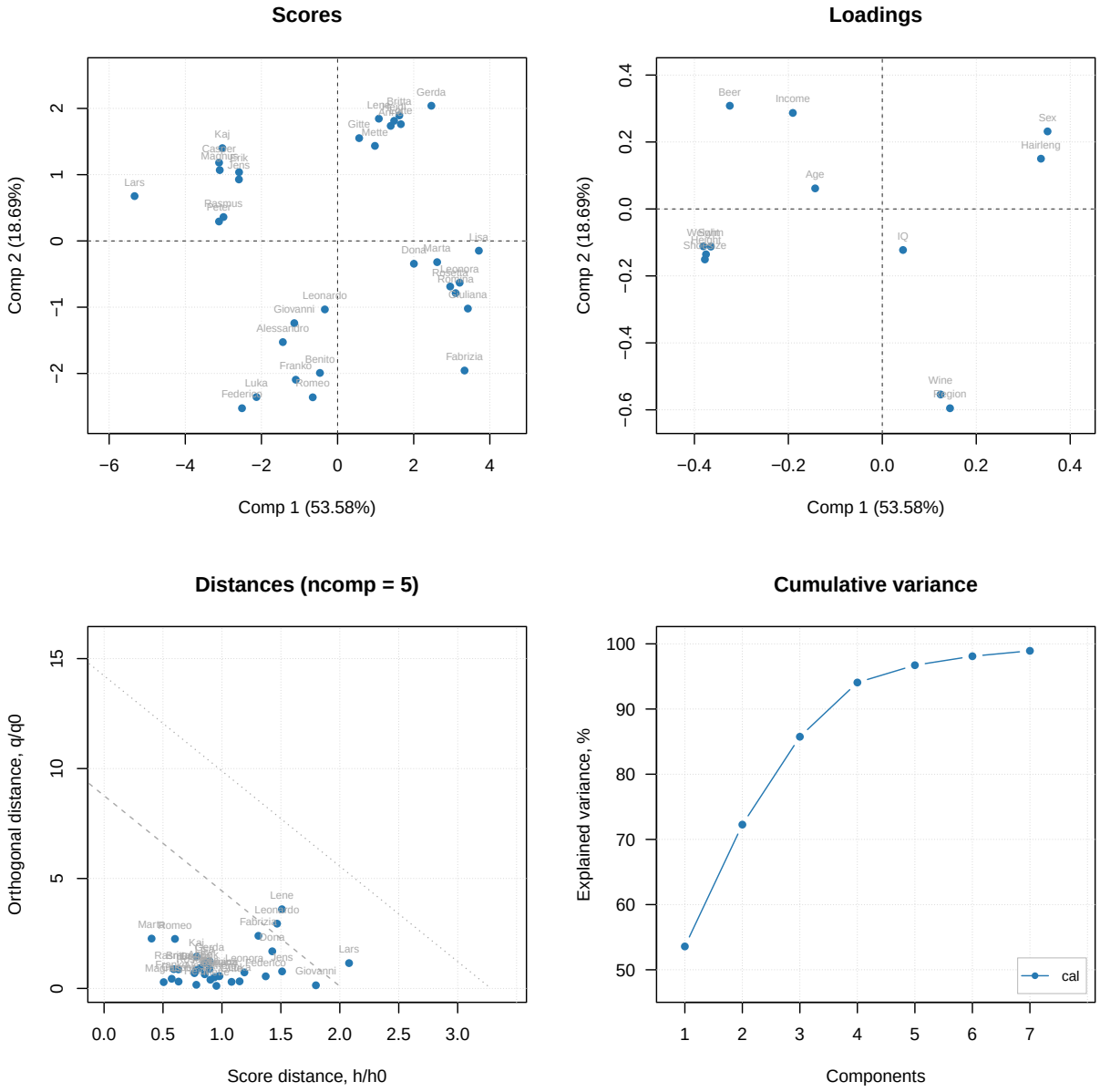
If model plot shows values from calibration and validation objects (e.g. scores, distances, etc), you can specify which result to show. In the example below left plot is made using default settings, for the right plot we specified list with results explicitly (we show calibration results with label "train" and results for new measurements with label "new"). Note, that the results should be specified as a named list (we also assume here that you ran code from the previous part of this section and have object `res` in your environment).

```
par(mfrow = c(1, 2))
plotResiduals(m)
plotResiduals(m, res = list("train" = m$res$cal, "new" = res))
```



Finally, method `plot()` shows the four main PCA plots as a model (or results) overview.

```
plot(m, show.labels = TRUE)
```



You do not have to care about labels, names, legend and so on, however if necessary you can always change almost anything. See full list of methods available for PCA by `?pca` and `?pcares`.

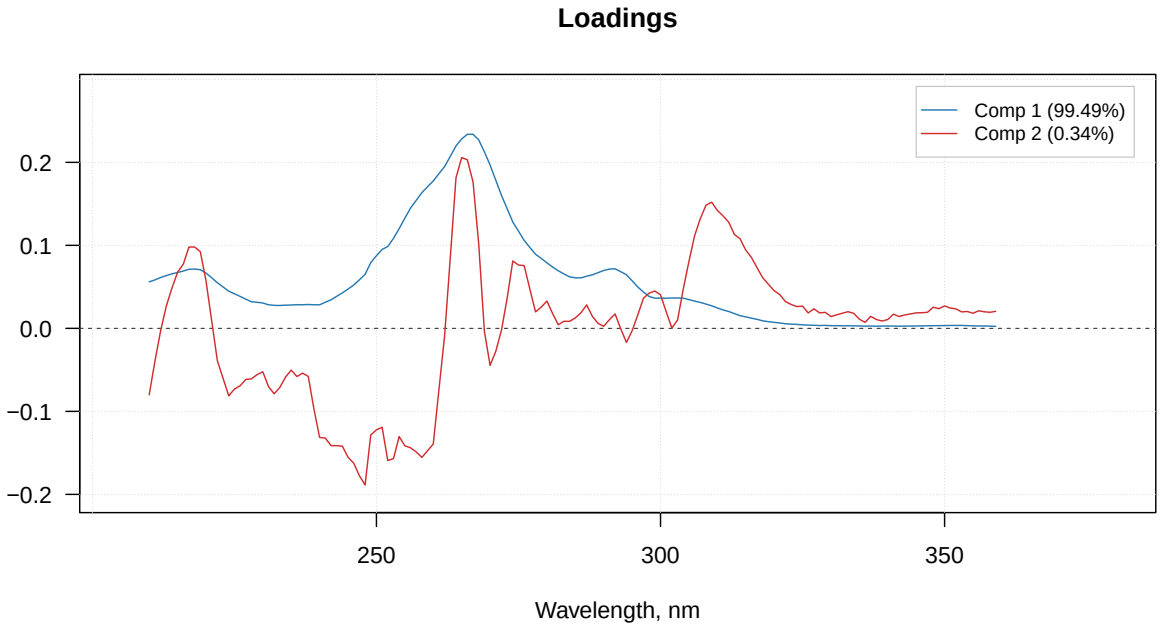
Manual x-values for loading line plot

As it was discussed in the previous chapter, you can specify a special attribute, `"xaxis.values"` to a dataset, which will be used as manual x-values in bar and line plots. When we create any model and/or results the most important attributes, including this one, are inherited. For example when you make a loading line plot it will be shown using the attribute values.

Here is an example that demonstrate this feature using PCA decomposition of the *Simdata* (UV/Vis spectra).

```
data(simdata)

X = simdata$spectra.c
```

```
attr(X, "xaxis.name") = "Wavelength, nm"
attr(X, "xaxis.values") = simdata$wavelength

m = pca(X, 3)
plotLoadings(m, 1:2, type = "l")
```



Excluding rows and columns

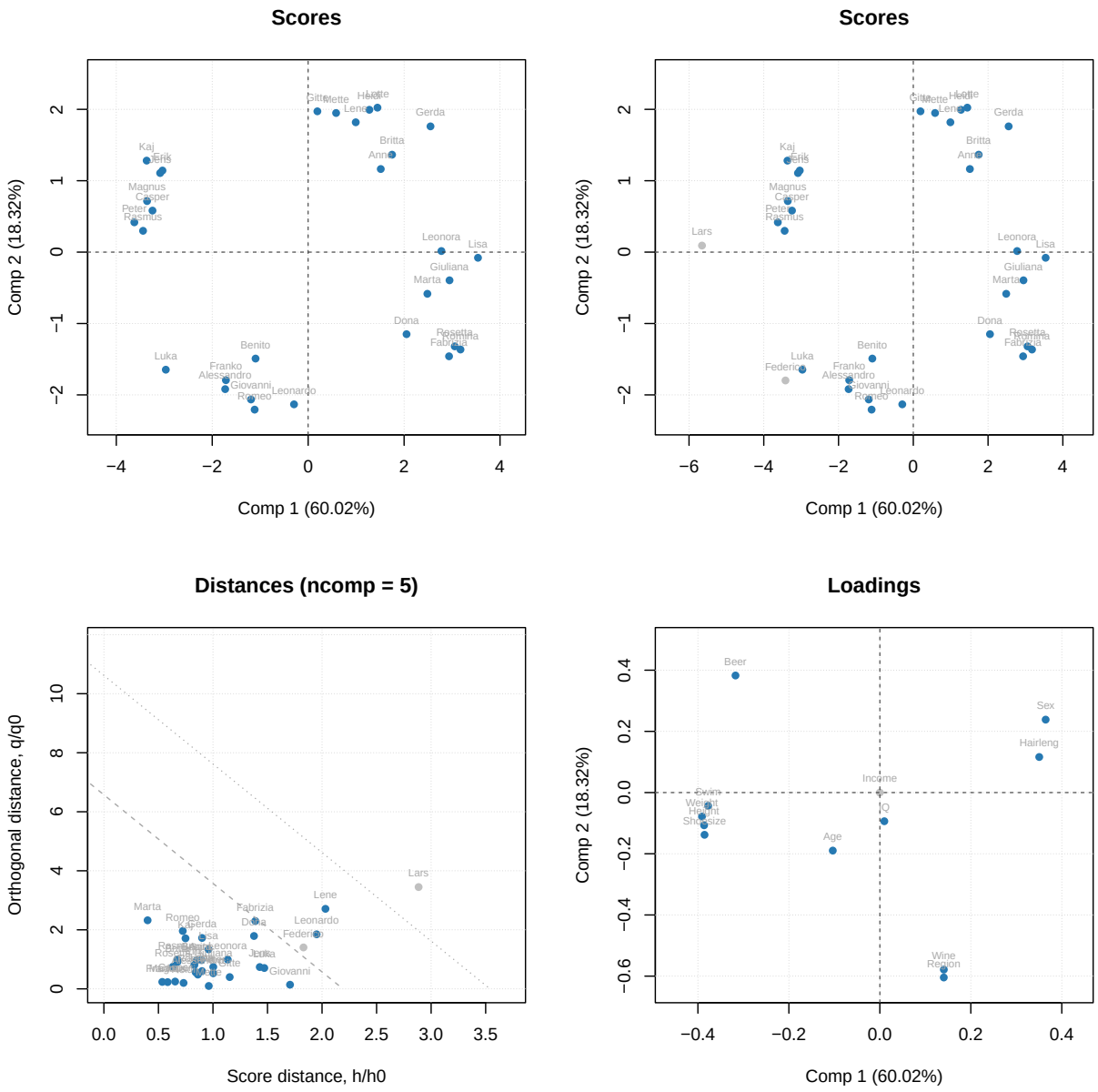
PCA as well as any other method in `mdatools` can exclude rows and columns from calculations. For example, it can be useful if you have some candidates for outliers or do variable selection and do not want to remove rows and columns from the data matrix. In this case you can just specify two additional parameters, `exclcols` and `exclrows`, using either numbers or names of rows/columns to be excluded. You can also specify a vector with logical values (all `TRUE`s will be excluded).

The excluded rows are not used for creating a model and calculation of model's and results' performance (e.g. explained variance). However main results (for PCA — scores and residual distances) are calculated for these rows as well and set hidden, so you will not see them on plots. You can always show values for excluded objects by using parameter `show.excluded = TRUE`. It is implemented via attributes "known" for plotting methods from *mdatools* so if you use e.g. *ggplot2* you will see all points.

The excluded columns are not used for any calculations either, the corresponding results (e.g. loadings or regression coefficients) will have zero values for such columns and be also hidden on plots. Here is a simple example.

```
data(people)

m = pca(people, 5, scale = TRUE, exclrows = c("Lars", "Federico"), exclcols = "Income")

par(mfrow = c(2, 2))
plotScores(m, show.labels = TRUE)
plotScores(m, show.labels = TRUE, show.excluded = TRUE)
plotResiduals(m, show.labels = TRUE, show.excluded = TRUE)
```

```
plotLoadings(m, show.labels = TRUE, show.excluded = TRUE)
```



As you can see, the excluded values (or variables in case of loadings plot) are either hidden completely or shown as gray points if `show.excluded` parameter is set to `TRUE`.

Here is the matrix with loadings, note that variable `Income` has zeros for loadings and the matrix has attribute `exclrows` set to 6 (which is a position of the variable):

```
# show matrix with loadings (look at row Income and attribute "exclrows")
show(m$loadings)
```

```
##                Comp 1      Comp 2       Comp 3      Comp 4      Comp 5
## Height    -0.386393327 -0.10697019 -0.004829174 -0.12693029 -0.13128331
## Weight    -0.391013398 -0.07820097  0.051916032 -0.04049593 -0.14757465
## Hairleng   0.350435073  0.11623295 -0.103852349  0.04969503 -0.73669997
```

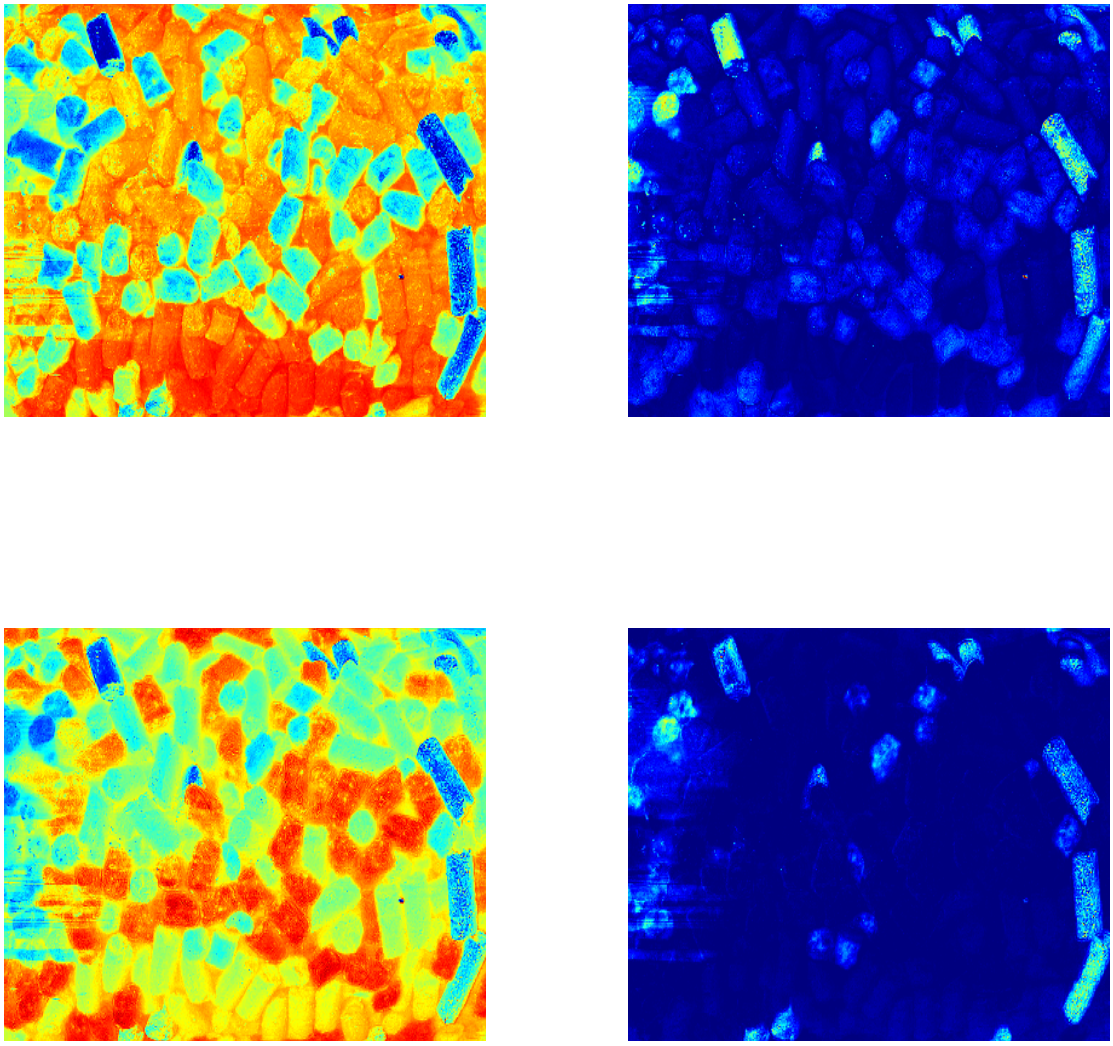
```
## Shoesize -0.385424793 -0.13805817 -0.069172117 -0.01049098 -0.17075488
## Age      -0.103466285 -0.18964288 -0.337243182  0.89254403 -0.02998028
## Income    0.000000000  0.00000000  0.000000000  0.00000000  0.00000000
## Beer     -0.317356319  0.38259695  0.044338872  0.03908064 -0.21419831
## Wine      0.140711271 -0.57861817 -0.059833970 -0.12347379 -0.41488773
## Sex       0.364537185  0.23838610  0.010818891 -0.04025631 -0.18263577
## Swim     -0.377470722 -0.04330411  0.008151288 -0.18149268 -0.30163601
## Region    0.140581701 -0.60435183  0.040969200 -0.15147464  0.17857614
## IQ        0.009849911 -0.09372132  0.927669306  0.32978247 -0.11815762
## attr(,"exclrows")
## [1] 6
## attr(,"name")
## [1] "Loadings"
## attr(,"xaxis.name")
## [1] "Components"
## attr(,"yaxis.name")
## [1] "Variables"
```

Such behavior will help to exclude and include rows and columns interactively, some examples will be shown later.

Support for images

As it was described before, images can be used as a source of data for any methods. In this case the results, related to objects/pixels will inherit all necessary attributes and can be show as images as well. In the example below we make a PCA model for the image data from the package and show scores and distances.

```
data(pellets)
X = mda.im2data(pellets)
m = pca(X)

par(mfrow = c(2, 2))
imshow(m$res$cal$scores)
imshow(m$res$cal$Q)
imshow(m$res$cal$scores, 2)
imshow(m$res$cal$Q, 2)
```



Model validation

PCA model can be validated using a separate validation set (in some literature *validation set* is also called *a test set*, although usually test set is used only for estimation of performance of the final, optimized model) — measurements for objects taken from the same population as the calibration set but which were not utilized for the model calibration. If you do not have a separate validation set you can generate one using Procrustes cross-validation. This section will show both approaches.

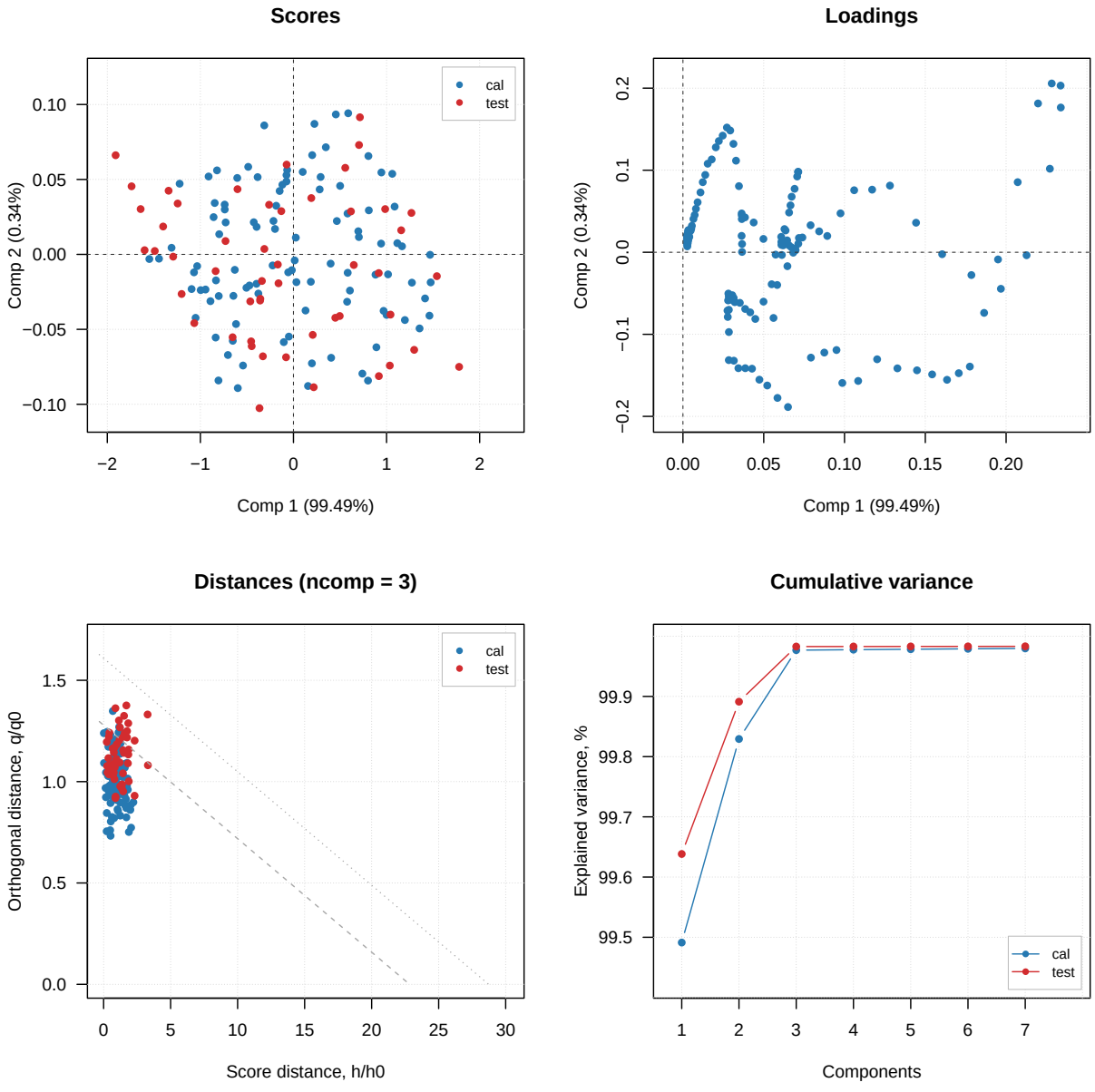
Using validation set

If validation set is available it can be provided to `pca()` function as a value for named parameter `x.test`. In the example above we get UV/Vis spectra from `simdata` datasets. This dataset has spectra for bot calibration and validation/test set. So we use the first set to calbirate a PCA model and then use the second set as

validation/test set. After that we created plot overview for the model which has results from both sets.

```
data(simdata)
Xc = simdata$spectra.c
Xt = simdata$spectra.t
m = pca(Xc, 7, x.test = Xt)
```

```
plot(m, ncomp = 3)
```



The validation results are, of course, represented by *result objects*, which are fields of a *model object* similar to `cal`, but with names `test`. And here is the info for both result objects:

```
print(m$res$cal)
```

```
##
## Results for PCA decomposition (class pcares)
```

```
##
## Major fields:
## $scores - matrix with score values
## $T2 - matrix with T2 distances
## $Q - matrix with Q residuals
## $ncomp.selected - selected number of components
## $expvar - explained variance for each component
## $cumexpvar - cumulative explained variance
```

```
print(m$res$test)
```

```
##
## Results for PCA decomposition (class pcares)
##
## Major fields:
## $scores - matrix with score values
## $T2 - matrix with T2 distances
## $Q - matrix with Q residuals
## $ncomp.selected - selected number of components
## $expvar - explained variance for each component
## $cumexpvar - cumulative explained variance
```

Let us compare, for example, the explained variance values for the results:

```
var = data.frame(
   cal = m$res$cal$expvar,
   test = m$res$test$expvar
)
show(round(var, 1))
```

```
##         cal test
## Comp 1 99.5 99.6
## Comp 2  0.3  0.3
## Comp 3  0.1  0.1
## Comp 4  0.0  0.0
## Comp 5  0.0  0.0
## Comp 6  0.0  0.0
## Comp 7  0.0  0.0
```

Every model and every result object has a method `summary()`, which shows some statistics for evaluation of a model performance. Here are some examples.

```
# summary for whole model
summary(m)
```

```
##
## Summary for PCA model (class pca)
## Type of limits: ddmoments
## Alpha: 0.05
## Gamma: 0.01
##
##        Eigenvals Expvar Cumexpvar  Nq Nh
## Comp 1     0.596  99.49     99.49   4  2
## Comp 2     0.002   0.34     99.83   3  5
## Comp 3     0.001   0.15     99.98 125  7
## Comp 4     0.000   0.00     99.98 116  8
## Comp 5     0.000   0.00     99.98 113 10
```

```
## Comp 6     0.000   0.00     99.98 107 12
## Comp 7     0.000   0.00     99.98 112 12
```

```
# summary for calibration results
summary(m$res$cal)
```

```
##
## Summary for PCA results
##
## Selected components: 7
##
##        Expvar Cumexpvar
## Comp 1  99.49     99.49
## Comp 2   0.34     99.83
## Comp 3   0.15     99.98
## Comp 4   0.00     99.98
## Comp 5   0.00     99.98
## Comp 6   0.00     99.98
## Comp 7   0.00     99.98
```

```
# summary for validation/test results
summary(m$res$test)
```

```
##
## Summary for PCA results
##
## Selected components: 7
##
##        Expvar Cumexpvar
## Comp 1  99.64     99.64
## Comp 2   0.25     99.89
## Comp 3   0.09     99.98
## Comp 4   0.00     99.98
## Comp 5   0.00     99.98
## Comp 6   0.00     99.98
## Comp 7   0.00     99.98
```

The same methodology is used for any other method, e.g. PLS or SIMCA. In the next section we will look at how to use plotting functions for models and results.

Procrustes cross-validation

If you do not have a dedicated validation/test set you can generate one using Procrustes cross-validation. The theory behid the method is available in this paper (it is Open Access), and all additional information can be found in GitHub repository of the project.

In order to use PCV you need to install a dedicated package, `pcv`, first. Here is a code which loads the `pcv` package (here I assume that you have already installed it) and create a PV-set (analogue of validation set generated by PCV) using PCA model.

```
library(pcv)
Xpv = pcvpca(Xc, ncomp = 20, cv = list("ven", 4))
```

The idea behind PCV is as follows. PCV first splits rows of calibration set into several segments. Number of segments and the way to make the split (systematic or random) is defined by a value of parameter `cv`. As you can see in the code above, I used `cv = list("ven", 4)`. This means that I want to create 4 segments (subsets) using systematic split "Venetian blinds". So in the first segment I will have rows number 1, 11, 21,

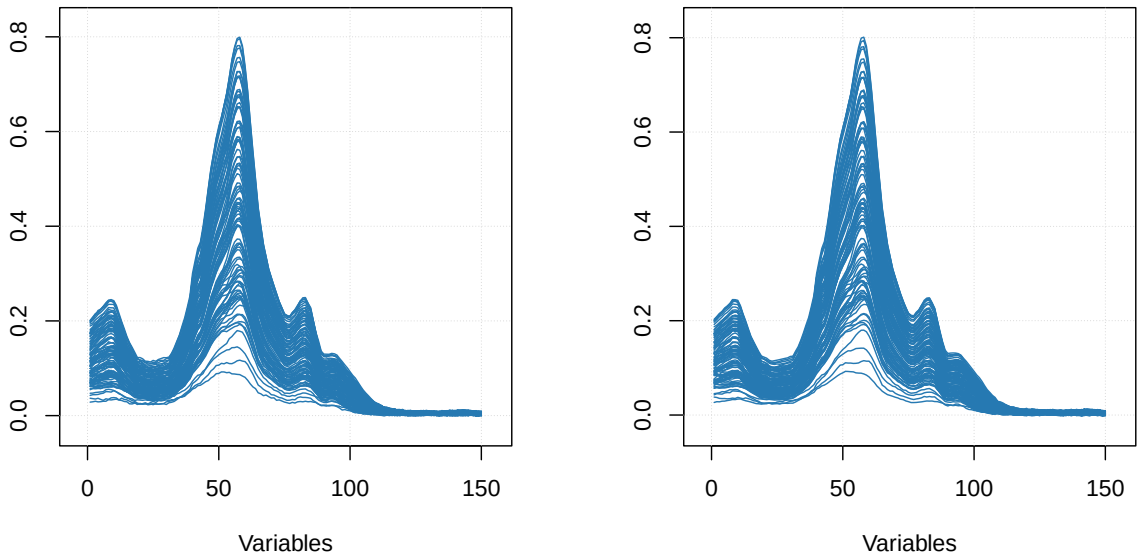
. . . and in the second segment I will have rows: 2, 12, 22, etc.

The idea of splits is similar to cross-validation, and PCV is based on cross-validation resampling. You can also use random splits or leave-one-out splits. But in general Venetial blinds with relatively few segemnts (from 3 to 8) will work best. You also need to speciy number of components in the model, just use any number much larger than the expected optimal number. PCV is not sensitive to overfitting.

After the split is done it creates local PCA models different subsets of your calibration set, measured uncertainties among the local models and then introduces this uncertainty to the calibration set hence creating a new set of measurements.

The plot below compares spectra from the original calbiration set (left) and the generated PV set (right). As you can see, although the shape of individual spectra look similar the spectra are not identical.

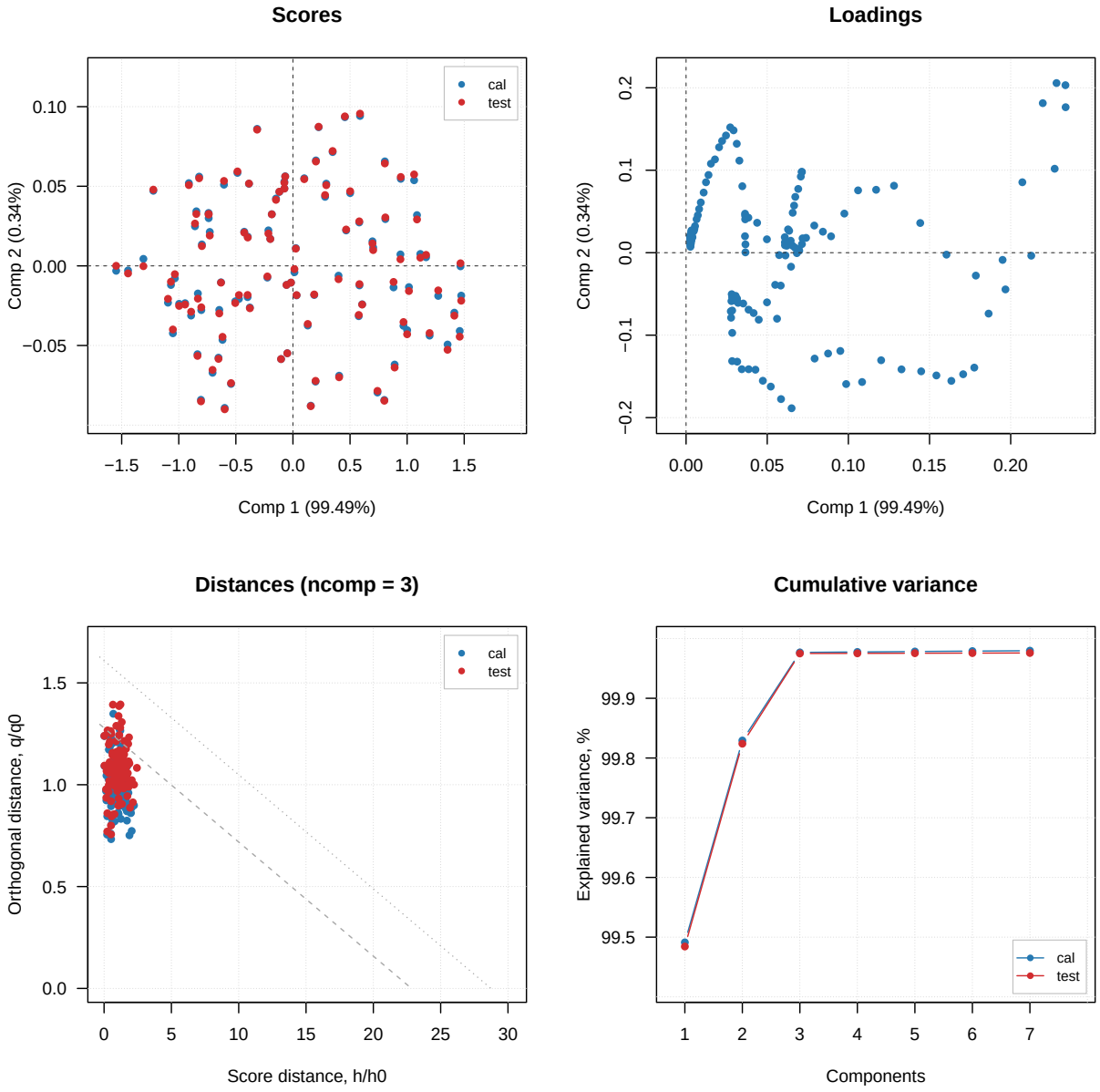
```
par(mfrow = c(1, 2))
mdaplot(Xc, type = "l")
mdaplot(Xpv, type = "l")
```



Now we can create a PCA model and validate using the created PV-set similar to what we have done in case of dedicated validation set:

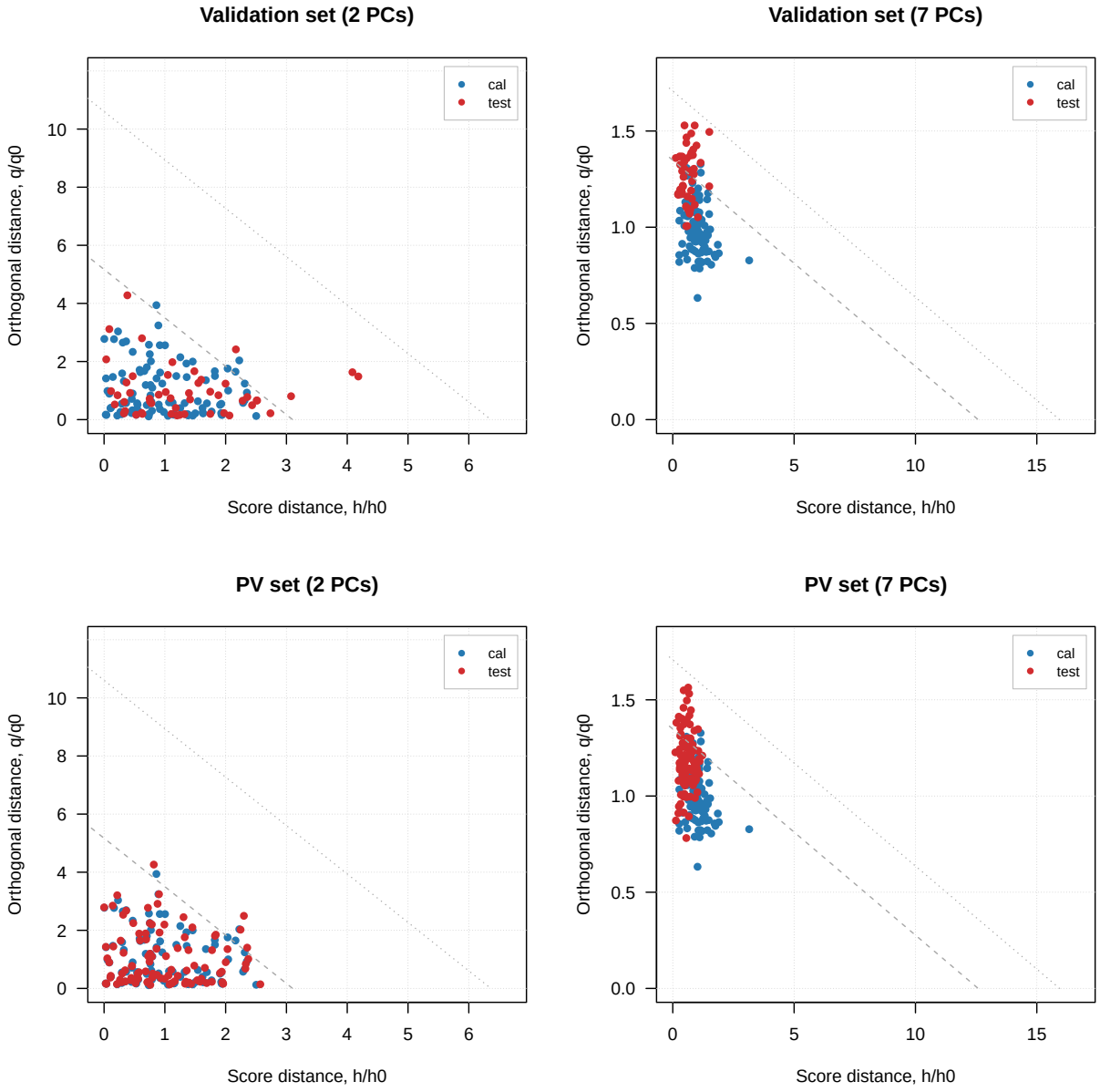
```
m2 = pca(Xc, 7, x.test = Xpv)
```

```
plot(m2, ncomp = 3)
```



The four plots below show distance plots for model validated using separate validation set (top) and the PV-set (bottom). The left plots a made for 2 principal components and the right plots are made for 7 components. As you can see both separate validation set and the one generated using PCV show similar patterns.

```
par(mfrow = c(2, 2))
plotResiduals(m, ncomp = 2, main = "Validation set (2 PCs)")
plotResiduals(m, ncomp = 7, main = "Validation set (7 PCs)")
plotResiduals(m2, ncomp = 2, main = "PV set (2 PCs)")
plotResiduals(m2, ncomp = 7, main = "PV set (7 PCs)")
```



Distances and critical limits

Distance to model and score distance

As it was written in the brief theoretical section about PCA, when data objects are being projected to a principal component space, two distances are calculated and stored as a part of PCA results (`pcares` object). The first is a squared orthogonal Euclidean distance from original position of an object to the PC space, also known as *orthogonal distance* and denoted as Q or q. The distance shows how well the object is fitted by the PCA model and allows to detect objects that do not follow a common trend, captured by the PCs.

The second distance shows how far a projection of an object to the PC space is from the origin. This distance is also known as Hotelling T^2 distance or a *score distance*. To compute T^2 distance scores should be first normalized by dividing them to corresponding singular values (or eigenvalues if we deal with squared scores). The T^2 distance allows to see extreme objects — which are far from the origin. In the package as well as in

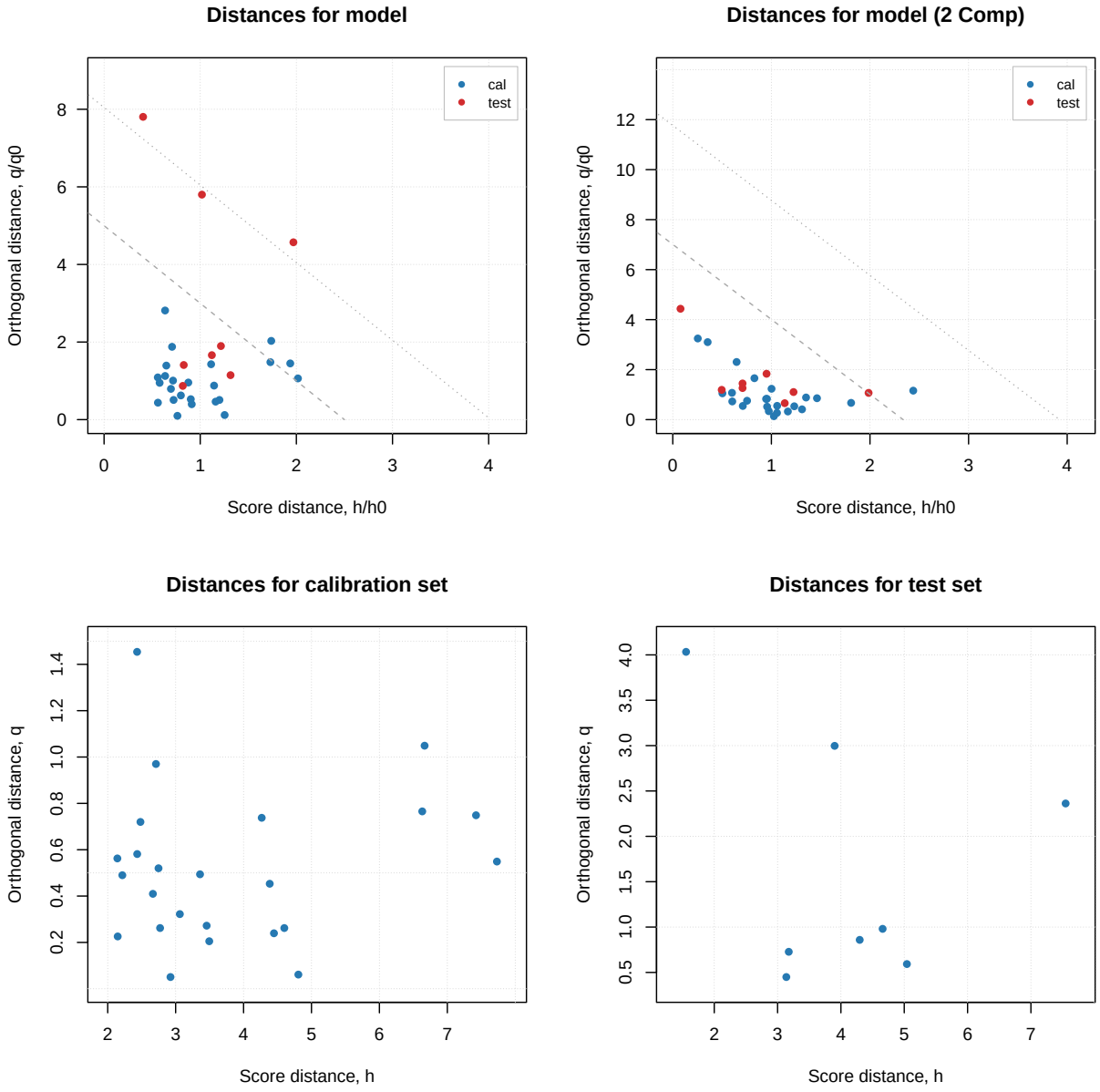
this tutorial we will also use letter h to denote this distance, so Q and q are for orthogonal distance and T^2 and h stand for the score distance.

> Historically, the distances are called in this package and also in some other software as *Residual distances* or just *residuals*. Therefore functions, which visualize the corresponding distances, are called `plotResiduals()` (`plotXResiduals()`, `plotYResiduals()`, `plotXYResiduals()` in PLS based methods). However, this is not fully correct, and in the text of the tutorial we will refer to them as *distances* most of the time. Changing function names though will lead to large degree of incompatibility.

In *mdatools* both distances are calculated for all objects in dataset and all possible components. So every distance is represented by $I \times A$ matrix (will remind here that I is number of objects or rows in data matrix and A is number of components in PCA model), the first column contains distances for a model with only one component, second — for model with two components, and so on. The distances can be visualized using method `plotResiduals()` which is available both for PCA results as well as for PCA model. In the case of model the plot shows distances both for calibration set and test set (if available).

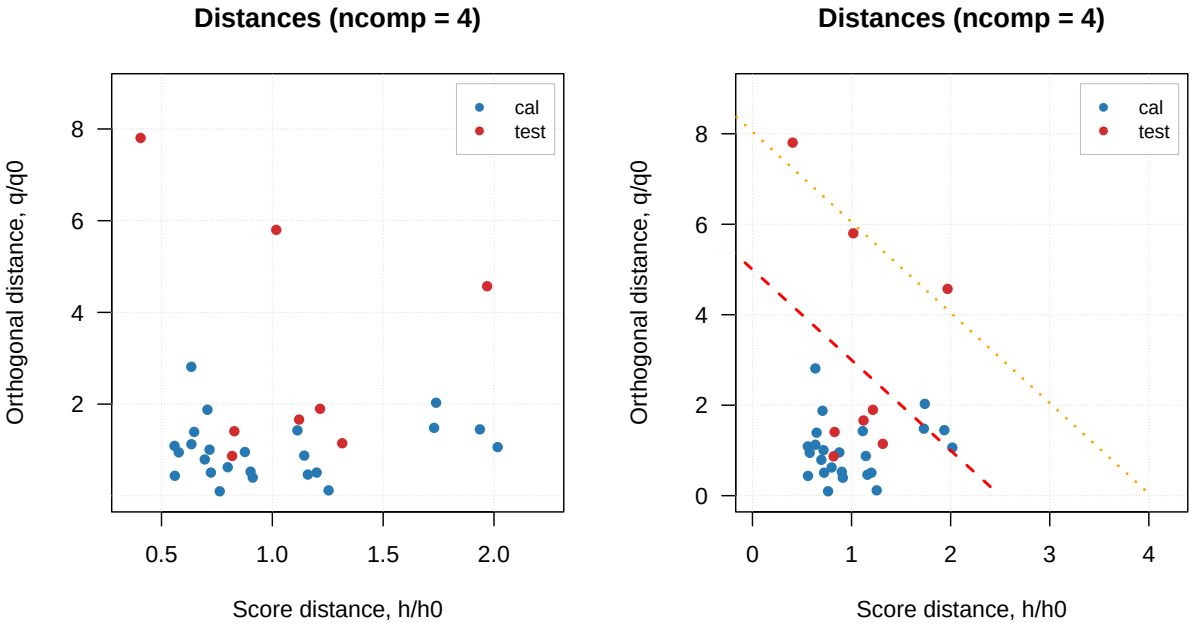
Here is an example, where I split People data to calibration and test set as in one of the examples in previous sections and show distance plot for model and result objects.

```
data(people)
idx = seq(4, 32, 4)
Xc = people[-idx, ]
Xt = people[idx, ]

m = pca(Xc, 4, scale = TRUE, x.test = Xt)

par(mfrow = c(2, 2))
plotResiduals(m, main = "Distances for model")
plotResiduals(m, ncomp = 2, main = "Distances for model (2 Comp)")
plotResiduals(m$res$cal, main = "Distances for calibration set")
plotResiduals(m$res$test, main = "Distances for test set")
```



As it was briefly mentioned before, if residual/distance plot is made for a model, it also shows two critical limits: the dashed line is a limit for extreme objects and the dotted line is a limit for outliers. How they are computed is explained later in this section. If you want to hide them, just use additional parameter `show.limits = FALSE`. You can also change the color, line type and width for the lines by using options `lim.col`, `lim.lty` and `lim.lwd` as it is shown below.

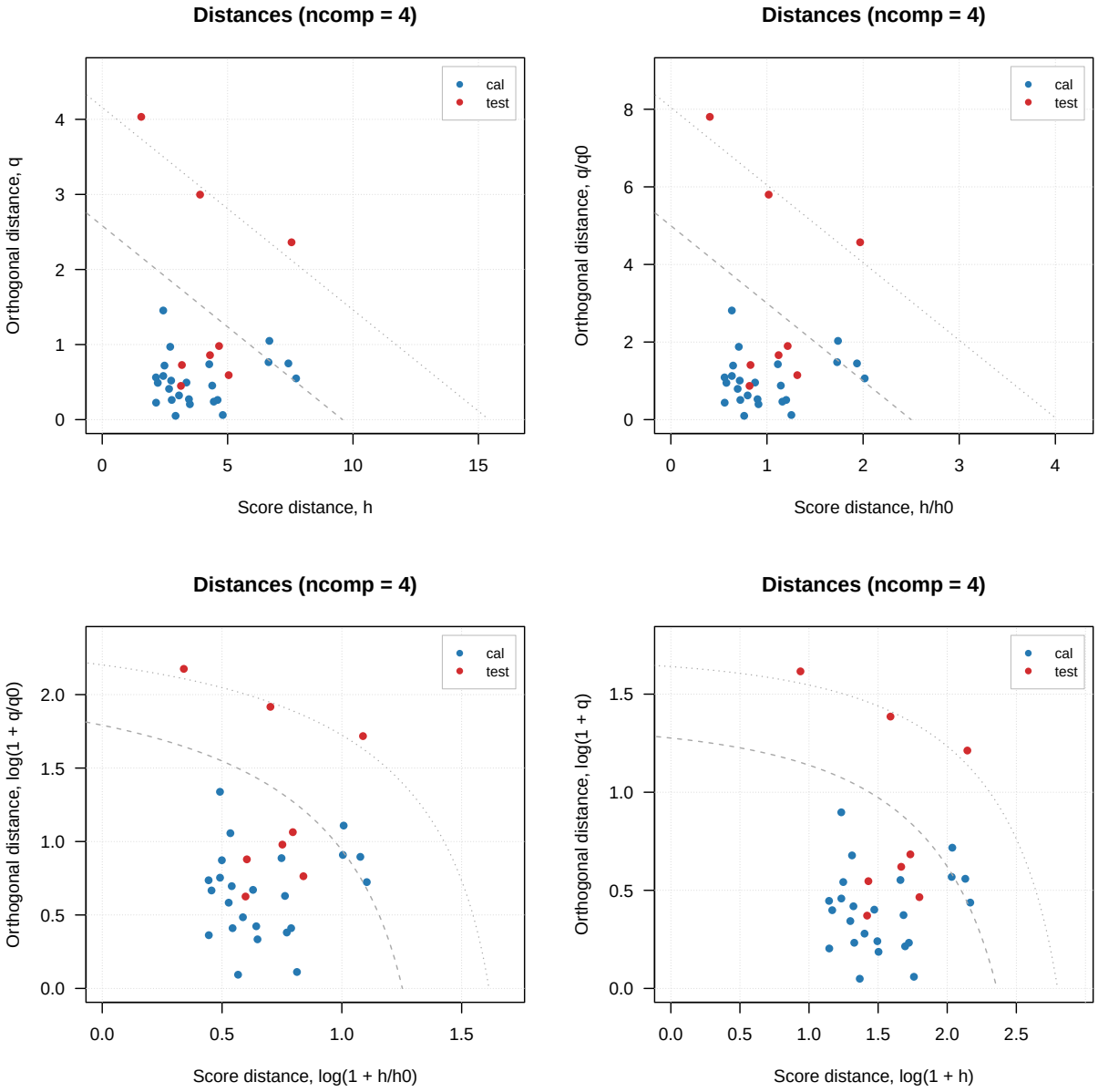
```
par(mfrow = c(1, 2))
plotResiduals(m, show.limits = FALSE)
plotResiduals(m, lim.col = c("red", "orange"), lim.lwd = c(2, 2), lim.lty = c(1, 2))
```



It is necessary to provide a vector with two values for each of the argument (first for extreme objects and second for outliers border).

Finally, you can also notice that for model plot the distance values are normalized (h/h_0 and q/q_0 instead of just h and q). This is needed to show the limits correctly and, again, will be explained in detail below. You can switch this option by using logical parameter `norm`. Sometimes, it is also make sense to use log transform for the values, which can be switched by using parameter `log`. Code below shows example for using these parameters.

```
par(mfrow = c(2, 2))
plotResiduals(m, norm = FALSE)
plotResiduals(m, norm = TRUE)
plotResiduals(m, norm = TRUE, log = TRUE)
plotResiduals(m, norm = FALSE, log = TRUE)
```



Critical limits

If PCA model is made for dataset taken from the same population, the orthogonal and score distances can be used to find outliers and extreme objects. One of the ways to do this is to compute critical limits for the distances assuming that they follow certain theoretical distribution.

Critical limits are also important for SIMCA classification as they are directly used for making decision on class belongings. This package implements several methods to compute the limits, which are explained in this section.

Data driven approach

Starting from version 0.10.0, by default the limits are computed using data driven approach proposed and then extended by Pomerantsev and Rodionova.

It was known before, that both distances are well described by chi-square distribution. The distribution in general describes behavior of sum of squared random values taken from standardized normal distribution, which means, that the distances has to be standardized first. The scaling factor as well as number of degrees of freedom (DoF) can be estimated from the distance values — hence the name (data driven).

Since the estimation procedure is identical for the both distances, we will use a generalized version. Let's say we have a vector of distance values u (where u can be either score distance values, h, or orthogonal distance values, q) for given number of components. Then the scaling factor, u_0 and corresponding DoF, N_u can be found as:

$$u_0 = m_u = \frac{1}{I}\sum_{i=1}^{I} u$$

$$N_u = \text{int}\left(\frac{2u_0^2}{s_u^2}\right)$$

Here s_u^2 is a variance of the distances. Then the the normalized distance values will follow chi-square distribution with N_u degrees of freedom (just replace u to q or h to get the formula for particular distance):

$$N_u \frac{u}{u_0} \propto \chi^2(N_u)$$

However, this does not take into account the fact that h and q are, strictly speaking, not independent. It is well known that adding an additional component to PCA model leads to increase of score distance and decrease of orthogonal distance and removing a component has the opposite effect. This relationship can be taken into account by computing a joint or a *full distance*, f, as follows:

$$f = N_h \frac{h}{h_0} + N_q \frac{q}{q_0}$$

The full distance, f also follows chi-square distribution with degrees of freedom: $N_f = N_q + N_h$. The critical limits for the full distance are computed using inverse cumulative distribution function (ICDF) also known as quantile function. For extreme objects (shown on distance plot as dashed line) as:

$$f_{crit} = \chi^{-2}(1-\alpha, N_f)$$

Where α is a significance level — expected number of extreme objects (e.g. for $\alpha = 0.05$ we expect that 5% of objects will be categorized as extreme). Critical limit for outliers is found as:

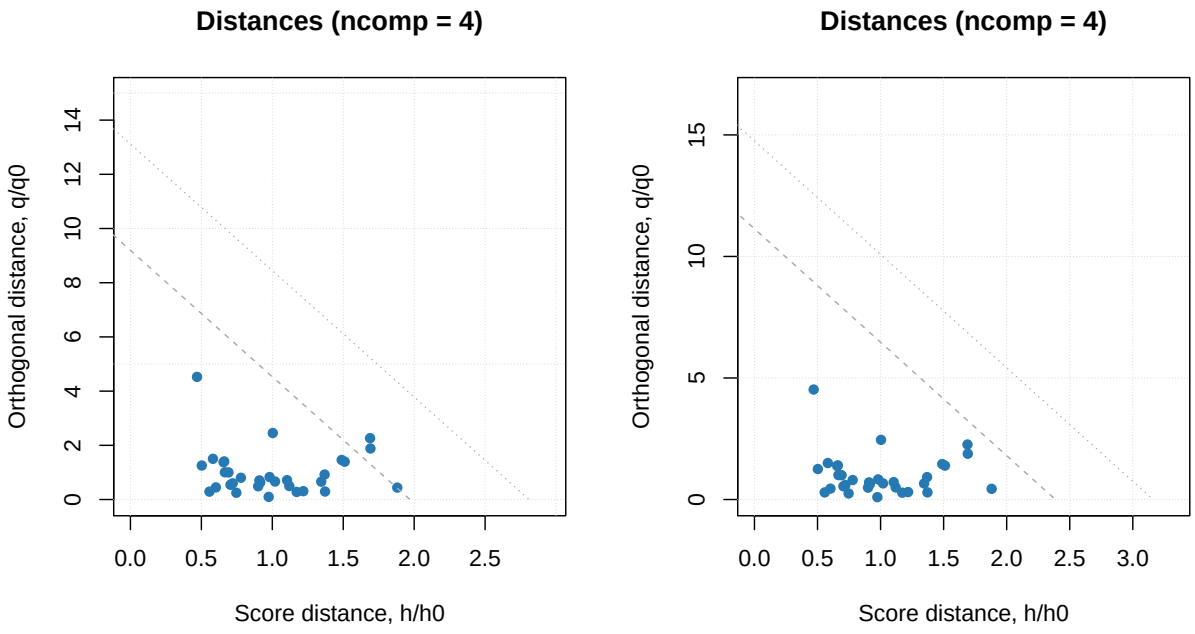
$$f_{crit} = \chi^{-2}((1-\gamma)^{1/I}, N_f)$$

Here γ is a significance level for outliers, so if it is 0.01 every object has 1% chance to be detected as outlier. The test for outliers should treat every object independently, so it requires Bonferroni correction, as one can see from the formula.

You can change the significance level both for extreme objects and outliers either when you calibrate the model, by providing parameters `alpha` and `gamma`, or for existent model, by using function `setDistanceLimits()` as shown in the example below.

```
# calibrate a model with alpha = 5% and gamma = 5%
m = pca(people, 4, scale = TRUE, alpha = 0.05, gamma = 0.05)

par(mfrow = c(1, 2))
plotResiduals(m)
```

```
# change both levels to 1%
m = setDistanceLimits(m, alpha = 0.01, gamma = 0.01)
plotResiduals(m)
```



This function also allows to change method for computing the limits, which is discussed in next subsection.

Other methods for computing critical limits

The package implements several other methods for computing critical limits for residuals distances. The needed method can be selected by providing additional argument, `lim.type`, both when calibrate PCA model or when adjust the limits for existent model with `setDistanceLimits()` function. By default, `lim.type = "ddmoments"`, which corresponds to the method described above — data driven approach based on classical estimators (statistical moments).

When data is contaminated with outliers, using statistical moments may lead to wrong estimators, in this case it is recommended to use robust version instead by specifying `lim.type = "ddrobust"`. The robust approach utilizes median and inter-quartile range instead of mean and standard deviation (see the paper for details). The use of robust and classical data driven approaches also helps to identify a proper model complexity, which will be discussed in the next section.

In addition to the data driven method, *mdatools*, also allows to use state-of-art approach, where limits for each distance are computed independently giving rectangular acceptance area on the distance plot. In this case, two methods are available for the orthogonal distance: either based on chi-square distribution but only for q values (`lim.type="chisq"`) or using Jackson-Mudholkar method (`lim.type="jm"`). If one of the two methods is selected, critical limits for the score distances, h, will be computed using Hotelling's T^2 distribution. This way of computing critical limits can be found in many popular chemometric software, for example, PLS_Toolbox.

Categorization of data rows based on critical limits

It is possible to categorize every row (object, measurement) in a dataset based on the two distances and critical limits computed for given model and limit parameters (method and significance limit). Function

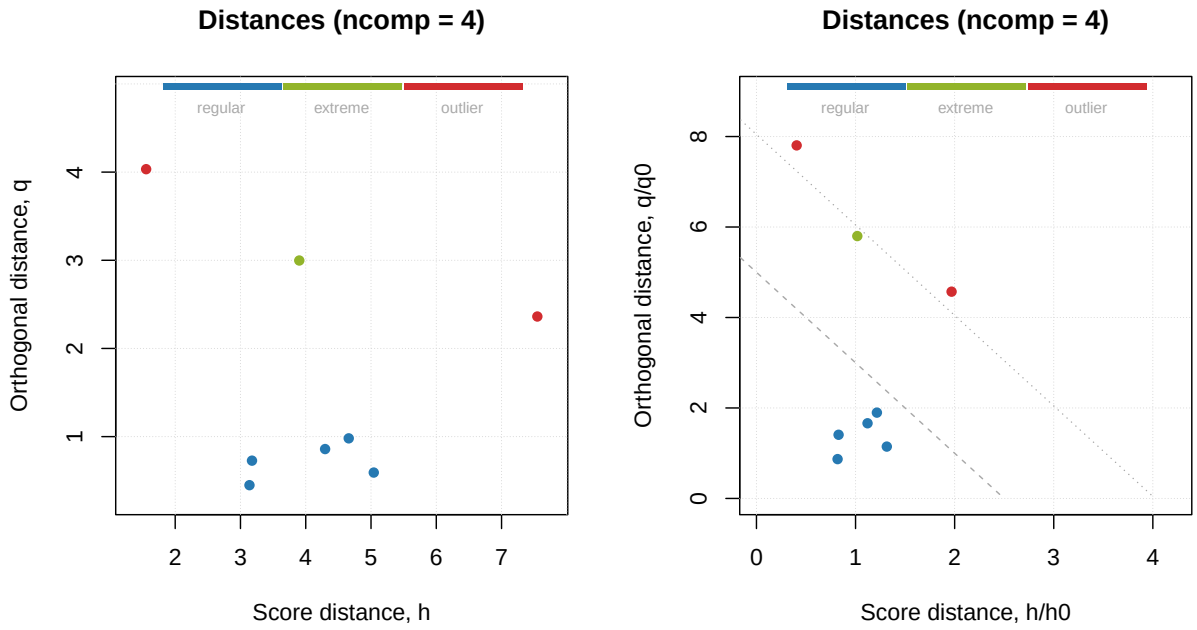
`categorize()` is made for that. An example below shows how to categorize new predictions using this function.

```
data(people)

idx = seq(4, 32, 4)
Xc = people[-idx, ]
Xt = people[idx, ]

m = pca(Xc, 4, scale = TRUE)
r = predict(m, Xt)

c = categorize(m, r)
print(c)
```

```
## [1] outlier regular regular regular extreme regular outlier regular
## Levels: regular extreme outlier
```

As one can see, there are two outliers, one extreme and five regular objects. This can be visualized on distance plot:

```
par(mfrow = c(1, 2))
plotResiduals(r, cgroup = c)
plotResiduals(m, res = list("new" = r), cgroup = c)
```

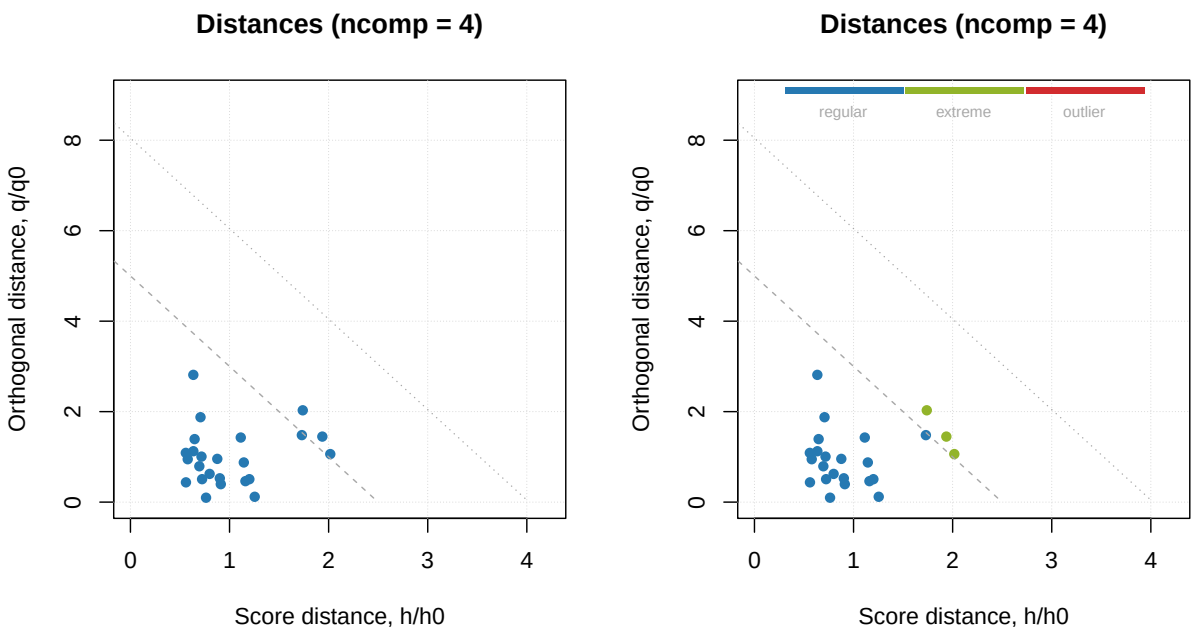


The first plot is a normal distance/residuals plot for results, while the second plot is made for model with manually specified result list. The last has possibility to show critical limits and see why actually the objects have been categorized like this.

When the plot is made for calibration set, this option can be simplified by providing specific value for parameter `cgroup` as shown below.

```
par(mfrow = c(1, 2))
plotResiduals(m)
```

```
plotResiduals(m, cgroup = "categories")
```



This option will work only if plot is made for one result object.

Model complexity

Complexity of PCA model is first of all associated with selection of proper (optimal) number of components. The optimal number should explain the systematic variation of the data points and keep the random variation uncaptured. Traditionally, number of components is selected by investigation of eigenvalues or looking at plot with residual or explained variance. However, this is quite complicated issue and result of selection depends very much on quality of data and purpose PCA model is built for. More details can be found in this paper.

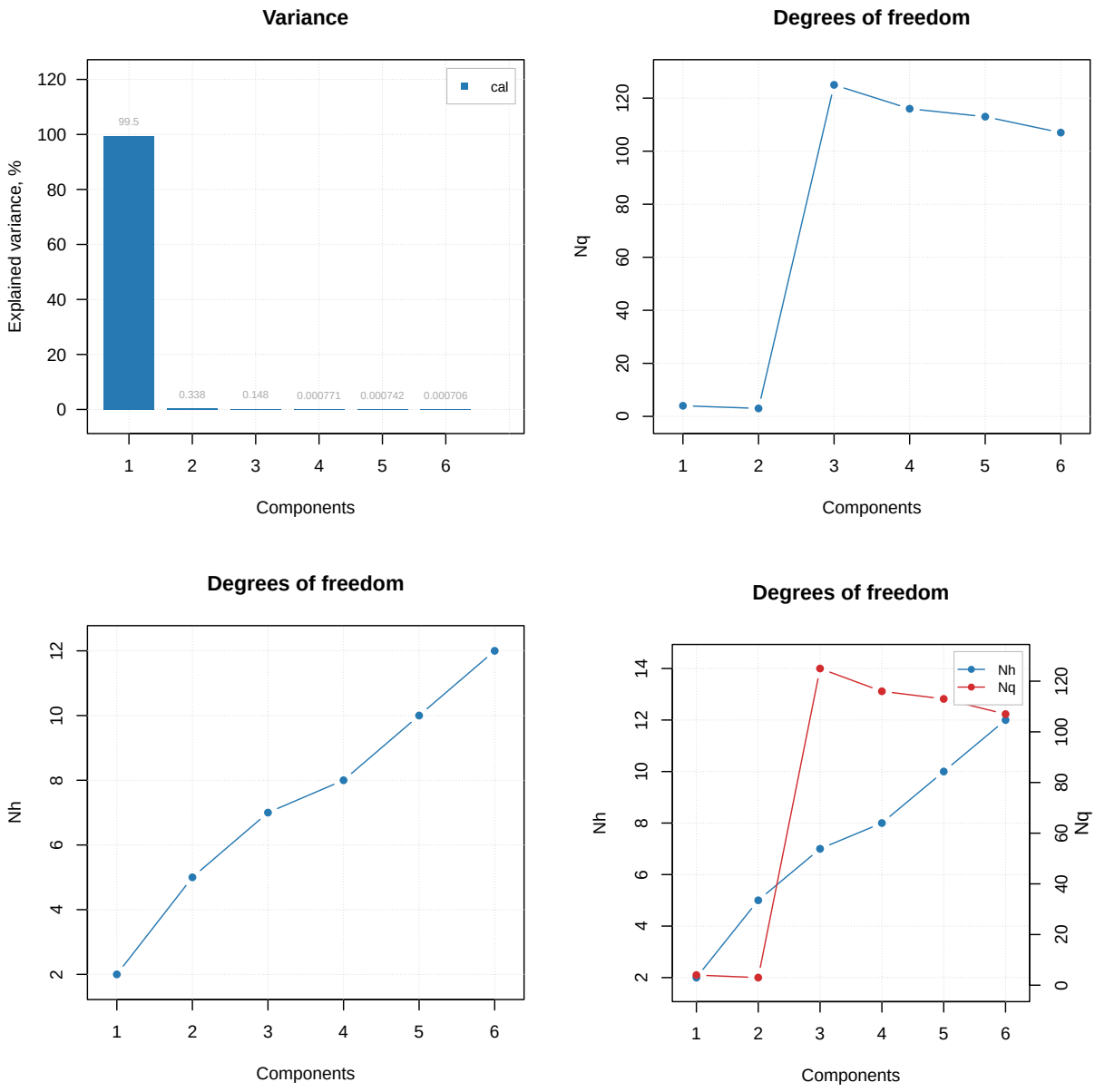
In *mdatools* there are several additional instruments both to select proper number of components as well as to see if, for example, new data (test set or new set of measurements) are well fitted by the model.

The first tool is the use of degrees of freedom for orthogonal and score distances, we mentioned earlier. A simple plot, where number of the degrees of freedom is plotted against number of components in PCA model can often reveal the presence of overfitting very clearly — the DoF value for the orthogonal distance, N_q jumps up significantly.

The code and its outcome below show how to make such plot separately for each distance and for both. The first plot in the figure is conventional plot with variance explained by each component. The model is made for *Simdata*, where it is known that the optimal number of components is two.

```
data(simdata)
Xc = simdata$spectra.c
Xt = simdata$spectra.t
m = pca(Xc, 6)

par(mfrow = c(2, 2))
plotVariance(m, type = "h", show.labels = TRUE)
plotQDoF(m)
```

```
plotT2DoF(m)
plotDistDoF(m)
```

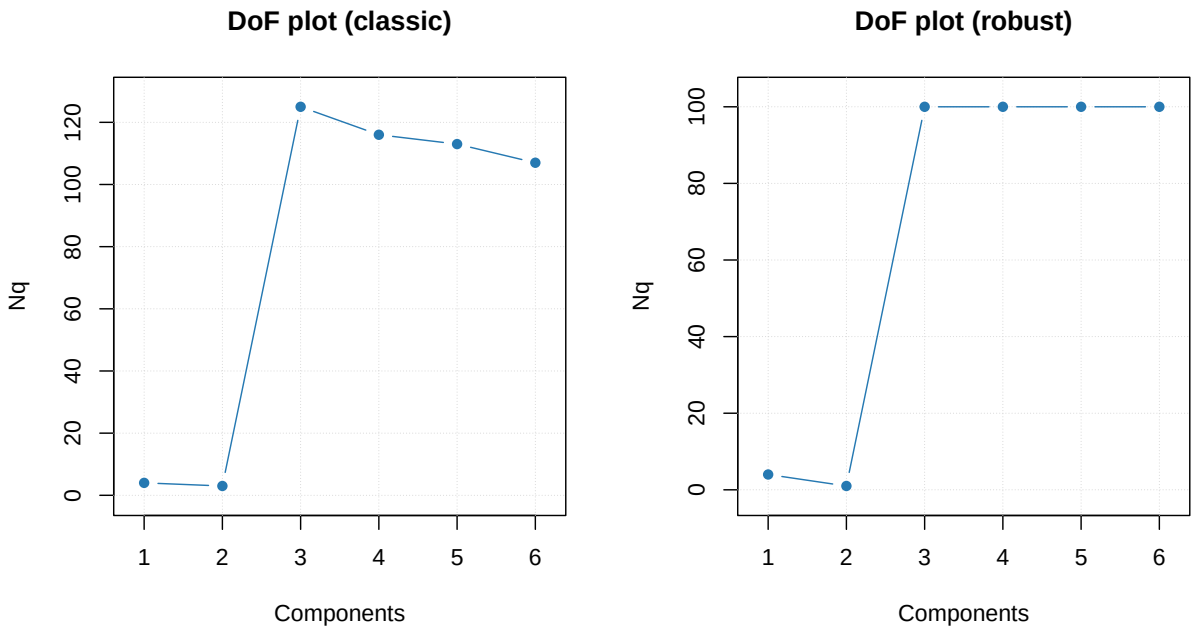


As you can see from the plots, second component explains less than 1% of variance and can be considered as non significant. However, plot for N_q shows a clear break at $A = 3$, indicating that both first and second PCs are important. The N_h values in this case do not provide any useful information.

In case if outliers present it can be useful to investigate the plot for N_q using classical and robust methods for estimation as shown in example below.

```
m = pca(Xc, 6)

par(mfrow = c(1, 2))
plotQDoF(m, main = "DoF plot (classic)")
```

```
m = setDistanceLimits(m, lim.type = "ddrobust")
plotQDoF(m, main = "DoF plot (robust)")
```



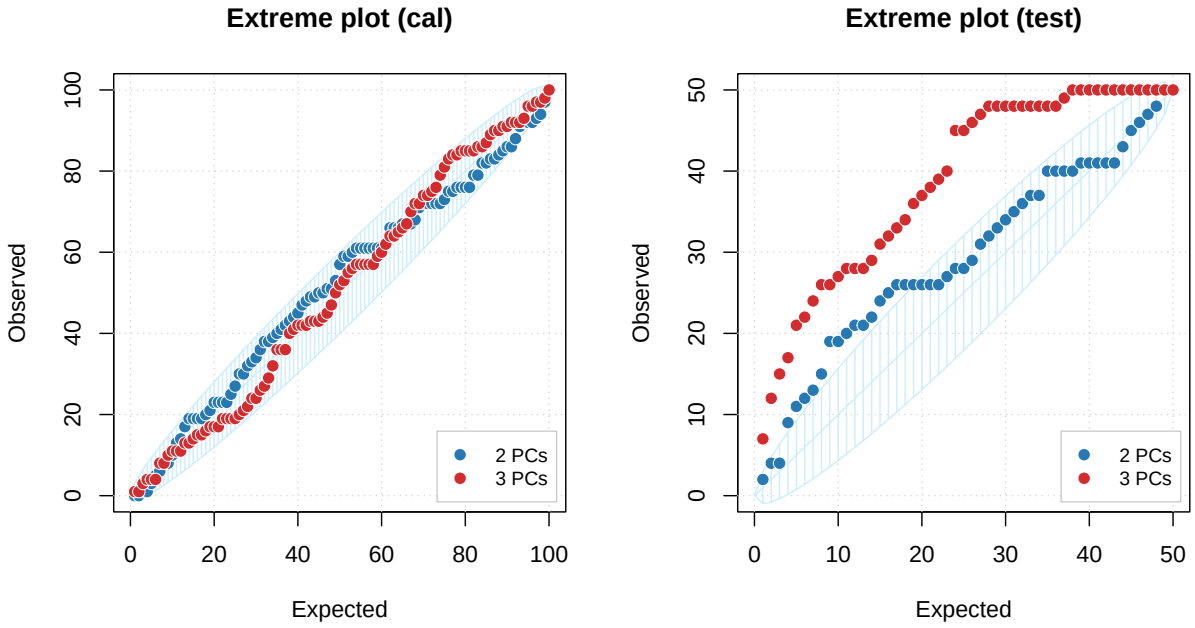
In this case both plots demonstrate quite similar behavior, however if they look differently it can be an indicator for a presence of outliers. The DoF plots work only if data driven method is used for computing of critical limits.

Another way to see how many components are optimal in the model, in case if you have a test set or just a new set of measurements you want to use the model with, is to employ the Extreme plot. The plot shows a number of extreme values for different α. Imaging that you make several distance plots with different α values and count how many objects model found as extremes. On the other hand you know that the expected value is αI. If you plot the number of extreme values vs. the expected number — this is the Extreme plot.

If model captures systematic variation both for calibration and test set the points on this plot will lie within a confidence ellipse shown using light blue color. However, if model is overfitted, the points are getting outside it.

Below you can see two Extreme plots made for the same data as the previous example. The left plot show results for calibration set for number of components in the PCA model equal to 2 and 3. The right plot show the results for the test set.

```
Xc = simdata$spectra.c
Xt = simdata$spectra.t
m = pca(Xc, 6, x.test = Xt)

par(mfrow = c(1, 2))
plotExtreme(m, comp = 2:3, main = "Extreme plot (cal)")
plotExtreme(m, comp = 2:3, res = m$res$test, main = "Extreme plot (test)")
```



As you can see, in case of calibration set, the points are lying withing confidence ellipse for both $A = 2$ and $A = 3$. However for the test set, the picture is quite different. In case of $A = 2$ most of the points are inside the interval, but for $A = 3$ all of them are clearly outside.

We hope these new tools will make the use of PCA more efficient.

Randomized PCA algorithms

Both SVD and NIPALS are not very efficient when number of rows in dataset is very large (e.g. hundreds of thousands values or even more). Such datasets can be easily obtained in case of for example hyperspectral images. Direct use of the traditional algorithms with such datasets often leads to a lack of memory and long computational time.

One of the solution here is to use probabilistic algorithms, which allow to reduce the number of values needed for estimation of principal components. Starting from *0.9.0* one of the probabilistic approach is also implemented in *mdatools*. The original idea can be found in this paper and some examples on using the approach for PCA analysis of hyperspectral images are described here.

The approach is based on calculation of matrix B, which has much smaller number of rows comparing to the original data, but captures the action of the data. To produce proper values for B, several parameters are used. First of all it is a rank of original data, k, if it is known. Second, it is oversampling parameter, p, which is important if rank is underestimated. The idea is to use p to overestimate the rank thus making solution more stable. The third parameter, q, is a number of iterations needed to make B more robust. Usually using $p = 5$ and $q = 1$ will work well on most of the datasets and, at the same time, will take less time for finding a solution comparing with conventional methods. By default, k is set to number of components, used for PCA decomposition.

The example below uses two methods (classic SVD and randomized SVD) for PCA decomposition of 100 000 x 300 dataset and compares time and main outcomes (loadings and explained variance).

```
# create a dataset as a linear combination of three sin curves with random "concentrations"
n = 100000
X = seq(0, 29.9, by = 0.1)
```

```
S = cbind(sin(X), sin(10 * X), sin(5 * X))
C = cbind(runif(n, 0, 1), runif(n, 0, 2), runif(n, 0, 3))
D = C %*% t(S)
D = D + matrix(runif(300 * n, 0, 0.5), ncol = 300)
show(dim(D))
```

```
## [1] 100000    300
```

```
# conventional SVD
t1 = system.time({m1 = pca(D, ncomp = 2)})
show(t1)
```

```
##    user  system elapsed
##  19.465   0.403  19.878
```

```
# randomized SVD with p = 5 and q = 1
t2 = system.time({m2 = pca(D, ncomp = 2, rand = c(5, 1))})
show(t2)
```

```
##    user  system elapsed
##   2.897   0.304   3.217
```

```
# compare variances
summary(m1)
```

```
##
## Summary for PCA model (class pca)
## Type of limits: ddmoments
## Alpha: 0.05
## Gamma: 0.01
##
##        Eigenvals Expvar Cumexpvar Nq Nh
## Comp 1   112.539  62.07     62.07  4  3
## Comp 2    49.978  27.56     89.63  6  5
```

```
summary(m2)
```

```
##
## Summary for PCA model (class pca)
##
## Parameters for randomized algorithm: q = 5, p = 1
## Type of limits: ddmoments
## Alpha: 0.05
## Gamma: 0.01
##
##        Eigenvals Expvar Cumexpvar Nq Nh
## Comp 1   112.539  62.07     62.07  4  3
## Comp 2    49.978  27.56     89.63  6  5
```

```
# compare loadings
show(m1$loadings[1:10, ])
```

```
##              Comp 1        Comp 2
##  [1,]  7.973072e-05  1.775574e-05
##  [2,]  3.865753e-02  6.927834e-02
##  [3,]  6.826246e-02  7.516323e-02
##  [4,]  8.135736e-02  1.241170e-02
##  [5,]  7.459189e-02 -6.123400e-02
```

```
##  [6,]  4.932195e-02 -7.792343e-02
##  [7,]  1.167070e-02 -2.290599e-02
##  [8,] -2.892920e-02  5.313597e-02
##  [9,] -6.217601e-02  7.988179e-02
## [10,] -7.993952e-02  3.244552e-02
```

```
show(m2$loadings[1:10, ])
```

```
##              Comp 1        Comp 2
##  [1,] -7.973072e-05 -1.775574e-05
##  [2,] -3.865753e-02 -6.927834e-02
##  [3,] -6.826246e-02 -7.516323e-02
##  [4,] -8.135736e-02 -1.241170e-02
##  [5,] -7.459189e-02  6.123400e-02
##  [6,] -4.932195e-02  7.792343e-02
##  [7,] -1.167070e-02  2.290599e-02
##  [8,]  2.892920e-02 -5.313597e-02
##  [9,]  6.217601e-02 -7.988179e-02
## [10,]  7.993952e-02 -3.244552e-02
```

As you can see the explained variance values, eigenvalues and loadings are identical in the two models and the second method is several times faster.

It is possible to make PCA decomposition even faster if only loadings and scores are needed. In this case you can use method `pca.run()` and skip other steps, like calculation of distances, variances, critical limits and so on. But in this case data matrix must be centered (and scaled if necessary) manually prior to the decomposition. Here is an example using the data generated in previous code.

```
D = scale(D, center = TRUE, scale = FALSE)

# conventional SVD
t1 = system.time({m1 = pca.run(D, method = "svd", ncomp = 2)})
show(t1)
```

```
##    user  system elapsed
##  18.058   0.179  18.238
```

```
# randomized SVD with p = 5 and q = 1
t2 = system.time({m2 = pca.run(D, method = "svd", ncomp = 2, rand = c(5, 1))})
show(t2)
```

```
##    user  system elapsed
##   1.607   0.031   1.638
```

```
# compare loadings
show(m1$loadings[1:10, ])
```

```
##                [,1]          [,2]
##  [1,]  7.973072e-05  1.775574e-05
##  [2,]  3.865753e-02  6.927834e-02
##  [3,]  6.826246e-02  7.516323e-02
##  [4,]  8.135736e-02  1.241170e-02
##  [5,]  7.459189e-02 -6.123400e-02
##  [6,]  4.932195e-02 -7.792343e-02
##  [7,]  1.167070e-02 -2.290599e-02
##  [8,] -2.892920e-02  5.313597e-02
##  [9,] -6.217601e-02  7.988179e-02
## [10,] -7.993952e-02  3.244552e-02
```

```
show(m2$loadings[1:10, ])
```

```
##                [,1]          [,2]
##  [1,] -7.973072e-05 -1.775574e-05
##  [2,] -3.865753e-02 -6.927834e-02
##  [3,] -6.826246e-02 -7.516323e-02
##  [4,] -8.135736e-02 -1.241170e-02
##  [5,] -7.459189e-02  6.123400e-02
##  [6,] -4.932195e-02  7.792343e-02
##  [7,] -1.167070e-02  2.290599e-02
##  [8,]  2.892920e-02 -5.313597e-02
##  [9,]  6.217601e-02 -7.988179e-02
## [10,]  7.993952e-02 -3.244552e-02
```

As you can see the loadings are still the same but the probabilistic algorithm is about 10 times faster. The exact difference depends on computer of course so if you run these examples on your compute you will get a bit different results, however the randomized algorithm will always be a clear winner.

Partial least squares regression

Partial least squares regression (PLS) is a linear regression method, which uses principles similar to PCA: data is decomposed using latent variables. Because in this case we have two datasets, matrix with predictors ($\mathbf{X}$) and matrix with responses ($\mathbf{Y}$) we do decomposition for both, computing scores, loadings and residuals: $\mathbf{X} = \mathbf{T}\mathbf{P}^{\mathrm{T}} + \mathbf{E}_x$, $\mathbf{Y} = \mathbf{U}\mathbf{Q}^{\mathrm{T}} + \mathbf{E}_y$. In addition to that, orientation of latent variables in PLS is selected to maximize the covariance between the X-scores, $\mathbf{T}$, and Y-scores $\mathbf{U}$. This approach makes possible to work with datasets where more traditional Multiple Linear Regression fails — when number of variables exceeds number of observations and when X-variables are mutually correlated. But, at the end, PLS-model is a linear model, where response value is just a linear combination of predictors, so the main outcome is a vector with regression coefficients.

There are two main algorithms for PLS, *NIPALS* and *SIMPLS*, in the *mdatools* only the last one is implemented. PLS model and PLS results objects have a lot of properties and performance statistics, which can be visualized via plots. Besides that, there is also a possibility to compute selectivity ratio (SR) and VIP scores, which can be used for selection of most important variables. Another additional option is a randomization test which helps to select optimal number of components. We will discuss most of the methods in this chapter and you can get the full list using `?pls`.

Models and results

Like we discussed for PCA, *matools* creates two types of objects — a model and a result. Every time you build a PLS model you get a *model object*. Every time you apply the model to a dataset you get a *result object*. For PLS, the objects have classes `pls` and `plsres` correspondingly.

Model calibration

Let's use the same *People* data and create a PLS-model for prediction of *Shoesize* (column number four) using other 11 variables as predictors. As usual, we start with preparing datasets (we will also split the data into calibration and test subsets):

```
library(mdatools)
data(people)

idx = seq(4, 32, 4)
Xc = people[-idx, -4]
yc = people[-idx, 4, drop = FALSE]
Xt = people[idx, -4]
yt = people[idx, 4, drop = FALSE]
```

So `Xc` and `yc` are predictors and response values for calibration subset. Now let's calibrate the model and show an information about the model object:

```
m = pls(Xc, yc, 7, scale = TRUE, info = "Shoesize prediction model")
```

```
## Warning in selectCompNum.pls(model, selcrit = ncomp.selcrit): No validation results were found.
```

You can notice that the calibration succeeded but there is also a warning about lack of validation results. For supervised models, which have complexity parameter (in this case — number of components), doing proper validation is important as it helps to find the optimal complexity. When you calibrate PLS model the calibration also tries to find the optimal number (details will be discussed later in this chapter) and this needs some validation. How to do proper validation of PLS models is discussed in the next section.

Here is an info for the model object:

```
print(m)
```

```
##
## PLS model (class pls)
##
## Call:
## selectCompNum.pls(obj = model, selcrit = ncomp.selcrit)
##
## Major fields:
## $ncomp - number of calculated components
## $ncomp.selected - number of selected components
## $coeffs - object (regcoeffs) with regression coefficients
## $xloadings - vector with x loadings
## $yloadings - vector with y loadings
## $weights - vector with weights
## $res - list with results (calibration, cv, etc)
##
## Try summary(model) and plot(model) to see the model performance.
```

As expected, we see loadings for predictors and responses, matrix with weights, and a special object (`regcoeffs`) for regression coefficients.

Result object

Similar to PCA, model object contains list with result objects (`res`), obtained using calibration set (`cal`), cross-validation (`cv`) and test set validation (`test`). All three have class `plsres`, here is how `res$cal` looks like:

```
print(m$res$cal)
```

```
##
## PLS results (class plsres)
##
## Call:
## plsres(y.pred = yp, y.ref = y.ref, ncomp.selected = object$ncomp.selected,
##     xdecomp = xdecomp, ydecomp = ydecomp)
##
## Major fields:
## $ncomp.selected - number of selected components
## $y.pred - array with predicted y values
## $y.ref - matrix with reference y values
## $rmse - root mean squared error
## $r2 - coefficient of determination
## $slope - slope for predicted vs. measured values
## $bias - bias for prediction vs. measured values
## $ydecomp - decomposition of y values (ldecomp object)
## $xdecomp - decomposition of x values (ldecomp object)
```

The `xdecomp` and `ydecomp` are objects similar to `pcares`, they contain scores, residuals and variances for decomposition of X and Y correspondingly.

```
print(m$res$cal$xdecomp)
```

```
##
## Results of data decomposition (class ldecomp).
##
## Major fields:
## $scores - matrix with score values
## $T2 - matrix with T2 distances
## $Q - matrix with Q residuals
## $ncomp.selected - selected number of components
## $expvar - explained variance for each component
## $cumexpvar - cumulative explained variance
```

Other fields are mostly various performance statistics, including slope, coefficient of determination (R^2), bias, and root mean squared error (RMSE). Besides that, the results also include reference y-values and array with predicted y-values. The array has dimension *nObjects x nComponents x nResponses*.

PLS predictions for a new set can be obtained using method `predict`:

```
res = predict(m, Xt, yt)
print(res)
```

```
##
## PLS results (class plsres)
##
## Call:
## plsres(y.pred = yp, y.ref = y.ref, ncomp.selected = object$ncomp.selected,
##     xdecomp = xdecomp, ydecomp = ydecomp)
##
## Major fields:
## $ncomp.selected - number of selected components
## $y.pred - array with predicted y values
## $y.ref - matrix with reference y values
## $rmse - root mean squared error
## $r2 - coefficient of determination
## $slope - slope for predicted vs. measured values
## $bias - bias for prediction vs. measured values
## $ydecomp - decomposition of y values (ldecomp object)
## $xdecomp - decomposition of x values (ldecomp object)
```

If reference y-values are not provided to `predict()` function, then all predictions are computed anyway, but performance statistics (and corresponding plot) will be not be available.

Validation

Validation of PLS (and PLS-DA) models can be done by one of the following methods:

1. Using separate validation/test set
2. Uisng Procrustes cross-validation
3. Using conventional cross-validation

Below we will show each of the method in detail. For all examples we will use *Simdata* with UV/VIS spectra of mixtures of three carbonhydrates, this dataset has been already used for examples in Preprocessing and PCA chapters. It contains spectra and reference concentrations for two sets of measurements.

Using validation/test set

In case if you have a dedicated validation/test set, you need to provide it as two separate matrices — one for predictors (parameter `x.test`) and one for response (`y.test`) values. The rest will be done automatically and you will see the results for test set in all plots and summary tables.

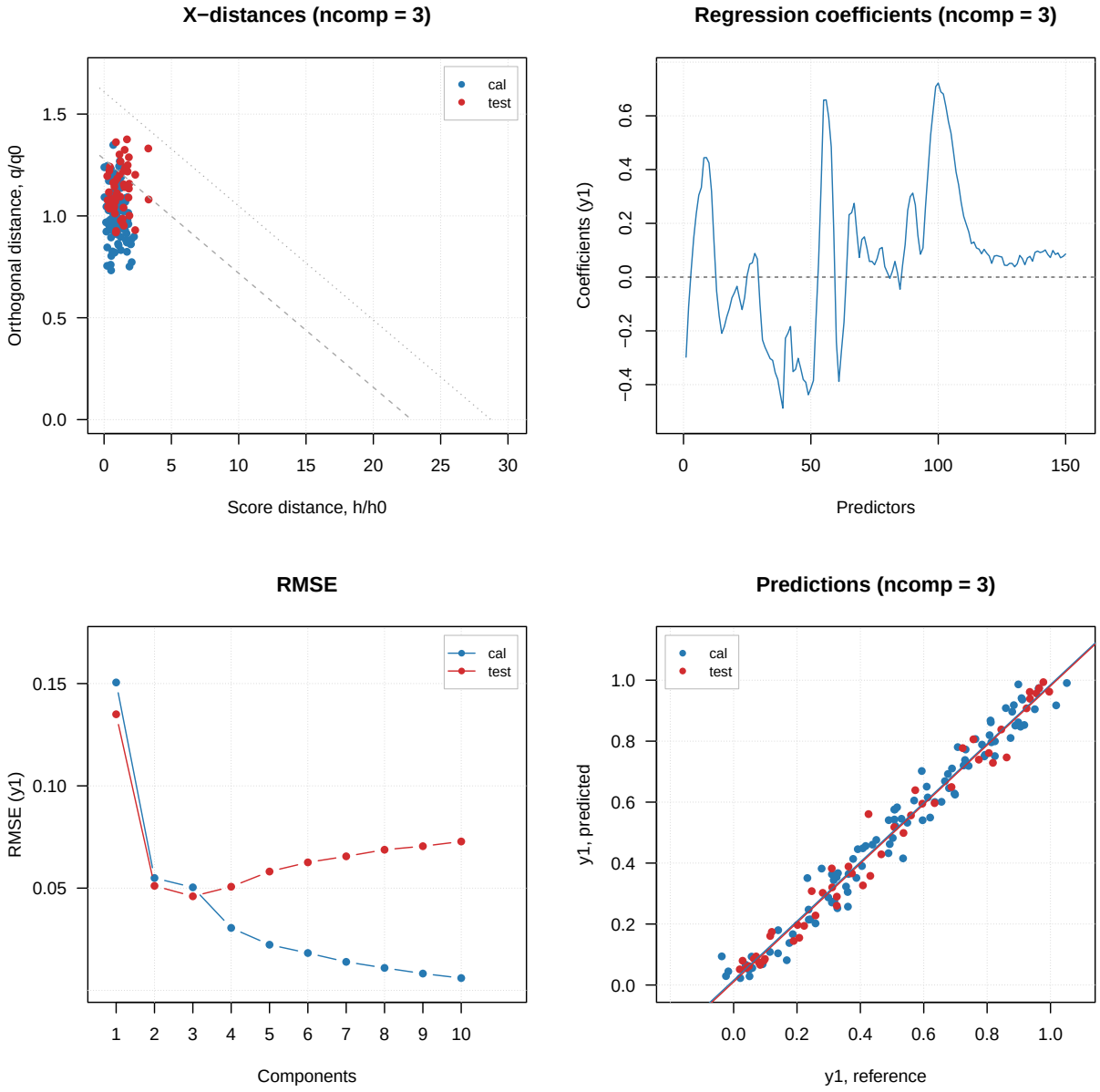
The example below shows how to create such model for *Simdata* (here we will make PLS1 model for prediction of concentration of the first mixture component — first column of the concentration matrix):

```
data(simdata)
Xc = simdata$spectra.c
yc = simdata$conc.c[, 1]
Xt = simdata$spectra.t
yt = simdata$conc.t[, 1]

m = pls(Xc, yc, 10, x.test = Xt, y.test = yt)
summary(m)
```

```
##
## PLS model (class pls) summary
## -------------------------------
## Info:
## Number of selected components: 3
## Cross-validation: none
##
##      X cumexpvar Y cumexpvar    R2  RMSE Slope  Bias  RPD
## Cal       99.977      96.960 0.970 0.050 0.970 0.000 5.76
## Test      99.983      98.153 0.979 0.046 0.971 0.003 6.93
```

As you can see, in this case we do not have a warning about absence of validation results as we have seen in the example above. Summary information is avaiable for optimal number of components for both calibration and test set. Model was able to detect that 3 components is optimal despite the provided number of 10 components. How this selection is done is described later in this chapter, meanwhile let's look at the plot overview:

```
plot(m)
```



Distance plot, RMSE and plot with predicted and measured y-values have points for both calibration and test set. The RMSE plot shows clearly that after third component the model becomes overfitted as error for the test set is increasing.

As mentioned, method `summary()` for validated model shows performance statistics calculated using optimal number of components for each of the results (calibration and test set validation in our case).

If you want more details run `summary()` for one of the result objects.

```
summary(m$res$test)
```

```
##
## PLS regression results (class plsres) summary
## Info: test set validation results
## Number of selected components: 3
##         X expvar X cumexpvar Y expvar Y cumexpvar    R2  RMSE Slope    Bias  RPD
```

```
## Comp 1    99.638      99.638   84.099      84.099 0.816 0.135 0.868 -0.0234 2.39
## Comp 2     0.246      99.884   13.621      97.720 0.974 0.051 0.962  0.0039 6.24
## Comp 3     0.099      99.983    0.433      98.153 0.979 0.046 0.971  0.0030 6.93
## Comp 4     0.000      99.983   -0.395      97.758 0.974 0.051 0.975  0.0037 6.29
## Comp 5     0.000      99.983   -0.705      97.053 0.966 0.058 0.982  0.0024 5.48
## Comp 6     0.000      99.983   -0.472      96.581 0.961 0.063 0.989  0.0035 5.09
## Comp 7     0.000      99.983   -0.330      96.251 0.957 0.066 0.996  0.0064 4.88
## Comp 8     0.000      99.983   -0.378      95.873 0.952 0.069 0.995  0.0083 4.66
## Comp 9     0.000      99.983   -0.209      95.664 0.950 0.070 0.996  0.0099 4.56
## Comp 10    0.000      99.983   -0.288      95.376 0.947 0.073 0.997  0.0106 4.42
```

In this case, the statistics are shown for all available components and explained variance for individual components is added.

Using Procrustes cross-validation

As in case of PCA, you can also generate the validation set using Procrustes cross-validation. Here I assume that you already have package `pcv` instealled and read about the method (or at least tried it with PCA examples).

Let's create the PV-set first:

```
library(pcv)
Xpv = pcvpls(Xc, yc, 20, cv = list("ven", 4))
```

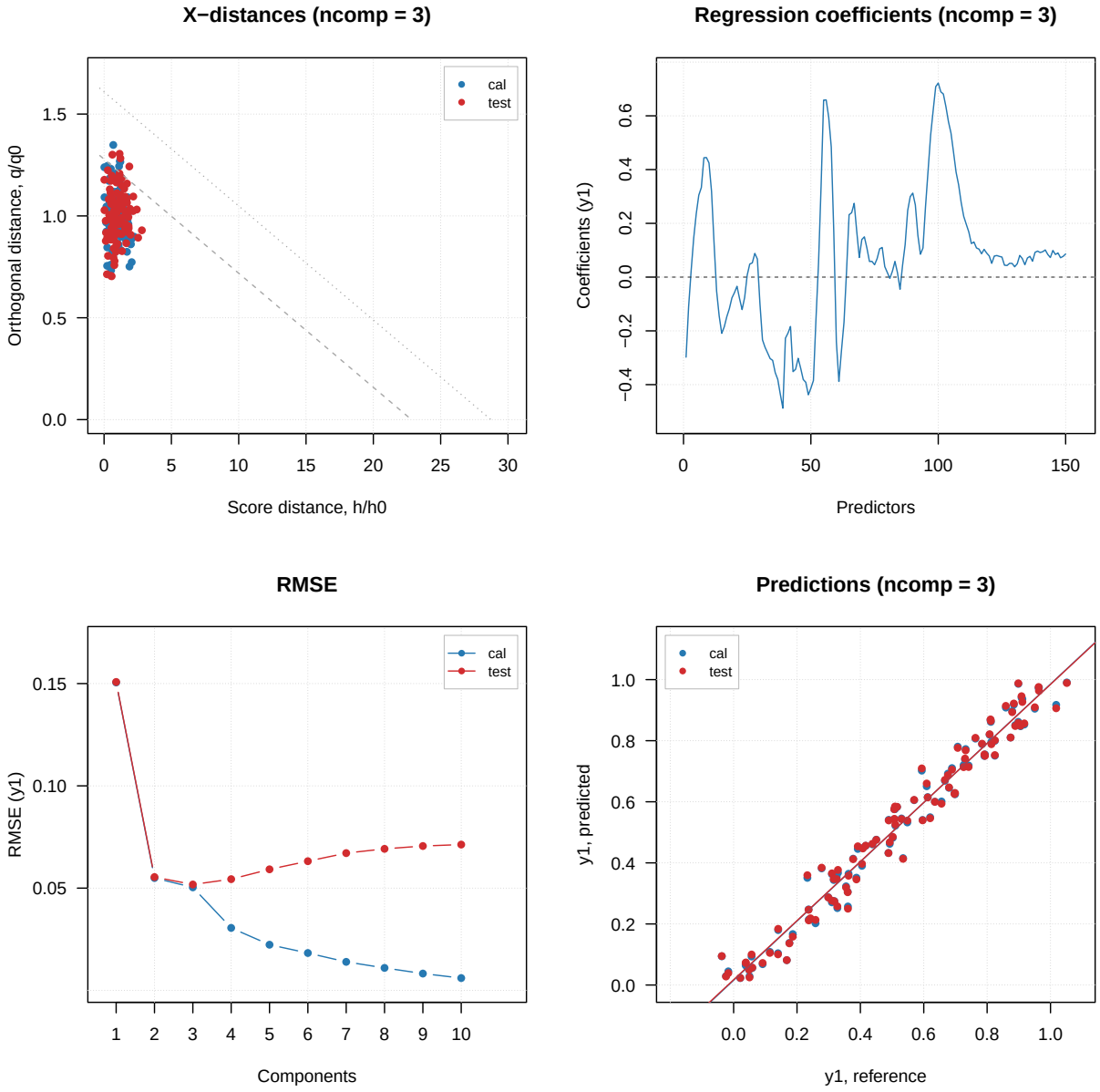
In this case we generate only predictors, the response values will be the same as for the calibration set. Now we can create a model similar to what we did for dedicated validation set (note that we provide `yc` as values for the `y.test` argument in this case):

```
m2 = pls(Xc, yc, 10, x.test = Xpv, y.test = yc)
summary(m2)
```

```
##
## PLS model (class pls) summary
## -------------------------------
## Info:
## Number of selected components: 3
## Cross-validation: none
##
##      X cumexpvar Y cumexpvar    R2  RMSE Slope   Bias  RPD
## Cal       99.977      96.960 0.970 0.050 0.970  0e+00 5.76
## Test      99.977      96.988 0.968 0.052 0.967 -4e-04 5.61
```

As you can see in this case PLS also selected 3 components as optimal. Here is the overview of the model.

```
plot(m2)
```



The behaviour of RMSE values is quite similar to the one we got for the test set.

Using conventional cross-validation

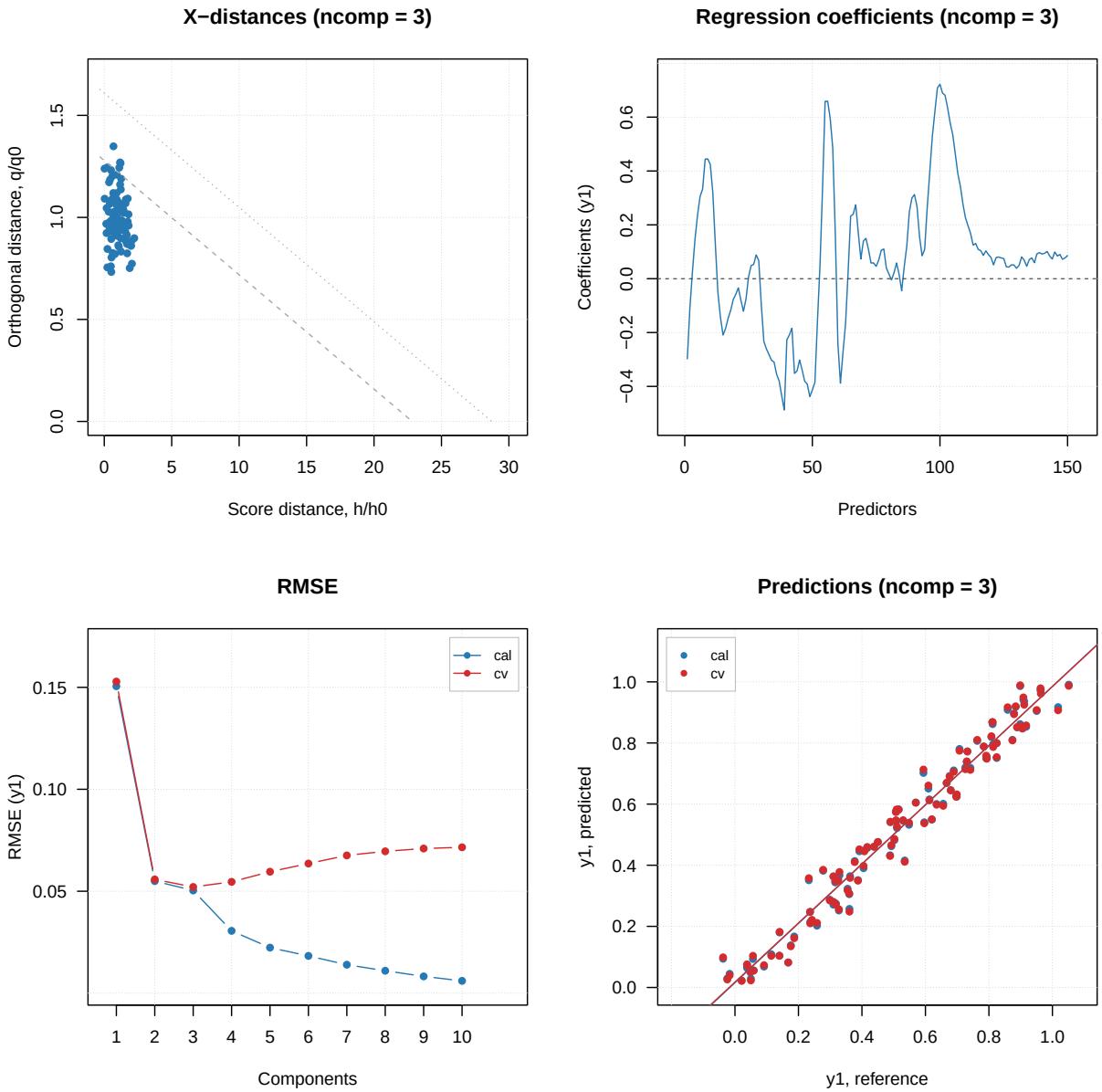
Alternatively you can use conventional cross-validation as well. But in this case only figures of merit (e.g. RMSE and R2 values) will be computed, as computing variances and distances can not be done correctly. This is enough though to optimize the parameters of your model (number of components, preprocessing, variable selection) and then apply the final optimized model to the test set.

In this case you do not need to provide any extra data, but just set cross-validation parameter `cv` similar to what you do when generate PV-set:

```
m3 = pls(Xc, yc, 10, cv = list("ven", 4))
summary(m3)
```

```
##
## PLS model (class pls) summary
## -------------------------------
## Info:
## Number of selected components: 3
## Cross-validation: venetian blinds with 4 segments
##
##     X cumexpvar Y cumexpvar    R2  RMSE Slope   Bias  RPD
## Cal    99.97681    96.96024 0.970 0.050 0.970  0e+00 5.76
## Cv           NA          NA 0.968 0.052 0.967 -5e-04 5.58
```

```
plot(m3)
```



As you can see, all results are very similar to what you got with PV-set, however some of the outcomes (e.g. distances, explained variance, etc) are not available.

In the examples above we set arameter `cv` to `list("ven", 4)`. But in fact it can be a number, a list or a vector with segment values manually assigned for each measurement.

If it is a number, it will be used as number of segments for random cross-validation, e.g. if `cv = 2` cross-validation with two segments will be carried out with measurements being randomly split into the segments. For full cross-validation use `cv = 1` like in the example above. This option is a bit confusing, logically we have to set `cv` equal to number of measurements in the calibration set. But this option was used many years ago when the package was created and it is kept for backward compatibility.

For more advanced selection you can provide a list with name of cross-validation method, number of segments and number of iterations, e.g. `cv = list("rand", 4, 4)` for running random cross-validation with four segments and four repetitions or `cv = list("ven", 8)` for systematic split into eight segments (*venetian blinds*). In case of venetian splits, the measurements will be automatically ordered from smallest to largest response (y) values.

Finally, you can also provide a vector with manual slits. For example, if you create a model with the following value for `cv` parameter: `cv = rep(1:4, length.out = nrow(Xc))` the vector of values you provide will look as `1 2 3 4 1 2 3 4 1 2 ...`, which corresponds to systematic splits but without ordering y-values, so it will use the current order of measurements (rows) in the data. Use this option if you have specific requirements for cross-validation and the implemented methods do not meet them.

Local and global CV scope

From *0.14.1* it is possible to set how cross-validation subsets are centred and scaled inside the cross-validation loop. This is regulated by a new parameter `cv.scope` in methods `pls()`, `plsda()`and `ipls()`.

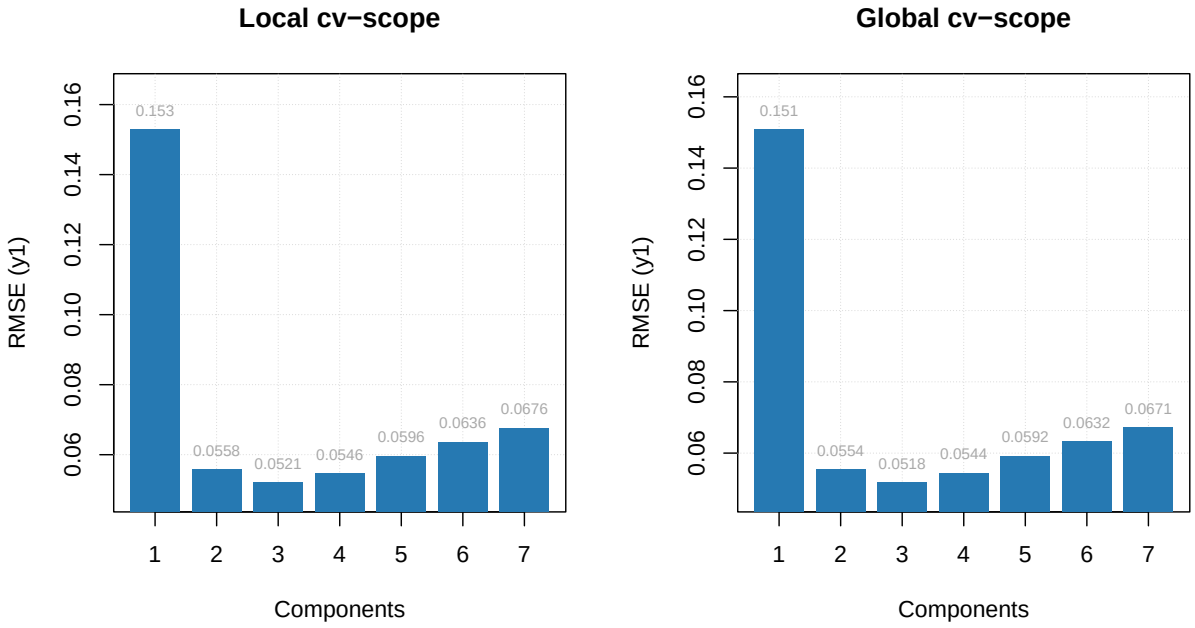
Cross-validation splits the original training set into local calibration and validation sets inside the loop. For example, if you have dataset with 100 objects/observations and use cross-validation with four segments (with random or systematic splits), in each iteration 25 objects will be taken as local validation set and remaining 75 objects will for a local calibration set.

If `cv.scope` parameter is set to `"local"`, then mean values for centreing and standard deviation values for scaling (if this option is selected) will be computed based on the local set with 75 objects in the example below. So, every local model will have its own center in the variable space.

If `cv.scope` parameter is set to `"global"`, then the mean values and standard deviation values will be computed for the original (global) training set hence all local models will have the same center as the global model.

The default value is `"local"` and this was also the way `mdatools` worked before this option was introduced in *0.14.1*. The small example below shows how both options work (it is assumed that you already have `Xc` and `yc` variables from the previos examples):

```
m1 = pls(Xc, yc, 7, cv = list("ven", 4), cv.scope = "local")
m2 = pls(Xc, yc, 7, cv = list("ven", 4), cv.scope = "global")

par(mfrow = c(1, 2))
plotRMSE(m1$res$cv, type = "h", show.labels = TRUE, main = "Local cv-scope")
plotRMSE(m2$res$cv, type = "h", show.labels = TRUE, main = "Global cv-scope")
```



As you can see the cross-validation results are very similar, however RMSE values for locally autoscaled models tend to be a bit higher as expected.

Selection of optimal number of components

When you create a PLS model, it tries to select optimal number of components automatically (which, of course, you can always change later). To do that, the method uses RMSE values, calculated for different number of components and validation predictions. So, if we do not use any validation, it warns us about this as it was mentioned in the beginning of this chapter.

If validation (any of the three methods described above) was used, there are two different ways/criteria for automatic selection of the components best on validation results. One is using first local minimum on the RMSE plot and second is so called Wold criterion, based on a ratio between PRESS values for current and next component.

You can select which criterion to use by specifying parameter `ncomp.selcrit` (either `'min'` or `'wold'`) as it is shown below (here we use cross-validation).

```
m1 = pls(Xc, yc, 7, cv = list("ven", 4), ncomp.selcrit = "min")
show(m1$ncomp.selected)
```

```
## [1] 3
```

```
m2 = pls(Xc, yc, 7, cv = list("ven", 4), ncomp.selcrit = "wold")
show(m2$ncomp.selected)
```

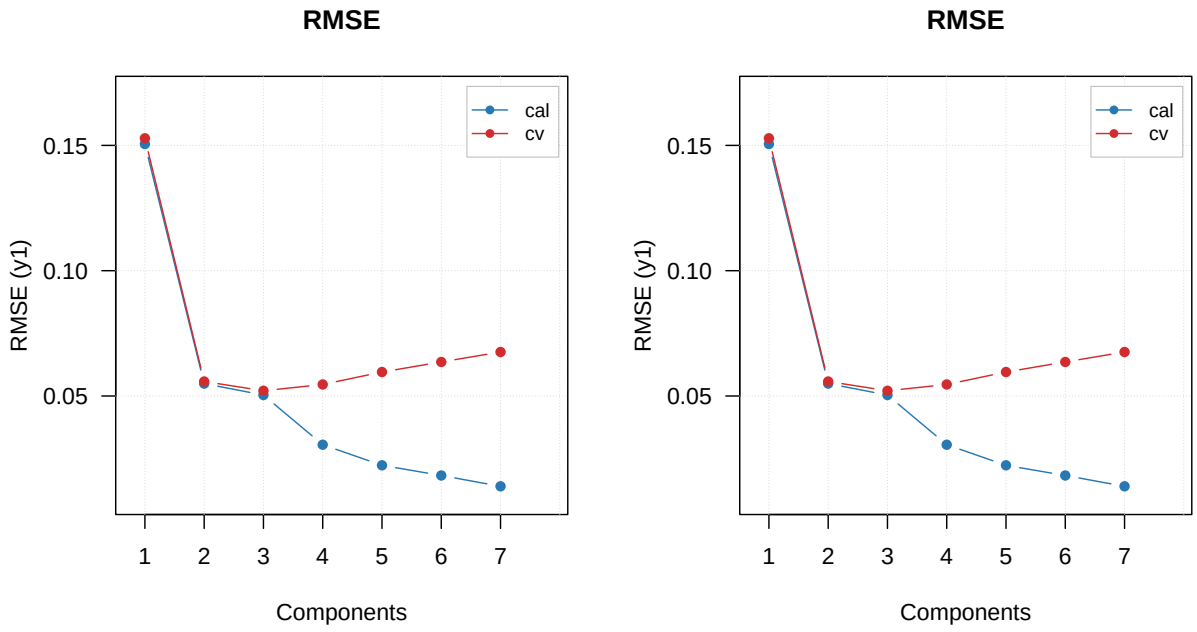
```
## [1] 3
```

Well, in this case both pointed on the same number, 3, but sometimes they give different solutions.

And here are the RMSE plots (they are identical of course), where you can see how error depends on number of components for both calibration set and cross-validated predictions. Apparently the minimum for cross-validation error (RMSECV) is indeed at 3:

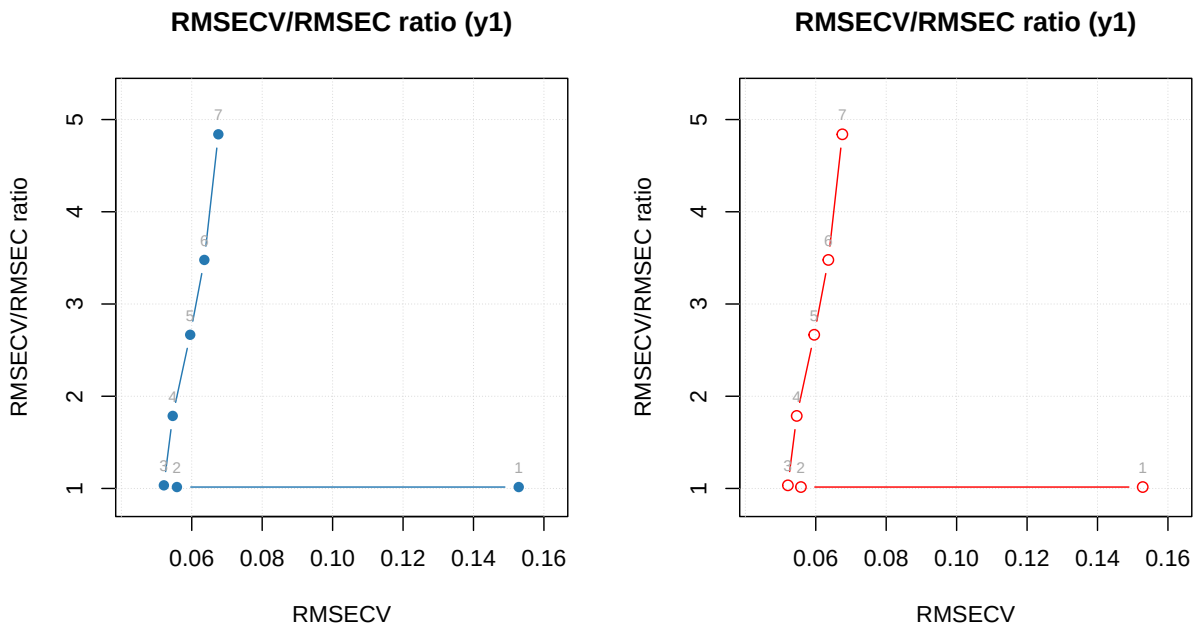
```
par(mfrow = c(1, 2))
plotRMSE(m1)
```

```
plotRMSE(m2)
```



Another useful plot in this case is a plot which shows ratio between cross-validated RMSE values, RMSECV, and the calibrated ones, RMSEC. You can see an example in the figure below and read more about this plot in blog post by Barry M. Wise.

```
par(mfrow = c(1, 2))
plotRMSERatio(m1)
plotRMSERatio(m2, pch = 1, col = "red")
```



Regression coefficients

First of all let us re-create the PLS model for prediction of Showsize from People data we used at the beginning of this chapter:

```
library(mdatools)
data(people)

idx = seq(4, 32, 4)
Xc = people[-idx, -4]
yc = people[-idx, 4, drop = FALSE]
Xt = people[idx, -4]
yt = people[idx, 4, drop = FALSE]
```

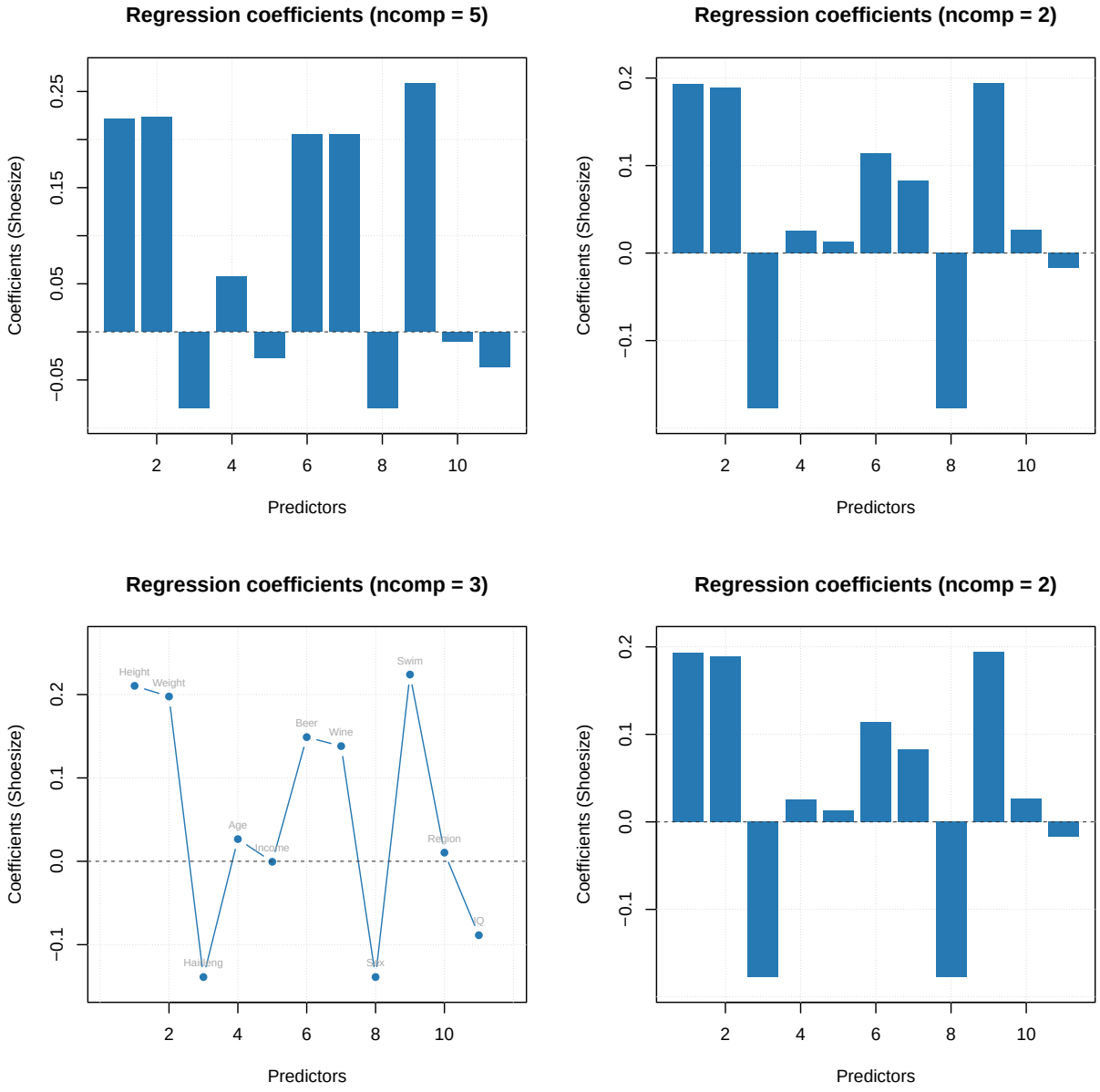
But this time I will create the model using full cross-validation:

```
m = pls(Xc, yc, 7, scale = TRUE, cv = 1, info = "Shoesize prediction model")
```

The values for regression coefficients are available in `m$coeffs$values`, it is an array with dimension *nVariables x nComponents x nPredictors*. The reason to use the object instead of just an array is mainly for being able to get and plot regression coefficients for different methods. Besides that, it is possible to calculate confidence intervals and other statistics for the coefficients using Jack-Knife method (will be shown later), which produces extra entities.

The regression coefficients can be shown as plot using either function `plotRegcoeffs()` for the PLS model object or function `plot()` for the object with regression coefficients. You need to specify for which predictor (if you have more than one y-variable) and which number of components you want to see the coefficients for. By default it shows values for the optimal number of components and first y-variable as it is shown on example below.

```
par(mfrow = c(2, 2))
plotRegcoeffs(m)
plotRegcoeffs(m, ncomp = 2)
plot(m$coeffs, ncomp = 3, type = "b", show.labels = TRUE)
plot(m$coeffs, ncomp = 2)
```



The model keeps regression coefficients, calculated for centered and standardized data, without intercept, etc. Here are the values for three PLS components.

```
show(m$coeffs$values[, 3, 1])
```

```
##       Height       Weight     Hairleng          Age       Income         Beer         Wine            S
##  0.210411676  0.197646483 -0.138824482  0.026613035 -0.000590693  0.148917913  0.138138095 -0.138824
##           IQ
## -0.088658626
```

You can see a summary for the regression coefficients object by calling function `summary()` for the object `m$coeffs` like it is show below. By default it shows only estimated regression coefficients for the selected y-variable and number of components.

However, if you use cross-validation, Jack-Knifing method will be used to compute some statistics, including standard error, p-value (for test if the coefficient is equal to zero in population) and confidence interval. All

statistics in this case will be shown automatically with `summary()` as you can see below.

```
summary(m$coeffs)
```

```
##
## Regression coefficients for Shoesize (ncomp = 1)
## -------------------------------------------------
##                Coeffs  Std. err. t-value p-value        2.5%       97.5%
## Height    0.176077659 0.01594024   11.03   0.000  0.14310275  0.20905257
## Weight    0.175803980 0.01598815   10.98   0.000  0.14272997  0.20887799
## Hairleng -0.164627444 0.01638528  -10.04   0.000 -0.19852297 -0.13073192
## Age       0.046606027 0.03718827    1.25   0.225 -0.03032377  0.12353583
## Income    0.059998121 0.04047132    1.47   0.155 -0.02372318  0.14371942
## Beer      0.133136867 0.01116749   11.89   0.000  0.11003515  0.15623859
## Wine      0.002751573 0.03542518    0.08   0.936 -0.07053100  0.07603415
## Sex      -0.164627444 0.01638528  -10.04   0.000 -0.19852297 -0.13073192
## Swim      0.173739533 0.01516461   11.44   0.000  0.14236915  0.20510992
## Region   -0.031357608 0.03590576   -0.87   0.395 -0.10563433  0.04291911
## IQ       -0.003353428 0.03841171   -0.08   0.934 -0.08281410  0.07610725
##
## Degrees of freedom (Jack-Knifing): 23
```

You can also get the corrected coefficients, which can be applied directly to the raw data (without centering and standardization), by using method `getRegcoeffs()`:

```
show(getRegcoeffs(m, ncomp = 3))
```

```
##               Estimated
## Intercept  1.251537e+01
## Height     8.105287e-02
## Weight     5.110732e-02
## Hairleng  -5.375404e-01
## Age        1.147785e-02
## Income    -2.580586e-07
## Beer       6.521476e-03
## Wine       1.253340e-02
## Sex       -5.375404e-01
## Swim       1.164947e-01
## Region     4.024083e-02
## IQ        -2.742712e-02
## attr(,"name")
## [1] "Regression coefficients for  Shoesize"
```

Plotting methods

Plotting methods, again, work similar to PCA, so in this section we will look more detailed at the available methods instead of on how to customize them. PLS has a lot of different results and much more possible plots. Here is a list of methods, which will work both for a model and for a particular results.

Plotting methods for summary statistics.

Method	Description
`plotRMSE(obj, ny, ...)`	RMSE values vs. number of components in a model
`plotRMSERatio(obj, ny, ...)`	RMSECV/RMSEC ratio values vs. RMSECV in a model
`plotXVariance(obj, ...)`	explained variance for X decomposition for each component

Method	Description
`plotXCumVariance(obj, ...)`	same as above but for cumulative variance
`plotYVariance(obj, ...)`	explained variance for Y decomposition for each component
`plotYCumVariance(obj, ...)`	same as above but for cumulative variance

Plotting methods for objects.

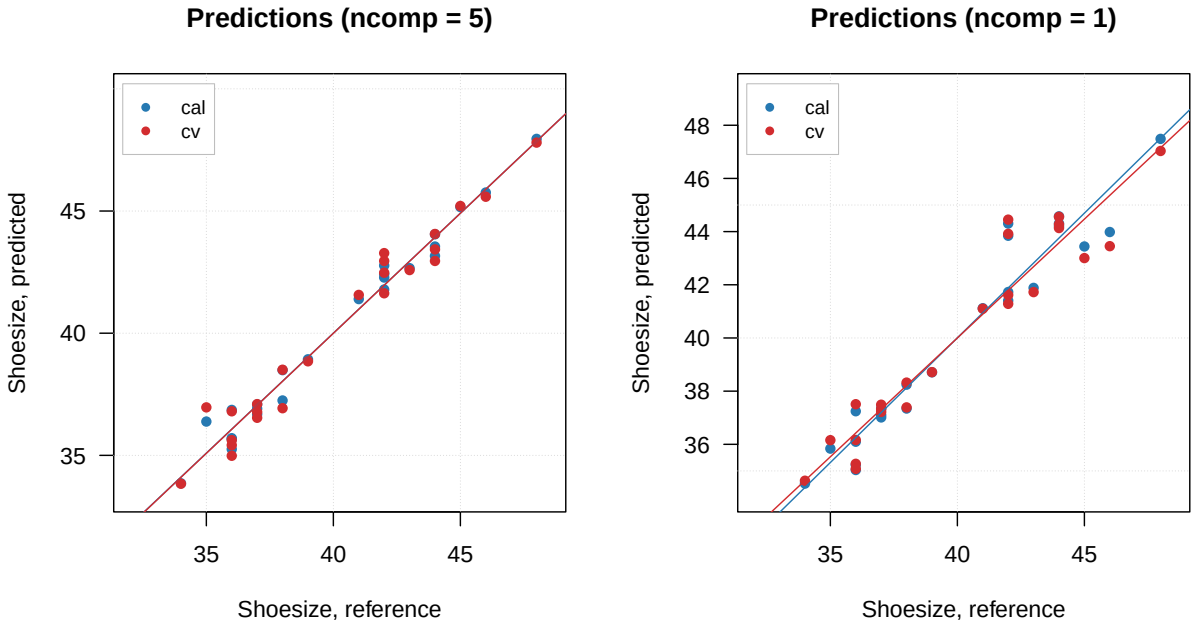
Method	Description
`plotPredictions(obj, ny, ncomp, ...)`	plot with predicted vs. measured (reference) y-values
`plotXScores(obj, comp, ...)`	scores for decomposition of X (similar to PCA plot)
`plotYScores(obj, comp, ...)`	scores for decomposition of Y (similar to PCA plot)
`plotXResiduals(obj, ncomp, ...)`	distance plot for decomposition of X (similar to PCA)
`plotYResiduals(obj, ncomp, ...)`	distance plot for decomposition of Y
`plotXYResiduals(obj, ncomp, ...)`	distance plot for both X and Y decomposition
`plotXYScores(obj, ncomp, ...)`	Y-scores vs. X-scores for a particular PLS component.

Parameter `obj` is either a model or a result object and it is the only mandatory argument for the plots. All other parameters have reasonable default values. Parameter `ny` is used to specify which y-variable you want to see a plot for (if **Y** is multivariate). You can also provide any parameter from `mdaplot()` or `mdaplotg()` thus change limits or labels for axis, main title, colors, line and marker style etc.

Parameter `comp` allows to provide a number of selected components (one or several) to show the plot for, while parameter `ncomp` assumes that only one number is expected (number of components in a model or particular individual component). So if e.g. you create model for five components and selected three as optimal, you can also see, for example, prediction plot for having only one or for components in your model.

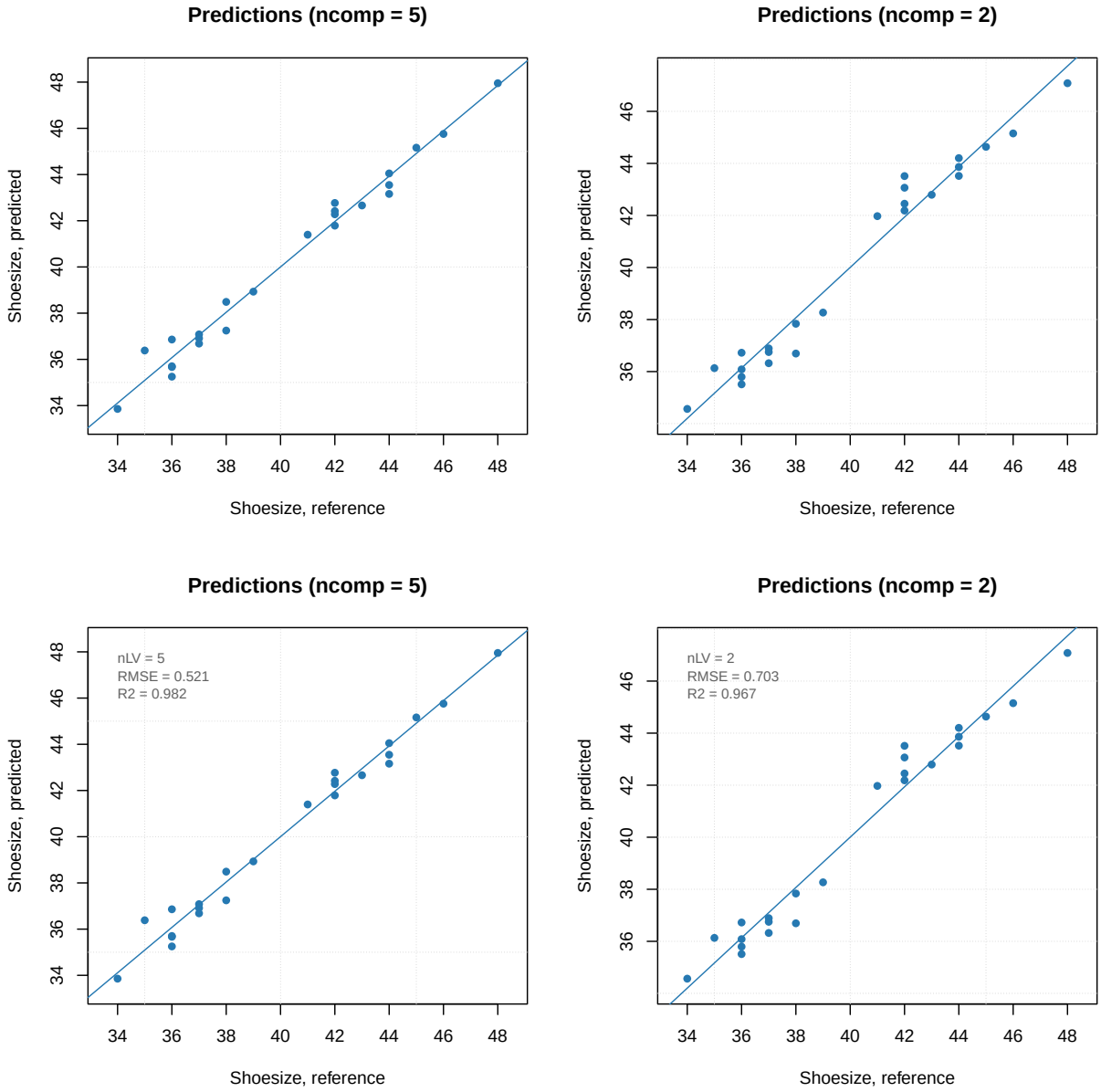
Here is an example for creating prediction plot for `m` model (left for automatically selected number of components, right for user specified value):

```
par(mfrow = c(1, 2))
plotPredictions(m)
plotPredictions(m, ncomp = 1)
```



By the way, when `plotPredictions()` is made for results object, you can show performance statistics on the plot:

```
par(mfrow = c(2, 2))
plotPredictions(m$res$cal)
plotPredictions(m$res$cal, ncomp = 2)
plotPredictions(m$res$cal, show.stat = TRUE)
plotPredictions(m$res$cal, ncomp = 2, show.stat = TRUE)
```

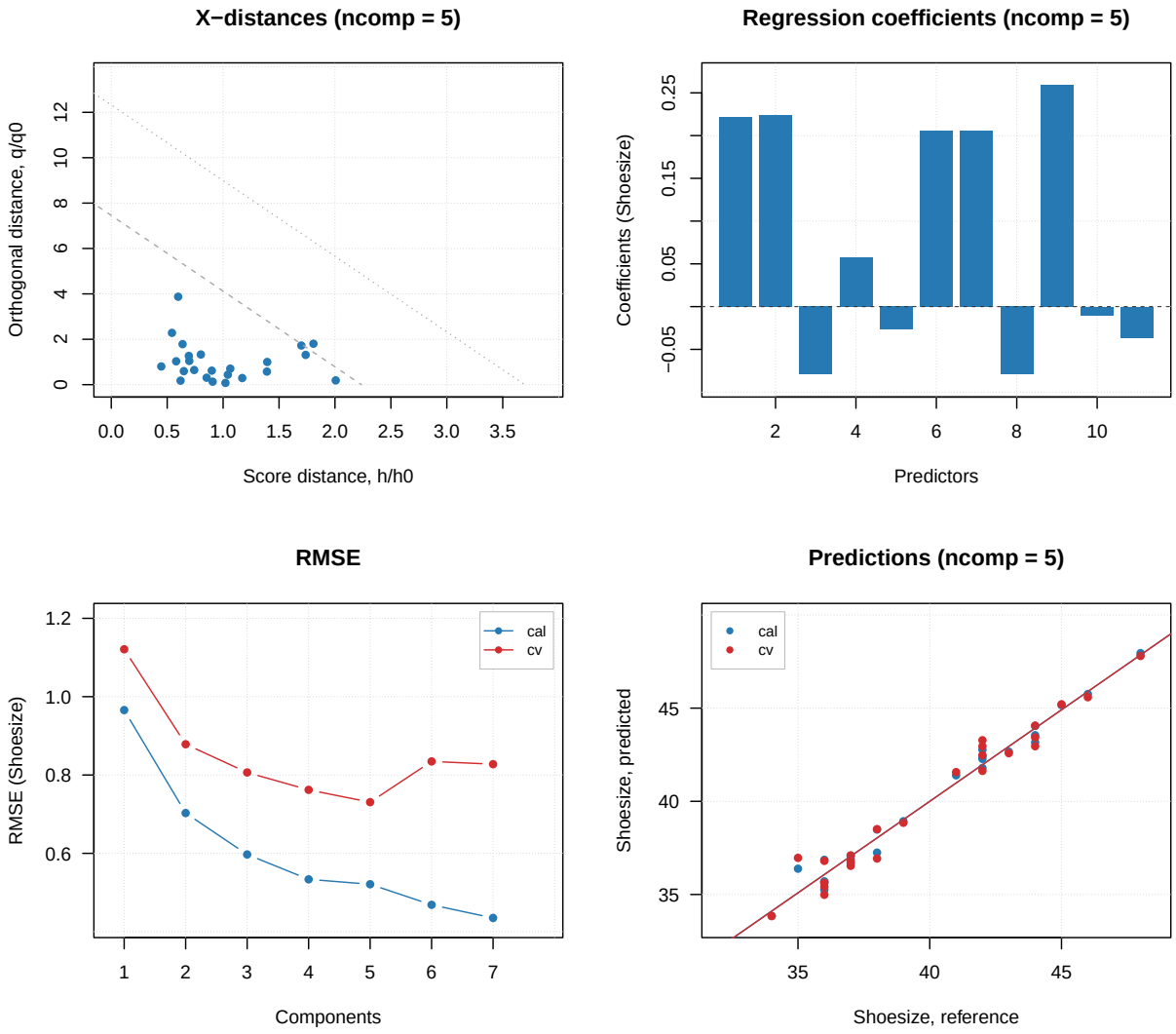


The plots for variables are available only for a model object and include:

Methods	Description
`plotXLoadings(obj, comp)`	loadings plot for decomposition of X
`plotXYLoadings(obj, comp)`	loadings plot for both X and Y decomposition
`plotWeights(obj, comp)`	plot with weights (W) for PLS decomposition
`plotRegcoeffs(obj, ny, ncomp)`	plot with regression coefficients
`plotVIPScores(obj, ny)`	VIP scores plot
`plotSelectivityRatio(obj, ny, ncomp)`	Selectivity ratio plot

And, of course, both model and result objects have method `plot()` for giving an overview.

```
plot(m)
```



Excluding rows and columns

PLS, like PCA, also can exclude rows and columns from calculations. The implementation works similar to what was described for PCA. For example, it can be useful if you have some candidates for outliers or do variable selection and do not want to remove rows and columns physically from the data matrix. In this case you can just specify two additional parameters, `exclcols` and `exclrows`, using either numbers or names of rows/columns to be excluded. You can also specify a vector with logical values (all `TRUE`s will be excluded).

The excluded rows are not used for creating a model and calculation of model's and results' performance (e.g. explained variance). However main results (for PLS — scores, predictions, distances) are calculated for these rows as well and set hidden, so you will not see them on plots. You can always e.g. show scores for excluded objects by using `show.excluded = TRUE`. It is implemented via attributes "known" for plotting methods from *mdatools* so if you use e.g. *ggplot2* you will see all points.

The excluded columns are not used for any calculations either, the corresponding results (e.g. loadings, weights or regression coefficients) will have zero values for such columns and be also hidden on plots.

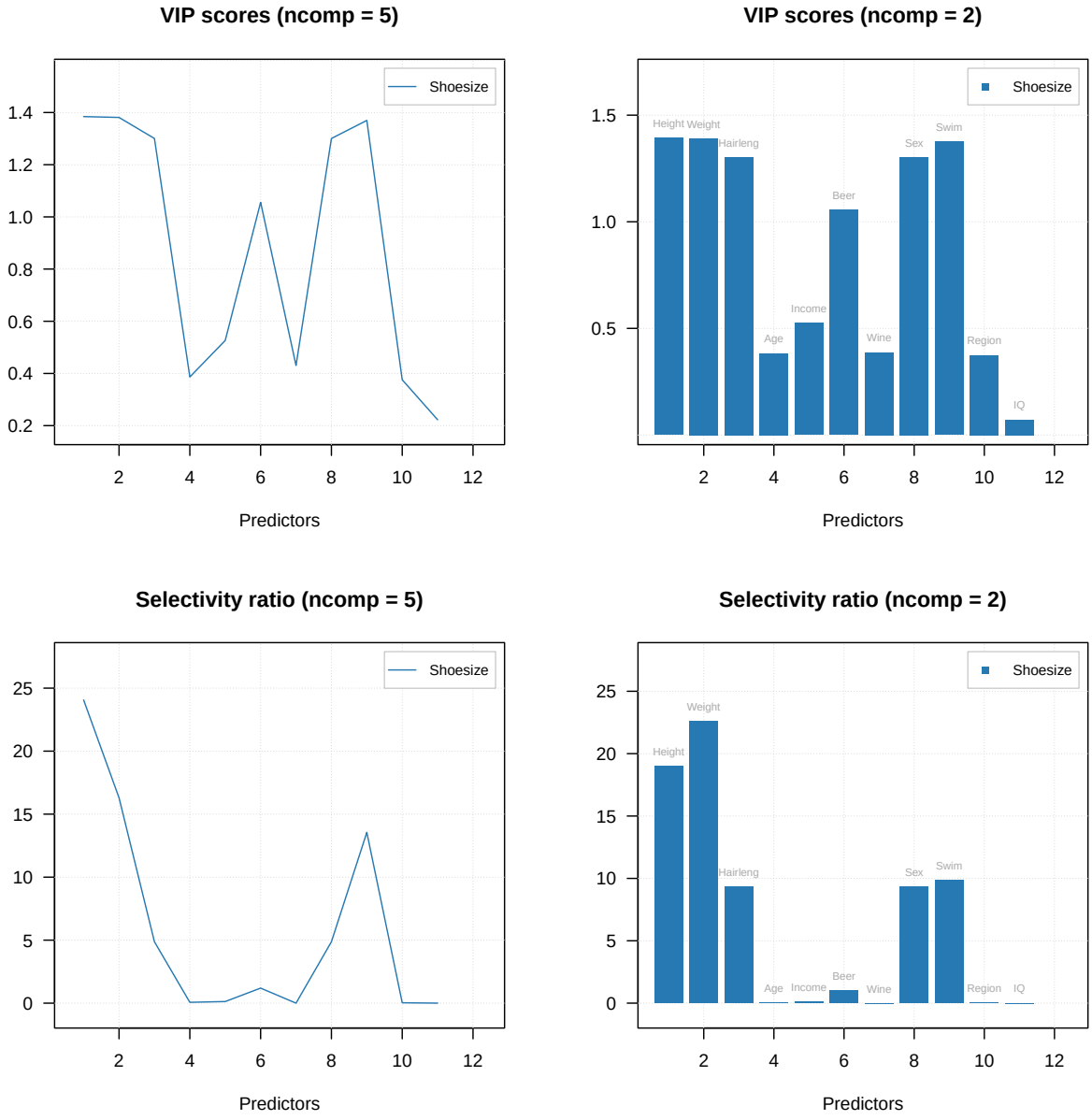
Variable selection

PLS allows to calculate several statistics, which can be used to select most important (or remove least important) variables in order to improve performance and make model simpler. The first two are VIP-scores (variables important for projection) and Selectivity ratio. All details and theory can be found e.g. here.

Both parameters can be shown as plots and as vector of values for a selected y-variable. Take into account that when you make a plot for VIP scores or Selectivity ratio, the corresponding values should be computed first, which can take some time for large datasets.

Here is an example of corresponding plots.

```
par(mfrow = c(2, 2))
plotVIPScores(m)
plotVIPScores(m, ncomp = 2, type = "h", show.labels = TRUE)
plotSelectivityRatio(m)
plotSelectivityRatio(m, ncomp = 2, type = "h", show.labels = TRUE)
```



To compute the values without plotting use `vipscores()` and `selratio()` functions. In the example below, I create two other PLS models by excluding variables with VIP score or selectivity ratio below a threshold (I use 0.5 and 2 correspondingly) and show the performance for both.

```
vip = vipscores(m, ncomp = 2)
m2 = pls(Xc, yc, 4, scale = T, cv = 1, exclcols = (vip < 0.5))
summary(m2)
```

```
##
## PLS model (class pls) summary
## -------------------------------
## Info:
## Number of selected components: 2
## Cross-validation: full (leave one out)
##
```

```
## Response variable: Shoesize
##     X cumexpvar Y cumexpvar    R2  RMSE Slope   Bias  RPD
## Cal    93.51369    95.27241 0.953 0.842 0.953 0.0000 4.70
## Cv           NA          NA 0.933 1.005 0.935 0.0254 3.94
```

```
sr = selratio(m, ncomp = 2)
m2 = pls(Xc, yc, 4, scale = T, cv = 1, exclcols = (sr < 2))
summary(m2)
```

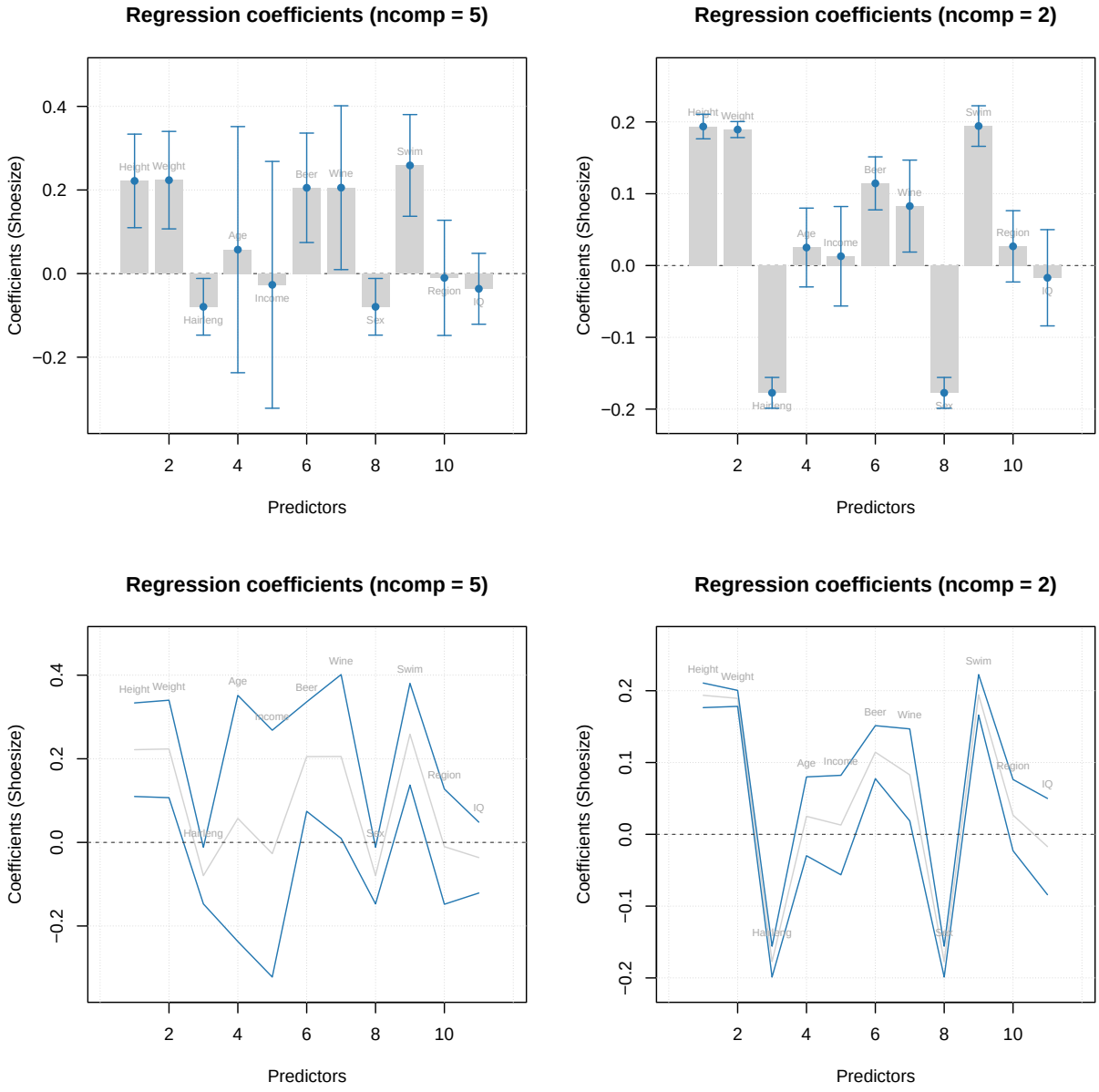
```
##
## PLS model (class pls) summary
## ------------------------------
## Info:
## Number of selected components: 2
## Cross-validation: full (leave one out)
##
## Response variable: Shoesize
##     X cumexpvar Y cumexpvar    R2  RMSE Slope  Bias  RPD
## Cal    99.44331    95.79889 0.958 0.794 0.958 0.000 4.98
## Cv           NA          NA 0.946 0.903 0.944 0.014 4.38
```

Another way is to make an inference about regression coefficients and calculate confidence intervals and p-values for each variable. This can be done using Jack-Knife approach, when model is cross-validated using efficient number of segments (at least ten) and statistics are calculated using the distribution of regression coefficient values obtained for each step. The statistics are automatically computed when you use full cross-validation.

```
mjk = pls(Xc, yc, 7, scale = TRUE, cv = 1)
```

The statistics are calculated for each y-variable and each available number of components. When you show a plot for regression coefficients, you can show the confidence intervals by using parameter `show.ci = TRUE` as shown in examples below.

```
par(mfrow = c(2, 2))
plotRegcoeffs(mjk, type = "h", show.ci = TRUE, show.labels = TRUE)
plotRegcoeffs(mjk, ncomp = 2, type = "h", show.ci = TRUE, show.labels = TRUE)
plotRegcoeffs(mjk, type = "l", show.ci = TRUE, show.labels = TRUE)
plotRegcoeffs(mjk, ncomp = 2, type = "l", show.ci = TRUE, show.labels = TRUE)
```



Calling function `summary()` for regression coefficients allows to get all calculated statistics.

```
summary(mjk$coeffs, ncomp = 2)
```

```
##
## Regression coefficients for Shoesize (ncomp = 2)
## -------------------------------------------------
##               Coeffs   Std. err. t-value p-value        2.5%       97.5%
## Height     0.19357436 0.008270469   23.39   0.000  0.17646559  0.21068313
## Weight     0.18935489 0.005386414   35.12   0.000  0.17821224  0.20049754
## Hairleng -0.17730737 0.010368912  -17.09   0.000 -0.19875710 -0.15585764
## Age        0.02508113 0.026525148    0.95   0.351 -0.02979032  0.07995258
## Income     0.01291497 0.033491494    0.39   0.702 -0.05636746  0.08219740
## Beer       0.11441573 0.017851922    6.40   0.000  0.07748621  0.15134524
## Wine       0.08278735 0.030960265    2.68   0.013  0.01874117  0.14683354
```

```
## Sex      -0.17730737 0.010368912  -17.09   0.000 -0.19875710 -0.15585764
## Swim      0.19426871 0.013654151   14.21   0.000  0.16602295  0.22251448
## Region    0.02673161 0.023989628    1.12   0.275 -0.02289471  0.07635794
## IQ       -0.01705852 0.032399035   -0.53   0.602 -0.08408103  0.04996399
##
## Degrees of freedom (Jack-Knifing): 23
```

```
summary(mjk$coeffs, ncomp = 2, alpha = 0.01)
```

```
##
## Regression coefficients for Shoesize (ncomp = 2)
## ------------------------------------------------
##               Coeffs   Std. err. t-value p-value         0.5%       99.5%
## Height    0.19357436 0.008270469   23.39   0.000  0.170356376  0.21679234
## Weight    0.18935489 0.005386414   35.12   0.000  0.174233417  0.20447636
## Hairleng -0.17730737 0.010368912  -17.09   0.000 -0.206416384 -0.14819835
## Age       0.02508113 0.026525148    0.95   0.351 -0.049383866  0.09954612
## Income    0.01291497 0.033491494    0.39   0.702 -0.081106896  0.10693683
## Beer      0.11441573 0.017851922    6.40   0.000  0.064299387  0.16453206
## Wine      0.08278735 0.030960265    2.68   0.013 -0.004128502  0.16970321
## Sex      -0.17730737 0.010368912  -17.09   0.000 -0.206416384 -0.14819835
## Swim      0.19426871 0.013654151   14.21   0.000  0.155936928  0.23260050
## Region    0.02673161 0.023989628    1.12   0.275 -0.040615325  0.09407855
## IQ       -0.01705852 0.032399035   -0.53   0.602 -0.108013488  0.07389645
##
## Degrees of freedom (Jack-Knifing): 23
```

Function `getRegcoeffs()` in this case may also return corresponding t-value, standard error, p-value, and confidence interval for each of the coefficient (except intercept) if user specifies a parameter `full`. The standard error and confidence intervals are also computed for raw, non-standardized, variables (similar to coefficients).

```
show(getRegcoeffs(mjk, ncomp = 2, full = TRUE))
```

```
##                Estimated    Std. err.     t-value      p-value          2.5%         97.5%
## Intercept  1.342626e+01           NA          NA           NA            NA            NA
## Height     7.456695e-02 3.185875e-03  23.3854388 0.000000e+00  0.0679764667  8.115743e-02
## Weight     4.896328e-02 1.392816e-03  35.1242970 0.000000e+00  0.0460820210  5.184454e-02
## Hairleng  -6.865495e-01 4.014933e-02 -17.0926429 1.443290e-14 -0.7696047104 -6.034943e-01
## Age        1.081716e-02 1.143995e-02   0.9524214 3.507861e-01 -0.0128481769  3.448250e-02
## Income     5.642220e-06 1.463158e-05   0.3875343 7.019238e-01 -0.0000246255  3.590994e-05
## Beer       5.010542e-03 7.817789e-04   6.3971862 1.579903e-06  0.0033933089  6.627775e-03
## Wine       7.511377e-03 2.809055e-03   2.6802915 1.336266e-02  0.0017004042  1.332235e-02
## Sex       -6.865495e-01 4.014933e-02 -17.0926429 1.443290e-14 -0.7696047104 -6.034943e-01
## Swim       1.010496e-01 7.102257e-03  14.2147650 7.018830e-13  0.0863574493  1.157417e-01
## Region     1.035071e-01 9.288992e-02   1.1174095 2.753561e-01 -0.0886503166  2.956646e-01
## IQ        -5.277164e-03 1.002285e-02  -0.5285419 6.021868e-01 -0.0260110098  1.545668e-02
## attr(,"name")
## [1] "Regression coefficients for  Shoesize"
```

It is also possible to change significance level for confidence intervals.

```
show(getRegcoeffs(mjk, ncomp = 2, full = TRUE, alpha = 0.01))
```

```
##                Estimated    Std. err.     t-value      p-value          0.5%         99.5%
## Intercept  1.342626e+01           NA          NA           NA            NA            NA
## Height     7.456695e-02 3.185875e-03  23.3854388 0.000000e+00  6.562313e-02  8.351077e-02
```

```
## Weight     4.896328e-02 1.392816e-03  35.1242970 0.000000e+00  4.505318e-02  5.287338e-02
## Hairleng  -6.865495e-01 4.014933e-02 -17.0926429 1.443290e-14 -7.992621e-01 -5.738369e-01
## Age        1.081716e-02 1.143995e-02   0.9524214 3.507861e-01 -2.129862e-02  4.293295e-02
## Income     5.642220e-06 1.463158e-05   0.3875343 7.019238e-01 -3.543353e-05  4.671797e-05
## Beer       5.010542e-03 7.817789e-04   6.3971862 1.579903e-06  2.815826e-03  7.205258e-03
## Wine       7.511377e-03 2.809055e-03   2.6802915 1.336266e-02 -3.745830e-04  1.539734e-02
## Sex       -6.865495e-01 4.014933e-02 -17.0926429 1.443290e-14 -7.992621e-01 -5.738369e-01
## Swim       1.010496e-01 7.102257e-03  14.2147650 7.018830e-13  8.111117e-02  1.209880e-01
## Region     1.035071e-01 9.288992e-02   1.1174095 2.753561e-01 -1.572661e-01  3.642803e-01
## IQ        -5.277164e-03 1.002285e-02  -0.5285419 6.021868e-01 -3.341467e-02  2.286034e-02
## attr(,"name")
## [1] "Regression coefficients for  Shoesize"
```

The p-values, t-values and standard errors are stored each as a 3-way array similar to regression coefficients. The selection can be made by comparing e.g. p-values with a threshold similar to what we have done with VIP-scores and selectivity ratio.

```
exclcols = mjk$coeffs$p.values[, 2, 1] > 0.05
show(exclcols)
```

```
##   Height   Weight Hairleng      Age   Income     Beer     Wine      Sex     Swim   Region       IQ
##    FALSE    FALSE    FALSE     TRUE     TRUE    FALSE    FALSE    FALSE    FALSE     TRUE     TRUE
```

Here `p.values[, 2, 1]` means values for all predictors, model with two components, first y-variable.

```
newm = pls(Xc, yc, 3, scale = TRUE, cv = 1, exclcols = exclcols)
summary(newm)
```

```
##
## PLS model (class pls) summary
## -------------------------------
## Info:
## Number of selected components: 3
## Cross-validation: full (leave one out)
##
## Response variable: Shoesize
##     X cumexpvar Y cumexpvar    R2  RMSE Slope    Bias  RPD
## Cal    98.63485    97.79127 0.978 0.575 0.978  0.0000 6.87
## Cv           NA          NA 0.965 0.727 0.972 -0.0161 5.44
```

```
show(getRegcoeffs(newm))
```

```
##              Estimated
## Intercept  4.952691046
## Height     0.091079963
## Weight     0.052708955
## Hairleng  -0.315927643
## Age        0.000000000
## Income     0.000000000
## Beer       0.009320427
## Wine       0.018524717
## Sex       -0.315927643
## Swim       0.134354170
## Region     0.000000000
## IQ         0.000000000
## attr(,"exclrows")
##    Age Income Region     IQ
```

```
##      4      5     10     11
## attr(,"name")
## [1] "Regression coefficients for  Shoesize"
```

As you can see, the variables *Age*, *Income*, *Region* and *IQ* have been excluded as they are not related to the *Shoesize*, which seems to be correct.

Variable selection as well as all described above can be also carried out for PLS discriminant analysis (PLS-DA), which can be explained later in one of the next chapters.

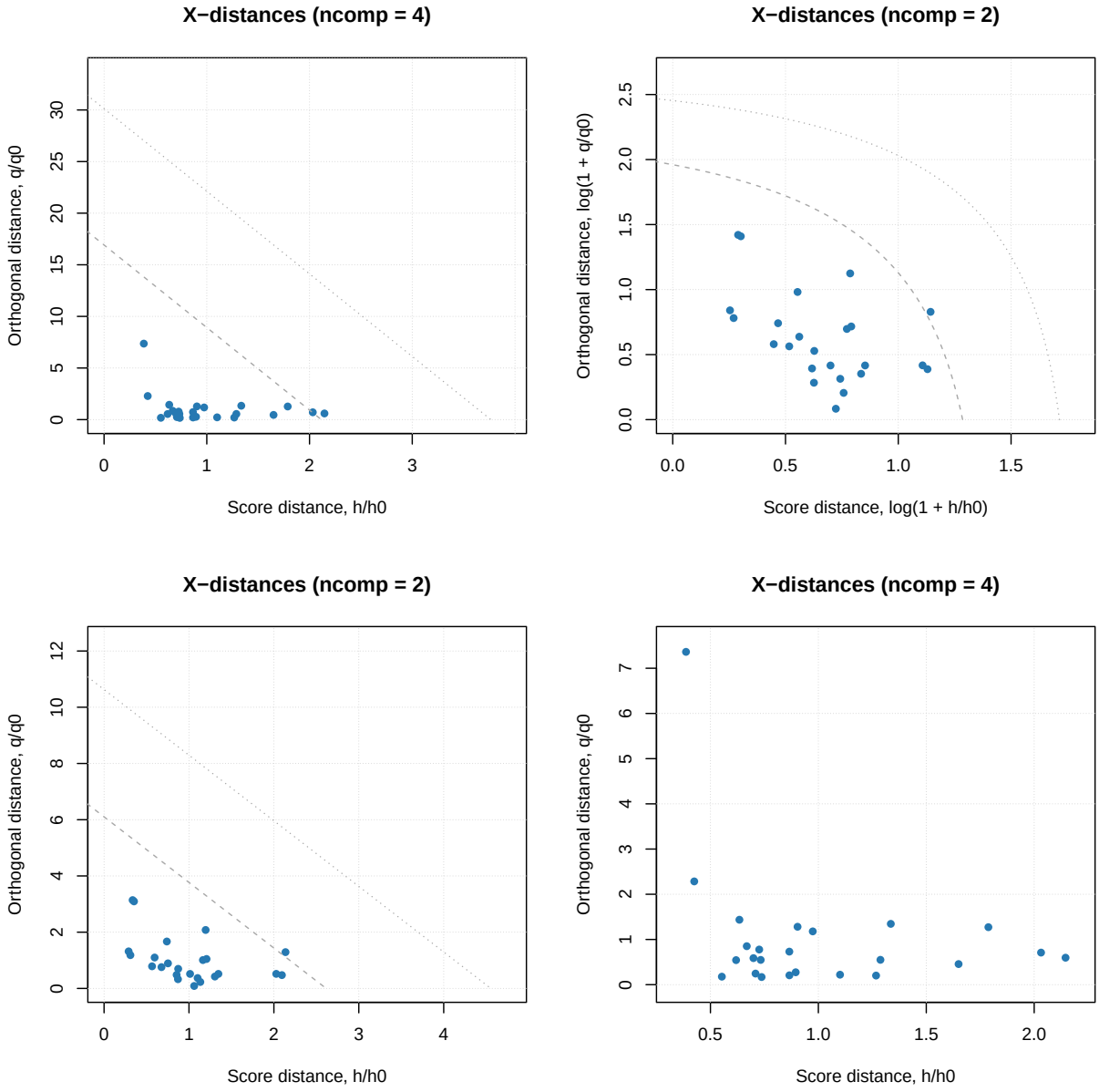
Distances and outlier detection

Distance plot for X-decomposition

For decomposition of X-values the orthogonal and score distances are computed and treated in the same way as in PCA. The `pls()` constructor takes the three arguments for computing the critical limits, similar to PCA (`lim.type`, `alpha`, `gamma`). The default value for `lim.type` is the same as for PCA (`"ddmoments"`). Distance plot can be made using `plotXResiduals()`, it works identical to `plotResiduals()` for PCA.

Below is example on how to use the plot.

```
m = pls(Xc, yc, 4, scale = TRUE, cv = 1)

par(mfrow = c(2, 2))
plotXResiduals(m)
plotXResiduals(m, ncomp = 2, log = TRUE)
plotXResiduals(m, ncomp = 2, res = list("cal" = m$res$cal))
plotXResiduals(m$res$cal)
```



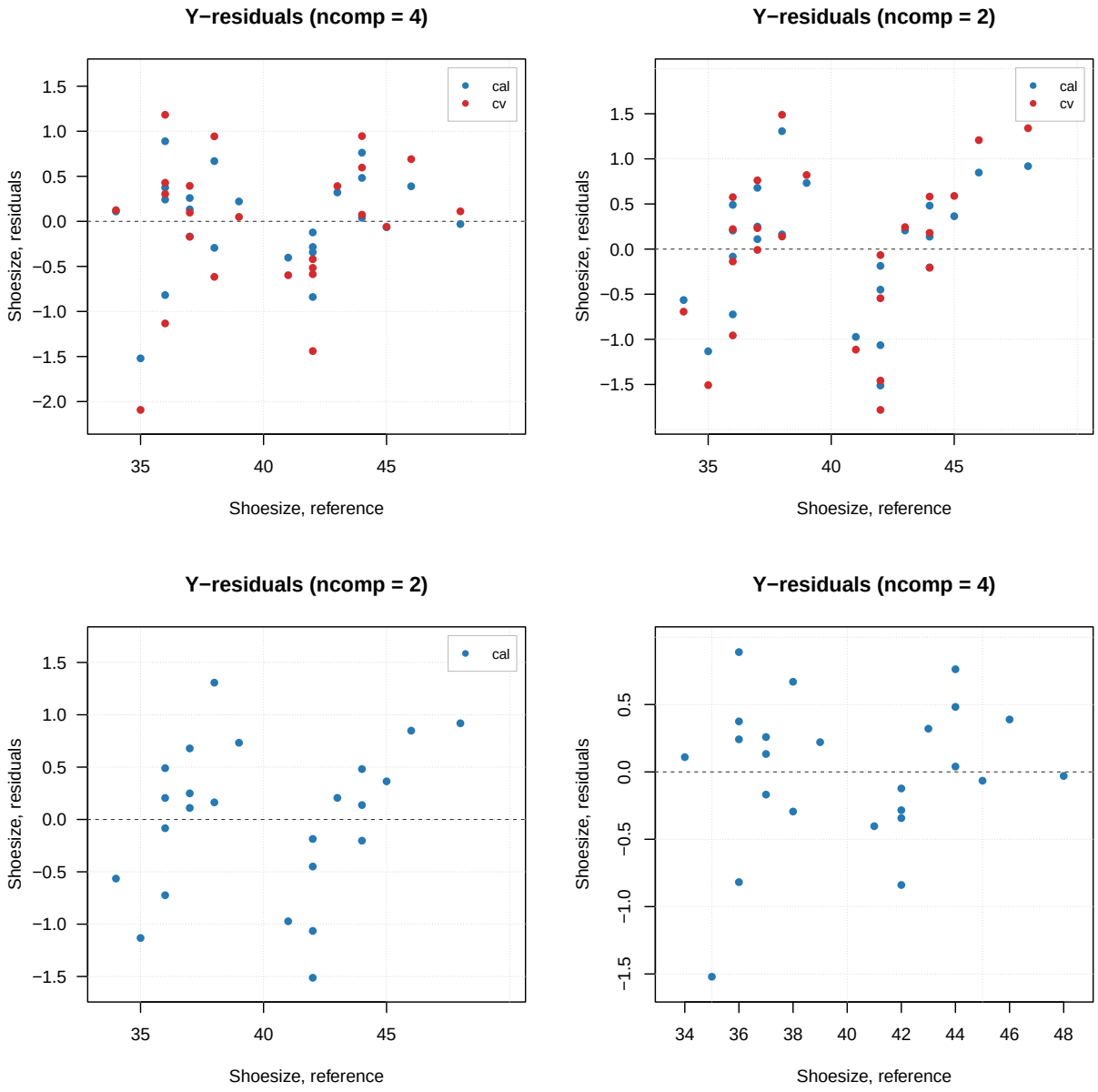
PLS also has function `categorize()` which allows to identify extreme objects and outliers. However, it works in a different way and takes into account not only total distances for X-decomposition but also similar measure for decomposition of Y. This is explained in following sections.

Distance plot for Y-decomposition and total distance

The distance for Y-decomposition is calculated in a different way — as a difference between predicted and reference y-values. The difference for selected y-variable is expected to be random and normally distributed. It can be checked visually by using function `plotYResiduals()` as shown in example below. In this plot the difference is plotted against the reference value.

```
par(mfrow = c(2, 2))
plotYResiduals(m)
plotYResiduals(m, ncomp = 2)
plotYResiduals(m, ncomp = 2, res = list("cal" = m$res$cal))
```

```
plotYResiduals(m$res$cal)
```



Since predictions are also computed for cross-validation, the plot shows the distance values for cross-validated results.

The distance between reference and predicted values can be used to compute orthogonal squared distance, similar to how it is done for X-decomposition, as it is shown below:

$$z_i = \sum_{k=1}^{K} (y_{ik} - \hat{y}_{ik})^2$$

Here y_{ik} is a reference value for sample i and y-variable k and $\hat{y}_{ik}$ is the corresponding predicted value, K is the number of y-variables. Apparently, z values also follow chi-square distribution similar to how it is done in PCA:

$$N_z \frac{z}{z_0} \propto \chi^2(N_z)$$

In case of PLS1 (when there is only one y-variable), $N_z = 1$. Otherwise, both z_0 and N_z are computed using data driven approach, similar to q_0, N_q, h_0, N_h, using either classical (based on moments) or robust approach.

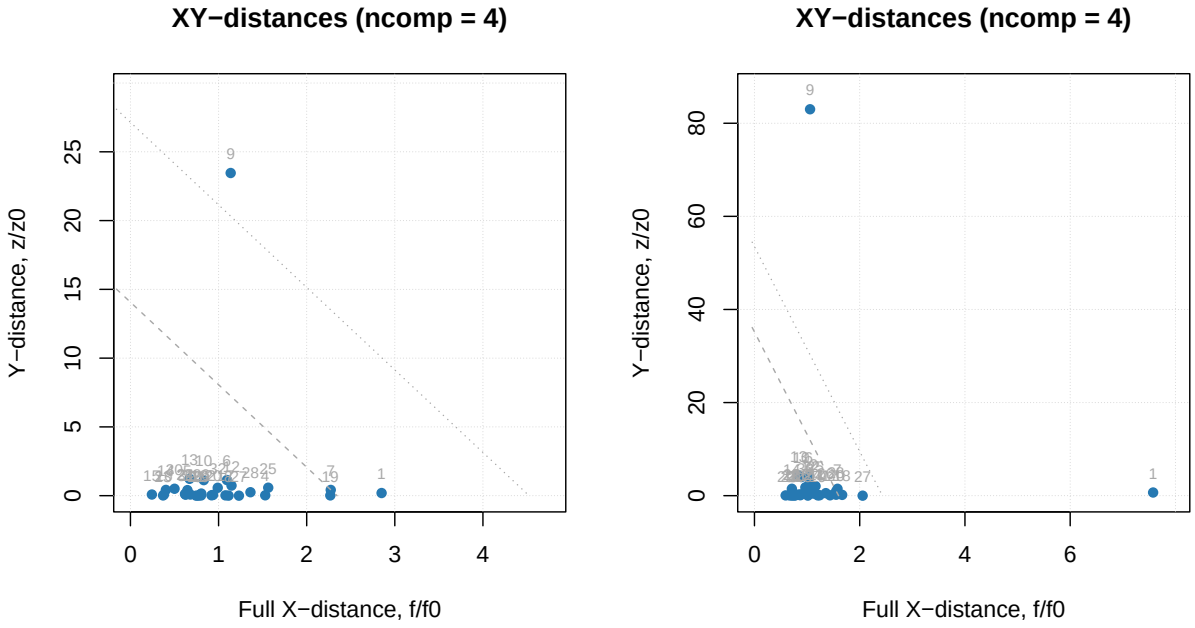
Full distance for X-decomposition, f and the Y-distance, z can be combined into XY total distance, g:

$$g = N_f \frac{f}{f_0} + N_z \frac{z}{z_0} \propto \chi^2(N_g)$$

Here $N_g = N_f + N_z$. This approach was proposed by Rodionova and Pomerantsev and described in this paper. The distances and the critical limits can be visualized using function `plotXYResiduals()` as illustrated in the next example.

The example is based on People data (prediction of shoesize), however we will first introduce two outliers. We will change response value for row #9 to 25, which is apparently too small value for shoesize of an adult person. The predictor values will be the same. Then we will change height of first person (row #1) to 125 cm, which is again quite small value. And keep the response value unchanged. After that, we create a PLS model with classic data driven approach for computing critical limits and show the XY-distance plot. Finally, that we change the method for limits to robust and show the plot again:

```
# prepare data
data(people)
X = people[, -4]
y = people[, 4, drop = FALSE]

# add outliers
y[9] = 25
X[1, 1] = 125

# create model and show plots
m = pls(X, y, 4, scale = TRUE, cv = 1, lim.type = "ddmoments")

par(mfrow = c(1, 2))
plotXYResiduals(m, show.labels = TRUE, labels = "indices")

m = setDistanceLimits(m, lim.type = "ddrobust")
plotXYResiduals(m, show.labels = TRUE, labels = "indices")
```



As one can see, both classic and robust approach detect sample number 9 (with wrong y-value) as a clear outlier. Using robust approach helps to identify the second outlier as well. However, you can see that the location of the two samples is different — sample #9 has large Y-distance (z/z_0), while sample #1 has large full X-distance (f/f_0). Which is in agreement with the way we created the outliers.

The limits shown on the plot are made for the total distance, g. For example, to detect outliers we need to compare the total distance, g, with critical limit computed for predefined significance level, γ (shown as dotted line on the plots):

$$g > \chi^{-2}\big((1-\gamma)^{1/I}, N_g\big)$$

The proper procedure for detection and removing outliers can be found in the paper and will be very briefly touched upon here. The detection can be done using function `categorize()` which works similar to the same function for PCA decomposition — returns vector with categories for each measurement, as shown in an example below:

```
c = categorize(m, m$res$cal)
print(c)
```

```
##  [1] outlier regular regular regular regular regular extreme regular outlier regular regular regular
## [17] regular extreme regular regular regular regular regular regular regular regular extreme regular
## Levels: regular extreme outlier
```

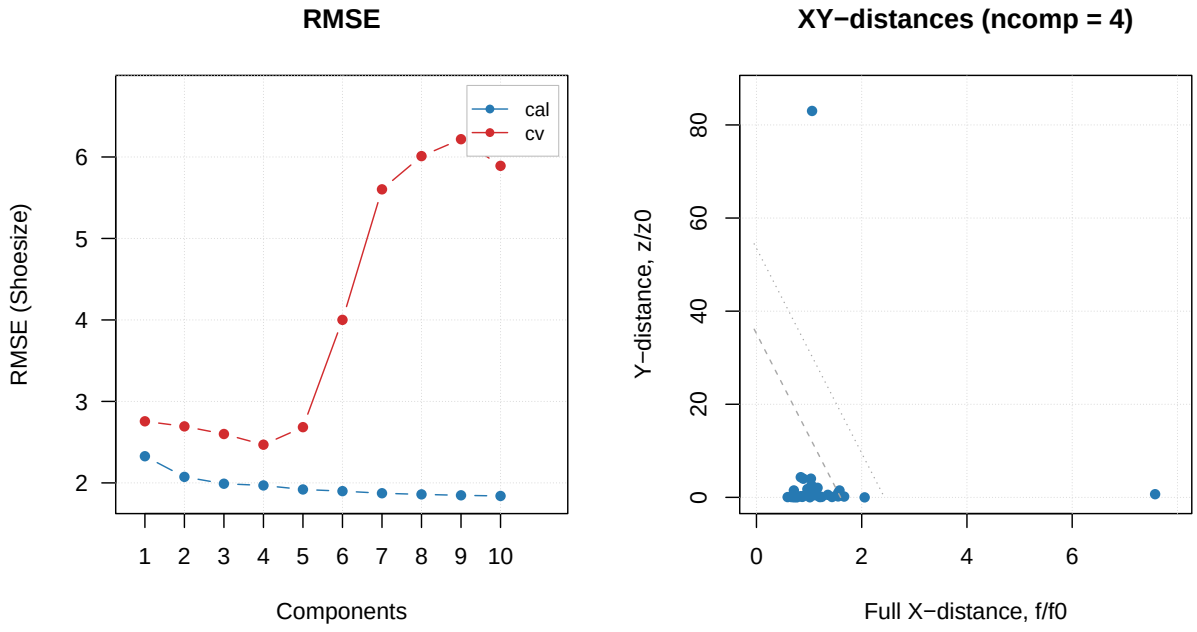
A simplified description of the recommended procedure proposed in the mentioned paper is following:

1. Make a PLS model with robust estimator of critical limits
2. Detect outliers if any, remove them and keep the removed objects separately
3. Repeat steps 1-2 until no outliers are detected
4. Apply model to the collected outliers (use `predict`)
5. If any of them appear as regular objects, return them back to the dataset and go to step 1
6. Repeat all steps until step 5 does not reveal any regular objects
7. Make a final PLS model using classic estimator for the critical limits

If test set validation is used, the outlier detection should be done for both sets. And it is of course important to use optimal number of components, which can be identified using RMSE plot and plot for explained Y-variance.

The code chunks below show the steps for detection of outliers in the People data used in an example earlier. We start with creating PLS model for the whole data.

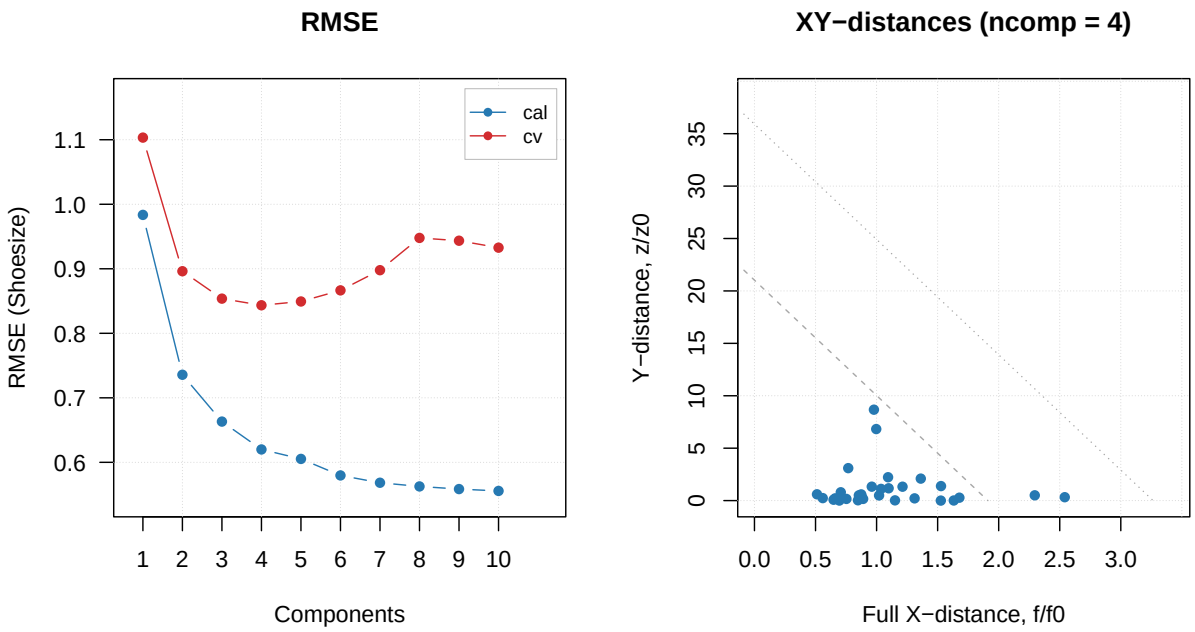
```
# prepare data
data(people)
X = people[, -4]
y = people[, 4, drop = FALSE]

# add outliers
y[9] = 25
X[1, 1] = 125

# compute initial PLS model with all data
m = pls(X, y, 10, scale = TRUE, cv = 1, lim.type = "ddrobust")

# look at RMSE and XY-residuals plot
par(mfrow = c(1, 2))
plotRMSE(m)
plotXYResiduals(m)
```



Apparently four components selected automatically is indeed a proper value. The XY-distance plot shows that we have two outliers, let's find and remove them:

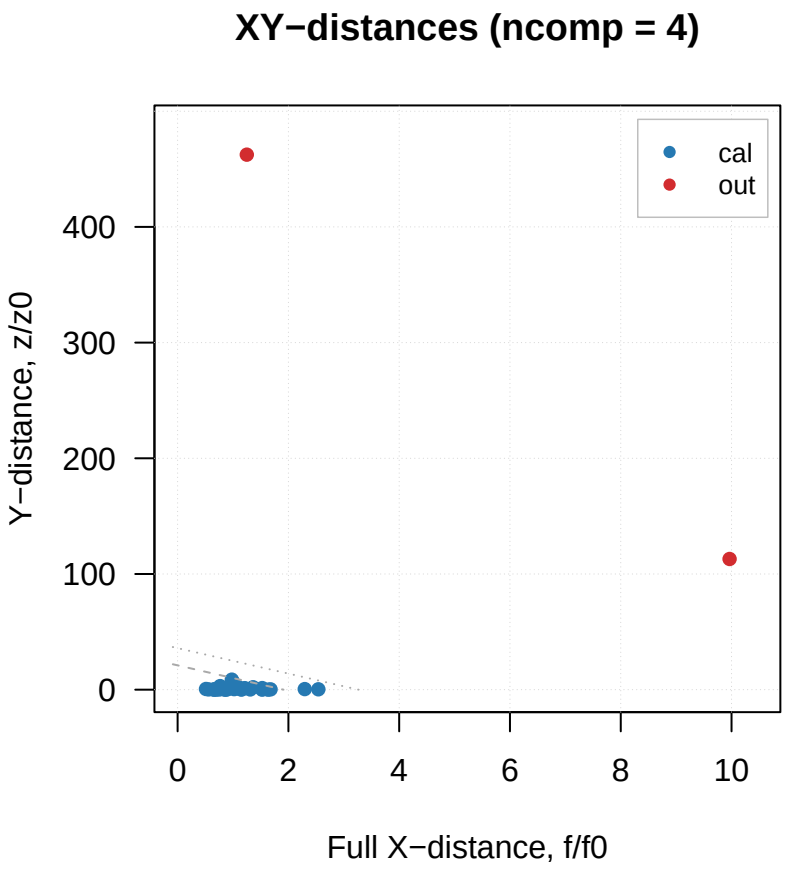
```
# get row indices for outliers in calibration set
outliers = which(categorize(m, m$res$cal) == "outlier")

# keep data for outliers in separate matrices
Xo = X[outliers, , drop = FALSE]
yo = y[outliers, , drop = FALSE]
```

```
# remove the rows with outliers from the data
X = X[-outliers, , drop = FALSE]
y = y[-outliers, , drop = FALSE]

# make a new model for oulier free data
m = pls(X, y, 10, scale = TRUE, cv = 1, lim.type = "ddrobust")

# look at RMSE and XY-distance plot
par(mfrow = c(1, 2))
plotRMSE(m)
plotXYResiduals(m)
```



Again, four or three components seem to be optimal and this time no extra outliers are detected. Let's now apply the model to the two outliers found previously and see their status.

```
res = predict(m, Xo, yo)
plotXYResiduals(m, res = list("cal" = m$res$cal, "out" = res))
```

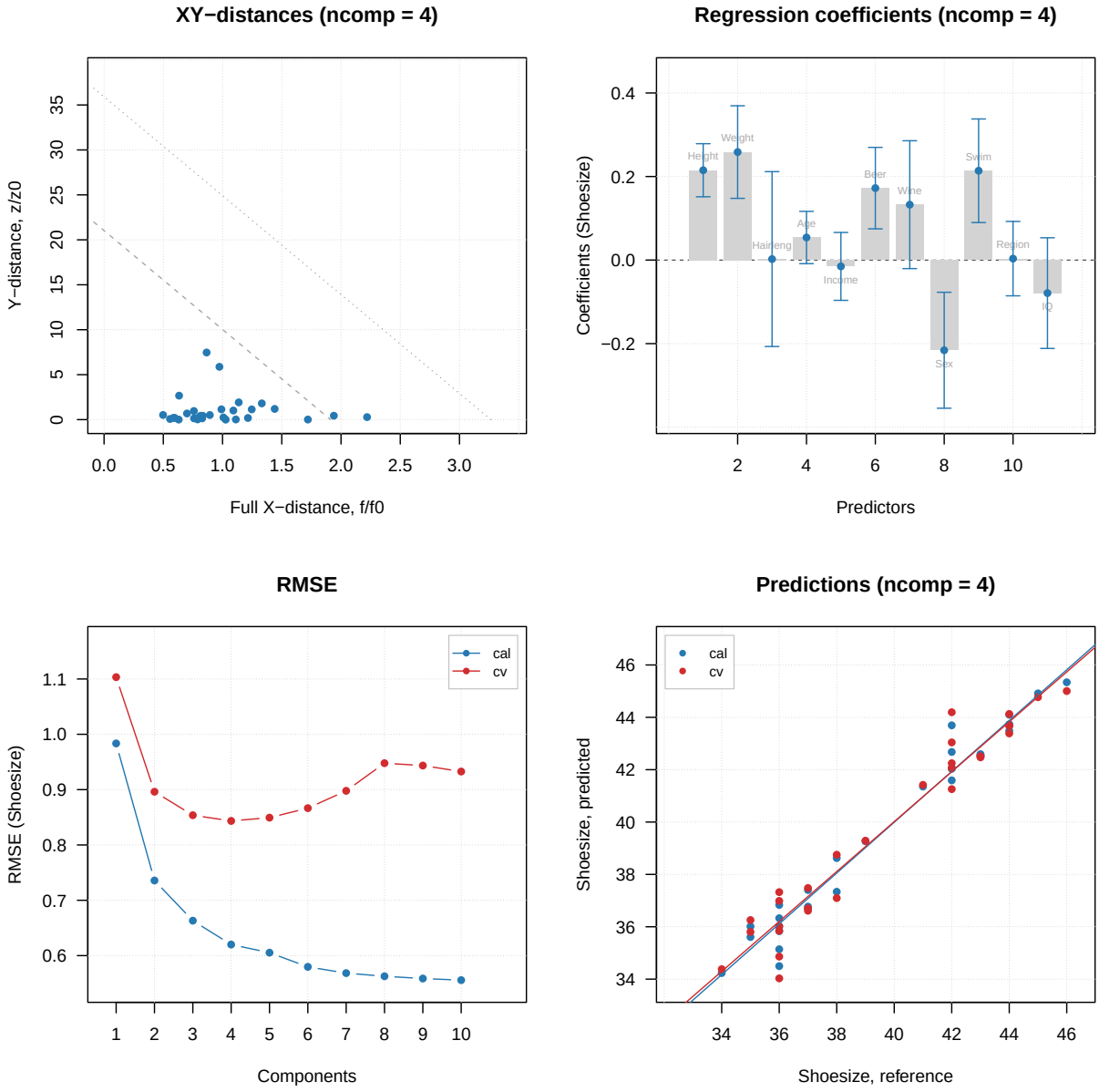


The two objects seem now even more extreme for the model built using the outliers free data, no need to have them back. So we just need to make a final model and look at it:

```
# make a new model for outlier free data and explore it
m = pls(X, y, 10, scale = TRUE, cv = 1, lim.type = "ddmoments")
summary(m)
```

```
##
## PLS model (class pls) summary
## ------------------------------
## Info:
## Number of selected components: 4
## Cross-validation: full (leave one out)
##
## Response variable: Shoesize
##     X cumexpvar Y cumexpvar    R2  RMSE Slope    Bias  RPD
## Cal     85.9168    97.03856 0.970 0.620 0.970  0.0000 5.91
## Cv           NA          NA 0.945 0.843 0.953 -0.0261 4.35
```

```
par(mfrow = c(2, 2))
plotXYResiduals(m)
plotRegcoeffs(m, type = "h", show.labels = TRUE, show.ci = TRUE)
plotRMSE(m)
plotPredictions(m)
```



Randomization test

Another additional option for PLS regression implemented in *mdatools* is randomization test for estimation of optimal number of components. The description of the method can be found in this paper. The basic idea is that for each component from 1 to `ncomp` we compute a statistic T, which is a covariance between X-scores and the reference Y values. After that, this procedure is repeated for randomly permuted Y-values and distribution of the statistic is obtained. A parameter `alpha` is computed to show how often the statistic T, calculated for permuted Y-values, is the same or higher than the same statistic, calculated for original response values without permutations.

If a component is important, then the covariance for non-permuted data should be larger than the covariance for permuted data and therefore the value for `alpha` will be quite small (there is still a small chance to get similar covariance). This makes `alpha` very similar to p-value in a statistical test.

The function `randtest()` calculates alpha for each component, the values can be observed using `summary()` or `plot()` functions. There are also several functions, allowing e.g. to show distribution of statistics and the critical value for each component.

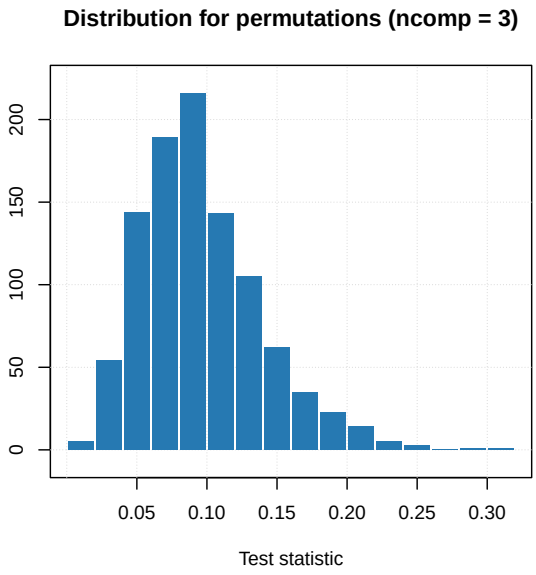
In example of code below most of the functions are shown.

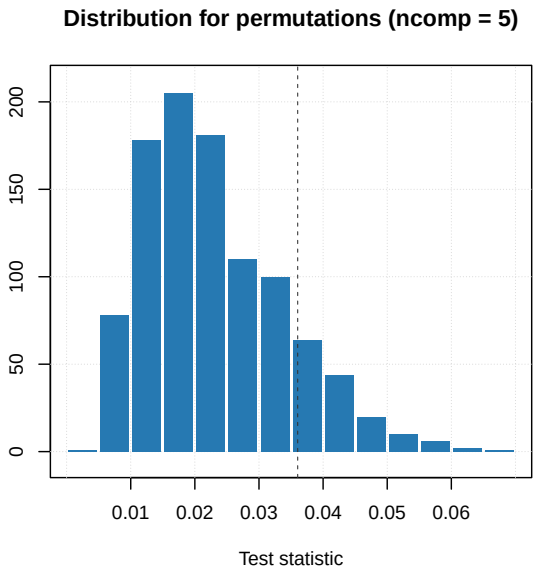
```
data(people)

y = people[, 4, drop = FALSE]
X = people[, -4]

r = randtest(X, y, ncomp = 5, nperm = 1000, silent = TRUE)
summary(r)
```

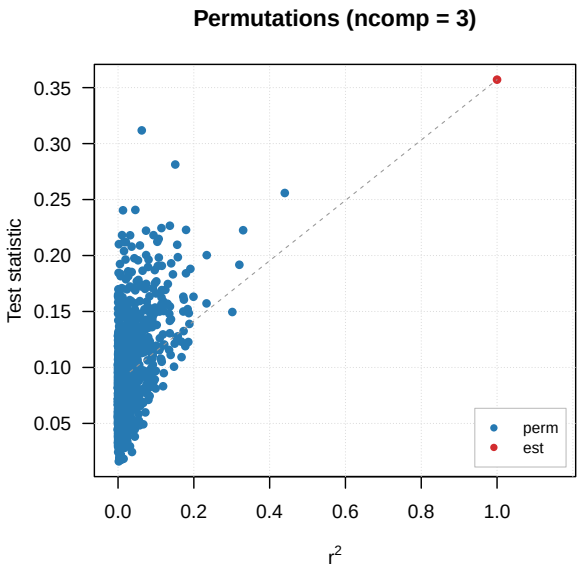
```
##
## Summary for permutation test results
## Number of permutations: 1000
## Suggested number of components: 4
##
## Statistics and alpha values:
##              Comp 1    Comp 2    Comp 3    Comp 4     Comp 5
## Alpha     0.0560000 0.0000000 0.0000000 0.0000000 0.13400000
## Statistic 0.2403837 0.4349094 0.3571243 0.2678767 0.03603819
```

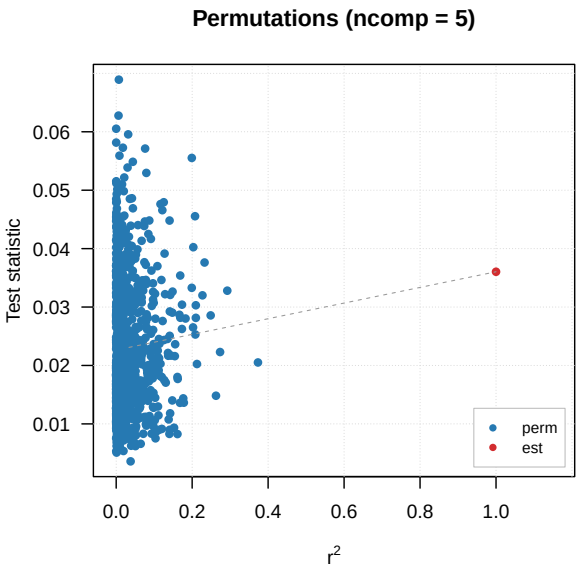
As you can see, `alpha` is very small for components 2–4 and then jumps up.

```
par( mfrow = c(2, 2))
plotHist(r, ncomp = 3)
plotHist(r, ncomp = 5)
plotCorr(r, ncomp = 3)
plotCorr(r, ncomp = 5)
```


Distribution for permutations (ncomp = 3)
0
50
100
150
200
0.05
0.10
0.15
0.20
0.25
0.30
Test statistic


Distribution for permutations (ncomp = 5)
0
50
100
150
200
0.01
0.02
0.03
0.04
0.05
0.06
Test statistic


Permutations (ncomp = 3)
Test statistic
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.0
0.2
0.4
0.6
0.8
1.0
r2
perm
est


Permutations (ncomp = 5)
Test statistic
0.01
0.02
0.03
0.04
0.05
0.06
0.0
0.2
0.4
0.6
0.8
1.0
r2
perm
est

PLS Discriminant Analysis

PLS Discriminant Analysis (PLS-DA) is a discrimination method based on PLS regression. At some point the idea of PLS-DA is similar to logistic regression — we use PLS for a dummy response variable, y, which is equal to +1 for objects belonging to a class, and -1 for those that do not (in some implementations it can also be 1 and 0 correspondingly). Then a conventional PLS regression model is calibrated and validated, which means that all methods and plots, you already used in PLS, can be used for PLS-DA models and results as well.

The extra step in PLS-DA is, actually, classification, which is based on thresholding of predicted y-values. If the predicted value is above 0, a corresponding object is considered as a member of a class and if not — as a stranger. In *mdatools* this is done automatically using methods `plsda()` and `plsdares()`, which inhertit all `pls()` and `plsres()` methods. Plus they have something extra to represent classification results, which you have already read about in the chapter devoted to SIMCA. If you have not, it makes sense to do this first, to make the understanding of PLS-DA implementation easier.

In this chapter we will describe shortly how PLS-DA implementation works. All examples are based on the well-known Iris dataset, which will be split into two subsets — calibration (75 objects, 25 for each class) and validation (another 75 objects). Two PLS-DA models will be built — one only for *virginica* class and one for all three classes.

Calibration of PLS-DA model

Calibration of PLS-DA model is very similar to conventional PLS with one difference — you need to provide information about class membership of each object instead of a matrix or a vector with response values. This can be done in two different ways. If you have multiple classes, it is always recommended to provide your class membership data as a *factor* with predefined labels or a vector with class names as text values. The labels/values in this case will be used as class names. It is also acceptable to use numbers as labels but it will make interpretation of results less readable and can possible cause problems with performance statistics calculations. So use names!

It is very important that you use the same labels/names for e.g. calibration and test set, because this is the way model will identify which class an object came from. And if you have e.g. a typo in a label value, model will assume that the corresponding object is a stranger.

So let's prepare our data.

```
data(iris)

cal.ind = c(1:25, 51:75, 101:125)
val.ind = c(26:50, 76:100, 126:150)

Xc = iris[cal.ind, 1:4]
Xv = iris[val.ind, 1:4]
```

```
cc.all = iris[cal.ind, 5]
cv.all = iris[val.ind, 5]
```

In this case, the fifth column of dataset *Iris* is already factor, otherwise we have to make it as a factor explicitely. Lets check if it is indeed correct.

```
show(cc.all)
```

```
##  [1] setosa     setosa     setosa     setosa     setosa     setosa     setosa     setosa     setosa
## [12] setosa     setosa     setosa     setosa     setosa     setosa     setosa     setosa     setosa
## [23] setosa     setosa     setosa     versicolor versicolor versicolor versicolor versicolor versicol
## [34] versicolor versicolor versicolor versicolor versicolor versicolor versicolor versicolor versicol
## [45] versicolor versicolor versicolor versicolor versicolor versicolor virginica  virginica  virgini
## [56] virginica  virginica  virginica  virginica  virginica  virginica  virginica  virginica  virgini
## [67] virginica  virginica  virginica  virginica  virginica  virginica  virginica  virginica  virgini
## Levels: setosa versicolor virginica
```

However, for the model with just one class, *virginica*, we need to prepare the class variable in a different way. In this case it is enought to provide a vector with logical values, where `TRUE` will correspond to a member and `FALSE` to a non-member of the class. Here is an example how to do it (we will make two — one for calibration and one for validation subsets).

```
cc.vir = cc.all == "virginica"
cv.vir = cv.all == "virginica"
show(cc.vir)
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [22] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [43] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [64]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

Now we can calibrate the models:

```
m.all = plsda(Xc, cc.all, 3, cv = 1)
m.vir = plsda(Xc, cc.vir, 3, cv = 1, classname = "virginica")
```

You could notice one important difference. In case when parameter `c` is a vector with logical values you also need to provide a name of the class. If you do not do this, a default name will be used, but it may cause problems when you e.g. validate your model using test set where class membership is a factor as we have in this example.

Let's look at the summary for each of the model. As you can see below, summary for multi class PLS-DA simply shows one set of results for each class. The performance statistics include explained X and Y variance (cumulative), values for confusion matrix (true positives, false positives, true negatives, false negatives) as well as specificity, sensitivity and accuracy values.

```
summary(m.all)
```

```
##
## PLS-DA model (class plsda) summary
## -----------------------------------
## Info:
## Number of selected components: 1
## Cross-validation: full (leave one out)
##
## Class #1 (setosa)
##     X cumexpvar Y cumexpvar TP FP TN FN Spec. Sens. Accuracy
## Cal       91.97       46.36 25  0 50  0     1     1        1
```

```
## Cv           NA          NA 25  0 50  0     1     1        1
##
## Class #2 (versicolor)
##     X cumexpvar Y cumexpvar TP FP TN FN Spec. Sens. Accuracy
## Cal       91.97       46.36  0  0 50 25  1.00     0    0.667
## Cv           NA          NA  0  3 47 25  0.94     0    0.627
##
## Class #3 (virginica)
##     X cumexpvar Y cumexpvar TP FP TN FN Spec. Sens. Accuracy
## Cal       91.97       46.36 24  5 45  1  0.90  0.96    0.920
## Cv           NA          NA 24  6 44  1  0.88  0.96    0.907
```

Dealing with the multi-class PLS-DA model is similar to dealing with PLS2 models, when you have several y-variables. Every time you want to show a plot or results for a particular class, just provide the class number using parameter `nc`. For example this is how to show summary only for the third class (*virginica*).

```
summary(m.all, nc = 3)
```

```
##
## PLS-DA model (class plsda) summary
## -----------------------------------
## Info:
## Number of selected components: 1
## Cross-validation: full (leave one out)
##
## Class #3 (virginica)
##     X cumexpvar Y cumexpvar TP FP TN FN Spec. Sens. Accuracy
## Cal       91.97       46.36 24  5 45  1  0.90  0.96    0.920
## Cv           NA          NA 24  6 44  1  0.88  0.96    0.907
```

You can also show statistics only for calibration or only for cross-validation parts, in this case you will see details about contribution of every component to the model.

```
summary(m.all$calres, nc = 3)
```

```
##
## PLS-DA results (class plsdares) summary:
## Number of selected components: 1
##
## Class #3 (virginica):
##        X expvar X cumexpvar Y expvar Y cumexpvar TP FP TN FN Spec. Sens. Accuracy
## Comp 1   91.969      91.969   46.356      46.356 24  5 45  1  0.90  0.96    0.920
## Comp 2    5.500      97.468    6.878      53.233 21  5 45  4  0.90  0.84    0.880
## Comp 3    2.186      99.654    4.816      58.049 24  3 47  1  0.94  0.96    0.947
```

For one class models, the behaviour is actually similar, but there will be always one set of results — for the corresponding class. Here is the summary.

```
summary(m.vir)
```

```
##
## PLS-DA model (class plsda) summary
## -----------------------------------
## Info:
## Number of selected components: 3
## Cross-validation: full (leave one out)
##
## Class #1 (virginica)
```

```
##     X cumexpvar Y cumexpvar TP FP TN FN Spec. Sens. Accuracy
## Cal       98.53       61.31 24  3 47  1  0.94  0.96    0.947
## Cv           NA          NA 24  3 47  1  0.94  0.96    0.947
```

Like in SIMCA you can also get a confusion matrix for particular result. Here is an example for multiple classes model.

```
getConfusionMatrix(m.all$calres)
```

```
##            setosa versicolor virginica None
## setosa         25          0         0    0
## versicolor      0          0         5   20
## virginica       0          0        24    1
```

And for the one-class model.

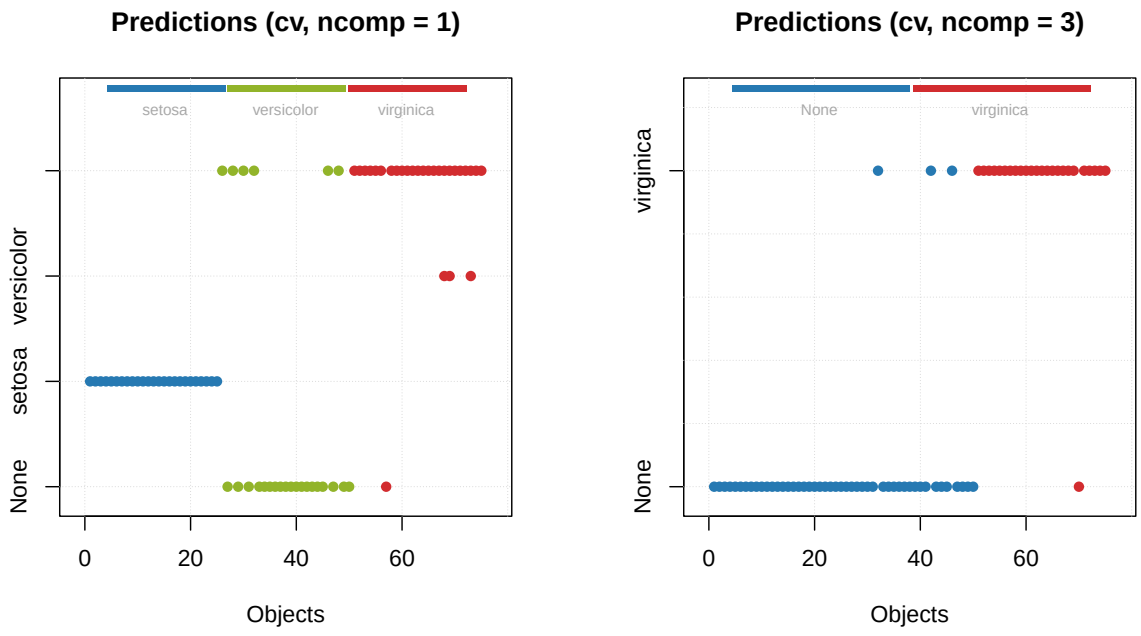
```
getConfusionMatrix(m.vir$calres)
```

```
##           virginica None
## virginica        24    1
## None              3   47
```

Classification plots

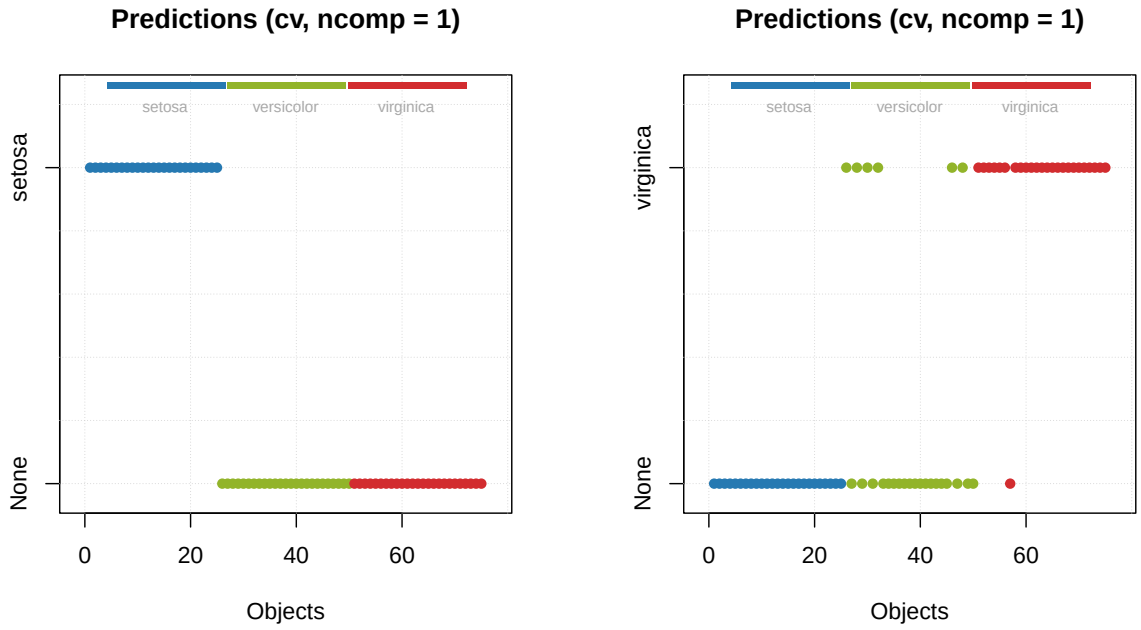
Most of the plots for visualisation of classification results described in SIMCA chapter can be also used for PLS-DA models and results. Let's start with classification plots. By default it is shown for cross-validation results (we change position of the legend so it does not hide the points). You can clearly spot for example three false positives and one false negatives in the one-class PLS-DA model for virginica.

```
par(mfrow = c(1, 2))
plotPredictions(m.all)
plotPredictions(m.vir)
```



In case of multiple classes model you can select which class to show the predictions for.

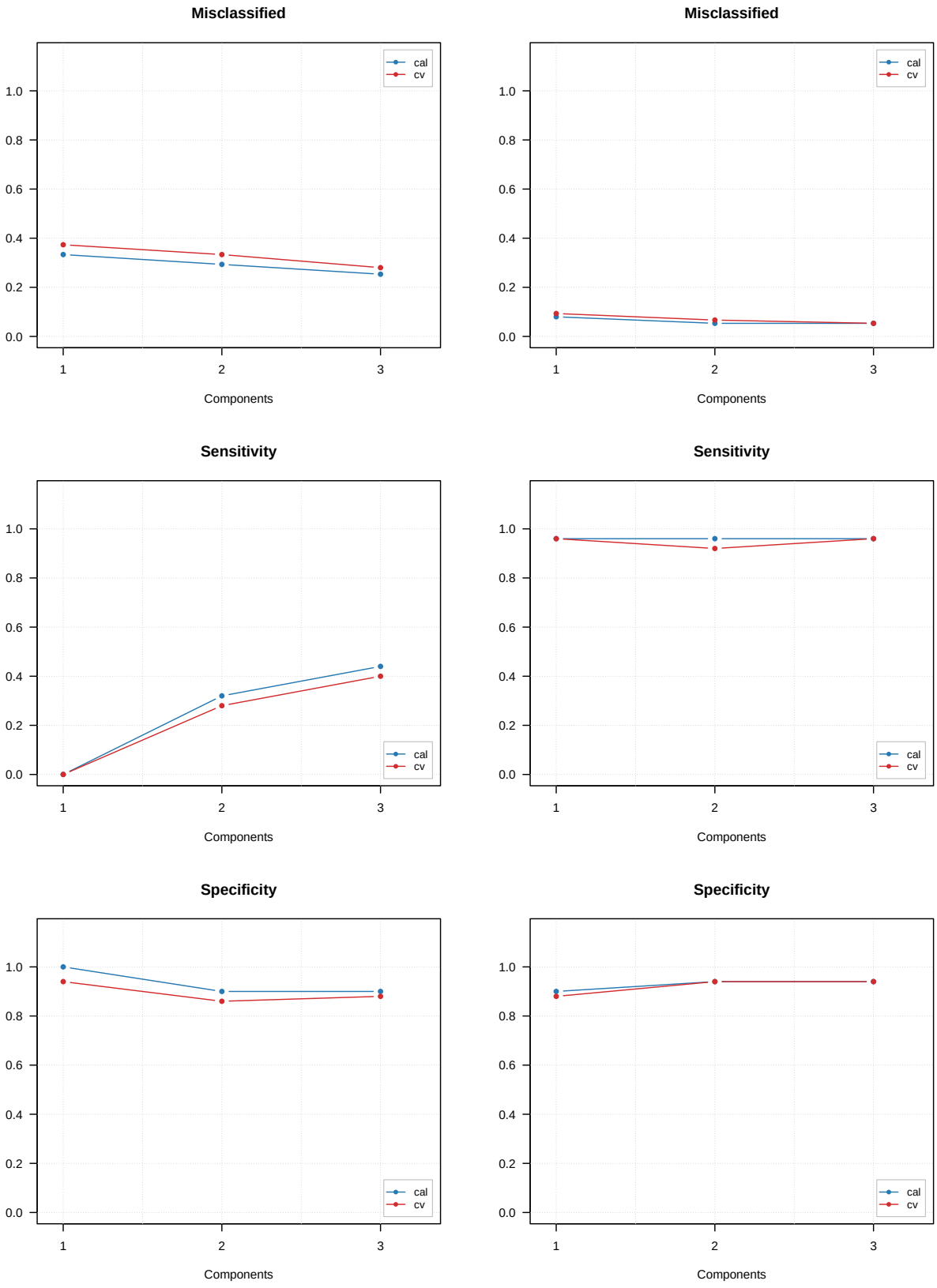
```
par(mfrow = c(1, 2))
plotPredictions(m.all, nc = 1)
plotPredictions(m.all, nc = 3)
```



Performance plots

As in SIMCA you can show how sensitivity, specificity and total amount of misclassified samples depending on number of components by using corresponding plots. In case of multiple-classes model you can also provide a class number to show the plot for (by default package will show the plot for overall statistic computed for all classes).

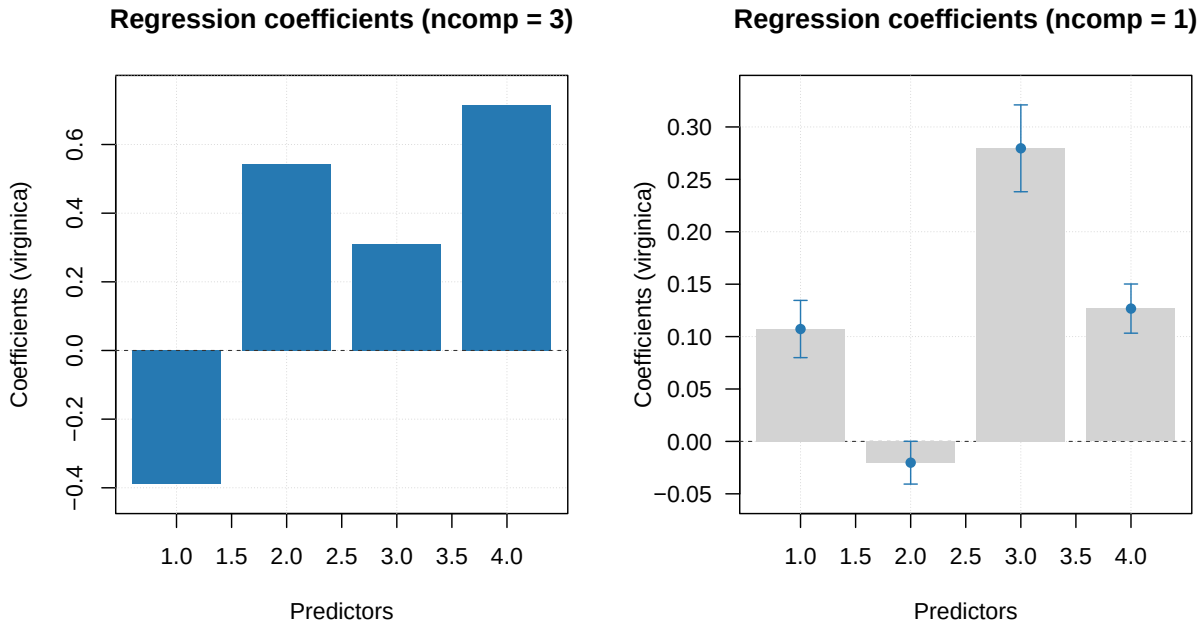
```
par(mfrow = c(3, 2))
plotMisclassified(m.all, nc = 2)
plotMisclassified(m.vir)
plotSensitivity(m.all, nc = 2)
plotSensitivity(m.vir)
plotSpecificity(m.all, nc = 2)
plotSpecificity(m.vir)
```



As usual, you can also change type of plot to line or scatter-line, change colors, etc. All PLS regression plots,

including RMSE, X and Y variance, etc. will also work smoothly with PLS-DA models or results. You can also show regression coefficients like in the example below. To show regression coefficients for particular class you need to provide its number using parameter `ny` (the reason we use `ny` and not `nc` here is that this plot is inherited from PLS model). So in our example both plots show regression coefficients for prediction of virginica objects (left obtained using multiple class model and right — using one class model).

```
par(mfrow = c(1, 2))
plotRegcoeffs(m.all, ncomp = 3, ny = 3)
plotRegcoeffs(m.vir, ncomp = 1, show.ci = TRUE)
```



Predictions for a new data

Again very similar to PLS — just use method `predict()` and provide at least matrix or data frame with predictors (which should contain the same number of variables/columns). For test set validation you can also provide class reference information similar to what you have used for calibration of PLS-DA models.

In case of multiple class model, the reference values should be provided as a factor or vector with class names as text values. Here is an example.

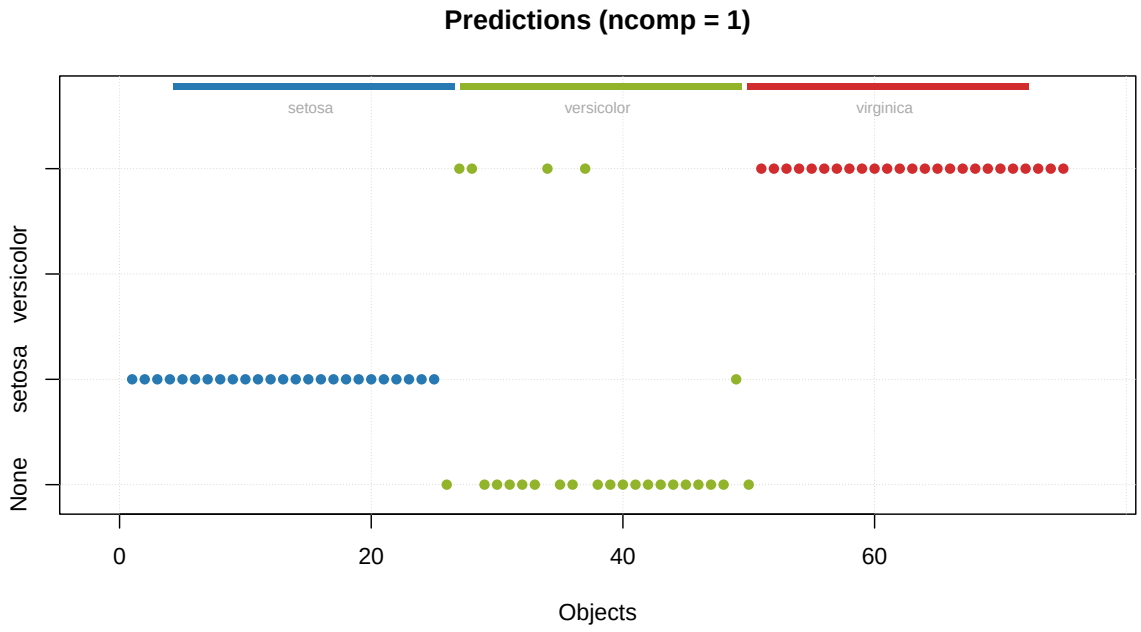
```
res = predict(m.all, Xv, cv.all)
summary(res)
```

```
##
## PLS-DA results (class plsdares) summary:
## Number of selected components: 1
##
## Class #1 (setosa):
##        X expvar X cumexpvar Y expvar Y cumexpvar TP FP TN FN Spec. Sens. Accuracy
## Comp 1   92.924      92.924   42.703      42.703 25  1 49  0  0.98     1    0.987
## Comp 2    4.560      97.484   11.216      53.920 25  0 50  0  1.00     1    1.000
## Comp 3    1.790      99.274    1.717      55.637 25  0 50  0  1.00     1    1.000
##
##
```

```
## Class #2 (versicolor):
##        X expvar X cumexpvar Y expvar Y cumexpvar TP FP TN FN Spec. Sens. Accuracy
## Comp 1   92.924      92.924   42.703      42.703  0  0 50 25  1.00   0.0    0.667
## Comp 2    4.560      97.484   11.216      53.920 10  4 46 15  0.92   0.4    0.747
## Comp 3    1.790      99.274    1.717      55.637 10  6 44 15  0.88   0.4    0.720
##
##
## Class #3 (virginica):
##        X expvar X cumexpvar Y expvar Y cumexpvar TP FP TN FN Spec. Sens. Accuracy
## Comp 1   92.924      92.924   42.703      42.703 25  4 46  0  0.92  1.00    0.947
## Comp 2    4.560      97.484   11.216      53.920 25  4 46  0  0.92  1.00    0.947
## Comp 3    1.790      99.274    1.717      55.637 24  4 46  1  0.92  0.96    0.933
```

And the corresponding plot with predictions.

```
par(mfrow = c(1, 1))
plotPredictions(res)
```



If vector with reference class values contains names of classes model knows nothing about, they will simply be considered as members of non of the known clases ("None").

In case of one-class model, the reference values can be either factor/vector with names or logical values, like the ones used for calibration of the model. Here is an example for each of the cases.

```
res21 = predict(m.vir, Xv, cv.all)
summary(res21)
```

```
##
## PLS-DA results (class plsdares) summary:
## Number of selected components: 3
##
## Class #1 (virginica):
##        X expvar X cumexpvar Y expvar Y cumexpvar TP FP TN FN Spec. Sens. Accuracy
## Comp 1   93.107      93.107   54.394      54.394 25  4 46  0  0.92  1.00    0.947
```

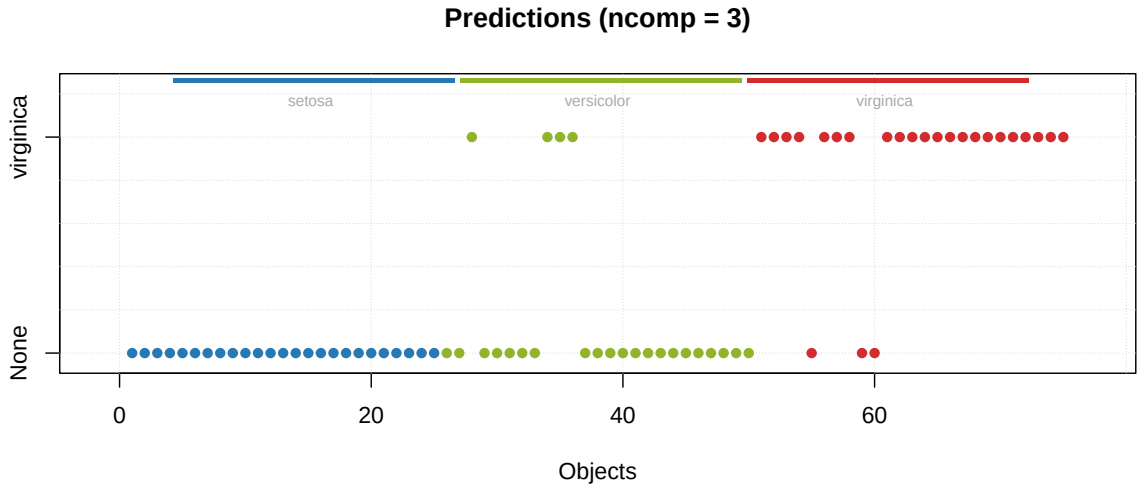
```
## Comp 2    1.588      94.695    6.039      60.433 24  4 46  1  0.92  0.96    0.933
## Comp 3    2.641      97.336   -0.149      60.284 22  4 46  3  0.92  0.88    0.907
```

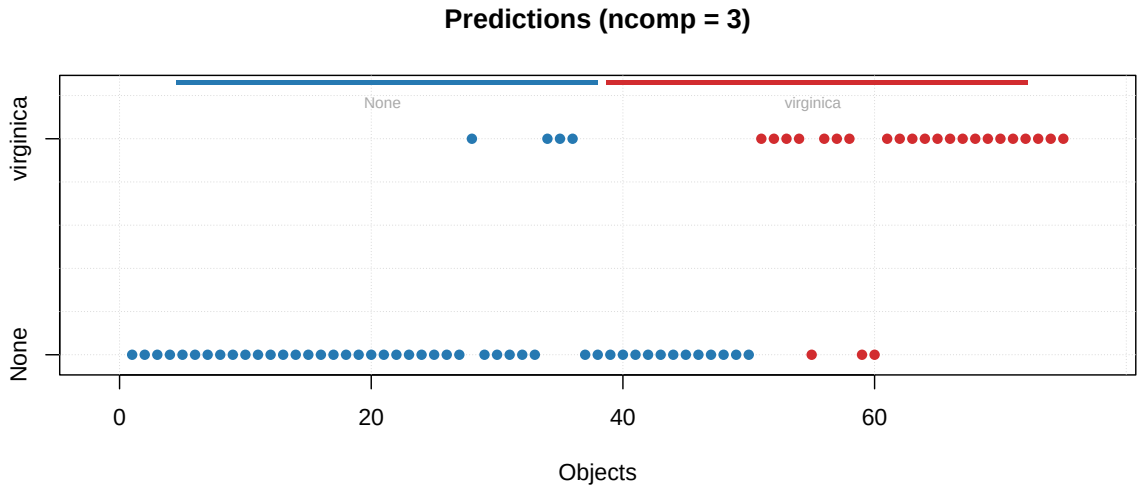
```
res22 = predict(m.vir, Xv, cv.vir)
summary(res22)
```

```
##
## PLS-DA results (class plsdares) summary:
## Number of selected components: 3
##
## Class #1 (virginica):
##        X expvar X cumexpvar Y expvar Y cumexpvar TP FP TN FN Spec. Sens. Accuracy
## Comp 1   93.107      93.107   54.394      54.394 25  4 46  0  0.92  1.00    0.947
## Comp 2    1.588      94.695    6.039      60.433 24  4 46  1  0.92  0.96    0.933
## Comp 3    2.641      97.336   -0.149      60.284 22  4 46  3  0.92  0.88    0.907
```

As you can see, statistically results are identical. However, predictions plot will look a bit different for these two cases, as you can see below.

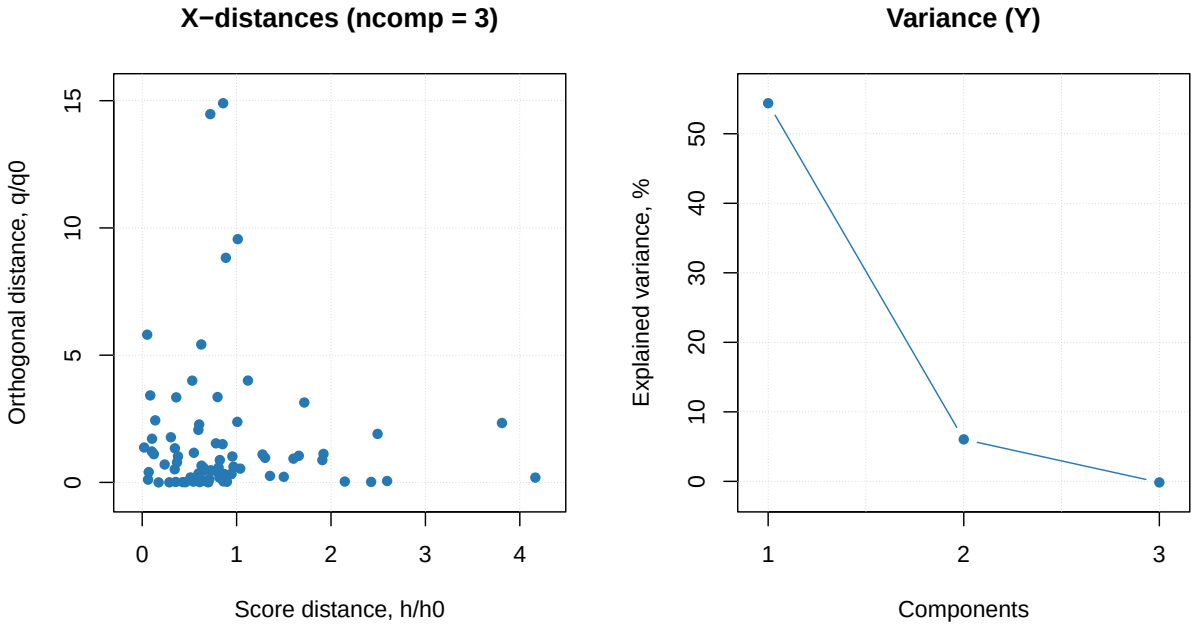
```
par(mfrow = c(2, 1))
plotPredictions(res21)
plotPredictions(res22)
```





And because `predict()` returns an object with results you can also use most of the plots available for PLS regression results. In the last example below you will find plots for X-distance and Y-variance.

```
par(mfrow = c(1, 2))
plotXResiduals(res21)
plotYVariance(res22)
```


X−distances (ncomp = 3)
Orthogonal distance, q/q0
0
5
10
15
0
1
2
3
4
Score distance, h/h0
Variance (Y)
Explained variance, %
0
10
20
30
40
50
1
2
3
Components

SIMCA/DD-SIMCA classification

SIMCA (Soft Independent Modelling of Class Analogy) is a simple but efficient one-class classification method mainly based on PCA. The general idea is to create a PCA model using only samples/objects belonging to a class and classify new objects based on how good the model can fit them. The decision is made using the two distances we discussed in detail in the corresponding PCA chapter — orthogonal and score distances and corresponding critical limits.

Critical limits computed for both distances (or their combination) are used to cut-off the strangers (extreme objects) and accept class members with a pre-define expected ratio of false negatives (α). If data driven approach (either classic/moments or robust) are used to compute the critical limits, then the method is called **DD-SIMCA** (Data Driven SIMCA). You can find more details about the method in this paper.

The classification performance can be assessed using number of true/false positives and negatives and statistics, showing the ability of a classification model to recognize class members (*sensitivity* or true positive rate) and how good the model is for identifying strangers (*specificity* or true negative rate). In addition to that, model also calculates a percent of misclassified objects. All statistics are calculated for calibration and validation (if any) results, but one must be aware that specificity can not be computed without objects not belonging to the class and, therefore, calibration and cross-validation results in SIMCA do not have specificity values.

> You can think that SIMCA is actually a PCA model where function `categorize()` is used to make a decision: if object is categorized as regular, it will be considered as member of the class, otherwise — it is a stranger. Therefore read carefully how PCA works in general and how critical limits for distances are computed in particular, to understand how SIMCA works.

It must be also noted that any SIMCA model is also a PCA model object and any SIMCA result is also a PCA result object, therefore all plots, methods, statistics, available for PCA, can be used for SIMCA model and result objects as well.

Calibration and validation

The model calibration is similar to PCA, but there are several additional arguments, which are important for classification. First of all it is a class name, which is a second mandatory argument. Class name is a string, which can be used later e.g. for identifying class members for testing. The second important argument is a level of significance, `alpha`. This parameter is used for calculation of statistical limits and can be considered as probability for false negatives. The default value is 0.05. Finally the parameter `lim.type` allows to select the method for computing critical limits for the distances, as it is described in the PCA chapter. By default `lim.type = "ddmoments"` as in PCA.

In this chapter as well as for describing other classification methods we will use a famous Iris dataset, available in R. The dataset includes 150 measurements of three Iris species: *Setosa*, *Virginica* and *Versicola*. The measurements are length and width of petals and sepals in cm. Use `?iris` for more details.

Let's get the data and split it to calibration and test sets.

```
data(iris)
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```
# generate indices for calibration set
idx = seq(1, nrow(iris), by = 2)

# split the values
Xc = iris[idx, 1:4]
cc = iris[idx, 5]

Xt = iris[-idx, 1:4]
ct = iris[-idx, 5]
```

Now, because for calibration we need only objects belonging to a class, we will split the `X.c` into three matrices — one for each species. The data is ordered by the species, so it can be done relatively easy by taking every 25 rows.

```
X.set = Xc[1:25, ]
X.ver = Xc[26:50, ]
X.vir = Xc[51:75, ]
```

Let's start with creating a model for class *Versicolor* and exploring available statistics and plots. In this case default values for method and significance level to compute the critical limits (`lim.type = "ddmoments"` and `alpha = 0.05`) are used.

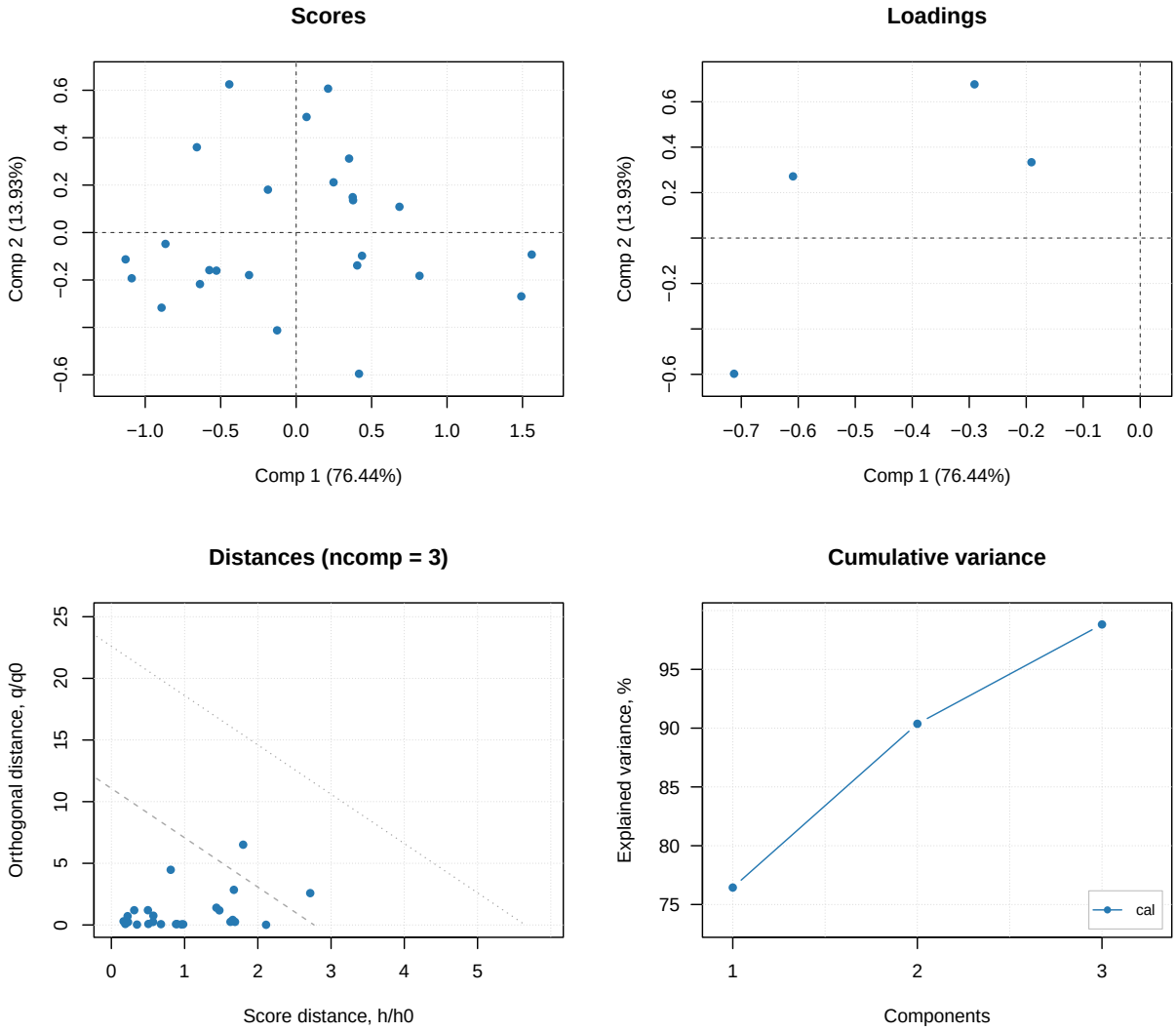
```
m = simca(X.ver, "versicolor", ncomp = 3)
summary(m)
```

```
##
## SIMCA model for class 'versicolor' summary
##
##
## Number of components: 3
## Type of limits: ddmoments
## Alpha: 0.05
## Gamma: 0.01
##
##     Expvar Cumexpvar TP FP TN FN Spec. Sens. Accuracy
## Cal   8.45     98.82 23  0  0  2    NA  0.92     0.92
```

The summary output shows (in addition to explained and cumulative explained variance) number of true positives, false positives, true negatives, false negatives as well as specificity, sensitivity and accuracy of classification. All statistics are shown for each available result object (in this case only calibration) and only for optimal number of components (in this case 3).

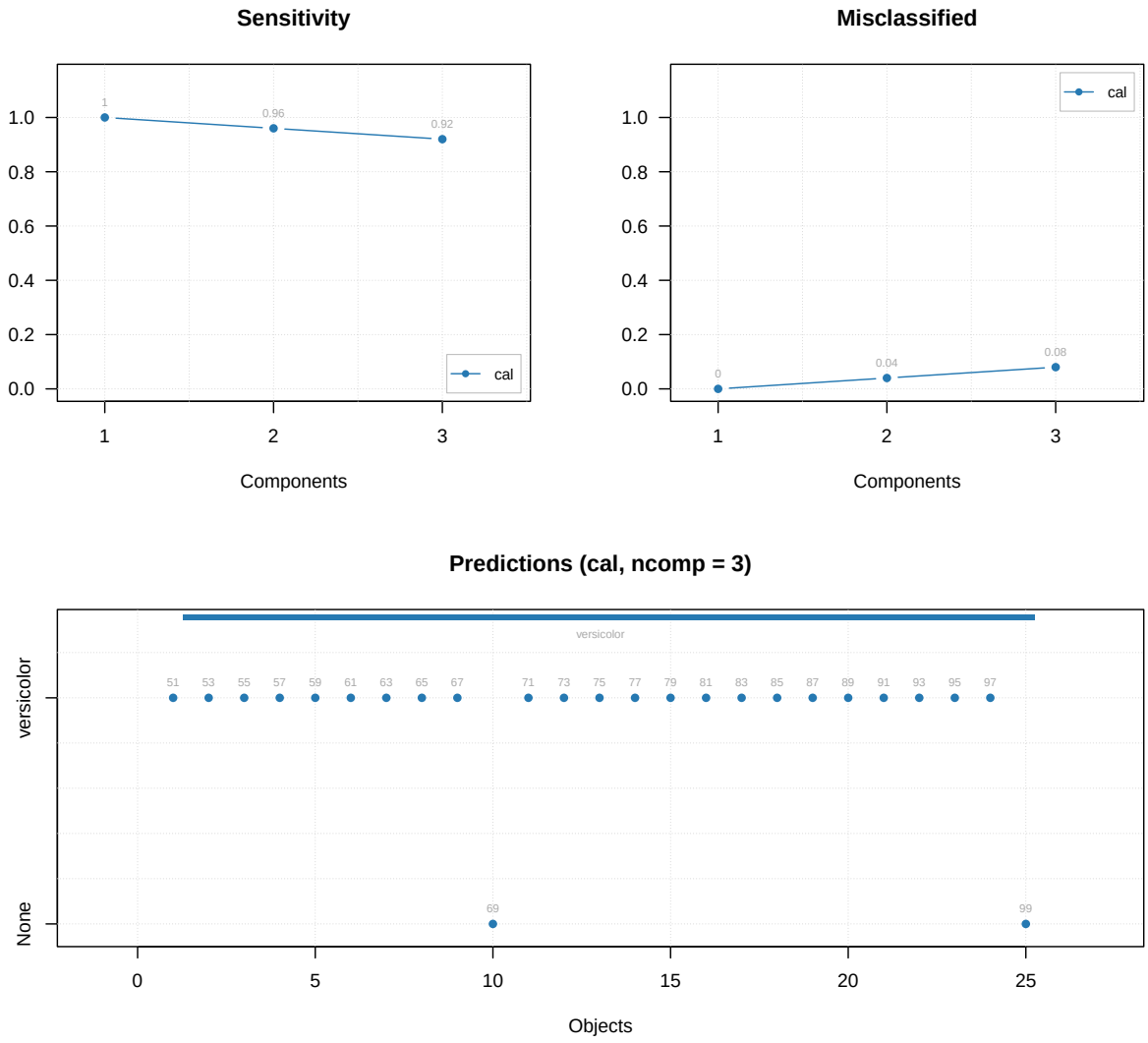
The summary plot look very much similar to what we have seen for PCA.

```
plot(m)
```



In addition to standard PCA plots, SIMCA model (as well as any other classification model, e.g. PLS-DA) can show plot for sensitivity and ratio of misclassified values depending on the number of components. Plus a prediction plot, which shows classification results for each object. See the example below.

```
layout(matrix(c(1, 3, 2, 3), ncol = 2))
plotSensitivity(m, show.labels = TRUE)
plotMisclassified(m, show.labels = TRUE)
plotPredictions(m, show.labels = TRUE)
```



Validation

Because SIMCA is based on PCA, you can use any validation method described in PCA section. Just keep in mind that when cross-validation is used, only performance statistics will be computed (in this case classification performance). Therefore cross-validated result object will not contain scores, distances, explained variance etc. and corresponding plots will not be available.

Here I will show briefly an example based on Procrustes cross-validation. First we load the `pcv` package and create a PV-set for the target class (versicolor):

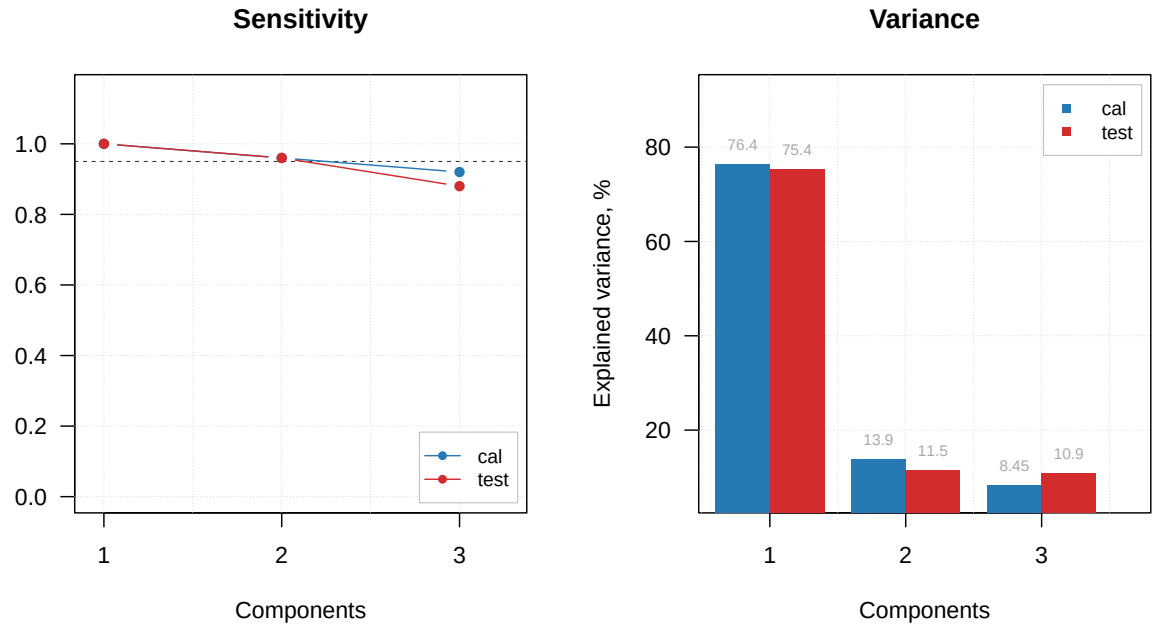
```
library(pcv)
Xpv = pcvpca(as.matrix(X.ver), 4, center = TRUE, scale = TRUE, cv = list("ven", 4))
```

Then we create a SIMCA model with PV-set as test set:

```
m = simca(X.ver, "versicolor", ncomp = 3, x.test = Xpv)
```

Let's look at the sensitivity and explained variance plots:

```
par(mfrow = c(1, 2))
plotSensitivity(m, show.line = c(NA, 0.95))
plotVariance(m, type = "h", show.labels = TRUE)
```



Based on the plot we can select 2 PCs as optimal number. We set this value and show the summary:

```
m = selectCompNum(m, 2)
summary(m)
```

```
##
## SIMCA model for class 'versicolor' summary
##
##
## Number of components: 2
## Type of limits: ddmoments
## Alpha: 0.05
## Gamma: 0.01
##
##      Expvar Cumexpvar TP FP TN FN Spec. Sens. Accuracy
## Cal   13.93     90.37 24  0  0  1    NA  0.96     0.96
## Test  11.49     86.87 24  0  0  1    NA  0.96     0.96
```

Predictions and testing the model

When model is calibrated and optimized, one can test it using a test set with know classes. In this case we will use objects from all three species and be able to see how good the model performs on strangers (and calculate the specificity). In order to do that we will provide both matrix with predictors, `Xt`, and a vector with names of the classes for corresponding objects/rows (`ct`). The values with known classes in this case can be:

- a vector with text values (names)
- a factor using the names as labels (also as a vector)
- a vector with logical values (`TRUE` for class members and `FALSE` for strangers)

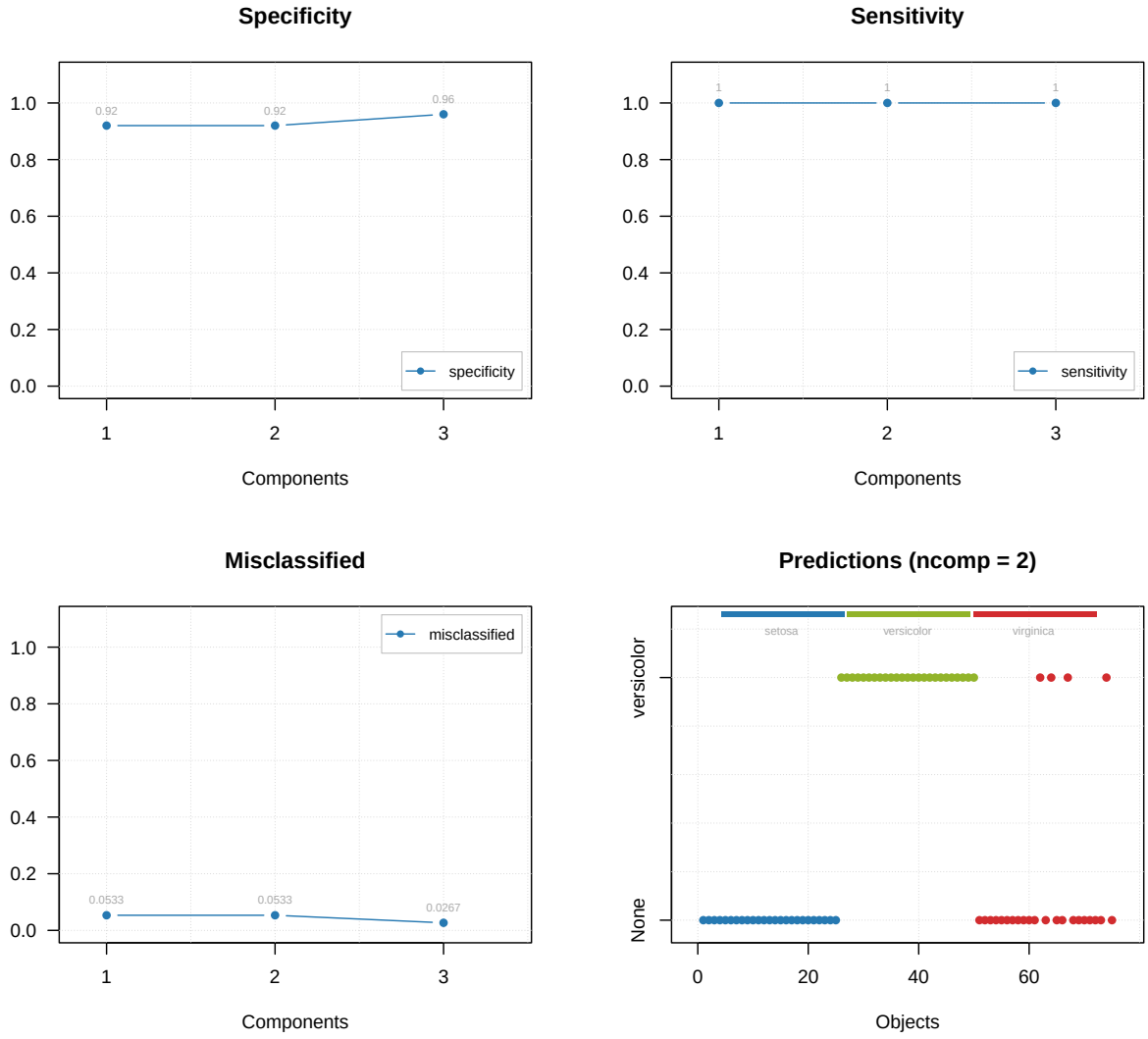
In our case we have a vector with text values, which will be automatically converted to a factor by the function `predict()`. Instead of creating a new model and providing the values as test set we will simply make predictions.

```
res = predict(m, Xt, ct)
summary(res)
```

```
##
## Summary for SIMCA one-class classification result
##
## Class name: versicolor
## Number of selected components: 2
##
##        Expvar Cumexpvar TP FP TN FN Spec. Sens. Accuracy
## Comp 1  64.27     64.27 25  4 46  0  0.92     1    0.947
## Comp 2   1.67     65.95 25  4 46  0  0.92     1    0.947
## Comp 3  32.45     98.40 25  2 48  0  0.96     1    0.973
```

In this case we can also see the specificity values and corresponding plot can be made, as shown below together with other plots.

```
par(mfrow = c(2, 2))
plotSpecificity(res, show.labels = TRUE)
plotSensitivity(res, show.labels = TRUE)
plotMisclassified(res, show.labels = TRUE)
plotPredictions(res)
```



As you can see, the prediction plot looks a bit different in this case. Because the test set has objects from several classes and the class belongings is known, this information is shown as color bar legend. For instance, in the example above we can see, that two *Virginica* objects were erroneously classified as members of *Versicolor*.

You can also show the predictions as a matrix with -1 and $+1$ using method `showPredictions()` or get the array with predicted class values directly as it is shown in the example below (for 10 rows in the middle of the data, different number of components and the first classification variable).

```
show(res$c.pred[45:55, 1:3, 1])
```

```
##      Comp 1 Comp 2 Comp 3
## 90        1      1      1
## 92        1      1      1
## 94        1      1      1
## 96        1      1      1
## 98        1      1      1
## 100       1      1      1
## 102      -1     -1     -1
## 104      -1     -1     -1
```

```
## 106     -1     -1     -1
## 108     -1     -1     -1
## 110     -1     -1     -1
```

You can also get and show the confusion matrix (rows correspond to real classes and columns to the predicted class) for an object with SIMCA results (as well as results obtained with any other classification method, e.g. PLS-DA).

```
show(getConfusionMatrix(res))
```

```
##            versicolor None
## versicolor         25    0
## setosa              0   25
## virginica           4   21
```

Class belonging probabilities

In addition to the array with predicted class, the object with SIMCA results also contains an array with class belongings probabilities. The probabilities are calculated depending on how close a particular object is to the the critical limit border.

To compute the probability we use the theoretical distribution for score and orthogonal distances as when computing critical values (defined by the parameter `lim.type`). The distribution is used to calculate a p-value — chance to get object with given distance value or larger. The p-value is then compared with significance level, α, and the probability, π is calculated as follows:

$$\pi = 0.5(p/\alpha)$$

So if p-value is the same as significance level (which happens when object is lying exactly on the acceptance line) the probability is 0.5. If p-value is e.g. 0.04, $\pi = 0.4$, or 40%, and the object will be rejected as a stranger (here we assume that the $\alpha = 0.05$). If the p-value is e.g. 0.06, $\pi = 0.6$, or 60%, and the object will be accepted as a member of the class. If p-value is larger than $2 \times \alpha$ the probability is set to 1.

In case of rectangular acceptance area (`lim.type = "jm"` or `"chisq"`) the probability is computed separately for q and h values and the smallest of the two is taken. In case of triangular acceptance area (`lim.type = "ddmoments"` or `"ddrobust"`) the probability is calculated for a full distance, f.

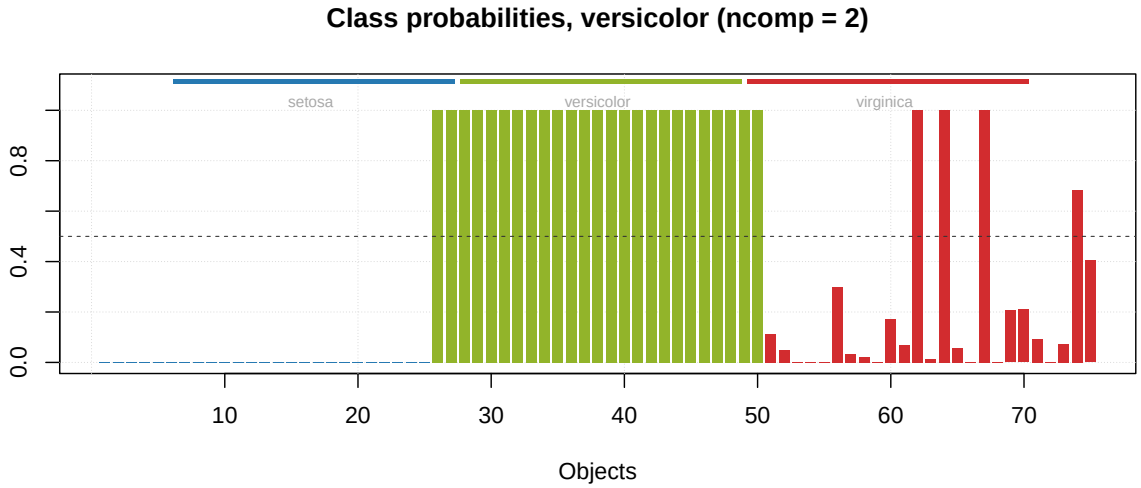
Here is how to show the probability values, that correspond to the predictions shown in the previous code chunk. I round the probability values to four decimals for better output.

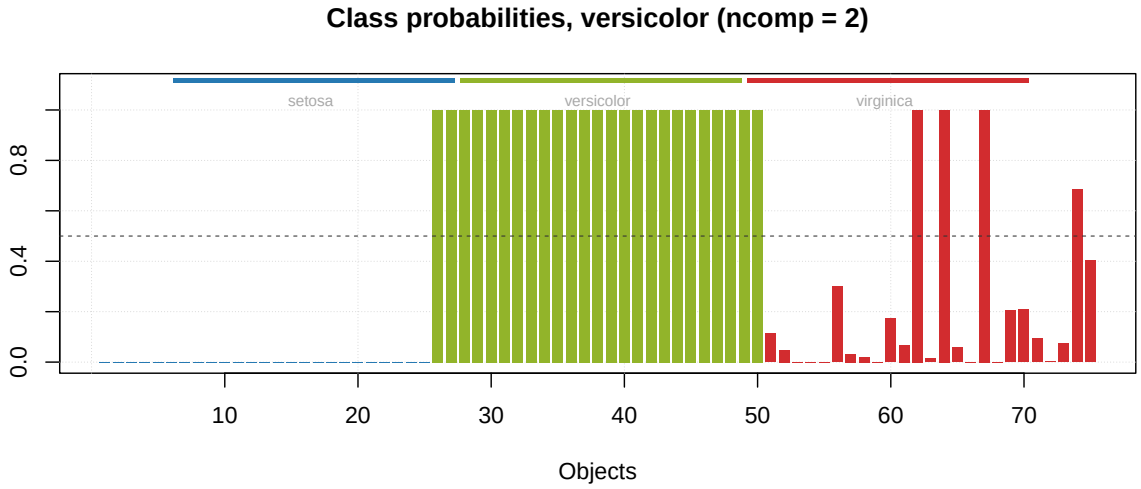
```
show(round(res$p.pred[45:55, 1:3, 1], 4))
```

```
##     Comp 1 Comp 2 Comp 3
## 90  1.0000 1.0000 1.0000
## 92  1.0000 1.0000 1.0000
## 94  1.0000 1.0000 1.0000
## 96  1.0000 1.0000 1.0000
## 98  1.0000 1.0000 1.0000
## 100 1.0000 1.0000 1.0000
## 102 0.0376 0.1121 0.0028
## 104 0.0285 0.0477 0.0216
## 106 0.0000 0.0000 0.0000
## 108 0.0006 0.0002 0.0000
## 110 0.0001 0.0000 0.0000
```

It is also possible to show the probability values as a plot with method `plotProbabilities()`:

```
par(mfrow = c(2, 1))
plotProbabilities(res, cgroup = ct)
plotProbabilities(res, ncomp = 2, cgroup = ct)
```





The plot can be shown for any SIMCA results (including e.g. calibration set or cross-validated results).

Multiclass classification

Several SIMCA models can be combined to a special object `simcam`, which is used to make a multiclass classification. Besides this, it also allows calculating distance between individual models and a *discrimination power* — importance of variables to discriminate between any two classes. Let's see how it works.

First we create three single-class SIMCA models with individual settings, such as number of optimal components and alpha.

```
m.set = simca(X.set, "setosa", 3, alpha = 0.01)
m.set = selectCompNum(m.set, 1)

m.vir = simca(X.vir, "virginica", 3)
```

```
m.vir = selectCompNum(m.vir, 2)

m.ver = simca(X.ver, "versicolor", 3)
m.ver = selectCompNum(m.ver, 1)
```

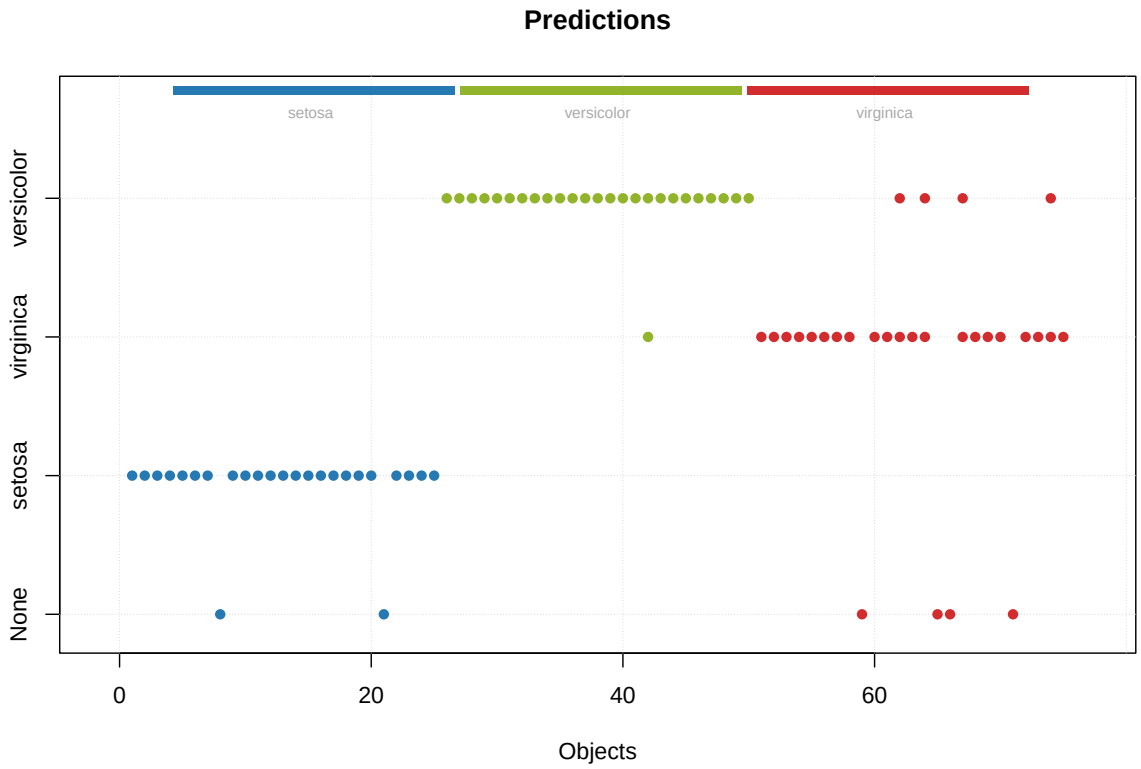
Then we combine the models into a `simcam` model object. Summary will show the performance on calibration set, which is a combination of calibration sets for each of the individual models

```
mm = simcam(list(m.set, m.vir, m.ver))
summary(mm)
```

```
##
## SIMCA multiple classes classification (class simcam)
##
## Number of classes: 3
## Info:
##
## Summary for calibration results
##            Ncomp TP FP TN FN Spec. Sens. Accuracy
## setosa         1 25  0 50  0  1.00  1.00     1.00
## virginica      2 22  3 47  3  0.94  0.88     0.92
## versicolor     1 25  3 47  0  0.94  1.00     0.96
```

Now we apply the combined model to the test set and look at the predictions.

```
res = predict(mm, Xt, ct)
plotPredictions(res)
```



In this case, the predictions are shown only for the number of components each model found optimal. The

names of classes along y-axis are the individual models. Similarly we can show the predicted values.

```
show(res$c.pred[20:30, 1, ])
```

```
##    setosa virginica versicolor
## 40      1        -1         -1
## 42     -1        -1         -1
## 44      1        -1         -1
## 46      1        -1         -1
## 48      1        -1         -1
## 50      1        -1         -1
## 52     -1        -1          1
## 54     -1        -1          1
## 56     -1        -1          1
## 58     -1        -1          1
## 60     -1        -1          1
```

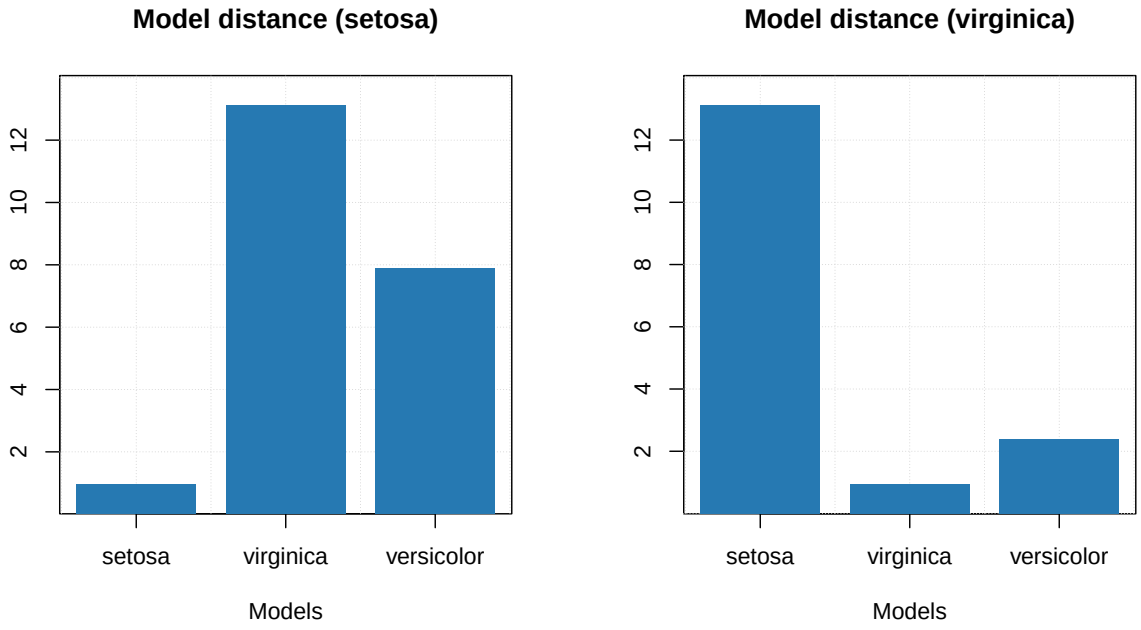
Method `getConfusionMatrix()` is also available in this case.

```
show(getConfusionMatrix(res))
```

```
##            setosa virginica versicolor None
## setosa         23         0          0    2
## virginica       0        21          4    4
## versicolor      0         1         25    0
```

There are three additional plots available for multiclass SIMCA model. First of all it is a distance between a selected model and the others.

```
par(mfrow = c(1, 2))
plotModelDistance(mm, 1)
plotModelDistance(mm, 2)
```

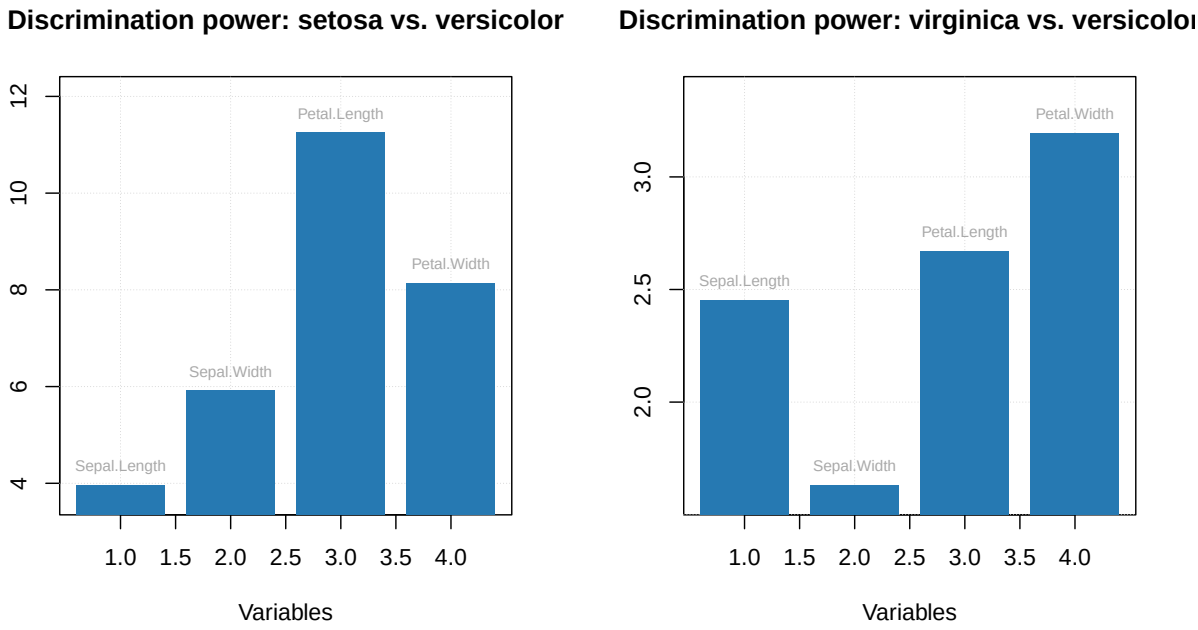


The plot shows not a real distance but rather a similarity between a selected model and the others as a ratio of residual variances. You can find more detailed description about how model is calculated in description of

the method or in help for `plotModelDistance.simcam` function.

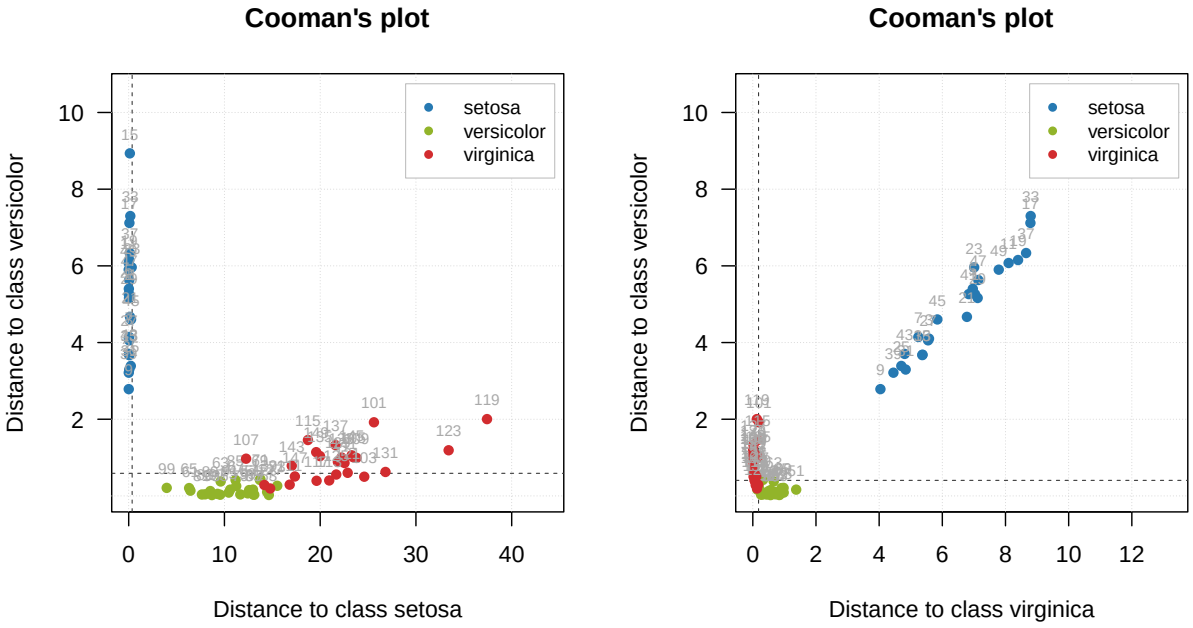
The second plot is a discrimination power, mentioned in the beginning of the section.

```
par(mfrow = c(1, 2))
plotDiscriminationPower(mm, c(1, 3), show.labels = TRUE)
plotDiscriminationPower(mm, c(2, 3), show.labels = TRUE)
```



And, finally, a Cooman's plot showing an orthogonal distance, q, from objects to two selected classes/models.

```
par(mfrow = c(1, 2))
plotCooman(mm, c(1, 3), show.labels = TRUE)
plotCooman(mm, c(2, 3), show.labels = TRUE)
```



The limits, shown as dashed lines, are computed using chi-square distribution but only for q values.

Interval PLS

Interval PLS (iPLS) is a variable selection method used mostly for spectroscopic data proposed by Lars Noergaard *at al* in 2000. The main idea is try different parts (or intervals — hence the name) of spectra and their combinations to find the most relevant for prediction of a response variable. There are two algorithms — forward and backward.

Forward iPLS

The forward approach works as follows

1. Split spectral data into N intervals
2. Create an empty vector with selected intervals
3. Create a model where intervals in the vector (already slected) are combined with one of the rest. If combination improves the model, add this new interval to the vector.
4. Repeat previous step until there is no improvements.

Apparently, when nothing is selected, on step 3 the algorithm simply create model for every interval available. As you can see, iPLS is based on iterative procedure. If you have 20 intervals you need to create 20 models on the first step, 19 models on the second step and so on.

In *mdatools*, you can run iPLS selection by using function `ipls()`. You have to specify matrix with predictors (`x`) and responses (`y`), number of components in global model (`glob.ncomp`), and define the intervals. In addition to that, you can also specify parameters of cross-validation (`cv`) as well as criterion for selection of optimal number of components (`ncomp.selcrit`) for local models similar to PLS. Forward iPLS is used as a default algorithm.

Intervals can be defined using one of the following three parameters: number of intervals (`int.num`), width of an interval (`int.width`) or specify start and end of each interval as a two-column matrix with variable indices (`int.limits`). In addition to that, you can specify maximum number of component for local models (`int.ncomp`) and maximum number of iterations — so method will stop regardless if you have improvements or not (`int.niter`). By default the maximum number of iterations is limited to 30.

Here is an example of applying iPLS for Simdata (for concentration of second chemical component, C2) using 15 intervals.

```
data(simdata)
X = simdata$spectra.c
y = simdata$conc.c[, 2, drop = FALSE]

m = ipls(X, y, glob.ncomp = 4, int.num = 15)
```

```
##
## Model with all intervals: RMSE = 0.027572, nLV = 3
## Iteration   1/ 15... selected interval   6 (RMSE = 0.030405, nLV = 3)
## Iteration   2/ 15... selected interval  11 (RMSE = 0.027882, nLV = 3)
## Iteration   3/ 15... selected interval   1 (RMSE = 0.027509, nLV = 3)
```

```
## Iteration   4/ 15... selected interval   2 (RMSE = 0.027243, nLV = 2)
## Iteration   5/ 15... selected interval  13 (RMSE = 0.027210, nLV = 2)
## Iteration   6/ 15... selected interval  12 (RMSE = 0.027201, nLV = 2)
## Iteration   7/ 15... no improvements, stop.
```

As you can see, by default method shows information for every step in the console. Use parameter `silent = TRUE` to change this behaviour.

From the example above we can see that the global model had RMSECV = 0.027625 with 3 components (by default the method uses systematic corss-validation, "Venetian blinds", with 10 segments). Creating local models with individual intervals gave the best performance with RMSECV = 0.029830 (interval #6). Combination of the interval #6 with interval #1 (next step) gave RMSECV = 0.0027809. The small improvement was obtained for adding 3 more intervals giving final result with RMSECV = 0.027246 with 5 selected intervals in total (6, 1, 11, 3, 12).

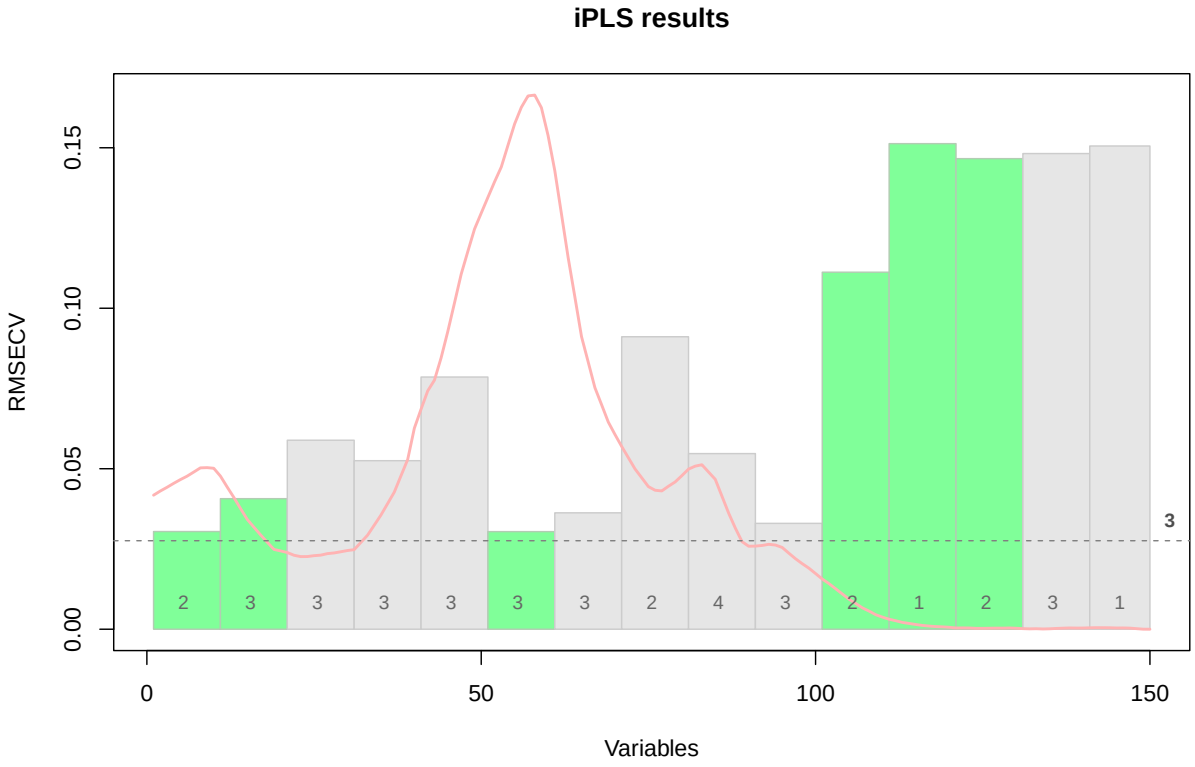
There are several ways to explore the iPLS results both graphically and numerically. First of all, summary will show full information about the selected intervals, including index of variables on both ends:

```
summary(m)
```

```
##
## iPLS variable selection results
##   Method: forward
##   Validation: venetian blinds with 10 segments
##   Number of intervals: 15
##   Number of selected intervals: 6
##   RMSECV for global model: 0.027572 (3 LVs)
##   RMSECV for optimized model: 0.027201 (2 LVs)
##
## Summary for selection procedure:
##    n start end selected nComp       RMSE    R2
## 1  0     1 150    FALSE     3 0.02757234 0.965
## 2  6    51  60     TRUE     3 0.03040477 0.958
## 3 11   101 110     TRUE     3 0.02788242 0.965
## 4  1     1  10     TRUE     3 0.02750915 0.966
## 5  2    11  20     TRUE     2 0.02724335 0.966
## 6 13   121 130     TRUE     2 0.02720961 0.966
## 7 12   111 120     TRUE     2 0.02720089 0.966
```

Also you can see the first step (performance of individual models) and the selected interval by using function `plot()` for the whole model:

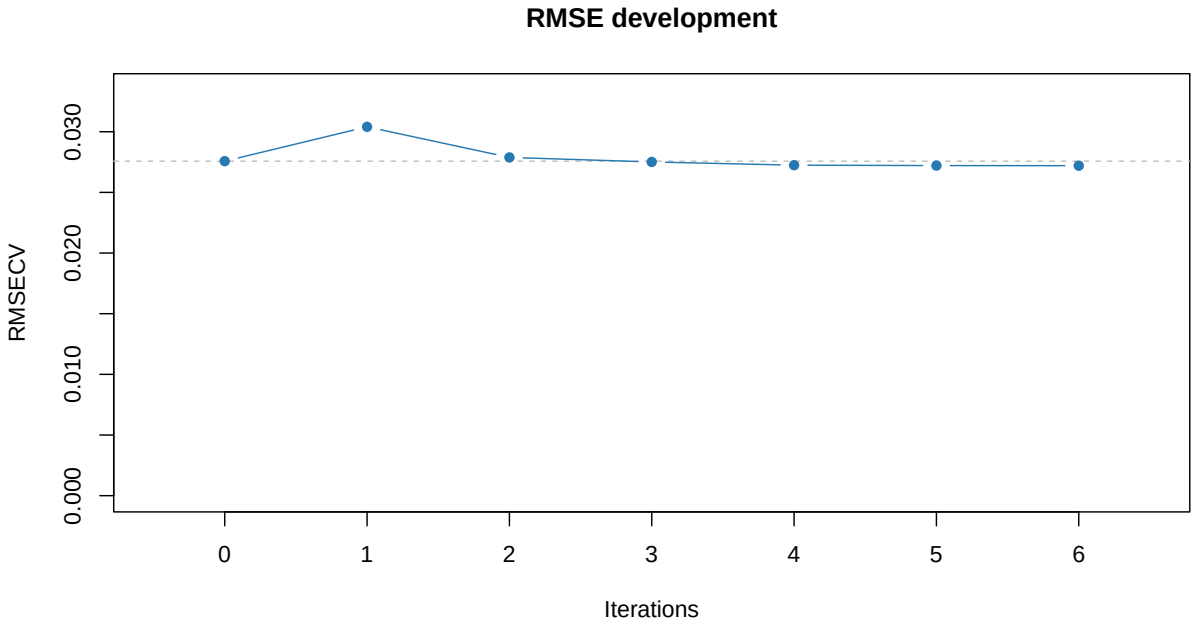
```
plot(m)
```



The red curve here is the average spectrum. Bars are intervals, height of each bar corresponds to the RMSECV value obtained for local model made using variables from this interval as predictors (first iteration). Number inside each bar is number of PLS components used in the local model. Green color shows intervals which have been selected at the end of the procedure and dashed line shows error for the global model

You can also see the improvements of RMSECV with iPLS iterations graphically:

```
plotRMSE(m)
```



Dashed line in this case shows RMSECV value for the global model with all intervals included.

Finally, you can get both selected intervals and corresponding indices of variables (all of them not just interval limits) as follows:

```
show(m$int.selected)
```

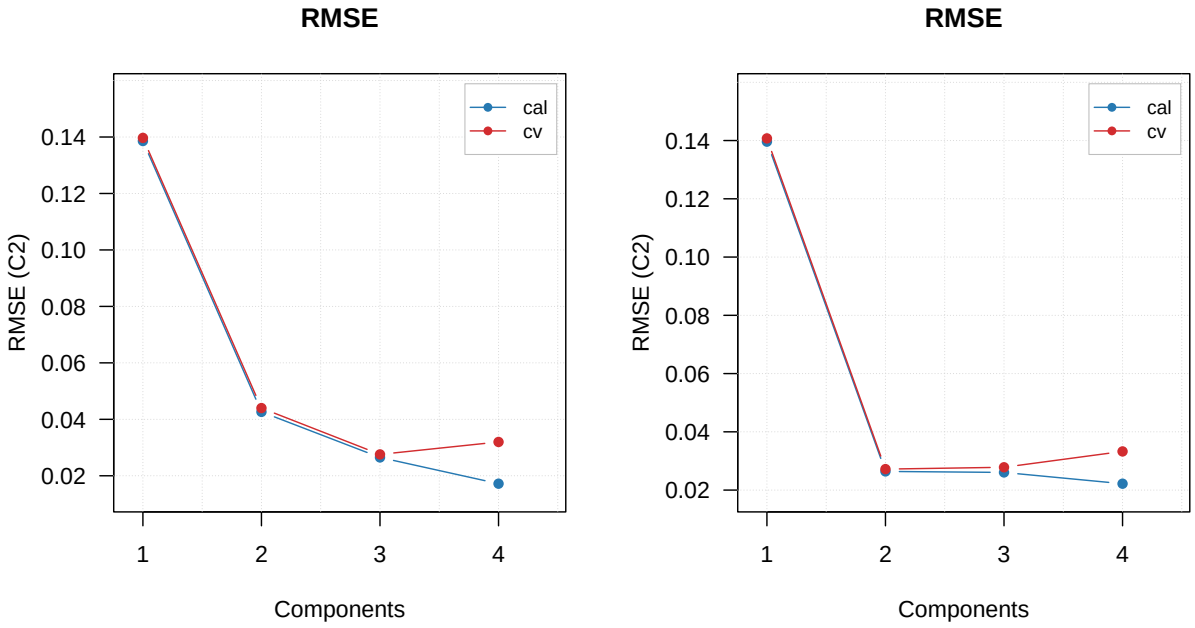
```
## [1]  6 11  1  2 13 12
```

```
show(m$var.selected)
```

```
##  [1]  51  52  53  54  55  56  57  58  59  60 101 102 103 104 105 106 107 108 109 110   1   2   3   4
## [33]  13  14  15  16  17  18  19  20 121 122 123 124 125 126 127 128 129 130 111 112 113 114 115 116
```

Also the model object contains the initial global model for full data (`m$gm`) and final optimized PLS model made using only variables from the selected intervals, `m$om`. In the following example I compare RMSE plots for both:

```
par(mfrow = c(1, 2))
plotRMSE(m$gm)
plotRMSE(m$om)
```



Running full procedure

By default the iPLS procedure will stop when no improvement is observed. This means if RMSECV value for the next iteration is not smaller than the RMSECV value for the current iteration. However, you can change this behavior if you set a logical parameter `full = TRUE`. In this case the procedure will continue until the maximum number of iterations is reached.

Here are some examples. First of all let's run the same example as in the prevous section with 15 intervals:

```
data(simdata)
X = simdata$spectra.c
y = simdata$conc.c[, 2, drop = FALSE]

m = ipls(X, y, glob.ncomp = 4, int.num = 15)
```

```
##
## Model with all intervals: RMSE = 0.027572, nLV = 3
## Iteration   1/ 15... selected interval   6 (RMSE = 0.030405, nLV = 3)
## Iteration   2/ 15... selected interval  11 (RMSE = 0.027882, nLV = 3)
## Iteration   3/ 15... selected interval   1 (RMSE = 0.027509, nLV = 3)
## Iteration   4/ 15... selected interval   2 (RMSE = 0.027243, nLV = 2)
## Iteration   5/ 15... selected interval  13 (RMSE = 0.027210, nLV = 2)
## Iteration   6/ 15... selected interval  12 (RMSE = 0.027201, nLV = 2)
## Iteration   7/ 15... no improvements, stop.
```

As you can see, the procedure stops at iteration #6 because no improvement is observed. And if you look at statistics, indeed only 5 intervals are selected:

```
summary(m)
```

```
##
## iPLS variable selection results
##   Method: forward
##   Validation: venetian blinds with 10 segments
```

```
##   Number of intervals: 15
##   Number of selected intervals: 6
##   RMSECV for global model: 0.027572 (3 LVs)
##   RMSECV for optimized model: 0.027201 (2 LVs)
##
## Summary for selection procedure:
##    n start end selected nComp       RMSE    R2
## 1  0     1 150    FALSE     3 0.02757234 0.965
## 2  6    51  60     TRUE     3 0.03040477 0.958
## 3 11   101 110     TRUE     3 0.02788242 0.965
## 4  1     1  10     TRUE     3 0.02750915 0.966
## 5  2    11  20     TRUE     2 0.02724335 0.966
## 6 13   121 130     TRUE     2 0.02720961 0.966
## 7 12   111 120     TRUE     2 0.02720089 0.966
```

```
show(m$int.selected)
```

```
## [1]  6 11  1  2 13 12
```

Now let's do the same but with parameter `full = TRUE`.

```
m = ipls(X, y, glob.ncomp = 4, int.num = 15, full = TRUE)
```

```
##
## Model with all intervals: RMSE = 0.027572, nLV = 3
## Iteration   1/ 15... selected interval   6 (RMSE = 0.030405, nLV = 3)
## Iteration   2/ 15... selected interval  11 (RMSE = 0.027882, nLV = 3)
## Iteration   3/ 15... selected interval   1 (RMSE = 0.027509, nLV = 3)
## Iteration   4/ 15... selected interval   2 (RMSE = 0.027243, nLV = 2)
## Iteration   5/ 15... selected interval  13 (RMSE = 0.027210, nLV = 2)
## Iteration   6/ 15... selected interval  12 (RMSE = 0.027201, nLV = 2)
## Iteration   7/ 15... selected interval   8 (RMSE = 0.027219, nLV = 2)
## Iteration   8/ 15... selected interval  14 (RMSE = 0.027245, nLV = 2)
## Iteration   9/ 15... selected interval  15 (RMSE = 0.027331, nLV = 2)
## Iteration  10/ 15... selected interval   4 (RMSE = 0.027443, nLV = 3)
## Iteration  11/ 15... selected interval   3 (RMSE = 0.027391, nLV = 3)
## Iteration  12/ 15... selected interval   9 (RMSE = 0.027353, nLV = 3)
## Iteration  13/ 15... selected interval   5 (RMSE = 0.027378, nLV = 3)
## Iteration  14/ 15... selected interval   7 (RMSE = 0.027487, nLV = 3)
## Iteration  15/ 15... selected interval  10 (RMSE = 0.027572, nLV = 3)
```

Now it runs 15 iterations, because by default this is the largest number of iterations in this case (all possible intervals).

In case of full procedure, the selection of intervals and corresponding variables is done by finding a global minimum of RMSECV. As you can see from the output above, the first local minimum (RMSECV = 0.027246) was indeed at the 5th iteration, after which the procedure stopped in our previous example. However, when we use `full = TRUE` it continues and the global minimum is observed at iteration #8 (RMSECV = 0.027216).

Therefore the number of selected intervals will be 8:

```
summary(m)
```

```
##
## iPLS variable selection results
##   Method: forward
##   Validation: venetian blinds with 10 segments
##   Number of intervals: 15
```

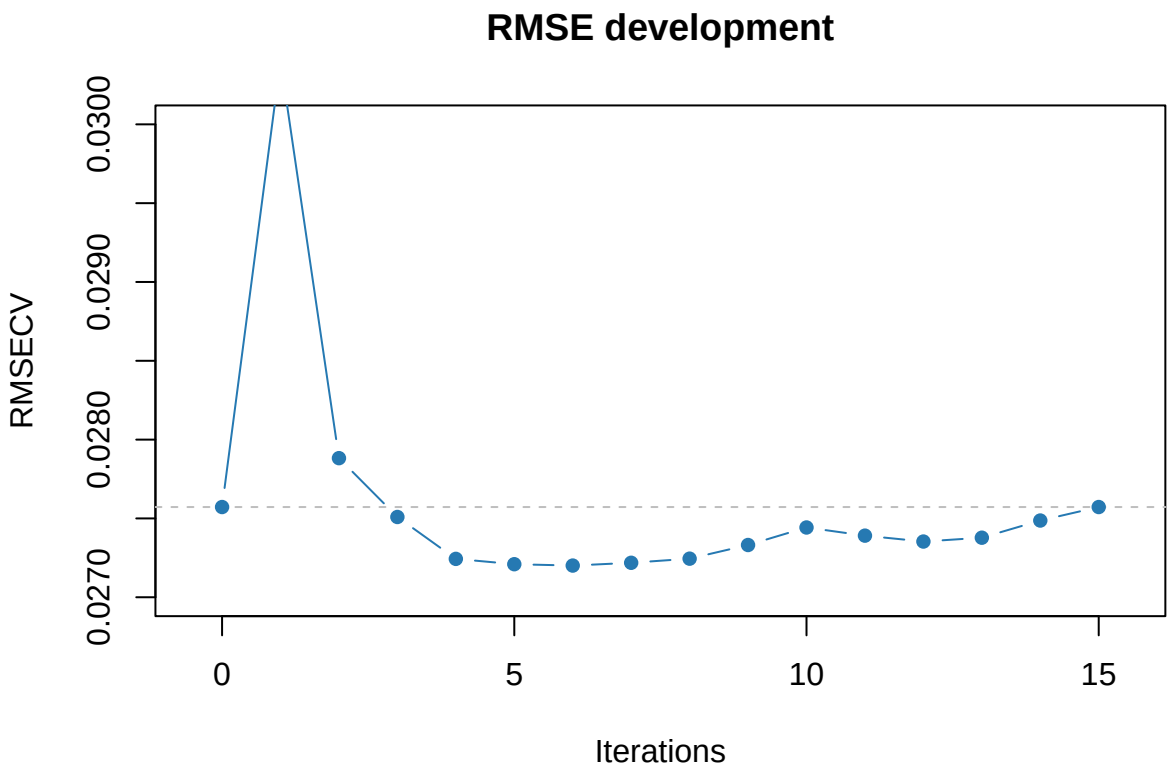
```
##   Number of selected intervals: 6
##   RMSECV for global model: 0.027572 (3 LVs)
##   RMSECV for optimized model: 0.027201 (2 LVs)
##
## Summary for selection procedure:
##     n start end selected nComp       RMSE    R2
## 1   0     1 150    FALSE     3 0.02757234 0.965
## 2   6    51  60     TRUE     3 0.03040477 0.958
## 3  11   101 110     TRUE     3 0.02788242 0.965
## 4   1     1  10     TRUE     3 0.02750915 0.966
## 5   2    11  20     TRUE     2 0.02724335 0.966
## 6  13   121 130     TRUE     2 0.02720961 0.966
## 7  12   111 120     TRUE     2 0.02720089 0.966
## 8   8    71  80    FALSE     2 0.02721858 0.966
## 9  14   131 140    FALSE     2 0.02724466 0.966
## 10 15   141 150    FALSE     2 0.02733144 0.966
## 11  4    31  40    FALSE     3 0.02744254 0.966
## 12  3    21  30    FALSE     3 0.02739080 0.966
## 13  9    81  90    FALSE     3 0.02735341 0.966
## 14  5    41  50    FALSE     3 0.02737763 0.966
## 15  7    61  70    FALSE     3 0.02748661 0.966
## 16 10    91 100    FALSE     3 0.02757234 0.965
```

```
show(m$int.selected)
```

```
## [1]  6 11  1  2 13 12
```

You can also see this on RMSECV vs iterations plot:

```
plotRMSE(m, ylim = c(0.027, 0.030))
```



The first minimum is clearly observed at 5th iteration and the global minimum at 8th. Of course in this particular case this effect is most probably random, however there are cases where running full iPLS procedure is useful.

If the number of intervals is large, by default, the maximum number of iterations `ipls()` will try, is limited by 30. You can change this by specifying an additional parameter, `iter.niter`, for example `iter.niter = 100`.

Using test set for validation

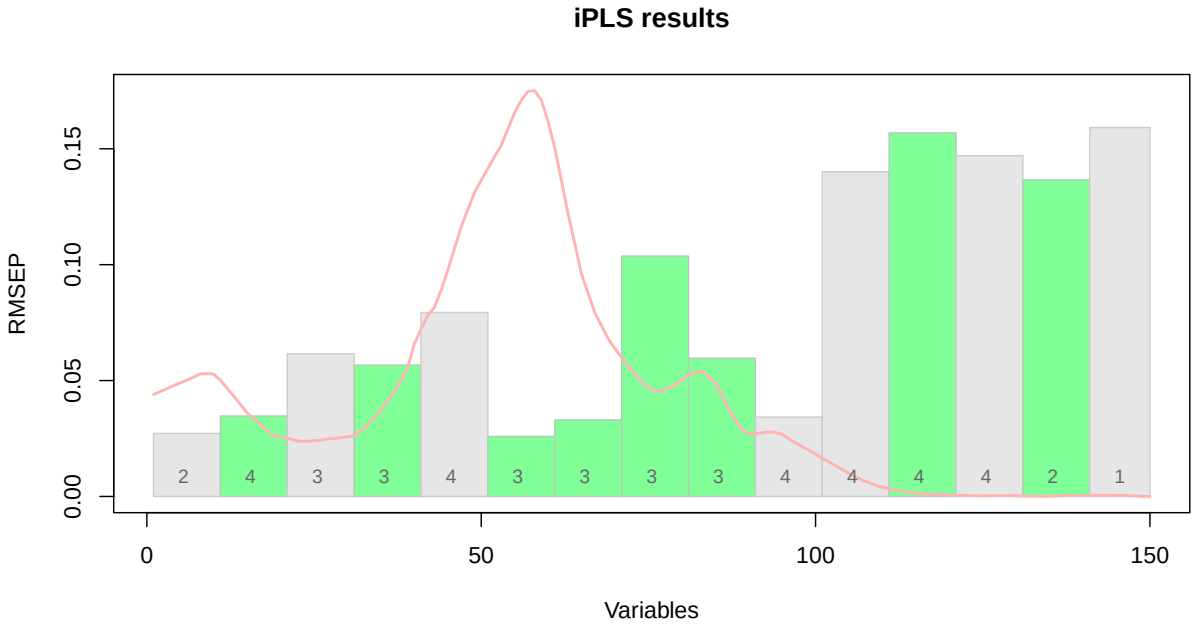
Although iPLS was developed for using with cross-validation, sometimes, especially if dataset is large, it can give very large computational time. In this case you can provide test set for validation instead. The syntax and the parameter names are similar to test validation in `pls()`. Here is an example:

```
data(simdata)
X = simdata$spectra.c
y = simdata$conc.c[, 2, drop = FALSE]

X.t = simdata$spectra.t
y.t = simdata$conc.t[, 2, drop = FALSE]
m = ipls(X, y, glob.ncomp = 4, int.num = 15, x.test = X.t, y.test = y.t)
```

```
##
## Model with all intervals: RMSE = 0.024003, nLV = 3
## Iteration   1/ 15... selected interval   6 (RMSE = 0.025899, nLV = 3)
## Iteration   2/ 15... selected interval   7 (RMSE = 0.023865, nLV = 3)
## Iteration   3/ 15... selected interval  12 (RMSE = 0.023390, nLV = 3)
## Iteration   4/ 15... selected interval   2 (RMSE = 0.022984, nLV = 3)
```

```
## Iteration   5/ 15... selected interval   4 (RMSE = 0.022741, nLV = 3)
## Iteration   6/ 15... selected interval   9 (RMSE = 0.022551, nLV = 3)
## Iteration   7/ 15... selected interval   8 (RMSE = 0.022474, nLV = 3)
## Iteration   8/ 15... selected interval  14 (RMSE = 0.022428, nLV = 3)
## Iteration   9/ 15... no improvements, stop.
```

```
plot(m)
```



Backward iPLS

In backward iPLS, instead of selecting best intervals we do the opposite — get rid of the worst. So, at the first step, we try to remove every interval from the data to see if it gives any improvement. If it does, we keep it as exluded and then try to remove another one. The process continues until no improvement is observed.

To use the backward method simply specify parameter `method = "backward"` when call `ipls()`. The rest, including plots and statistics, is the same.

Multivariate Curve Resolution

Multivariate Curve Resolution (MCR) is a group of methods which can be used to solve the curve resolution problem in spectroscopy, which, in its general form, can be defined as follows. Let's say we have a mixture of A chemical components (e.g. ribose, fructose and lactose). Every individual component is usually called a *pure* component. Every pure component i has a spectrum (IR, NIR, Raman, etc.), which can be represented as a column vector $\mathbf{s}_i$, with size $J \times 1$, where J is a number of values in each spectrum (corresponding to the number of wavelength, wavenumbers, chemical shifts, etc.).

According to the Beer-Lambert law, if you mix the components into one mixture and take a spectrum of this mixture, the spectrum will be just a linear combination of the spectra of the pure components. This can be written as follows:

$$\mathbf{d} = c_1\mathbf{s}_1^T + c_2\mathbf{s}_2^T + \cdots + c_A\mathbf{s}_A^T$$

In this equation, $\mathbf{d}$ is a vector of spectral values representing the spectrum of the mixture $(1 \times J)$, $c_1, c_2, ..., c_A$ are concentrations of the pure components in the mixture and $\mathbf{s}_1, \mathbf{s}_2, ..., \mathbf{s}_A$ are the spectra of the pure components. If we combine the concentration values into a $1 \times A$ row-vector $\mathbf{c} = [c_1, c_2, ..., c_A]$ then the equation can be written in a more compact form:

$$\mathbf{d} = \mathbf{c}\mathbf{S}^T$$

Where $\mathbf{S}$ is a $J \times A$ matrix containing spectra of all pure components as columns.

Apparently, if we have more than one mixture and concentrations of the pure components vary, we can combine all concentration values into a matrix $\mathbf{C}$, where every row will correspond to a particular mixture. In this case we can write the equation as follows:

$$\mathbf{D} = \mathbf{C}\mathbf{S}^T$$

The task of the MCR methods is to get $\mathbf{C}$ and $\mathbf{S}$ by knowing $\mathbf{D}$, so we sort of resolve the mixtures into individual components and their concentrations. This is not a trivial task as the expression above does not have a unique solution. For example, one of the solutions is what PCA gives, but neither scores correspond to the real concentration values nor loadings represent the spectra of the pure components.

In fact, it is impossible to get $\mathbf{C}$ and $\mathbf{S}$ precisely, what we get is a sort of estimate, which can be denoted as $\hat{\mathbf{C}}$ and $\hat{\mathbf{S}}$. In this case we can rewrite the equation as:

$$\mathbf{D} = \hat{\mathbf{C}}\hat{\mathbf{S}}^T + \mathbf{E}$$

Where $\mathbf{E}$ is a matrix with residuals.

So there are many different methods and tricks which help to get a decent solution in this case. In *mdatools*, starting from v. 0.11.0, there are two MCR methods available — based on the purity approach (`mcrpure()`),

also known as SIMPLISMA, and, based on the constrained alternating least squares (`mcrals()`). This chapter explains how to use both for practical tasks.

More information about the MCR methods in general can be found in this book.

Starting from v. 0.11.0 the *mdatools* package contains additional dataset, `carbs`, which has three objects: `carbs$S` is a matrix (1401 × 3) of Raman spectra of three carbohydrates: fructose, lactose, and ribose; `carbs$D` contains 21 simulated spectra of their mixtures and `carbs$C` contains concentrations used to create the mixtures. The mixtures spectra also contain some random noise, which is uniformly distributed between 0 and 3% of maximum intensity. The spectra of the pure components were taken from publicly available SPECARB library created by S.B. Engelsen. This dataset will be mainly used in this chapter to show how the implemented MCR methods work.

Purity based

The purity based approach implemented in *mdatools* was proposed by Willem Windig and co-authors in 2005 as an alternative to classical SIMPLISMA method. The general idea of the approach is to find variables (wavelength, wavenumbers, etc.) in $\mathbf{D}$, which are influenced mostly by one chemical component. Such variables are called as *pure variables*. If we identify pure variable for each of the components, then we can solve the MCR problem by using ordinary least squares method:

$$\hat{\mathbf{S}} = \mathbf{D}^T\mathbf{D}_R(\mathbf{D}_R^T\mathbf{D})^{-1}$$

$$\hat{\mathbf{C}} = \mathbf{D}\hat{\mathbf{S}}(\hat{\mathbf{S}}^T\hat{\mathbf{S}})^{-1}$$

Here $\hat{\mathbf{S}}$ and $\hat{\mathbf{C}}$ are the spectra and concentrations (contributions) of the pure components estimated by this method. The matrix $\mathbf{D}_R$ is a reduced version of $\mathbf{D}$ where only pure variables are kept (one for each pure component, so this matrix has a dimension $A \times A$). So the question is how to find the pure variables?

Windig and co-authors proposed to do it by computing angles between the spectral variables. For the first component angle between all variables and a vector of ones is computed and first pure variable is selected as the one having the largest angle. Then, angles between the first selected pure variable and the rest are computed and, again, variable with the largest angle is selected as the purest. This continues until pure variables for all components are identified.

To reduce the influence of noisy variables, correction factor, is computed for each variable as follows:

$$\mathbf{n} = \frac{\mathbf{m}}{\mathbf{m} + offset \times \max(\mathbf{m})}$$

Here $\mathbf{m}$ is a mean spectrum computed for the original data, $\mathbf{D}$, and the $offset$ is a tuning parameter defined by a user. Usually a value between 0.001 and 0.05 is a good choice for the offset. More details about the method can be found in the above mentioned paper.

An example

Function `mcrpure()` implements the purity based method in *mdatools*. It has two mandatory arguments — matrix with the original spectra and number of pure components — as well as several extra parameters. The most important one is the offset, which by default is 0.05.

The code below shows how to resolve the spectra from the `carbs` dataset and show summary information:

```
data(carbs)
m = mcrpure(carbs$D, ncomp = 3)
summary(m)
```

```
##
## Summary for MCR Purity case (class mcrpure)
##        Expvar Cumexpvar Varindex Purity
## Comp 1  61.94     61.94      782 36.038
## Comp 2  24.96     86.91     1245 56.703
## Comp 3  12.65     99.56     1059 46.941
```

The summary shows explained variance, cumulative explained variance, index of pure variable and its purity value for each of the three components. For example, in case of the first component, the purest variable is located in column 782 and its purity (in this case an angle between this variable and a vector of ones) is equal to 36.038 degrees. As one can see, current solution explains 99.6% of the total variance.

Any MCR method in *mdatools* contains the resolved spectra (as `m$resspec`) and contributions (as `m$rescont`) as well as some extra data, e.g. purity spectra for each component, etc. You can see a list of all objects available by simply printing the object with *mcrpure* results:

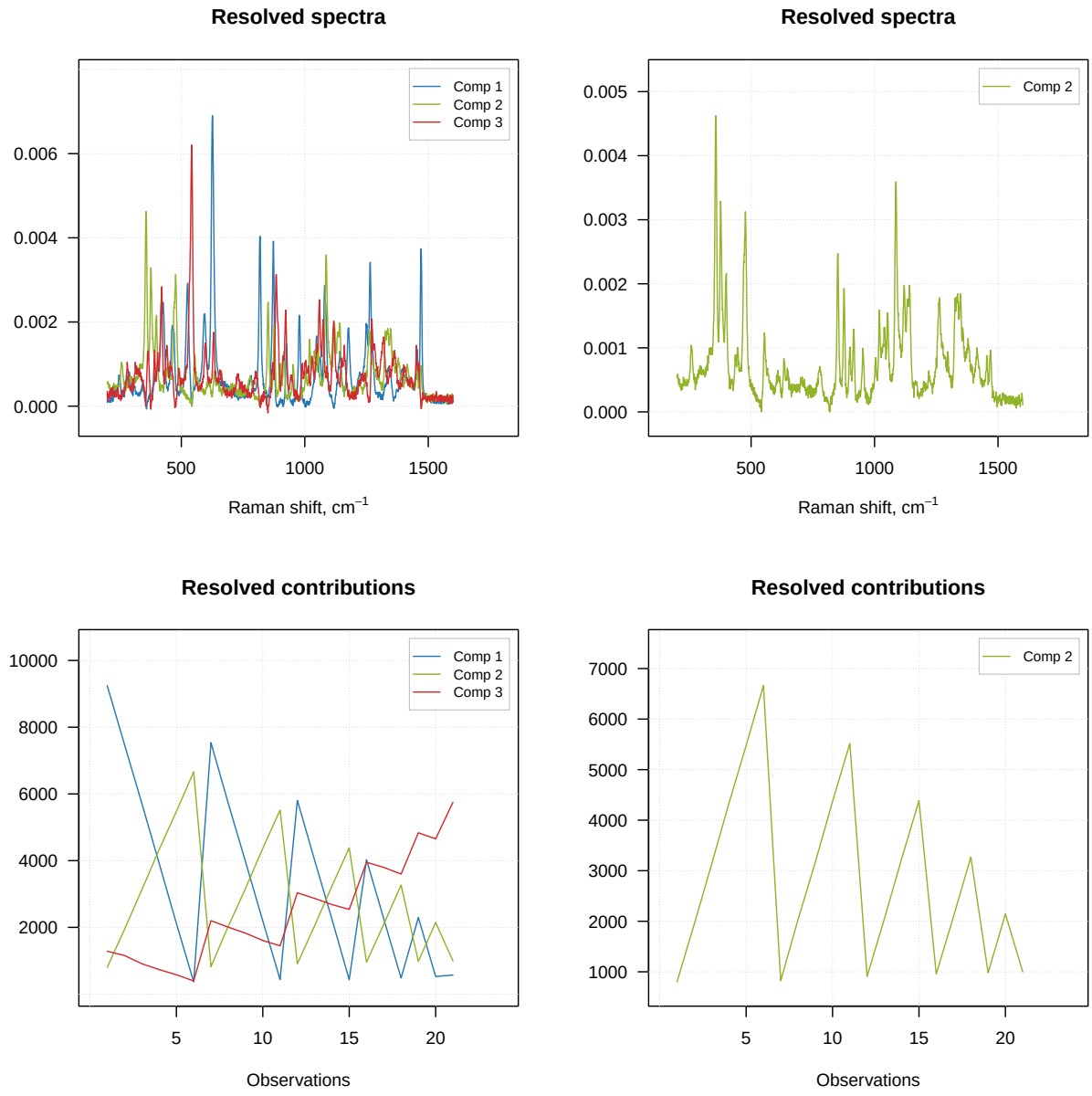
```
show(m)
```

```
##
## MCR Purity unmixing case (class mcrpure)
##
##
## Call:
## mcrpure(x = carbs$D, ncomp = 3)
##
## Major model fields:
## $ncomp - number of calculated components
## $resspec - matrix with resolved spectra
## $rescont - matrix with resolved contributions
## $purityspec - matrix with purity spectra
## $purevars - vector with indices of pure variables
## $purevals - purity values for the selected pure variables
## $expvar - vector with explained variance
## $cumexpvar - vector with cumulative explained variance
## $offset - offset value used to compute the purity
## $info - case info provided by user
```

The reason we use word *contributions* instead of *concentrations* is that the method does not give the real concentrations, measured in the units of interest. The resolved values are rather in arbitrary units and can be later scaled/re-scaled accordingly.

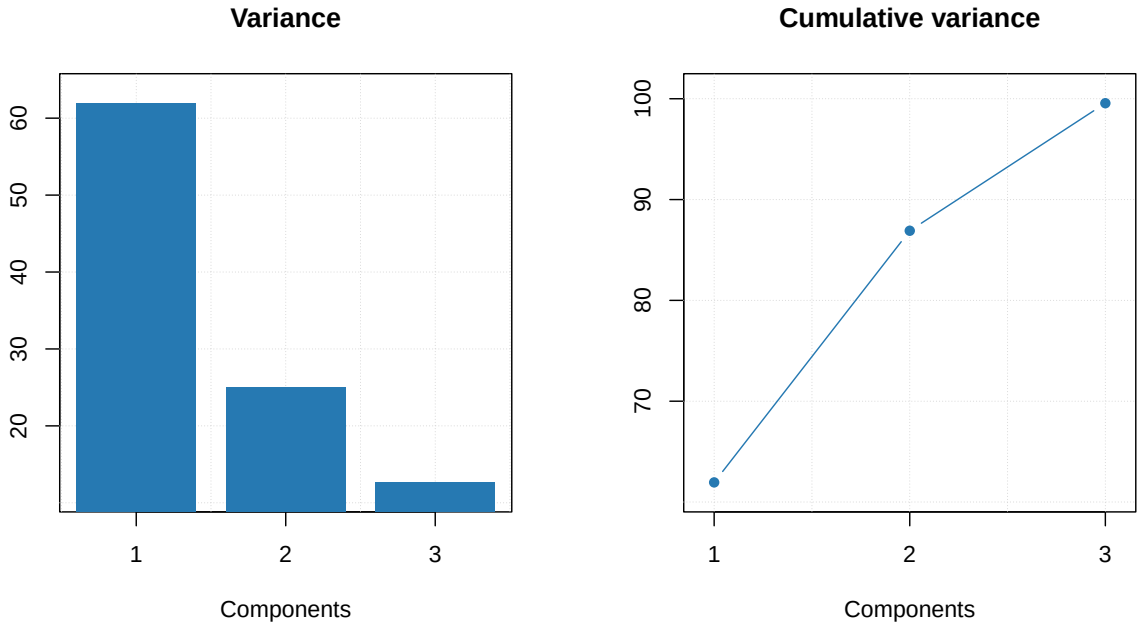
The result object also has several graphical methods, which let user to plot the resolved spectra (`plotSpectra()`) and the resolved contributions (`plotContributions()`). The example below demonstrates how to show the plots for all three components as well as for a selected one.

```
par(mfcol = c(2, 2))
plotSpectra(m)
plotContributions(m)

plotSpectra(m, comp = 2)
plotContributions(m, comp = 2)
```



As you can see, when you select particular components the plot preserves their color. One can also create the explained variance plot (both individual as well as cumulative) as it is shown below.

```
par(mfcol = c(1, 2))
plotVariance(m)
plotCumVariance(m)
```

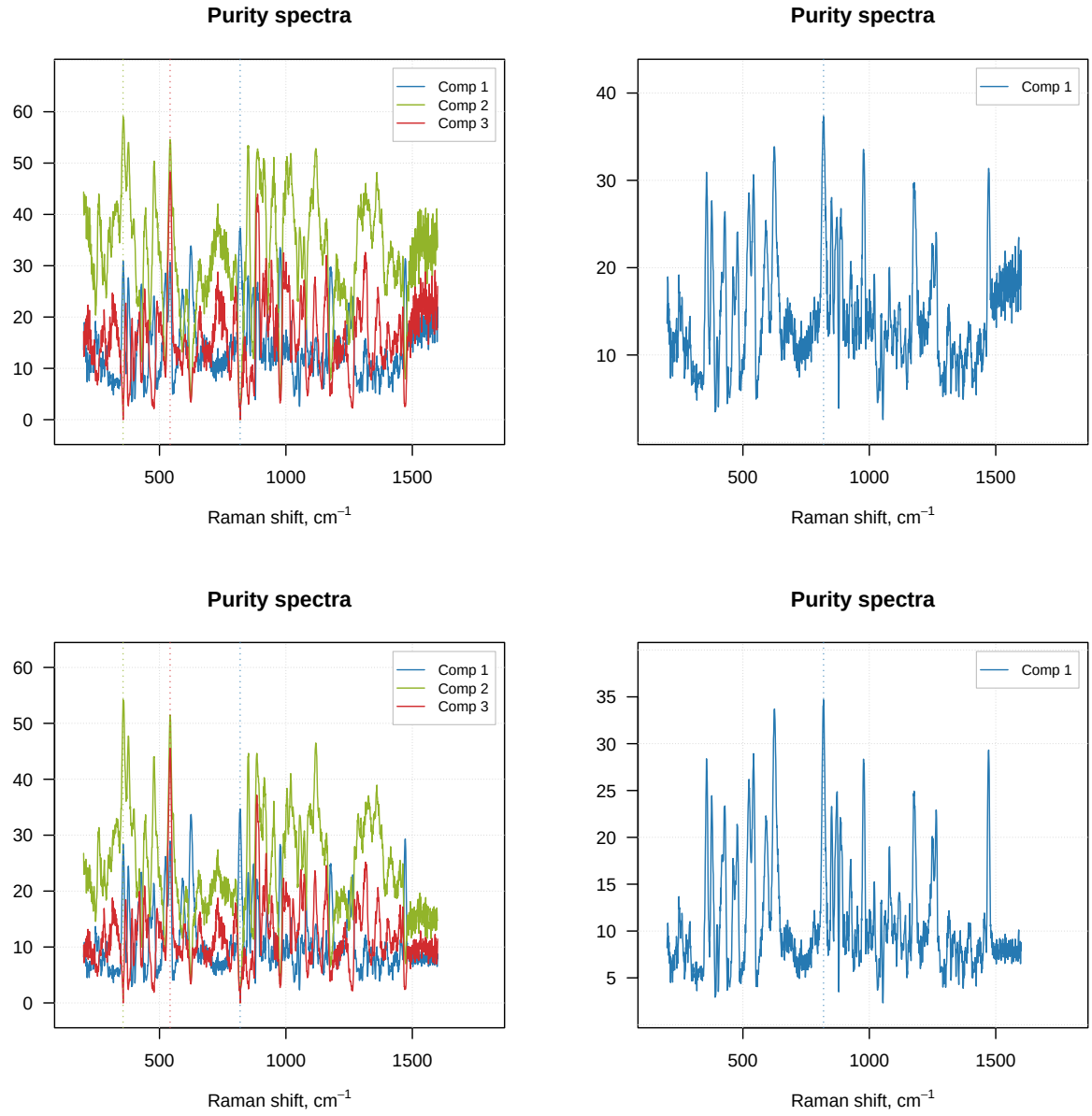


All plot parameters are similar to what you have used for other methods (e.g. PCA, PLS, etc), so you can change type of plot, colors, etc. in a similar way.

Purity values and spectra

The purity spectra and the purity values for each component are quite important and investigation of the spectra can be very useful. For example, the code below creates two solutions using different offset values and shows purity spectra for each solution. The plots on the left side shows purity spectra for all three components, while the plots on the right side show purity only for the first component.

```
m1 = mcrpure(carbs$D, ncomp = 3, offset = 0.01)
m2 = mcrpure(carbs$D, ncomp = 3, offset = 0.10)

par(mfrow = c(2, 2))
plotPuritySpectra(m1)
plotPuritySpectra(m1, comp = 1)
plotPuritySpectra(m2)
plotPuritySpectra(m2, comp = 1)
```

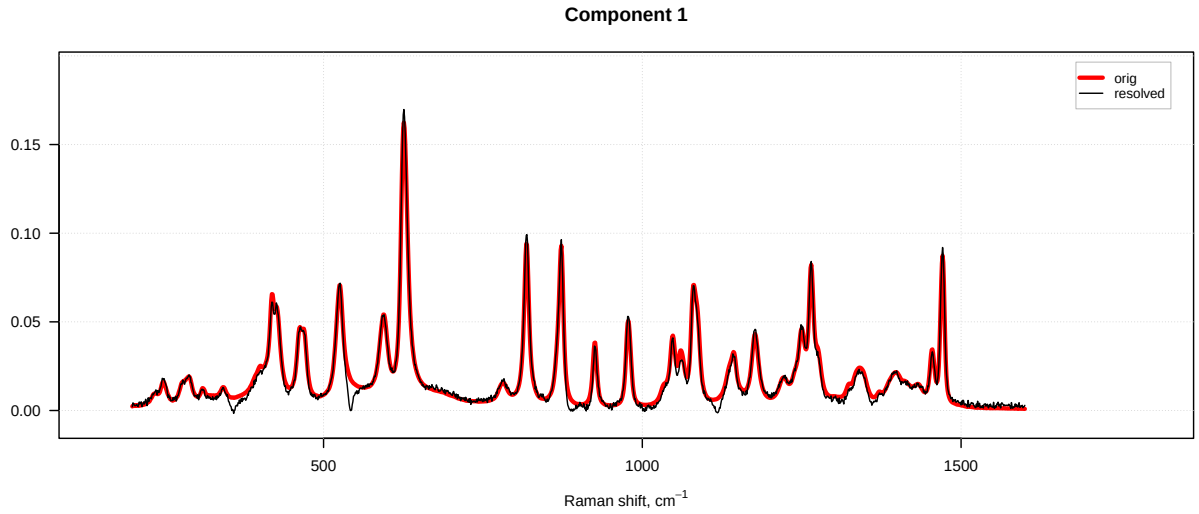


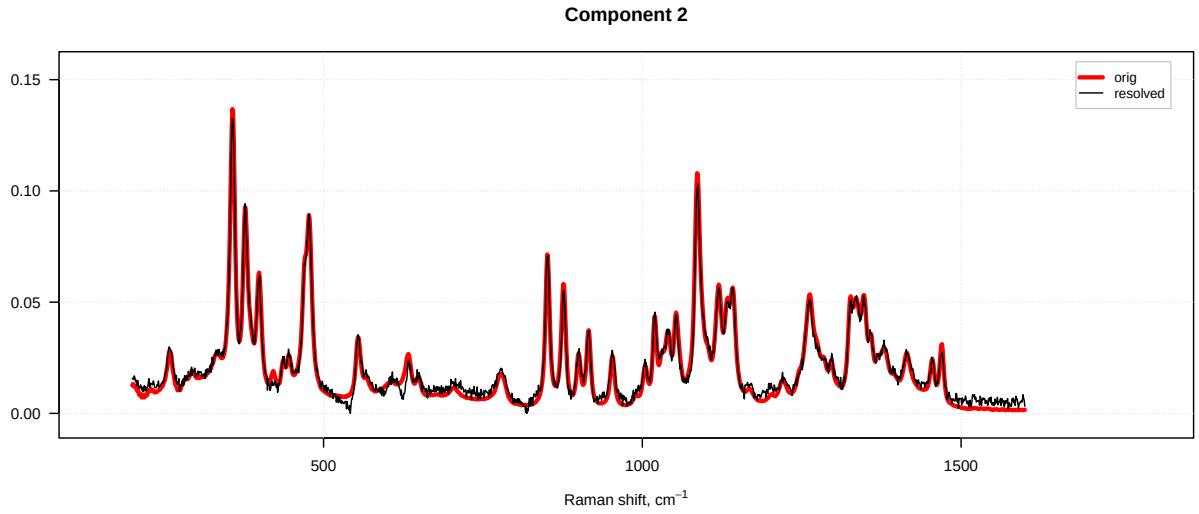
The vertical dashed lines on the purity spectra plot show the selected pure variables. As we can see, indeed they correspond to the largest values in the corresponding purity spectrum (so have the highest purity). We can also notice that when offset is small (top plots, 1%) the purity spectra look quite noisy. On the other hand, using too large offset can lead to a selection of less pure variables, so this parameter should be selected with caution. It is always a good idea to vary the parameter and investigate all plots before the final decision.

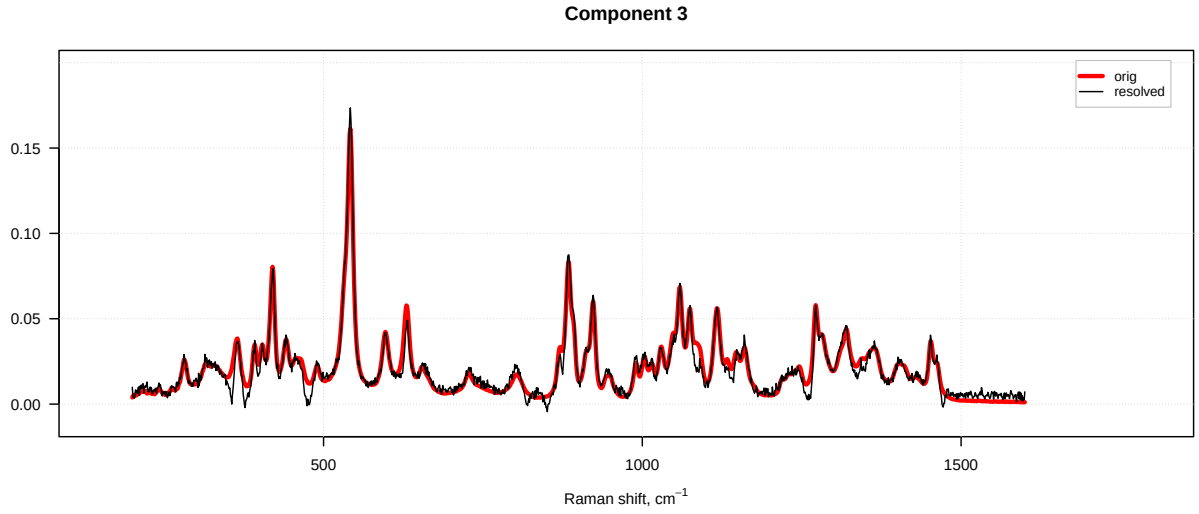
Since dataset `carbs` contains spectra of pure components as well, we can compare the resolved spectra with the original ones as shown in the example below. To do that we normalize both sets of spectra to a unit length to avoid a scaling problem. The original spectra are shown as red and thick curves while the resolved spectra are black and thin.

```
# apply purity method
m = mcrpure(carbs$D, ncomp = 3, offset = 0.05)

# get spectra, transpose and normalize to unit area
```

```
S      = prep.norm(mda.t(carbs$S), "length")
S.hat = prep.norm(mda.t(m1$resspec), "length")

# define color and line width for the spectral curves
col = c("red", "black")
lwd = c(3, 1)

# show the plots separately for each component and each result object
par(mfrow = c(3, 1))
for (a in 1:3) {
   s = mda.subset(S, a)
   s.hat = mda.subset(S.hat, a)
   mdaplotg(list(orig = s, resolved = s.hat), type = "l", col = col, lwd = lwd,
      main = paste0("Component ", a))
}
```







In the code we use `mda.t()` and `mda.subset()` instead of just `t` and `subset()` to preserve values for Raman

shift and axis titles which are defined as attributes of the matrices with spectra, you can read more details in section about Attributes and factors.

As one can see, the quality of resolving is quite good in both cases. However, in case with offset = 10% (right plot), one can also notice large artifacts for example for the third component, which looks like a sort of negative peaks. You can try and play with the offset parameter and see how it influences the quality of the resolution.

Predictions

It is also possible to predict concentration for one or several spectra based on already resolved data. In the code chunk below we predict the concentration values using the matrix with spectra of the pure components. Then we scale the predicted values, so they sum up to one and show the results.

```
c = predict(m, mda.t(carbs$S))
c = c / apply(c, 2, sum)
show(c)
```

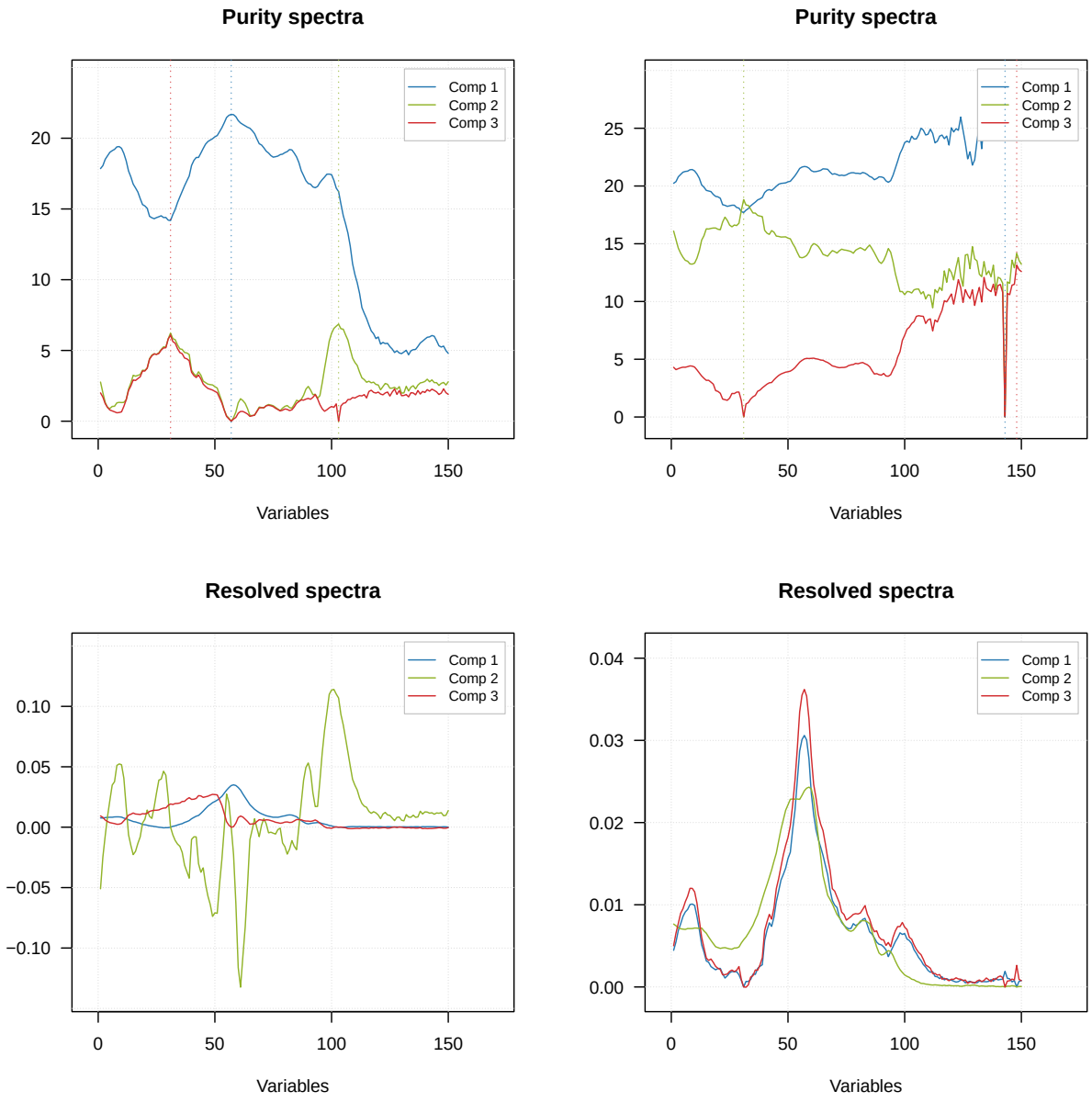
```
##              Comp 1     Comp 2      Comp 3
## fructose 0.94329239 0.03701313 0.094968338
## lactose  0.02396884 0.86686908 0.003781956
## ribose   0.06152230 0.09344879 0.847020106
```

In ideal case we should see an identity matrix. However in this case, we got, for example, $[0.943, 0.024, 0.062]$ for the first component instead of expected $[1, 0, 0]$. The result is a bit worse for the second and the third components.

Tuning the offset parameter

One of the ways to improve the result of resolution based on the purity approach is to tune the offset. The code below shows the result of applying `mcrpure()` to the *Simdata*, which consists of 150 UV/Vis spectra with very broad peaks.

```
data(simdata)
D <- simdata$spectra.c

m1 <- mcrpure(D, ncomp = 3)
m2 <- mcrpure(D, ncomp = 3, offset = 0.001)


par(mfrow = c(2, 2))
plotPuritySpectra(m1)
plotPuritySpectra(m2)

plotSpectra(m1)
plotSpectra(m2)
```



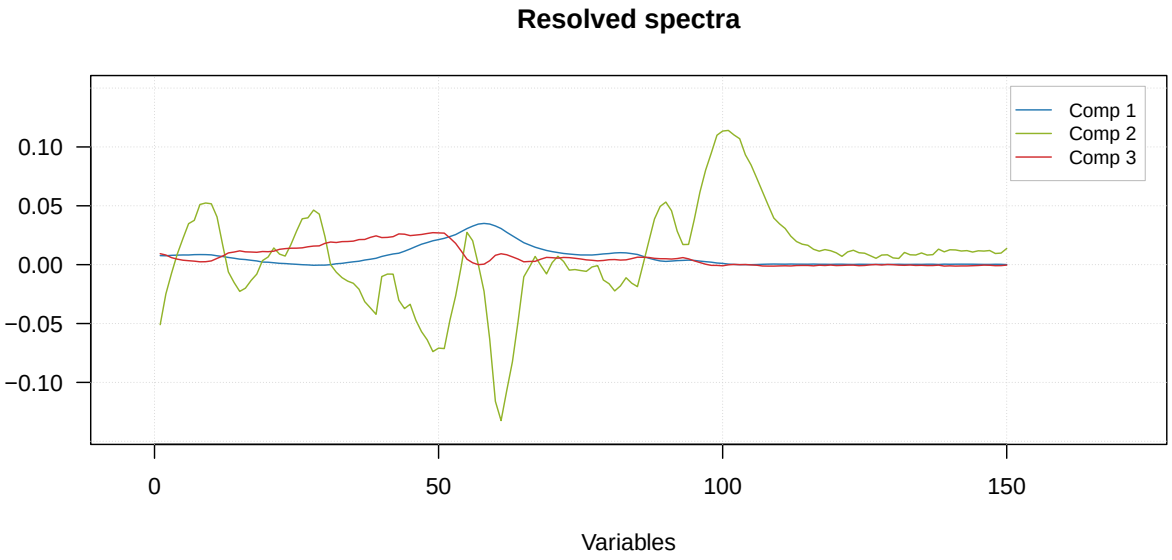
Using the default settings (left plots) does not allow to get any meaningful solutions (you can notice, for example, negative peaks). However, tuning the offset value results in a very good outcome — the resolved spectra are very similar to the original ones (original spectra are not shown on the plots).

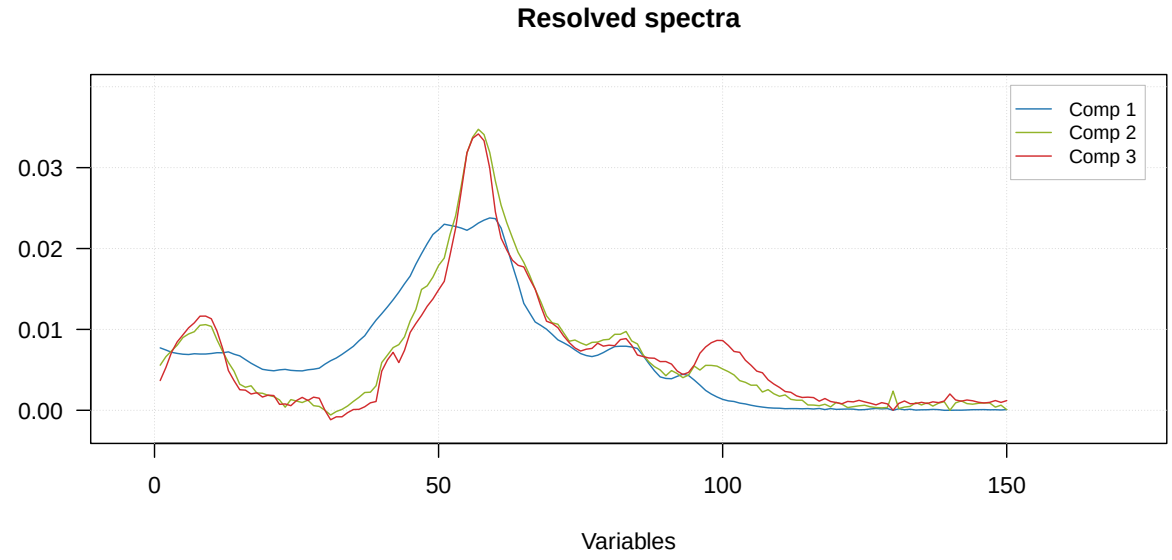
Providing indices of pure variables

Finally you can provide the indices of pure variables if you know them a priory. In this case the method will skip the first step and try to resolve the spectra based on the provided values, as shown in the example below.

```
data(simdata)
D <- simdata$spectra.c

m1 <- mcrpure(D, ncomp = 3)
m2 <- mcrpure(D, ncomp = 3, purevars = c(30, 130, 140))
```

```
par(mfrow = c(2, 1))
plotSpectra(m1)
plotSpectra(m2)
```






```
summary(m1)
```

```
##
## Summary for MCR Purity case (class mcrpure)
##        Expvar Cumexpvar Varindex Purity
## Comp 1  94.75     94.75       57 21.687
## Comp 2  -0.36     94.39      103  6.857
## Comp 3   5.61    100.00       31  6.073
```

```
summary(m2)
```

```
##
## Summary for MCR Purity case (class mcrpure)
##        Expvar Cumexpvar Varindex Purity
## Comp 1  89.69     89.69       30 14.210
## Comp 2   6.15     95.85      130  2.452
## Comp 3   4.15     99.99      140  0.787
```

Alternating least squares

The alternating least squares allows to get $\hat{\mathbf{C}}$ and $\hat{\mathbf{S}}$ by using iterative algorithm, when the ordinary least squares is applied consequently, first to resolve the concentrations by knowing spectra and then to resolve the spectra by using the resolved concentrations:

$$\hat{\mathbf{C}} = \mathbf{D}\hat{\mathbf{S}}(\hat{\mathbf{S}}^T\hat{\mathbf{S}})^{-1}$$

$$\hat{\mathbf{S}} = \mathbf{D}^T\hat{\mathbf{C}}(\hat{\mathbf{C}}^T\hat{\mathbf{C}})^{-1}$$

These two steps continue until a convergence criteria is met. Apparently, there are several issues with the algorithm, that have to be clarified. First of all, to run the first iteration we need to have values for the matrix $\hat{\mathbf{S}}$. This is what is called *initial estimates*. In *mdatools* the initial estimates for the pure component spectra generated automatically as a matrix with random values taken from uniform distribution (between 0 and 1). Apparently the matrix $(\hat{\mathbf{S}}^T\hat{\mathbf{S}})$ should not be singular and using random values will ensure this.

On the other hand, using random values does not always provide reproducible results, so some alternatives can be considered. Therefore user can provide any pre-computed values for $\hat{\mathbf{S}}$ using parameter `spec.ini` as it will also shown below.

The second issue is that using ordinary least squares (OLS) method for the iterations will result in a solution with negative values, which is not physically meaningful. To tackle this problem we either need to constrain the solution, for example by setting all negative values to zero, after each iteration, or by using non-negative least squares for solving the equations. In *mdatools* both options are available.

Finally, as we mentioned before, curve resolution problem does not have a unique solution. To narrow the range of feasible solutions down, and also to move them towards the right one, we can apply different constraints. Some of the constraints are implemented and available for user, some will be implemented later. However, user can easily provide a manual constraint function as it will be shown below.

We will take all these issues gradually and let's start with non-negativity and constraints.

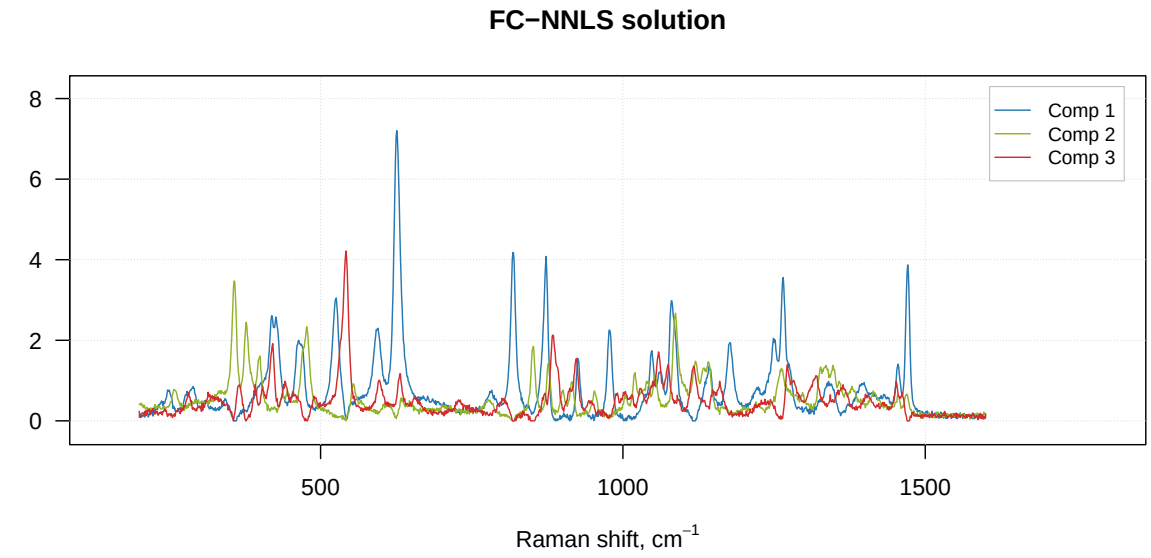
Non-negativity

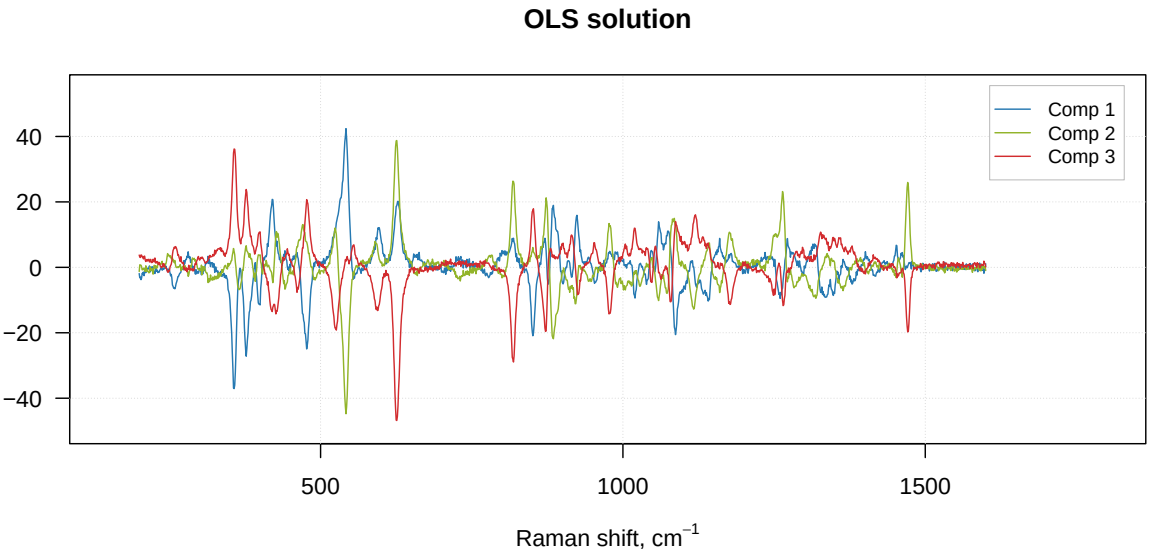
To tackle the non-negativity problem, in *mdatools* you can chose one of the following options:

1. Use standard OLS solver and apply non-negativity constraint by setting all negative values to zero.
2. Use non-negative least squares solver (NNLS), e.g. proposed by Lawsen and Hanson.
3. Use its faster version — Fast Combinatorial NNLS (FC-NNLS) proposed by Benthem and Keenan.
4. Use your own solver

The default option is nr. 3 and this is what we would recommend to use. It gives a solution identical to nr. 2 however is much faster in case of multivariate data. You can also provide a function, which implements your own solver. The solver can be changed both for spectra and for contributions, you can even use different solvers for each. The code below shows how to apply MCR-ALS using OLS and FC-NNLS solvers without any constraints.

```
data(carbs)

# apply MCR-ALS with default solver (FC-NNLS)
m1 = mcrals(carbs$D, ncomp = 3)

# apply MCR-ALS with OLS solver
m2 = mcrals(carbs$D, ncomp = 3, spec.solver = mcrals.ols, cont.solver = mcrals.ols)

# show the resolved spectra
par(mfrow = c(2, 1))
plotSpectra(m1, main = "FC-NNLS solution")
plotSpectra(m2, main = "OLS solution")
```





As you can see, the OLS solution without non-negativity constraint contains negative values. Once again, if

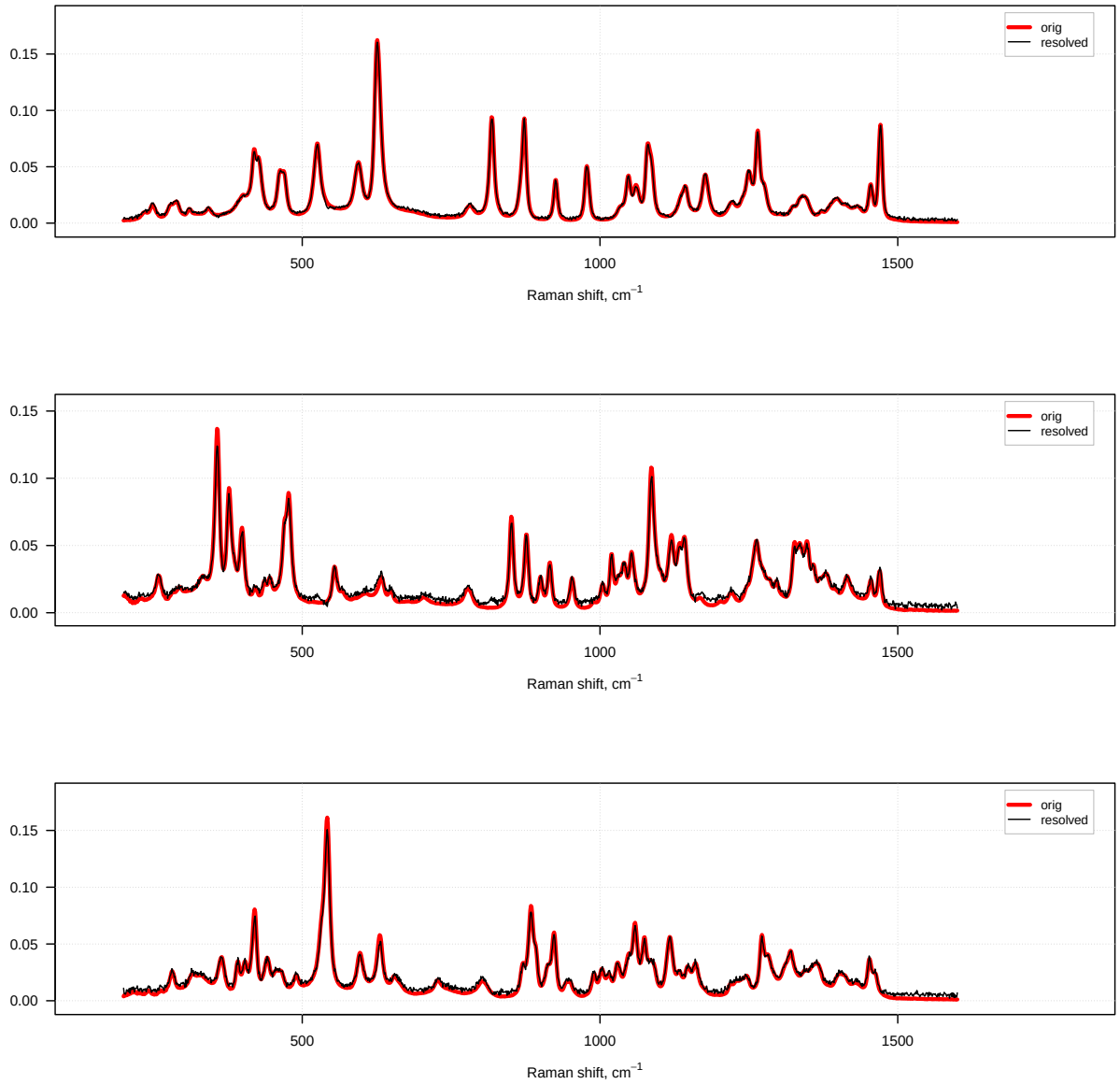
you do not experiment with negative values, using the default option (FCNNLS) will be the best.

Constraints

Constraints are small functions that can be applied separately to current estimates of spectra or contributions on every step of the algorithm in order to improve the solution. A constraint function takes the matrix with spectra or contributions as an input, does something with the values and returns the matrix of the same size as an output. For example, the simplest constraint is non-negativity, which sets all negative values in the matrix to zeros.

In *mdatools* constraints can be combined together using lists. Spectra and contributions should have separate sets of constrains. In the example below we show how to use some of the built in constraints:

```
# define constraints for contributions
cc = list(
   constraint("norm", params = list(type = "sum"))
)

# define constraints for spectra
sc = list(
   constraint("angle", params = list(weight = 0.05)),
   constraint("norm", params = list(type = "length"))
)

# run MCR-ALS with the constraints
set.seed(6)
m = mcrals(carbs$D, ncomp = 3, spec.constraints = sc, cont.constraints = cc)

# normalzie the original spectra to unit length
S = prep.norm(mda.t(carbs$S), "length")

# get resolved spectra and transpose them
S.hat = mda.t(m$resspec)

# define color and line width for the spectral curves
col = c("red", "black")
lwd = c(3, 1)

# show the plots separately for each component and each result object
par(mfrow = c(3, 1))
for (a in 1:3) {
   s = mda.subset(S, a)
   s.hat = mda.subset(S.hat, a)
   mdaplotg(list(orig = s, resolved = s.hat), type = "l", col = col, lwd = lwd)
}
```



As you can see, we use one constraint to normalize the contributions, so they sum up to one on every step. And we use two constraints for the spectra — first angle constraint, which makes spectra less contrast and increases contrast among the resolved concentration profiles. Then we normalize the spectra to the unit length. The constraints are applied to the spectra or contributions on every iteration and in the same order as they are defined in the list.

You may have noticed the following instruction: `set.seed(6)` in the code block above. This is needed to make results more reproducible. Since the initial estimates for the resolved spectra are obtained using random number generator, the final result can be different from run to run. To avoid this we seed the generator with number 6, so the final result will be the same regardless how many times we run this code.

The list of available constraints and their parameters can be shown by running `constraints.list()`:

```
constraints.list()
```

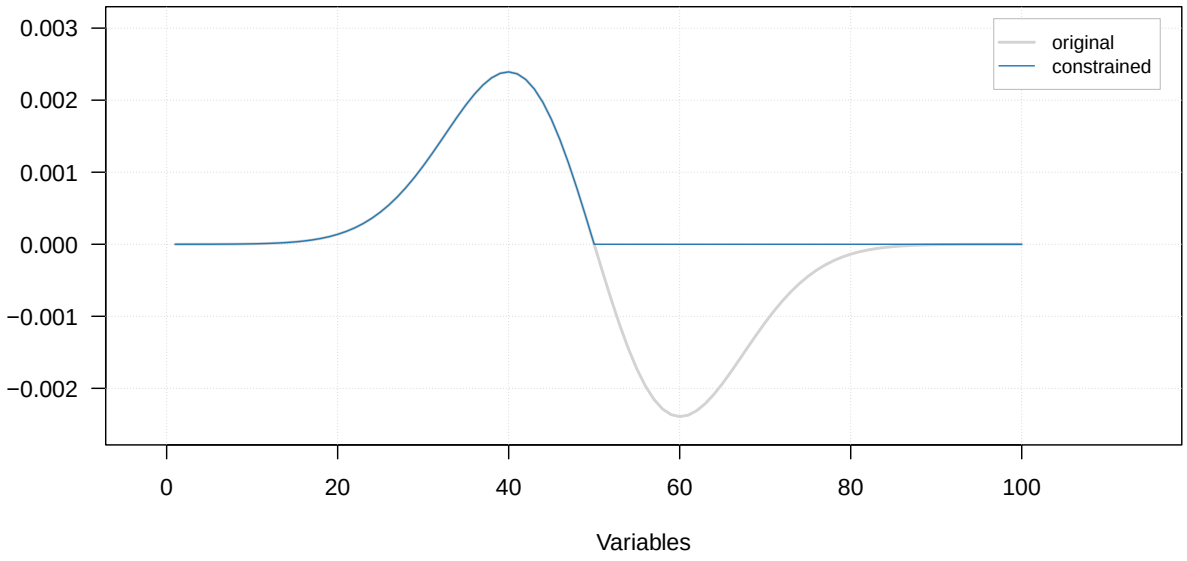
```
##
```

```
## List of constraints available for mcrals():
##
##
##  Non-negativity (sets negative values to zero)
##  ---------------
##  name: 'nonneg'
##  no parameters required
##
##
##  Unimodality (forces contribution or spectral profile to have a single maximum)
##  ---------------
##  name: 'unimod'
##  parameters:
##   'tol': tolerance (between 0 and 1)
##
##
##  Closure (forces contributions or spectral profiles sum up to constant value)
##  ---------------
##  name: 'closure'
##  parameters:
##   'sum': value, the data rows should sum up to
##
##
##  Normalization (normalize spectra or contributions)
##  ---------------
##  name: 'norm'
##  parameters:
##   'type': type of normalization: 'length', 'area' or 'sum'
##
##
##  Angle (increases contrast among resolved spectra or contributions)
##  ---------------
##  name: 'angle'
##  parameters:
##   'weight': how much of mean will be added: between 0 and 1
```

In the next few subsections you will find a short description of the currently implemented constraints with some illustrations, as well as instructions how to make a user defined constraint.

Non-negativity constraint

The non-negativity constraint simply takes a signal (spectra or contributions for a particular component) and set all negative values to zeros. A plot below demonstrates how the constraint works, the gray line represents the original signal, the blue — signal after applying the constraint.



It does not have any parameters, so to add the constraint to the list simply use `constraint("nonneg")`, e.g.:

```
cc <- list(
   constraint("nonneg")
)
```

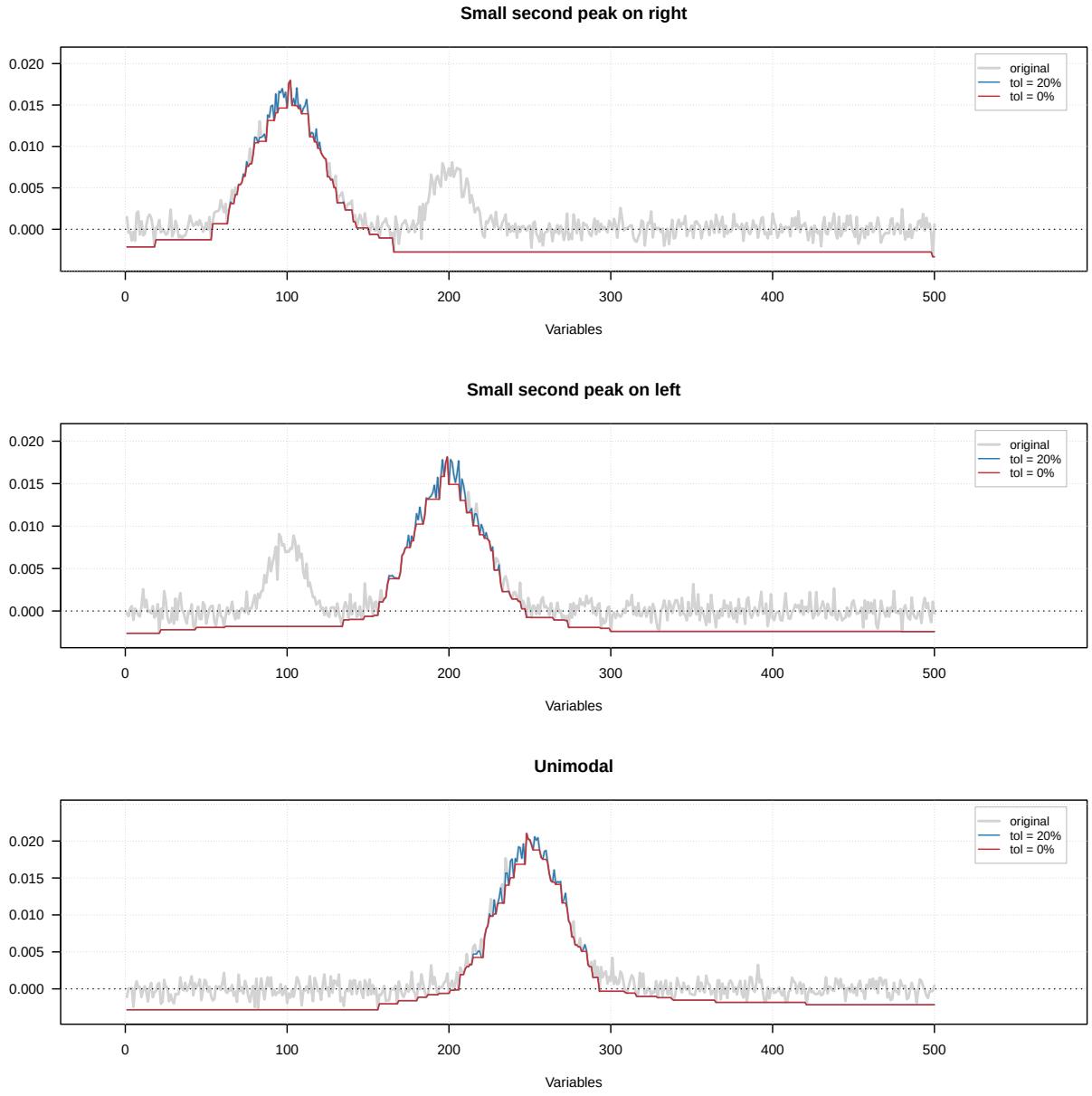
Unimodality constraint

The unimodality constraint is used to force signals having only one peak (maximum). This can be particularly useful for constraining contribution profiles, e.g. when spectra came from a reaction and it is appriory known that concentration developing of components has one peak.

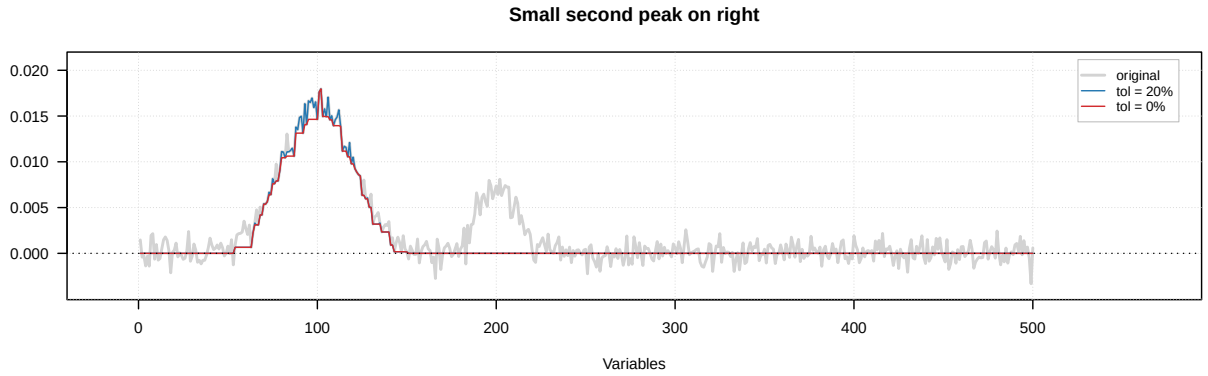
The constraint works as follows. First it finds a global maximum in the signal. Then it goes along each side from the peak and checks if intensity of next point in the signal does not exceed the value of the previous one. If it does, constraint will replace the current value by the previous one.

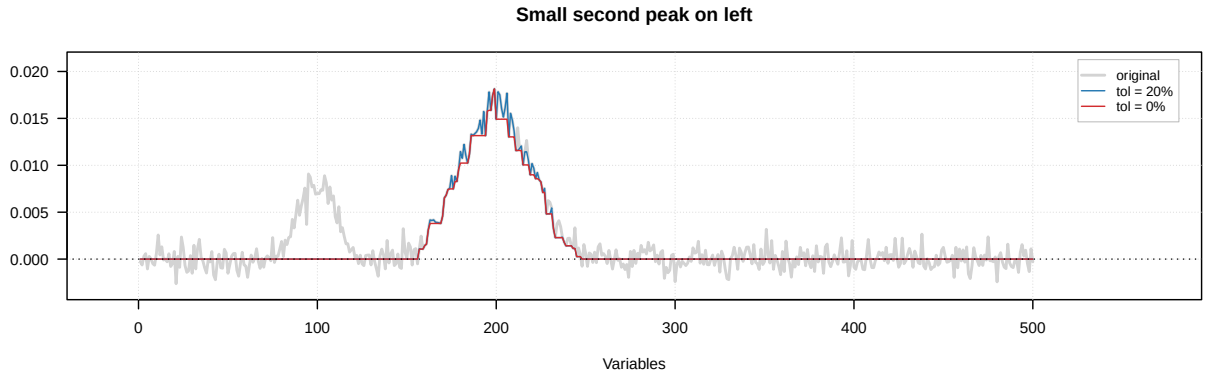
It is also possible to define a tolerance — value between 0 and 1, which allows taking into account small fluctuation of the intensity due to e.g. noise. Thus if tolerance is 0.05, then only when intensity of current point exceeds the intensity of the previous point by more than 5% it will trigger the constraint.

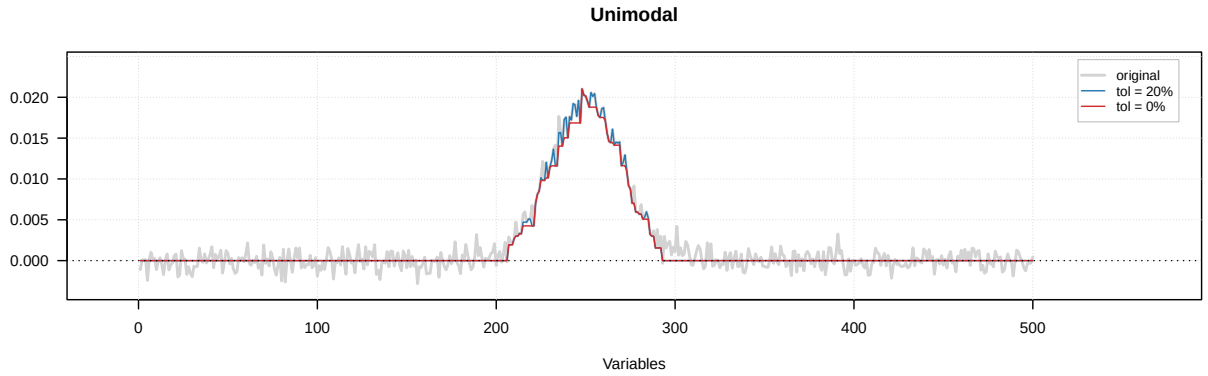
The figure below shows how the constraint works using three signals — one with small extra peak on the left, one with small extra peak on the right and one unimodal signal. A small amount of noise was added to all signals. There are one original signal (gray) and two constrained signals on the plot. The red one shows the result for 0% tolerance and the blue one shows the result for 20% tolerance.



As we can see, because of the noise, the constrained profiles have negative values. In this case it makes sense to combine the unimodality and non-negativity constrains. The result of this combination is shown below.







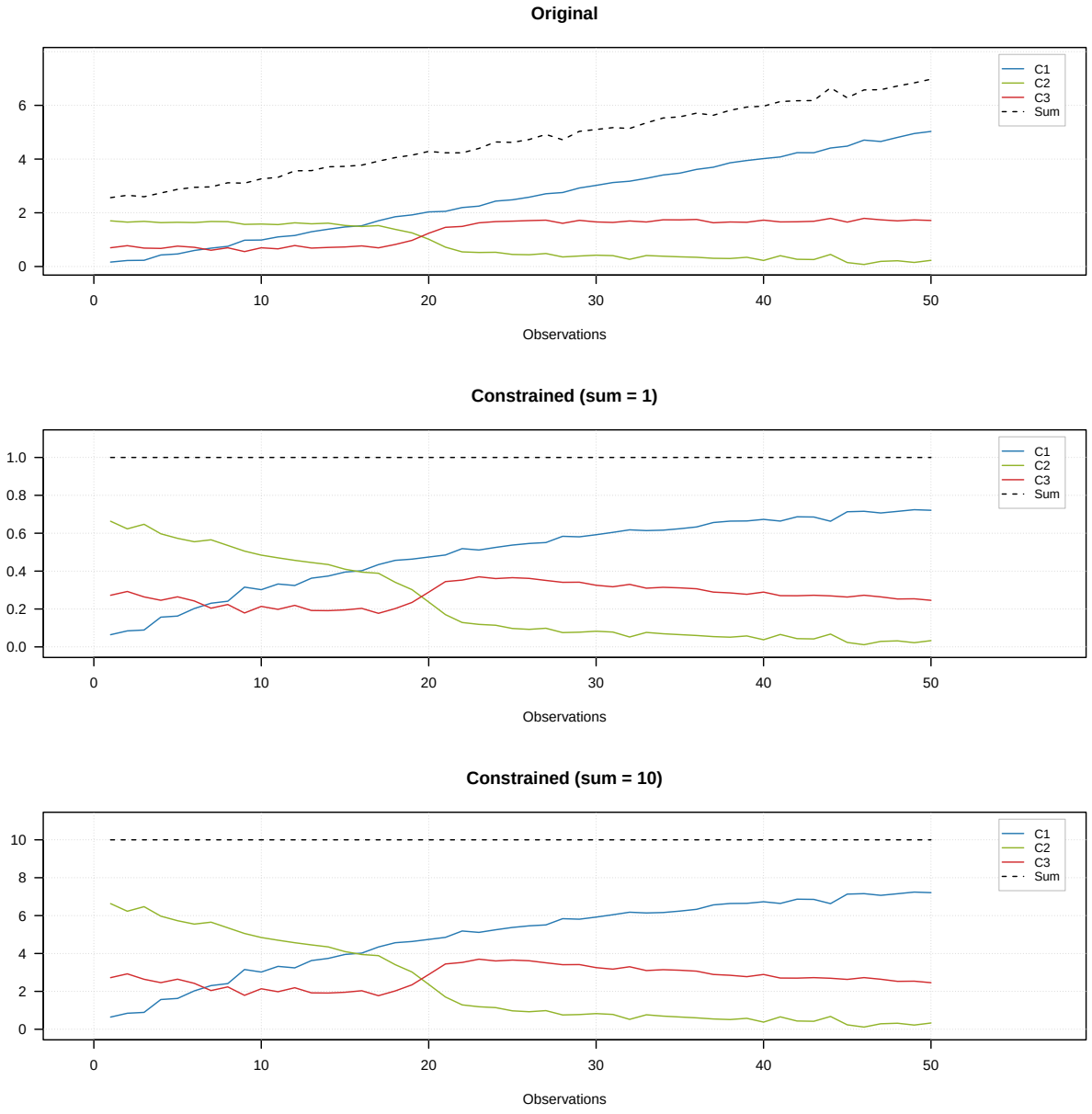
In order to add the constraint use `constraint("unimod")` for zero tolerance or `constraint("unimod", params = list(tol = 0.05))` for 5% tolerance. In the code chunck below you see how to create a set of unimodality and non-negativity constrains e.g. for contribution profiles

```
cc <- list(
   constraint("unimod", params = list(tol = 0.05)),
   constraint("nonneg")
)
```

Closure constraint

Closure constraint, like unimodality, is mostly applicable to the contribution profiles and aims at preserve the mass balance — so sum of the concetrations of pure components is constant. From that point of view the closure constraint can be thought of normalization made along the observations (or wavelength/wavenumbers in case of spectra) instead of along the components.

The figure below shows how it works using contribution profiles of three components. The colored lines show the profiles and the black dashed line shows the sum of contributions for a given measurements.



The top plot shows the original profiles. The middle plot shows the constrained profiles, where the closure constrained which limits sum of the contributions to one was applied. The bottom plot shows results for the constrains limited the sum to 10.

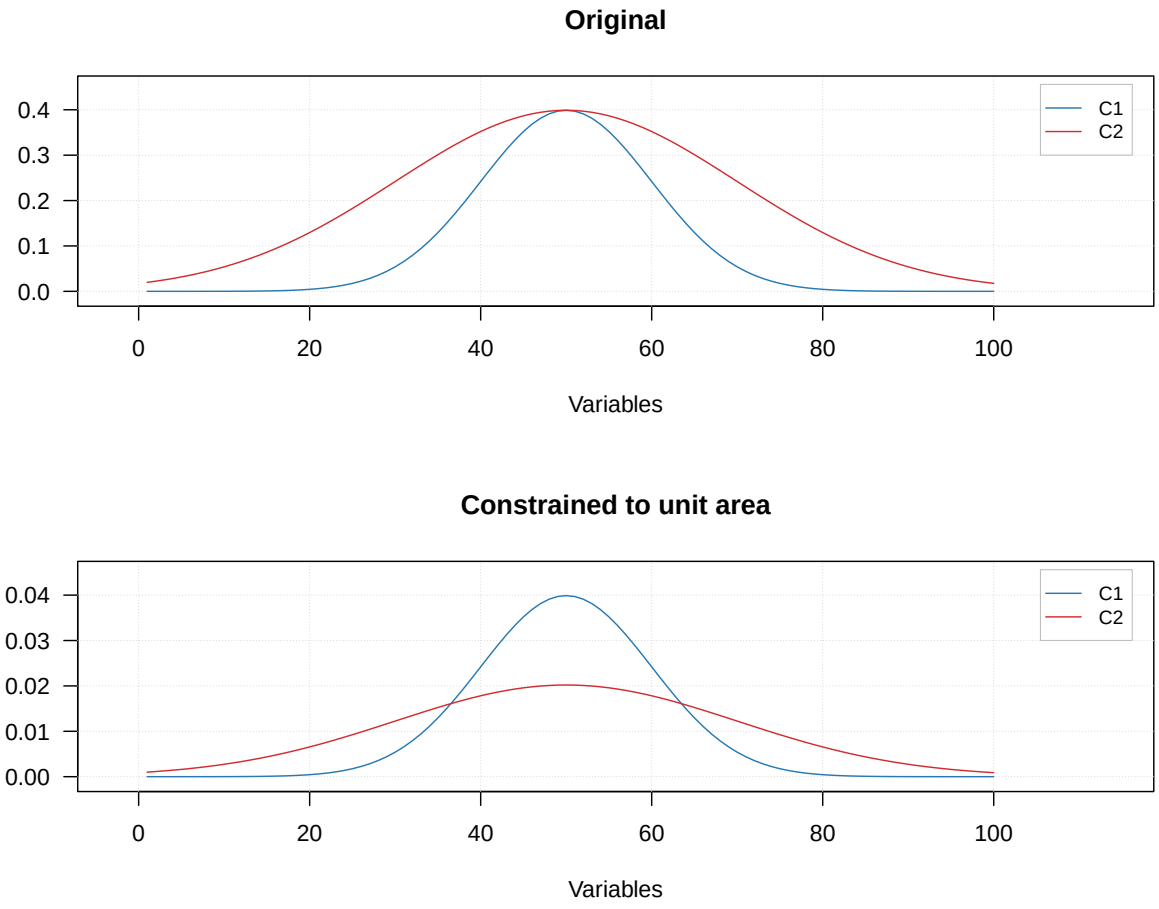
As you can see in the last two plots, the dashed line show the constant desired value for all measurements. In order to add the constraint use `constraint("closure")` for unit sum or `constraint("closure", params = list(sum = 10))` for e.g. 10.

Normalization constraint

Normalization costraint aims at making spectra or contribution profiles of invididual components normalized. The normalization can be made by unit length, area or sum. For example, in case of sum normalization, sum

of all values of each spectrum or contribution profile will be equal to one.

An example below shows two original signals and the result of normalization of the signals to the unit area (so area under each curve is equal to one).



The constrain can be added using `constraint("closure", params = list(type = "area"))`. The value for parameter `type` can be `"area"`, `"length"`, or `"sum"`

Angle constraint

Angle constraint allows to change a contrast among the contribution profiles or resolved spectra. The idea was proposed by Windig *at al* and all details can be found in this paper.

The constraint works as follows. First, mean spectrum or mean contribution profile are computed using the original data, **D**. Then, on every iteration, a portion of the mean signal (e.g. 5%) is added, for example, to the resolved contributions. This will lead to smaller contrast among the contribution profiles but, at the same time, will increase the contrast for the resolved spectra. And vice versa, to increase the contrast among the resolved contribution profiles, the constraint should be applied to the spectra.

The constraint can be added using `constraint("angle")`. By default the portion of mean to be added to the signals is 5% (0.05). To change this value to, e.g. 10%, use `constraint("angle", params = list(weight = 0.10))`.

User defined constraints

We plan to implement more constraints gradually. However you do not need to wait and can use your own constraints, if necessary. This section shows how to do it step by step and how to combine user defined constraint with the implemented ones.

Let's say, we want to add a random noise to our resolved spectra on each iteration (sounds very stupid, but let's use it just for fun in this example). So, first we need to create the constraint function. It should have two mandatory arguments: `x` which will take a current estimates of spectra or contributions and `d` which is original data used for resolving. There can be any number of optional arguments, in our case we will use argument `noise.level` which will define a level of noise in percent of maximum value in `x`:

Here is the function:

```
# constraint function which adds random noise from normal distribution
myConstraintFunction <- function(x, d, noise.level = 0.01) {
   nr <- nrow(x)
   nc <- ncol(x)
   noise <- rnorm(nr * nc, mean = 0, sd = max(x) * noise.level)

   return(x + matrix(noise, nr, nc))
}
```

As you can see we use normal distribution for generating the noise, with zero mean and standard deviation equal to the 1% of maximum value by default. Load this function to the global environment by running this code.

Next we need to create a constraint object, which will use this function, and set the required level of noise, like it is shown here (we take 2% instead of the default value of 1%):

```
# create a constraint with noise level of 2%
myConstraint = list(method = myConstraintFunction, params = list(noise.level = 0.02))
class(myConstraint) = "constraint"
```

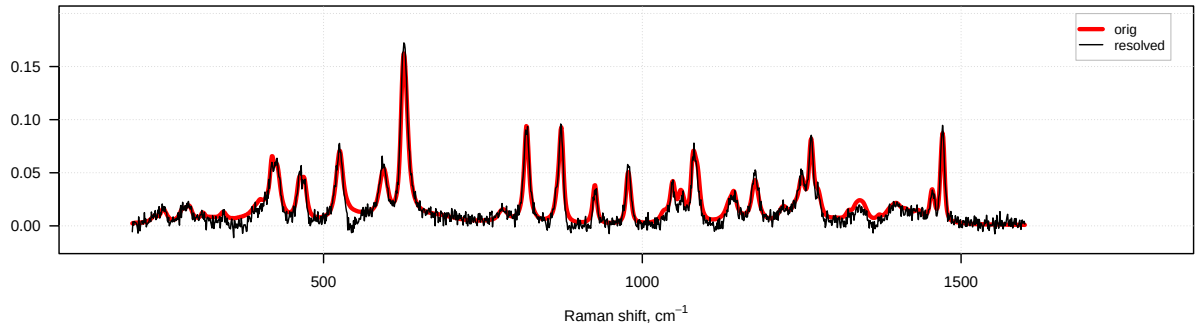
Next we need to define sets of constraints for contributions and for spectra by using lists, like we did for built in constraints, and add our manual constraint object to the list:

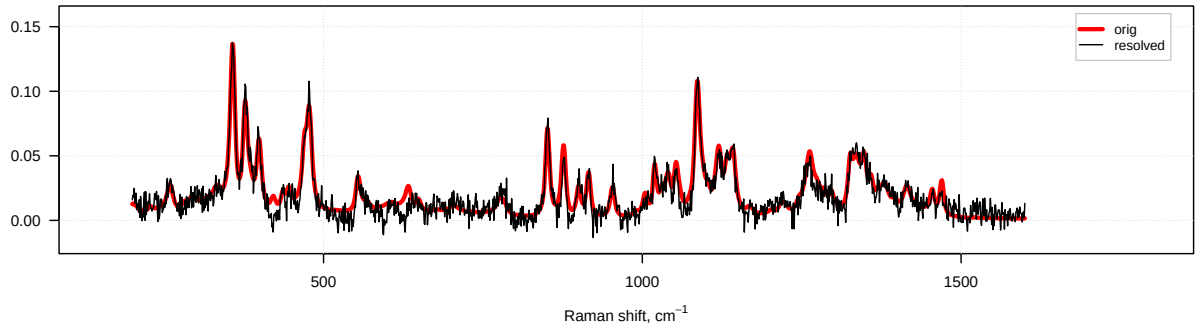
```
# constraints for contributions
cc = list(
   constraint("norm", params = list(type = "sum"))
)

# constraints for spectra
sc = list(
   myConstraint,
   constraint("norm", params = list(type = "length"))
)
```

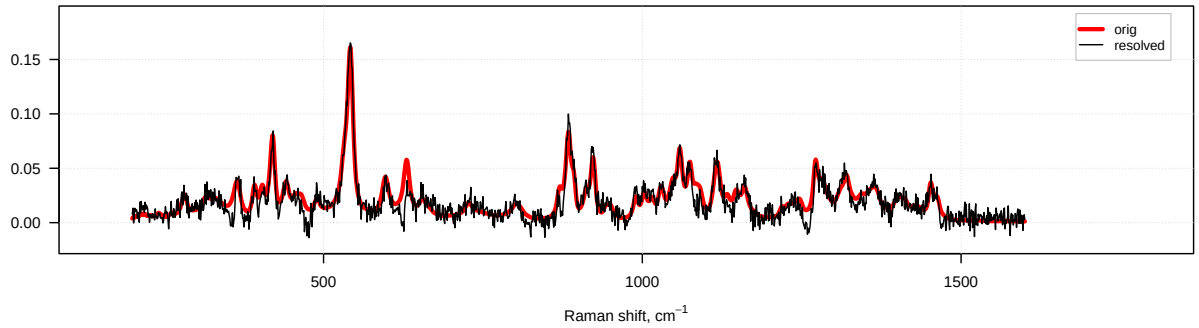
Finally, we just run the MCR-ALS algorithm and check the results. I will again seed the random number generator to make results reproducible.

```
# run mcrals
set.seed(6)
m <- mcrals(carbs$D, ncomp = 3, cont.constraints = cc, spec.constraints = sc)

# normalize the original spectra to unit length
S = prep.norm(mda.t(carbs$S), "length")

# get the resolved spectra and transpose them
```

```
S.hat = mda.t(m$resspec)

# define color and line width for the spectral curves
col = c("red", "black")
lwd = c(3, 1)

# show the plots separately for each component and each result object
par(mfrow = c(3, 1))
for (a in 1:3) {
   s = mda.subset(S, a)
   s.hat = mda.subset(S.hat, a)
   mdaplotg(list(orig = s, resolved = s.hat), type = "l", col = col, lwd = lwd)
}
```







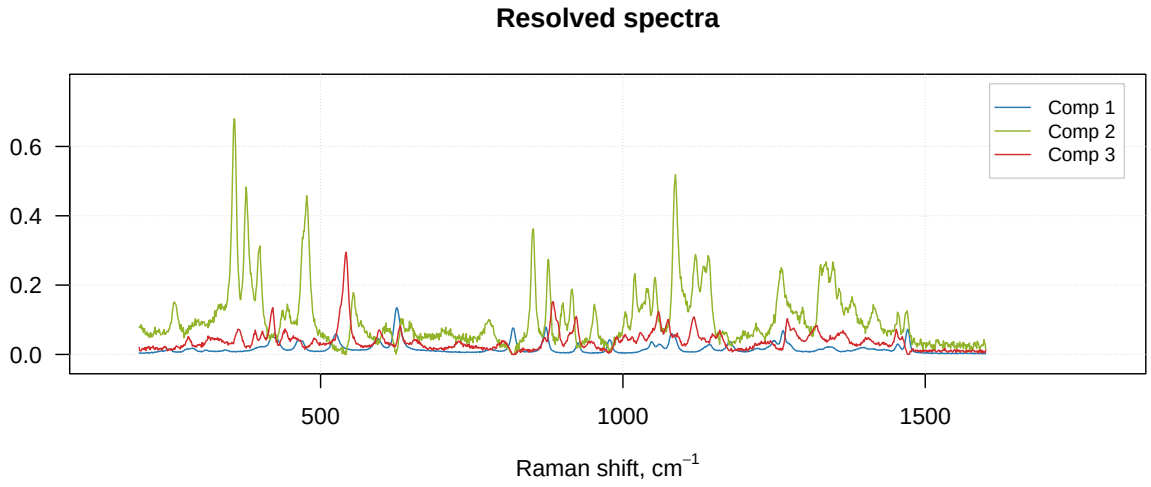
As you can see, even with this rather strange constraint the method works, although the resolved spectra are quite noisy. The presence of negative values in the resolved spectra is because we use normal distribution and the constraint is applied after the solver, which provides non-negative solution.

Initial estimates

As it was mentioned above, one can provide any initial estimates for the spectra as a matrix with size $J \times A$, where A is the number of components and J is number of variables (columns) in the original data. However, the matrix with provided values should not be singular, otherwise the algorithm will fail.

In the example below we provide absolute values of PCA loadings as the initial estimates. No constraints are provided in this case for the sake of simplicity.

```
m.pca = pca(carbs$D, ncomp = 3, center = FALSE)
m = mcrals(carbs$D, ncomp = 3, spec.ini = abs(m.pca$loadings))

plotSpectra(m)
```



Forcing contribution and spectral values

Sometimes we know the exact concentration of components or their absence for given measurements. For example we may know the concentration of ingredients at the beginning of a reaction or measure concentration of reaction products at the end. Same for the spectra, e.g. we know wavenumbers or wavelength where no peaks are expected for a particular component. In order to provide this information we can let `mcrals()` force these values to be equal to zero. Actually it can be any number, but since the resolved contributions and spectral intensities are relative, you need to be careful with providing the non-zero values.

In order to use this possibility you need to provide values for two parameters: `cont.forced` for the contributions and `spec.forced` for the spectra. For example if we resolve spectra and contributions for three component system and we do not expect presence of third component at the beginning of a reaction, we can set the values as follows:

```
cont.forced = matrix(NA, nrow(D), ncomp)
cont.forced[1, 3] = 0
```

The first line in this chunk creates a matrix and fill it with missing (`NA`) values. The matrix has the same dimension as future matrix with resolved contributions — same number of rows as the matrix with mixtures, `D`, and same number of columns as the expected number of pure components.

After that, we set a value at the first row and the third column to zero. Which means "force concentration of the third component for the first observation to zero". Inside the ALS loop, on every iteration the `mcrals()` method will replace the resolved concentration with the provided value.

Here is an example, based on the carbs data we used before. In this case I add the pure spectra on top of the matrix with mixtures. And since these are spectra of pure components, I know that the spectrum for the first observation does not contain any of C2 and C3, the second observation does not contain C1 and C3 and, finally, the third observation does not contain C1 and C2. Here is how to do this:

```
ncomp <- 3
Dplus <- mda.rbind(mda.t(carbs$S), carbs$D)

cont.forced = matrix(NA, nrow(Dplus), ncomp)
cont.forced[1, ] = c(NA, 0, 0)
cont.forced[2, ] = c(0, NA, 0)
cont.forced[3, ] = c(0, 0, NA)

m <- mcrals(Dplus, ncomp = ncomp, cont.forced = cont.forced)
```

Here are the resolved contributions:

```
show(head(m$rescont))
```

```
##         Comp 1     Comp 2     Comp 3
## [1,] 7.6462723  0.0000000 0.11344608
## [2,] 0.1331184 13.1168977 0.09604429
## [3,] 0.1492307  0.0000000 7.61280985
## [4,] 7.8180191  0.6992354 0.64461538
## [5,] 6.3207464  3.2922424 0.70466107
## [6,] 4.8315353  5.9818678 0.60008529
```

As one can see, the ones we forced to be zero, are either zeros or just very small numbers. Forcing the spectral values works in a similar way.

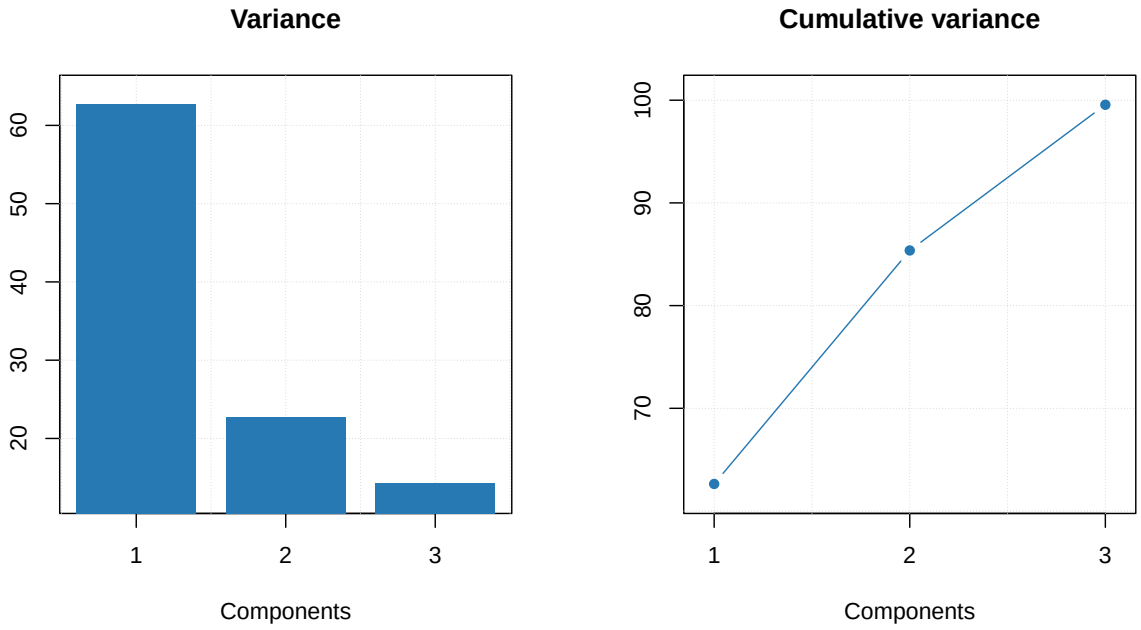
Overview of resolving process and results

The `mcrals()` also computes the explained variance, which can be shown using `plotVariance()` and `plotCumVariance()` functions. It also has `predict()` method which works similar to `mcrpure`. Some examples are shown below:

```
# define constraints for contributions
cc = list(
   constraint("norm", params = list(type = "sum"))
)

# define constraints for spectra
sc = list(
   constraint("angle", params = list(weight = 0.05)),
   constraint("norm", params = list(type = "length"))
)

# run MCR-ALS with the constraints
set.seed(6)
m = mcrals(carbs$D, ncomp = 3, spec.constraints = sc, cont.constraints = cc)

# show summary
summary(m)
```

```
##
## Summary for MCR ALS case (class mcrals)
##
## Constraints for spectra:
##  -  angle
##  -  norm
##
## Constraints for contributions:
##  -  norm
##
##        Expvar Cumexpvar
## Comp 1  62.64     62.64
## Comp 2  22.73     85.37
## Comp 3  14.19     99.56
```

```
# make variance plots
par(mfrow = c(1, 2))
plotVariance(m)
plotCumVariance(m)
```



Since MCR-ALS is iterative algorithm, it should stop when convergence criteria is met. There are two ways to stop MCR-ALS iterations in *mdatools*. First is a tolerance (`tol`) parameter, which tells a minimum difference in explained variance to make the algorithm continue. For example, if explained variance on previous step is 0.95467 and on the current step it is 0.95468, the difference is 0.00001 and the algorithm will continue if the tolerance is smaller than this value.

Another way to limit the iterations is to define the maximum number by using parameter `max.iter`. By default is is set to 100. The `mcrals()` has a verbose mode which shows improvements in each iteration as it is demonstrated in the example below (we use the same constraint as in the previous chunk of code)

```
m = mcrals(carbs$D, ncomp = 3, spec.constraints = sc, cont.constraints = cc, verbose = TRUE)
```

```
##
## Starting iterations (max.niter = 100):
```

```
## Iteration    1, R2 = 0.000000
## Iteration    2, R2 = 0.957402
## Iteration    3, R2 = 0.976722
## Iteration    4, R2 = 0.987305
## Iteration    5, R2 = 0.992613
## Iteration    6, R2 = 0.994525
## Iteration    7, R2 = 0.995165
## Iteration    8, R2 = 0.995385
## Iteration    9, R2 = 0.995449
## Iteration   10, R2 = 0.995453
## No more improvements.
```