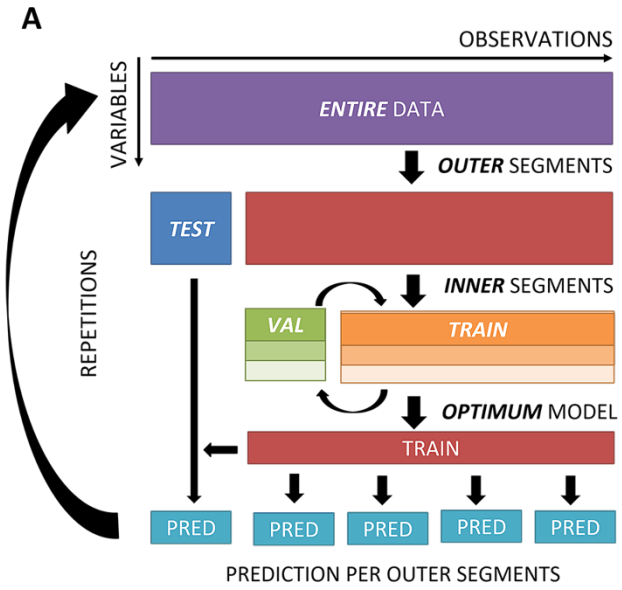
A Brief Tutorial on MUVR:
Multivariate methods with Unbiased Variable selection in R

Lin Shi and Carl Brunius, Gothenburg, 19 April 2018

What is MUVR?

The MUVR package is an algorithm for multivariate modelling, aimed at finding associations between predictor data (an X matrix) and a response (a Y vector; continuous for *regression* or factor for *classification*). MUVR is particularly useful to cope with data that has large numbers of variables and few observations, and to construct robust, parsimonious multivariate models that generalize well, minimize overfitting and facilitate interpretation of results (Shi et al 2018).

From a technical perspective, MUVR is a statistical validation framework, incorporating a recursive variable selection procedure within a repeated double cross validation (rdCV) scheme. MUVR allows for partial least squares (PLS) and random forest (RF) core modelling, selects both minimal-optimal variables (useful e.g. for predictive biomarker discovery) and all-relevant variables (e.g. for biological interpretation and mechanistic investigation) with minimal variable selection bias. MUVR supports several different data analytical problems/data types: regression, classification and multilevel (i.e. data with sample dependency, e.g. before/after or cross-over interventions).



B

For all *nRep* repetitions {
  Randomly sample the ***ENTIRE*** data set into *nOuter* segments
  for all *nOuter* segments {
    Hold out a ***TEST*** set from ***ENTIRE*** data
    Define the ***INNER*** set as the rest of the ***ENTIRE*** data
    Repeat until number of remaining variables = 2 {
      Randomly sample the ***INNER*** data into *nInner* segments
      for all *nInner* segments {
        Hold out a validation ***VAL*** set from the ***INNER*** data
        Define the training ***TRAIN*** set as the rest of the *INNER* data
        Train core method (i.e. PLS or RF) on ***TRAIN*** data
        Tune core method parameter settings using ***VAL*** data
      }
      Average and rank variable importance over the *nInner* segments
      Perform backwards elimination of proportion lowest ranking variables
    }
    Determine optimum parameters and variables from validation performance
    Train consensus model on ***INNER*** data
    Predict the ***TEST*** segment
  }
  Average parameter settings over *nOuter* segments
  Average and rank variable importance over *nOuter* segments
}
Average parameter setting over *nRep* repetitions
Average and rank variable importance over *nRep* repetitions

Working principle of MUVR. A: Graphical representation of the MUVR algorithm. The original data is randomly subdivided into **OUTER** segments. For each outer segment, the remaining (**INNER**) data is used for training and tuning of model parameters, including recursive ranking and backwards elimination of variables. Each outer segment is then predicted using an optimized consensus model trained on all inner observations, ensuring that the holdout test set was never used for training or tuning modelling parameters. The procedure is then repeated for improved modelling performance. B: Pseudocode of the MUVR algorithm.

Installation

We assume that R has been downloaded and installed on your computer (https://www.r-project.org/). Furthermore, for practical data analytical work we recommend to download, install and work in RStudio (https://www.rstudio.com/) or another IDE of your choice, which has several advantages over working in "simple" command line R. There are several online resources for learning to work efficiently with R and RStudio (e.g. https://www.rstudio.com/online-learning/) and you can use any search engine to find good, freely available material. In this tutorial, R code is shown in `red monotype font`.

1. Install the release version of 'devtools' from CRAN:
   ```
   install.packages("devtools")
   ```
2. Make sure that a working development environment has been properly installed.
   - **Windows**: Install Rtools (https://cran.r-project.org/bin/windows/Rtools/).
   - **Mac**: Install Xcode from the Mac App Store.
   - **Linux**: install the R development package, usually called 'r-devel' or 'r-base-dev'.
3. Install MUVR from Gitlab.
   ```
   library(devtools)
   install_git("https://gitlab.com/CarlBrunius/MUVR.git")
   ```
4. To reduce computation time, MUVR uses the 'doParallel' package for parallel processing:
   ```
   install.packages("doParallel", repos="http://R-Forge.R-project.org")
   ```

Data

The MUVR algorithm uses predictor (X) and response (Y) data which are matched by observation, i.e. each position in the Y response is matched with the corresponding row in the X matrix. The X matrix thus has observations in the rows and variables in the columns. The X columns need to have unique names (variable identifiers). This can be checked with `colnames(X)`.

Some data types are not readily modelled by PLS without some preprocessing, such as microbiota data. One approach is to perform an offset of all zero values and then perform a log-transformation of the data. For convenience, MUVR provides the function `preProcess()` which performs 5 preprocessing tasks: offset (of all data); zero offset (of zero values in the data), transformation (log, sqrt or none), centering (mean, none or custom by variable) and scaling (unit variance, pareto, none or custom by variable).

The effect of predictor variable scaling in MUVR will depend on the choice of core algorithm: Random forest is a scale invariant technique and therefor insensitive to transformations such as log or sqrt, to centering and to scaling. PLS is, on the other hand, very sensitive to both transformations and scaling. The default in MUVR-PLS is to internally mean center and scale data to unit variance in all underlying submodels (scale=TRUE). If other scaling options are wanted, the user should disable automatic internal scaling (using the MUVR parameter scale=FALSE) and pre-perform a desired preprocessing technique, e.g. using the `preProcess()` function.

Classification analysis

We will walk through a MUVR classification analysis using Random Forest core modelling. However, classification using PLS is easily achieved by changing the 'method' parameter (see code below). We will use the "mosquito" dataset, which has data on microbiota composition data (16S rRNA OTU data) from 29 Anopheles gambiae mosquitoes sampled from 3 different villages in western Burkina Faso (Buck et al 2016). Typing `data("mosquito")` will load 2 objects: A categorical 'Yotu' response variable (three levels, i.e. villages) for 29 samples and; A numeric 'Xotu' matrix, consisting of 1678 16S rRNA operational taxonomic units (OTU) measured for the 29 samples. Note, that the MUVR algorithm performs resampling of the data in each repetition leading to slightly different results each time an analysis is run, wherefore results may differ slightly from those reported here. First, we set up libraries, data and parameters for multivariate modelling:

```
#####################################################
# Classification example using the "mosquito" data

# Call in relevant libraries
library(doParallel)     # Parallel processing
library(MUVR)           # Multivariate modelling

# Call in the "freelive" data from the MUVR package
data("mosquito")

# Check number of observations per class
table(Yotu)             # As a general principle, nOuter ⩽ n of the smallest class (i.e. 8)

# Set method parameters
nCore=detectCores()-1   # Number of processor threads to use
nRep=nCore              # Number of MUVR repetitions
nOuter=8                # Number of outer cross-validation segments
varRatio=0.8            # Proportion of variables kept per iteration
method='RF'             # Selected core modelling algorithm
```

It is often practical to separate between nCore (i.e. number of computer cores to use for processing) and nRep (number of repetitions in the MUVR algorithm). `nCore=detectCores()-1` uses all but one thread (kept for everyday computer usage), which makes for efficient processor use. `nCore=detectCores()` will perform slightly faster calculations, but leave you with practically no possibility to use your computer for other tasks during calculations. nRep is usually set to a multiplier of nCore for effective processor usage. For initial "quick'n'dirty" modelling, we normally set nRep=nCore. For final processing we often set nRep between 20 and 50 and check modelling convergence using the `plotStability()` function (see below). We normally set the number of outer cross-validation segments between 6-8, with higher number of segments when there are fewer observations – to increase the number of observations in the model training. A general recommendation is also to ensure that all classes are present in all segments (by ensuring that nOuter is not larger than the smallest class size within the response variable). The variable ratio (varRatio) parameter governs the proportion of variables kept for iteration of the recursive variable elimination in the inner loop. We normally start out low (varRatio=0.75) and increase towards 0.85-0.9 for

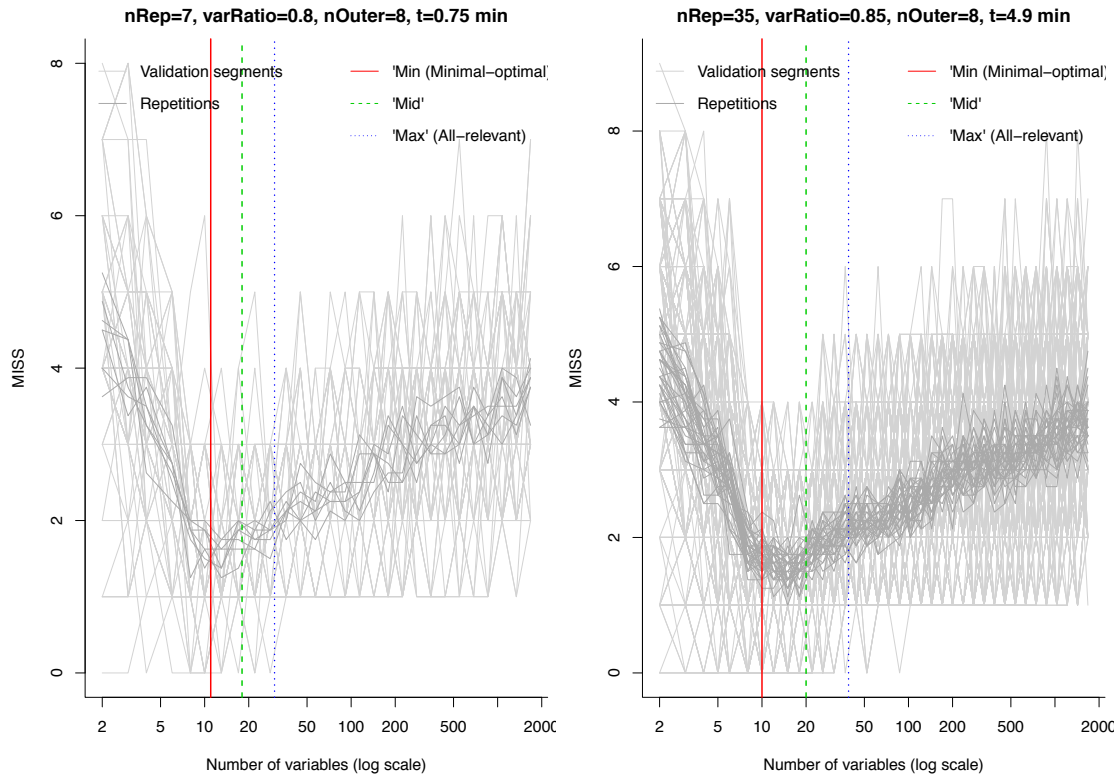
final processing. After setting parameters, it is time to initialise parallel processing and perform the actual modelling:

```
# Set up parallel processing using doParallel
cl=makeCluster(nCore)
registerDoParallel(cl)

# Perform modelling
classModel = MUVR(X=Xotu, Y=Yotu, nRep=nRep, nOuter=nOuter, varRatio=varRatio, method=method)

# Stop parallel processing
stopCluster(cl)
```

Depending on your parameter settings and size of data, the modelling may take minutes or even hours. “Quick’n’dirty” modelling is thus encouraged to get a feeling for the modelling potential before final processing. The above code requires approximately 0.75 mins using 7 threads on a Mac Powerbook Pro mid-2015 with 2,8 GHz Intel Core i7. Now, let’s look at some output:

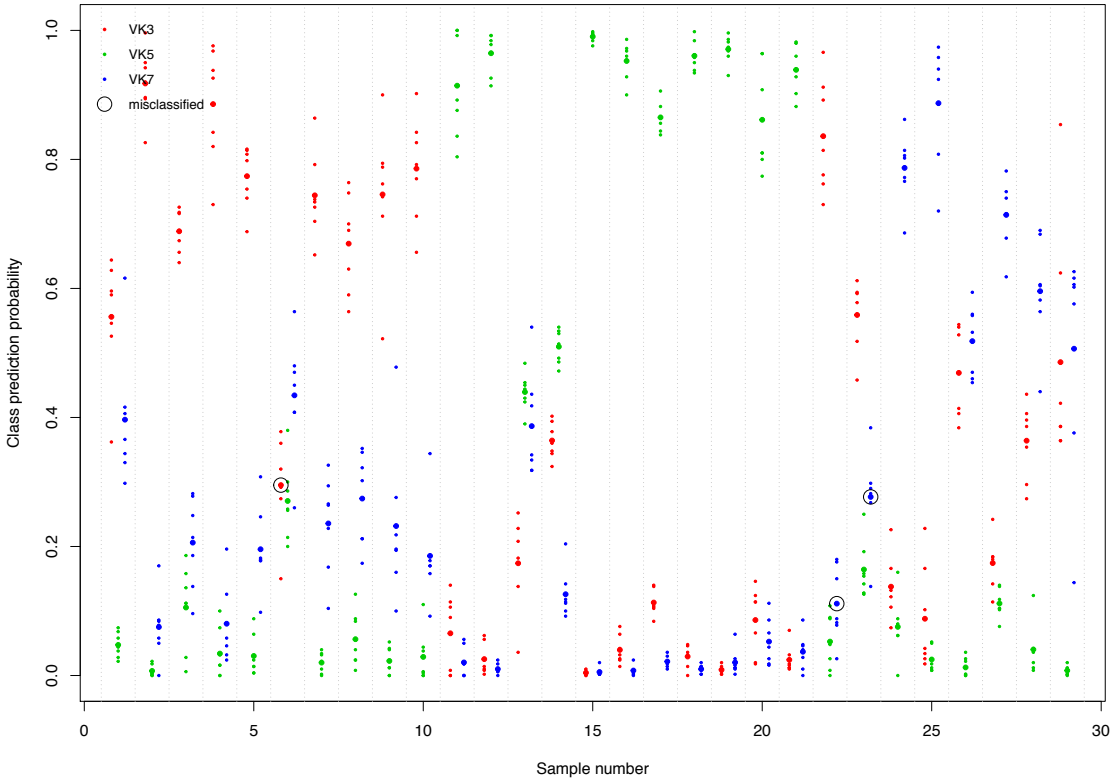
```
# Examine model performance and output

classModel$miss                  # Number of misclassifications for min, mid and max models
# min mid max
#  5   3   4

classModel$nVar                  # Number of variables for min, mid and max models
# min mid max
#  10  18  33

cbind(Yotu, classModel$yClass)   # Actual class side-by-side with min, mid and max predictions
#       Yotu min mid max
# 1_ID1  VK3 VK3 VK3 VK3
# 2_ID2  VK3 VK3 VK3 VK3
# 3_ID3  VK3 VK3 VK3 VK3
# 4_ID4  VK3 VK3 VK3 VK3
# …  …    …   …   …   …
```

```
plotVAL(classModel)
```



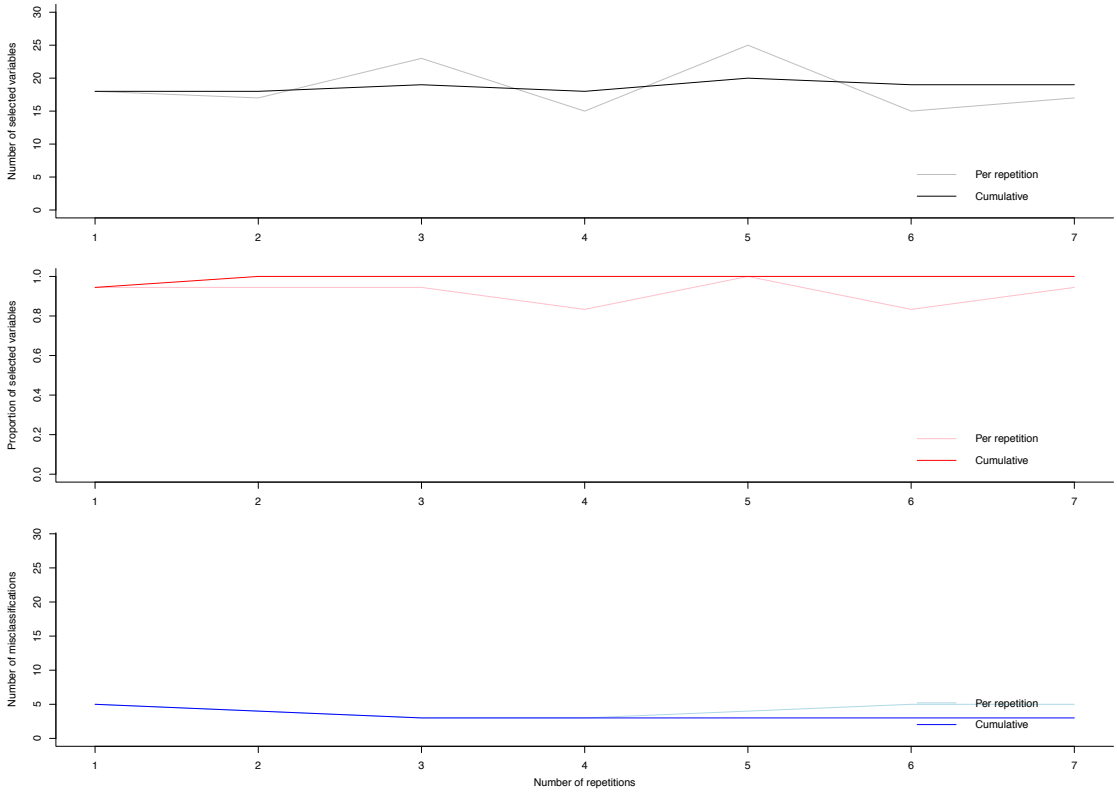
Light grey lines represent validation performance for the individual inner segments, whereas the darker grey lines represent inner segment validation curves averaged over the repetitions. The “Quick’n’dirty” model with few repetitions on the left and the final model on the right show similar validation trends, although with improved resolution in the final model. Minimal-optimal (‘Min’) and all-relevant (‘Max’) models represent the outer borders of variable selections where the validation performance (in this case, number of misclassifications) is minimal. This is in practice determined as having validation performance within 5% slack allowance from the actual minimum. The minimal-optimal model thus represents the minimal variable set required for optimal method performance, i.e. with the strongest predictors e.g. suitable for biomarker discovery. The all-relevant model instead represents the data set with all variables with relevant signal-to-noise in relation to the research question: i.e. the strongest predictors and, additionally, variables with redundant but not erroneous information. The ‘Mid’ model represents a trade-off between the ‘Min’ and ‘Max’ model and is found at the geometric mean.

```
plotMV(classModel, model='mid')         # Look at the model of choice: min, mid or max
```



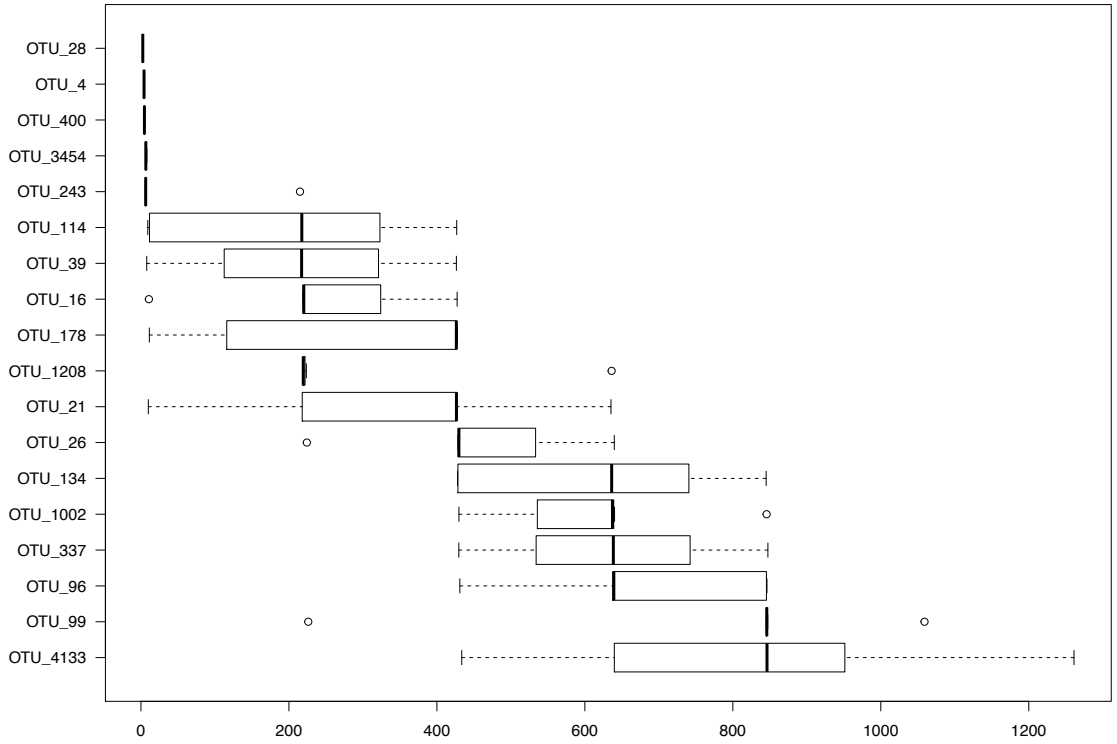
For classification analysis, the plotMV() function generates a “swim lane” plot, where each swim lane shows individual and overall predictions for each observation. Classes are color coded and jittered by class. The smaller dots represent predictions from individual repetitions and the larger dots represent class prediction probability averaged over all repetitions. Misclassified predictions are circled. The spread in the individual predictions gives an intuitive graphical overview of prediction precision. Using the model argument, the data analyst can easily switch between ‘Min’, ‘Mid’ and ‘Max’ models.

```
plotStability(classModel, model='mid')
```



The stability plot for classification analysis generates three subplots: (i) Number of selected variables for each repetition as well as cumulative average over the repetitions; (ii) The proportion of selected variables reports the ratio of the final variable selection found in each repetition and cumulatively, averaged over the number of repetitions; (iii) Number of misclassifications per repetition and cumulatively. For all subplots, there may be some variability between individual repetitions due to the random sampling of observations into the cross-validation segments. However, cumulative averages converge rapidly and we normally observe convergence within 20-50 observations or even faster, depending of the strength of the analysis.

```
plotVIP(classModel, model='mid')
```



The plotMV() function generates a boxplot of the variables automatically selected from optimal modelling performance. This output generates an intuitive overview of which variables are reproducibly selected with low rank (lower is better), which may thus be the strongest predictors of the response variable. The selected variables can also be obtained from:

```
getVIP(classModel, model='mid')   # Extract most informative variables: Lower rank is better
#          order     name       rank
# OTU_28       1   OTU_28   2.556122
# OTU_4        2    OTU_4   4.188776
# OTU_400      3  OTU_400   4.635204
# OTU_3454     4 OTU_3454   6.897959
#  …  …        …     …      …
```

Regression analysis with repeated samples

The general outline of a regression analysis follows the same approach as the classification analysis described above, including considerations for parameter settings, albeit with the difference that the Y response vector is numeric rather than consisting of factor levels. In this regression example, we will use the “freelive” dataset, which has data on dietary exposure to whole grain rye and urine metabolomics data from 58 individuals sampled on 2 occasions to identify biomarkers of whole grain rye exposure (Hanhineva et al 2015). Typing `data("freelive")` will load 3 objects: A continuous ‘YR’ response vector with wholegrain rye consumption for 112 samples (some individuals did not provide repeated samples); A

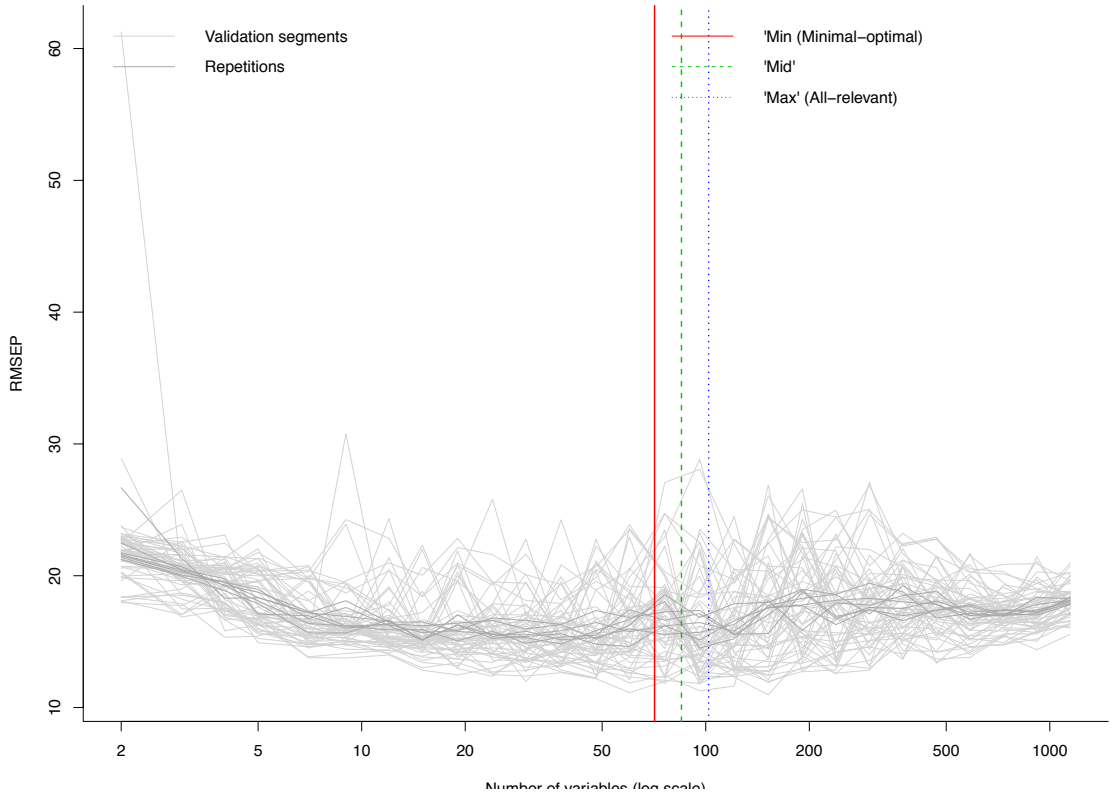
numeric ‘XRVIP’ matrix, consisting of 1147 metabolomics features for the 112 samples; An ‘IDR’ vector with identifiers of the individuals (numerical IDR).

As mentioned, there are 2 observations for most individuals in this data set. Standard procedures for cross-validation will not take the experimental unit into account during division of data into cross-validation segments (folds). Consequently, there is high likelihood of overfitting to the data by having observations from the same individual present in both model training, validation and/or testing segments. To reduce overfitting in this example, identifiers are added to the MUVR algorithm using the ID argument to ensure that samples from the same individual are always kept together in the cross-validation segments. However, if each observation in your experiment corresponds to one unique experimental unit, you may ignore the ID argument.

```
#####################################################
# Regression example using the "freelive" data

# Call in relevant libraries
library(doParallel)     # Parallel processing
library(MUVR)           # Multivariate modelling

# Call in the "freelive" data from the MUVR package
data("freelive")

# Set method parameters
nCore=detectCores()-1   # Number of processor threads to use
nRep=nCore              # Number of MUVR repetitions
nOuter=8                # Number of outer cross-validation segments
varRatio=0.8            # Proportion of variables kept per iteration
method='PLS'            # Selected core modelling algorithm

# Set up parallel processing
cl=makeCluster(nCore)
registerDoParallel(cl)

# Perform modelling
regrModel = MUVR(X=XRVIP, Y=YR, ID=IDR, nRep=nRep, nOuter=nOuter, varRatio=varRatio, method=method)
# 1.4 mins using 7 threads on a Mac Powerbook Pro mid-2015 with 2,8 GHz Intel Core i7.

# Stop parallel processing
stopCluster(cl)

# Examine model performance and output

regrModel$fitMetric             # Look at fitness metrics for min, mid and max models
# $R2
# [1] 0.8980212 0.8889288 0.8923386
# $Q2
# [1] 0.5916388 0.6125862 0.6231076

regrModel$nComp                 # Number of components for min, mid and max models
# min mid max
#   4   4   4

regrModel$nVar                  # Number of variables for min, mid and max models
# min mid max
#  71  85 102

cbind(YR, regrModel$yPred)      # Actual exposures side-by-side with min, mid and max predictions
                 YR        min        mid        max
```

```
# 1_ID1     11.0666667  12.070264  11.172260  11.372327
# 2_ID100   12.1000000  11.139881  11.744029   9.930332
# 3_ID101   57.7333333  54.948522  57.546783  61.453082
# …          …           …          …          …
```

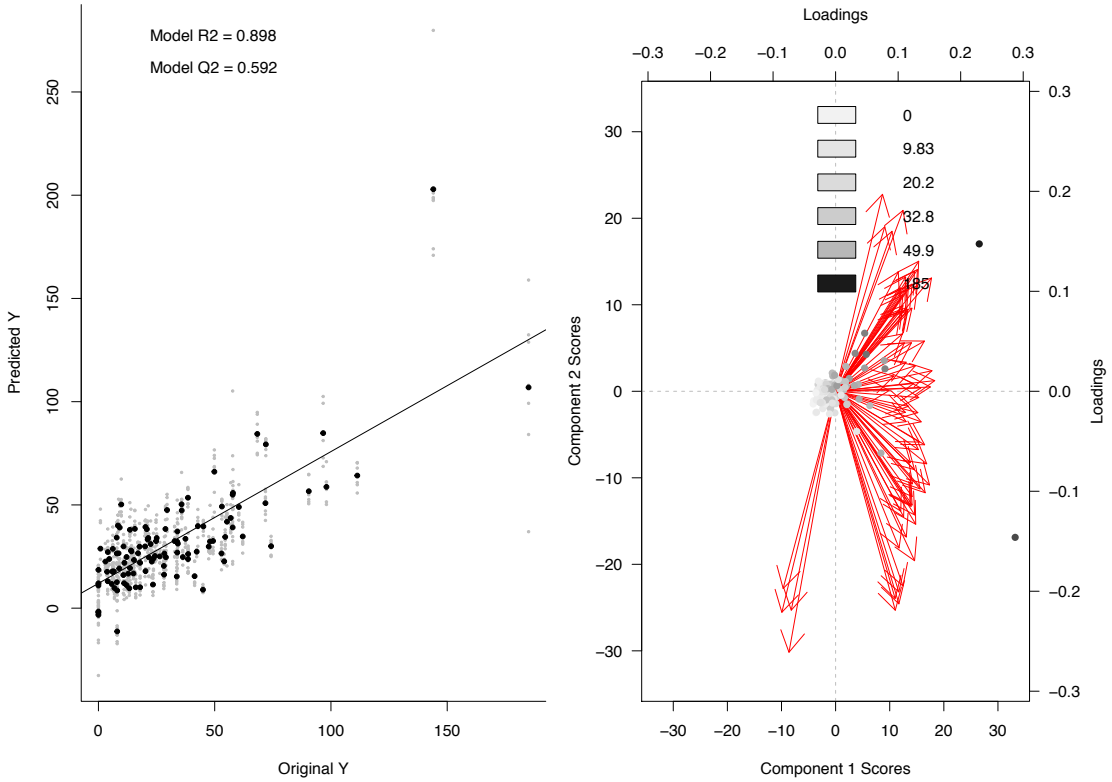
```
plotVAL(regrModel)
```



The validation plot shows a similar behaviour as for classification, albeit with a different fitness metric (i.e. root mean squared error of prediction (RMSEP) instead of number of misclassifications).

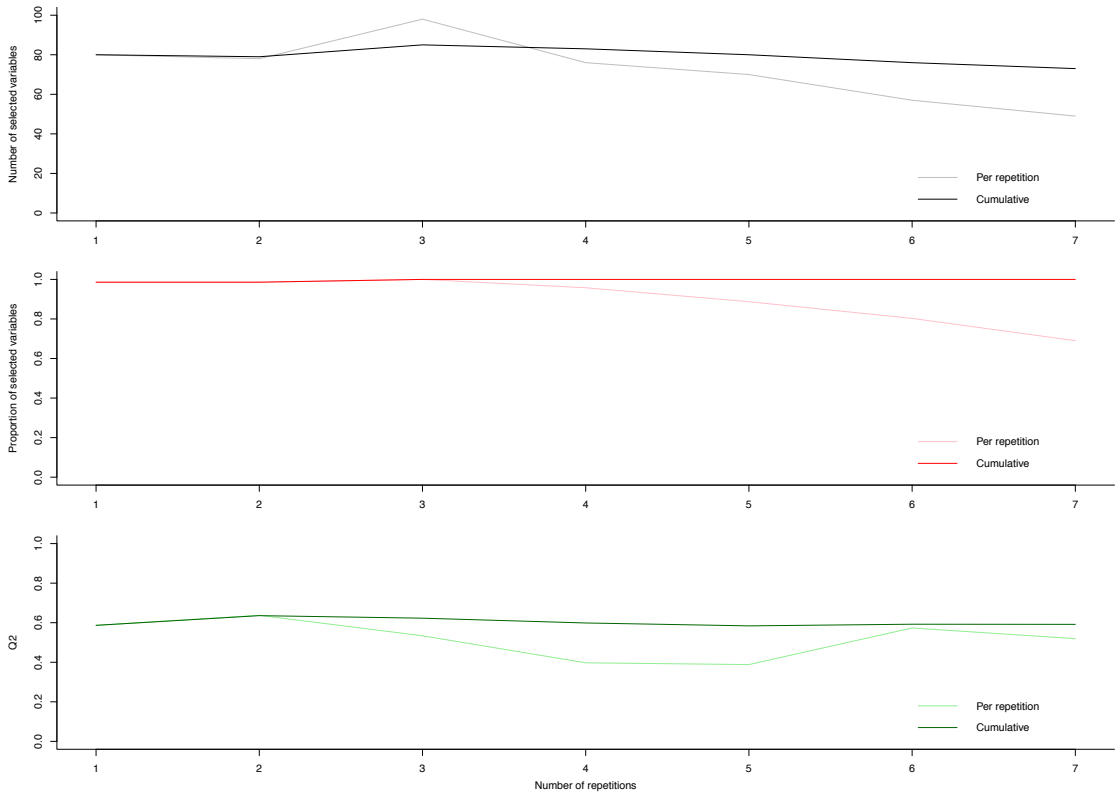
```
plotMV(regrModel, model='min')        # Look at the model of choice: min, mid or max
PLSfit=regrModel$Fit$plsFitMin        # Extract consensus PLS model
```

```
biplotPLS(PLSfit, comps=1:2, xCol = YR, labPlSc = FALSE, labPlLo = FALSE)
```



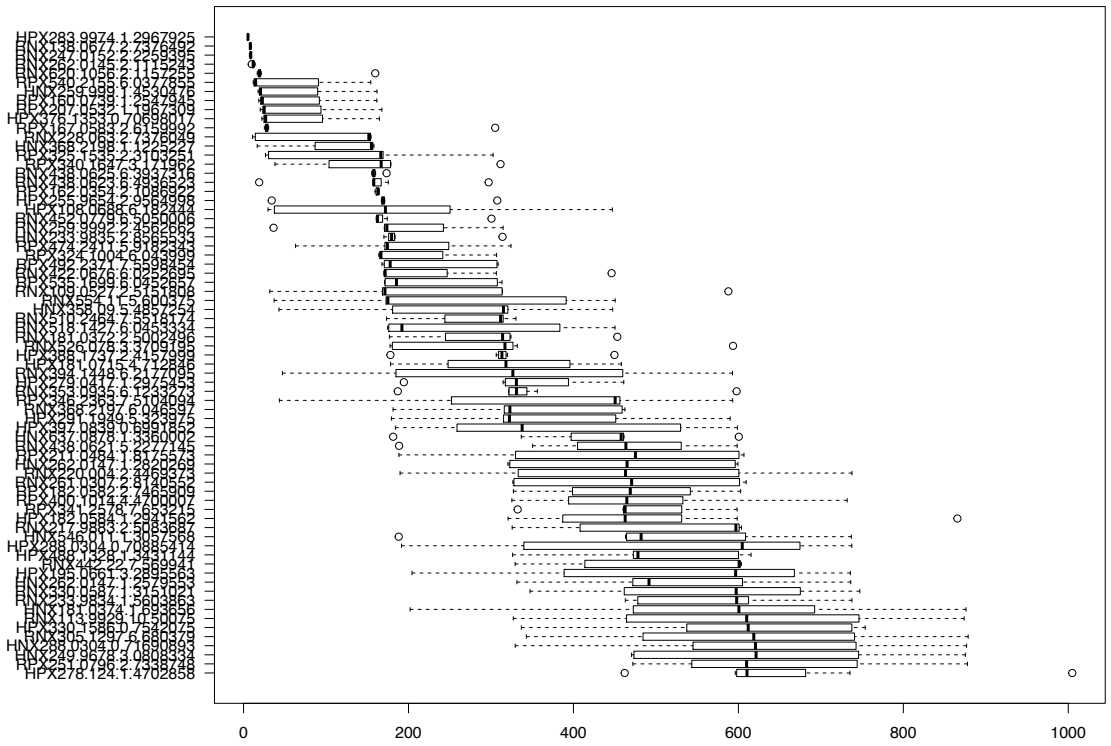
The prediction plot for regression analysis is different from classification. Instead of swim lanes, a regression is plotted with the actual response variable on the x-axis and predictions on the y-axis. Predictions from individual repetitions are represented as smaller, grey dots, whereas overall predictions are represented by larger black dots. Again, this provides an overview of prediction precision between repetitions. Inlaid are validation R^2 obtained from overall consensus model and Q^2 obtained from MUVR modelling. For convenience, the MUVR package also provides a PLS biplot function for visual interpretation of PLS models by giving an overview of observation scores and variable loadings. In the present case, the symbols are color coded (grey scale) according to their exposure (xCol = YR). To avoid cluttering, score and loading labels were omitted.

```
plotStability(regrModel, model='min')
```



The stability plot gives Q^2 instead of misclassifications in the bottom subplot, but is otherwise similar to the classification stability plot. There is higher variability in this model compared with the random forest classification described above, wherefore a higher nRep parameter setting is warranted.

```
plotVIP(regrModel, model='min')
```



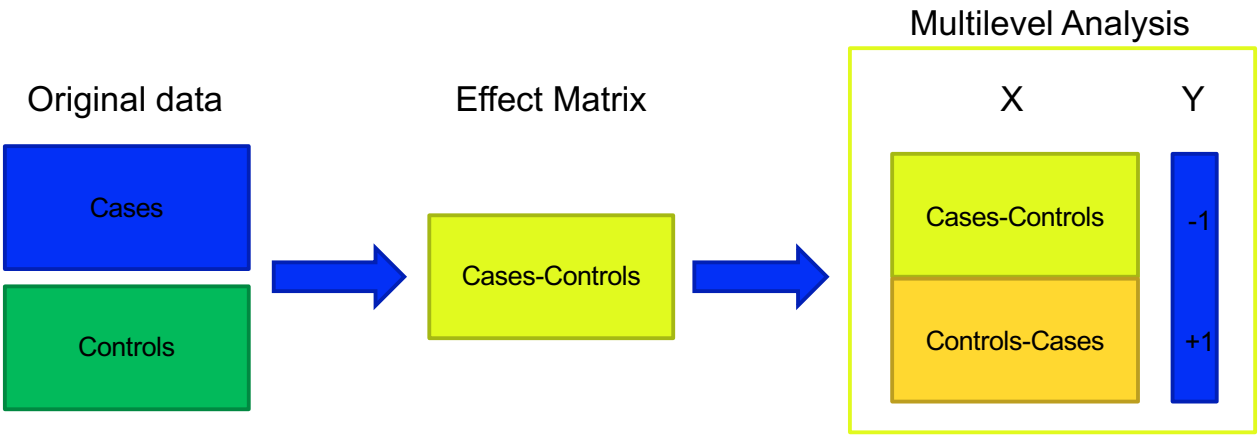

```
getVIP(regrModel, model='min')   # Extract most informative variables: Lower rank is better
#                          order                    name       rank
# HPX283.9974.1.2967925        1  HPX283.9974.1.2967925   5.255102
# RNX138.0677.2.7376492        2  RNX138.0677.2.7376492   8.239796
# RNX247.0152.2.2259395        3  RNX247.0152.2.2259395   8.538265
# RNX262.0145.2.1115243        4  RNX262.0145.2.1115243  11.617347
# …                            …  …                       …
```

Selected variables are obtained similar to the classification analysis.

Multilevel analysis

In the PLS regression example above, there was a dependency between observations due to repeated sampling. However, since the repeated measures were not related to a systematic effect, the dependency was managed by co-sampling observations from the same individual into the same cross-validation segments using the ID parameter. However, sample dependency may also be related to a systematic effect, from e.g. before-vs-after intervention or the effect of treatments A-vs-B in a cross-over design. Analysing such data using standard classification modelling will result in both (i) having observations from the same individual both in model training and testing as in the previous example and, in addition, (ii) conflation of between-individual effect and treatment-related systematic within-individual effects and. This sample dependency by experimental unit needs to be addressed in a different manner, e.g. by multilevel multivariate analysis (Ref Westerhuis). This should, however, not be confused with the multilevel model concept in classical statistics (https://en.wikipedia.org/wiki/Multilevel_model).

A standard classification analysis can analogously be viewed as a multivariate extension of the unpaired t-test. Using the same analogy, a multilevel model would correspond to a paired multivariate t-test, i.e. with pairwise linked samples (Ref Westerhuis). In reality, multilevel modelling is not a separate modelling technique, but rather a clever data pre-processing trick to manage sample dependency: Instead of modelling the original data from two discrete time points (or two treatments) as separate observations, an effect matrix is instead calculated as: $EM = X_B - X_A$, which is in turn modelled by regression using a dummy variable for the Y response vector. In MUVR, multilevel analysis is invoked by setting the ML parameter to TRUE. The user has to manually perform the effect matrix pre-processing and supply the resulting effect matrix as the X argument. The Y response vector is deliberately left out and calculated internally. Within the MUVR algorithm, further data pre-processing will be done to set up the multilevel analysis:

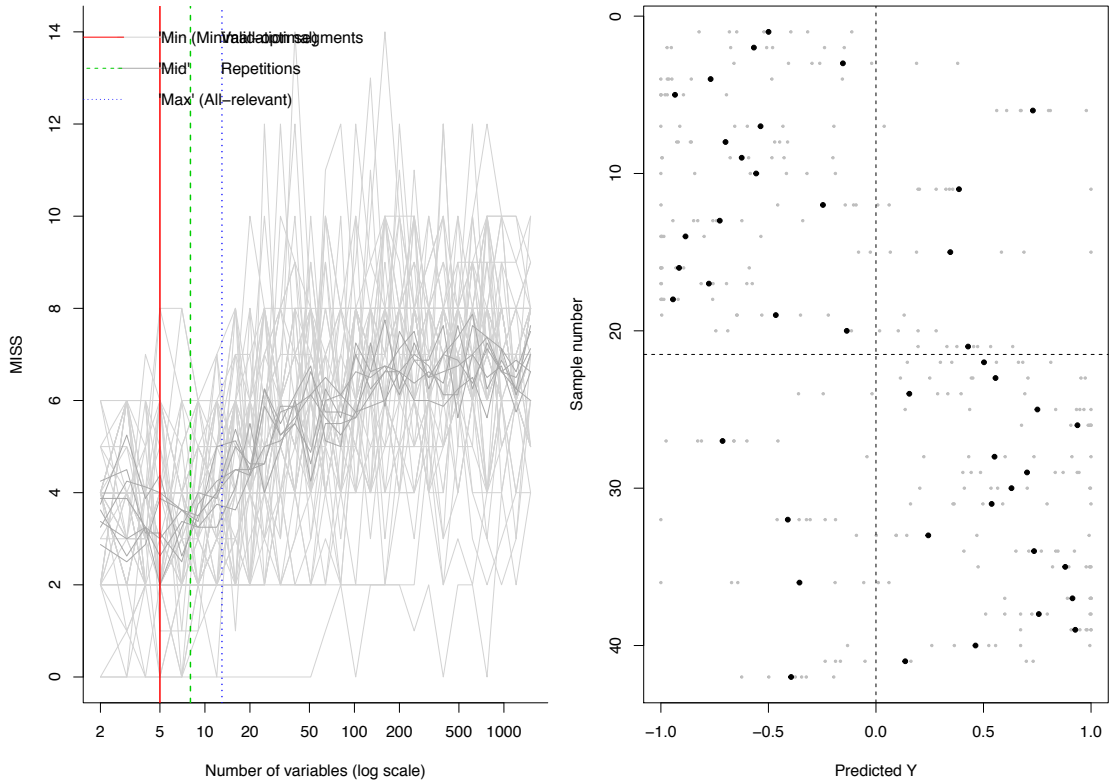


To make a multilevel analysis using MUVR, the user must pre-process the original data into an effect matrix (EM). MUVR is then called using the parameters X=EM and ML=TRUE. A new X matrix and a dummy Y response vector will then be calculated internally within MUVR.

In this multilevel example, the "crisp" dataset is used, which has untargeted plasma metabolomics data from two different dietary interventions delivered to 21 subjects in a cross-over design, i.e. where each participant received both diets. There is thus a clear sample dependency (by individual) advocating multilevel analysis. Typing `data("crisp")` will load an effect matrix ('crispEM') with 21 rows (one row per individual) and 1508 columns, consisting of difference in area-under-the-curve values (AUC) of metabolomics features measured after two different breakfast meal interventions.

```
#####################################################
# Multilevel example using the "crisp" data set

# Call in relevant libraries
library(doParallel)     # Parallel processing
library(MUVR)           # Multivariate modelling

# Call in the "crisp" data from the MUVR package
data("crisp")

# Set method parameters
nCore=detectCores()-1   # Number of processor threads to use
```

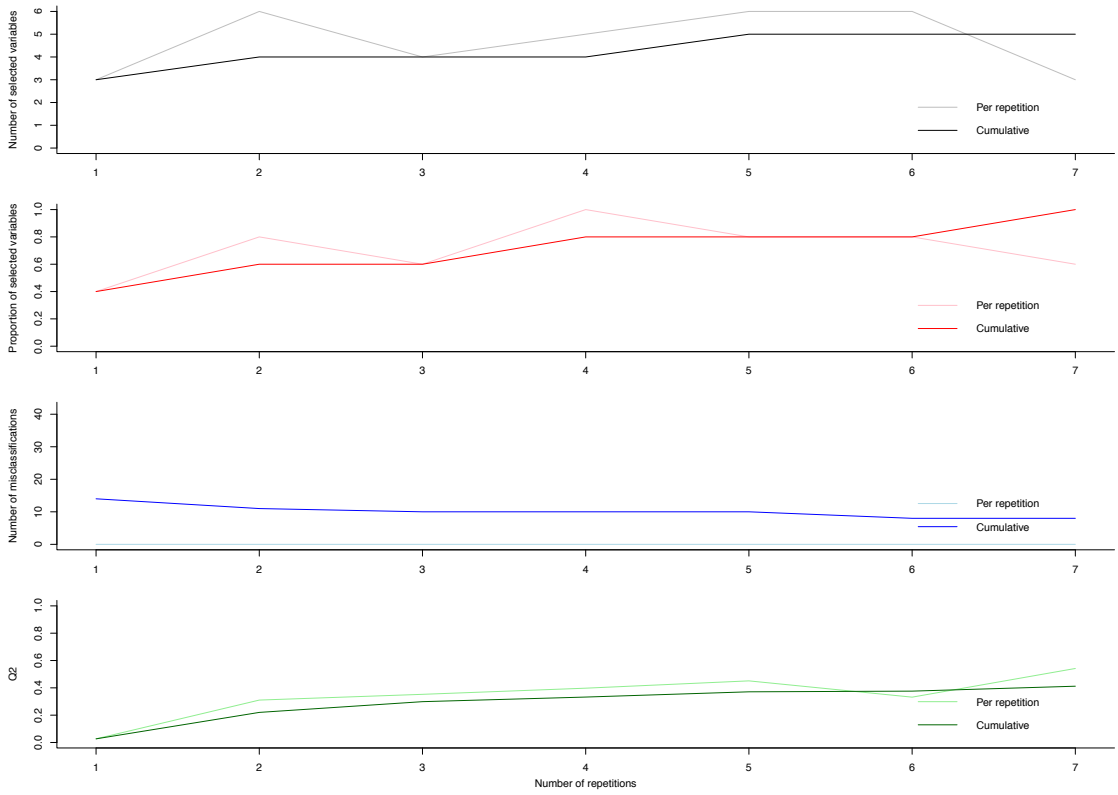
```
nRep=nCore              # Number of MUVR repetitions
nOuter=8                # Number of outer cross-validation segments
varRatio=0.8            # Proportion of variables kept per iteration
method='RF'             # Selected core modelling algorithm

# Set up parallel processing
cl=makeCluster(nCore)
registerDoParallel(cl)

# Perform modelling
MLModel = MUVR(X=crispEM, ML=TRUE, nRep=nRep, nOuter=nOuter, varRatio=varRatio, method=method)
# 1.0 mins using 7 threads on a Mac Powerbook Pro mid-2015 with 2,8 GHz Intel Core i7.

# Stop parallel processing
stopCluster(cl)

# Examine model performance and output
MLModel$nVar                    # Number of variables for min, mid and max models
# min mid max
#   5   8  13

MLModel$miss                    # Misclassified observations
# min mid max
#   8   8   8

MLModel$fitMetric               # Fitness metrics for min, mid and max models dummy regressions
# $R2
# [1] 0.7466235 0.7339821 0.7465286
# $Q2
# [1] 0.4119234 0.4026895 0.4296411

plotVAL(MLModel)
plotMV(MLModel, model='min')    # Look at the model of choice: min, mid or max
```



Again, the validation plot shows a similar behaviour as for classification. The multilevel prediction plot is different from both classification swim lane and regression line plots. The upper/lower half of the prediction plot represents the predicted Y response for the positive/negative half of the effect matrix. The expected Y values are the dummy values -1/1 for the upper/lower half and predictions have a decision boundary at Y=0. Consequently, 4 out of 21 individuals are misclassified. The MLModel$miss will however report 8 misses since the value is based on two instances per individual (i.e. upper and lower half). The stability is similar to both classification and regression and will provide both Q2 (for the dummy regression) and misclassifications both per repetition and cumulative.

```
MLModel$yPred                 # Multilevel predictions from min, mid and max models
#                min        mid        max
# 1_ID1   -0.4994952 -0.5607143 -0.5912095
# 2_ID2   -0.5676190 -0.5373714 -0.5622952
# 3_ID3   -0.1537429 -0.1745143 -0.2184952
# 4_ID4   -0.7686476 -0.8319429 -0.8445238
# …         …          …          …

MLModel$yClass                # Predicted class from min, mid and max models
#         min mid max
# 1_ID1    -1  -1  -1
# 2_ID2    -1  -1  -1
# 3_ID3    -1  -1  -1
# 4_ID4    -1  -1  -1
# …         …   …   …
```

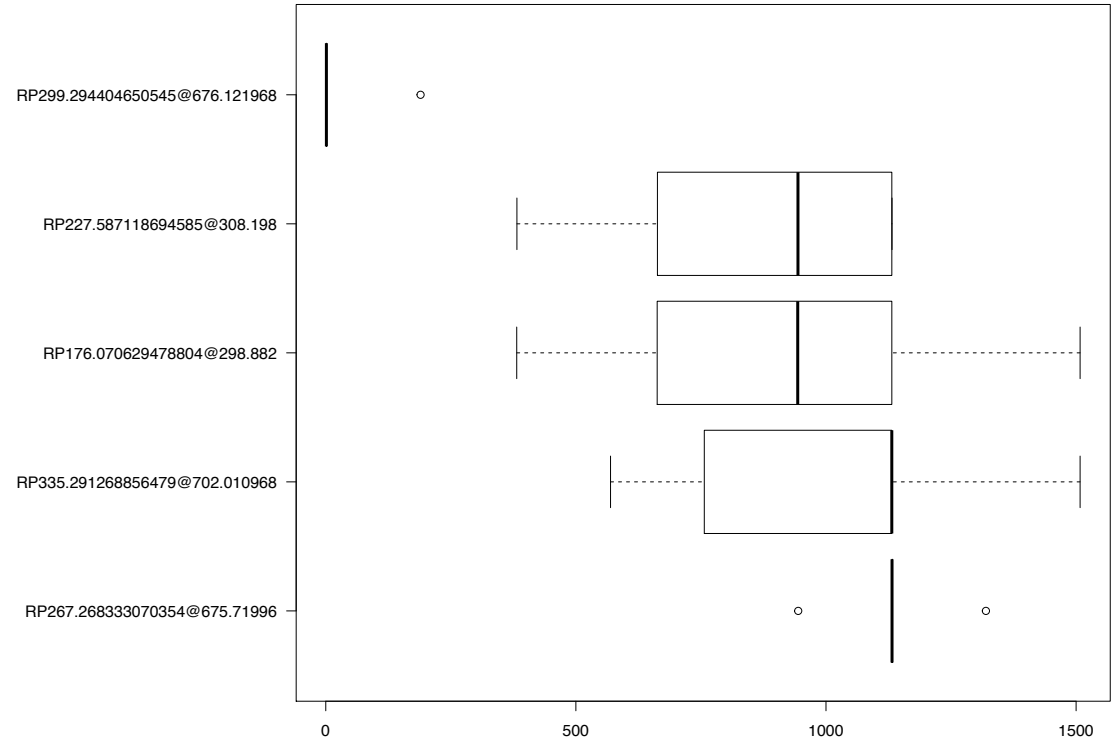
```
plotStability(MLModel, model='min')
```




```
getVIP(MLModel, model='min')   # Extract most informative variables: Lower rank is better
#                              order                          name       rank
```

```
# RP299.294404650545@676.121968     1 RP299.294404650545@676.121968   28.14796
# RP227.587118694585@308.198        2    RP227.587118694585@308.198  863.91071
# RP176.070629478804@298.882        3    RP176.070629478804@298.882  917.36480
# RP335.291268856479@702.010968     4 RP335.291268856479@702.010968  997.93878
# RP267.268333070354@675.71996      5  RP267.268333070354@675.71996 1132.23214
```

```
plotVIP(MLModel, model='min')
```

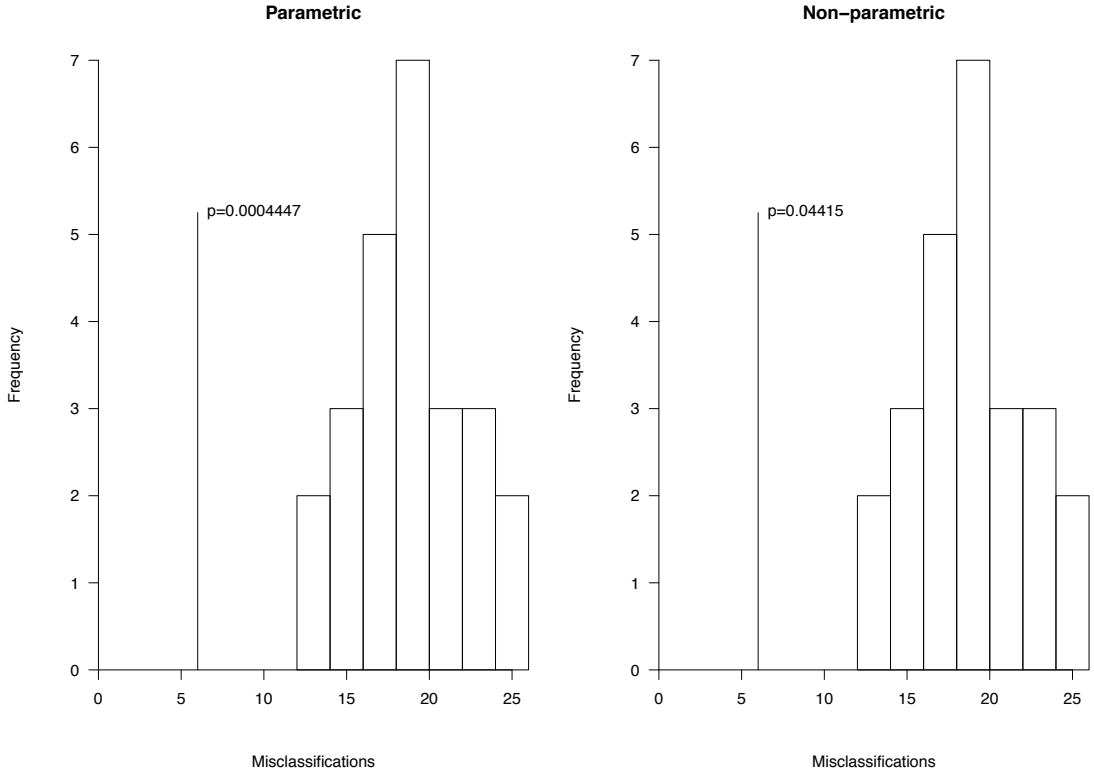


The variable selection output is also similar to classification and regression.

Permutation analysis

Permutation tests can be used to construct formal tests of significance of obtained analytical results (REFs). In brief, permutation tests are conducted by randomly sampling the response variable (permutation) and then modelling the permuted response using the original predictors. This modelling of random responses is repeated a number of times and statistical significance obtained from comparing the actual modelling result to the permutation distribution. Here, we will show a permutation of the random forest classification analysis above. The model parameter settings should ideally be identical between the actual model and the permuted models. However, to decrease computational cost, compromises need sometimes be made to reduce nRep, nOuter and varRatio parameter settings. Although not shown here, we have found that prediction correlations are usually very high for different parameter settings. But with reduced parameter settings, precision is usually decreased leading to a wider variability in estimates and therefore in fitness metrics. The use of reduced parameter settings will thus contribute to a wider permutation distribution and consequently to larger p-values (i.e. erring on the side of caution).

```
data("mosquito")

# Declare modelling parameters
nCore=detectCores()-1
nRep=2*nCore   # Number of repetitions per actual model and permutations
nOuter=5       # Number of validation segments
varRatio=0.75  # Proportion of variables to keep per iteration during variable selection
method='RF'    # Core modelling technique
model=1        # 1 for min, 2 for mid and 3 for max
nPerm=25       # Number of permutations (here set to 25 for illustration; normally set to ≥100)
permFit=numeric(nPerm)   # Allocate vector for permutation fitness

# Compute actual model and extract fitness metric; Approx 0.5 min
cl=makeCluster(nCore)
registerDoParallel(cl)
actual=MUVR(X=Xotu,Y=Yotu,nRep=nRep,nOuter=nOuter,varRatio=varRatio,method=method)
actualFit=actual$miss[model]

# Compute permuted models and extract fitness metrics; Approx 12 mins
for (p in 1:nPerm) {
  cat('\nPermutation',p,'of',nPerm)
  YPerm=sample(Yotu)
  perm=MUVR(X=Xotu,Y=YPerm,nRep=nRep,nOuter=nOuter,varRatio=varRatio,method=method)
  permFit[p]=perm$miss[model]
}
stopCluster(cl)

# Parametric (Student's) permutation test significance
pPerm(actual = actualFit, h0 = permFit)
plotPerm(actual = actualFit, h0 = permFit) # Look at histogram to assess whether Gaussian in shape

# Non-parametric (rank-Student's) permutation test significance
pPerm(actual = actualFit, h0 = permFit, type = 'non') # If not Gaussian, make a non-parametric test
plotPerm(actual = actualFit, h0 = permFit, type = 'non') # And plot with non-parametric p-value instead
```



The assumption of gaussianity of the permutation distribution can be inspected from the permutation plot. It is normally useful to perform initial permutation tests with few repetitions to get an indication of actual model fitness in relation to the permutation distribution. Additional repetitions can then easily be added to the permutation distribution at will. For regression tests, the actual and permuted fitness measures to be extracted should instead be `actual$fitMetric$Q2[model]` and `perm$fitMetric$Q2[model]`, respectively. For multilevel analysis, misclassifications should be used as fitness metric. However, to accommodate for permuted responses per individual summing to zero (i.e. one -1 and on 1 per individual), a permuted Y variable should be specified instead of omitting the Y variable. Internally, for multilevel analysis, the full Y response vector is obtained by Y <- c(Y, -Y), similar to X <- cbind(X, -X). The permutation loop should thus look like:

```
for (p in 1:nPerm) {
  cat('\nPermutation',p,'of',nPerm)
  YPerm=sample(c(-1,1),size = nrow(crispEM),replace=TRUE) # Make permuted classes per individual
  perm=MUVR(X=crispEM,Y=YPerm,ML=TRUE,nRep=nRep,nOuter=nOuter,varRatio=varRatio,method=method)
  permFit[p]=perm$miss[size]
}
```

References