

Introduction to the **pls** Package

Bjørn-Helge Mevik

University Center for Information Technology, University of Oslo
Norway

Ron Wehrens

Biostatistics and Data Management
Center for Research and Innovation, Fondazione Edmund Mach
San Michele all'Adige (TN)
Italy

August 22, 2015

Abstract

The **pls** package implements Principal Component Regression (PCR) and Partial Least Squares Regression (PLSR) in R, and is freely available from the CRAN website, licensed under the Gnu General Public License (GPL).

The user interface is modelled after the traditional formula interface, as exemplified by **lm**. This was done so that people used to R would not have to learn yet another interface, and also because we believe the formula interface is a good way of working interactively with models. It thus has methods for generic functions like **predict**, **update** and **coef**. It also has more specialised functions like **scores**, **loadings** and **RMSEP**, and a flexible cross-validation system. Visual inspection and assessment is important in chemometrics, and the **pls** package has a number of plot functions for plotting scores, loadings, predictions, coefficients and RMSEP estimates.

The package implements PCR and several algorithms for PLSR. The design is modular, so that it should be easy to use the underlying algorithms in other functions. It is our hope that the package will serve well both for interactive data analysis and as a building block for other functions or packages using PLSR or PCR.

We will here describe the package and how it is used for data analysis, as well as how it can be used as a part of other packages. Also included is a section about formulas and data frames, for people not used to the R modelling idioms.

1 Introduction

This vignette is meant as an introduction to the **pls** package. It is based on the paper ‘The pls Package: Principal Component and Partial Least Squares Regression in R’, published in *Journal of Statistical Software* [17].

The PLSR methodology is shortly described in Section 2. Section 3 presents an example session, to get an overview of the package. In Section 4 we describe formulas and data frames (as they are used in **pls**). Users familiar with formulas and data frames in R can skip this section on first reading. Fitting of models is described in Section 5, and cross-validatory choice of components is discussed in Section 6. Next, inspecting and plotting models is described

(Section 7), followed by a section on predicting future observations (Section 8). Finally, Section 9 covers more advanced topics such as parallel computing, setting options, using the underlying functions directly, and implementation details.

2 Theory

Multivariate regression methods like Principal Component Regression (PCR) and Partial Least Squares Regression (PLSR) enjoy large popularity in a wide range of fields, including the natural sciences. The main reason is that they have been designed to confront the situation that there are many, possibly correlated, predictor variables, and relatively few samples—a situation that is common, especially in chemistry where developments in spectroscopy since the seventies have revolutionised chemical analysis. In fact, the origin of PLSR lies in chemistry (see, e.g., [20, 12]). The field of *near-infrared* (NIR) spectroscopy, with its highly overlapping lines and difficult to interpret overtones, would not have existed but for a method to obtain quantitative information from the spectra.

Also other fields have benefited greatly from multivariate regression methods like PLSR and PCR. In medicinal chemistry, for example, one likes to derive molecular properties from the molecular structure. Most of these Quantitative Structure-Activity Relations (QSAR, and also Quantitative Structure-Property Relations, QSPR), and in particular, Comparative Molecular Field Analysis (ComFA) [2], use PLSR. Other applications range from statistical process control [10] to tumour classification [18] to spatial analysis in brain images [15] to marketing [5].

In the usual multiple linear regression (MLR) context, the least-squares solution for

$$\mathbf{Y} = \mathbf{X}\mathbf{B} + \mathcal{E} \quad (1)$$

is given by

$$\mathbf{B} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{Y} \quad (2)$$

The problem often is that $\mathbf{X}^T \mathbf{X}$ is singular, either because the number of variables (columns) in \mathbf{X} exceeds the number of objects (rows), or because of collinearities. Both PCR and PLSR circumvent this by decomposing \mathbf{X} into orthogonal scores \mathbf{T} and loadings \mathbf{P}

$$\mathbf{X} = \mathbf{T}\mathbf{P} \quad (3)$$

and regressing \mathbf{Y} not on \mathbf{X} itself but on the first a columns of the scores \mathbf{T} . In PCR, the scores are given by the left singular vectors of \mathbf{X} , multiplied with the corresponding singular values, and the loadings are the right singular vectors of \mathbf{X} . This, however, only takes into account information about \mathbf{X} , and therefore may be suboptimal for prediction purposes. PLSR aims to incorporate information on both \mathbf{X} and \mathbf{Y} in the definition of the scores and loadings. In fact, for one specific version of PLSR, called SIMPLS [4], it can be shown that the scores and loadings are chosen in such a way to describe as much as possible of the *covariance* between \mathbf{X} and \mathbf{Y} , where PCR concentrates on the *variance* of \mathbf{X} . Other PLSR algorithms give identical results to SIMPLS in the case of one \mathbf{Y} variable, but deviate slightly for the multivariate \mathbf{Y} case; the differences are not likely to be important in practice.

2.1 Algorithms

In PCR, we approximate the \mathbf{X} matrix by the first a Principal Components (PCs), usually obtained from the singular value decomposition (SVD):

$$\mathbf{X} = \tilde{\mathbf{X}}_{(a)} + \mathcal{E}_X = (\mathbf{U}_{(a)}\mathbf{D}_{(a)})\mathbf{V}_{(a)}^T + \mathcal{E}_X = \mathbf{T}_{(a)}\mathbf{P}_{(a)}^T + \mathcal{E}_X$$

Next, we regress \mathbf{Y} on the scores, which leads to regression coefficients

$$\mathbf{B} = \mathbf{P}(\mathbf{T}^T\mathbf{T})^{-1}\mathbf{T}^T\mathbf{Y} = \mathbf{V}\mathbf{D}^{-1}\mathbf{U}^T\mathbf{Y}$$

where the subscripts a have been dropped for clarity.

For PLSR, the components, called Latent Variables (LVs) in this context, are obtained iteratively. One starts with the SVD of the crossproduct matrix $\mathbf{S} = \mathbf{X}^T\mathbf{Y}$, thereby including information on variation in both \mathbf{X} and \mathbf{Y} , and on the correlation between them. The first left and right singular vectors, w and q , are used as weight vectors for \mathbf{X} and \mathbf{Y} , respectively, to obtain scores t and u :

$$t = \mathbf{X}w = \mathbf{E}w \quad (4)$$

$$u = \mathbf{Y}q = \mathbf{F}q \quad (5)$$

where \mathbf{E} and \mathbf{F} are initialised as \mathbf{X} and \mathbf{Y} , respectively. The X scores t are often normalised:

$$t = t/\sqrt{t^T t} \quad (6)$$

The Y scores u are not actually necessary in the regression but are often saved for interpretation purposes. Next, X and Y loadings are obtained by regressing against the *same* vector t :

$$p = \mathbf{E}^T t \quad (7)$$

$$q = \mathbf{F}^T t \quad (8)$$

Finally, the data matrices are ‘deflated’: the information related to this latent variable, in the form of the outer products tp^T and tq^T , is subtracted from the (current) data matrices \mathbf{E} and \mathbf{F} .

$$\mathbf{E}_{n+1} = \mathbf{E}_n - tp^T \quad (9)$$

$$\mathbf{F}_{n+1} = \mathbf{F}_n - tq^T \quad (10)$$

The estimation of the next component then can start from the SVD of the crossproduct matrix $\mathbf{E}_{n+1}^T\mathbf{F}_{n+1}$. After every iteration, vectors w , t , p and q are saved as columns in matrices \mathbf{W} , \mathbf{T} , \mathbf{P} and \mathbf{Q} , respectively. One complication is that columns of matrix \mathbf{W} can not be compared directly: they are derived from successively deflated matrices \mathbf{E} and \mathbf{F} . It has been shown that an alternative way to represent the weights, in such a way that all columns relate to the original \mathbf{X} matrix, is given by

$$\mathbf{R} = \mathbf{W}(\mathbf{P}^T\mathbf{W})^{-1} \quad (11)$$

Now, we are in the same position as in the PCR case: instead of regressing \mathbf{Y} on \mathbf{X} , we use scores \mathbf{T} to calculate the regression coefficients, and later convert these back to the realm of the original variables by pre-multiplying with matrix \mathbf{R} (since $\mathbf{T} = \mathbf{X}\mathbf{R}$):

$$\mathbf{B} = \mathbf{R}(\mathbf{T}^T\mathbf{T})^{-1}\mathbf{T}^T\mathbf{Y} = \mathbf{R}\mathbf{T}^T\mathbf{Y} = \mathbf{R}\mathbf{Q}^T$$

Again, here, only the first a components are used. How many components are optimal has to be determined, usually by cross-validation.

Many alternative formulations can be found in literature. It has been shown, for instance, that only one of \mathbf{X} and \mathbf{Y} needs to be deflated; alternatively, one can directly deflate the crossproduct matrix \mathbf{S} (as is done in SIMPLS, for example). Moreover, there are many equivalent ways of scaling. In the example above, the scores t have been normalised, but one can also choose to introduce normalisation at another point in the algorithm. Unfortunately, this can make it difficult to directly compare the scores and loadings of different PLSR implementations.

2.2 On the use of PLSR and PCR

In theory, PLSR should have an advantage over PCR. One could imagine a situation where a minor component in \mathbf{X} is highly correlated with \mathbf{Y} ; not selecting enough components would then lead to very bad predictions. In PLSR, such a component would be automatically present in the first LV. In practice, however, there is hardly any difference between the use of PLSR and PCR; in most situations, the methods achieve similar prediction accuracies, although PLSR usually needs fewer latent variables than PCR. Put the other way around: with the same number of latent variables, PLSR will cover more of the variation in \mathbf{Y} and PCR will cover more of \mathbf{X} . In turn, both behave very similar to ridge regression [6].

It can also be shown that both PCR and PLSR behave as shrinkage methods [8], although in some cases PLSR seems to increase the variance of individual regression coefficients, one possible explanation of why PLSR is not always better than PCR.

3 Example session

In this section we will walk through an example session, to get an overview of the package.

To be able to use the package, one first has to load it:

```
> library(pls)
```

This prints a message telling that the package has been attached, and that the package implements a function `loadings` that masks a function of the same name in package `stats`. (The output of the commands have in some cases been suppressed to save space.)

Three example data sets are included in `pls`:

yarn A data set with 28 near-infrared spectra (**NIR**) of PET yarns, measured at 268 wavelengths, as predictors, and density as response (**density**) [19]. The data set also includes a logical variable **train** which can be used to split the data into a training data set of size 21 and test data set of size 7. See `?yarn` for details.

oliveoil A data set with 5 quality measurements (**chemical**) and 6 panel sensory panel variables (**sensory**) made on 16 olive oil samples [14]. See `?oliveoil` for details.

gasoline A data set consisting of octane number (**octane**) and NIR spectra (**NIR**) of 60 gasoline samples [9]. Each NIR spectrum consists of 401 diffuse reflectance measurements from 900 to 1700 nm. See `?gasoline` for details.

These will be used in the examples that follow. To use the data sets, they must first be loaded:

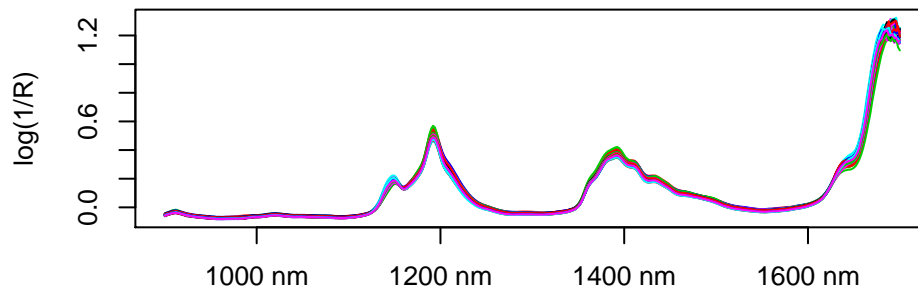


Figure 1: Gasoline NIR spectra

```
> data(yarn)
> data(oliveoil)
> data(gasoline)
```

For the rest of the paper, it will be assumed that the package and the data sets have been loaded as above. Also, all examples are run with `options(digits = 4)`.

In this section, we will do a PLSR on the `gasoline` data to illustrate the use of `pls`. The spectra are shown in Figure 1. We first divide the data set into train and test data sets:

```
> gasTrain <- gasoline[1:50,]
> gasTest <- gasoline[51:60,]
```

A typical way of fitting a PLSR model is

```
> gas1 <- pls(octane ~ NIR, ncomp = 10, data = gasTrain, validation = "LOO")
```

This fits a model with 10 components, and includes *leave-one-out* (LOO) cross-validated predictions [11]. We can get an overview of the fit and validation results with the `summary` method:

```
> summary(gas1)
```

```
Data:           X dimension: 50 401
          Y dimension: 50 1
Fit method: kernelpls
Number of components considered: 10
```

VALIDATION: RMSEP

Cross-validated using 50 leave-one-out segments.

	(Intercept)	1 comps	2 comps	3 comps	4 comps	5 comps	6 comps
CV	1.545	1.357	0.2966	0.2524	0.2476	0.2398	0.2319
adjCV	1.545	1.356	0.2947	0.2521	0.2478	0.2388	0.2313
	7 comps	8 comps	9 comps	10 comps			
CV	0.2386	0.2316	0.2449	0.2673			
adjCV	0.2377	0.2308	0.2438	0.2657			

TRAINING: % variance explained

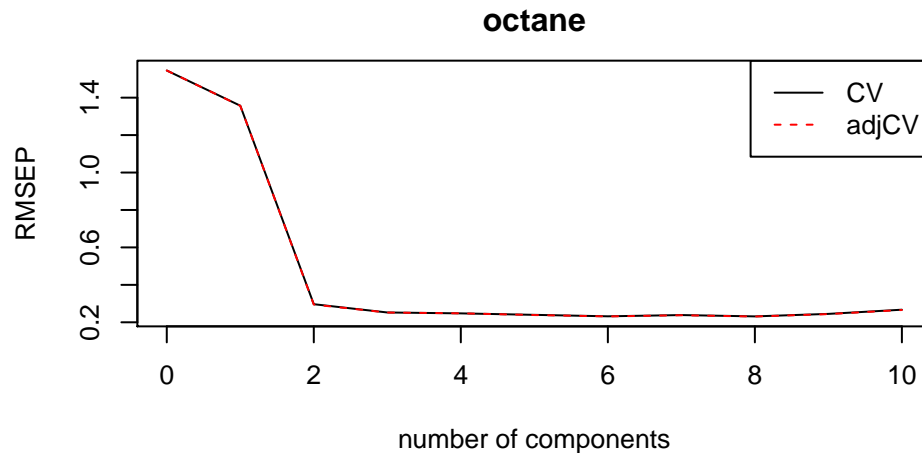


Figure 2: Cross-validated RMSEP curves for the gasoline data

	1 comps	2 comps	3 comps	4 comps	5 comps	6 comps	7 comps	8 comps
X	78.17	85.58	93.41	96.06	96.94	97.89	98.38	98.85
octane	29.39	96.85	97.89	98.26	98.86	98.96	99.09	99.16
	9 comps	10 comps						
X	99.02	99.19						
octane	99.28	99.39						

The validation results here are *Root Mean Squared Error of Prediction* (RMSEP). There are two cross-validation estimates: **CV** is the ordinary CV estimate, and **adjCV** is a bias-corrected CV estimate [16]. (For a LOO CV, there is virtually no difference).

It is often simpler to judge the RMSEPs by plotting them:

```
> plot(RMSEP(gas1), legendpos = "topright")
```

This plots the estimated RMSEPs as functions of the number of components (Figure 2). The **legendpos** argument adds a legend at the indicated position. Two components seem to be enough. This gives an RMSEP of 0.297. As mentioned in the introduction, the main practical difference between PCR and PLSR is that PCR often needs more components than PLSR to achieve the same prediction error. On this data set, PCR would need three components to achieve the same RMSEP.

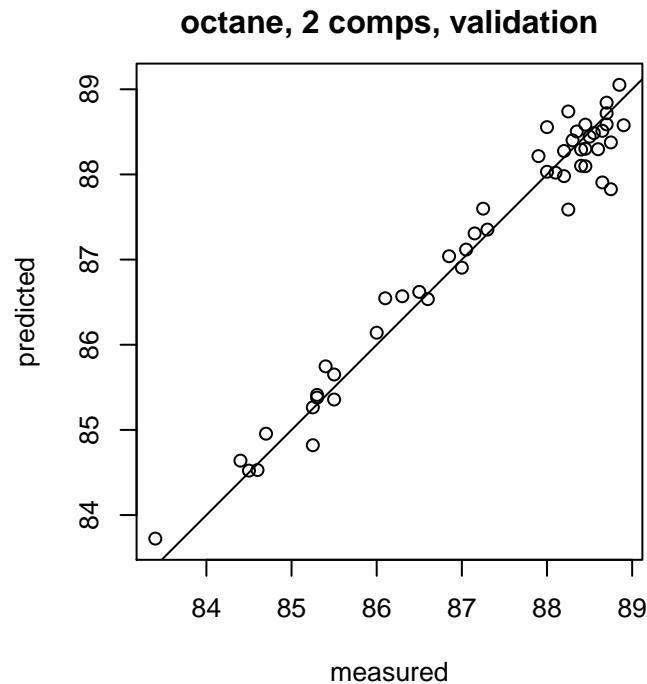
Once the number of components has been chosen, one can inspect different aspects of the fit by plotting predictions, scores, loadings, etc. The default plot is a prediction plot:

```
> plot(gas1, ncomp = 2, asp = 1, line = TRUE)
```

This shows the cross-validated predictions with two components versus measured values (Figure 3). We have chosen an aspect ratio of 1, and to draw a target line. The points follow the target line quite nicely, and there is no indication of a curvature or other anomalies.

Other plots can be selected with the argument **plottype**:

```
> plot(gas1, plottype = "scores", comps = 1:3)
```

Figure 3: Cross-validated predictions for the `gasoline` data

This gives a pairwise plot of the score values for the three first components (Figure 4). Score plots are often used to look for patterns, groups or outliers in the data. (For instance, plotting the two first components for a model built on the `yarn` dataset clearly indicates the experimental design of that data.) In this example, there is no clear indication of grouping or outliers. The numbers in parentheses after the component labels are the relative amount of X variance explained by each component. The explained variances can be extracted explicitly with

```
> explvar(gas1)
```

Comp 1	Comp 2	Comp 3	Comp 4	Comp 5	Comp 6	Comp 7	Comp 8	Comp 9	Comp 10
78.1708	7.4122	7.8242	2.6578	0.8768	0.9466	0.4922	0.4723	0.1688	0.1694

The loading plot (Figure 5) is much used for interpretation purposes, for instance to look for known spectral peaks or profiles:

```
> plot(gas1, "loadings", comps = 1:2, legendpos = "topleft",
+       labels = "numbers", xlab = "nm")
> abline(h = 0)
```

The `labels = "numbers"` argument makes the plot function try to interpret the variable names as numbers, and use them as x axis labels.

A fitted model is often used to predict the response values of new observations. The following predicts the responses for the ten observations in `gasTest`, using two components:

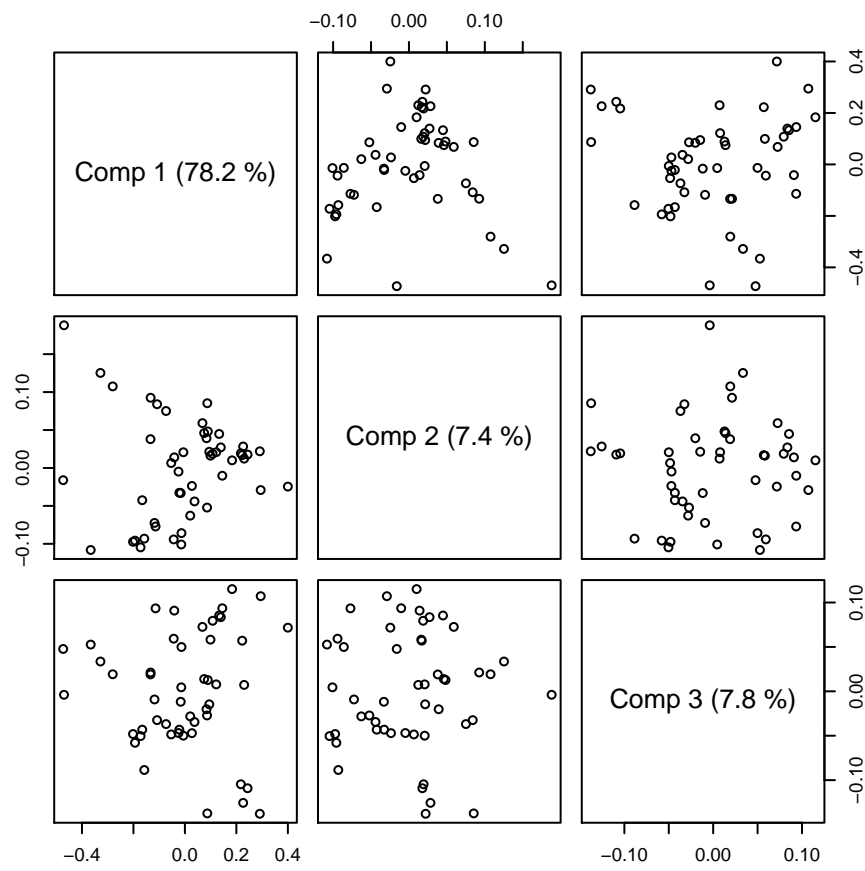


Figure 4: Score plot for the gasoline data

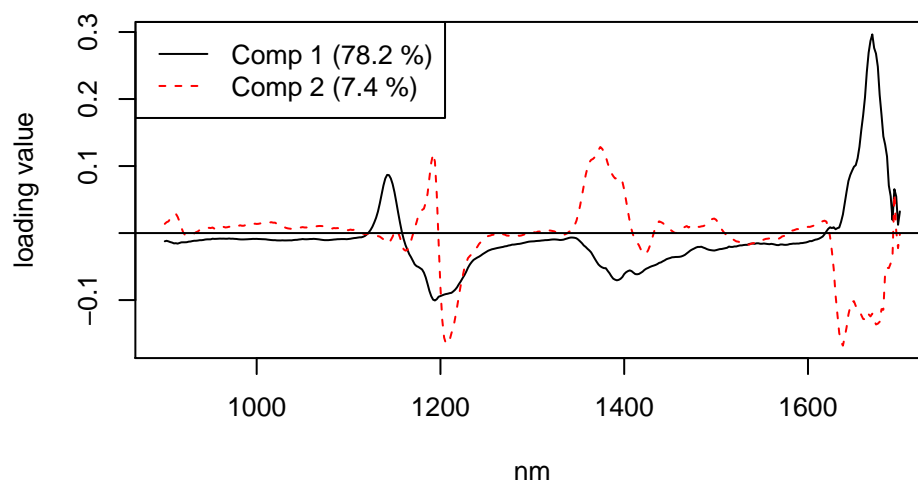


Figure 5: Loading plot for the gasoline data


```
> predict(gas1, ncomp = 2, newdata = gasTest)
```

```
, , 2 comps
```

```
      octane
51  87.94
52  87.25
53  88.16
54  84.97
55  85.15
56  84.51
57  87.56
58  86.85
59  89.19
60  87.09
```

Because we know the true response values for these samples, we can calculate the test set RMSEP:

```
> RMSEP(gas1, newdata = gasTest)
```

(Intercept)	1 comps	2 comps	3 comps	4 comps	5 comps
1.5369	1.1696	0.2445	0.2341	0.3287	0.2780
6 comps	7 comps	8 comps	9 comps	10 comps	
0.2703	0.3301	0.3571	0.4090	0.6116	

For two components, we get 0.244, which is quite close to the cross-validated estimate above (0.297).

4 Formulas and data frames

The **pls** package has a formula interface that works like the formula interface in R's standard **lm** functions, in most ways. This section gives a short description of formulas and data frames as they apply to **pls**. More information on formulas can be found in the **lm** help file, in Chapter 11 of 'An Introduction to R', and in Chapter 2 of 'The White Book' [1]. These are good reads for anyone wanting to understand how R works with formulas, and the user is strongly advised to read them.

4.1 Formulas

A *formula* consists of a *left hand side* (lhs), a tilde (\sim), and a *right hand side* (rhs). The lhs consists of a single term, representing the response(s). The rhs consists of one or more terms separated by $+$, representing the regressor(s). For instance, in the formula $a \sim b + c + d$, a is the response, and b , c , and d are the regressors. The intercept is handled automatically, and need not be specified in the formula.

Each term represents a matrix, a numeric vector or a factor (a factor should not be used as the response). If the response term is a matrix, a multi-response model is fit. In **pls**, the right hand side quite often consists of a single term, representing a matrix regressor: $y \sim X$.

It is also possible to specify transformations of the variables. For instance, `log(y) ~ msc(Z)` specifies a regression of the logarithm of `y` onto `Z` after `Z` has been transformed by *Multiplicative Scatter (or Signal) Correction* (MSC) [7], a pre-treatment that is very common in infrared spectroscopy. If the transformations contain symbols that are interpreted in the formula handling, e.g., `+`, `*` or `^`, the terms should be protected with the `I()` function, like this: `y ~ x1 + I(x2 + x3)`. This specifies *two* regressors: `x1`, and the sum of `x2` and `x3`.

4.2 Data frames

The fit functions first look for the specified variables in a supplied data frame, and it is advisable to collect all variables there. This makes it easier to know what data has been used for fitting, to keep different variants of the data around, and to predict new data.

To create a data frame, one can use the `data.frame` function: if `v1`, `v2` and `v3` are factors or numeric vectors, `mydata <- data.frame(y = v1, a = v2, b = v3)` will result in a data frame with variables named `y`, `a` and `b`.

PLSR and PCR are often used with a matrix as the single predictor term (especially when one is working with spectroscopic data). Also, multi-response models require a matrix as the response term. If `Z` is a matrix, it has to be protected by the ‘protect function’ `I()` in calls to `data.frame`: `mydata <- data.frame(..., Z = I(Z))`. Otherwise, it will be split into separate variables for each column, and there will be no variable called `Z` in the data frame, so we cannot use `Z` in the formula. One can also add the matrix to an existing data frame:

```
> mydata <- data.frame(...)
> mydata$Z <- Z
```

This will also prevent `Z` from being split into separate variables. Finally, one can use `cbind` to combine vectors and matrices into matrices on the fly in the formula. This is most useful for the response, e.g., `cbind(y1, y2) ~ X`.

Variables in a data frame can be accessed with the `$` operator, e.g., `mydata$y`. However, the `pls` functions access the variables automatically, so the user should never use `$` in formulas.

5 Fitting models

The main functions for fitting models are `pcr` and `plsr`. (They are simply wrappers for the function `mvr`, selecting the appropriate fit algorithm). We will use `plsr` in the examples in this section, but everything could have been done with `pcr` (or `mvr`).

In its simplest form, the function call for fitting models is `plsr(formula, ncomp, data)` (where `plsr` can be substituted with `pcr` or `mvr`). The argument `formula` is a formula as described above, `ncomp` is the number of components one wishes to fit, and `data` is the data frame containing the variables to use in the model. The function returns a fitted model (an object of class “`mvr`”) which can be inspected (Section 7) or used for predicting new observations (Section 8). For instance:

```
> dens1 <- plsr(density ~ NIR, ncomp = 5, data = yarn)
```

If the response term of the formula is a matrix, a multi-response model is fit, e.g.,

```
> dim(oliveoil$sensory)
```

```
[1] 16 6
```

```
> plsr(sensory ~ chemical, data = oliveoil)
```

Partial least squares regression , fitted with the kernel algorithm.

Call:

```
plsr(formula = sensory ~ chemical, data = oliveoil)
```

(As we see, the `print` method simply tells us what type of model this is, and how the fit function was called.)

The argument `ncomp` is optional. If it is missing, the maximal possible number of components are used. Also `data` is optional, and if it is missing, the variables specified in the formula is searched for in the global environment (the user's workspace). Usually, it is preferable to keep the variables in data frames, but it can sometimes be convenient to have them in the global environment. If the variables reside in a data frame, e.g. `yarn`, *do not* be tempted to use formulas like `yarn$density ~ yarn$NIR`! Use `density ~ NIR` and specify the data frame with `data = yarn` as above.

There are facilities for working interactively with models. To use only part of the samples in a data set, for instance the first 20, one can use arguments `subset = 1:20` or `data = yarn[1:20,]`. Also, if one wants to try different alternatives of the model, one can use the function `update`. For instance

```
> trainind <- which(yarn$train == TRUE)
> dens2 <- update(dens1, subset = trainind)
```

will refit the model `dens1` using only the observations which are marked as `TRUE` in `yarn$train`, and

```
> dens3 <- update(dens1, ncomp = 10)
```

will change the number of components to 10. Other arguments, such as `formula`, can also be changed with `update`. This can save a bit of typing when working interactively with models (but it doesn't save computing time; the model is refitted each time). In general, the reader is referred to 'The White Book' [1] or 'An Introduction to R' for more information about fitting and working with models in R.

Missing data can sometimes be a problem. The PLSR and PCR algorithms currently implemented in `ppls` do not handle missing values intrinsically, so observations with missing values must be removed. This can be done with the `na.action` argument. With `na.action = na.omit` (the default), any observation with missing values will be removed from the model completely. With `na.action = na.exclude`, they will be removed from the fitting process, but included as NAs in the residuals and fitted values. If you want an explicit error when there are missing values in the data, use `na.action = na.fail`. The default `na.action` can be set with `options()`, e.g., `options(na.action = quote(na.fail))`.

Standardisation and other pre-treatments of predictor variables are often called for. In `ppls`, the predictor variables are always centered, as a part of the fit algorithm. Scaling can be requested with the `scale` argument. If `scale` is `TRUE`, each variable is standardised by dividing it by its standard deviation, and if `scale` is a numeric vector, each variable is divided by the corresponding number. For instance, this will fit a model with standardised chemical measurements:

```
> olive1 <- plsr(sensory ~ chemical, scale = TRUE, data = oliveoil)
```

As mentioned earlier, MSC [7] is implemented in **pls** as a function **msc** that can be used in formulas:

```
> gas2 <- plsr(octane ~ msc(NIR), ncomp = 10, data = gasTrain)
```

This scatter corrects NIR prior to the fitting, and arranges for new spectra to be automatically scatter corrected (using the same reference spectrum as when fitting) in **predict**:

```
> predict(gas2, ncomp = 3, newdata = gasTest)
```

There are other arguments that can be given in the fit call: **validation** is for selecting validation, and **...** is for sending arguments to the underlying functions, notably the cross-validation function **mvrCv**. For the other arguments, see **?mvr**.

6 Choosing the number of components with cross-validation

Cross-validation, commonly used to determine the optimal number of components to take into account, is controlled by the **validation** argument in the modelling functions (**mvr**, **plsr** and **pcr**). The default value is **"none"**. Supplying a value of **"CV"** or **"LOO"** will cause the modelling procedure to call **mvrCv** to perform cross-validation; **"LOO"** provides leave-one-out cross-validation, whereas **"CV"** divides the data into segments. Default is to use ten segments, randomly selected, but also segments of consecutive objects or interleaved segments (sometimes also referred to as ‘Venetian blinds’) are possible through the use of the argument **segment.type**. One can also specify the segments explicitly with the argument **segments**; see **?mvrCv** for details.

When validation is performed in this way, the model will contain an element comprising information on the out-of-bag predictions (in the form of predicted values, as well as MSEP and R2 values). As a reference, the MSEP error using no components at all is calculated as well. The validation results can be visualised using the **plottype = "validation"** argument of the standard plotting function. An example is shown in Figure 2 for the gasoline data; typically, one would select a number of components after which the cross-validation error does not show a significant decrease. Unfortunately, no generally applicable tests are available, so the decision on how many components to retain will always be subjective to some extent.

When a pre-treatment that depends on the composition of the training set is applied, the cross-validation procedure as described above is not optimal, in the sense that the cross-validation errors are biased downward. As long as the only purpose is to select the optimal number of components, this bias may not be very important, but it is not too difficult to avoid it. The modelling functions have an argument **scale** that can be used for auto-scaling per segment. However, more elaborate methods such as MSC need explicit handling per segment. For this, the function **crossval** is available. It takes an **mvr** object and performs the cross-validation as it should be done: applying the pre-treatment for each segment. The results can be shown in a plot (which looks very similar to Figure 2) or summarised in numbers.

```
> gas2.cv <- crossval(gas2, segments = 10)
> plot(MSEP(gas2.cv), legendpos="topright")
> summary(gas2.cv, what = "validation")
```

```
Data:           X dimension: 50 401
           Y dimension: 50 1
Fit method: kernelpls
Number of components considered: 10
```

VALIDATION: RMSEP

Cross-validated using 10 random segments.

	(Intercept)	1 comps	2 comps	3 comps	4 comps	5 comps	6 comps
CV	1.545	1.336	0.2921	0.2657	0.2500	0.2482	0.2622
adjCV	1.545	1.332	0.2898	0.2640	0.2455	0.2442	0.2555

	7 comps	8 comps	9 comps	10 comps
CV	0.2548	0.2736	0.2905	0.3036
adjCV	0.2489	0.2662	0.2809	0.2925

Applying MSC in this case leads to nearly identical cross-validation estimates of prediction error.

When the scaling does not depend on the division of the data into segments (e.g., log-scaling), functions `crossval` and `mvrCv` give the same results; however, `crossval` is much slower.

Cross-validation can be computationally demanding (especially when using the function `crossval`). Therefore, both `mvrCv` and `crossval` can perform the calculations in parallel on a multi-core machine or on several machines. How to do this is described in Section 9.2.

7 Inspecting fitted models

A closer look at the fitted model may reveal interesting agreements or disagreements with what is known about the relations between X and Y. Several functions are implemented in `pls` for plotting, extracting and summarising model components.

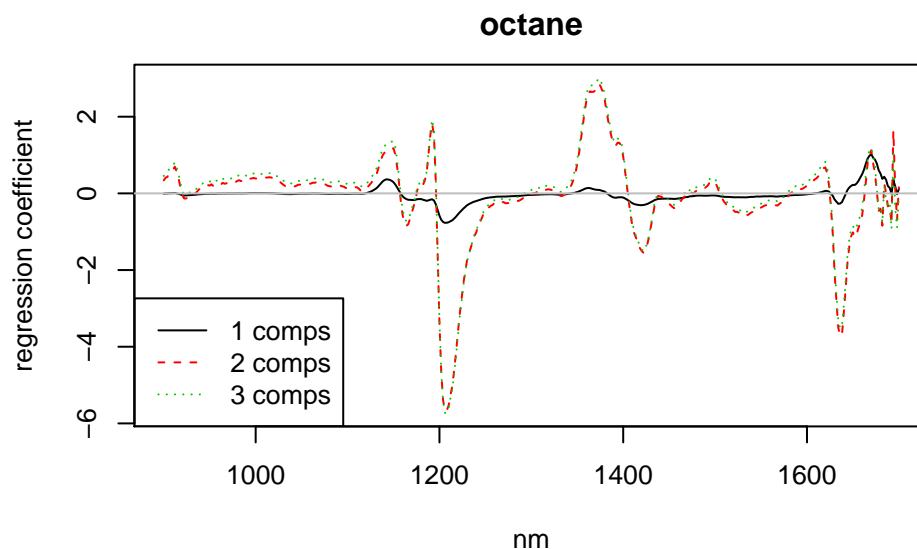
7.1 Plotting

One can access all plotting functions through the `"plottype"` argument of the `plot` method for `mvr` objects. This is simply a wrapper function calling the actual plot functions; the latter are available to the user as well.

The default plot is a prediction plot (`predplot`), showing predicted versus measured values. Test set predictions are used if a test set is supplied with the `newdata` argument. Otherwise, if the model was built using cross-validation, the cross-validated predictions are used, otherwise the predictions for the training set. This can be overridden with the `which` argument. An example of this type of plot can be seen in Figure 3. An optional argument can be used to indicate how many components should be included in the prediction.

To assess how many components are optimal, a validation plot (`validationplot`) can be used such as the one shown in Figure 2; this shows a measure of prediction performance (either RMSEP, MSEP, or R^2) against the number of components. Usually, one takes the first local minimum rather than the absolute minimum in the curve, to avoid over-fitting.

The regression coefficients can be visualised using `plottype = "coef"` in the `plot` method, or directly through function `coefplot`. This allows simultaneous plotting of the regression

Figure 6: Regression coefficients for the `gasoline` data

vectors for several different numbers of components at once. The regression vectors for the `gasoline` data set using MSC are shown in Figure 6 using the command

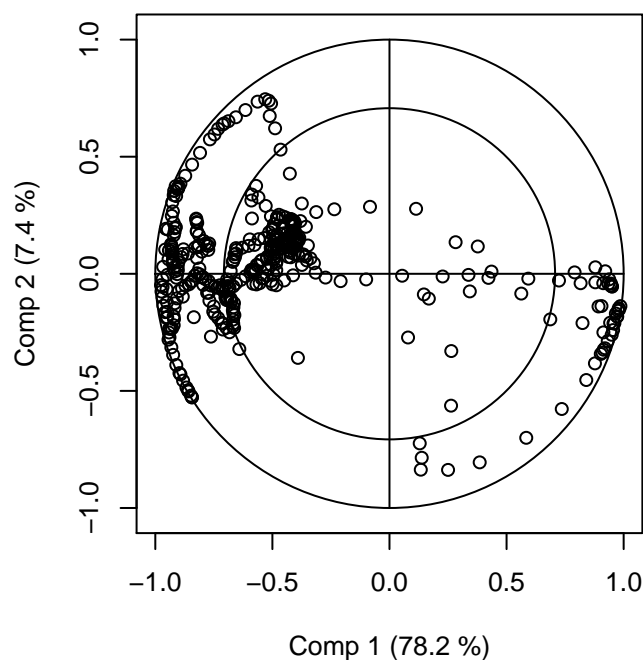
```
> plot(gas1, plottype = "coef", ncomp=1:3, legendpos = "bottomleft",
+      labels = "numbers", xlab = "nm")
```

Note that the coefficients for two components and three components are similar. This is because the third component contributes little to the predictions. The RMSEPs (see Figure 2) and predictions (see Section 8) for two and three components are quite similar.

Scores and loadings can be plotted using functions `scoreplot` (an example is shown in Figure 4) and `loadingplot` (in Figure 5), respectively. One can indicate the number of components with the `comps` argument; if more than two components are given, plotting the scores will give a pairs plot, otherwise a scatter plot. For `loadingplot`, the default is to use line plots.

Finally, a ‘correlation loadings’ plot (function `corrplot`, or `plottype = "correlation"` in `plot`) shows the correlations between each variable and the selected components (see Figure 7). These plots are scatter plots of two sets of scores with concentric circles of radii given by `radii`. Each point corresponds to an X variable. The squared distance between the point and the origin equals the fraction of the variance of the variable explained by the components in the panel. The default values for `radii` correspond to 50% and 100% explained variance, respectively.

The plot functions accept most of the ordinary plot parameters, such as `col` and `pch`. If the model has several responses or one selects more than one model size, e.g. `ncomp = 4:6`, in some plot functions (notably prediction plots (see below), validation plots and coefficient plots) the plot window will be divided and one plot will be shown for each combination of response and model size. The number of rows and columns are chosen automatically, but can be specified explicitly with arguments `nRows` and `nCols`. If there are more plots than fit the plot window, one will be asked to press return to see the rest of the plots.

Figure 7: Correlation loadings plot for the `gasoline` data

7.2 Extraction

Regression coefficients can be extracted using the generic function `coef`; the function takes several arguments, indicating the number of components to take into account, and whether the intercept is needed (default is `FALSE`).

Scores and loadings can be extracted using functions `scores` and `loadings` for X, and `Yscores` and `Yloadings` for Y. These also return the percentage of variance explained as attributes. In PLSR, weights can be extracted using the function `loading.weights`. When applied to a PCR model, the function returns `NULL`.

Note that commands like `plot(scores(gas1))` are perfectly correct, and lead to exactly the same plots as using `scoreplot`.

7.3 Summaries

The `print` method for an object of class `"mvr"` shows the regression type used, perhaps indicating the form of validation employed, and shows the function call. The `summary` method gives more information: it also shows the amount of variance explained by the model (for all choices of a , the number of latent variables). The `summary` method has an additional argument (`what`) to be able to focus on the training phase or validation phase, respectively. Default is to print both types of information.

8 Predicting new observations

Fitted models are often used to predict future observations, and `pls` implements a `predict` method for PLSR and PCR models. The most common way of calling this function is

`predict(mymod, ncomp = myncomp, newdata = mynewdata)`, where `mymod` is a fitted model, `myncomp` specifies the model size(s) to use, and `mynewdata` is a data frame with new X observations. The data frame can also contain response measurements for the new observations, which can be used to compare the predicted values to the measured ones, or to estimate the overall prediction ability of the model. If `newdata` is missing, `predict` uses the data used to fit the model, i.e., it returns fitted values.

If the argument `ncomp` is missing, `predict` returns predictions for models with 1 component, 2 components, ..., A components, where A is the number of components used when fitting the model. Otherwise, the model size(s) listed in `ncomp` are used. For instance, to get predictions from the model built in Section 3, with two and three components, one would use

```
> predict(gas1, ncomp = 2:3, newdata = gasTest[1:5,])
```

```
, , 2 comps
```

```
      octane
51  87.94
52  87.25
53  88.16
54  84.97
55  85.15
```

```
, , 3 comps
```

```
      octane
51  87.95
52  87.30
53  88.21
54  84.87
55  85.24
```

(We predict only the five first test observations, to save space.) The predictions with two and three components are quite similar. This could be expected, given that the regression vectors (Figure 6) as well as the estimated RMSEPs for the two model sizes were similar.

One can also specify explicitly which components to use when predicting. This is done by specifying the components in the argument `comps`. (If both `ncomp` and `comps` are specified, `comps` takes precedence over `ncomp`.) For instance, to get predictions from a model with only component 2, one can use

```
> predict(gas1, comps = 2, newdata = gasTest[1:5,])
```

```
      octane
51  87.53
52  86.30
53  87.35
54  85.82
55  85.32
```


The results are different from the predictions with two components (i.e., components one and two) above. (The intercept is always included in the predictions. It can be removed by subtracting `mymod$Ymeans` from the predicted values.)

The `predict` method returns a three-dimensional array, in which the entry (i, j, k) is the predicted value for observation i , response j and model size k . Note that singleton dimensions are not dropped, so predicting five observations for a uni-response model with `ncomp = 3` gives an $5 \times 1 \times 1$ array, not a vector of length five. This is to make it easier to distinguish between predictions from models with one response and predictions with one model size. (When using the `comps` argument, the last dimension is dropped, because the predictions are always from a single model.) One can drop the singleton dimensions explicitly by using `drop(predict(...))`:

```
> drop(predict(gas1, ncomp = 2:3, newdata = gasTest[1:5,]))
      2 comps 3 comps
51    87.94   87.95
52    87.25   87.30
53    88.16   88.21
54    84.97   84.87
55    85.15   85.24
```

Missing values in `newdata` are propagated to NAs in the predicted response, by default. This can be changed with the `na.action` argument. See `?na.omit` for details.

The `newdata` does not have to be a data frame. Recognising the fact that the right hand side of PLSR and PCR formulas very often are a single matrix term, the `predict` method allows one to use a matrix as `newdata`, so instead of

```
newdataframe <- data.frame(X = newmatrix)
predict(..., newdata = newdataframe)
```

one can simply say

```
predict(..., newdata = newmatrix)
```

However, there are a couple of caveats: First, this *only* works in `predict`. Other functions that take a `newdata` argument (such as `RMSEP`) must have a data frame (because they also need the response values). Second, when `newdata` is a data frame, `predict` is able to perform more tests on the supplied data, such as the dimensions and types of variables. Third, with the exception of scaling (specified with the `scale` argument when fitting the model), any transformations or coding of factors and interactions have to be performed manually if `newdata` is a matrix.

It is often interesting to predict scores from new observations, instead of response values. This can be done by specifying the argument `type = "scores"` in `predict`. One will then get a matrix with the scores corresponding to the components specified in `comps` (`ncomp` is accepted as a synonym for `comps` when predicting scores).

Predictions can be plotted with the function `predplot`. This function is generic, and can also be used for plotting predictions from other types of models, such as `lm`. Typically, `predplot` is called like this:

```
> predplot(gas1, ncomp = 2, newdata = gasTest, asp = 1, line = TRUE)
```

This plots predicted (with 2 components) versus measured response values. (Note that `newdata` must be a data frame with both X and Y variables.)

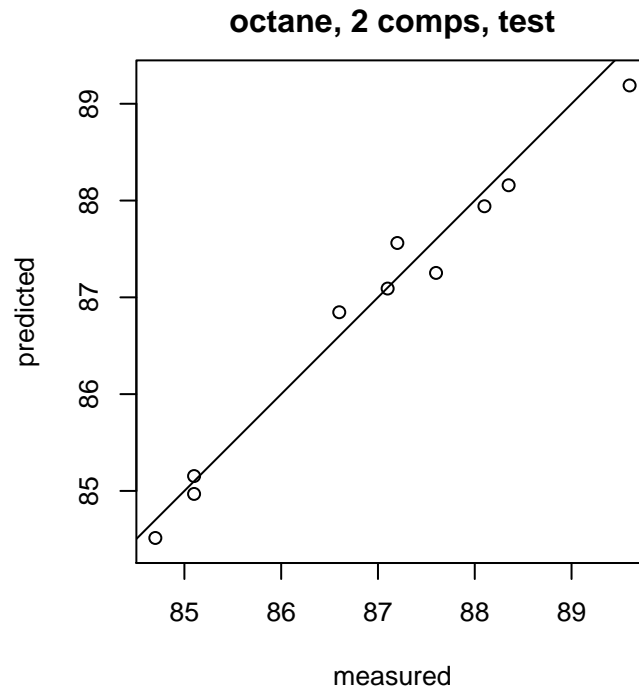


Figure 8: Test set predictions

9 Further topics

This section presents a couple of slightly technical topics for more advanced use of the package.

9.1 Selecting fit algorithms

There are several PLSR algorithms, and the **pls** package currently implements three of them: the kernel algorithm for tall matrices (many observations, few variables) [3], the classic orthogonal scores algorithm (A.K.A. NIPALS algorithm) [13] and the SIMPLS algorithm [4]. The kernel and orthogonal scores algorithms produce the same results (the kernel algorithm being the fastest of them for most problems). SIMPLS produces the same fit for single-response models, but slightly different results for multi-response models. It is also usually faster than the NIPALS algorithm.

The factory default is to use the kernel algorithm. One can specify a different algorithm with the `method` argument; i.e., `method = "oscorespls"`.

If one's personal taste of algorithms does not coincide with the defaults in **pls**, it can be quite tedious (and error prone) having to write e.g. `method = "oscorespls"` every time (even though it can be shortened to e.g. `me = "o"` due to partial matching). Therefore, the defaults can be changed, with the function `pls.options`. Called without arguments, it returns the current settings as a named list:

```
> pls.options()
```

```
$mvralg
```

```

[1] "kernelpls"

$plsralg
[1] "kernelpls"

$cpplsalg
[1] "cppls"

$pcralg
[1] "svdpc"

$parallel
NULL

$w.tol
[1] 2.22e-16

$X.tol
[1] 1e-12

```

The options specify the default fit algorithm of `mvr`, `pls`, and `pcr`. To return only a specific option, one can use `pls.options("plsralg")`. To change the default PLSR algorithm for the rest of the session, one can use, e.g.

```
> pls.options(plsralg = "oscorespls")
```

Note that changes to the options only last until R exits. (Earlier versions of **pls** stored the changes in the global environment so they could be saved and restored, but current CRAN policies do not allow this.)

9.2 Parallel cross-validation

Cross-validation is a computationally demanding procedure. A new model has to be fitted for each segment. The underlying fit functions have been optimised, and the implementation of cross-validation that is used when specifying the `validation` argument to `mvr` tries to avoid any unneeded calculations (and house-keeping things like the formula handling, which can be surprisingly expensive). Even so, cross-validation can take a long time, for models with large matrices, many components or many segments.

By default, the cross-validation calculations in **pls** is performed serially, on one CPU (core). (In the following, we will use ‘CPU’ to denote both CPUs and cores.)

Since version 2.14.0, R has shipped with a package **parallel** for running calculations in parallel, on multi-CPU machines or on several machines. The **pls** package can use the facilities of **parallel** to run the cross-validations in parallel.

The **parallel** package has several ways of running calculations in parallel, and not all of them are available on all systems. Therefore, the support in **pls** is quite general, so one can select the ways that work well on the given system.

To specify how to run calculations in parallel, one sets the option `parallel` in `pls.options`. After setting the option, one simply runs cross-validations as before, and the calculations

will be performed in parallel. This works both when using the `crossval` function and the `validation` argument to `mvr`. The parallel specification has effect until it is changed.

The default value for `parallel` is `NULL`, which specifies that the calculations are done serially, using one CPU. Specifying the value 1 has the same effect.

Specifying an integer > 1 makes the calculations use the function `mclapply` with the given number as the number of CPUs to use. Note: `mclapply` depends on ‘forking’ which does not exist on MS Windows, so `mclapply` cannot be used there.

Example:

```
pls.options(parallel = 4) # Use mclapply with 4 CPUs
gas1.cv <- plsr(octane ~ NIR, ncomp = 10, data = gasTrain, validation = "LOO")
```

The `parallel` option can also be specified as a cluster object created by the `makeCluster` function from the package `parallel`. Any following cross-validation will then be performed with the function `parLapply` on that cluster. Any valid cluster specification can be used. The user should stop the cluster with `stopCluster(pls.options()$parallel)` when it is no longer needed.

```
library(parallel) # Needed for the makeCluster call
pls.options(parallel = makeCluster(4, type = "PSOCK")) # PSOCK cluster, 4 CPUs
gas1.cv <- plsr(octane ~ NIR, ncomp = 10, data = gasTrain, validation = "LOO")
## later:
stopCluster(pls.options()$parallel)
```

Several types of clusters are available: `FORK` uses forking, so starting the cluster is very quick, however it is not available on MS Windows. `PSOCK` starts R processes with the `Rscript` command, which is slower, but is supported on MS Windows. It can also start worker processes on different machines (see `?makeCluster` for how). `MPI` uses `MPI` to start and communicate with processes. This is the most flexible, but is often slower to start up than the other types. It also depends on the packages `snow` and `Rmpi` to be installed and working. It is especially useful when running batch jobs on a computing cluster, because `MPI` can interact with the queue system on the cluster to find out which machines to use when the job starts.

Here is an example of running a batch job on a cluster using `MPI`:

R script (myscript.R):

```
library(parallel) # for the makeCluster call
pls.options(parallel = makeCluster(16, type = "MPI")) # MPI cluster, 16 CPUs
gas1.cv <- plsr(octane ~ NIR, ncomp = 10, data = gasTrain, validation = "LOO")
## later:
save.image(file = "results.RData")
stopCluster(pls.options()$parallel)
mpi.exit() # stop Rmpi
```

To run the job:

```
mpirun -np 1 R --slave --file=myscript.R
```

The details of how to run `mpirun` varies between the different `MPI` implementations and how they interact with the queue system used (if any). The above should work for `OpenMPI` or `Intel MPI` running under the `Slurm` queue system. In other situations, one might have to specify which machines to use with, e.g., the `-host` or `-machinefile` switch.

9.3 Package design

The **pls** package is designed such that an interface function **mvr** handles the formula and data, and calls an underlying fit function (and possibly a cross-validation function) to do the real work. There are several reasons for this design: it makes it easier to implement new algorithms, one can easily skip the time-consuming formula and data handling in computing-intensive applications (simulations, etc.), and it makes it easier to use the **pls** package as a building block in other packages.

The plotting facilities are implemented similarly: the **plot** method simply calls the correct plot function based on the **plottype** argument. Here, however, the separate plot functions are meant to be callable interactively, because some people like to use the generic **plot** function, while others like to use separate functions for each plot type. There are also **plot** methods for some of the components of fitted models that can be extracted with **extract** functions, like score and loading matrices. Thus there are several ways to get some plots, e.g.:

```
plot(mymod, plottype = "scores", ...)
scoreplot(mymod, ...)
plot(scores(mymod), ...)
```

One example of a package that uses **pls** is **lspls**, available on CRAN. In that package LS is combined with PLS in a regression procedure. It calls the fit functions of **pls** directly, and also uses the plot functions to construct score and loading plots. There is also the **plsgenomics** package, which includes a modified version of (an earlier version of) the SIMPLS fit function **simpls.fit**.

9.4 Calling fit functions directly

The underlying fit functions are called **kernelpls.fit**, **oscorespls.fit**, and **simpls.fit** for the PLSR methods, and **svdpc.fit** for the PCR method. They all take arguments **X**, **Y**, **ncomp**, and **stripped**. Arguments **X**, **Y**, and **ncomp** specify **X** and **Y** (as matrices, not data frames), and the number of components to fit, respectively. The argument **stripped** defaults to **FALSE**. When it is **TRUE**, the calculations are stripped down to the bare minimum required for returning the **X** means, **Y** means, and the regression coefficients. This is used to speed up cross-validation procedures.

The fit functions can be called directly, for instance when one wants to avoid the overhead of formula and data handling in repeated fits. As an example, this is how a simple leave-one-out cross-validation for a uni-response-model could be implemented, using the SIMPLS:

```
> X <- gasTrain$NIR
> Y <- gasTrain$octane
> ncomp <- 5
> cvPreds <- matrix(nrow = nrow(X), ncol = ncomp)
> for (i in 1:nrow(X)) {
+   fit <- simpls.fit(X[-i,], Y[-i], ncomp = ncomp, stripped = TRUE)
+   cvPreds[i,] <- (X[i,] - fit$Xmeans) %*% drop(fit$coefficients) +
+     fit$Ymeans
+ }
```

The RMSEP of the cross-validated predictions are

```
> sqrt(colMeans((cvPreds - Y)^2))
```

```
[1] 1.3570 0.2966 0.2524 0.2476 0.2398
```

which can be seen to be the same as the (unadjusted) CV results for the `gas1` model in Section 3.

9.5 Formula handling in more detail

The handling of formulas and variables in the model fitting is very similar to what happens in the function `lm`: The variables specified in the formula are looked up in the data frame given in the `data` argument of the fit function (`pls`, `pcr` or `mvr`), or in the calling environment if not found in the data frame. Factors are coded into one or more of columns, depending on the number of levels, and on the contrasts option. All (possibly coded) variables are then collected in a numerical model matrix. This matrix is then handed to the underlying fit or cross-validation functions. A similar handling is used in the `predict` method.

The intercept is treated specially in `pls`. After the model matrix has been constructed, the intercept column is removed. This ensures that any factors are coded as if the intercept was present. The underlying fit functions then center the rest of the variables as a part of the fitting process. (This is intrinsic to the PLSR and PCR algorithms.) The intercept is handled separately. A consequence of this is that explicitly specifying formulas without the intercept (e.g., $y \sim a + b - 1$) will only result in the coding of any factors to change; the intercept will still be fitted.

References

- [1] John M. Chambers and Trevor J. Hastie. *Statistical Models in S*. Chapman & Hall, London, 1992.
- [2] R.D. Cramer, D.E. Patterson, and J.D. Bunce. Comparative molecular-field analysis (ComFA). 1. Effect of shape on binding of steroids to carrier proteins. *Journal of the American Chemical Society*, 110(18):5959–5967, 1988.
- [3] B. S. Dayal and J. F. MacGregor. Improved pls algorithms. *Journal of Chemometrics*, 11(1):73–85, 1997.
- [4] Sijmen de Jong. SIMPLS: an alternative approach to partial least squares regression. *Chemometrics and Intelligent Laboratory Systems*, 18:251–263, 1993.
- [5] C. Fornell and F.L. Bookstein. Two structural equation models – LISREL and PLS applied to consumer-exit voice theory. *Journal of Marketing Research*, 19(4):440–452, 1982.
- [6] Ildiko E. Frank and Jerome H. Friedman. A statistical view of some chemometrics regression tools (with discussion). *Technometrics*, 35(2):109–148, 1993.
- [7] P. Geladi, D. MacDougall, and H. Martens. Linearization and scatter-correction for NIR reflectance spectra of meat. *Applied Spectroscopy*, 39:491–500, 1985.

- [8] T. Hastie, R. Tibshirani, and J.H. Friedman, editors. *The Elements of Statistical Learning*. Springer, New York, 2001.
- [9] John H. Kalivas. Two data sets of near infrared spectra. *Chemometrics and Intelligent Laboratory Systems*, 37:255–259, 1997.
- [10] J.V. Kresta, J.F. MacGregor, and T.E. Marlin. Multivariate statistical monitoring of process operating performance. *The Canadian Journal of Chemical Engineering*, 69(1):35–47, 1991.
- [11] Peter A. Lachenbruch and M. Ray Mickey. Estimation of error rates in discriminant analysis. *Technometrics*, 10(1):1–11, 1968.
- [12] H. Martens. Reliable and relevant modelling of real world data: a personal account of the development of PLS regression. *Chemometrics and Intelligent Laboratory Systems*, 58(2):85–95, 2001.
- [13] Harald Martens and Tormod Næs. *Multivariate Calibration*. Wiley, Chichester, 1989.
- [14] D. L. Massart, B. G. M. Vandeginste, L. M. C. Buydens, S. de Jong, P. J. Lewi, and J. Smeyers-Verbeke. *Handbook of Chemometrics and Qualimetrics: Part B*. Elsevier, 1998.
- [15] A.R. McIntosh, F.L. Bookstein, J.V. Haxby, and C.L. Grady. Spatial pattern analysis of functional brain images using partial least squares. *Neuroimage*, 3(3):143–157, 1996.
- [16] Bjørn-Helge Mevik and Henrik René Cederkvist. Mean squared error of prediction (MSEP) estimates for principal component regression (PCR) and partial least squares regression (PLSR). *Journal of Chemometrics*, 18(9):422–429, 2004.
- [17] Bjørn-Helge Mevik and Ron Wehrens. The pls package: Principal component and partial least squares regression in R. *Journal of Statistical Software*, 18(2):1–24, 2007.
- [18] D.V. Nguyen and D.M. Rocke. Tumor classification by partial least squares using microarray gene expression data. *Bioinformatics*, 18(1):39–50, 2002.
- [19] H. Swierenga, A. P. de Weijer, R. J. van Wijk, and L. M. C. Buydens. Strategy for constructing robust multivariate calibration models. *Chemometrics and Intelligent Laboratory Systems*, 49(1):1–17, 1999.
- [20] S. Wold. Personal memories of the early PLS development. *Chemometrics and Intelligent Laboratory Systems*, 58(2):83–84, 2001.