# CMA package vignette

Martin Slawski *
Anne-Laure Boulesteix †

Department of Computer Science, Saarland University, D - 66041 Saarbruecken, Germany
Department of Medical Informatics, Biometry and Epidemiology (IBE) University of Munich , D-81377 Munich, Germany

## 1 Statistical background

For the last few years, microarray-based class prediction has been a major topic in statistics and machine learning. Traditional methods often yield unsatisfactory results or are even inapplicable in the $p \gg n$ setting. Hence, microarray studies have stimulated the development of new approaches and motivated the adaptation of known traditional methods to high-dimensional data.

Moreover, model selection and evaluation of prediction rules proves to be highly difficult in this situation for several reasons. Firstly, the hazard of overfitting, which is common to all prediction problems, is increased by high dimensionality. Secondly, the usual evaluation scheme based on the splitting into learning and test data sets often applies only partially in the case of small samples. Lastly, modern classification techniques rely on the proper choice of hyperparameters whose optimization is highly computer-intensive, especially in the case of high-dimensional data.

## 2 Class prediction based high-dimensional data with small samples

### 2.1 Settings

The classification problem can be briefly outlined as follows:

- we have a predictor space $\mathcal{X}$, here $\mathcal{X} \subseteq \mathbb{R}^p$
- we have a finite set of class labels $\mathcal{Y} = \{0, \ldots, K-1\}$, with $K$ denoting the total number of classes
- $P(\boldsymbol{x}, y)$ denotes the joint probability distribution on $\mathcal{X} \times \mathcal{Y}$
- we are given a finite sample $S = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n)\}$ of $n$ predictor-class pairs.

*ms@cs.uni-sb.de
†www.ibe.med.uni-muenchen.de/organisation/mitarbeiter/020_professuren/boulesteix/

The task is to find a decision function

$$\begin{aligned} \widehat{f}: \mathcal{X} &\to \mathcal{Y} \\ \boldsymbol{x} &\mapsto \widehat{f}(\boldsymbol{x}) \end{aligned}$$

(the $\widehat{\cdot}$ indicates estimation from the given sample $S$) such that the *generalization error*

$$R[f] = \mathbf{E}_{P(\boldsymbol{x},y)}[L(f(\boldsymbol{x}), y)] = \int_{\mathcal{X}\times\mathcal{Y}} L(y, f(\boldsymbol{x}))\, dP(\boldsymbol{x}, y) \tag{1}$$

is minimized. $L(\cdot,\cdot)$ is a suitable loss function, usually taken to be the indicator loss $(1,\ \text{if } f(\boldsymbol{x}) \neq y,\ 0, \text{otherwise})$.

## 2.2 Estimation of prediction accuracy

As we are only equipped with a finite sample $S$ and the underlying distribution is unknown, approximations to (**??**) have to be found. Its empirical counterpart

$$R[f]_{\text{emp}} = n^{-1} \sum_{i=1}^{n} L(y_i, f(\boldsymbol{x}_i)) \tag{2}$$

has a (usually large) negative bias for (**??**), thus model selection based on (**??**) leads to overfitting the sample $S$.
A first improvement involves a split of into parts $\mathcal{L}$ (learning sample), $\mathcal{T}$ (test sample) with the intention to separate model selection and -evaluation, by doing model selection only with $\mathcal{L}$ and evaluating the resulting decision function $f(\cdot)$ only on $\mathcal{T}$.

For microarray data, the sample sizes $n$ is usually very small, leading to serious problems when estimating prediction accuracy and when constructing a prediction rule based on the available data, a problem which is also related to model choice (see Section **??**).

When splitting the original data into two approximately equally sized data sets (learning set and test set), the performance of $f(\cdot)$ is strongly diminuished due to a further reduction of the sample size and the evaluation is unreliable and highly variant. Hence, alternative designs are needed.

In the package `CMA`, we pursue the following route for mitigating at least the second consequence, by resampling and aggregating:

- Generate $B$ splits of $S = (\mathcal{L}_b, \mathcal{T}_b), b = 1, \ldots, B$ into learning- and test sample
- Obtain $\widehat{f}_b$ from $\mathcal{L}_b,\ b = 1, \ldots, B$
- Setting $\mathcal{I}_b = \{i : y_i \notin \mathcal{L}_b\}$, we use

$$\widehat{\varepsilon} = \frac{1}{B} \sum_{b=1}^{B} \frac{1}{|\mathcal{I}_b|} \sum_{i\in\mathcal{I}_b} I(y_i \neq \widehat{f}_b(\boldsymbol{x}_i)) \tag{3}$$

  as estimator for (**??**).

The strategy is to reduce the variance of the error estimator by averaging.
If $n$ is large, this procedure will not improve much on a simple splitting.
As splitting rules, the function `GenerateLearningsets` implements:

- **Leaving-one-out cross-validation** :
  $\mathcal{T}_b$ consists only of one observation, this is repeated for each observation in $S$, so that $B = n$.

- $k$-**fold cross-validation** (`method = "CV", fold = , niter = `):
  $S$ is split into $k$ parts of equal size. For each iteration $b$, the $b$-th part is used as $\mathcal{T}_b$ and the union of the remaining parts as $\mathcal{L}_b$. Setting `fold = n` is equivalent to (method = "LOOCV"). As the splitting is not uniquely determined for `fold` $< n$, the whole procedure can be repeated `niter` times.

- Monte-Carlo-cross-validation (`method = "MCCV", fold=, ntrain=, niter= `):
  $B$=`niter` random learning samples of cardinality `ntrain` are generated.

- Bootstrap (`method = "bootstrap", ntrain = , niter = `):
  $B$=`niter` bootstrap samples of cardinality `ntrain` are used as learning samples.

Furhermore, *stratified sampling* is possible by setting the argument `strat = TRUE`. This implies, that for each $\mathcal{L}_b$, the proportion of the classes $\{0, \ldots, K-1\}$ is the same as for the full $S$. This option is very useful (and sometimes even necessary) in order to guarantee that each class is sufficiently often represented in each $\mathcal{L}_b$, in particular if there are classes that are small in size.
Benefits and drawbacks of above splitting rules are discussed in **?** and **?**.

## 2.3 Constructing a prediction rule

The second main issue is the construction of a appropriate prediction rule. In microarray data analysis, we have to deal with the $n \ll p$ situation, i.e. the number of predictors exceeds by far the number of observations. Some class prediction methods only work for the case $n \ll p$, e.g. linear or qudratic discriminant analysis which are based on the inversion of a matrix of size $p \times p$ and rank $n - 1$. In the $n \ll p$ setting, the hazard of overfitting is especially acute: perfect separation of the classes for a given sample based on a high number of predictors is always possible. However, the resulting classification rule may generalize poorly on independent test data.

There are basically three approaches to cope with the $n \ll p$ setting:

1. variable selection using, e.g., univariate statistical tests

2. regularization or shrinkage methods, such as the Support Vector Machine (**?**), $\ell_2$ or $\ell_1$ penalized logistic regression (**?**; **?**) or Boosting (**?**; **?**) from which some also perform variable selection

3. dimension reduction or feature extraction. Most prominent is the Partial Least Squares method (**?**).

Most classification methods depend on a vector of hyperparameters $\boldsymbol{\lambda}$ that have to be correctly chosen. Together with variable selection, this is part of the model selection and has thus to be performed separately for each learning sample $\mathcal{L}_b$ to avoid bias. An optimal vector of hyperparameters $\boldsymbol{\lambda}^{\text{opt}}$ is determined by defining a discrete set of values whose performance is then measured by cross-validation. This involves a further splitting step, which is sometimes called 'inner loop' or 'nested' cross-validation (**?**). More precisely, each learning set $\mathcal{L}_b$ is divided into $l = 1, \ldots, k$ parts such that the $l$-th part forms the test sample for the $l$-th (inner) iteration. Analogously to (**??**), this procedure can be used to derive an error estimator for each value on the grid of candiate hyperparameter values. The optimal vector $\boldsymbol{\lambda}^{\text{opt}}$ is chosen to minimize this

error criterion. Note that there are often several such minimizers. Furthermore, the minimizer found by this procedure can be relatively far away from the true minimizer, depending on how fine the discrete grid has been chosen.
The choice of the inner cross-validation scheme is difficult. With a high $k$, computation times soon become prohibitively high. With a low $k$, the size of $\mathcal{L}_{b_l},\ l = 1, \ldots, k$ is strongly reduced compared to the complete sample $S$, which may have an impact on the derived optimal parameter values. Nevertheless, nested cross-validation is commonly used in this context (**?**).

Considering the computational effort required for hyperparameter optimization and the small sample sizes, one may prefer class prediction methods that do not depend on many hyperparameters and/or behave robustly against changes of the hyperparameter values.

# 3 Overview of CMA features

In a nutshell, the package has the following features.

- It offers a uniform, user-friendly interface to a total of 21 classification methods (**??**), comprising classical approaches (such as discriminant analysis) as well as more sophisticated methods, e.g. Support Vector Machines (SVM) or boosting techniques. User-friendliness means that the input formats are uniform among different methods, that the user may choose between three different input formats and that the output is highly self-explicable and informative.
- Probability estimations for predicted observations are provided by most of the classifiers, with only a few exceptions. This is more informative than only returning class labels and enables a more precise comparison of different classifiers.
- It automatically generates learning samples as explained in section **??**, including the generation of stratified samples.
- Preliminary variable selection (if any) is performed for each iteration separately based on one of the following ranking procedures, using the method `GeneSelection`:
  - ordinary two-sample t.test (`method = "t.test"`)
  - Welch modification of the t.test (`method = "welch.test"`)
  - Wilcoxon rank sum test (`method = "wilcox.test"`)
  - F test (`method = "f.test"`) when $K > 2$
  - Kruskal-Wallis test (`method = "kruskal.test"`) when $K > 2$
  - 'moderated' t and F test, respectively, using the package `limma`(**?**) (`method = "limma"`)
  - One-step Recursive Feature Elimination (**?**) (`method = "rfe"`)
  - random forest variable importance measure (`method = "rf"`)
  - the Lasso (`method = "lasso"`)
  - the elastic net (`method = "elasticnet"`)
  - componentwise boosting (`method = "boosting"`)
  - the ad-hoc criterion used in **?**

For most methods, the implementation is very fast. The package `CMA` uses own functions instead of the pre-defined `R` functions. Additionally, the multi-class case is fully supported, even if the chosen `method` is not defined for it. The workaround is realized by using either a pairwise or a one-vs-all scheme.

- Hyperparameter tuning is carried out using the scheme outlined in section section **??**, for a *fixed* (sub)set of variables. It can be performed in a fully automatically manner using pre-defined grids. Alternatively, it can be completely customized by the user.
- The method `classification` enables the user to combine gene selection, hyperparameter tuning and class prediction into one single step. But each step can also be performed separately.
- Performance can be assessed using several performance measures which are commonly used in practice:
  - the misclassification rate
  - the sensitivity and specifity, for $K = 2$
  - the empirical area under the curve (AUC), for $K = 2$, if the class prediction method returns a probability,
  - the Brier score
  - the average probability of correct classification

  Each performance measure can be 1) averaged for all predictions globally, 2) averaged within each iteration first and then over all iterations or 3) averaged within each observation first and then over all observations. Based on the results, the function `obsinfo` can be used to identify observations that are frequenly misclassified (and are thus candidates for outliers).
- Comparison of the performance of several classifiers can be performed using one or several of the above performance measures. This comparison can be tabulated and visualized using the method `comparison`.
- Most results can quickly be summarized and visualized using pre-defined convenience methods, for example:
  - `plot,cloutput-method` produces a probability plot, also known as 'voting plot'
  - `plot,genesel-method` visualizes variable importance via a barplot
  - `roc,cloutput-method` draws the empirical ROC curve
  - `toplist,genesel-method` shows the most important variables
  - `summary,evaloutput-method` Makes a summary out of iteration- or observationwise performance measures
- The implementation is fully organized in `S4` classes, thus making the extension of `CMA` very easy. In particular, own classification methods can easily be integrated if they return a proper object of class `cloutput`.
- The class prediction methods implemented in `CMA` are summarized in Table **??**.

| **method name** | `CMA` **function name** | **Package** | **Reference** |
|---|---|---|---|
| Componentwise Boosting | `compBoostCMA` | `CMA` | ? |
| Diagonal Discriminant Analysis | `dldaCMA` | `CMA` | ? |
| Elastic Net | `ElasticNetCMA` | `glmpath` | ? |
| Fisher's Discriminant Analysis | `fdaCMA` | `CMA` | ? |
| Flexible Discriminant Analysis | `flexdaCMA` | `mgcv` | ? |
| Tree-based Boosting | `gbmCMA` | `gbm` | ? |
| $k$-nearest neighbours | `knnCMA` | `class` | ? |
| Linear Discriminant Analysis $*$ | `ldaCMA` | `MASS` | ? |
| Lasso | `LassoCMA` | `glmpath` | ? |
| Feed-Forward Neural Networks | `nnetCMA` | `nnet` | ? |
| Probalistic nearest neighbours | `pknnCMA` | `CMA` | – |
| Penalized Logistic Regression | `plrCMA` | `CMA` | ? |
| Partial Least Squares $\star$ + $*$ | `pls_ldaCMA` | `plsgenomics` | ? |
| $\star$ + logistic regression | `pls_lrCMA` | `plsgenomics` | ? |
| $\star$ + Random Forest | `pls_rfCMA` | `plsgenomics` | ? |
| Probabilistic Neural Networks | `pnnCMA` | `CMA` | ? |
| Quadratic Discriminant Analysis $*$ | `qdaCMA` | `MASS` | ? |
| Random Forest | `rfCMA` | `randomForest` | ? |
| PAM | `scdaCMA` | `CMA` | ? |
| Shrinkage Discriminant Analysis | `shrinkldaCMA` | `CMA` | – |
| Support Vector Machine | `svmCMA` | `e1071` | ? |

# 4 Comparison with existing packages

The idea of an interace for the integration of classification methods for microarray data is not new and we here argue why `CMA` can be a significant improvement with respect to the following aspects: standardized and reproducible analysis, neutral comparisons of existing methods, comfortable use.

The `CMA` package shows similarities to the `Bioconductor` package `MLInterfaces` standing for 'An interace to various machine learning methods' (**?**), see also the Bioconductor textbook for a presentation of an older version.

Contrary to `CMA`, `MLInterfaces` also offers access to popular 'unsupervised learning' methods such as clustering, independent component analysis etc. that can be beneficial for exploratory analyses as well as for revealing classes in a preparatory step preceding supervised classification. The package architecture is very similar the `CMA` structure in the sense that wrapper functions are used to call classification methods from other packages.

Up to now, `CMA` includes more predefined features than `MLInterfaces` as far as variable selection, hyperparameter tuning, classifier evaluation and comparison are concerned. While the method `xval` is flexible for experienced users, it provides only LOOCV or 'leave-one-group out' as predefined options. As this package adresses also unexperienced users, we decided to include the most common validation schemes in a standardized manner. As consequence, we additionally hope to increase reproducibility of results which is a major concern in statistics in general and for microarray data analysis in particular.

In the current version, variable selection can also be carried separately for each different learning set, but this seems not to be a standard procedure. In the examples presented in the book mentionned above, variable selection is only performed once using the *complete* sample $S$ although this procedure is widely known to yield optimistically biased results (**?**).

Moreover, hyperparameter tuning is completely missing in `MLInterfaces`. In our opinion, this makes the objective comparison of different class prediction methods difficult. If tuning is ignored, simpler methods without (or with few) tuning parameters tend to perform seemingly better than more complex algorithms.

We 'borrowed' from `MLInterface` some ideas regarding the selection of classification methods (section **??**) and some functionalities such as the variable importance plot, or the `Planarplot`.

We would also like to mention the package `e1071` (**?**) whose `tune` function served as the basic idea for the `CMA` tuning functionalities.

The package `MCRestimate` (**?**) emphasizes very similar aspects as `CMA`, focussing on the estimation of misclassification rates and cross-validation for model selection and evaluation. It is (to our knowledge) the only `Biconductor` package that supports hyperparameter tuning, but obviously referring to the function `e1071:::tune`. Compared to `CMA`, it is a bit less comprehensive, in particular with respect to variable selection. Moreover, the package structure is, to our opinion, less stringent than that of CMA, mainly because it lacks a class structure (neither `S3` nor `S4` is used).

# 5 Example 1: Focussing on one method

The following two sections demonstrate the usual workflow when using `CMA` and its essential features, methods and objects. While this section focusses on optimizing and evaluating one method, the following section is devoted to classifier comparison.

We use the famous leukaemia dataset of **?** as a data example. The sample consists of 38 observations in total, from which 27 belong to class 0 (acute lymphoblastic leukemia) and 11 to class 1 (acute myeloid leukemia). We start by loading and the dataset and extracting gene expression and class labels, respectively.

```
> data(golub)
> golubY <- golub[,1]
> golubX <- as.matrix(golub[,-1])
>
```

Following the approach described in section **??**, we generate several learning samples by different splitting rules, taking into account that each of them has advantages and disadvantages. Learning samples are generated by the function `GenerateLearningsets`, which returns an object of class `learningsets`:

```
> loodat <- GenerateLearningsets(y=golubY, method ="LOOCV")
> class(loodat)
> getSlots(class(loodat))
> show(loodat)
>
```

For five-fold cross-validation, we use the following commands:

```
> set.seed(321)
> fiveCVdat <- GenerateLearningsets(y=golubY, method = "CV", fold = 5, strat = TRUE)
>
>
```

Note that stratified learning sets are generated by setting the argument `strat = TRUE`. The random seed should be set for reproducibility. We can proceed analogously for Monte-Carlo cross-validation and bootstrap:

```
> set.seed(456)
> MCCVdat <- GenerateLearningsets(y=golubY, method = "MCCV", niter = 3,
+                                 ntrain = floor(2/3*length(golubY)), strat = TRUE)
> set.seed(651)
> bootdat <- GenerateLearningsets(y=golubY, method = "bootstrap", niter = 3, strat = TRUE)
>
>
```

In this example, we choose the Support Vector Machine with linear kernel as classification method. Variable selection is not strictly necessary, but it has empirically been shown that the performance of the SVM method can significantly be improved when noise features are removed (**?**). For simplicity, we choose the distribution free Wilcoxon-Test to rank the variables, separately for each learning sample :

```
> varsel_fiveCV <- GeneSelection(X = golubX, y=golubY, learningsets = fiveCVdat, method = "wilcox.test"
> varsel_MCCV <- GeneSelection(X = golubX, y=golubY, learningsets = MCCVdat, method = "wilcox.test")
> varsel_boot <- GeneSelection(X = golubX, y=golubY, learningsets = bootdat, method = "wilcox.test")
>
>
```

Now let us have a closer look at `varsel_fiveCV`. The `toplist` methods provides easy access to the top-ranked variables:

```
> show(varsel_fiveCV)
> toplist(varsel_fiveCV, iter=1)
> seliter <- numeric()
> for(i in 1:5)
+ seliter <- c(seliter, toplist(varsel_fiveCV, iter = i, top = 10, show = FALSE)$index)
> sort(table(seliter), dec = TRUE)
>
```

We see that just 1 variable is among the top 10 in every learning sample.

The next step is hyperparameter tuning. It is well-known that the SVM needs much tuning (**?**) and that its performance can decrease drastically without tuning. For simplicity, we use a linear kernel, thus avoiding the tuning of kernel parameters. The choice of the parameter $C$ in the primal objective of the SVM

$$P(\boldsymbol{w}) = \|\boldsymbol{w}\|^2 + C\sum_{i=1}^{n}\xi_i,\ C > 0$$

has still to be done. $\boldsymbol{w}$ denotes the weight vector of the maximum margin hyperplane, while the $\{\xi_i\}_{i=1}^n$ quantify the amount of violation of this hyperplane by the learning sample. Increasing $C$ penalizes violations more severely, forcing the hyperplane to separate the learning sample (and thus probably producing overfitting).
In order to find an appropriate value for $C$ (which corresponds to the argument `cost` in `e1071:::svm`), we take the best 100 genes from the previous variable ranking step. As hyperparameter tuning is very time-intensive, we use the five-fold CV procedure only in this demonstrating example. The resulting best value for `cost` will then also be used for the other other CV procedures. In this example, we consider five candidate

values: 0.1,1,10,100,200. Due to the nested cross-validation procedure, the SVM is trained $3 \times 5 \times 5 = 75$ times (!) in the following example. Three is to the number of inner cross-validation folds, the first five is the number of candidate values and the second five is the number of outer cross-validation folds.

```
> set.seed(351)
> tuningstep <- CMA:::tune(X = golubX, y=golubY, learningsets = fiveCVdat,
+                          genesel = varsel_fiveCV, nbgene = 100, classifier = svmCMA,
+                          grids = list(cost = c(0.1, 1, 10, 100, 200)),probability=T)
>

> show(tuningstep)
> unlist(best(tuningstep))
>
```

The results seem to equivocally favour `cost=0.1`. However, if we visualize the tuning results, we see that almost all tuning parameter values perform quite well. Please note that CMA will by default choose the smallest value if performances of several values are equal:

```
> par(mfrow = c(2,2))
> for(i in 1:4)
+ plot(tuningstep, iter = i, main = paste("iteration", i))
```

Nonetheless, we will choose `cost=0.1` since it produced good results.
We can now turn the attention to class prediction:

```
> #class_loo <- classification(X = golubX, y=golubY, learningsets = loodat,
> #                           genesel = varsel_loo, nbgene = 100, classifier = svmCMA,
> #                           cost = 100)
>
> class_fiveCV <- classification(X = golubX, y=golubY, learningsets = fiveCVdat,
+                          genesel = varsel_fiveCV, nbgene = 100, classifier = svmCMA,
+                          cost = 0.1,probability=T)
> class_MCCV <- classification(X = golubX, y=golubY, learningsets = MCCVdat,
+                          genesel = varsel_MCCV, nbgene = 100, classifier = svmCMA,
+                          cost = 0.1,probability=T)
> class_boot <- classification(X = golubX, y=golubY, learningsets = bootdat,
+                          genesel = varsel_boot, nbgene = 0.1, classifier = svmCMA,
+                          cost = 100,probability=T)
>
>
```

The results of `classification` are lists where each element is an object of class `cloutput`, generated for each learning sample. For visualization purpose, we use the `join` function to combine the single list elements into 'big' objects. We first put the results from the four splitting schemes into one list and then use `lapply()`:

```
> resultlist <- list(class_fiveCV, class_MCCV, class_boot)
> result <- lapply(resultlist, join)
>
```

The probability (or voting) plot is one of the most popular visualizatio method in microarray-based classification:

```
> schemes <- c("five-fold CV", "MCCV", "bootstrap")
> par(mfrow = c(3,1))
> for(i in seq(along = result))
+  plot(result[[i]], main = schemes[i])
>
```

`ftable` applied to objects of class `cloutput` yields confusion matrices:

```
> invisible(lapply(result, ftable))
>
```

`roc` draws simple ROC cuves:

```
> par(mfrow = c(2,2))
> for(i in seq(along = result)) roc(result[[i]])
>
```

We can now `join` again to aggregate over the different splitting rules:

```
> totalresult <- join(result)
> ftable(totalresult)
>
```

Confusion matrices implicity quantify performance via misclasification. For more advanced performance evaluation, one can use `evaluation`. Note that the input has to be a list (and not an object of class `cloutput`). For the Monte-Carlo cross-validation scheme, we have:

```
> av_MCCV <- evaluation(class_MCCV, measure = "average probability")
> show(av_MCCV)
> boxplot(av_MCCV)
> summary(av_MCCV)
>
```

`measure = "average probability"` stands for the average predicted probability for the correct class, or formally:

$$\sum_{b=1}^{B} \sum_{i \in \mathcal{L}_b} \sum_{k=0}^{K-1} I(y_i = k) \widehat{p}(y_i = k | \boldsymbol{x}_i),$$

with $\widehat{p}(k|\boldsymbol{x})$ denoting the condtional predicted probability for class $k$, given $\boldsymbol{x}$.

By default, the evalution scheme is iterationwise, but it can also be done observationwise :

```
> av_obs_MCCV <- evaluation(class_MCCV, measure = "average probability", scheme = "obs")
> show(av_obs_MCCV)
>
```

One might also wonder which observations are misclassified very often. To find it out, one can use:

```
> obsinfo(av_obs_MCCV, threshold = 0.6)
>
```

# 6 Example 2: classifier comparison

CMA implements a complete bundle of methods based on the principle of discriminant analysis. Here, we compare six of them: diagonal-, linear- and quadratic discrimininant analysis, discriminant analysis by Fisher, shrunken centroids discriminant analyis (also known as PAM) and Partial Least Squares followed by linear discriminant analysis, , applied to the small blue round cell tumour dataset of **?** which comprises 65 samples from four tumour classes.
From a theoretical point of view, linear-, quadratic- and Fisher's discriminant analysis are apriori inferior due to the fact that they do not work in the $p \gg n$ without variable selection. Shruken centroids discriminant analysis is assumed to work better than the simple diagonal discriminant analysis because it can 'shrink-out' noise variables. Partial Least Squares is also expected to work well.
But let as see how this looks in pratice. As data basis, we will use (stratified) five-fold cross-validation, repeated ten times in order to achive more stable results.

```
> data(khan)
> khanY <- khan[,1]
> khanX <- as.matrix(khan[,-1])
> set.seed(27611)
> fiveCV5iter <- GenerateLearningsets(y=khanY, method = "CV", fold = 5, niter = 5, strat = TRUE)
>
>
```

Contrary to the step-by-step procedure in the previous example, we will here always use the flexible method `classification`. We start with diagonal discriminant analysis which neither needs variable selection nor tuning.

```
> class_dlda <- classification(X = khanX, y=khanY, learningsets = fiveCV5iter,
+                              classifier = dldaCMA)
>
>
```

We now rank the variables (genes) according to the t statistic as basis for variable selection necessary for linear-, quadratic- and Fisher's linear discriminant analysis:

```
> genesel_da <- GeneSelection(X=khanX, y=khanY, learningsets = fiveCV5iter,
+                             method = "t.test", scheme = "one-vs-all")
>
```

For the class prediction that follows, we pass the generated learning sets, the gene rankings (`genesel_da`) to `classification`. The number of genes that are retained is specified by the argument `nbgene`.

```
> class_lda <- classification(X = khanX, y=khanY, learningsets = fiveCV5iter,
+                             classifier = ldaCMA, genesel= genesel_da,
+                             nbgene = 10)
> class_fda <- classification(X = khanX, y=khanY, learningsets = fiveCV5iter,
+                             classifier = fdaCMA, genesel = genesel_da,
+                             nbgene = 10, comp = 2)
> class_qda <- classification(X = khanX, y=khanY, learningsets = fiveCV5iter,
+                             classifier = qdaCMA, genesel = genesel_da,
+                             nbgene = 1)
>
```

Shrunken centroids discriminant analysis does not do variable selection, but hyperparameter tuning for the shinkage intensity. We here use the pre-specified grid by setting `tuninglist` to an empty list.

```
> set.seed(876)
> class_scda <- classification(X = khanX, y=khanY, learningsets = fiveCV5iter,
+                              classifier = scdaCMA, tuninglist = list(grids = list()))
>
```

At last, we use partial least squares (with two components, which is the default).

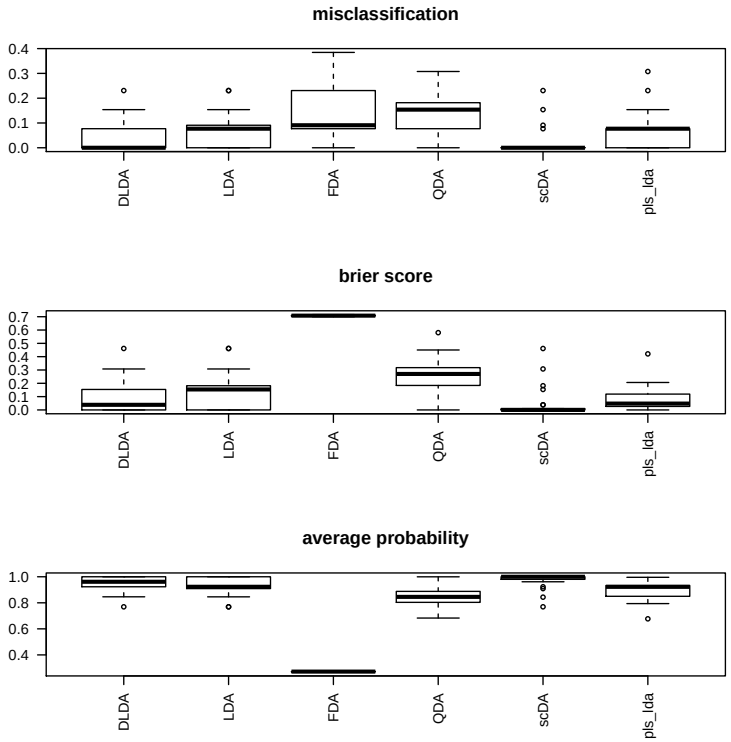
```
> class_plsda <- classification(X = khanX, y=khanY, learningsets = fiveCV5iter,
+                              classifier = pls_ldaCMA)
>
```

A comparison can be performed very quickly, needing only a very few lines. The method `comparison` does the whole job:

```
> dalike <- list(class_dlda, class_lda, class_fda, class_qda, class_scda, class_plsda)
> par(mfrow = c(3,1))
> comparison <- compare(dalike, plot = TRUE, measure = c("misclassification", "brier score", "average p
> print(comparison)

        misclassification brier.score average.probability
DLDA           0.03804196  0.08905717           0.9456035
LDA            0.06881119  0.13624745           0.9315248
FDA            0.14069930  0.70802926           0.2717240
QDA            0.15328671  0.26916266           0.8485696
scDA           0.02209790  0.04806483           0.9731392
pls_lda        0.05958042  0.08822419           0.8950797

>
```


misclassification
0.4
0.3
0.2
0.1
0.0
DLDA
LDA
FDA
QDA
scDA
pls_lda


brier score
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
DLDA
LDA
FDA
QDA
scDA
pls_lda


average probability
1.0
0.8
0.6
0.4
DLDA
LDA
FDA
QDA
scDA
pls_lda