clustlasso vignette

Introduction

In this document, we illustrate the use of the clustlasso package. For this purpose, we run a classical process of cross-validation / model selection / test-set prediction using both a standard lasso and a cluster-lasso approach, and compare the results obtained.

We first load the package, set the random seed and load a dataset taken from from Eyre, 2017. This dataset is included in the package for illustration purposes. It contains the k-mer profiles of 664 *Neisseria gonorrhoeae* strains, stored in the matrix X, and a binary phenotype, stored in the vector y, encoding their susceptibility or non-susceptibility to the cefixime antibiotic.

```
# load package
suppressWarnings(suppressMessages(library(clustlasso)))
# specify / set random seed
seed = 42
set.seed(seed)
# load example dataset
input.file = system.file("data", "NG-dataset.Rdata", package = "clustlasso")
load(input.file)
```

We then split the dataset to into a training and a test dataset, respectively made of 80% and 20% of the entire dataset. Note that we consider that we have at our disposal a covariate defining sub-groups in our population, stored here in the `pop_structure` column of the `meta` data-frame that is part of the dataset. For this example, the sub-groups simply correspond to the geographic origin of the samples, but in practice one could consider alternative strategies based on actual population structure covariates (e.g., sequence types - STs) to define more or less challenging settings (e.g., maintaining the STs prevalence or considering distinct STs between the training and test sets).

```
# pick 20% for test
test.frac = 0.2
# stratify by origin / population structure
ind.by.struct = split(seq(nrow(meta)), meta$pop_structure)
ind.sample = sapply(ind.by.struct, function(x) {
    sample(x, round(test.frac * length(x)))
})
ind.test = unlist(ind.sample)
# split
X.test = X[ind.test, ]
y.test = y[ind.test]
meta.test = meta[ind.test, ]
X.train = X[-ind.test, ]
y.train = y[-ind.test]
meta.train = meta[-ind.test, ]
```

Standard Lasso process

We first run a standard lasso process, which includes :

1. carrying out a cross-validation study
2. selecting the best model
3. making predictions on the test set and computing the predictive performance

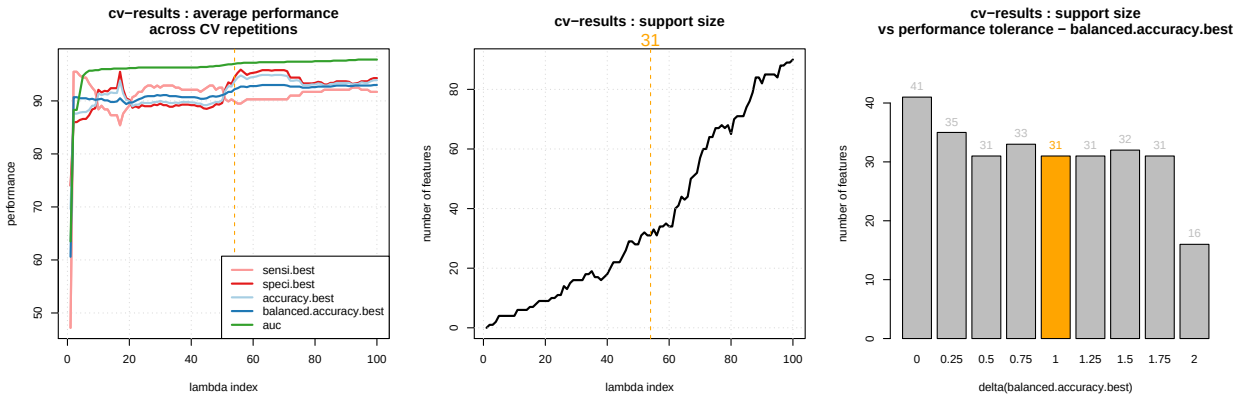
1. Cross-validation process

We run a cross-validation process, using the **lasso_cv()** function included in the package. The `n.folds`, `n.repeat` and `n.lambda` arguments respectively define the number of folds to consider, the number of times the cross-validation is to be repeated and the number of candidate values to consider for the lasso regularization parameter. Note that a covariate defining sub-groups in our population, stored here in the `pop_structure` column of the `meta` data-frame that is part of the toy dataset, can be specified using the `subgroup` argument, to ensure that the folds are stratified by outcomes and sub-groups.

```
# specify cross-validation parameters
n.folds = 10
n.lambda = 100
n.repeat = 3
# run cross-validation process
cv.res.lasso = lasso_cv(X.train, y.train, subgroup = meta.train$pop_structure,
    n.lambda = n.lambda, n.folds = n.folds, n.repeat = n.repeat,
    seed = seed, verbose = FALSE)
```

We can then show the results using the **show_cv_overall()** function of the package, as follows. The `modsel.criterion` and `best.eps` arguments are used to actually select the best model in the subsequent step, as described below. Specifying them at this stage allows to explicitly identify this model (in orange) in the figures shown below.

```
par(mfcol = c(1, 3))
show_cv_overall(cv.res.lasso, modsel.criterion = "balanced.accuracy.best",
    best.eps = 1)
```



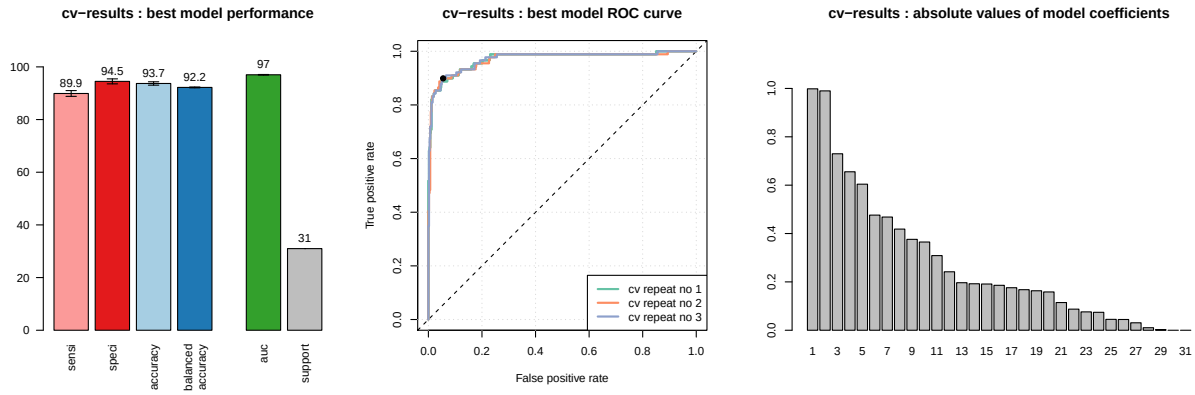
2. Selecting the best model

Our model selection strategy relies on a performance indicator (through the `modsel.criterion` argument) and a tolerance criterion (through the `best.model` argument): the best model is defined as the sparsest

model allowing to reach the highest value of the model selection criterion, up to the specified tolerance. The `modsel.criterion` can be defined as the area under the ROC curve (`modsel.criterion="auc"`) or the so-called "best" balanced-accuracy (`modsel.criterion="balanced.accuracy.best"`), which is defined as the mean sensibility and specificity obtained when the ROC curve is closest to the optimal model.

We can first show a summary of the best model using the **show_cv_best()** function of the package, as follows.

```
layout(matrix(c(1, 2, 3), nrow = 1, byrow = TRUE), width = c(0.3,
    0.3, 0.4), height = c(1))
perf.best.lasso = show_cv_best(cv.res.lasso, modsel.criterion = "balanced.accuracy.best",
    best.eps = 1, method = "lasso")
```



Note that this function returns a vector containing the cross-validation performance of the selected model.

```
# print cross-validation performance of best model
print(perf.best.lasso)
```

```
##             sensi.best             speci.best          accuracy.best
##                 "89.9"                 "94.5"                 "93.7"
## balanced.accuracy.best                    auc                support
##                 "92.2"                   "97"                   "31"
##       thresh-best_mean        thresh-best_all
##                "0.224"    "0.256-0.222-0.194"
```

We then extract the model into a dedicated object using the **extract_best_model()** function. This object is a list containing the indices of the features entering the model, their associated coefficients, the intercept of the model and the decision threshold considered by default. This threshold is defined as the threshold that allowed to reach, on the ROC curve of the best model, the sensibility / specificity trade-off closest to the optimal model in the cross-validation study (show as the black dot in the middle graph above).

```
best.model.lasso = extract_best_model(cv.res.lasso, modsel.criterion = "balanced.accuracy.best",
    best.eps = 1)
```

3. Making predictions and measuring performance

Finally, we can use the selected model to make predictions on the test set and assess predictive performance, using the **predict_clustlasso()** and **compute_perf()** functions.

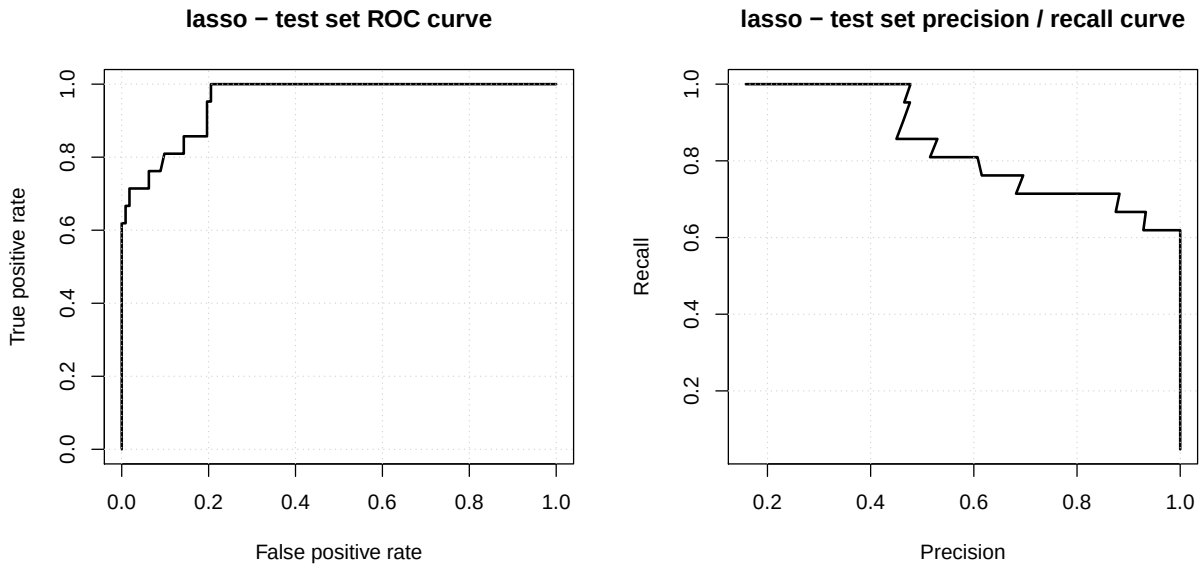
```
# make predictions
preds.lasso = predict_clustlasso(X.test, best.model.lasso)
# compute performance
```

```r
perf.lasso = compute_perf(preds.lasso$preds, preds.lasso$probs,
    y.test)
# print
print(t(perf.lasso$perf))
```

```
##                   [,1]
## accuracy          91.0
## sensi             71.4
## speci             94.6
## balanced.accuracy 83.0
## auc               95.6
## recall            71.4
## precision         71.4
## f1                71.4
```

Note that the object returned by the **compute_perf()** function contains standard performance indicators, as shown above, as well as ROC and precision/recall curves, as shown below.

```r
par(mfcol = c(1, 2))
plot(perf.lasso$roc.curves[[1]], lwd = 2, main = "lasso - test set ROC curve")
grid()
plot(perf.lasso$pr.curves[[1]], lwd = 2, main = "lasso - test set precision / recall curve")
grid()
```



Cluster-Lasso process

We then run the same process using the cluster-lasso model.

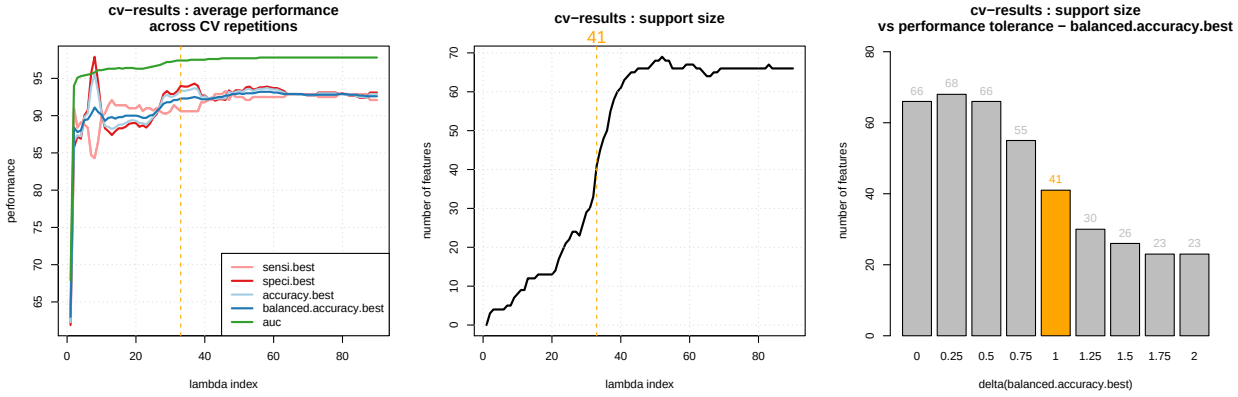
1. Cross-validation process

We run a similar cross-validation process, using the **clusterlasso_cv()** function included in the package. In practice, we need to specify the values to consider for the screening threshold (`screen.thresh` argument) and the clustering threshold (`clust.thresh` argument).

```
# specify screening and clustering thresholds
screen.thresh = 0.95
clust.thresh = 0.95
# run cross-validation process
cv.res.cluster = clusterlasso_cv(X.train, y.train, subgroup = meta.train$pop_structure,
    n.lambda = n.lambda, n.folds = n.folds, n.repeat = n.repeat,
    seed = seed, screen.thresh = screen.thresh, clust.thresh = clust.thresh,
    verbose = FALSE)
```

We can then show the results using the **show_cv_overall()** function of the package, as before.

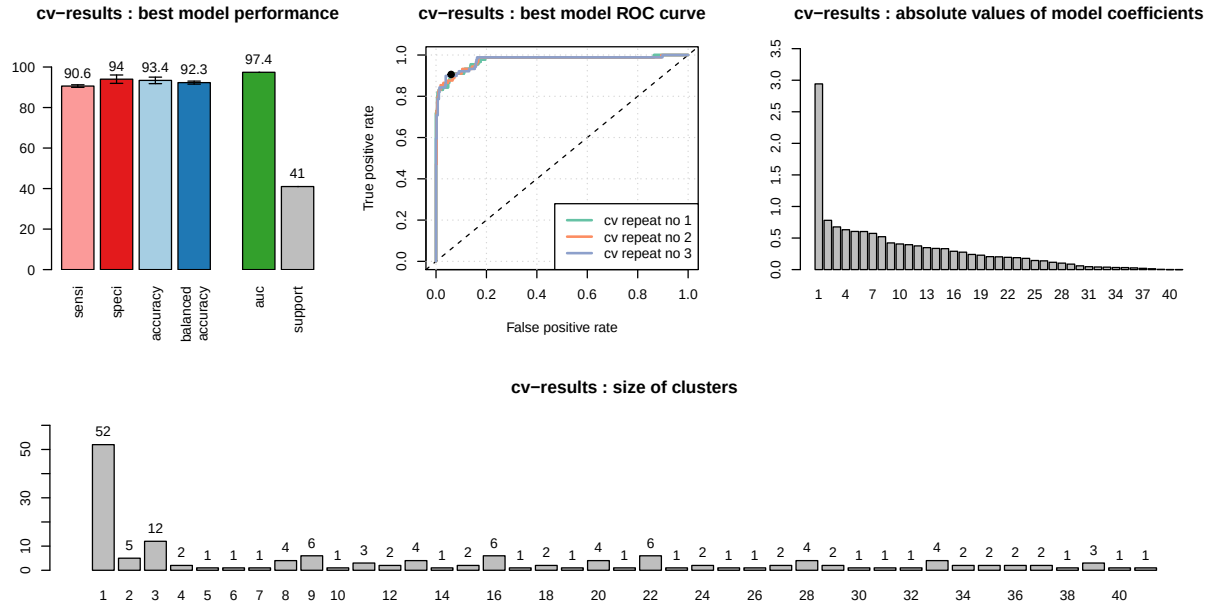
```
par(mfcol = c(1, 3))
show_cv_overall(cv.res.cluster, modsel.criterion = "balanced.accuracy.best",
    best.eps = 1)
```



2. Selecting the best model

We adopt the same model selection strategy, and first show a summary of the best model using the **show_cv_best()** function of the package. Note that this figure contains an additional plot, representing the number of features associated to each selected cluster.

```
layout(matrix(c(1, 2, 3, 4, 4, 4), nrow = 2, byrow = TRUE), width = c(0.3,
    0.3, 0.4), height = c(0.6, 0.4))
perf.best.cluster = show_cv_best(cv.res.cluster, modsel.criterion = "balanced.accuracy.best",
    best.eps = 1, method = "clusterlasso")
```



We emphasize that this function returns a vector containing the cross-validation performance of the selected model.

```
# print cross-validation performance of best model
print(perf.best.cluster)
```

```
##             sensi.best              speci.best          accuracy.best
##                  "90.6"                    "94"                  "93.4"
## balanced.accuracy.best                     auc                 support
##                  "92.3"                  "97.4"                    "41"
##       thresh-best_mean         thresh-best_all
##                  "0.195"    "0.195-0.147-0.244"
```

We then extract the model into a dedicated object using the **extract_best_model()** function. This object contains the same information as for a lasso model, and an additional field providing the clusters definition.

```
best.model.cluster = extract_best_model(cv.res.cluster, modsel.criterion = "balanced.accuracy.best",
    best.eps = 1, method = "clusterlasso")
```

3. Making predictions and measuring performance

Finally, we can use the selected model to make predictions on the test set and assess predictive performance, using the **predict_clustlasso()** and **compute_perf()** functions.

```
# make predictions
preds.cluster = predict_clustlasso(X.test, best.model.cluster,
    method = "clusterlasso")
# compute performance
perf.cluster = compute_perf(preds.cluster$preds, preds.cluster$probs,
    y.test)
# print
print(t(perf.cluster$perf))
```

```
##                   [,1]
```

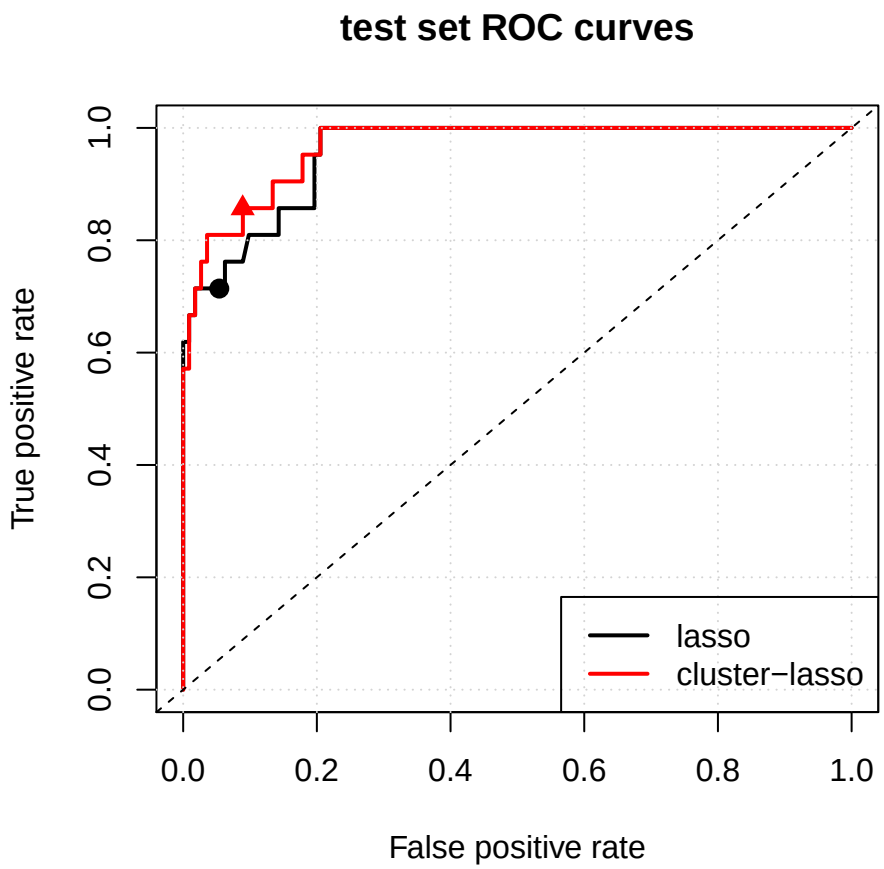
```
## accuracy          90.2
## sensi             85.7
## speci             91.1
## balanced.accuracy 88.4
## auc               96.6
## recall            85.7
## precision         64.3
## f1                73.5
```

Comparing the results

We can finally compare the results obtained by the standard lasso and cluster-lasso approaches in terms of predictive performance measured on the test set, and in terms of signatures.

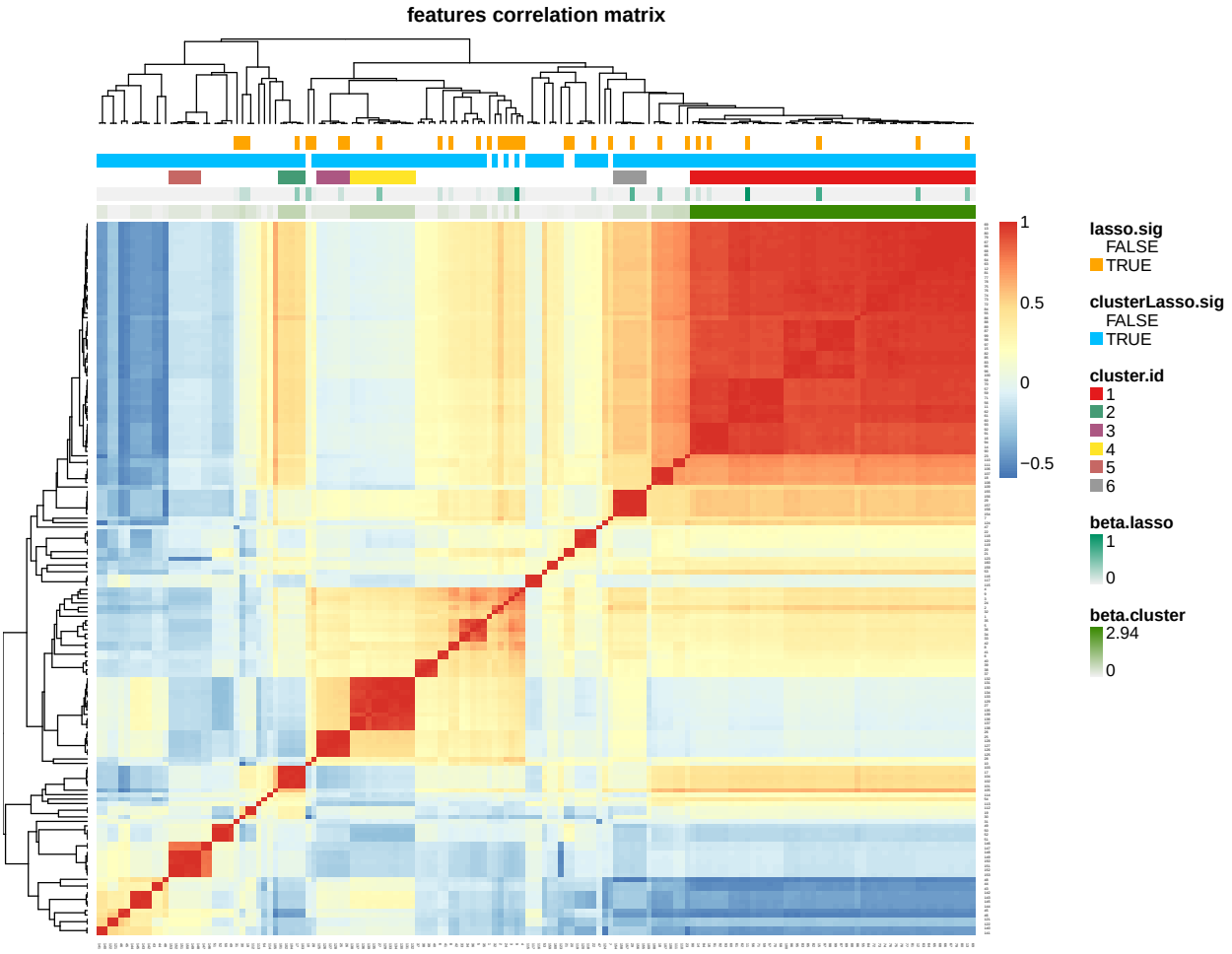
We first plot the ROC curves measured on the test set. Note that the filled dot and triangles represent the "best" sensitivities and specificities, obtained according to the decision thresholds optimized during the cross-validation process.

```
plot(perf.lasso$roc.curves[[1]], lwd = 2, main = "test set ROC curves")
points(1 - (perf.lasso$perf$speci)/100, perf.lasso$perf$sensi/100,
    pch = 19, col = 1, cex = 1.25)
plot(perf.cluster$roc.curves[[1]], lwd = 2, col = 2, add = TRUE)
points(1 - (perf.cluster$perf$speci)/100, perf.cluster$perf$sensi/100,
    pch = 17, col = 2, cex = 1.25)
grid()
abline(0, 1, lty = 2)
legend("bottomright", c("lasso", "cluster-lasso"), col = c(1,
    2), lwd = 2)
```



Finally, the **heatmap_correlation_signatures()** function of the package allows to compare the level of correlation existing among features involved in both models.

```
heatmap_correlation_signatures(X, best.model.lasso, best.model.cluster,
    clust.min = 5, plot.title = "features correlation matrix")
```



This figure represents as a heatmap the correlation matrix built from the features selected by the lasso and the cluster-lasso (identified by the orange and blue bars shown above the heatmap, respectively). The corresponding (absolute) values of model coefficients are represented by green bars. The major clusters (involving more than `clust.min` features, set here to 5) of the cluster-lasso signatures are identified by a dedicated color ranging from red to grey.