A Brief Tutorial on MUVR 2.0:
Multivariate methods with Unbiased Variable selection in R

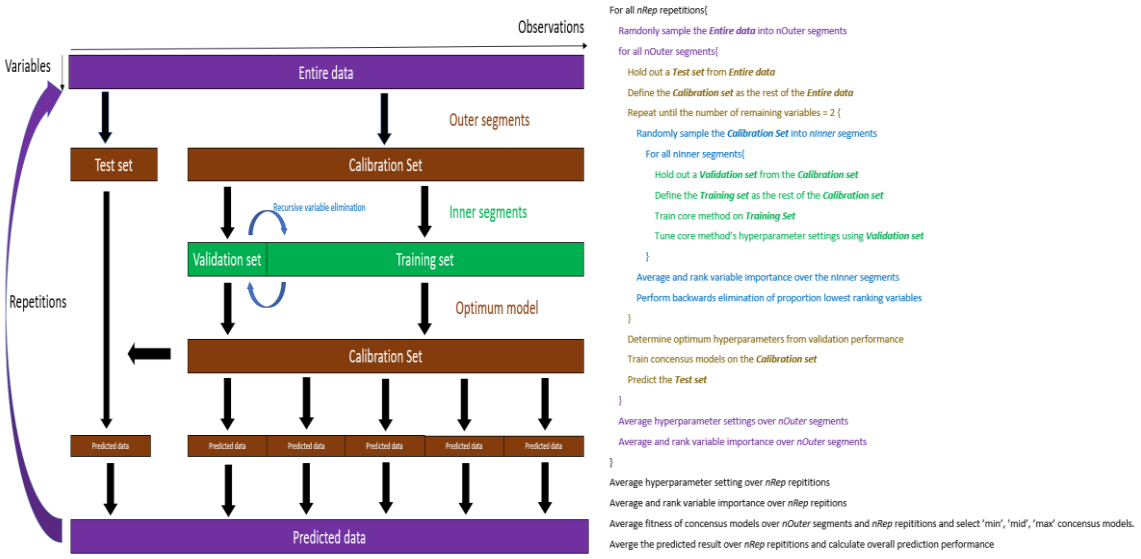
Lin Shi, Yingxiao Yan and Carl Brunius, Gothenburg, 29 May 2023

Catalogue

What is MUVR?

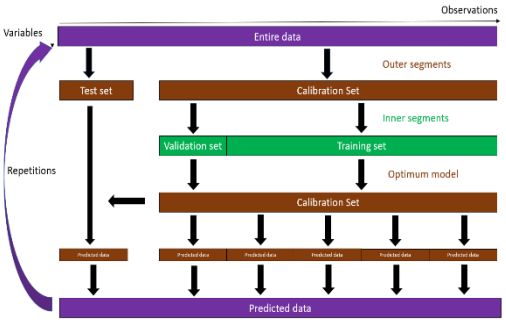
The MUVR package is an algorithm for multivariate modelling, aimed at finding associations between predictors (an X matrix of continuous and categorical variables) and a target variable (a Y vector; continuous for *regression* or factor for *classification*). MUVR is particularly useful to cope with 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 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 and 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). Our recently upgraded MUVR (version 2.0) allows partial least squares (PLS), random forest (RF) and elastic net (EN) core modelling and can perform covariate adjustment in its EN core modelling.




```
For all nRep repetitions{
  Ramdonly sample the Entire data into nOuter segments
  for all nOuter segments{
    Hold out a Test set from Entire data
    Define the Calibration set as the rest of the Entire data
    Repeat until the number of remaining variables = 2 {
      Randomly sample the Calibration Set into nInner segments
      For all nInner segments{
        Hold out a Validation set from the Calibration set
        Define the Training set as the rest of the Calibration set
        Train core method on Training Set
        Tune core method's hyperparameter settings using Validation set
      }
      Average and rank variable importance over the nInner segments
      Perform backwards elimination of proportion lowest ranking variables
    }
    Determine optimum hyperparameters from validation performance
    Train concensus models on the Calibration set
    Predict the Test set
  }
  Average hyperparameter settings over nOuter segments
  Average and rank variable importance over nOuter segments
}
Average hyperparameter setting over nRep repititions
Average and rank variable importance over nRep repitions
Average fitness of concensus models over nOuter segments and nRep repititions and select 'min', 'mid', 'max' concensus models.
Averge the predicted result over nRep repititions and calculate overall prediction performance
```

Working principle of MUVR-PLS and MUVR-RF modelling. A: Graphical representation of the MUVR algorithm for PLS and RF core modelling. 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 for PLS and RF core modelling.




```
For all nRep repetitions{
  Ramdonly sample the Entire data into nOuter segments
  for all nOuter segments{
    Hold out a Test set from Entire data
    Define the Calibration set as the rest of the Entire data
    Randomly sample the Calibration Set into nInner segments
    For all nInner segments{
      Hold out a Validation set from the Calibration set
      Define the Training set as the rest of the Calibration set
      Train core method on Training Set
      Tune core method's hyperparameter settings using Validation set
    }
    Determine optimum hyperparameters from validation performance
    Train concensus models on the Calibration set
    Predict the Test set
  }
}
Average hyperparameter setting over nOuter segments and nRep repititions
Calculate and rank variable importance over nOuter segments and nRep repitions
Average fitness of concensus models over nOuter segments and nRep repititions and select 'min', 'mid', 'max' concensus models.  ## here the consensus models is each calibration set
Averge the predicted result over nRep repititions and calculate overall prediction performance
```

Working principle of MUVR-EN modelling. A: Graphical representation of the MUVR algorithm for EN core modelling. 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. 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 for EN core modelling.

Installation

Please download and install R 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 that you may run is shown in `red monotype font`. Functions, hyperparameters and command is written in *italic*.

1. Install the release version of 'remotes' from CRAN:

```
install.packages("remotes")
```

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(remotes)
install_gitlab("YingxiaoYan/MUVR")
```

4. To reduce computation time, MUVR uses the 'doParallel' package for parallel processing:

```
install.packages("doParallel", repos="http://R-Forge.R-project.org")
```

Preparing data

The MUVR algorithm uses predictor matrix (X) and target variable (Y) data which are matched by observations, i.e. each position in the Y target variable 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 customed by variables) and scaling (unit variance, pareto, none or customed by variables).

The effect of predictor variables' scaling in MUVR will depend on the choice of core algorithm. For example, 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 center to mean 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.'

The next 3 sections are 3 examples of how you can use different core modeling methods in different problems, i.e. MUVR-PLS for regression analysis with repeated samples, MUVR-EN for classification analysis and MUVR-RF for multilevel analysis. Importantly, to avoid any confusion, we emphasize that you can use any of the 3 core modelling methods for any of the 3 problems, with or without repeated samples. This means that you could use MUVR-PLS, MUVR-RF and MUVR-EN for regression problems, with or without repeated samples; You could use MUVR-PLS, MUVR-RF and MUVR-EN for classification problems, with or without repeated samples; You could use MUVR-PLS, MUVR-RF and MUVR-EN for multilevel problems, with or without repeated samples.

Regression analysis with repeated samples

We will walk through a MUVR regression analysis with repeated samples using PLS core modelling. 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). The corresponding explanation is written above each chunk of code.

First, we set up libraries, data and parameters for multivariate modelling: Typing `library(doParallel)` and `library(MUVR)` will call in relevant libraries and then typing `data("freelive")` will load 3 objects: A continuous 'YR' target variable 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).

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

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

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 & 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). 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.

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.

```
# 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=0.75            # Proportion of variables kept per iteration
method='PLS'            # Selected core modelling algorithm
```

After setting parameters, it is time to initialize parallel processing and perform the actual modelling. Importantly, 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.

```
# 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)
# 1.35 mins using 7 threads on a laptop computer with 11th Gen Intel I Core I i7 -1185G7 processor
with 8 cores and 31.7 Gb internal memory.

# Stop parallel processing
stopCluster(cl)
```

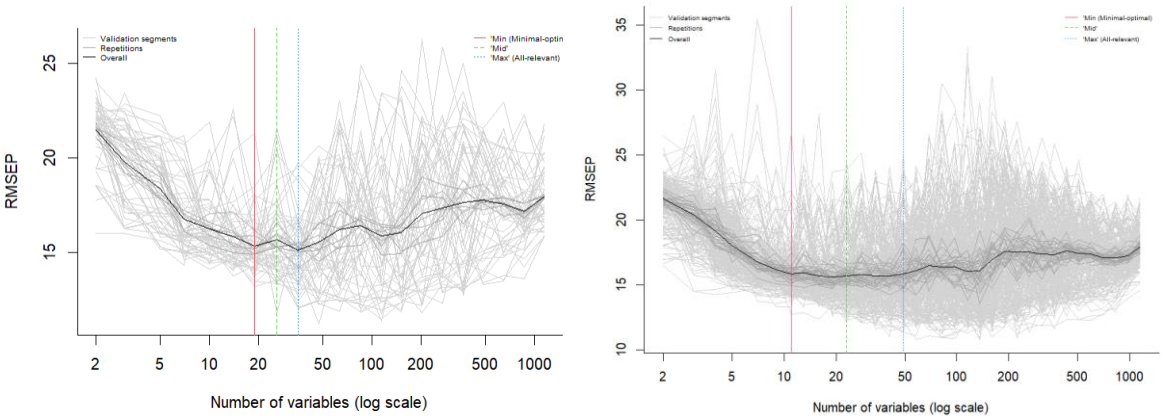
After the model finishes running, we could check the prediction performance, hyperparameter selection and number of variable selected by 'min', 'mid', 'max' consensus models and shows actual target variables side by side with 'min', 'mid', 'max' predictions.

```
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
# …         …           …          …          …
```

In the validation plot below, 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 & 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, root mean squared error of prediction (RMSEP)) is minimal. This is in practice determined as having validation performance within certain percentage of slack allowance from the actual minimum (default 5%). 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.

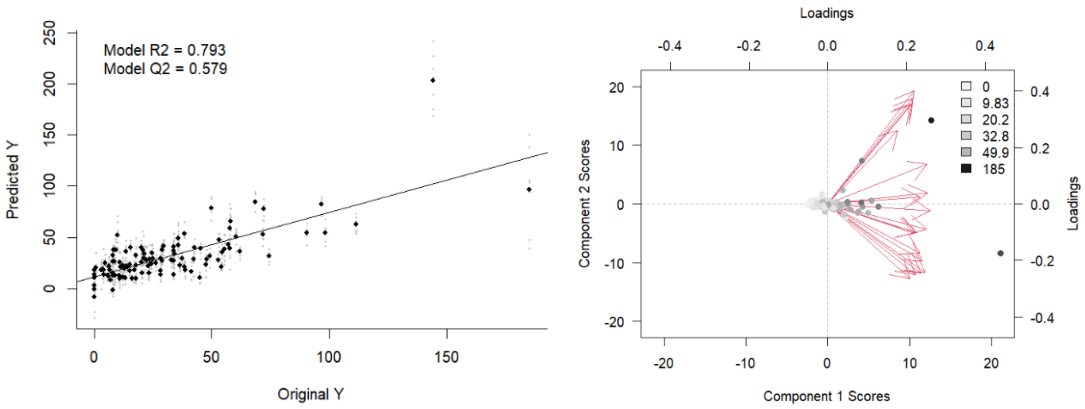
```
plotVAL(regrModel)
```

nRep=7, varRatio=0.75, nOuter=8, t=1.25min nRep=35, varRatio=0.85, nOuter=8, t=9.32min



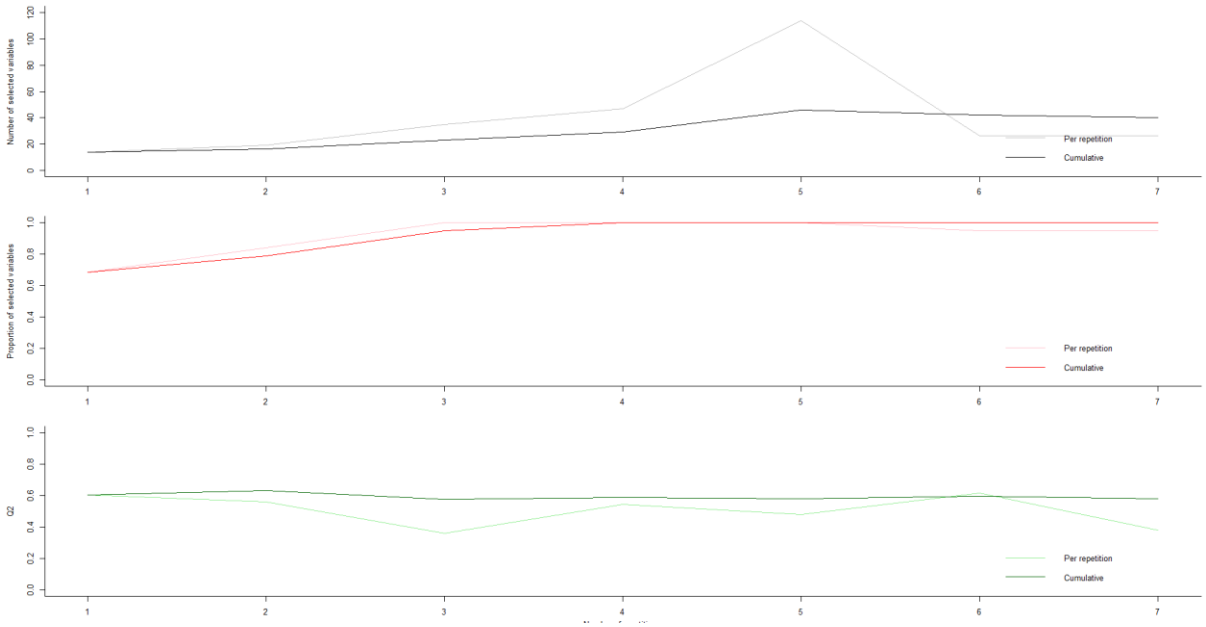
For regression analysis, *plotMV()* function plots the actual response variable on the x-axis and predictions on the y-axis. Predictions from individual repetitions are represented as smaller, grey dots, whereas overall predictions are represented by larger black dots. This provides an overview of prediction 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.

```
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)
```



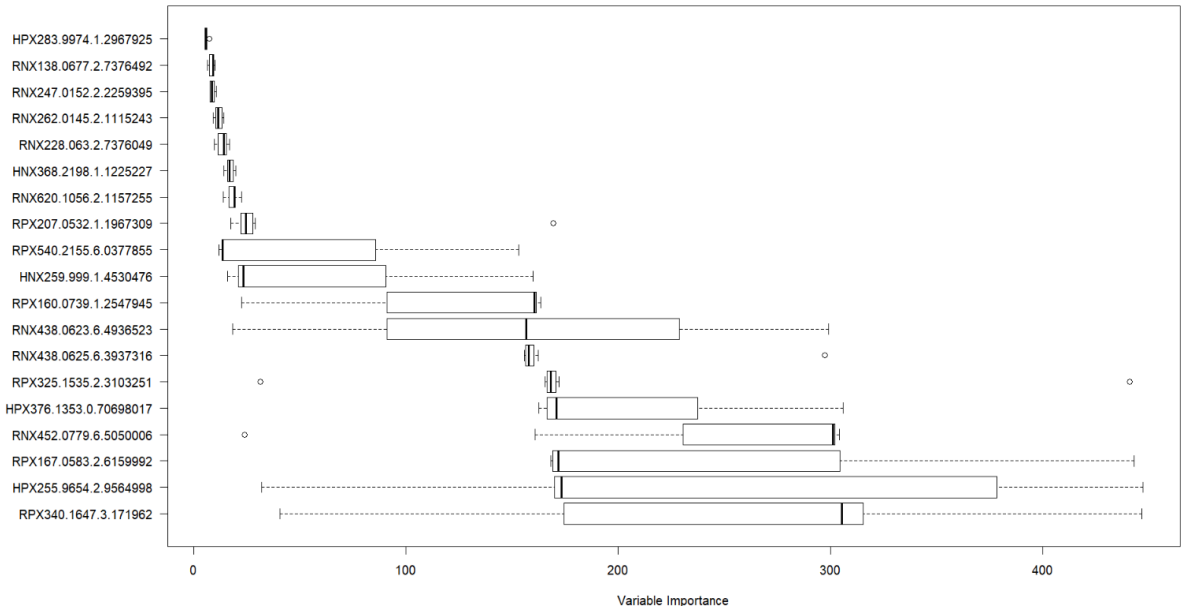
The stability plot from *plotStability()* for regression analysis generates three subplots: (i) Number of selected variables for each repetition as well as cumulative average over the repetitions; (ii) The proportion of selected variables reports the ratio of the final variable selection found in each repetition and cumulatively, averaged over the number of repetitions; (iii) Q^2 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 on the strength of the analysis. If there is higher variability in the model a higher nRep parameter setting is warranted.

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



The *plotVIRank()* 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 target variable.

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



The selected variables can also be obtained from:

```
getVIRank(regrModel, model='min')     # Extract most informative variables: Lower rank is better
#                          order                    name       rank
# HPX283.9974.1.2967925      1     HPX283.9974.1.2967925   5.885204
# RNX138.0677.2.7376492      2     RNX138.0677.2.7376492   8.436224
# RNX247.0152.2.2259395      3     RNX247.0152.2.2259395   8.793367
# RNX262.0145.2.1115243      4     RNX262.0145.2.1115243  11.793367
```

Note that regression using Random forest is easily achieved by changing the 'method' parameter (see code below) and regression using elastic net will be achieved by the *rdCVnet()* function (see next section)

Classification analysis

The general outline of a classification analysis follows the same approach as the regression analysis described above, including considerations for parameter settings, albeit with the difference that the Y target variable consists of factor levels rather than numeric values. We will walk through a MUVR classification analysis using elastic net (EN) core modelling. 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).

First, we set up libraries, data and parameters for multivariate modelling: We call in relevant libraries and typing `data("mosquito")` will load 2 objects: A categorical 'Yotu' target 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.

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

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

We then check for the number of observations per class to make sure *nOuter* is not bigger than the number of observations of the smallest class.

```
# Check number of observations per class
table(Yotu)             # As a general principle, nOuter ⩽ n of the smallest class (i.e. 8)
# Yotu
# VK3 VK5 VK7
# 10  11   8
```

The elastic net in the MUVR algorithm is slightly different from PLS and RF. First, we used *rdCVnet()* function instead of *MUVR()* to perform the modelling and we do not need to specify a 'method' since *rdCVnet()* only performs elastic net core modelling. Second, EN core modelling in MUVR does not perform recursive variable elimination based on variable importance rankings as MUVR-PLS and MUVR-RF for variable selection. Instead, it ranks variables by times of having non-zero beta coefficients over *nOuter* segments and *nRep* repetitions. Variables that have non-zero beta coefficients more times are ranked lower (lower is better). Finally, *rdCVnet()* does not give number of variables selected (*nVar*) for 'min' 'mid' and 'max' consensus models directly as output. One needs to put the result from *rdCVnet()* to *rdCVnet_gerVar()* to generate *nVar* and *Var* in the result.

Bearing the clarification above and we initialise parallel processing and perform the actual modelling

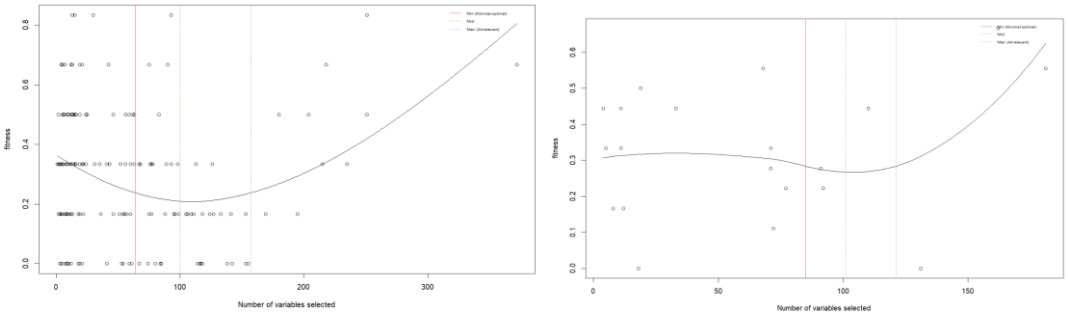
```
# Set method parameters
nRep=30             # Number of MUVR repetitions
nOuter=6                # Number of outer cross-validation segments
varRatio=0.75            # Proportion of variables kept per iteration


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

```
# Perform modelling
classModel = rdCVnet(X=Xotu, Y=Yotu, nRep=nRep, nOuter=nOuter, varRatio=varRatio, DA=T)

classModel<-rdCVnet_getVar(classModel, option="smoothcurve",percent_smoothcurve=0.05,outlier=F)

# Stop parallel processing
stopCluster(cl)
```

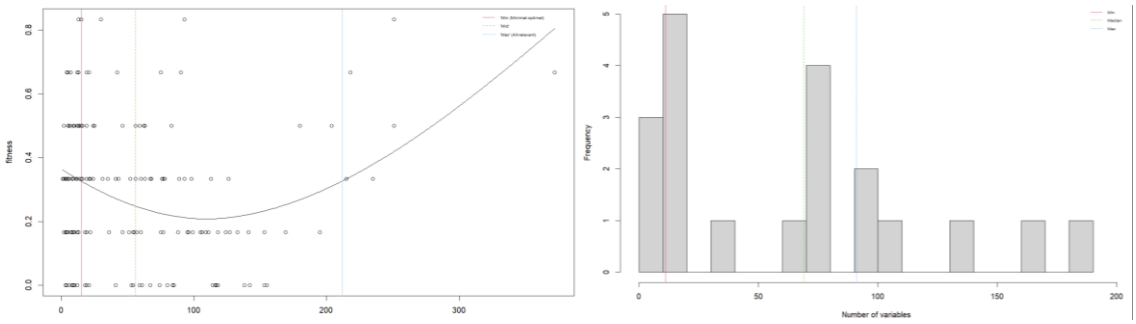
We did not put *nVar* as the output of MUVR-EN directly because *nVar* in MUVR-EN is not chosen from a known set of options decided by the total number of variables and *varRatio* (as in MUVR-PLS and MUVR-RF), but from a smooth curve generated from *nRep*nOuter* number of data points visualized by *plotVAL()*, where the x-axis is numbers of variables that have non-zero coefficient (for each repetition and outer segment) and y-axis is validation performance calculated from each *test set*. Note that the 'min' 'mid' and 'max' of *nVar* is decided similarly as MUVR-PLS and MUVR-RF (validation performance within a certain percentage of slack allowance from the actual minimum (default *percent_smoothcurve* = 5%).)

Given that the smooth curve depends on the data points and *nVar* does not have a known set of options, we want users to use *plotVAL()* to plot first to judge if the 'min' 'mid' and 'max' selected in the plot are reasonable and make adjustments before settling the *nVar* values. For example, in our first plot generated from the code above, 'min' 'mid' and 'max' all reside at the places where major of data points do not reside, which is counterintuitive for our selection of the number of variables. In the second plot, it seems the curve (not a U shape) overfits the data points therefore the 'min' 'mid' and 'max' generated from the curve may be not convincing. Note that the second plot shown here is an example generated from different a setting of *nRep* and *nOuter*, only for the purpose of illustration.

```
plotVAL(classModel)
```



So how can you make adjustments if something does not make sense in the plot? In the first plot above, since you may believe that the problem comes from the selection of 'min' 'mid' and 'max' from the curve but not the curve itself, you could simply run *rdCVnet_getVar()* again by adding more allowance for the deviation from the optimum validation performance by using *percent_smoothcurve* (See the plot on the left below). Sometimes, the smooth curve overfits the data points (e.g. squiggling curve, or the second plot above). In this case, one could try to remove outlier (*outlier=T*) and adjust how well the smooth curve should fit the data points (*span*) in *rdCVnet_getVar()* to get a U shape curve in *plotVAL().* If these 2 tricks do not work (e.g. on the second plot above), we need to avoid to 'min' 'mid' and 'max' from the curve since we cannot remove obvious overfitting. There is also an *option="quantile"* option and *percent_quantile* that allow users to choose 'min' 'mid' and 'max' for *nVar* directly based on quantiles, which we could choose to use in this case. For example, *option="quantile", percent_quantile=0.25* selects the first quartile, median and the third quartile for the 'min' 'mid' and 'max' of *nVar* (See the plot on the right below).

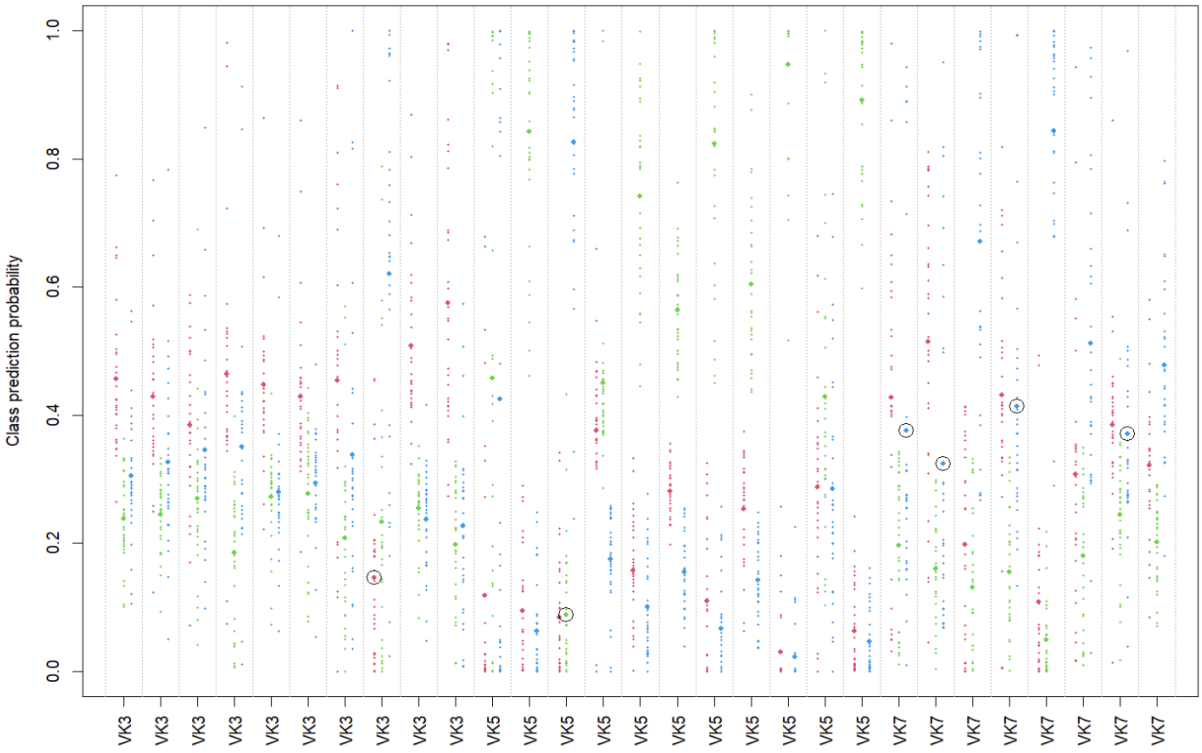


We then look at the outputs as the regression analysis elaborated before. Note that classification analysis uses different fitness metrics (i.e. balanced error rate (BER)) instead of Q2 and RMSEP in regression. Since there is no recursive variable elimination for MUVR-EN and *nVar* is selected after model building, we only have one BER and predicted target variable for the model instead of one for 'min' 'mid' and 'max' each.

```
classModel$ber                    # Number of misclassifications for min, mid and max models
# 0.230303

classModel$nVar                    # Number of variables for min, mid and max models
# min  mid   max
# 64   100   157

cbind.data.frame(Yotu, classModel$yClass)   # Actual class side-by-side with min, mid and max predictions
#     Yotu         classModel2$yClass
# 1   VK3                VK3
# 2   VK3                VK3
# 3   VK3                VK3
# 4   VK3                VK3
# …   …                  …
```

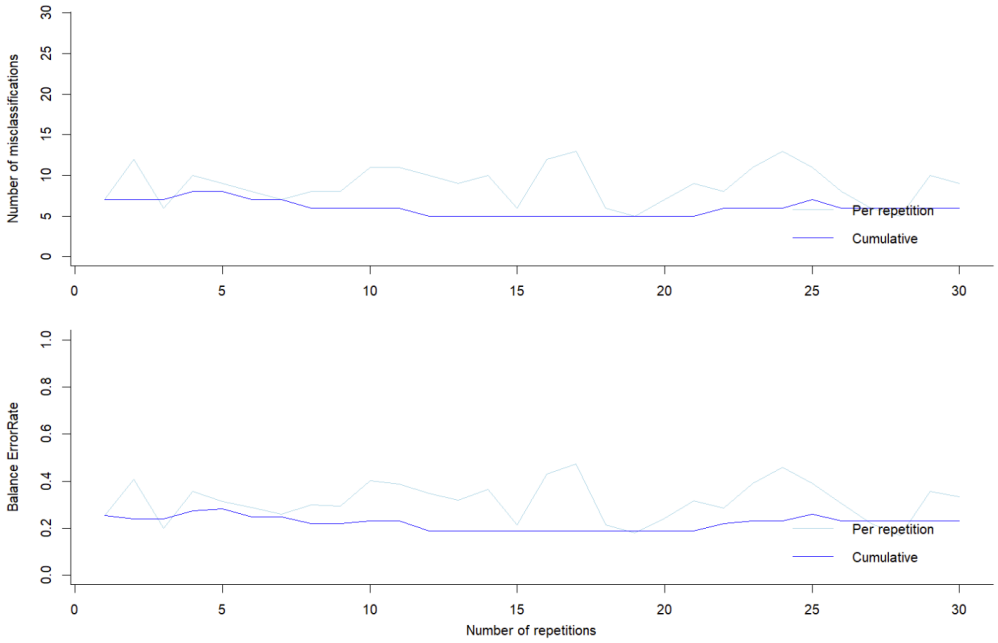
The prediction plot for classification analysis is different from regression. The plotMV() function generates a "swim lane" plot, where each swim lane shows individual and overall predictions for each observation. Classes are color-coded and jittered by class. The smaller dots represent predictions from 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. In MUVR-PLS and MUVR-RF, using the *model* argument, the users can easily switch between 'min', 'mid' and 'max' models.

```
plotMV(classModel)          # Look at the model of choice in MUVR-PLS and MUVR-RF: min, mid or max
```



The stability plot from gives BER instead of Q^2 in the bottom subplot, but is otherwise similar to the regression stability plot. Note that since there is no recursive variable elimination, it is not applicable to show how number and proportion of selected variables change with number of repetitions.

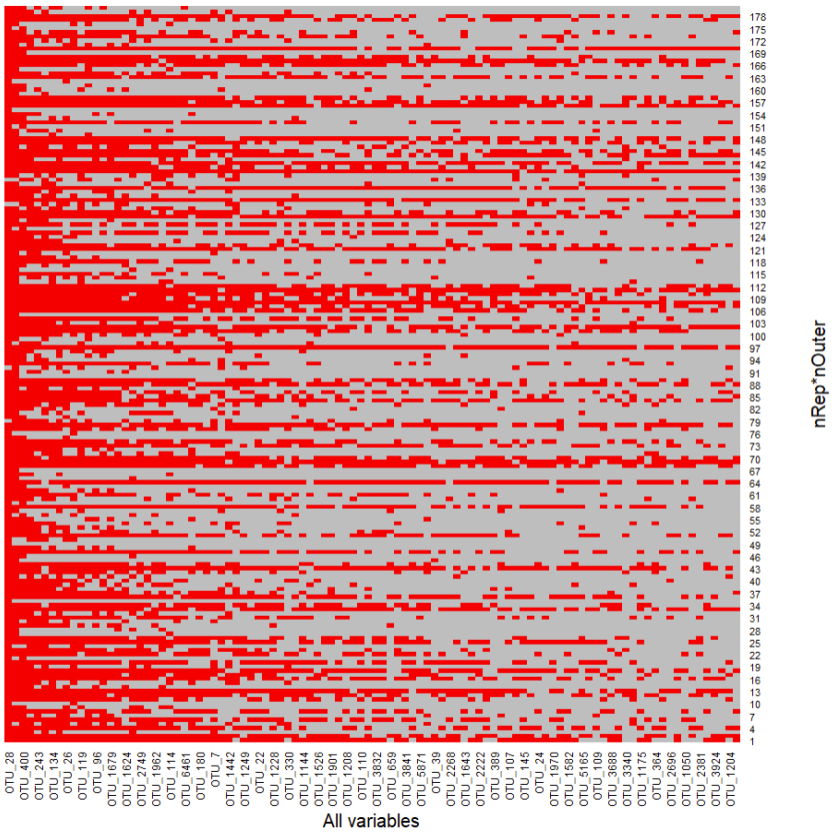
```
plotStability(classModel)
```



Selected variables can be obtained similar to the regression analysis by using *plotVIRank()* and *getVIRank()*. Note that *plotVIRank()* in MUVR-EN gives a different plot than MUVR-PLS and MUVR-RF, since variable selection in MUVR-EN is based on the variables that have non-zero beta-coefficient over *nOuter* segments and *nRep* repetitions rather than recursive variable elimination. In the plot below, each column represents one variable and variables selected more times over *nOuter* segments and

nRep repetitions (i.e. high variable importance and lower rank) is placed at the left side of the plot. Each row is the variable selection result using each outer CV segment in each repetition, where red represents that a variable has non-zero beta-coefficient.

```
plotVIRank(classModel, model='mid',maptype="heatmap")
```

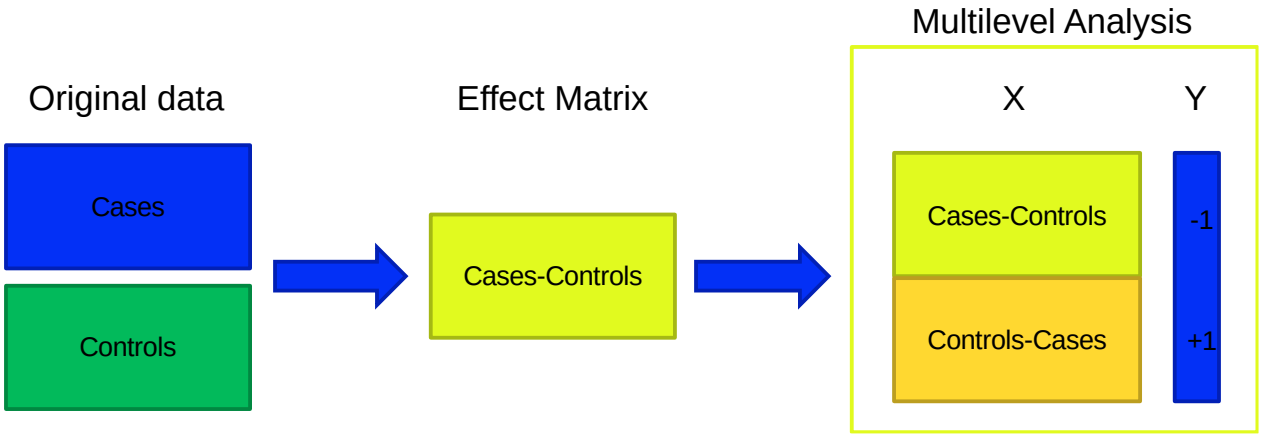



```
getVIRank(classModel, model='mid')   # Extract most informative variables: Lower rank is better
#             order     name       rank
# OTU_28        1     OTU_28       1.0
# OTU_4133      2     OTU_4133     2.0
# OTU_1620      3     OTU_1620     3.0
# OTU_400       4     OTU_400      4.0
# …             …     …            …
```

Multilevel analysis

In the PLS regression example, 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 (Westerhuis et al, 2010). 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 (Westerhuis et al, 2010). 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 target variable. In the MUVR package, multilevel analysis is invoked by setting the ML parameter to TRUE. The user has to manually perform pre-processing to generate the effect matrix and supply the effect matrix as predictors. The Y target variable 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 predictor matrix and a dummy Y target variable will then be calculated internally within MUVR.

In this multilevel example using MUVR-RF, 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.

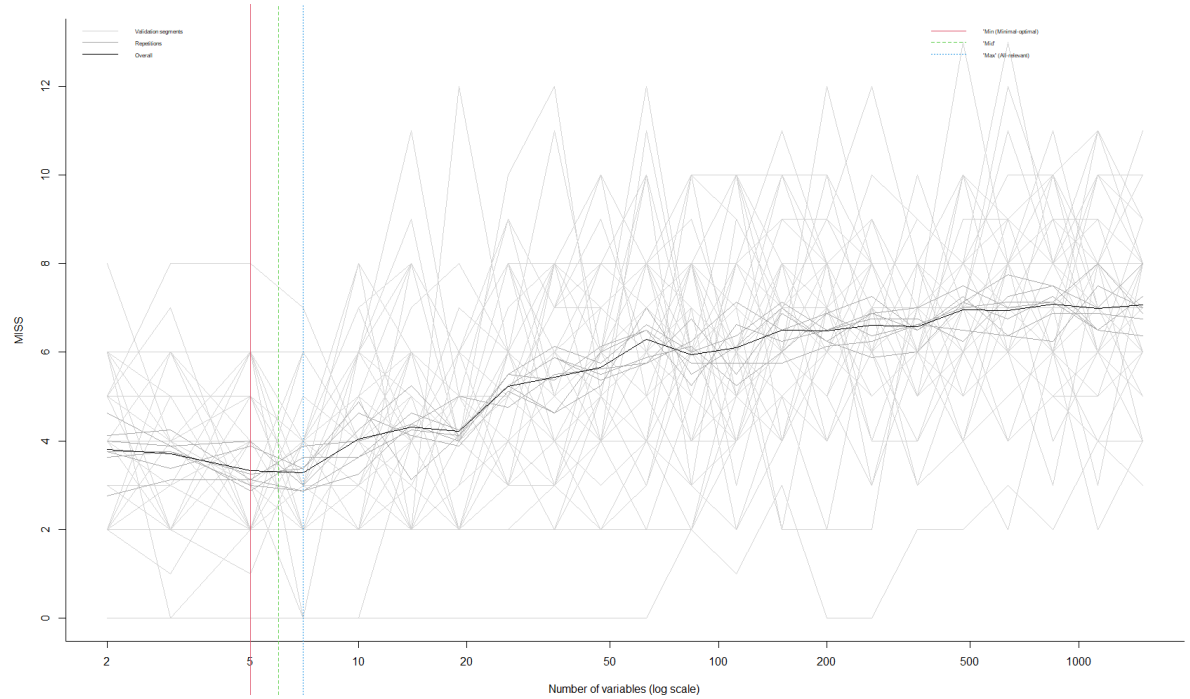
Given the details of the codes and results have been mostly elaborated in the regression and classification analysis, we only add explanations for the part that is different in multilevel analysis.

```
# 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=0.75           # Proportion of variables kept per iteration
method='RF'             # Selected core modelling algorithm
```

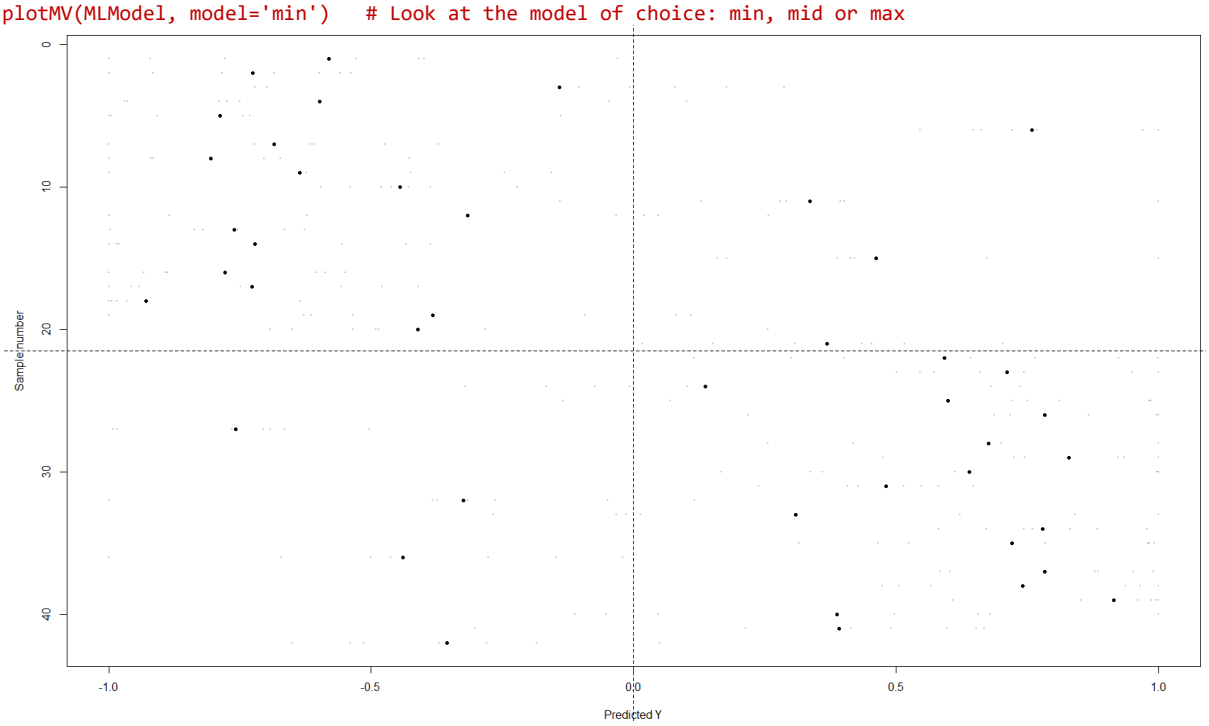
```
# Set up parallel processing
cl=makeCluster(nCore)
registerDoParallel(cl)

# Perform modelling
MLModel = MUVR(X=crispEM, ML=TRUE, nRep=nRep, nOuter=nOuter, varRatio=varRatio, method=method)

# 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   6  7

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

MLModel$ber
# min       mid       max
# 0.1904762 0.1904762 0.1904762

MLModel$fitMetric               # Fitness metrics for min, mid and max models dummy regressions
# $R2
#       min       mid       max
# 0.7382263 0.7560964 0.7495876

# $Q2
#       min       mid       max
# 0.4265355 0.3909502 0.3860555

MLModel$yPred                   # Multilevel predictions from min, mid and max models
#                min        mid         max
# 1_ID1   -0.5804444 -0.5025397 -0.55692063
# 2_ID2   -0.7255238 -0.6231111 -0.63400000
# 3_ID3   -0.1409365 -0.1821270 -0.22317460
# 4_ID4   -0.5986032 -0.7677460 -0.72241270
# …        …          …          …

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
# …         …   …   …
```

The validation plot from *plotVAL()* shows a similar behaviour as for regression. The multilevel prediction plot from *plotMV()* is different from both regression line plots and classification swim lane. The upper/lower half of the prediction plot represents the predicted Y target variable 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 (number of miss classifications) will however report 8 misses since the value is based on two instances per individual (i.e. upper and lower half).

```
plotVAL(MLModel)
```

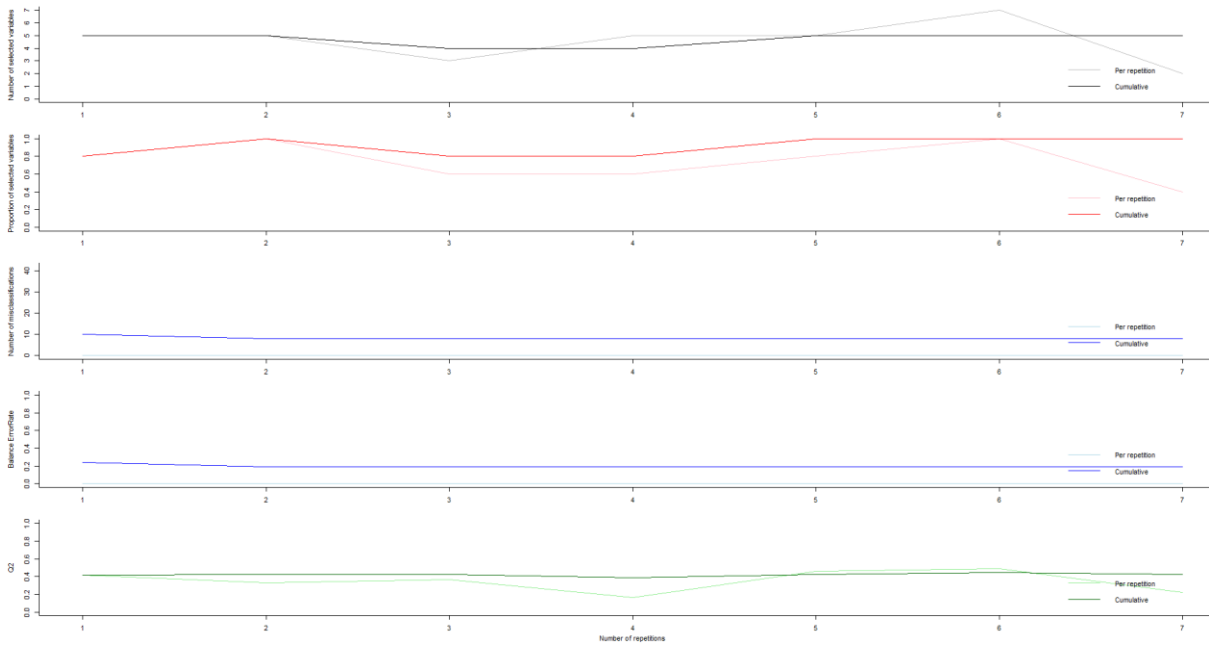



```
plotMV(MLModel, model='min')   # Look at the model of choice: min, mid or max
```



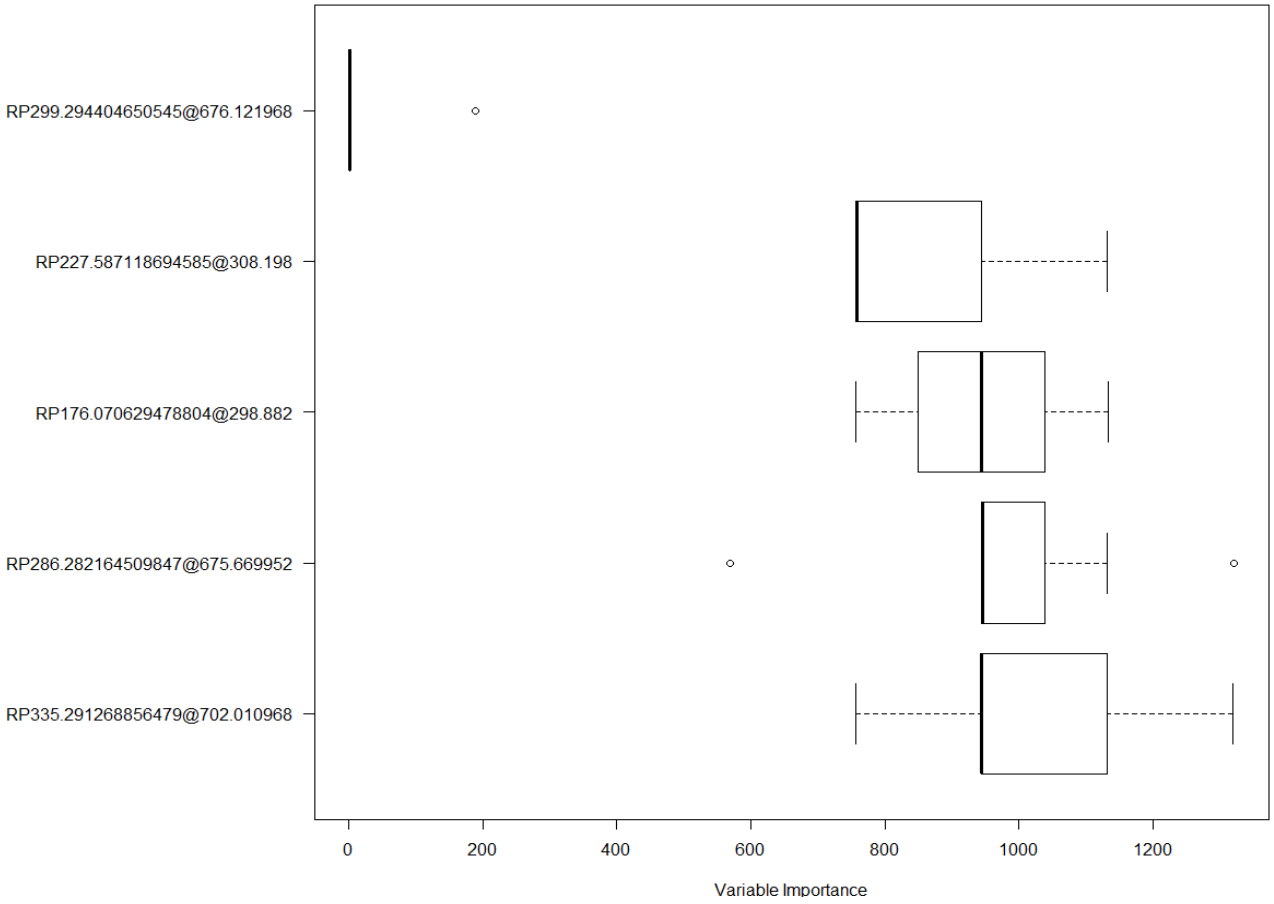
The stability plot from *plotStability()* is similar to both regression and classification and will provide Q^2 (for the dummy regression), number of misclassification and BER both per repetition and cumulative.

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



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

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




```
getVIRank(MLModel, model='min')   # Extract most informative variables: Lower rank is better
#                    order                          name                rank
# RP299.294404650545@676.121968     1    RP299.294404650545@676.121968   28.24362
# RP227.587118694585@308.198        2    RP227.587118694585@308.198      863.95408
# RP176.070629478804@298.882        3    RP176.070629478804@298.882      944.25000
```

```
# RP286.282164509847@675.669952     4    RP286.282164509847@675.669952   971.42092
# RP335.291268856479@702.010968     5    RP335.291268856479@702.010968   1024.52551
```

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 for the regression, classification and multilevel analysis reported here.

Permutation analysis

Permutation tests can be used to construct formal tests of significance of obtained analytical results (Lindgren, et al, 1996). 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. 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).

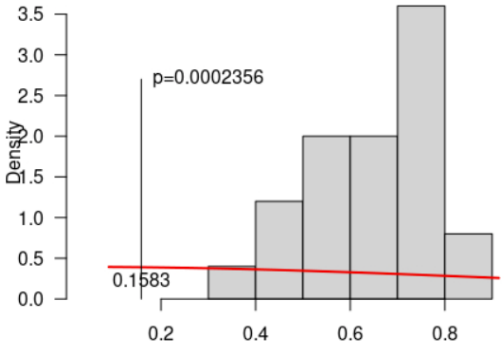
We first generate the actual modelling fitness to the permutation distribution:

```
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$ber[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$nber[model]
}
stopCluster(cl)
```

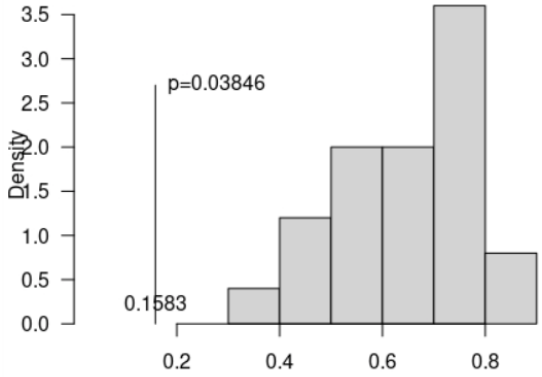
Using what statistical tests to compare the actual modelling fitness to the permutation distribution should be considered. We introduced 3 different types of tests here: i) A t-test; ii) A non-parametric test, where p-values were calculated from the ranking of fitness values, with analytical solution limited to 1/(*nperm*+1) (Westerhuis et al, 2010). iii) A parametric test, where p-values were calculated from the smoothed curve generated from fitness permutation distribution.

The assumption of gaussianity for a t-test 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. We can generate the permutation plots using different statistical tests by the codes below

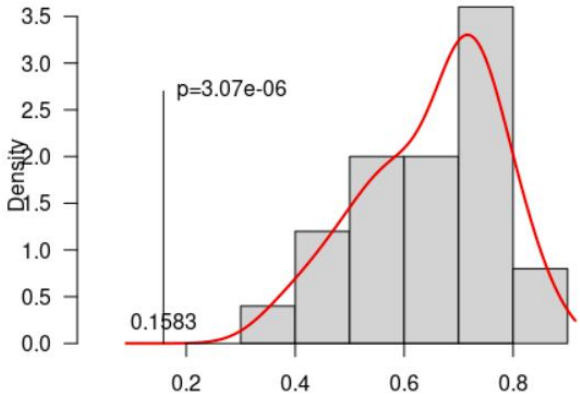
```
# Parametric permutation test significance based on student's t test:
plotPerm(actual = actualFit, distribution = permFit,type="t") # Look at histogram to assess whether Gaussian in shape
```




```
# Non-parametric permutation test significance basing on the rankings of fitness values:
plotPerm(actual = actualFit, distribution = permFit, type = 'rank') # If not Gaussian, make a non-parametric test or:
```




```
# Parametric test based on the smoothed curve generated from fitness permutation distribution_
plotPerm(actual = actualFit, distribution = permFit, type = 'smooth')
```



Above is the example for permutation tests in classification using RF. In regression, the actual and permuted fitness measures to be extracted should instead be `actual$fitMetric$Q2[model]` and `perm$fitMetric$Q2[model]` for PLS and `actual$fitMetric$Q2` and `perm$fitMetric$Q2` for EN, respectively.

For multilevel analysis, BER, number of misclassifications and Q2, could 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$ber[size]
}
```

Additional highlights

Our MUVR package also provides other functionalities that could facilitate users in regression, classification, and multilevel analysis. some highlights are:

MUVR-EN has integrated its core modelling with epidemiological covariate adjustment. Users could simply write keep=c("names of the variables") to include variables that will be treated as covariates to achieve covariate adjustments. The brief mechanism behind this is that these variables will always be manually set as having a non-zero beta-coefficient in the *nOuter* and *nRep* loops of MUVR-EN.

Our *one_hot_encoding()* function could transform categorical variables (number of classes=n) into n binary numeric variables with 0 and 1 classes. By this transformation, transformed categorical variables can be included as predictors in the core modeling methods that do not allow categorical variables (i.e. MUVR-PLS and MUVR-EN).

Functions *get_rmsep()*, *Q2_calculation()* and *getBER()* could, facilitate the calculation of prediction performance and could be used independently.

References

Shi, L., Westerhuis, J. A., Rosén, J., Landberg, R., & Brunius, C. (2019). Variable selection and validation in multivariate modelling. Bioinformatics, 35(6), 972-980.

Hanhineva, K., Brunius, C., Andersson, A., Marklund, M., Juvonen, R., Keski-Rahkonen, P., ... & Landberg, R. (2015). Discovery of urinary biomarkers of whole grain rye intake in free-living subjects using nontargeted LC-MS metabolite profiling. Molecular nutrition & food research, 59(11), 2315-2325.

Buck, M., Nilsson, L. K., Brunius, C., Dabiré, R. K., Hopkins, R., & Terenius, O. (2016). Bacterial associations reveal spatial population dynamics in Anopheles gambiae mosquitoes. Scientific Reports, 6(1), 1-9.

Westerhuis, J. A., van Velzen, E. J., Hoefsloot, H. C., & Smilde, A. K. (2010). Multivariate paired data analysis: multilevel PLSDA versus OPLSDA. Metabolomics, 6(1), 119-128.

Lindgren, F., Hansen, B., Karcher, W., Sjöström, M., & Eriksson, L. (1996). Model validation by permutation tests: applications to variable selection. Journal of Chemometrics, 10(5-6), 521-532.