Working with the mlr package

To become familiar with the mlr package and its widespread functionality, I created an Rmarkdown document with basic and advanced features from the mlr tutorial on mostly regression and classification problems with data available in R.

Getting started

The first step is to create a **task**; this can be a classification, regression, survival, cluster, cost-sensitive classification or multilabel task. For example, let us start with the `iris` data set from which we wish to predict the `Species` based on features comprised of width and length measurements of sepals and petals.

```
task <- makeClassifTask(data = iris, target = "Species")
```

There exist other options that can be set when calling the task function like, for instance, weights and blocking. The latter is for observations that are required to be considered together such that when resampling or cross-validating, grouped observations are included either all in the train or all in the test set.

Next, we choose a learner. Here, we choose to learn based on a classification tree fitted through the `rpart` function.

```
lrn <- makeLearner("classif.rpart")
```

We can get a description of all possible parameter settings for a learner using `?getParamSet(lrn)`:

```
getParamSet(lrn)
```

```
##                     Type len  Def   Constr Req Tunable Trafo
## minsplit         integer   -   20 1 to Inf   -    TRUE     -
## minbucket        integer   -    - 1 to Inf   -    TRUE     -
## cp               numeric   - 0.01   0 to 1   -    TRUE     -
## maxcompete       integer   -    4 0 to Inf   -    TRUE     -
## maxsurrogate     integer   -    5 0 to Inf   -    TRUE     -
## usesurrogate    discrete   -    2    0,1,2   -    TRUE     -
## surrogatestyle  discrete   -    0      0,1   -    TRUE     -
## maxdepth         integer   -   30  1 to 30   -    TRUE     -
## xval             integer   -   10 0 to Inf   -   FALSE     -
## parms            untyped   -    -        -   -    TRUE     -
```

Further, we can create a description object for a resampling startegy using `makeResampleDesc`. For example, we may wish to carry out a 3-fold cross-validation of `rpart` on `iris`.

```
cv3f <- makeResampleDesc("CV", iters = 3, stratify = TRUE )
```

Finally, we can fit the model specified by `lrn` on the `task` and calculate predictions and performance measures for all training and all test sets specified by the above resampling description.

```
set.seed(123)
r <- resample(lrn, task, cv3f)
r$aggr
```

```
## mmce.test.mean
##     0.07295185
```

The mean misclassification error is given by 0.0729518. The default measure for a classification task is the misclassification error, `mmce` but it is possible to choose a different measure, say *accuracy*, `acc` which is given by 1 - `mmce`. We will see later that we can also define our own measures.

```
set.seed(123)
r <- resample(lrn, task, cv3f, measures = acc)
r$aggr
```

```
## acc.test.mean
##     0.9270482
```

Basics

Tasks

As previously mentioned, depending on the type of problem at hand, one can define an appropriate task. We have seen an instance of a classification problem above, let us now look at a supervised regression problem using the `BostonHousing` data. By printing task, we get some information on the task object we have just created and the associated data.

```
data(BostonHousing, package = "mlbench")
task <- makeRegrTask(data = BostonHousing, target = "medv")
print(task)
```

```
## Supervised task: BostonHousing
## Type: regr
## Target: medv
## Observations: 506
## Features:
## numerics  factors  ordered
##       12        1        0
## Missings: FALSE
## Has weights: FALSE
## Has blocking: FALSE
```

Modifying a task Once the task is created, it can be modified in several ways. For example, the function `subsetTask` allows the selection of certain observations and/or features.

```
names(BostonHousing)
```

```
##  [1] "crim"    "zn"      "indus"   "chas"    "nox"     "rm"      "age"
##  [8] "dis"     "rad"     "tax"     "ptratio" "b"       "lstat"   "medv"
```

```
modifiedtask <- subsetTask(task, subset = 1:400, features = c("crim", "age", "dis", "lstat"))
str(getTaskData(modifiedtask))
```

```
## 'data.frame':    400 obs. of  5 variables:
##  $ crim : num  0.00632 0.02731 0.02729 0.03237 0.06905 ...
##  $ age  : num  65.2 78.9 61.1 45.8 54.2 58.7 66.6 96.1 100 85.9 ...
##  $ dis  : num  4.09 4.97 4.97 6.06 6.06 ...
##  $ lstat: num  4.98 9.14 4.03 2.94 5.33 ...
##  $ medv : num  24 21.6 34.7 33.4 36.2 28.7 22.9 27.1 16.5 18.9 ...
```

Using `getTaskData` we can see that the modified task now includes 400 observations and 5 variables (the 4 included in the features character vector above and the target variable which will always be included). Some other useful functions are the following:

1. `removeConstantFeatures(<task name>)`:

constant features may arise dur to an inherent feature in the data collected or as a result of choosing a subset of observations.

2. `dropFeatures(task, c("rm", "nox"))`:

remove selected features from the task.

3. `normalizeFeatures(task, method = "standardize")`:

normalize numerical features by different methods (nonnumerical features are left untouched). The normalizing method can be one of `center` (subtract mean), `scale` (divide by standard deviation), `standardize` (center and scale), `range` (scale to a given range - default is [0, 1]). The optional argument `exclude` may be used to supply a character vector of columns to be excluded from the normalization. Type `?normalizeFeatures` for more info.

Learners

Many of the popular learning algorithms are already implemented in `mlr`. The `makeLearner` function requires the user to specify the learning method. Additionally, it is possible to modify defaults on the prediction type (i.e. for classification, we may choose `predict.type = "prob"` for probabilities), or set hyperparameters using a `list` passed to the `par.vals` argument.

```
class.lrn <- makeLearner("classif.randomForest", predict.type = "prob", fix.factors.prediction = TRUE)
regr.lrn <- makeLearner("regr.gbm", par.vals = list(n.trees = 500, interaction.depth = 4))
#cluster.lrn <- makeLearner("cluster.SimpleKMeans", N = 5)
```

Note that the `fix.factors.prediction = TRUE` argument is useful in situations where a factor level is present in the training data set but not the test data set; it adds a factor level for missing data in the test set thus avoiding problems.

The Learner object is a list and information can be extracted using the `$`, e.g. `regr.lrn$par.vals`. This information can also be accessed using other functions in `mlr`, for instance, `getHyperPars(lrn)` retrieves the current hyperparameter settings of the learner `lrn`. We also used above `getParamSet(lrn)` to get a description of all possible parameter settings for `lrn`.

Modifying a learner Just like it was possible to modify an existing task, we can also do so for a learner. Modifications include changing the `id` (this is either user specified or, if omitted, it is automatically set to the algorithm name), the prediction type, hyperparameter values, and more.

```
class.lrn <- setPredictType(class.lrn, "response")
regr.lrn <- setHyperPars(regr.lrn, n.trees = 400)
regr.lrn <- removeHyperPars(regr.lrn, c("n.trees", "interaction.depth"))
```

Note `removeHyperPars` sets the hyperparameters back to their default values.

Train

Once we create the task from the data set and identify the learning algorithm, the next step is to train the learner using the `train` command.

```
task <- makeClassifTask(data = iris, target = "Species")
lrn <- makeLearner("classif.lda")
mod <- mlr::train(lrn, task)
mod
```

```
## Model for learner.id=classif.lda; learner.class=classif.lda
## Trained on: task.id = iris; obs = 150; features = 4
## Hyperparameters:
```

The function `train` returns a list. The fitted model can be extracted using the `getLearnerModel(mod)` command. It is possible to choose a subset of observations to be used to train the model; this is achieved via the `subset` argument in `train`. Note that this is usually not needed since resampling strategies are supported. In the following example, we use the `BostonHousing` data to fit a linear model to the regression task. Note that `getTaskSize` returns the number of observations in a task.

```
data(BostonHousing, package = "mlbench")
task <- makeRegrTask(data = BostonHousing, target = "medv")
lrn <- makeLearner("regr.lm")
train.set <- sample(getTaskSize(task), size = getTaskSize(task)/3)
mod <- mlr::train(lrn, task, subset = train.set)
getLearnerModel(mod)
```

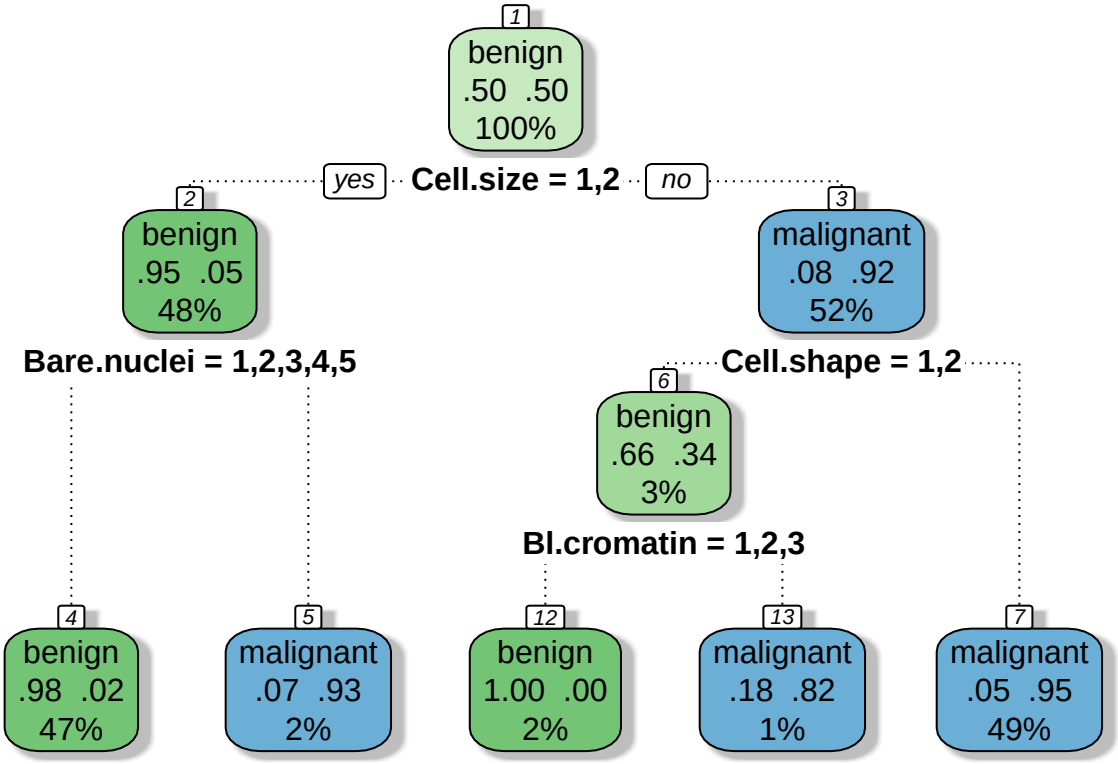
```
##
## Call:
## stats::lm(formula = f, data = d)
##
## Coefficients:
## (Intercept)         crim           zn        indus        chas1
##   38.578002    -0.118996     0.060324     0.102993     1.118329
##         nox           rm          age          dis          rad
##  -26.669634     3.957466    -0.003478    -1.616870     0.378537
##         tax      ptratio            b        lstat
##   -0.015915    -0.793560     0.007297    -0.512330
```

Finally, a note on weights passed as an argument in the `train` function. As an example for an application of weights [1] used in `train`, consider the `BreastCancer` data for which the target variable `Class` identifies 241 malignant and 458 benign cases. To deal with the imbalanced classes, we can incorporate weights in an attempt to allow the two classes to be equally represented in training the classifier. Here we use the

[1] note that mlr offers alternatives with more functionality for imbalanced classification problems

predefined task `bc.task` in mlr. The `getTaskTargets` function gets the target data from a task (in this example, this is equivalent to the vector `BreastCancer$Class`). Note that if weights are defined in task as well then those would be overwritten by the weights in `train`.

```
target <- getTaskTargets(bc.task)
tab <- as.numeric(table(target))
#obtain inverse class frequencies for the weights
w <- 1/tab[target]
mod <-mlr:: train("classif.rpart", task = bc.task, weights = w)
fancyRpartPlot(getLearnerModel(mod), sub = "")
```



Weights can also be useful as a means to grant more importance to recently collected data versus older data or to reduce the influence of outliers.

Predict

To predict target values, we use the `predict` function which takes as input the object returned by `train` and data for which we want predictions. The data can either come from the task (using the `task` argument) or it can be a data frame (passed using the `newdata` argument). Similarly to the `train` function, the `subset` argument may be used to pass different portions of the data in task.

```
n <- getTaskSize(bh.task)
train.set <- seq(1, n, by = 2)
test.set <- seq(2, n, by = 2)
lrn <- makeLearner("regr.gbm", n.trees = 100, interaction.depth = 4)
mod <- mlr::train(lrn, bh.task, subset = train.set)
preds <- predict(mod, task = bh.task, subset = test.set)
preds
```

```
## Prediction: 253 observations
## predict.type: response
## threshold:
## time: 0.00
##    id truth response
## 2   2  21.6 22.45613
## 4   4  33.4 23.47600
## 6   6  28.7 22.67572
## 8   8  27.1 21.83028
## 10 10  18.9 21.86525
## 12 12  18.9 22.33298
```

The function `predict` returns a list; The `$data` element of that list contains the true values of the target variable (in case of supervised learning) and the predictions. A direct way to obtain the true and predicted values of the target variable is through the `getPredictionTruth(preds)` and `getPredictionResponse(preds)` commands where `preds` is the list returned by the `predict` function.

```
head(getPredictionTruth(preds), 10)
```

```
##  [1] 21.6 33.4 28.7 27.1 18.9 18.9 20.4 19.9 17.5 18.2
```

```
head(getPredictionResponse(preds), 10)
```

```
##  [1] 22.45613 23.47600 22.67572 21.83028 21.86525 22.33298 22.52053
##  [8] 22.51713 22.24673 22.33986
```

For classification problems, class labels are predicted. We can obtain a confusion matrix through the command `getConfMatrix`. To get predicted posterior probabilities, we need to create the learner with `predict.type = "prob"`.

```
lrn <- makeLearner("classif.rpart", predict.type = "prob")
mod <- mlr::train(lrn, iris.task)
preds <- predict(mod, newdata = iris)
head(as.data.frame(preds))
```

```
##    truth prob.setosa prob.versicolor prob.virginica response
## 1 setosa           1               0              0   setosa
## 2 setosa           1               0              0   setosa
## 3 setosa           1               0              0   setosa
## 4 setosa           1               0              0   setosa
## 5 setosa           1               0              0   setosa
## 6 setosa           1               0              0   setosa
```

```
head(getPredictionProbabilities(preds))
```

```
##   setosa versicolor virginica
## 1      1          0         0
## 2      1          0         0
## 3      1          0         0
## 4      1          0         0
## 5      1          0         0
## 6      1          0         0
```

We can also adjust the threshold value that is used to map the predicted posterior probabilities to the class labels. The default value for binary classification is 0.5; however, it may be necessary to increase/decrease this value in various situations such as in the case of classifying cancer rates where one would be concerned with false negatives (prediction of no cancer when the truth is yes cancer). By changing the threshold we change the sensitivity of the model. An example for adjusting the threshold in a binary classification setting is shown below.

```
data(Sonar)
table(Sonar$Class)
```

```
##
##   M   R
## 111  97
```

```
getTaskDescription(sonar.task)$positive
```

```
## [1] "M"
```

```
lrn <- makeLearner("classif.rpart", predict.type = "prob")
mod <- mlr::train(lrn, task = sonar.task)
# predict with default threshold
preds <- predict(mod, sonar.task)
preds$threshold
```

```
##   M   R
## 0.5 0.5
```

```
# set threshold value for +ve class
preds2 <- setThreshold(preds, 0.9)
# confusion matrices
getConfMatrix(preds)
```

```
##        predicted
## true     M  R -SUM-
##   M     95 16    16
##   R     10 87    10
##   -SUM- 10 16    26
```

```
getConfMatrix(preds2)
```

```
##        predicted
## true     M  R -SUM-
##   M     84 27    27
##   R      6 91     6
##   -SUM-  6 27    33
```

For multiclass classification problems, the threshold is given by a named vector specifying the values by which each probability will be divided.

Performance

There are many available performance measures implemented in mlr but one can also create their own performance measures. For a particular prediction object, say `preds` calling `performance(preds)` will give the calculated performance measure. It is also possible to calculate the time needed to train the learner (passing `timetrain` as an argument - see below), the time needed to compute the prediction (`timepredict`) or both (`timeboth`).

To obtain a list of available measures suitable for a particular problem type or `task`, use the `listMeasures` argument. The function `getDefaultMeasure` shows the defaults for a particular learner or task.

```
listMeasures("classif", properties = "classif.multi")
```

```
## [1] "timepredict"    "acc"            "ber"            "featperc"
## [5] "mmce"           "timeboth"       "timetrain"      "multiclass.auc"
```

```
getDefaultMeasure(bh.task)
```

```
## Name: Mean of squared errors
## Performance measure: mse
## Properties: regr,req.pred,req.truth
## Minimize: TRUE
## Best: 0; Worst: Inf
## Aggregated by: test.mean
## Note:
```

The following piece of `R` code shows how to obtain the performance measure from the prediction object.

```
n <- getTaskSize(bh.task)
train.set <- seq(1, n, by = 2)
test.set <- seq(1, n, by = 2)
lrn <- makeLearner("regr.gbm", n.trees = 1000)
mod <- mlr::train(lrn, task = bh.task, subset = train.set)
preds <- predict(mod, task = bh.task, subset = test.set)
performance(preds)
```

```
##      mse
## 38.45852
```

To change the performance measure, we can do so via the `measures` argument. It is possible to calculate several performance measures by passing them as a list.

```
performance(preds, measures = medse)
```

```
##    medse
## 10.29158
```

```
performance(preds, measures = list(mse, medse, mae))
```

```
##       mse     medse       mae
## 38.458520 10.291584  4.412879
```

It is a necessary requirement to pass the model or the task in order to calculate some performance measures. For instance, for `timetrain` calculations, the model also needs to be passed as ar argument.

```
performance(preds, measures = timetrain, model = mod)
```

```
## timetrain
##     0.061
```

For clustering problems, the task is required.

```
lrn <- makeLearner("cluster.kmeans", centers = 3)
mod <- mlr::train(lrn, task = mtcars.task)
preds <- predict(mod, task = mtcars.task)
performance(preds, measures = dunn, task = mtcars.task)
```

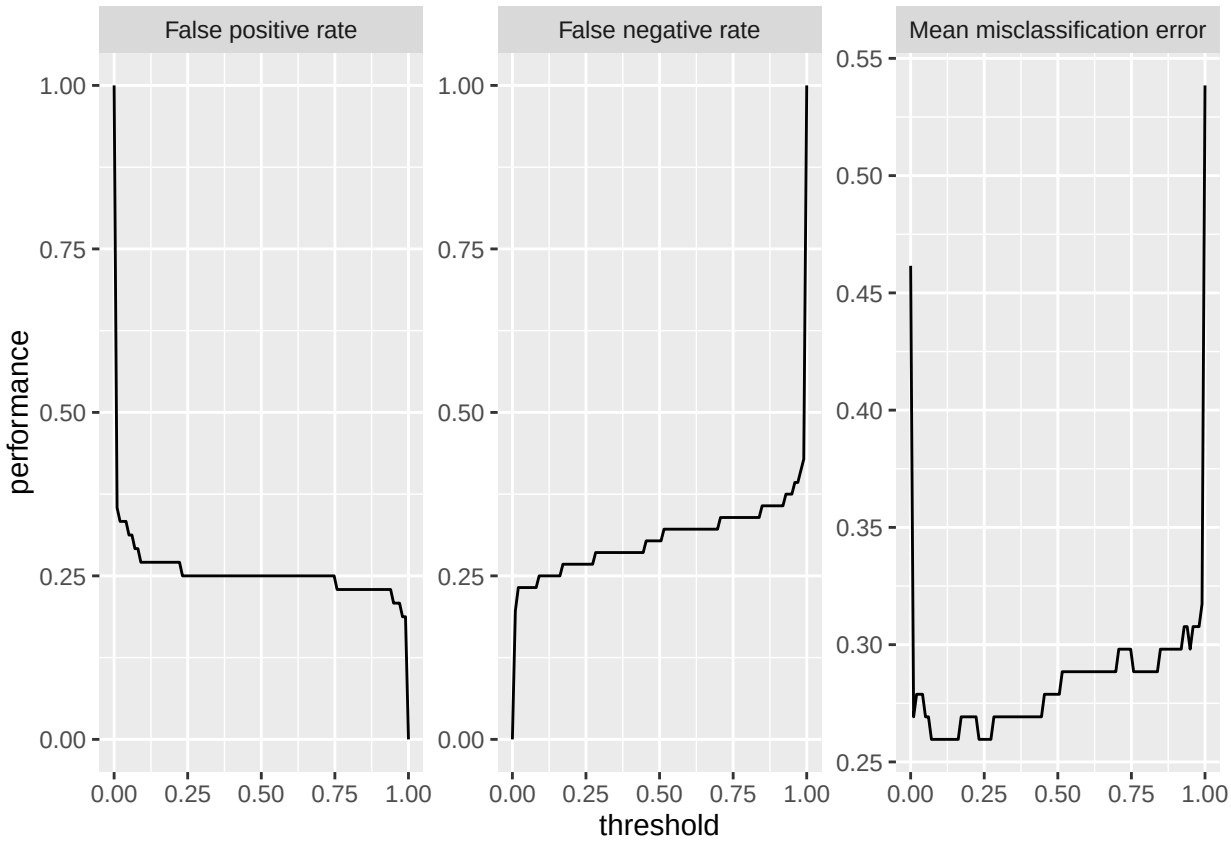
```
##      dunn
## 0.1178415
```

As previously mentioned, the threshold for classification problems alters the sensitivity of the model and therefore affects performance. The command `generateThreshVsPerfData` used on the prediction object along with a performance measure (or a list of them) generates data on the learner performance versus the threshold. T

```
lrn <- makeLearner("classif.lda", predict.type = "prob")
n <- getTaskSize(sonar.task)
train.set <- seq(1, n, by = 2)
test.set <- seq (2, n, by = 2)
mod <- mlr::train(lrn, task = sonar.task, subset = train.set)
preds <- predict(mod, task = sonar.task, subset = test.set)
performance(preds, measures = list(fpr, fnr, mmce))
```

```
##       fpr       fnr      mmce
## 0.2500000 0.3035714 0.2788462
```

```
d <- generateThreshVsPerfData(preds, measures = list(fpr, fnr, mmce))
plotThreshVsPerf(d)
```



Resampling

Resampling strategies are often used to assess the performance of a learning algorithm by splitting the data into multiple training and test sets. Each training set is used to train a learner and each test set is reserved for predictions. We get the mean performance measure obtained by agreggating all individual performances. The `makeResampleDesc` function is used to choose the resampling strategy; the method argument can be set to one of cross-validation, leave-one-out cross-validation, repeated cross-validation, out-of-bag bootstrap, subsampling (a.k.a. *Monte-Carlo* cross-validation), and holdout. Additional arguments can be passed to the function depending on the chosen method. Once the resampling description is specified, we use the function `resample` to fit a model specified by a learner on a task, which calculates predictions and performance measures for all training and test sets as specified by the resampling description.

```
rdesc <- makeResampleDesc("CV", iters = 3, stratify = TRUE)
task <- makeClassifTask(data = iris, target = "Species")
lrn <- makeLearner("classif.rpart")
r <- resample(lrn, task, rdesc)
```

```
## [Resample] cross-validation iter: 1

## [Resample] cross-validation iter: 2

## [Resample] cross-validation iter: 3

## [Resample] Result: mmce.test.mean=0.0732
```

```
r
```

```
## Resample Result
## Task: iris
## Learner: classif.rpart
## mmce.aggr: 0.07
## mmce.mean: 0.07
## mmce.sd: 0.01
## Runtime: 0.0305369
```

```
r$measures.test
```

```
##   iter       mmce
## 1    1 0.07843137
## 2    2 0.06122449
## 3    3 0.08000000
```

```
r$aggr
```

```
## mmce.test.mean
##     0.07321862
```

```
r$measures.train
```

```
##   iter mmce
## 1    1   NA
## 2    2   NA
## 3    3   NA
```

The `resample` function returns a list whose elements we can access using the `$` notation. In the above exmaple `r$measures.test` gives the performance on the 3 individual test sets (as specified in the resampling description) and `r$measures.train` returns missing values since no predictions on the training sets were made. If we wish to have predictions on the training sets we can set `predict = "both"` or `predict = "train"` as an argument in `makeResampleDesc`.

Next, we can access `r$pred$data` which gives a data frame of the predictions and true (in supervised learning) values of the target variable. Note that it is possible to pass multiple measures as a list in `resample` (including `timetrain`).

```
head(r$pred$data)
```

```
##   id  truth response iter  set
## 1  4 setosa   setosa    1 test
## 2  5 setosa   setosa    1 test
## 3  6 setosa   setosa    1 test
## 4  8 setosa   setosa    1 test
## 5 16 setosa   setosa    1 test
## 6 20 setosa   setosa    1 test
```

Stratified resampling ensures that the same proportion of the classes falls in all partitions of the data such that, in a classification setting, each training/test set trains a model with no class being under-represented. This is especially important in small data sets as well as imbalanced classification problems. The `stratify = TRUE` argument is passed when making the resampling description (as in the `R` code above). The `stratify.cols = <col. name>` argument is used to stratify factor variable inputs to ensure that all subgroups are represented in the data partitions.

Accessing individual learners By default, `resample` does not return the individual learners but can do so by passing the argument `models = TRUE` when calling `resample`. More useful still, is to extract certain information from each model like, for instance, the variable importance for chosen models. This is achieved via the `extract` argument.

```
rdesc <- makeResampleDesc("CV", iters = 3)
lrn <- makeLearner("regr.rpart")
r <- resample(lrn, task = bh.task, resampling = rdesc, extract = function(x)
  x$learner.model$variable.importance, models = FALSE)
```

```
## [Resample] cross-validation iter: 1

## [Resample] cross-validation iter: 2

## [Resample] cross-validation iter: 3

## [Resample] Result: mse.test.mean=25.4
```

```
r$extract
```

```
## [[1]]
##         rm      lstat      indus    ptratio        nox        age
## 18617.3719 14071.2906  4926.4376  3997.0454  3367.8188  2993.7238
##       crim        tax        dis         zn          b        rad
##  2937.2363  2827.4408  2576.0851  2132.4695   711.8396   298.7435
##
## [[2]]
##         rm      lstat       crim        age      indus    ptratio
## 17499.4752 12103.1335  3718.0924  3143.2565  3060.3740  3006.6774
##        dis        nox         zn        rad        tax          b
##  2945.8373  2892.6036  1575.9143   726.0415   686.3487   510.1821
##       chas
##   474.2825
##
## [[3]]
##     lstat     indus       nox      crim        rm       age       dis
## 13384.166  8649.907  8450.450  7779.603  7727.961  7517.997  4408.284
##   ptratio       tax        zn
##  2077.555  1430.559   172.779
```

Now, `r$extract` is going to give information on the given function from which we asked to obtain the variable importance. With `models = FALSE` we do not obtain other information on the model so attempting to call `getLearnerModel(r$models[[1]])` would return an error.

Resample instance The command `makeResampleInstance` takes as arguments an object of class ResampleDesc and a task [or the size of the data set i.e. nrow()]. This creates a ResampleInstance object which mainly stores the indices of the training and test sets used in each iteration. This feature may be useful in situations where we want to perform paired experiments like testing the performance of several learners on exactly the same data.

```
rdesc <- makeResampleDesc("CV", iters = 3, stratify = TRUE)
rin <- makeResampleInstance(rdesc, task = iris.task)
rin$train.inds[[2]]
```

```
##   [1]  21  24   2  13  27  17  35   6  19  12  30  32  45  26  18   5  22
##  [18]  43  15  33  49  47  41  29  44  14  23   1   3  38   4  34  28  86
##  [35]  99  93  89  56  53  84  51  69  67  91  80  77  62  58  95  85  97
##  [52]  74  65  68  83  98  63  70  92  57  61  66  79  52  90  88 104 108
##  [69] 107 148 113 149 124 105 126 141 127 129 119 109 150 144 125 101 118
##  [86] 114 138 112 142 140 115 110 103 106 139 134 111 137 102 135
```

In `makeResampleInstance` the indices are drawn randomly; the function `makeFixedHoldoutInstance` allows the training and test sets to be specified manually.

```
rin <- makeFixedHoldoutInstance(train.inds = 1:100, test.inds = 101:150, size = 150)
rin
```

```
## Resample instance for 150 cases.
## Resample description: holdout with 0.67 split rate.
## Predict: test
## Stratification: FALSE
```

In resampling startegies, we get a performance measure which is aggregated over all the measures calculated for each iteration. By default, the aggregated score is the mean error on the test set. We can change the aggregation method for a measure via `setAggregation(measure, aggr)`. For the different options for `measure` and `aggr` type `?measures` and `?aggregations`, respectively.

```
m1 <- mmce
m2 <- setAggregation(tpr, test.median) # tpr = true positive rate
rdesc <- makeResampleDesc("CV", iters = 3)
r <- resample("classif.rpart", task = sonar.task, resampling = rdesc, measures = list(m1,m2))
```

```
## [Resample] cross-validation iter: 1

## [Resample] cross-validation iter: 2

## [Resample] cross-validation iter: 3

## [Resample] Result: mmce.test.mean=0.274,tpr.test.median= 0.8
```

```
r$aggr
```

```
##  mmce.test.mean tpr.test.median
##       0.2740511       0.8000000
```

To get predictions on both training and test sets, we need to set `predict = "both"` in `makeResampleDesc` and set the aggregation method to `train.mean` if we wish to calculate the mean. An example is shown below.

```
train.mmce <- setAggregation(mmce, train.mean)
rdesc <- makeResampleDesc("CV", iters = 3, predict = "both")
r <- resample("classif.rpart", task = sonar.task, resampling = rdesc, measures = list(mmce, train.mmce))
```

```
## [Resample] cross-validation iter: 1

## [Resample] cross-validation iter: 2

## [Resample] cross-validation iter: 3

## [Resample] Result: mmce.test.mean=0.289,mmce.train.mean=0.113
```

```
r$aggr
```

```
##  mmce.test.mean mmce.train.mean
##       0.2888199       0.1129357
```

There exist functions which act as convenience wrappers for the various existing resampling strategies. They don't offer as much flexibility as `resample` but can be quick and useful if trying out a number of learners initially. An example with `crossval` is shown below.

```
cviris <- crossval("classif.lda", task = iris.task, iters = 3, measures = list(mmce, ber))
```

```
## [Resample] cross-validation iter: 1

## [Resample] cross-validation iter: 2

## [Resample] cross-validation iter: 3

## [Resample] Result: mmce.test.mean=0.02,ber.test.mean=0.0223
```

```
cviris
```

```
## Resample Result
## Task: iris-example
## Learner: classif.lda
## mmce.aggr: 0.02
## mmce.mean: 0.02
## mmce.sd: 0.02
## ber.aggr: 0.02
## ber.mean: 0.02
## ber.sd: 0.02
## Runtime: 0.0299258
```

Benchmark experiments

Using `benchmark`, we can compare different learning algorithms across one or more tasks w.r.t. a given resampling strategy. The function has the advantage of conducting *paired* experiments thus comparing the same training/test sets for the different learners. In the following `R` code, we specify a single task (here, we use the built in `sonar.task`) and apply a LDA and a classification tree learner. The resampling strategy is chosen as `Holdout` hence the performance is calculated on a single randomly sampled test set.

```
lrns <- list(makeLearner("classif.lda"), makeLearner("classif.rpart"))
rdesc <- makeResampleDesc("Holdout")
bmr <- benchmark(learners = lrns, tasks = sonar.task, resamplings = rdesc)
```

```
## Task: Sonar-example, Learner: classif.lda

## [Resample] holdout iter: 1

## [Resample] Result: mmce.test.mean= 0.3

## Task: Sonar-example, Learner: classif.rpart

## [Resample] holdout iter: 1

## [Resample] Result: mmce.test.mean=0.314
```

```
bmr
```

```
##         task.id    learner.id mmce.test.mean
## 1 Sonar-example   classif.lda      0.3000000
## 2 Sonar-example classif.rpart      0.3142857
```

The function `benchmark` returns an object of class BenchmarkResult which contains a list of lists of ResampleResult objects ordered by task and followed by learner. The mlr `getBMR<...>` commands allows access to the benchmark results.

```
getBMRPerformances(bmr, as.df = TRUE)
```

Learner performances

```
##         task.id    learner.id iter      mmce
## 1 Sonar-example   classif.lda    1 0.3000000
## 2 Sonar-example classif.rpart    1 0.3142857
```

```
getBMRAggrPerformances(bmr, as.df = TRUE)
```

```
##         task.id    learner.id mmce.test.mean
## 1 Sonar-example   classif.lda      0.3000000
## 2 Sonar-example classif.rpart      0.3142857
```

The two results (top: individual performance in resampling runs, bottom: aggregated performance values) coincide since `Holdout` was used as the resampling strategy. The optional argument `as.df = TRUE` returns the results in the form of a data frame which is often more convenient.

Predictions By default, `keep.pred = TRUE` in `benchmark` which allows the user to access the predictions with `getBMRPredictions`. If `keep.pred = FALSE`, the following command will result in an error.

```r
head(getBMRPredictions(bmr, as.df = TRUE))
```

```
##         task.id  learner.id  id truth response iter  set
## 1 Sonar-example classif.lda  77     R        R    1 test
## 2 Sonar-example classif.lda 107     M        M    1 test
## 3 Sonar-example classif.lda  87     R        R    1 test
## 4 Sonar-example classif.lda  85     R        M    1 test
## 5 Sonar-example classif.lda  52     R        R    1 test
## 6 Sonar-example classif.lda 128     M        R    1 test
```

The learner and task ID is automatically set to the name of the algorithm and task if not explicitly specified. Using the ID in `getBMRPredictions`, it is possible to access results for certain learners or tasks. For instance below, we obtain the predictions from the classification tree learner.

```r
head(getBMRPredictions(bmr, learner.ids = "classif.rpart", as.df = TRUE))
```

```
##           task.id    learner.id  id truth response iter  set
## 77  Sonar-example classif.rpart  77     R        R    1 test
## 107 Sonar-example classif.rpart 107     M        R    1 test
## 87  Sonar-example classif.rpart  87     R        M    1 test
## 85  Sonar-example classif.rpart  85     R        R    1 test
## 52  Sonar-example classif.rpart  52     R        R    1 test
## 128 Sonar-example classif.rpart 128     M        M    1 test
```

If unsure of the ID for the learner, task, and performance measure, they can be accessed using `getBMRLearnerIds`, `getBMRTaskIds`, and `getBMRMeasureIds`, respectively.

Models Similarly, to *Predictions* above, the argument `models = FALSE` must be set in `benchmark` if the user does not want to keep the fitted models for all learners and tasks. If set to true (default), the models may be accessed through `getBMRModels`.

```r
getBMRModels(bmr, learner.ids = "classif.lda")
```

```
## $`Sonar-example`
## $`Sonar-example`$classif.lda
## $`Sonar-example`$classif.lda[[1]]
## Model for learner.id=classif.lda; learner.class=classif.lda
## Trained on: task.id = Sonar-example; obs = 138; features = 60
## Hyperparameters:
```

More `getBMR<...>` functions exist to extract information on learners, measures, and more.

Additional benchmark experiments and merging results Once a benchmark experiment has been conducted on a task, we may need to add more learners (or, alternatively, we may wish to extend existing learners to other tasks). We can perform another benchmark experiment and then merge the results (through either `mergeBenchmarkResultLearner` or `mergeBenchmarkResultTask`). As an example, we perform another benchmark experiment on `sonar.task` now with random forest and quadratic discriminant analysis learning algorithms. We then fuse this with the `bmr` object we obtained above to get a single `BenchmarkResult` object.

```
lrns2 <- list(makeLearner("classif.randomForest"), makeLearner("classif.qda"))
bmr2 <- benchmark(learners = lrns2, tasks = sonar.task, resamplings = rdesc, show.info = FALSE)
bmr2
```

```
##         task.id           learner.id mmce.test.mean
## 1 Sonar-example classif.randomForest      0.1571429
## 2 Sonar-example          classif.qda      0.3857143
```

```
bmrsingle <- mergeBenchmarkResultLearner(bmr, bmr2)
bmrsingle
```

```
##         task.id           learner.id mmce.test.mean
## 1 Sonar-example          classif.lda      0.3000000
## 2 Sonar-example        classif.rpart      0.3142857
## 3 Sonar-example classif.randomForest      0.1571429
## 4 Sonar-example          classif.qda      0.3857143
```

However, note that the resampling description was passed to `benchmark` twice, once to obtain the `bmr` and then the `bmr2` objects. The training/test pairs therefore were most likely different in the first benchmark call than the second. For more accurate merging of benchmark results, we can opt to work with `ResampleInstance` from the start, or extract the `ResampleInstance` from the resample description in the first benchmark call and pass it as argument in later benchmark calls. An example is given below.

```
rin <- getBMRPredictions(bmr)[[1]][[1]]$instance
rin
```

```
## Resample instance for 208 cases.
## Resample description: holdout with 0.67 split rate.
## Predict: test
## Stratification: FALSE
```

```
bmr3 <- benchmark(learners = lrns2, tasks = sonar.task, resamplings = rin, show.info = FALSE)
mergeBenchmarkResultLearner(bmr, bmr3)
```

```
##         task.id           learner.id mmce.test.mean
## 1 Sonar-example          classif.lda      0.3000000
## 2 Sonar-example        classif.rpart      0.3142857
## 3 Sonar-example classif.randomForest      0.1571429
## 4 Sonar-example          classif.qda      0.4000000
```

Benchmark analysis & visualization Once benchmark experiments are conducted for the various learners and tasks, we may wish to rank and assess the performance of various algorithms, perform hypotheses tests or visualize the results. The mlr package offers various functions to do so and we explore some of these below using a longer benchmark example than the one above.

```
# list of 3 learners
lrns <-list(makeLearner("classif.lda", id = "lda"), makeLearner("classif.rpart", id = "rpart"), makeLea
# convertMLBenchObjToTask does exactly what it says... here we create 2 tasks
ring.task <- convertMLBenchObjToTask("mlbench.ringnorm", n = 600)
wave.task <- convertMLBenchObjToTask("mlbench.waveform", n = 600)
```

```
tasks <- list(iris.task, sonar.task, pid.task, ring.task, wave.task)
# 10-fold cross-validation
rdesc <- makeResampleDesc("CV", iters = 10)
meas <- list(mmce, ber, timetrain)
bmr <- benchmark(learners = lrns, tasks = tasks, resamplings = rdesc, measures = meas, show.info = FALSE
bmr
```

```
##                         task.id   learner.id mmce.test.mean ber.test.mean
## 1                  iris-example          lda     0.02000000    0.02083333
## 2                  iris-example        rpart     0.05333333    0.05309524
## 3                  iris-example randomForest     0.03333333    0.02809524
## 4              mlbench.ringnorm          lda     0.36833333    0.37033082
## 5              mlbench.ringnorm        rpart     0.21500000    0.21793388
## 6              mlbench.ringnorm randomForest     0.06333333    0.06245273
## 7              mlbench.waveform          lda     0.17333333    0.17501347
## 8              mlbench.waveform        rpart     0.27333333    0.27457459
## 9              mlbench.waveform randomForest     0.15500000    0.15711244
## 10 PimaIndiansDiabetes-example          lda     0.23051948    0.27912739
## 11 PimaIndiansDiabetes-example        rpart     0.25264867    0.28804489
## 12 PimaIndiansDiabetes-example randomForest     0.23573137    0.27437875
## 13                Sonar-example          lda     0.25404762    0.25764222
## 14                Sonar-example        rpart     0.27857143    0.27835054
## 15                Sonar-example randomForest     0.15833333    0.15774143
##    timetrain.test.mean
## 1               0.0023
## 2               0.0034
## 3               0.0464
## 4               0.0079
## 5               0.0091
## 6               0.3491
## 7               0.0089
## 8               0.0096
## 9               0.3477
## 10              0.0046
## 11              0.0057
## 12              0.3715
## 13              0.0140
## 14              0.0101
## 15              0.2142
```

The individual performances on each iteration for each learner, task, and measure can be accessed via `getBMRPerformances` as shown below:

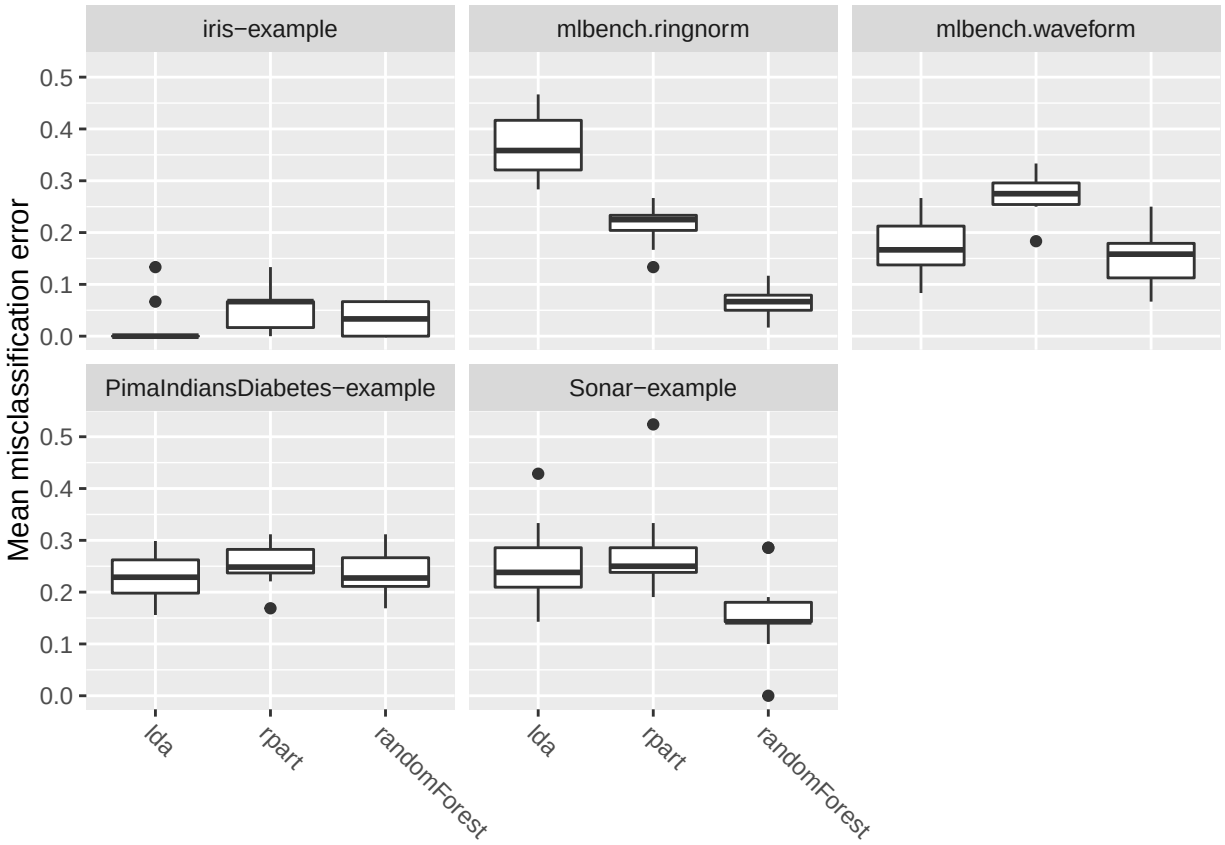
```
perf <- getBMRPerformances(bmr, as.df = TRUE)
head(perf)
```

```
##        task.id learner.id iter       mmce        ber timetrain
## 1 iris-example        lda    1 0.00000000 0.00000000     0.003
## 2 iris-example        lda    2 0.00000000 0.00000000     0.002
## 3 iris-example        lda    3 0.00000000 0.00000000     0.002
## 4 iris-example        lda    4 0.00000000 0.00000000     0.003
## 5 iris-example        lda    5 0.13333333 0.12500000     0.002
## 6 iris-example        lda    6 0.06666667 0.08333333     0.002
```

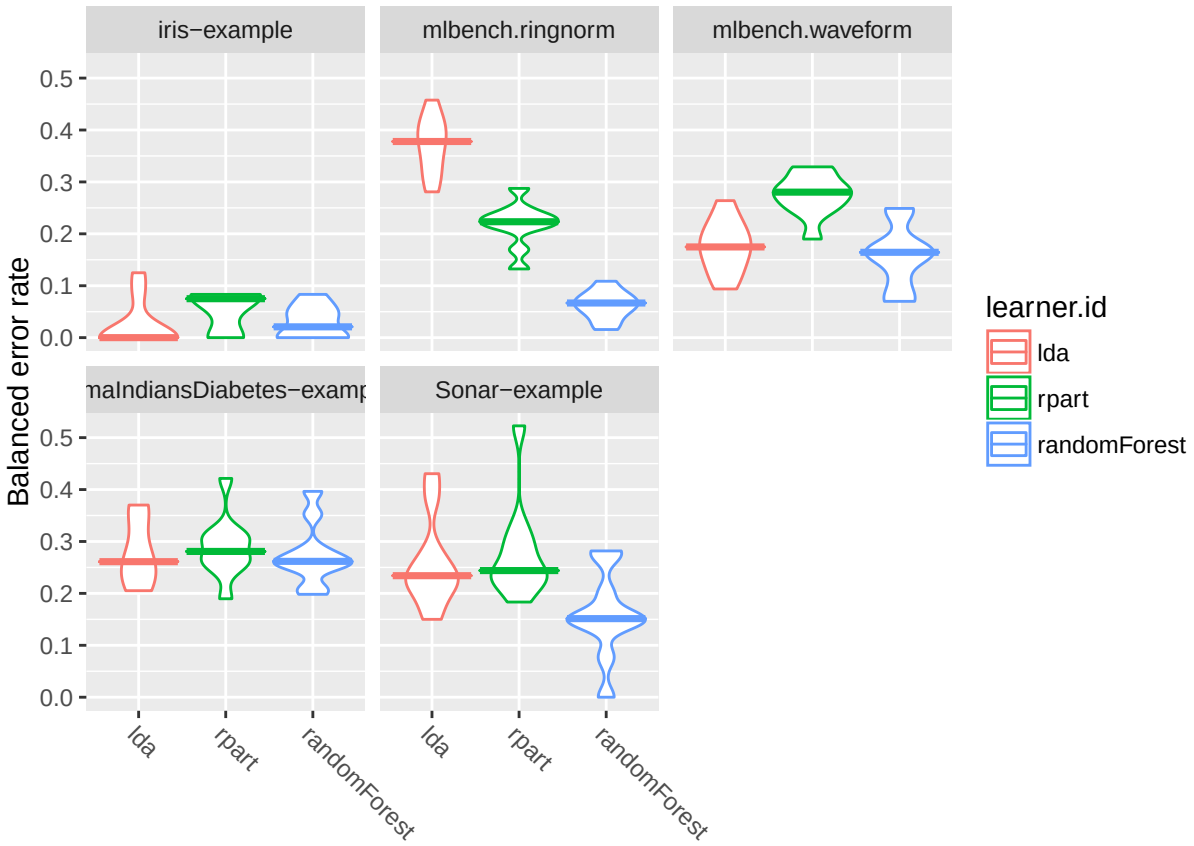
Performance tables like the one shown above get increasingly harder to read and comprehend with more experiments. A more convenient way to view the results is by plotting and visualization.

Plots The function `plotBMRBoxplots` takes as input a benchmark object and displays a selected performance measure for all tasks and for all learners as a box or violin plot using the `ggplot2` graphics.

```
plotBMRBoxplots(bmr, measure = mmce) + facet_wrap(~ task.id, nrow = 2)
```

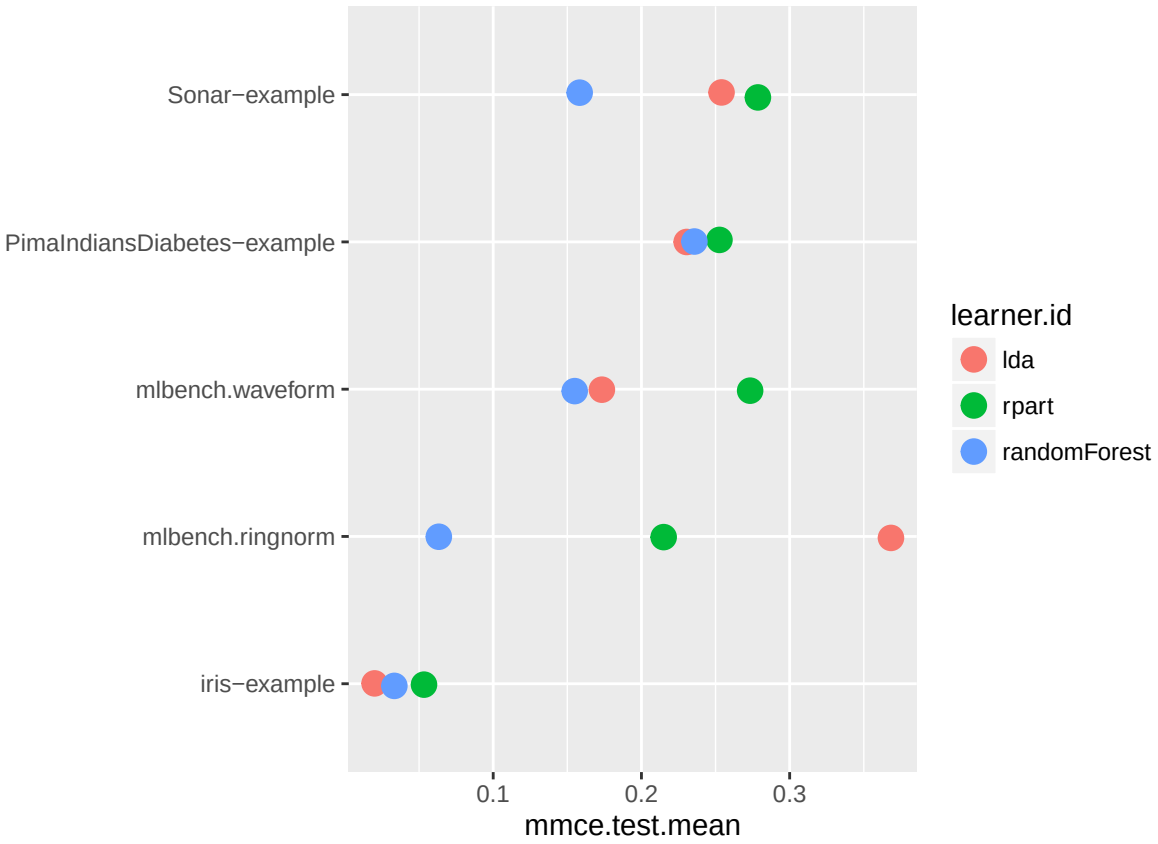



```
plotBMRBoxplots(bmr, measure = ber, style = "violin") + aes(color = learner.id) +
  facet_wrap(~ task.id, nrow = 2)
```



The aggregated measure score on the test set (e.g. `mmce.test.mean`) is retrieved from the `benchmark` output for each learner and task and displayed through `plotBMRSummary`. By default the first measure is used. Note that the argument `jitter = 0.05` is a vertical distance added between points to prevent overplotting.

```
plotBMRSummary(bmr)
```



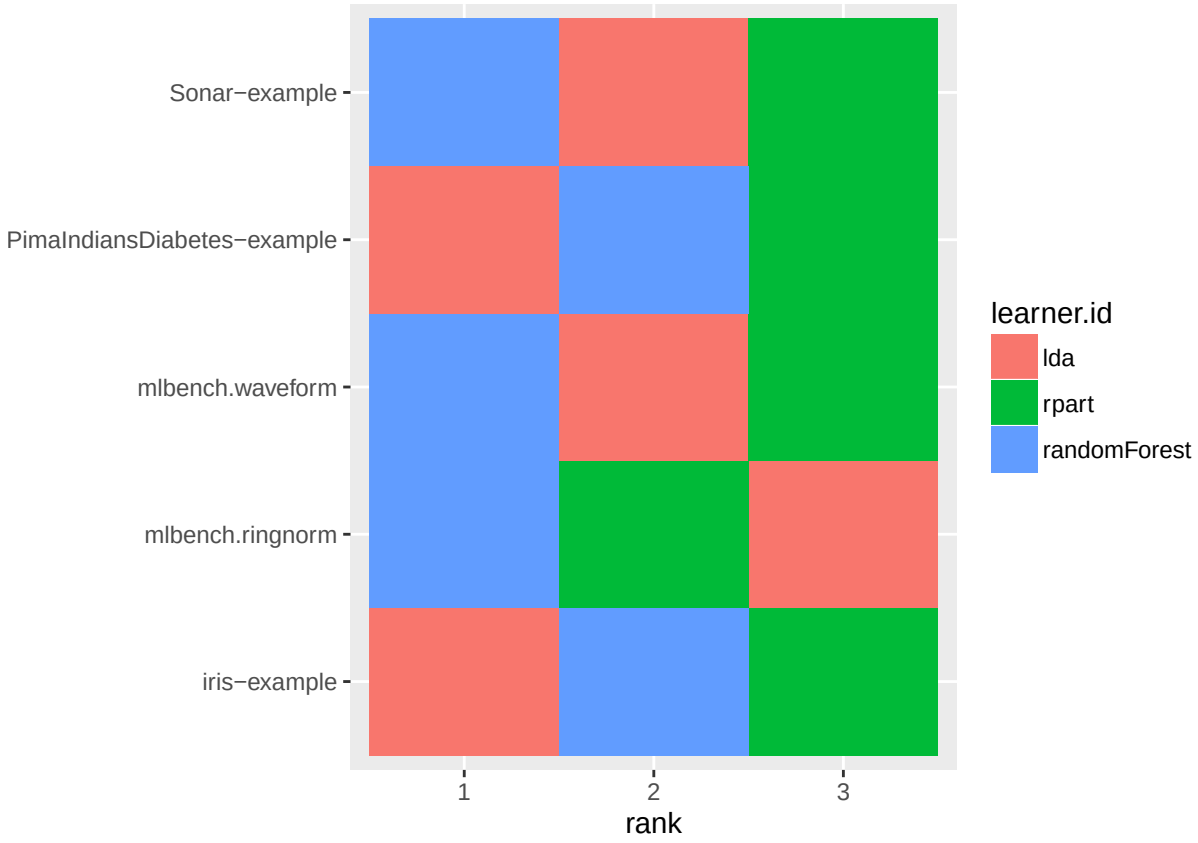
Clearly, functionality showing the relative performance is of interest and what we are usually after with the benchmark experiments. The function `convertBMRToRankMatrix` calculates the rank based on a selected learner aggregated performance measure.

```
m <- convertBMRToRankMatrix(bmr, measure = mmce)
m
```

```
##              iris-example mlbench.ringnorm mlbench.waveform
## lda                     1                3                2
## rpart                   3                2                3
## randomForest            2                1                1
##              PimaIndiansDiabetes-example Sonar-example
## lda                                    1             2
## rpart                                  3             3
## randomForest                           2             1
```

Alternatively, we can visualize the ranking results as a bar chart using `plotBMRRanksAsBarChart`. The ranks are displayed from best to worst on the horizontal axis and the tasks are shown on the vertical axis.

```
plotBMRRanksAsBarChart(bmr, pos = "tile")
```



Hypothesis tests Hypothesis tests can be used to conclude whether there is a significant difference between the performance of the various learners. While parametric hypothesis tests may have more power over nonparametric tests, they make assumptions about the underlying distributions from which the sample was drawn from which often means that, in order for the results to be at all reliable, we would need many data sets to show significance differences at reasonable significance levels. The mlr package provides the **Overall Friedman test** and the **Friedman-Nemenyi post hoc test**.

The Friedman test is the nonparametric alternative to a one-way ANOVA with repeated measures. We use it to compare three or more learners where the data used is the same in each learning algorithm. Unlike the ANOVA which requires the sample is drawn from a normal distribution and equal variances of the residuals, the Friedman test is free from such restrictions (but, as mentioned above, less powerful). The hypotheses for the comparison are H_0: The distributions (whatever they are) are the same across repeated measures and H_1: The distributions across repeated measures are different.

```
friedmanTestBMR(bmr)
```

```
##
##  Friedman rank sum test
##
## data:  mmce.test.mean and learner.id and task.id
## Friedman chi-squared = 5.2, df = 2, p-value = 0.07427
```

Next, if the Friedman test results show a significant p-value (depending on the significance level you set i.e. $p < \alpha$ for significance), then this would mean that we can reject the null that all learners perform the same but

at this point we don't know which ones are superior. Therefore, our next step will be to try and find out which pairs of our groups are significantly different then each other with a post hoc analysis. We carry this out with `friedmanPostHocTestBMR`. The following `R` code demonstrates this with a choice of a significance level of 0.1.

```
friedmanPostHocTestBMR(bmr, p.value = 0.1)
```

```
## Loading required package: PMCMR
```

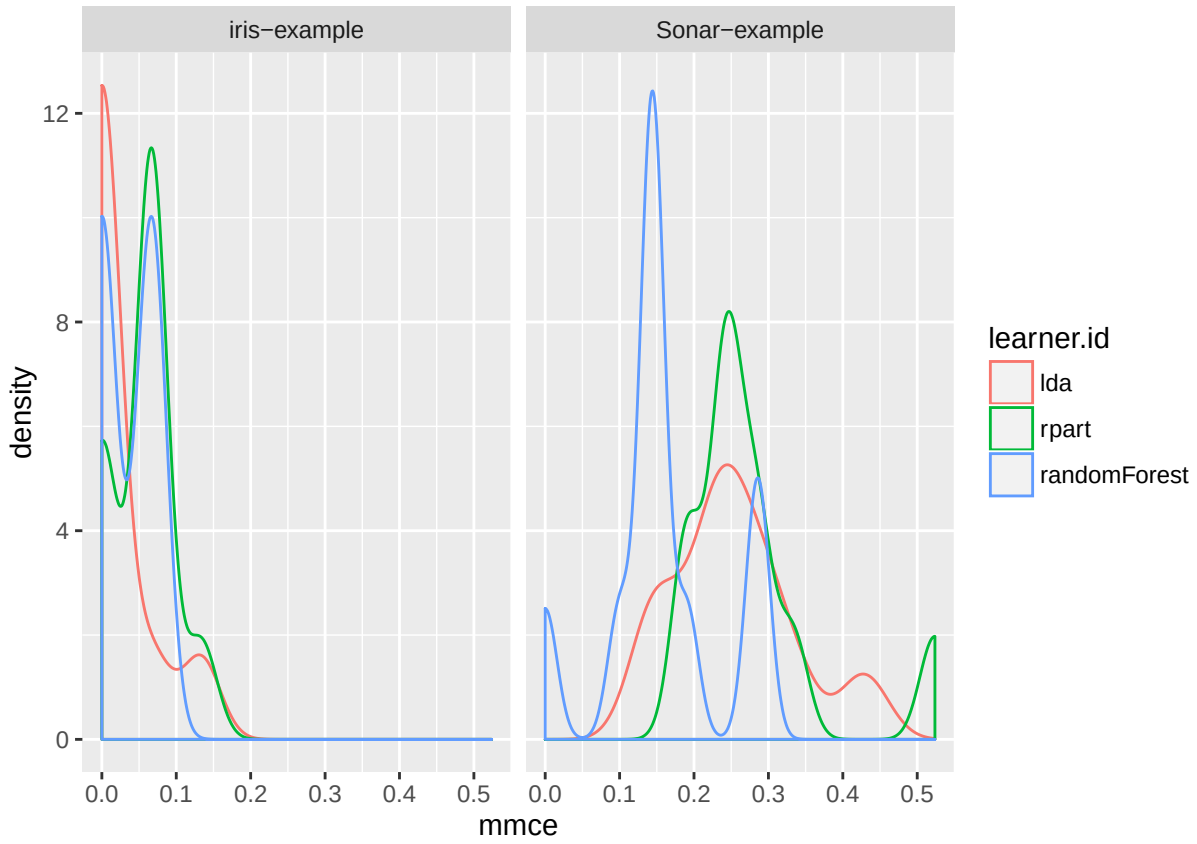
```
##
##  Pairwise comparisons using Nemenyi multiple comparison test
##              with q approximation for unreplicated blocked data
##
## data:  mmce.test.mean and learner.id and task.id
##
##              lda   rpart
## rpart        0.254 -
## randomForest 0.802 0.069
##
## P value adjustment method: none
```

The results show that a significance level of 0.1, we can reject the null that there exists no performance difference between `rpart` and `randomForest`.

Custom plots Examples of custom plots using the objects returned by `getBMRPerformances` and `getBMRAggrPerformances`.

Density plots

```
perf <- getBMRPerformances(bmr, as.df = TRUE)
qplot(mmce, colour = learner.id, facets = . ~ task.id,
data = perf[perf$task.id %in% c("iris-example", "Sonar-example"), ],
geom = "density")
```



In the following `R` code, we reshape the `perf` data frame by keeping the variables `task.id`, `learner.id`, and `iter` and collect all measures into a single column (`variable`) and their corresponding values in a second column (`value`).

```
head(perf)
```

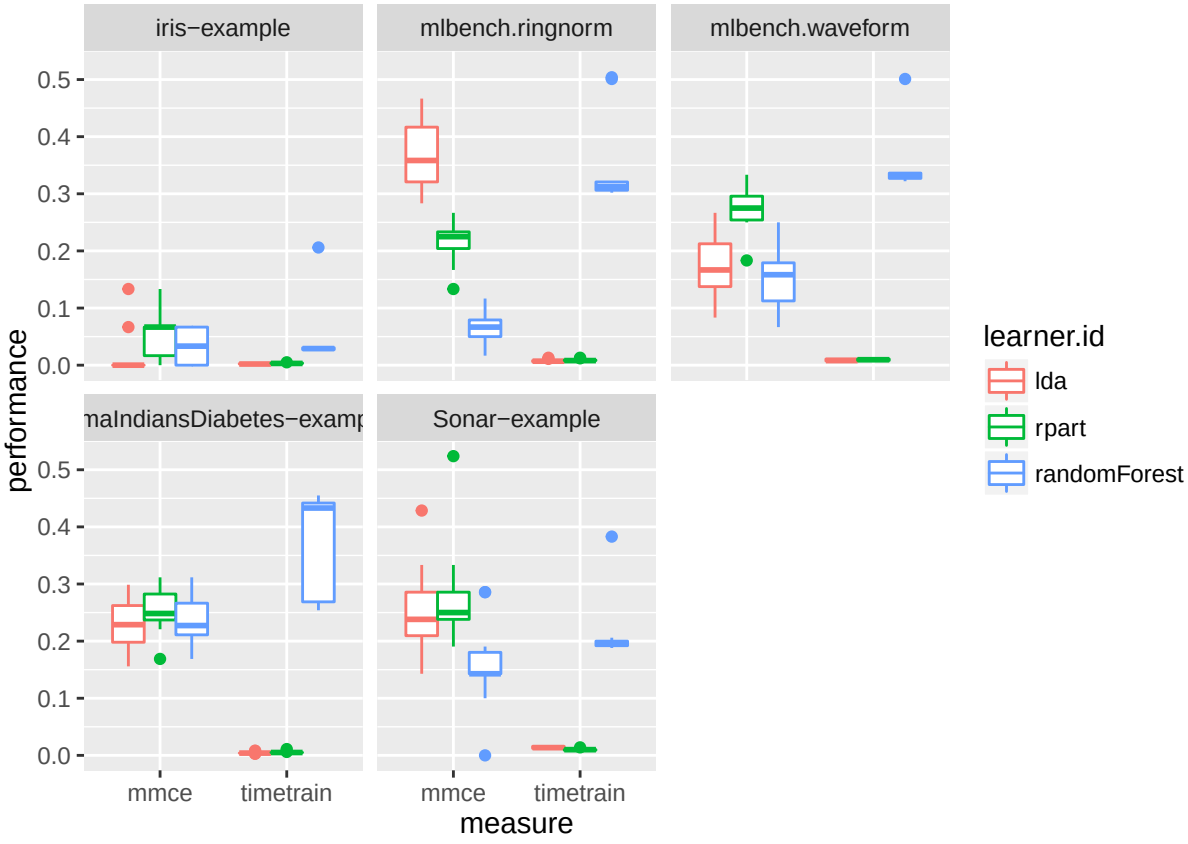
```
##        task.id learner.id iter       mmce        ber timetrain
## 1 iris-example        lda    1 0.00000000 0.00000000     0.003
## 2 iris-example        lda    2 0.00000000 0.00000000     0.002
## 3 iris-example        lda    3 0.00000000 0.00000000     0.002
## 4 iris-example        lda    4 0.00000000 0.00000000     0.003
## 5 iris-example        lda    5 0.13333333 0.12500000     0.002
## 6 iris-example        lda    6 0.06666667 0.08333333     0.002
```

```
perfdf <- reshape2::melt(perf, id.vars = c("task.id", "learner.id", "iter"))
head(perfdf)
```

```
##        task.id learner.id iter variable      value
## 1 iris-example        lda    1     mmce 0.00000000
## 2 iris-example        lda    2     mmce 0.00000000
## 3 iris-example        lda    3     mmce 0.00000000
## 4 iris-example        lda    4     mmce 0.00000000
## 5 iris-example        lda    5     mmce 0.13333333
## 6 iris-example        lda    6     mmce 0.06666667
```

We plot boxplots for `mmce` and `timetrain` for each task.

```
perfdf <- perfdf[perfdf$variable != "ber", ]
qplot(variable, value, data = perfdf, colour = learner.id, geom = "boxplot",
      xlab = "measure", ylab = "performance") +
  facet_wrap(~ task.id, nrow = 2)
```

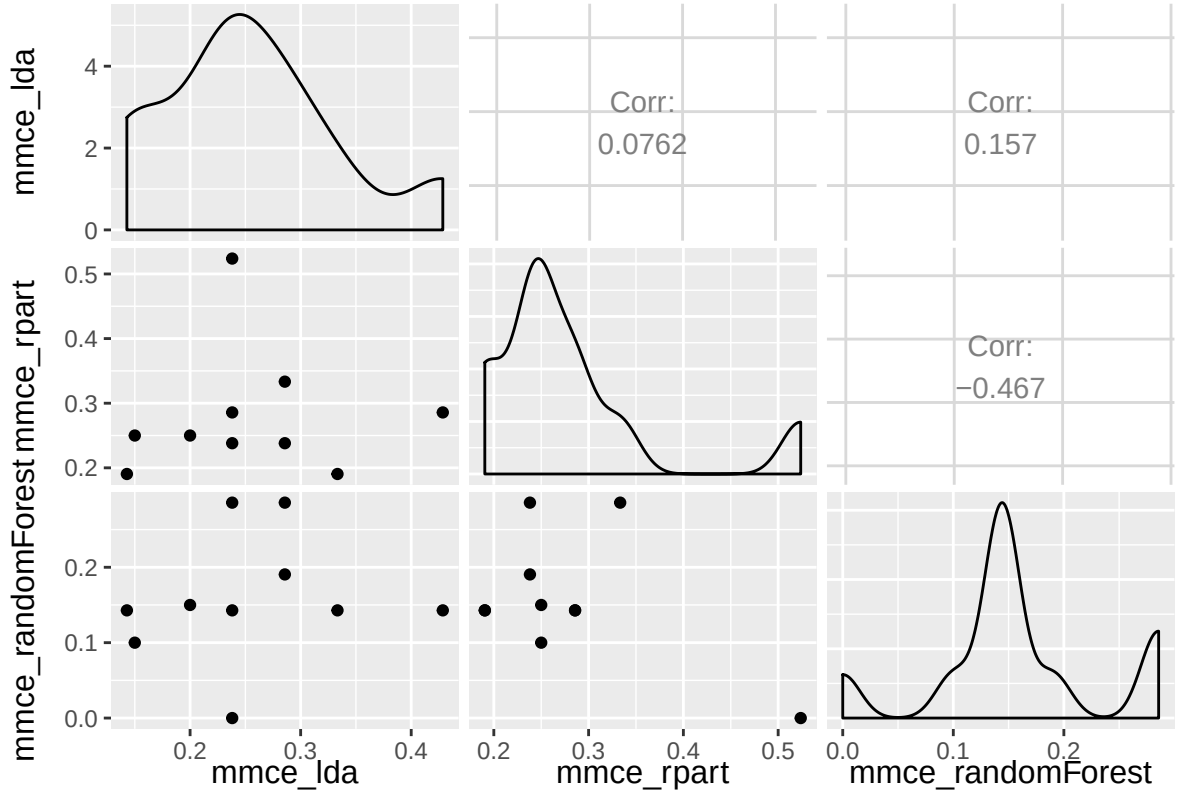


Further insight may be gained on learner performance by comparing the performance in each fold for a particular task; one learner could be performing exceptionally well in a single iteration while another may be performing exceptionally bad. We do this on the sonar data below by collecting the misclassification errors computed by the three learners we used.

```
perf <- getBMRPerformances(bmr, task.ids = "Sonar-example", as.df = TRUE)
df <- reshape2::melt(perf, id.vars = c("task.id", "learner.id", "iter"))
df <- df[df$variable == "mmce",]
df <- reshape2::dcast(df, task.id + iter ~ variable + learner.id)
head(df)
```

```
##         task.id iter  mmce_lda mmce_rpart mmce_randomForest
## 1 Sonar-example    1 0.1500000  0.2500000         0.1000000
## 2 Sonar-example    2 0.2857143  0.2380952         0.1904762
## 3 Sonar-example    3 0.2380952  0.2857143         0.1428571
## 4 Sonar-example    4 0.2380952  0.2380952         0.2857143
## 5 Sonar-example    5 0.1428571  0.1904762         0.1428571
## 6 Sonar-example    6 0.2857143  0.3333333         0.2857143
```

```
# scatterplot matrix
GGally::ggpairs(df, 3:5)
```



Parallelization

A number of instances are parallelizable with mlr. Parallelization is activated using `parallelMap::parallelStart`, the first loop mlr encounters (which is parallel executable) will be automatically parallelized.

```
# parallelStartSocket(2)
parallelStartMulticore(cpus = 2) # better for macosx?
```

```
## Starting parallelization in mode=multicore with cpus=2.
```

```
rdesc <- makeResampleDesc("CV", iters = 3)
r <- resample("classif.lda", task = iris.task, rdesc)
```

```
## Mapping in parallel: mode = multicore; cpus = 2; elements = 3.
```

```
## [Resample] Result: mmce.test.mean=0.04
```

```
parallelStop()
```

```
## Stopped parallelization. All cleaned up.
```

The `parallelStart` functions have an optional argument `levels`; by setting it, we can control which level gets parallelized. For instance, we may be running a few benchmark experiments which use an elaborate resampling description. In such a case, it would be best to have the resampling parallelized and not the benchmark so we could set `level = "mlr.resample"` in the `parallelStart` function. Not all levels are supported, the current version (2.8) supports `mlr.benchmark`, `mlr.resample`, `mlr.selectFeatures`, and `mlr.tuneParams`.

For custom learners, they need to be exported to the slave before calling the `parallelStart` functions:

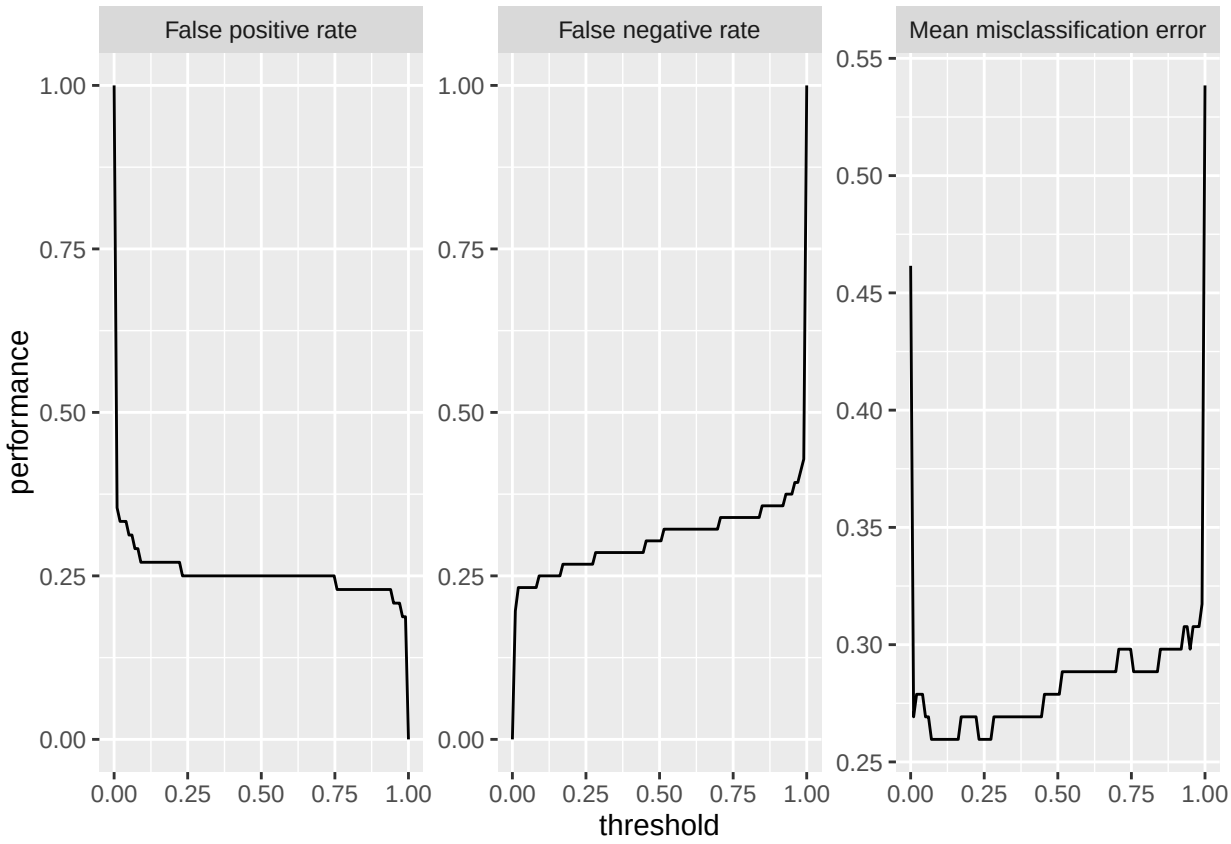
```
parallelExport("trainLearner.regr.<myregrlearner>", "predictLearner.regr.<myregrlearner>")
```

Visualizations We have already seen various `generate` functions within mlr which generate data that can then be used for plotting and visualization using `plotting` functions. Let us revisit the binary classification task with the sonar data set and plot the classifier performance against the boundary decision threshold.

```
lrn <- makeLearner("classif.lda", predict.type = "prob")
n <- getTaskSize(sonar.task)
mod <- mlr::train(learner = lrn, task = sonar.task, subset = seq(1, n, by = 2))
pred <- predict(mod, task = sonar.task, subset = seq(2, n, by = 2))
d <- generateThreshVsPerfData(pred, measures = list(fpr, fnr, mmce))
head(d$data)
```

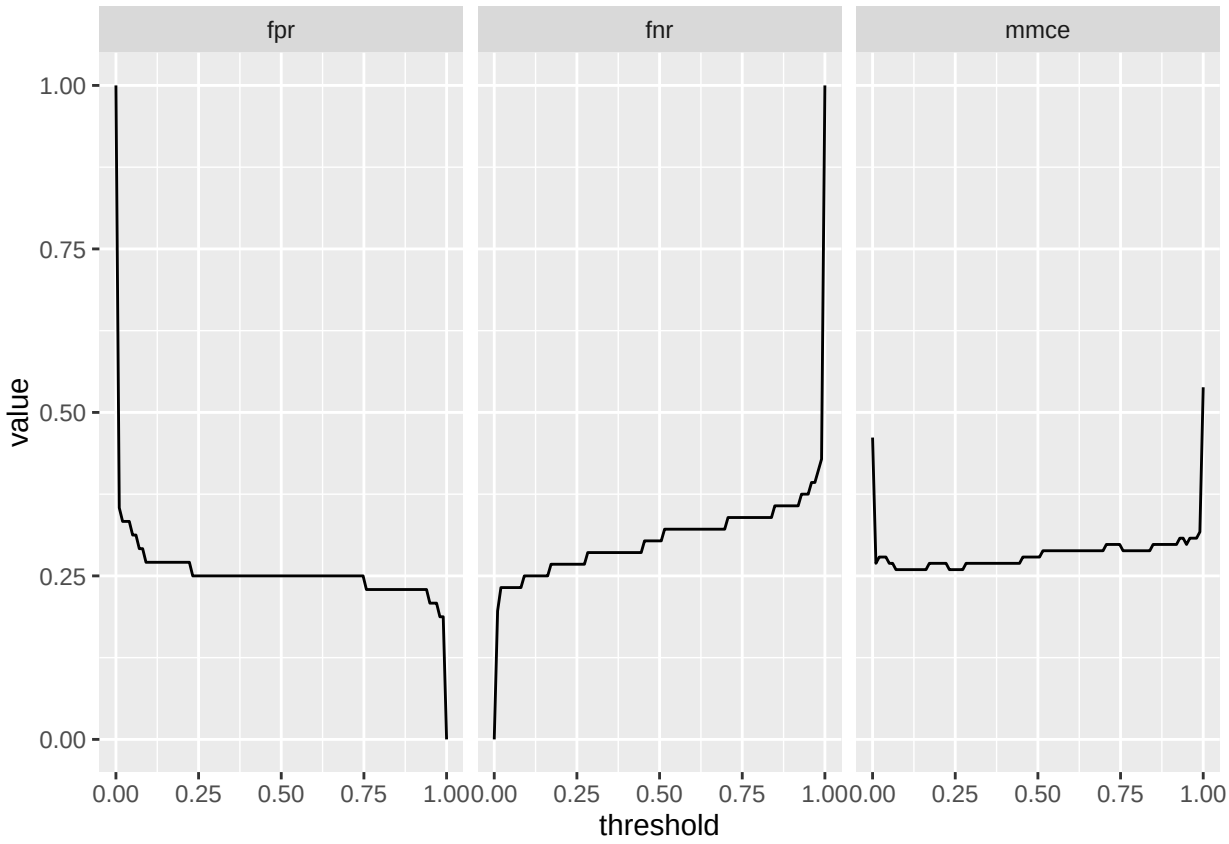
```
##         fpr       fnr      mmce  threshold
## 1 1.0000000 0.0000000 0.4615385 0.00000000
## 2 0.3541667 0.1964286 0.2692308 0.01010101
## 3 0.3333333 0.2321429 0.2788462 0.02020202
## 4 0.3333333 0.2321429 0.2788462 0.03030303
## 5 0.3333333 0.2321429 0.2788462 0.04040404
## 6 0.3125000 0.2321429 0.2692308 0.05050505
```

```
plotThreshVsPerf(d) #GGVIS is also possible but currently experimental
```



Alternatively, the plot can be manually created using the data generated from the `generate` function. Using ggplot we show this below.

```
dnew <- reshape2::melt(data = d$data, id.vars= c("threshold"))
ggplot(data = dnew, aes(threshold, value)) + geom_line() + facet_grid(. ~ variable,
                                                                      scales = "free_y")
```

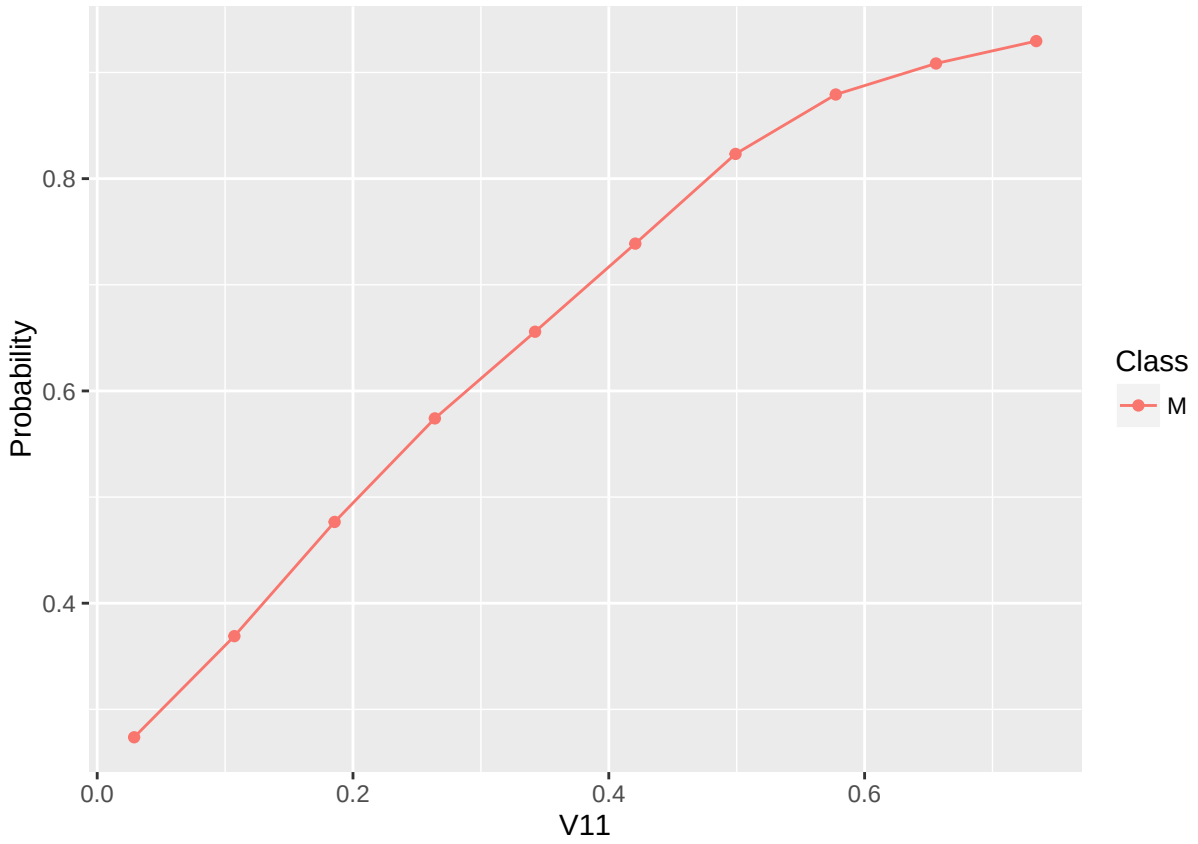


In this second example, we `generate` partial data on what we are interested in visualizing from the fitted model using `generatePartialPredictionData` and then create plot using `plotPartialPrediction`.

```
sonar <- getTaskData(sonar.task)
pd <- generatePartialPredictionData(mod, input = sonar, features = "V11")
plt <- plotPartialPrediction(pd)
head(plt$data)
```

```
##   Class Probability Feature     Value
## 1     M   0.9295997     V11 0.7342000
## 2     M   0.9084961     V11 0.6558333
## 3     M   0.8792694     V11 0.5774667
## 4     M   0.8232852     V11 0.4991000
## 5     M   0.7387962     V11 0.4207333
## 6     M   0.6557857     V11 0.3423667
```

```
plt
```



Advanced

Wrappers

Introductory example It is possible to add pre-processing, missing values imputation, tuning, feature selection and other functionality to a learner through the use of wrappers. Each time a wrapper is used around an mlr learner, a new learner is returned and this procedure can be repeated many times.

As a first example we use a bagging wrapper to create a random forest which supports weights and compare it to the unwrapped learner (`base.lrn` below).

```
data(iris)
task <- makeClassifTask(data = iris, target = "Species", weights = as.integer(iris$Species))
base.lrn <- makeLearner("classif.rpart")
```

The `makeBaggingWrapper` function takes as inputs the learner we want wrapped, the number of iterations which is the number of fitted models in bagging (for `rpart` learner, this would be `ntree` = N base learners), an option for sampling with/without replacement, proportion of randomly selected features (this would be the equivalent of `mtry`, here we set it at .5)

```
wrapped.lrn <- makeBaggingWrapper(learner = base.lrn, bw.iters = 100, bw.feats = 0.5 )
print(wrapped.lrn)
```

```
## Learner classif.rpart.bagged from package rpart
## Type: classif
```

```
## Name: ; Short name:
## Class: BaggingWrapper
## Properties: twoclass,multiclass,missings,numerics,factors,ordered,prob,weights
## Predict-Type: response
## Hyperparameters: xval=0,bw.iters=100,bw.feats=0.5
```

Next, we run a `benchmark` experiment with the list of learners comprised of the base and wrapped learners. Note that the default resampling strategy is 10-fold cross-validation.

```
bmr <- benchmark(learners = list(base.lrn, wrapped.lrn), task = task)
```

```
## Task: iris, Learner: classif.rpart

## [Resample] cross-validation iter: 1

## [Resample] cross-validation iter: 2

## [Resample] cross-validation iter: 3

## [Resample] cross-validation iter: 4

## [Resample] cross-validation iter: 5

## [Resample] cross-validation iter: 6

## [Resample] cross-validation iter: 7

## [Resample] cross-validation iter: 8

## [Resample] cross-validation iter: 9

## [Resample] cross-validation iter: 10

## [Resample] Result: mmce.test.mean=0.0867

## Task: iris, Learner: classif.rpart.bagged

## [Resample] cross-validation iter: 1

## [Resample] cross-validation iter: 2

## [Resample] cross-validation iter: 3

## [Resample] cross-validation iter: 4

## [Resample] cross-validation iter: 5

## [Resample] cross-validation iter: 6
```

```
## [Resample] cross-validation iter: 7

## [Resample] cross-validation iter: 8

## [Resample] cross-validation iter: 9

## [Resample] cross-validation iter: 10

## [Resample] Result: mmce.test.mean=0.0533
```

```
bmr
```

```
##   task.id           learner.id mmce.test.mean
## 1    iris        classif.rpart     0.08666667
## 2    iris classif.rpart.bagged     0.05333333
```

We may choose to carry out hyperparameter tuning to hopefully improve on the performance of the new learner. The function `makeTuneWrapper` fuses a learner with a search strategy to select its hyperparameters. This is set in effect when `train` is called (see below). First the algorithm for the hyperparameter optimization must be chosen - this is done via the `makeTuneControl` functions. A grid algorithm is traditional but it may be computationally expensive in high-dimensional spaces. In this example we only explore a small parameter space (we tune `minsplit` and `bw.feats`) with *random search*. The `minsplit` parameter is the minimum number of observations that must exist in a node in order for a split to be attempted and `bw.feats` was discussed above. In the random search, we choose to have sampled settings randomly selected 10 times (`maxit = 10`)

```
ctrl <- makeTuneControlRandom(maxit = 10)
rdesc <- makeResampleDesc("CV", iters = 3)
par.set <- makeParamSet(makeIntegerParam("minsplit", lower = 1, upper = 10),
                        makeNumericParam("bw.feats", lower = 0.25, upper = 1))

tuned.lrn <- makeTuneWrapper(learner = wrapped.lrn, resampling = rdesc,
                             measures = mmce, par.set = par.set, control = ctrl)
print(tuned.lrn)
```

```
## Learner classif.rpart.bagged.tuned from package rpart
## Type: classif
## Name: ; Short name:
## Class: TuneWrapper
## Properties: numerics,factors,ordered,missings,weights,prob,twoclass,multiclass
## Predict-Type: response
## Hyperparameters: xval=0,bw.iters=100,bw.feats=0.5
```

Now we have a learner object like before but this learner internally used `tuneParams` such as when called with `train`, the search strategy and resampling are invoked to choose an optimal set of parameters. The output from `train` is a tuned, bagged learner.

```
lrn <- mlr::train(learner = tuned.lrn, task = task)
```

```
## [Tune] Started tuning learner classif.rpart.bagged for parameter set:
```

```
##             Type len Def    Constr Req Tunable Trafo
## minsplit integer   -   -   1 to 10   -    TRUE     -
## bw.feats numeric   -   - 0.25 to 1   -    TRUE     -

## With control class: TuneControlRandom

## Imputation value: 1

## [Tune-x] 1: minsplit=6; bw.feats=0.432

## [Tune-y] 1: mmce.test.mean=0.04; time: 0.1 min; memory: 208Mb use, 545Mb max

## [Tune-x] 2: minsplit=10; bw.feats=0.449

## [Tune-y] 2: mmce.test.mean=0.04; time: 0.1 min; memory: 208Mb use, 545Mb max

## [Tune-x] 3: minsplit=4; bw.feats=0.429

## [Tune-y] 3: mmce.test.mean=0.04; time: 0.1 min; memory: 208Mb use, 545Mb max

## [Tune-x] 4: minsplit=4; bw.feats=0.771

## [Tune-y] 4: mmce.test.mean=0.0467; time: 0.1 min; memory: 208Mb use, 545Mb max

## [Tune-x] 5: minsplit=2; bw.feats=0.803

## [Tune-y] 5: mmce.test.mean=0.0467; time: 0.1 min; memory: 208Mb use, 545Mb max

## [Tune-x] 6: minsplit=5; bw.feats=0.569

## [Tune-y] 6: mmce.test.mean=0.0467; time: 0.1 min; memory: 208Mb use, 545Mb max

## [Tune-x] 7: minsplit=8; bw.feats=0.483

## [Tune-y] 7: mmce.test.mean=0.0533; time: 0.1 min; memory: 208Mb use, 545Mb max

## [Tune-x] 8: minsplit=10; bw.feats=0.362

## [Tune-y] 8: mmce.test.mean=0.0733; time: 0.1 min; memory: 208Mb use, 545Mb max

## [Tune-x] 9: minsplit=8; bw.feats=0.268

## [Tune-y] 9: mmce.test.mean=0.06; time: 0.1 min; memory: 208Mb use, 545Mb max

## [Tune-x] 10: minsplit=8; bw.feats=0.62

## [Tune-y] 10: mmce.test.mean=0.0467; time: 0.1 min; memory: 208Mb use, 545Mb max

## [Tune] Result: minsplit=10; bw.feats=0.449 : mmce.test.mean=0.04
```

```
print(lrn)
```

```
## Model for learner.id=classif.rpart.bagged.tuned; learner.class=TuneWrapper
## Trained on: task.id = iris; obs = 150; features = 4
## Hyperparameters: xval=0,bw.iters=100,bw.feats=0.5
```

```
getTuneResult(lrn)
```

```
## Tune result:
## Op. pars: minsplit=10; bw.feats=0.449
## mmce.test.mean=0.04
```

Data preprocessing The mlr package offers many options for data preprocessing. Some of them are directly applied to a task to modify it. These include `capLargeValues`, `createDummyFeatures` (for factor variables), `dropFeatures`, `joinClassLevels`, `mergeSmallFactorLevels`, `normalizeFeatures`, `removeConstantFeatures`, and `subsetTask` which are all pretty self explanatory from their name.

With the mlr wrapper functionality, the preprocessing steps are done at the time the learner is trained or predictions are made. This is important as it avoids the mistake of integrating processing steps which are data-dependent on the whole data set while the learner is trained only on training/test sets. A more honest performance of the learner is obtained if all preprocessing steps are included in the resampling. This is automatically done when fusing a learner with preprocessing.

`makePreprocWrapperCaret` permits access to all preprocessing options offered by `caret`'s `preProcess` function while`makePreprocWrapper` is used when writing custom preprocessing methods by defining the actions to be taken before training and before prediction. With these two functions, the preprocessing steps then belong to the learner. This is in contrast to the functions defined above (e.g `normalizeFeatures`) which alters the task. So, here, the task remains unchanged, preprocessing is done for every pair of training/test sets in resampling and not globally on the whole data set, and any parameters pertinent to preprocessing can be tuned with the learner's parameters.

Firstly, the usage of`makePreprocWrapperCaret` is similar to `caret`'s `preProcess` in the sense that it takes almost all of the formal arguments though their names are prefixed by `ppc.`, e.g. `knnImpute` becomes `ppc.knnImpute`. Secondly, there is no `method` argument; instead, the preprocessing options are passed as individual logical arguments, i.e. `pcc.knnImpute = TRUE`.

For knn imputation and pca in caret:

```
preProcess(x, method = c("knnImpute", "pca"), pcaComp = 10)
```

With the mlr wrapper:

```
makePreprocWrapperCaret(learner, ppc.knnImpute = TRUE, ppc.pca = TRUE, ppc.pcaComp = 10)
```

We show an example where we apply PCA on the sonar data (this poses a binary classification problem with 208 obs and 60 features) for dimensionality reduction. The `makePreprocWrapperCaret` function below is used to fuse quadratic discriminant analysis with PCA preprocessing where we set the threshold for PCA to retain 90 % (cumulative percent) of variance. Note that the data is automatically standardized prior to applying PCA.

```
lrn <- makePreprocWrapperCaret(learner = "classif.qda", ppc.pca = TRUE, ppc.thresh = 0.9)
lrn
```

```
## Learner classif.qda.preproc from package MASS
## Type: classif
```

```
## Name: ; Short name:
## Class: PreprocWrapperCaret
## Properties: twoclass,multiclass,numerics,factors,prob
## Predict-Type: response
## Hyperparameters: ppc.BoxCox=FALSE,ppc.YeoJohnson=FALSE,ppc.expoTrans=FALSE,ppc.center=TRUE,ppc.scale=
```

Now, that we have a wrapped learner, calling `train` with the task will train with the principal components returned after PCA has been applied.

```
mod <- mlr::train(lrn, sonar.task)
mod$learner.model
```

```
## Model for learner.id=classif.qda; learner.class=classif.qda
## Trained on: task.id = Sonar-example; obs = 208; features = 22
## Hyperparameters:
```

```
# or, for more info:
getLearnerModel(model = mod, more.unwrap = TRUE)
```

```
## Call:
## qda(f, data = getTaskData(.task, .subset, recode.target = "drop.levels"))
##
## Prior probabilities of groups:
##         M         R
## 0.5336538 0.4663462
##
## Group means:
##          PC1        PC2        PC3         PC4         PC5         PC6
## M  0.5976122 -0.8058235  0.9773518  0.03794232 -0.04568166 -0.06721702
## R -0.6838655  0.9221279 -1.1184128 -0.04341853  0.05227489  0.07691845
##          PC7         PC8        PC9       PC10        PC11          PC12
## M  0.2278162 -0.01034406 -0.2530606 -0.1793157 -0.04084466 -0.0004789888
## R -0.2606969  0.01183702  0.2895848  0.2051963  0.04673977  0.0005481212
##          PC13       PC14        PC15        PC16        PC17        PC18
## M -0.06138758 -0.1057137  0.02808048  0.05215865 -0.07453265  0.03869042
## R  0.07024765  0.1209713 -0.03213333 -0.05968671  0.08528994 -0.04427460
##          PC19         PC20        PC21         PC22
## M -0.01192247  0.006098658  0.01263492 -0.001224809
## R  0.01364323 -0.006978877 -0.01445851  0.001401586
```

Finally, we carry out a benchmark experiment to explore whether preprocessing with PCA has improved the performance of qda. We choose `stratify = TRUE` in resampling due to each class being represented by a small number of observations.

```
rin <- makeResampleInstance("CV", iters = 3, stratify = TRUE, task = sonar.task)
bmr <- benchmark(learners = list(makeLearner("classif.qda"), lrn), tasks = sonar.task, resamplings = rin
bmr
```

```
##         task.id          learner.id mmce.test.mean
## 1 Sonar-example         classif.qda      0.4035887
## 2 Sonar-example classif.qda.preproc      0.1733609
```

So far so good; it seems that PCA preprocessing is beneficial for the qda. However, the `thresh` value was chosen somewhat arbitrarily and resulted in 22 principal components. Perhaps with further tuning of the parameter, we can improve the performance of qda. Preprocessing and learner parameters can be tuned jointly. Calling `getParamSet` on the wrapped learner gives us all options.

```
getParamSet(lrn)
```

```
##                       Type len     Def                       Constr Req
## ppc.BoxCox         logical   -   FALSE                            -   -
## ppc.YeoJohnson     logical   -   FALSE                            -   -
## ppc.expoTrans      logical   -   FALSE                            -   -
## ppc.center         logical   -    TRUE                            -   -
## ppc.scale          logical   -    TRUE                            -   -
## ppc.range          logical   -   FALSE                            -   -
## ppc.knnImpute      logical   -   FALSE                            -   -
## ppc.bagImpute      logical   -   FALSE                            -   -
## ppc.medianImpute   logical   -   FALSE                            -   -
## ppc.pca            logical   -   FALSE                            -   -
## ppc.ica            logical   -   FALSE                            -   -
## ppc.spatialSign    logical   -   FALSE                            -   -
## ppc.thresh         numeric   -    0.95                    0 to Inf   -
## ppc.pcaComp        integer   -       -                    1 to Inf   -
## ppc.na.remove      logical   -    TRUE                            -   -
## ppc.k              integer   -       5                    1 to Inf   -
## ppc.fudge          numeric   -     0.2                    0 to Inf   -
## ppc.numUnique      integer   -       3                    1 to Inf   -
## method            discrete   -  moment            moment,mle,mve,t   -
## nu                 numeric   -       5                    2 to Inf   Y
## predict.method    discrete   - plug-in plug-in,predictive,debiased   -
##                   Tunable Trafo
## ppc.BoxCox           TRUE     -
## ppc.YeoJohnson       TRUE     -
## ppc.expoTrans        TRUE     -
## ppc.center           TRUE     -
## ppc.scale            TRUE     -
## ppc.range            TRUE     -
## ppc.knnImpute        TRUE     -
## ppc.bagImpute        TRUE     -
## ppc.medianImpute     TRUE     -
## ppc.pca              TRUE     -
## ppc.ica              TRUE     -
## ppc.spatialSign      TRUE     -
## ppc.thresh           TRUE     -
## ppc.pcaComp          TRUE     -
## ppc.na.remove        TRUE     -
## ppc.k                TRUE     -
## ppc.fudge            TRUE     -
## ppc.numUnique        TRUE     -
## method               TRUE     -
## nu                   TRUE     -
## predict.method       TRUE     -
```

In what follows, we tune the number of principal components (instead of `ppc.thresh`) and we try two different ways to estimate the posterior probabilities in qda: *plug-in estimates* and *unbiased estimates*. This

is controled via the `predict.method` parameter. The hyperparameter tuning is done via a *grid search* this time with a resolution of 10 (consider finer resolutions in real problems).

```
ps <- makeParamSet(
  makeIntegerParam("ppc.pcaComp", lower = 1, upper = getTaskNFeats(sonar.task)),
  makeDiscreteParam("predict.method", values = c("plug-in", "debiased"))
)

ctrl <- makeTuneControlGrid(resolution = 10)
res <- tuneParams(lrn, sonar.task, rin, par.set = ps, control = ctrl, show.info = FALSE)
res
```

```
## Tune result:
## Op. pars: ppc.pcaComp=14; predict.method=plug-in
## mmce.test.mean=0.168
```

The following example shows how to create a custom preprocessing wrapper using `makePreprocWrapper` which adds a scaling option to a learner by coupling it with the function `scale` (note this is actually possible through `makePreprocWrapperCaret`). Since wrappers are implemented using a *train* and *predict* method, we specify custom train and predict functions. The *train* function has to return a list with the preprocessed data set (`$data`) and an element which stores the information required to preprocess the data before prediction (`$control`). In our example, `$control` stores the scaling parameters which are to be used in the prediction.

```
trainfun <- function(data, target, args = list(center, scale)){
  cns <- colnames(data)
  # identify num data - exclude target
  nums <- setdiff(cns[sapply(data, is.numeric)], target)
  # extract numerical feats
  x <- as.matrix(data[, nums, drop = FALSE])
  x <- scale(x, center = args$center, scale = args$scale)
  # store the scaling parameters in control, needed to preprocess the data before preds.
  control <- args
  if (is.logical(control$center) && control$center)
    control$center = attr(x, "scaled:center")
  if (is.logical(control$scale) && control$scale)
    control$scale = attr(x, "scaled:scale")
  # recombine the data
  data <- data[, setdiff(cns, nums), drop = FALSE]
  data <- cbind(data, as.data.frame(x))
  return(list(data = data, control = control))
}
```

Next, the *predict* function takes in the data (without target variable), the name of the target variable, the`args` that were passed to `trainfun` and the `control` object returned by `trainfun`.

```
predictfun <- function(data, target, args, control){
  cns <- colnames(data)
  nums <- cns[sapply(data, is.numeric)]
  x <- as.matrix(data[, nums, drop = FALSE])
  x <- scale(x, center = control$center, scale = control$scale)
  data <- data[, setdiff(cns, nums), drop = FALSE]
  data <- cbind(data, as.data.frame(x))
  return(data)
}
```

Now we are going to use the functions we specified in a preprocessing wrapper. We use a regression neural network which does not have a scaling option and couple it with our own center + scale.

```
lrn <- makeLearner("regr.nnet", trace = FALSE, decay = 1e-02)
lrn <- makePreprocWrapper(lrn, train = trainfun, predict = predictfun,
                          par.vals = list(center = TRUE, scale = TRUE))
lrn
```

```
## Learner regr.nnet.preproc from package nnet
## Type: regr
## Name: ; Short name:
## Class: PreprocWrapper
## Properties: numerics,factors,weights
## Predict-Type: response
## Hyperparameters: size=3,trace=FALSE,decay=0.01
```

We now compare the performance of a `nnet` with and without scaling using the cross-validated mean squared error as a measure.

```
rdesc <- makeResampleDesc("CV", iters = 3)
r <- resample(lrn, bh.task, rdesc, measures = mse, show.info = FALSE)
r
```

```
## Resample Result
## Task: BostonHousing-example
## Learner: regr.nnet.preproc
## mse.aggr: 25.05
## mse.mean: 25.05
## mse.sd: 8.63
## Runtime: 0.093406
```

And without scaling.

```
lrn <- makeLearner("regr.nnet", trace = FALSE, decay = 1e-02)
r <- resample(lrn, bh.task, rdesc, measures = mse, show.info = FALSE)
r
```

```
## Resample Result
## Task: BostonHousing-example
## Learner: regr.nnet
## mse.aggr: 58.09
## mse.mean: 58.09
## mse.sd: 29.99
## Runtime: 0.058126
```

Joint tuning of preprocessing and learner parameters Usually we have some idea of what preprocessing we want to apply but it's not clear what options work best for each algorithm. It is possible to tune the preprocesssing and learner parameters as we have previously done.

```
lrn <- makeLearner("regr.nnet", trace = FALSE)
lrn <- makePreprocWrapper(lrn, train = trainfun, predict = predictfun,
                          par.set = makeParamSet(
                            makeLogicalLearnerParam("center"),
                            makeLogicalLearnerParam("scale")
                          ),
                          par.vals = list(center = TRUE, scale = TRUE)
                          )
lrn
```

```
## Learner regr.nnet.preproc from package nnet
## Type: regr
## Name: ; Short name:
## Class: PreprocWrapper
## Properties: numerics,factors,weights
## Predict-Type: response
## Hyperparameters: size=3,trace=FALSE,center=TRUE,scale=TRUE
```

```
getParamSet(lrn)
```

```
##             Type len    Def      Constr Req Tunable Trafo
## center   logical   -      -           -   -    TRUE     -
## scale    logical   -      -           -   -    TRUE     -
## size     integer   -      3    0 to Inf   -    TRUE     -
## maxit    integer   -    100    1 to Inf   -    TRUE     -
## linout   logical   -  FALSE           -   Y    TRUE     -
## entropy  logical   -  FALSE           -   Y    TRUE     -
## softmax  logical   -  FALSE           -   Y    TRUE     -
## censored logical   -  FALSE           -   Y    TRUE     -
## skip     logical   -  FALSE           -   -    TRUE     -
## rang     numeric   -    0.7 -Inf to Inf   -    TRUE     -
## decay    numeric   -      0    0 to Inf   -    TRUE     -
## Hess     logical   -  FALSE           -   -    TRUE     -
## trace    logical   -   TRUE           -   -   FALSE     -
## MaxNWts  integer   -   1000    1 to Inf   -    TRUE     -
## abstoll  numeric   - 0.0001 -Inf to Inf   -    TRUE     -
## reltoll  numeric   -  1e-08 -Inf to Inf   -    TRUE     -
```

We now tune via a grid search the decay parameter for the learner as well as the center and scale parameters for the preprocessing.

```
rdesc <- makeResampleDesc("Holdout")
ps <- makeParamSet(
  makeLogicalParam("center"),
  makeLogicalParam("scale"),
  makeDiscreteParam("decay", c(0, 0.05, 0.1))
)
ctrl <- makeTuneControlGrid()
res <- tuneParams(lrn, bh.task, rdesc, par.set = ps, control = ctrl, show.info = FALSE)
res
```

```
## Tune result:
```

```
## Op. pars: center=FALSE; scale=TRUE; decay=0.1
## mse.test.mean=8.95
```

```
as.data.frame(res$opt.path)
```

```
##    center scale decay mse.test.mean dob eol error.message exec.time
## 1    TRUE  TRUE     0     17.830530   1  NA          <NA>     0.034
## 2   FALSE  TRUE     0     83.387119   2  NA          <NA>     0.016
## 3    TRUE FALSE     0     66.564141   3  NA          <NA>     0.022
## 4   FALSE FALSE     0     83.387119   4  NA          <NA>     0.021
## 5    TRUE  TRUE  0.05     13.830167   5  NA          <NA>     0.031
## 6   FALSE  TRUE  0.05      9.457274   6  NA          <NA>     0.036
## 7    TRUE FALSE  0.05     54.261502   7  NA          <NA>     0.036
## 8   FALSE FALSE  0.05     14.007771   8  NA          <NA>     0.030
## 9    TRUE  TRUE   0.1     26.572742   9  NA          <NA>     0.031
## 10  FALSE  TRUE   0.1      8.949573  10  NA          <NA>     0.032
## 11   TRUE FALSE   0.1     50.246538  11  NA          <NA>     0.031
## 12  FALSE FALSE   0.1     51.864713  12  NA          <NA>     0.028
```

Imputation of missing values

In the mlr package, methods of imputation of missing values include imputation by a constant (mean, median, mode or some other constant), random numbers (some distribution), or based on predictions by a supervised learner. The possibility of custom imputation methods is also available. Note that some of the learning algorithms in mlr which have the `missings` property integrated, can deal with missing values in a sensible way (i.e. not simply deleting observations). Typing `listLearners("regr", properties = "missings")[c("class", "package")]` will give info on those packages if they are installed.

We first look at a simple example making use of the function `impute` on the `airquality` data set. To further demonstrate the functionality of `impute`, we add some missing values in the `Wind` column and then coerce into a factor using `cut`.

```
data("airquality")
airq <- airquality
ind <- sample(nrow(airquality), 10)
airq$Wind[ind] <- NA
airq$Wind <- cut(airq$Wind, c(0, 8, 16, 24))
summary(airq)
```

```
##      Ozone           Solar.R          Wind         Temp      
##  Min.   :  1.00   Min.   :  7.0   (0,8]  :49   Min.   :56.00  
##  1st Qu.: 18.00   1st Qu.:115.8   (8,16] :87   1st Qu.:72.00  
##  Median : 31.50   Median :205.0   (16,24]: 7   Median :79.00  
##  Mean   : 42.13   Mean   :185.9   NA's   :10   Mean   :77.88  
##  3rd Qu.: 63.25   3rd Qu.:258.8                3rd Qu.:85.00  
##  Max.   :168.00   Max.   :334.0                Max.   :97.00  
##  NA's   :37       NA's   :7                                   
##      Month            Day       
##  Min.   :5.000   Min.   : 1.0  
##  1st Qu.:6.000   1st Qu.: 8.0  
##  Median :7.000   Median :16.0  
##  Mean   :6.993   Mean   :15.8
```

```
##  3rd Qu.:8.000   3rd Qu.:23.0
##  Max.   :9.000   Max.   :31.0
##
```

```
sapply(airq, class)
```

```
##     Ozone   Solar.R      Wind      Temp     Month       Day
## "integer" "integer"  "factor" "integer" "integer" "integer"
```

The `Ozone` and `Solar.R` variables are of class integer and the `Wind` variable is of class factor. We choose to impute the integer features by the mean and the factor feature by the mode. We also choose to create dummy variables for all integer features indicating which observations were missing.

```
imp <- impute(airq, classes = list(integer = imputeMean(), factor = imputeMode()), dummy.classes = "int
head(imp$data)
```

```
##      Ozone  Solar.R   Wind Temp Month Day Ozone.dummy Solar.R.dummy
## 1 41.00000 190.0000  (0,8]   67     5   1       FALSE         FALSE
## 2 36.00000 118.0000  (0,8]   72     5   2       FALSE         FALSE
## 3 12.00000 149.0000 (8,16]   74     5   3       FALSE         FALSE
## 4 18.00000 313.0000 (8,16]   62     5   4       FALSE         FALSE
## 5 42.12931 185.9315 (8,16]   56     5   5        TRUE          TRUE
## 6 28.00000 185.9315 (8,16]   66     5   6       FALSE          TRUE
```

The function `impute` returns an object with the elements `$data` and `$desc`; the latter stores the information for the imputation. Note that "Imputed" refers to the class of features for which an imputation method was specified (here, 5 integers + 1 factor) and not the features which contain NA values. Here, the target variable was not specified. Next, let us look at a learning task example where the target variable is involved. In the `airquality` data set, we wish to predict the ozone pollution based on meteorological features. Our dataset therefore will have 4 columns: `Ozone`, `Solar.R`, `Wind`, and `Temp`.

```
# prepare data set
# remove Day and Month
airq <- dplyr::select(airq, -c(Day, Month))
# choose first 100 obs to make up the training set
airq.train <- airq[1:100, ]
airq.test <- airq[-c(1:100), ]
```

In this example, we do the following:

1) impute missing values in `Solar.R` with random numbers drawn from an empirical distribution;

2) Use function `imputeLearner`, which allows the user to use all supervised learning algorithms integrated in mlr, to impute missing values in the factor variable `Wind`. With `imputeLearner`, we need to create a learner whose type must match the variable to be imputed. So, here we go for `classif.<learning_algorithm>` since our feature variable is a factor. Then, all columns apart from the one imputed and the target variable are used as features in the learning algorithm chosen. Note that there are algorithms that can deal with missing values so in our example, to impute `NA`'s in `Wind` using say, a classification tree, only `Temp` and `Solar.R` will be used as features in the tree. The `rpart` algorithm will be able to handle the missing values in `Solar.R`.

```
imp <- impute(data = airq.train, target = "Ozone", cols = list(Solar.R = imputeHist(), Wind = imputeLea
imp$desc
```

```
## Imputation description
## Target: Ozone
## Features: 3; Imputed: 2
## impute.new.levels: TRUE
## recode.factor.levels: TRUE
## dummy.type: factor
```

```
summary(imp$data)
```

```
##      Ozone           Solar.R           Wind         Temp
##  Min.   :  1.00   Min.   :  7.0   (0,8]  :35   Min.   :56.00
##  1st Qu.: 16.00   1st Qu.:113.8   (8,16] :59   1st Qu.:69.00
##  Median : 34.00   Median :221.5   (16,24]: 6   Median :79.50
##  Mean   : 41.59   Mean   :192.0                Mean   :76.87
##  3rd Qu.: 63.00   3rd Qu.:274.2                3rd Qu.:84.00
##  Max.   :135.00   Max.   :334.0                Max.   :93.00
##  NA's   :31
##  Solar.R.dummy Wind.dummy
##  FALSE:93      FALSE:95
##  TRUE : 7      TRUE : 5
##
##
##
##
##
```

To impute the test data the same way as the train data, we can simply use the `imp$desc` object with the function `reimpute`.

```
airq.test.imp <- reimpute(airq.test, desc = imp$desc)
head(airq.test.imp)
```

```
##   Ozone Solar.R   Wind Temp Solar.R.dummy Wind.dummy
## 1   110     207  (0,8]   90         FALSE       TRUE
## 2    NA     222 (8,16]   92         FALSE      FALSE
## 3    NA     137 (8,16]   86         FALSE      FALSE
## 4    44     192 (8,16]   86         FALSE      FALSE
## 5    28     273 (8,16]   82         FALSE      FALSE
## 6    65     157 (8,16]   80         FALSE      FALSE
```

Fuse a learner with imputation When creating a learner with `makeImputeWrapper`, before training the resulting learner, impute is applied to the training set and, subsequently, before prediction, `reimpute` is called on the test set using the `$desc` object from the training stage. In what follows, we ask for the same imputation as above and we choose a linear regression model as the learner.

```
lrn <- makeImputeWrapper(learner = "regr.lm", cols = list(Solar.R = imputeHist(), Wind = imputeLearner(
                         )
lrn
```

```
## Learner regr.lm.preproc from package stats
## Type: regr
## Name: ; Short name:
## Class: ImputeWrapper
## Properties: numerics,factors,se,weights,missings
## Predict-Type: response
## Hyperparameters:
```

To create a task, we need to delete any observations from the target variable (`Ozone`) which are missing.

```
airq <- subset(airq, subset = !is.na(airq$Ozone))
task <- makeRegrTask(data = airq, target = "Ozone")
rdesc <- makeResampleDesc("CV", iters = 3)
r <- resample(lrn, task, rdesc, show.info = FALSE, models = TRUE)
r$aggr
```

```
## mse.test.mean
##       595.099
```

```
lapply(r$models, getLearnerModel, more.unwrap = TRUE)
```

```
## [[1]]
##
## Call:
## stats::lm(formula = f, data = d)
##
## Coefficients:
##       (Intercept)            Solar.R         Wind(8,16]
##         -108.75925            0.06951           -22.46964
##       Wind(16,24]               Temp  Solar.R.dummyTRUE
##           -7.89778            1.94691           -24.58073
##    Wind.dummyTRUE
##           13.95148
##
##
## [[2]]
##
## Call:
## stats::lm(formula = f, data = d)
##
## Coefficients:
##       (Intercept)            Solar.R         Wind(8,16]
##           -50.35825            0.07453           -37.09466
##       Wind(16,24]               Temp  Solar.R.dummyTRUE
##           -34.44710            1.34312            -2.86339
##    Wind.dummyTRUE
##           10.29853
##
##
## [[3]]
##
## Call:
```

```
## stats::lm(formula = f, data = d)
##
## Coefficients:
##       (Intercept)            Solar.R         Wind(8,16]
##         -87.13995            0.05501           -13.82936
##       Wind(16,24]               Temp  Solar.R.dummyTRUE
##         -10.97308            1.60398            -2.54234
##    Wind.dummyTRUE
##           4.16069
```

Note that `makePreprocWrapperCaret` is also available for fusion with imputation and a learner but it is somewhat limited.

Generic Bagging

With `makeBaggingWrapper`, it is possible to bag an mlr learner to use bagging as a technique to gain more stability. The `makeBaggingWrapper` function takes arguments which decide on the subsets to be chosen for each iteration of the bagging process. So, just like in `randomForest`, we need to train a learner on a subset of data.

- `bw.iters`: the number of subsets (samples) we want to train our learner
- `bw.replace`: logical. sample with replacement (bootstrapping) or without
- `bw.size`: percentage size of sampled bags
- `bw.feats`: percentage size of randomly selected features in bags

In the example below, we compare the performance of a pruned tree (using `RWeka`'s `PART`) with and without bagging on the `Sonar` data.

```
lrn <- makeLearner("classif.rpart")
bag.lrn <- makeBaggingWrapper(lrn, bw.iters = 50, bw.replace = TRUE, bw.size = 0.8, bw.feats = 3/4)
rdesc <- makeResampleDesc("CV", iters = 5)
r <- resample(learner = lrn, task = sonar.task, resampling = rdesc, show.info = FALSE)
r$aggr
```

```
## mmce.test.mean
##      0.3267131
```

```
rdesc <- makeResampleDesc("CV", iters = 5)
r2 <- resample(learner = bag.lrn, task = sonar.task, resampling = rdesc, show.info = FALSE)
r2$aggr
```

```
## mmce.test.mean
##      0.2260163
```

Changing the type of prediction Using `setPredictType`, we can change the type of prediction. For classification problems we can either have labels (these are determined by majority voting over the preductions of the individual models) or posterior class probabilities. Note that the predict type for the base learner always has to be "response".

```
bag.lrn <- setPredictType(bag.lrn, predict.type = "prob")
```

In the case of a regression problem, the options for prediction type are numeric, response or standard errors.

```
n <- getTaskSize(bh.task)
train.inds <- seq(1, n, 3)
test.inds <- setdiff(1:n, train.inds)
lrn <- makeLearner("regr.rpart")
bag.lrn <- makeBaggingWrapper(learner = lrn) # using defaults for the rest
bag.lrn <- setPredictType(bag.lrn, predict.type = "se")
mod <- mlr::train(bag.lrn, bh.task, subset = train.inds)
head(getLearnerModel(mod), 2)
```

```
## [[1]]
## Model for learner.id=regr.rpart; learner.class=regr.rpart
## Trained on: task.id = BostonHousing-example; obs = 169; features = 13
## Hyperparameters: xval=0
##
## [[2]]
## Model for learner.id=regr.rpart; learner.class=regr.rpart
## Trained on: task.id = BostonHousing-example; obs = 169; features = 13
## Hyperparameters: xval=0
```

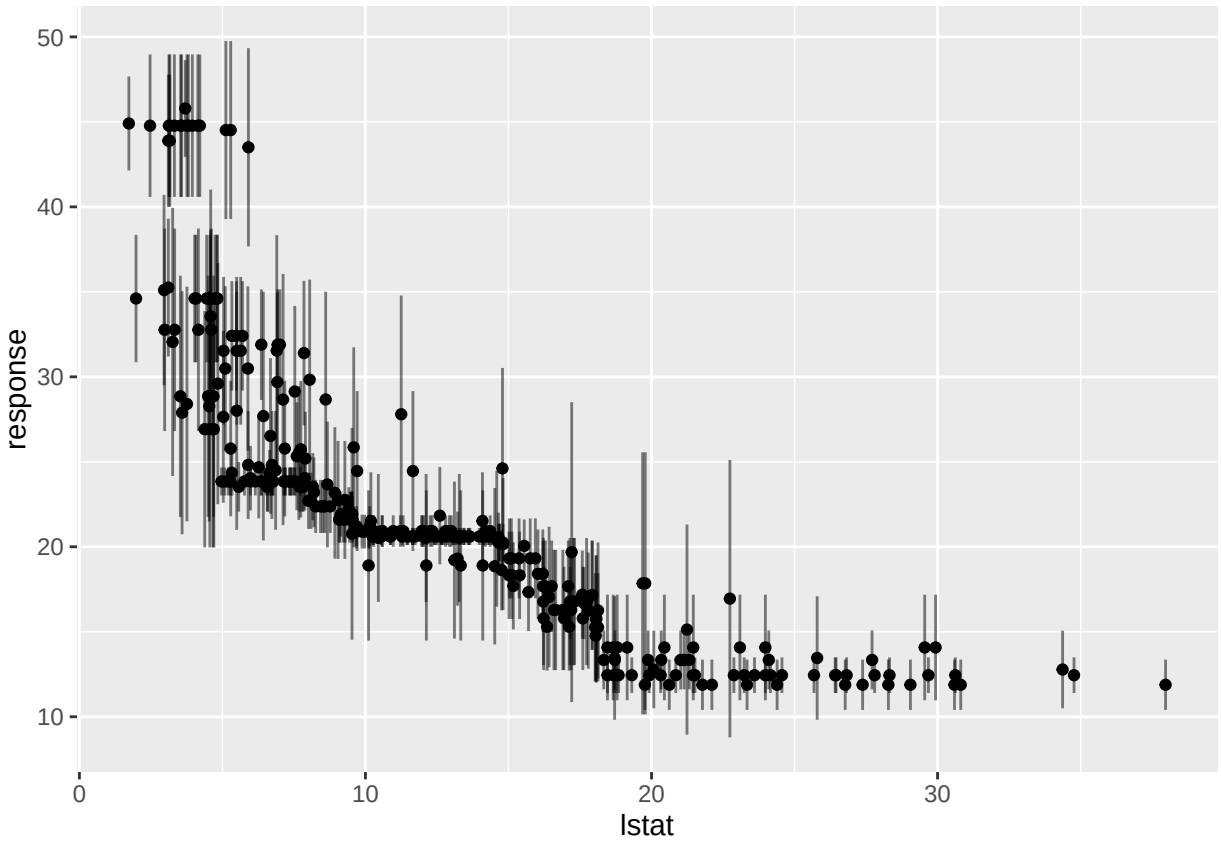
Next we predict the response and calculate the standard deviation for each prediction:

```
pred <- predict(mod, task = bh.task, subset = test.inds)
head(as.data.frame(pred))
```

```
##   id truth response       se
## 2  2  21.6 21.91884 1.320067
## 3  3  34.7 34.60776 3.748879
## 5  5  36.2 32.40824 3.217443
## 6  6  28.7 23.84344 0.823360
## 8  8  27.1 14.07722 3.103786
## 9  9  16.5 14.07722 3.103786
```

We can then plot the percentage of lower status of the population against the predicted response (predicted `medv`) with its calculated standard errors using `ggplot2`.

```
data <- cbind(as.data.frame(pred), getTaskData(bh.task, subset = test.inds))
g <- ggplot(data, aes(x = lstat, y = response, ymin = response - se, ymax = response + se), col = age)
g + geom_point() + geom_linerange(alpha = 0.5)
```



Tuning

We can set selected hyperparameters in machine learning algorithms by passing them on to `makeLearner`. By tuning the hyperparameters, we can automatically identify values that lead to the best performance. In order to do the tuning, we need to specify the *search space*, the *optimization algorithm*, and the *evaluation method* (i.e. a resampling strategy + a performance measure).

Grid search

1. Create the `ParamSet` object which describes the parameter space we want to search.

```
ps <- makeParamSet(
  makeDiscreteParam("C", values = 2^(-2:2)),
  makeDiscreteParam("sigma", values = 2^(-2:2))
)
```

2. Create the `TuneControl` object which describes the optimization strategy to be used and its settings.

```
ctrl <- makeTuneControlGrid()
```

3. Create the resampling description.

```
rdesc <- makeResampleDesc("CV", iters = 3L)
```

4. Tuning the parameters with `tuneParams`.

```
result <- tuneParams("classif.ksvm", task = iris.task, resampling = rdesc, par.set = ps, control = ctrl
```

```
## [Tune] Started tuning learner classif.ksvm for parameter set:

##           Type len Def         Constr Req Tunable Trafo
## C     discrete   -   - 0.25,0.5,1,2,4   -    TRUE     -
## sigma discrete   -   - 0.25,0.5,1,2,4   -    TRUE     -

## With control class: TuneControlGrid

## Imputation value: 1

## [Tune-x] 1: C=0.25; sigma=0.25

## [Tune-y] 1: mmce.test.mean=0.0467; time: 0.0 min; memory: 160Mb use, 545Mb max

## [Tune-x] 2: C=0.5; sigma=0.25

## [Tune-y] 2: mmce.test.mean=0.0467; time: 0.0 min; memory: 160Mb use, 545Mb max

## [Tune-x] 3: C=1; sigma=0.25

## [Tune-y] 3: mmce.test.mean=0.0333; time: 0.0 min; memory: 160Mb use, 545Mb max

## [Tune-x] 4: C=2; sigma=0.25

## [Tune-y] 4: mmce.test.mean=0.04; time: 0.0 min; memory: 160Mb use, 545Mb max

## [Tune-x] 5: C=4; sigma=0.25

## [Tune-y] 5: mmce.test.mean=0.0467; time: 0.0 min; memory: 160Mb use, 545Mb max

## [Tune-x] 6: C=0.25; sigma=0.5

## [Tune-y] 6: mmce.test.mean=0.0533; time: 0.0 min; memory: 160Mb use, 545Mb max

## [Tune-x] 7: C=0.5; sigma=0.5

## [Tune-y] 7: mmce.test.mean=0.04; time: 0.0 min; memory: 160Mb use, 545Mb max

## [Tune-x] 8: C=1; sigma=0.5

## [Tune-y] 8: mmce.test.mean=0.04; time: 0.0 min; memory: 160Mb use, 545Mb max
```

```
## [Tune-x] 9: C=2; sigma=0.5

## [Tune-y] 9: mmce.test.mean=0.0467; time: 0.0 min; memory: 160Mb use, 545Mb max

## [Tune-x] 10: C=4; sigma=0.5

## [Tune-y] 10: mmce.test.mean=0.0533; time: 0.0 min; memory: 160Mb use, 545Mb max

## [Tune-x] 11: C=0.25; sigma=1

## [Tune-y] 11: mmce.test.mean=0.06; time: 0.0 min; memory: 160Mb use, 545Mb max

## [Tune-x] 12: C=0.5; sigma=1

## [Tune-y] 12: mmce.test.mean=0.0467; time: 0.0 min; memory: 160Mb use, 545Mb max

## [Tune-x] 13: C=1; sigma=1

## [Tune-y] 13: mmce.test.mean=0.0533; time: 0.0 min; memory: 160Mb use, 545Mb max

## [Tune-x] 14: C=2; sigma=1

## [Tune-y] 14: mmce.test.mean=0.0533; time: 0.0 min; memory: 160Mb use, 545Mb max

## [Tune-x] 15: C=4; sigma=1

## [Tune-y] 15: mmce.test.mean=0.06; time: 0.0 min; memory: 160Mb use, 545Mb max

## [Tune-x] 16: C=0.25; sigma=2

## [Tune-y] 16: mmce.test.mean=0.08; time: 0.0 min; memory: 160Mb use, 545Mb max

## [Tune-x] 17: C=0.5; sigma=2

## [Tune-y] 17: mmce.test.mean=0.0667; time: 0.0 min; memory: 160Mb use, 545Mb max

## [Tune-x] 18: C=1; sigma=2

## [Tune-y] 18: mmce.test.mean=0.06; time: 0.0 min; memory: 160Mb use, 545Mb max

## [Tune-x] 19: C=2; sigma=2

## [Tune-y] 19: mmce.test.mean=0.0667; time: 0.0 min; memory: 160Mb use, 545Mb max

## [Tune-x] 20: C=4; sigma=2

## [Tune-y] 20: mmce.test.mean=0.0667; time: 0.0 min; memory: 160Mb use, 545Mb max
```

```
## [Tune-x] 21: C=0.25; sigma=4

## [Tune-y] 21: mmce.test.mean=0.107; time: 0.0 min; memory: 160Mb use, 545Mb max

## [Tune-x] 22: C=0.5; sigma=4

## [Tune-y] 22: mmce.test.mean=0.107; time: 0.0 min; memory: 160Mb use, 545Mb max

## [Tune-x] 23: C=1; sigma=4

## [Tune-y] 23: mmce.test.mean=0.0933; time: 0.0 min; memory: 160Mb use, 545Mb max

## [Tune-x] 24: C=2; sigma=4

## [Tune-y] 24: mmce.test.mean=0.0933; time: 0.0 min; memory: 160Mb use, 545Mb max

## [Tune-x] 25: C=4; sigma=4

## [Tune-y] 25: mmce.test.mean= 0.1; time: 0.0 min; memory: 160Mb use, 545Mb max

## [Tune] Result: C=1; sigma=0.25 : mmce.test.mean=0.0333
```

```
result
```

```
## Tune result:
## Op. pars: C=1; sigma=0.25
## mmce.test.mean=0.0333
```

The `show.info` argument can be set via `configureMlr`; the command `getMlrOptions()` shows the current settings. In the above example, `tuneParams` performs cross-validation for every element of the cross product (so in this case, we have 5 x 5 different combinations). The optimal parameters (which give the best mean performance) are selected in the end. In the example above, since no measure was selected, the default for classification was used (`mmce`). Alternatively, we can set a different measure; this is shown below.

```
result <- tuneParams("classif.ksvm", task = iris.task, resampling = rdesc, par.set = ps, control = ctrl
result
```

```
## Tune result:
## Op. pars: C=1; sigma=0.25
## acc.test.mean=0.947,acc.test.sd=0.0306
```

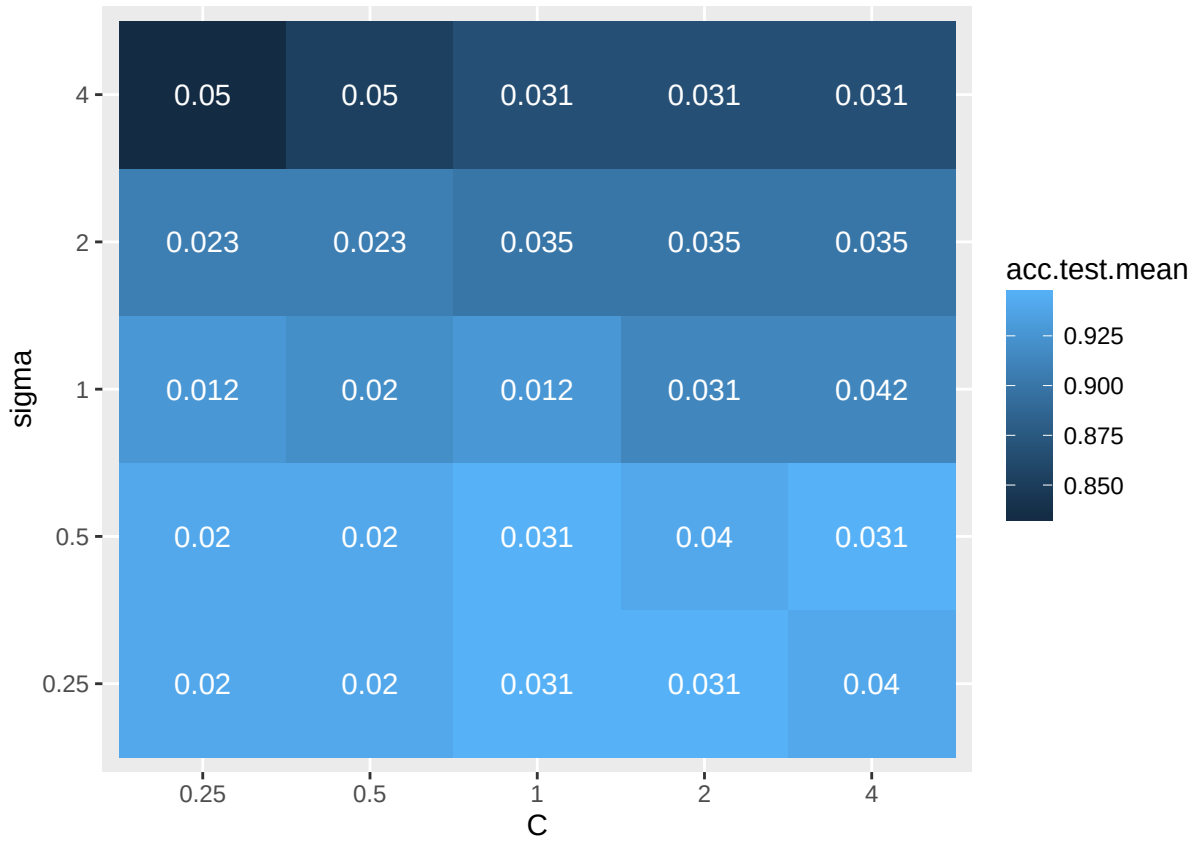
The object returned by `tuneParams` is a list which includes the best parameter settings (accessed via `$x`) and their estimated performance (accessed via `$y`). Through `$opt.path`, we can obtain the performance of all points evaluated.

```
opt.grid <- as.data.frame(result$opt.path)
opt.grid
```

```
##       C sigma acc.test.mean acc.test.sd dob eol error.message exec.time
## 1  0.25  0.25     0.9400000  0.02000000   1  NA          <NA>     0.044
## 2   0.5  0.25     0.9400000  0.02000000   2  NA          <NA>     0.040
## 3     1  0.25     0.9466667  0.03055050   3  NA          <NA>     0.049
## 4     2  0.25     0.9466667  0.03055050   4  NA          <NA>     0.045
## 5     4  0.25     0.9400000  0.04000000   5  NA          <NA>     0.039
## 6  0.25   0.5     0.9400000  0.02000000   6  NA          <NA>     0.043
## 7   0.5   0.5     0.9400000  0.02000000   7  NA          <NA>     0.040
## 8     1   0.5     0.9466667  0.03055050   8  NA          <NA>     0.046
## 9     2   0.5     0.9400000  0.04000000   9  NA          <NA>     0.047
## 10    4   0.5     0.9466667  0.03055050  10  NA          <NA>     0.043
## 11 0.25     1     0.9266667  0.01154701  11  NA          <NA>     0.043
## 12  0.5     1     0.9200000  0.02000000  12  NA          <NA>     0.045
## 13    1     1     0.9266667  0.01154701  13  NA          <NA>     0.045
## 14    2     1     0.9133333  0.03055050  14  NA          <NA>     0.052
## 15    4     1     0.9133333  0.04163332  15  NA          <NA>     0.042
## 16 0.25     2     0.9066667  0.02309401  16  NA          <NA>     0.049
## 17  0.5     2     0.9066667  0.02309401  17  NA          <NA>     0.043
## 18    1     2     0.9000000  0.03464102  18  NA          <NA>     0.044
## 19    2     2     0.9000000  0.03464102  19  NA          <NA>     0.043
## 20    4     2     0.9000000  0.03464102  20  NA          <NA>     0.043
## 21 0.25     4     0.8333333  0.05033223  21  NA          <NA>     0.047
## 22  0.5     4     0.8533333  0.05033223  22  NA          <NA>     0.047
## 23    1     4     0.8666667  0.03055050  23  NA          <NA>     0.047
## 24    2     4     0.8666667  0.03055050  24  NA          <NA>     0.045
## 25    4     4     0.8666667  0.03055050  25  NA          <NA>     0.047
```

Further, we can visualize a selected performance measure using `geom_tile()` from `ggplot2`. In the code below, we ask for the color of the tiles to represent the achieved accuracy while the labels on the tiles represent the standard deviation for each of the settings attempted.

```
g <- ggplot(opt.grid, aes(x = C, y = sigma, fill = acc.test.mean, label = round(acc.test.sd, 3)))
g + geom_tile() + geom_text(color = "white")
```



Using the optimal parameters Once we have the optimal hyperameter set, we generate a learner and pass the settings as an argument in `setHyperPars`.

```
lrn <- setHyperPars(makeLearner("classif.ksvm"), par.vals = result$x)
mod <- mlr::train(learner = lrn, task = iris.task)
predict(mod, task = iris.task)
```

```
## Prediction: 150 observations
## predict.type: response
## threshold:
## time: 0.00
##   id  truth response
## 1  1 setosa   setosa
## 2  2 setosa   setosa
## 3  3 setosa   setosa
## 4  4 setosa   setosa
## 5  5 setosa   setosa
## 6  6 setosa   setosa
```

Above, we used `mkeDiscreteParam` to discretize manually the parameters we wanted to set (here, `C` and `sigma`). We can also use their true numeric value and set `resolution = TRUE` in the control strategy to automatically discretize them. When setting the numerical parameter range, we can also pass a transformation function (`trafo`).

```
ps <- makeParamSet(
  makeNumericParam("C", lower = 12, upper = 12, trafo = function(x) 2^x),
  makeNumericParam("sigma", lower = -12, upper = 12)
)
ctrl <- makeTuneControlGrid(resolution = 3L)
rdesc <- makeResampleDesc("CV", iters = 2L)
res <- tuneParams("classif.ksvm", task = iris.task, resampling = rdesc, par.set = ps, control = ctrl)
```

```
## [Tune] Started tuning learner classif.ksvm for parameter set:
```

```
##          Type len Def    Constr Req Tunable Trafo
## C     numeric   -   -  12 to 12   -    TRUE     Y
## sigma numeric   -   - -12 to 12   -    TRUE     -
```

```
## With control class: TuneControlGrid
```

```
## Imputation value: 1
```

```
## [Tune-x] Setting hyperpars failed: Error in setHyperPars2.Learner(learner, insert(par.vals, args)) :
##   -12 is not feasible for parameter 'sigma'!
```

```
## [Tune-x] 1: C=4.1e+03; sigma=-12
```

```
## [Tune-y] 1: mmce.test.mean=  NA; time: 0.0 min; memory: 160Mb use, 545Mb max
```

```
## [Tune-x] 2: C=4.1e+03; sigma=0
```

```
## [Tune-y] 2: mmce.test.mean=0.693; time: 0.0 min; memory: 160Mb use, 545Mb max
```

```
## [Tune-x] 3: C=4.1e+03; sigma=12
```

```
## [Tune-y] 3: mmce.test.mean=0.147; time: 0.0 min; memory: 160Mb use, 545Mb max
```

```
## [Tune] Result: C=4.1e+03; sigma=12 : mmce.test.mean=0.147
```

```
res
```

```
## Tune result:
## Op. pars: C=4.1e+03; sigma=12
## mmce.test.mean=0.147
```

Note that calling `res$opt.path` as before returns the parameter values on the original scale before the transformation applied as set in the `trafo` option. The transformed parameters can be retrieved using `trafoOptPath`, as shown below.

```
# original scale
as.data.frame(res$opt.path)
```

```
##    C sigma mmce.test.mean dob eol
## 1 12   -12             NA   1  NA
## 2 12     0      0.6933333   2  NA
## 3 12    12      0.1466667   3  NA
##
## 1 Error in setHyperPars2.Learner(learner, insert(par.vals, args)) : \n  -12 is not feasible for para
## 2
## 3
##   exec.time
## 1        NA
## 2     0.032
## 3     0.029
```

```
# transformed
as.data.frame(trafoOptPath(res$opt.path))
```

```
##      C sigma mmce.test.mean dob eol
## 1 4096   -12             NA   1  NA
## 2 4096     0      0.6933333   2  NA
## 3 4096    12      0.1466667   3  NA
```

The mlr package supports additional tuning algorithms to the traditional grid. In previous parts of this tutorial, we used `makeTuneControlRandom` which uses random search. An iterated F-racing algorithm is also included; the algorithm starts by considering a set of candidate parameters and completely discards any for which any statistical evidence arises against them. We show an example below with the `iris.task`. In this example, we experiment with different kernels in the svm algorithm. However, we wish to tune certain parameters with particular kernels; we can set this using the `requires` argument which states requirements on other parameters. For instance, below we make `sigma` and `degree` dependent on `rbfdot` and `polydot` kernels, respectively.

```
ps <- makeParamSet(
  makeNumericParam("C", lower = -12, upper = 12, trafo = function(x) 2^x),
  makeDiscreteParam("kernel", values = c("vanilladot", "polydot", "rbfdot")),
  makeNumericParam("sigma", lower = -12, upper = 12, trafo = function(x) 2^x,
                   requires = quote(kernel == "rbfdot")),
  makeIntegerParam("degree", lower = 2L, upper = 5L,
                   requires = quote(kernel == "polydot"))
)
ctrl <- makeTuneControlIrace(maxExperiments = 200L)
rdesc <- makeResampleDesc("Holdout")
res <- tuneParams(learner = "classif.ksvm", task = iris.task, resampling = rdesc, par.set = ps, control
head(as.data.frame(res$opt.pat))
```

```
##            C     kernel      sigma degree mmce.test.mean dob eol
## 1 11.0511248     rbfdot -11.814987     NA           0.08   1  NA
## 2  0.4836376    polydot         NA      2           0.06   2  NA
## 3  1.2211800     rbfdot   3.547916     NA           0.22   3  NA
## 4  5.2512287     rbfdot   4.942283     NA           0.40   4  NA
## 5 10.1821147 vanilladot         NA     NA           0.10   5  NA
## 6 -0.9499436 vanilladot         NA     NA           0.08   6  NA
##   error.message exec.time
## 1          <NA>     0.017
## 2          <NA>     0.018
```

```
## 3          <NA>     0.019
## 4          <NA>     0.016
## 5          <NA>     0.017
## 6          <NA>     0.016
```

```
res
```

```
## Tune result:
## Op. pars: C=169; kernel=rbfdot; sigma=0.000427
## mmce.test.mean=0.0356
```

```
head(as.data.frame(res$opt.path))
```

```
##            C     kernel      sigma degree mmce.test.mean dob eol
## 1 11.0511248     rbfdot -11.814987     NA           0.08   1  NA
## 2  0.4836376    polydot         NA      2           0.06   2  NA
## 3  1.2211800     rbfdot   3.547916     NA           0.22   3  NA
## 4  5.2512287     rbfdot   4.942283     NA           0.40   4  NA
## 5 10.1821147 vanilladot         NA     NA           0.10   5  NA
## 6 -0.9499436 vanilladot         NA     NA           0.08   6  NA
##   error.message exec.time
## 1          <NA>     0.017
## 2          <NA>     0.018
## 3          <NA>     0.019
## 4          <NA>     0.016
## 5          <NA>     0.017
## 6          <NA>     0.016
```

One step further with ModelMultiplexer We can tune over different models at the same time and set parameters for each model using `makeModelMultiplexerParameterSet`. Once more we look at the `iris.task` classification problem set and tune SVM and randomForest algorithms at the same time.

```
base.learners <- list(
  makeLearner("classif.ksvm"),
  makeLearner("classif.randomForest")
)
lrn <- makeModelMultiplexer(base.learners)
ps <- makeModelMultiplexerParamSet(lrn,
       makeNumericParam("sigma", lower = -12, upper = 12, trafo = function(x) 2^x),
       makeIntegerParam("ntree", lower = 1L, upper = 500L)
       )
rdesc <- makeResampleDesc("CV", iters = 2L)
ctrl <- makeTuneControlIrace(maxExperiments = 200L)
res <- tuneParams(lrn, iris.task, rdesc, par.set = ps, control = ctrl, show.info = FALSE)
res
```

```
## Tune result:
## Op. pars: selected.learner=classif.ksvm; classif.ksvm.sigma=0.403
## mmce.test.mean=0.0458
```

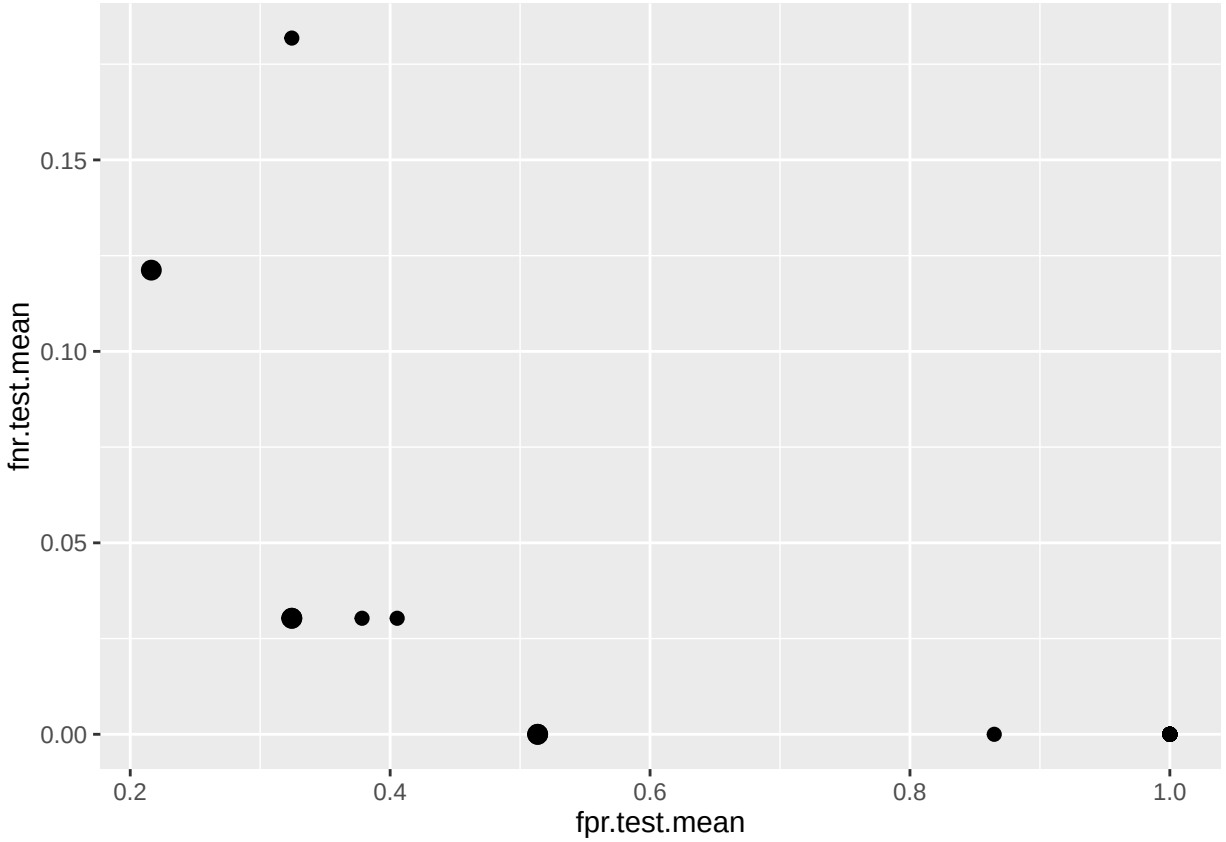
Control structures for multi-criteria tuning It is possible that we may want to optimize multiple performance measures simultaneously when tuning parameters. The mlr package offers multi-criteria tuning algorithms as control structures which are then passed onto `tuneParamsMultiCrit`. As an example, we tune SVM hyperparameters on the `sonar.task` classification task where we aim to optimize both `fpr` and `fnr`.

```
ps <- makeParamSet(
  makeNumericParam("C", lower = -12, upper = 12, trafo = function(x) 2^x),
  makeNumericParam("sigma", lower = -12, upper = 12, trafo = function(x) 2^x)
)

ctrl <- makeTuneMultiCritControlRandom(maxit = 30L)
rdesc <- makeResampleDesc("Holdout")
res <- tuneParamsMultiCrit("classif.ksvm", sonar.task, rdesc, measures = list(fpr, fnr), par.set = ps,
res
```

```
## Tune multicrit result:
## Points on front: 5
```

In the case of nontrivial multi-objective optimization problems, the objective functions are said to be conflicting. This could potentially result in the existence of a large number of optimal solutions. Here, as hyperparameters are tuned, the `fpr` and `fnr` measures are calculated and recorded. A *Pareto improvement* is said to occur if, the tuning of a certain parameter set, improves the lot of at least one of the measures. A solution is concerned *Pareto efficient* if no further Pareto improvements can be made. The Pareto efficient solutions make the **Pareto front**. The number of points on the front is the result shown when calling the object of the `tuneParamsMultiCrit`. Finally, the optimal solutions can be accessed using `res$x` (for the parameter sets) and `res$y` (for the estimated performance measures). All solutions cab be visualized using `plotTuneMultiCritResult` with the solutions on the Pareto front shown as bigger points (note that some - if more than one - may completely overlap).

```
plotTuneMultiCritResult(res)
```



Note that tuning works similarly in tasks other than classification and see also later section on *nested resampling* for unbiased performance estimation.

Feature Selection

The mlr package supports two different methods for feature selection which are discussed here in this section.

Filter methods Filter methods assign the features a value of **importance** which allows a ranking of the features which allows to subsequent subsetting of features to be used in a learning algorithm. A number of methods are available for calculating feature importance for classification, regression, and survival tasks (current support). The method is passed as an argument in `generateFilterValuesData`; the default is `rf.importance`.

```
fv <- generateFilterValuesData(task = iris.task, method = "information.gain")
fv
```

```
## FilterValues:
## Task: iris-example
##           name    type information.gain
## 1 Sepal.Length numeric        0.4521286
## 2  Sepal.Width numeric        0.2672750
## 3 Petal.Length numeric        0.9402853
## 4  Petal.Width numeric        0.9554360
```

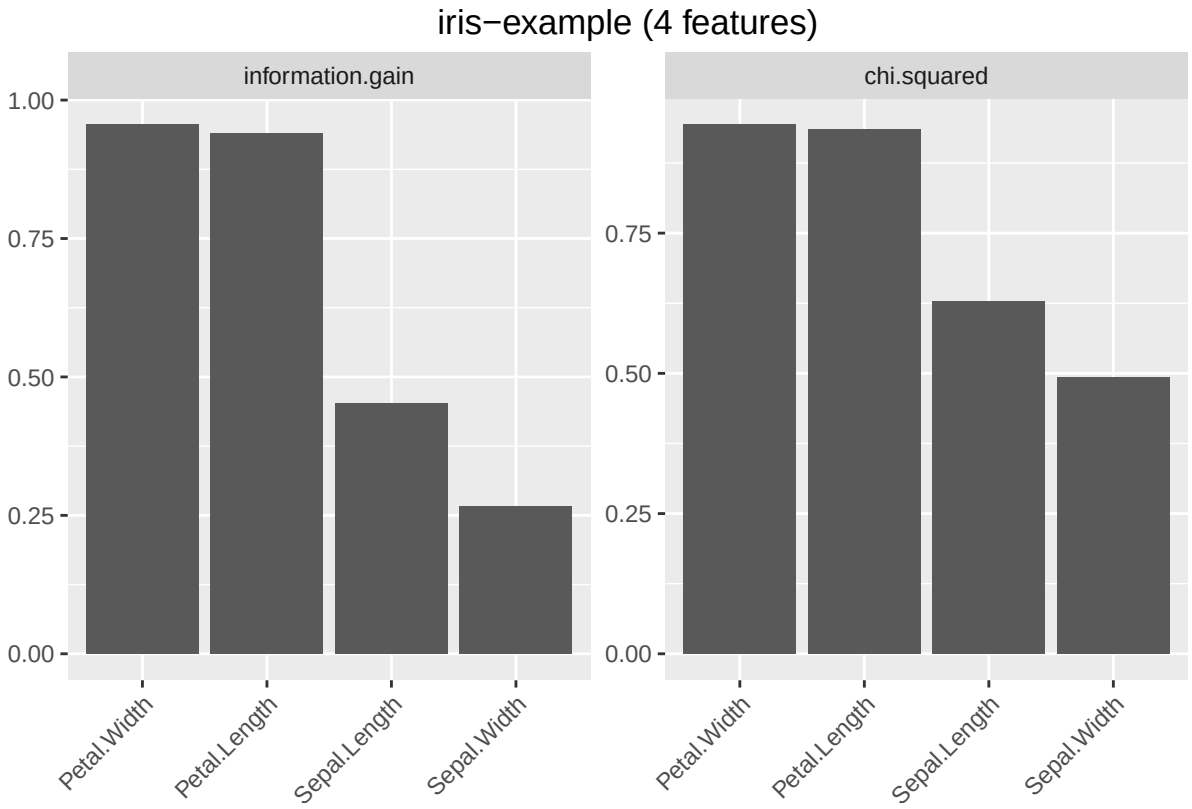
A vector of methods can also be supplied to obtain feature importance values for each specified method.

```
fv2 <- generateFilterValuesData(task = iris.task, method = c( "information.gain", "chi.squared"))
fv2
```

```
## FilterValues:
## Task: iris-example
##           name    type information.gain chi.squared
## 1 Sepal.Length numeric        0.4521286   0.6288067
## 2  Sepal.Width numeric        0.2672750   0.4922162
## 3 Petal.Length numeric        0.9402853   0.9346311
## 4  Petal.Width numeric        0.9554360   0.9432359
```

And these can be visualized by passing the resulting object into `plotFilterValues`.

```
plotFilterValues(fv2)
```



In this example, according to both measures, `Petal.Width` and `Petal.Length` contain the most information about the target variable `Species`.

Using this information, we can create a new task that includes a subset of the features specified by the user. To select the subset, we pass arguments to `filterFeatures`. We can choose from: `abs = K` where `K` is equal to some number of features to keep, `perc = r` where $0 <$ `r` < 1 is a percentage of the most important features to keep, and `threshold = m` where `m` denotes a threshold importance value (we keep features with values above `m`). In `filterFeatures`, we can either specify the method of calculating feature importance or, we can pass the object returned from `generateFilterValuesData` through the `fval` argument.

```
filtered.task <- filterFeatures(iris.task, method = "information.gain", abs = 2)
```

```
filtered.task <- filterFeatures(iris.task, fval = fv, perc = 0.25)

filtered.task <- filterFeatures(iris.task, fval = fv, threshold = 0.5)
filtered.task
```

```
## Supervised task: iris-example
## Type: classif
## Target: Species
## Observations: 150
## Features:
## numerics  factors  ordered
##        2        0        0
## Missings: FALSE
## Has weights: FALSE
## Has blocking: FALSE
## Classes: 3
##     setosa versicolor  virginica
##         50         50         50
## Positive class: NA
```

Fuse a learner with a filter method In an experiment with objectives to train a learner, we employ a resampling startegy to identify a validation method. Obviously feature selection is an important step contributing to the overall learner performance and we often want to carry out feature selection prior to each iteration identified in our resampling strategy. In what follows, we return the `iris.task` to train a fast k nearest neighbor classifier. The resampling strategy chosen is 10-fold cross-validation: in each iteration feature selection is applied via the filter method using `information.gain` as the measure and always keeping the 2 most important features.

```
lrn <- makeFilterWrapper(learner = "classif.fnn", fw.method = "information.gain", fw.abs = 2)
rdesc <- makeResampleDesc("CV", iters = 10)
r <- resample(lrn, iris.task, rdesc, show.info = FALSE, models = TRUE)
r$aggr
```

```
## mmce.test.mean
##     0.03333333
```

By calling `models = TRUE`, we store information on the models which allows us to access information like the two selected features for each model trained. We use `getFilteredFeatures` on each of the model fitted.

```
sfeats <- sapply(r$models, getFilteredFeatures)
table(sfeats)
```

```
## sfeats
## Petal.Length  Petal.Width
##           10           10
```

In this data set, the importance of `Petal.Length` and `Petal.Width` on `Species` seems to be very stable as they are consistently selected as the top 2 features.

Tuning the size of the feature subset Just as we tuned the hyperparameters of learning algorithms in the previous section, we can also tune the size of the feature subset. We pass on a control strategy and a range of values corresponding to the number/percentage of features or threshold into `tuneParams`.

```
lrn <- makeFilterWrapper(learner = "regr.lm", fw.method = "chi.squared")
ps <- makeParamSet(makeDiscreteParam("fw.perc", values = c(0.2, 0.5, 0.05)))
rdesc <- makeResampleDesc("CV", iters  = 3)
res <- tuneParams(lrn, bh.task, rdesc, par.set = ps, control = makeTuneControlGrid())
```

```
## [Tune] Started tuning learner regr.lm.filtered for parameter set:

##             Type len Def       Constr Req Tunable Trafo
## fw.perc discrete   -   - 0.2,0.5,0.05   -    TRUE     -

## With control class: TuneControlGrid

## Imputation value: Inf

## [Tune-x] 1: fw.perc=0.2

## [Tune-y] 1: mse.test.mean=34.6; time: 0.0 min; memory: 163Mb use, 545Mb max

## [Tune-x] 2: fw.perc=0.5

## [Tune-y] 2: mse.test.mean=28.5; time: 0.0 min; memory: 163Mb use, 545Mb max

## [Tune-x] 3: fw.perc=0.05

## [Tune-y] 3: mse.test.mean=54.6; time: 0.0 min; memory: 163Mb use, 545Mb max

## [Tune] Result: fw.perc=0.5 : mse.test.mean=28.5
```

```
res
```

```
## Tune result:
## Op. pars: fw.perc=0.5
## mse.test.mean=28.5
```

The `$x`, `$y`, and `$opt.path` elements work as we have seen them before. We can pass the tuned parameter to generate a new wrapped learner for further use.

```
lrn <- makeFilterWrapper("regr.lm", fw.method = "chi.squared", fw.perc = res$x$fw.perc)
mod <- mlr::train(lrn, bh.task)
mod
```

```
## Model for learner.id=regr.lm.filtered; learner.class=FilterWrapper
## Trained on: task.id = BostonHousing-example; obs = 506; features = 13
## Hyperparameters: fw.method=chi.squared,fw.perc=0.5
```

```
getFilteredFeatures(mod)
```

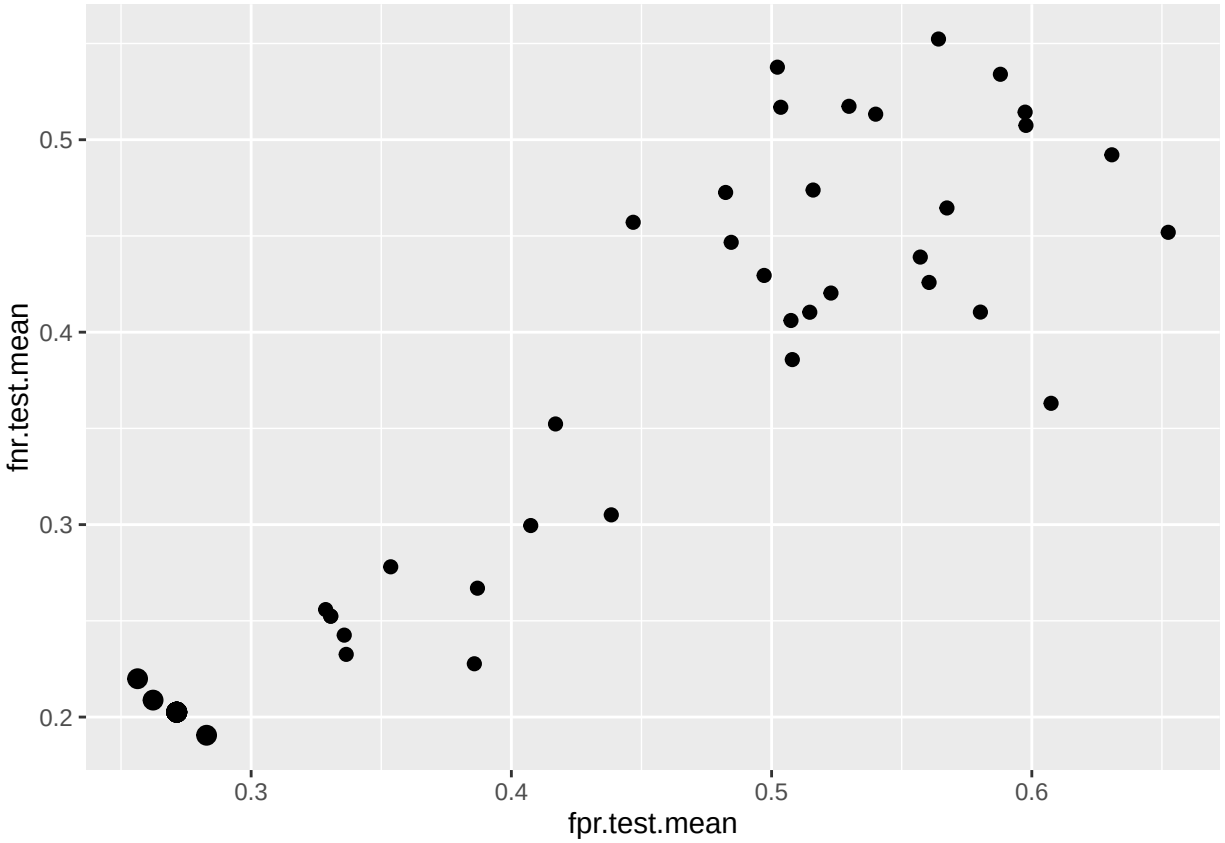
```
## [1] "crim"  "zn"    "rm"    "dis"   "rad"   "lstat"
```

We show another example using the `sonar.task` classification problem and make use of the multi-criteria tuning functions we introduced in the last section.

```
lrn <- makeFilterWrapper("classif.lda", fw.method = "chi.squared")
ps <- makeParamSet(makeNumericParam("fw.threshold", lower = 0.1, upper = 0.9))
rdesc <- makeResampleDesc("CV", iters = 10)
res <- tuneParamsMultiCrit(lrn, sonar.task, par.set = ps, resampling = rdesc, measures = list(fpr, fnr))
res
```

```
## Tune multicrit result:
## Points on front: 15
```

```
plotTuneMultiCritResult(res)
```



Wrapper methods In the previous subsection, feature selection was accomplished independently from the performance of a specific learning algorithm. We achieved optimal feature selection through maximizing or minimizing a criterion function. We now turn out attention to *wrapper* models where the effectiveness of the performance feature selection model is directly related to the performance of the learning algorithm, usually in terms of its predictive accuracy. Therefore, in the wrapper method, a learner is trained repeatedly on a different subset of features and the subset which leads to the learner with the best performance is chosen.

To do this, we need to specify which strategy we want to use to identify the various subsets to be used. Of course we also need to choose a learning algorithm and a resampling strategy by which we aim to assess performance.

A note on the search strategy for feature selection. The mlr package offers 4 methods:

1) `FeatSelControlExhaustive` : exhaustive search; all feature sets up to a certain number of `max.features` is searched.
2) `FeatSelControlRandom`: random search; feature vectors randomly drawn up to a certain number of `max.features`.
3) `FeatSelControlSequential`: deterministic forward or backward search. Forward (backward) search starts with smaller (bigger) feature subsets and adds (removes) sequentially.
4) `FeatSelControlGA`: genetic algorithm. An adaptive search technique based on the analogy with biology in which a set of possible solutions evolves via natural selection.

In the following example, we use the *Wisconsin Prognostic Breast Cancer* data set using a random search strategy.

```
ctrl <- makeFeatSelControlRandom(maxit = 20L)
ctrl
```

```
## FeatSel control: FeatSelControlRandom
## Same resampling instance: TRUE
## Imputation value: <worst>
## Max. features: <not used>
## Max. iterations: 20
## Tune threshold: FALSE
## Further arguments: prob=0.5
```

```
rdesc <- makeResampleDesc("Holdout")
sfeats <- selectFeatures(learner = "surv.coxph", task = wpbc.task, resampling = rdesc, control = ctrl, s
sfeats
```

```
## FeatSel result:
## Features (16): mean_radius, mean_perimeter, mean_smoothness, mean_concavepoints, mean_fractaldim, SE_
## cindex.test.mean=0.601
```

The selected features and corresponding performance are included in the object returned by `selectFeatures`.

```
sfeats$x
```

```
##  [1] "mean_radius"        "mean_perimeter"     "mean_smoothness"
##  [4] "mean_concavepoints" "mean_fractaldim"    "SE_radius"
##  [7] "SE_texture"         "SE_perimeter"       "SE_concavity"
## [10] "SE_symmetry"        "worst_texture"      "worst_perimeter"
## [13] "worst_compactness"  "worst_concavity"    "worst_fractaldim"
## [16] "pnodes"
```

```
sfeats$y
```

```
## cindex.test.mean
##         0.600639
```

We use the `bh.task` in a second example with forward sequential search. Features are added to the model until the performance improvement is smaller than some value (set by the `alpha` parameter).

```
ctrl <- makeFeatSelControlSequential(method = "sfs", alpha = 0.02)

rdesc <- makeResampleDesc("CV", iters = 10)
sfeats <- selectFeatures(learner = "regr.lm", task = bh.task, resampling = rdesc, control = ctrl, show.
sfeats
```

```
## FeatSel result:
## Features (11): crim, zn, chas, nox, rm, dis, rad, tax, ptratio, b, lstat
## mse.test.mean=23.4
```

Details of the sequential feature selection process can be retrieved using `analyzeFeatSelResult`:

```
analyzeFeatSelResult(sfeats)
```

```
## Features         : 11
## Performance      : mse.test.mean=23.4
## crim, zn, chas, nox, rm, dis, rad, tax, ptratio, b, lstat
##
## Path to optimum:
## - Features:    0  Init   :                       Perf = 84.717  Diff: NA  *
## - Features:    1  Add    : lstat                 Perf = 38.782  Diff: 45.934  *
## - Features:    2  Add    : rm                    Perf = 31.306  Diff: 7.4758  *
## - Features:    3  Add    : ptratio               Perf = 27.929  Diff: 3.3775  *
## - Features:    4  Add    : dis                   Perf = 27.054  Diff: 0.87535  *
## - Features:    5  Add    : nox                   Perf = 25.611  Diff: 1.4427  *
## - Features:    6  Add    : b                     Perf = 25.019  Diff: 0.59183  *
## - Features:    7  Add    : chas                  Perf = 24.586  Diff: 0.43327  *
## - Features:    8  Add    : zn                    Perf = 24.354  Diff: 0.23231  *
## - Features:    9  Add    : rad                   Perf = 24.215  Diff: 0.1382  *
## - Features:   10  Add    : tax                   Perf = 23.78  Diff: 0.4353  *
## - Features:   11  Add    : crim                  Perf = 23.377  Diff: 0.40282  *
##
## Stopped, because no improving feature was found.
```

Fuse a learner with a wrapper method Similarly to what we did above with fusing a learner with the filter, we can do so here with the wrapper. We fuse the feature selection and resampling strategies with a learner. The learner is trained on the selected feature subset.

```
rdesc <- makeResampleDesc("CV", iters = 3)
lrn <- makeFeatSelWrapper(learner = "surv.coxph", resampling = rdesc, control = makeFeatSelControlRando
mod <- train(lrn, wpbc.task)
mod
```

```
## Model for learner.id=surv.coxph.featsel; learner.class=FeatSelWrapper
## Trained on: task.id = wpbc-example; obs = 194; features = 32
## Hyperparameters:
```

```
sfeats <- getFeatSelResult(mod)
sfeats
```

```
## FeatSel result:
## Features (20): mean_radius, mean_texture, mean_area, mean_smoothness, mean_compactness, mean_symmetry
## cindex.test.mean=0.637
```

```
sfeats$x
```

```
##  [1] "mean_radius"         "mean_texture"        "mean_area"
##  [4] "mean_smoothness"     "mean_compactness"    "mean_symmetry"
##  [7] "mean_fractaldim"     "SE_radius"           "SE_smoothness"
## [10] "SE_compactness"      "SE_concavity"        "SE_symmetry"
## [13] "worst_radius"        "worst_perimeter"     "worst_smoothness"
## [16] "worst_concavity"     "worst_concavepoints" "worst_symmetry"
## [19] "worst_fractaldim"    "pnodes"
```

The 5-fold cross-validated performance of the learner we specified above can be computed through `resample`.

```
out.rdesc <- makeResampleDesc("CV", iters = 5)

r <- resample(learner = lrn, task = wpbc.task, resampling = out.rdesc, models = TRUE, show.info = FALSE)

r$aggr
```

```
## cindex.test.mean
##        0.6242516
```

With `models = TRUE` in `resample`, we can access the feature selection for each model by applying `getFeatSelResult`.

```
lapply(r$models, getFeatSelResult)
```

```
## [[1]]
## FeatSel result:
## Features (15): mean_texture, mean_area, mean_smoothness, mean_symmetry, SE_texture, SE_area, SE_compa
## cindex.test.mean=0.657
##
## [[2]]
## FeatSel result:
## Features (17): mean_texture, mean_area, mean_smoothness, mean_compactness, mean_fractaldim, SE_textu
## cindex.test.mean=0.669
##
## [[3]]
## FeatSel result:
## Features (18): mean_texture, mean_smoothness, mean_concavepoints, mean_symmetry, mean_fractaldim, SE_
## cindex.test.mean=0.683
##
## [[4]]
## FeatSel result:
## Features (19): mean_radius, mean_perimeter, mean_smoothness, mean_compactness, mean_concavepoints, me
```

```
## cindex.test.mean=0.656
##
## [[5]]
## FeatSel result:
## Features (13): mean_radius, mean_perimeter, mean_area, mean_fractaldim, SE_radius, SE_perimeter, SE_
## cindex.test.mean=0.735
```

Nested resampling

When we introduced *tuning*, we carry out the optimization over the same data. As a result, the estimated performance could be optimistically biased. To obtain an unbiased performance estimatation, we use **nested resampling**. In nested resampling, data preprocessing and feature selection is included in each training/test data resulting in an honest performance estimate. As an example consider a case where we want to train a learner with hyperparameter tuning. Suppose we choose 3-fold cross-validation as our resampling strategy; this creates three pairs of training/test sets which consist of the *outer resampling loop*. For *each* pair, we perform tuning with a new resampling strategy (say, 4-fold cross-validation), use the tuned parameters on the training set of that pair to fit a model and evaluate the model's performance on the test set. This gives one set of hyperparameters for each of the 3 pairs in the outer resampling.

The mlr package offers a way to do this nested resampling without actually programming any loops:

1) Generate a wrapped learner using `makeTuneWrapper` or `makeFeatSelWrapper` and specify the **inner** resampling strategy using the `resampling` argument.
2) Call function `resample` and pass the **outer** resampling startegy to its `resampling` argument.

Different inner and outer resampling strategies can be used. It is common to have the prediction and performance evaluation on a fixed outer set hence the outer strategy can be set using `makeFixedHoldoutInstance` to generate the outer resampling. For the inner resampling strategy, using `ResampleDesc` is preferable to `ResampleInstance`. The default setting is for the inner resampling strategy to be instantiated just once so tuning/feature selection are compared to the same inner training/test sets. However, if needed, this can be turned off by setting `same.resampling.instance = FALSE` in the `make` tuning/feature selection functions.

Parallelizing nested resampling is usually a good idea as it can be computationally expensive. However, for the purposes of this tutorial and to keep things fast, we keep the iterations in the loops, low.

Tuning We carry out nested resampling with tuning first. The objective is to train a learner using the `ksvm` learner class on a classification task (yes, the `iris` data of course!) and tune the hyperparameters `C` and `sigma`. To set this up, we need a search space and a search strategy, as before (see earlier section on *tuning*).

```
ps <- makeParamSet(
  makeDiscreteParam("C", values = 2^(-2:2)),
  makeDiscreteParam("sigma", values = 2^(-2:2))
)
ctrl <- makeTuneControlGrid() #using defaults
```

We choose *subsampling* as the inner strategy with 2 iterations (and the default split date of 2/3). In subsampling, the size of the training and test sets is independent of the number of folds (vs k-fold CV) but since the split is random, some obervations may never be selected and others may be selected many times. Further, the resampling here is done without replacement so each subsample of size b is a sample from the true, unknown distribution of the original sample. This is in contrast with bootstrap resampling where each sample (this is with replacement) has the same size as the original sample but is now from the empirical distribution associated with the original sample.

```
inner <- makeResampleDesc("Subsample", iters = 2)
lrn <- makeTuneWrapper("classif.ksvm", resampling = inner, par.set = ps, control = ctrl, show.info = FA
```

For the outer resampling loop, we choose 3-fold cross-validation.

```
outer <- makeResampleDesc("CV", iters = 3)
r <- resample(lrn, iris.task, resampling = outer, extract = getTuneResult, show.info = FALSE)
r
```

```
## Resample Result
## Task: iris-example
## Learner: classif.ksvm.tuned
## mmce.aggr: 0.06
## mmce.mean: 0.06
## mmce.sd: 0.04
## Runtime: 17.7222
```

```
#error rates on outer tests
r$measures.test
```

```
##   iter mmce
## 1    1 0.10
## 2    2 0.06
## 3    3 0.02
```

Recall that if we want access to the train error rates as well, we need to set `predict = "both"` in the *outer* resampling strategy.

In `resample` we can use the function `extract` to extract infromation from a fitted model during resampling. Keeping the entire model may be too expensive but this can be set using `models = TRUE`. The function `getTuneResult` allows us to access the optimal set of hyperparameters in each iteration in the `outer` loop. When calling `r$extract` we obtain the optimal hyperparameter values as well as the aggregated performance value from the *inner* iterations (here, we asked for 2 iters). This is in contrast with `r$measures.test`.

```
r$extract
```

```
## [[1]]
## Tune result:
## Op. pars: C=2; sigma=0.25
## mmce.test.mean=0.0147
##
## [[2]]
## Tune result:
## Op. pars: C=4; sigma=0.25
## mmce.test.mean=   0
##
## [[3]]
## Tune result:
## Op. pars: C=2; sigma=0.25
## mmce.test.mean=0.0147
```

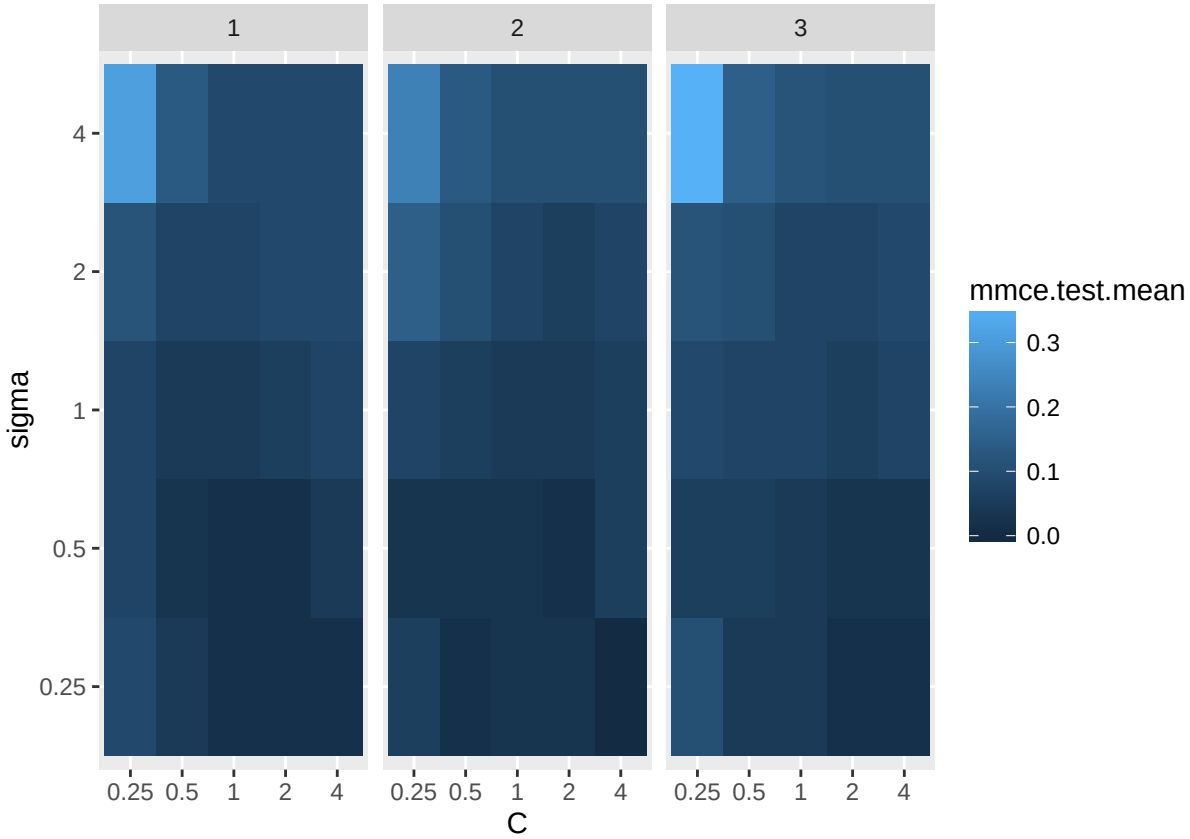
It is possible to compare the performance measures for each pair of settings for `C` and `sigma` tried in each iteration in the outer resampling.

```
opt.paths <- getNestedTuneResultsOptPathDf(r) #returns df
head(opt.paths, 10)
```

```
##       C sigma mmce.test.mean dob eol error.message exec.time iter
## 1  0.25  0.25     0.08823529   1  NA          <NA>     0.034    1
## 2   0.5  0.25     0.04411765   2  NA          <NA>     0.028    1
## 3     1  0.25     0.01470588   3  NA          <NA>     0.033    1
## 4     2  0.25     0.01470588   4  NA          <NA>     0.036    1
## 5     4  0.25     0.01470588   5  NA          <NA>     0.028    1
## 6  0.25   0.5     0.07352941   6  NA          <NA>     0.031    1
## 7   0.5   0.5     0.02941176   7  NA          <NA>     0.029    1
## 8     1   0.5     0.01470588   8  NA          <NA>     0.028    1
## 9     2   0.5     0.01470588   9  NA          <NA>     0.029    1
## 10    4   0.5     0.04411765  10  NA          <NA>     0.033    1
```

The optimal configuration depends on the data but from this experiment, one can identify a good range of hyperparameters that give good performance. Instead of going through the results in a table one by one, it is easier to visualize the results using a tile plot representing the search grid colored by the `mmce.test.mean`.

```
g <- ggplot(opt.paths, aes(x = C, y = sigma, fill = mmce.test.mean)) + geom_tile() + facet_wrap( ~ iter)
g
```



Lastly, if we want to access just the optimal hyperparameter values for each iteration, we can use the `resample` object with `getNestedTuneResultsX`.

```
getNestedTuneResultsX(r)
```

```
##   C sigma
## 1 2  0.25
## 2 4  0.25
## 3 2  0.25
```

Feature selection Nested resampling with feature selection wrapper works very similary to the tuning example in the previous subsection. We create a learner with the feature selection wrapper and pass an inner resampling strategy. For each iteration in the resampling strategy made up of training/test sets, feature selection according to the strategy chosen is carried out. Then, each training in the outer resampling is trained with its selected features and the performance is evaluated on the correspoding test set.

```
inner <- makeResampleDesc("CV", iters = 3)
lrn <- makeFeatSelWrapper("regr.lm", resampling = inner, control = makeFeatSelControlSequential(method =

outer <- makeResampleDesc("Subsample", iters = 2)
r <- resample(lrn, bh.task, resampling = outer, extract = getFeatSelResult, show.info = FALSE)
r
```

```
## Resample Result
## Task: BostonHousing-example
## Learner: regr.lm.featsel
## mse.aggr: 24.23
## mse.mean: 24.23
## mse.sd: 9.40
## Runtime: 37.0796
```

```
r$measures.test
```

```
##   iter      mse
## 1    1 30.88093
## 2    2 17.58244
```

Again, we can access the information we asked to be extracted using `r$extract`. We then access the features selected in the two iterations in the outer resampling.

```
r$extract
```

```
## [[1]]
## FeatSel result:
## Features (8): crim, zn, chas, nox, rm, dis, ptratio, lstat
## mse.test.mean=21.4
##
## [[2]]
## FeatSel result:
## Features (8): nox, rm, dis, rad, tax, ptratio, b, lstat
## mse.test.mean=28.9
```

```
r$extract[[1]]$x
```

```
## [1] "crim"    "zn"      "chas"    "nox"     "rm"      "dis"     "ptratio"
## [8] "lstat"
```

```
r$extract[[1]]$y
```

```
## mse.test.mean
##      21.38866
```

The optimization path for each iteration in the outer loop can be read using the function `analyzeFeatSelResult` which can be applied to `r$extract` in a loop.

```
opt.paths <- lapply(r$extract, function(x) analyzeFeatSelResult(x))
```

```
## Features         : 8
## Performance      : mse.test.mean=21.4
## crim, zn, chas, nox, rm, dis, ptratio, lstat
##
## Path to optimum:
## - Features:    0  Init   :                       Perf = 86.248  Diff: NA  *
## - Features:    1  Add    : lstat                 Perf = 38.645  Diff: 47.603  *
## - Features:    2  Add    : rm                    Perf = 28.621  Diff: 10.024  *
## - Features:    3  Add    : ptratio               Perf = 25.204  Diff: 3.4167  *
## - Features:    4  Add    : chas                  Perf = 24.014  Diff: 1.1898  *
## - Features:    5  Add    : dis                   Perf = 23.653  Diff: 0.36175  *
## - Features:    6  Add    : nox                   Perf = 22.045  Diff: 1.6073  *
## - Features:    7  Add    : zn                    Perf = 21.726  Diff: 0.31956  *
## - Features:    8  Add    : crim                  Perf = 21.389  Diff: 0.33704  *
##
## Stopped, because no improving feature was found.
## Features         : 8
## Performance      : mse.test.mean=28.9
## nox, rm, dis, rad, tax, ptratio, b, lstat
##
## Path to optimum:
## - Features:    0  Init   :                       Perf = 92.688  Diff: NA  *
## - Features:    1  Add    : lstat                 Perf = 43.42  Diff: 49.268  *
## - Features:    2  Add    : rm                    Perf = 36.497  Diff: 6.9235  *
## - Features:    3  Add    : ptratio               Perf = 33.083  Diff: 3.4137  *
## - Features:    4  Add    : dis                   Perf = 31.765  Diff: 1.318  *
## - Features:    5  Add    : nox                   Perf = 29.979  Diff: 1.7865  *
## - Features:    6  Add    : b                     Perf = 29.438  Diff: 0.54038  *
## - Features:    7  Add    : rad                   Perf = 29.206  Diff: 0.23183  *
## - Features:    8  Add    : tax                   Perf = 28.941  Diff: 0.265  *
##
## Stopped, because no improving feature was found.
```

```
opt.paths
```

```
## [[1]]
## NULL
##
## [[2]]
## NULL
```

Filter methods with tuning As seen previously, filter methods assign an importance value to each feature; upon setting a threshold value or a fixed number/percentage ranking the most important features, a subset of variables may be selected for training a learner. The threshold or fixed number/percentage of features to keep usually needs to be tuned. To do this, we need to make a `Filter` wrapper where we choose a filtering method - we tune the parameter associated with the subset selection in a `Tune` wrapper. So we wrap a base learner (here, we fuse linear regression on the `bh.task`) twice.

```
lrn <- makeFilterWrapper(learner = "regr.lm", fw.method = "chi.squared")
ps <- makeParamSet(
  makeDiscreteParam("fw.threshold", values = seq(0, 1, 0.2))
)
ctrl <- makeTuneControlGrid()
inner <- makeResampleDesc("CV", iters = 3)
lrn <- makeTuneWrapper(learner  = lrn, resampling = inner, par.set = ps,
                       control = ctrl, show.info = FALSE)
```

The subset selection parameter is tuned in the inner resampling loop using data from each iteration in the outer resampling. The chosen parameter is used to train a learner using each of the training set in the outer resampling. The outer resampling strategy here is chosen as 3-fold cross-validation.

```
outer <- makeResampleDesc("CV", iters = 3)
r <- resample(learner = lrn, task = bh.task, resampling = outer, models = TRUE, show.info = FALSE)
r
```

```
## Resample Result
## Task: BostonHousing-example
## Learner: regr.lm.filtered.tuned
## mse.aggr: 25.58
## mse.mean: 25.58
## mse.sd: 2.50
## Runtime: 6.54793
```

The regression task `bh.task` has 506 observations and 13 features. In the example above we kept the models in `resample` and so we can access information about them contained in the object returned by `resample`.

```
r$models
```

```
## [[1]]
## Model for learner.id=regr.lm.filtered.tuned; learner.class=TuneWrapper
## Trained on: task.id = BostonHousing-example; obs = 337; features = 13
## Hyperparameters: fw.method=chi.squared
##
## [[2]]
## Model for learner.id=regr.lm.filtered.tuned; learner.class=TuneWrapper
## Trained on: task.id = BostonHousing-example; obs = 337; features = 13
## Hyperparameters: fw.method=chi.squared
```

```
##
## [[3]]
## Model for learner.id=regr.lm.filtered.tuned; learner.class=TuneWrapper
## Trained on: task.id = BostonHousing-example; obs = 338; features = 13
## Hyperparameters: fw.method=chi.squared
```

Note that `r$models` gives us information on the number of observations used to train the learner and what hyperparameters were explored. The number of features indicated is the number included in the task, it does not represent the size of the subset of features selected. Feature selection information can be extracted using the function `getFilteredFeatures`.

```
lapply(r$models, getFilteredFeatures) #all features?
```

```
## [[1]]
##  [1] "crim"    "zn"      "indus"   "chas"    "nox"     "rm"      "age"
##  [8] "dis"     "rad"     "tax"     "ptratio" "b"       "lstat"
##
## [[2]]
##  [1] "crim"    "zn"      "indus"   "chas"    "nox"     "rm"      "age"
##  [8] "dis"     "rad"     "tax"     "ptratio" "b"       "lstat"
##
## [[3]]
##  [1] "crim"    "zn"      "indus"   "chas"    "nox"     "rm"      "age"
##  [8] "dis"     "rad"     "tax"     "ptratio" "b"       "lstat"
```

```
lapply(r$models, function(x) getFilteredFeatures(x$learner.model$next.model))
```

```
## [[1]]
##  [1] "crim"    "zn"      "indus"   "nox"     "rm"      "age"     "dis"
##  [8] "rad"     "tax"     "ptratio" "b"       "lstat"
##
## [[2]]
##  [1] "crim"    "zn"      "indus"   "chas"    "nox"     "rm"      "age"
##  [8] "dis"     "rad"     "tax"     "ptratio" "b"       "lstat"
##
## [[3]]
##  [1] "crim"    "zn"      "indus"   "nox"     "rm"      "age"     "dis"
##  [8] "rad"     "tax"     "ptratio" "b"       "lstat"
```

[Ok, so calling `getFilteredFeatures` on `r$models` returns all available features, I suppose that has to do something with the double wrapper but couldn't find info on `next.model` anywhere. When we just used the `FilterWrapper` learner in the previous section, `getFilteredFeatures(r$models)` returned the *selected* features. Since we often do not want to keep the models since it can get expensive, we can use `extract = function(x) getFilteredFeatures(x$learner.model$next.model)` in `resample` to get the filtered features in this case].

We can access the tune results from the full model using the function `getTuneResult`. Of course note that if the models are not kept, then we can still access information on the tuning and, from that, information on the optimization path by setting `extract = getTuneResult` in `resample`.

```
res <- lapply(r$models, getTuneResult)
res
```

```
## [[1]]
## Tune result:
## Op. pars: fw.threshold=0.4
## mse.test.mean=25.1
##
## [[2]]
## Tune result:
## Op. pars: fw.threshold=0
## mse.test.mean=23.2
##
## [[3]]
## Tune result:
## Op. pars: fw.threshold=0.4
## mse.test.mean=24.7
```

From `res`, we can then obtain information on the optimization paths.

```
opt.paths <- lapply(res, function(x) as.data.frame(x$opt.path))
opt.paths
```

```
## [[1]]
##   fw.threshold mse.test.mean dob eol error.message exec.time
## 1            0      25.15489   1  NA          <NA>     0.146
## 2          0.2      25.14266   2  NA          <NA>     0.164
## 3          0.4      25.07685   3  NA          <NA>     0.168
## 4          0.6      36.15393   4  NA          <NA>     0.137
## 5          0.8      88.66949   5  NA          <NA>     0.129
## 6            1      88.66949   6  NA          <NA>     0.129
##
## [[2]]
##   fw.threshold mse.test.mean dob eol error.message exec.time
## 1            0      23.20063   1  NA          <NA>     0.155
## 2          0.2      23.92397   2  NA          <NA>     0.154
## 3          0.4      23.42240   3  NA          <NA>     0.150
## 4          0.6      28.11603   4  NA          <NA>     0.145
## 5          0.8      87.40533   5  NA          <NA>     0.146
## 6            1      87.40533   6  NA          <NA>     0.146
##
## [[3]]
##   fw.threshold mse.test.mean dob eol error.message exec.time
## 1            0      24.72981   1  NA          <NA>     0.160
## 2          0.2      24.86387   2  NA          <NA>     0.159
## 3          0.4      24.70259   3  NA          <NA>     0.161
## 4          0.6      28.57466   4  NA          <NA>     0.146
## 5          0.8      79.65968   5  NA          <NA>     0.145
## 6            1      79.65968   6  NA          <NA>     0.129
```

Benchmark experiments We now extend the techniques used in nested resampling above to benchmark experiments where we compare different learners on one or more tasks. Tuning or feature selection occurs using different inner resampling descriptions and then benchmark is called with outer resampling for all tasks.

Example 1: Two tasks, two learners, tuning

In this first example, we use the `iris` and `sonar` task data with two learners (`ksvm` and `kknn`). We perform tuning on both of them.

```
tasks <- list(iris.task, sonar.task)

# svm tune wrapper in inner resampling loop
ps <- makeParamSet(
  makeDiscreteParam("C", 2^(-1:1)),
  makeDiscreteParam("sigma", 2^(-1:1))
)
ctrl <- makeTuneControlGrid()
inner <- makeResampleDesc("Holdout")
lrn1 <- makeTuneWrapper("classif.ksvm", resampling = inner, par.set = ps, control = ctrl, show.info = FA

# knn tune wrapper in inner resampling loop
ps <- makeParamSet(makeDiscreteParam("k", 3:5))
ctrl <- makeTuneControlGrid()
inner <- makeResampleDesc("Subsample", iters = 3)
lrn2 <- makeTuneWrapper("classif.knn", resampling = inner, control = ctrl, par.set = ps, show.info = FAL

lrns <- list(lrn1, lrn2)

# outer resampling loop
outer <- list(makeResampleDesc("Holdout"), makeResampleDesc("Bootstrap", iters = 2))
res <- benchmark(lrns, tasks, outer, measures = list(acc, ber), show.info = FALSE)
# aggregated performances from the outer resampling loop
res
```

```
##         task.id         learner.id acc.test.mean ber.test.mean
## 1  iris-example classif.ksvm.tuned     1.0000000    0.00000000
## 2  iris-example  classif.knn.tuned     0.9800000    0.02083333
## 3 Sonar-example classif.ksvm.tuned     0.5650000    0.48026316
## 4 Sonar-example  classif.knn.tuned     0.7535714    0.26017231
```

The `resamplings` argument in `benchmark` gives resampling strategies for *each* task. If only one is applied, it is replicated to match the number of tasks, if none are applied, the default 10-fold corss-validation is used. So, in the above example, the outer resampling for the `iris` task is set to *holdout* and for the `sonar` task is set to *bootstrap* with 2 iterations.

We can access the benchmark result using the `get` functions available in the mlr package. We can access the outer resampling individual performances using `getBMRPerformances`.

```
getBMRPerformances(res, as.df = TRUE)
```

```
##         task.id         learner.id iter       acc        ber
## 1  iris-example classif.ksvm.tuned    1 1.0000000 0.00000000
## 2  iris-example  classif.knn.tuned    1 0.9800000 0.02083333
## 3 Sonar-example classif.ksvm.tuned    1 0.5833333 0.46052632
## 4 Sonar-example classif.ksvm.tuned    2 0.5466667 0.50000000
## 5 Sonar-example  classif.knn.tuned    1 0.7738095 0.23627002
## 6 Sonar-example  classif.knn.tuned    2 0.7333333 0.28407461
```

This gives the result of each iteration specified in the outer resampling strategy for each task, trained on each learner. Since for the `iris` task we use *holdout*, the performance is tested on a single test set for each learner. With the `sonar` task, we have *bootstrap* with 2 iterations, we get 2 test sets for each learner.

To obtain the results of the tuning, we use `getBMRTuneResults` on the object returned by `benchmark`.

```
getBMRTuneResults(res, as.df = TRUE)
```

```
##         task.id         learner.id iter  C sigma mmce.test.mean  k
## 1  iris-example classif.ksvm.tuned    1  1   0.5     0.08823529 NA
## 2  iris-example  classif.knn.tuned    1 NA    NA     0.04901961  5
## 3 Sonar-example classif.ksvm.tuned    1  2   0.5     0.20000000 NA
## 4 Sonar-example classif.ksvm.tuned    2  1   2.0     0.25714286 NA
## 5 Sonar-example  classif.knn.tuned    1 NA    NA     0.18571429  3
## 6 Sonar-example  classif.knn.tuned    2 NA    NA     0.22857143  3
```

The `task.ids` and `learner.ids` arguments can be used in `getBMRTuneResults` to access the tuning results from individual learners and tasks. We can access the tuned hyperparameter settings from a nested tuning using `getNestedTuneResultsX`; this takes as an argument a `ResampleResult` so in the case of the `benchmark` object, we need to input the element from the list of results contained in the benchmark object `res$results`.

```
# tuned hyperparameters
getNestedTuneResultsX(res$results[["Sonar-example"]][["classif.ksvm.tuned"]])
```

```
##   C sigma
## 1 2   0.5
## 2 1   2.0
```

```
# get the opt.paths from each tuning step from the outer resampling
getNestedTuneResultsOptPathDf(res$results[["Sonar-example"]][["classif.ksvm.tuned"]])
```

```
##      C sigma mmce.test.mean dob eol error.message exec.time iter
## 1  0.5   0.5      0.2428571   1  NA          <NA>     0.031    1
## 2    1   0.5      0.2142857   2  NA          <NA>     0.032    1
## 3    2   0.5      0.2000000   3  NA          <NA>     0.034    1
## 4  0.5     1      0.2428571   4  NA          <NA>     0.028    1
## 5    1     1      0.2428571   5  NA          <NA>     0.029    1
## 6    2     1      0.2428571   6  NA          <NA>     0.035    1
## 7  0.5     2      0.2428571   7  NA          <NA>     0.032    1
## 8    1     2      0.2428571   8  NA          <NA>     0.032    1
## 9    2     2      0.2428571   9  NA          <NA>     0.030    1
## 10 0.5   0.5      0.4428571   1  NA          <NA>     0.031    2
## 11   1   0.5      0.2571429   2  NA          <NA>     0.030    2
## 12   2   0.5      0.2571429   3  NA          <NA>     0.027    2
## 13 0.5     1      0.4428571   4  NA          <NA>     0.033    2
## 14   1     1      0.2571429   5  NA          <NA>     0.028    2
## 15   2     1      0.2571429   6  NA          <NA>     0.032    2
## 16 0.5     2      0.4428571   7  NA          <NA>     0.036    2
## 17   1     2      0.2571429   8  NA          <NA>     0.027    2
## 18   2     2      0.2571429   9  NA          <NA>     0.032    2
```

Example 2: One task, two learners, feature selection

In this example, we wrap a linear regression learner with feature selection wrapper achieved by a sequential forward search. The features are selected using a two-iteration, subsample strategy (inner resampling). The second learner is a base `rpart` regression algorithm (no feature selection).

```r
ctrl <- makeFeatSelControlSequential(method = "sfs")
inner <- makeResampleDesc("Subsample", iters = 2)
lrn <- makeFeatSelWrapper("regr.lm", resampling = inner, control = ctrl, show.info = FALSE)

lrns <- list(makeLearner("regr.rpart"), lrn)

outer <- makeResampleDesc("Subsample", iters = 2)
res <- benchmark(learners = lrns, tasks = bh.task, resamplings = outer, show.info = FALSE)
res
```

```
##                 task.id      learner.id mse.test.mean
## 1 BostonHousing-example      regr.rpart      19.91970
## 2 BostonHousing-example regr.lm.featsel      30.56572
```

The selected features can be accessed using `getBMRFeatSelResults`:

```r
getBMRFeatSelResults(res)
```

```
## $`BostonHousing-example`
## $`BostonHousing-example`$regr.rpart
## NULL
##
## $`BostonHousing-example`$regr.lm.featsel
## $`BostonHousing-example`$regr.lm.featsel[[1]]
## FeatSel result:
## Features (9): crim, nox, rm, dis, rad, tax, ptratio, b, lstat
## mse.test.mean=22.2
##
## $`BostonHousing-example`$regr.lm.featsel[[2]]
## FeatSel result:
## Features (4): crim, rm, ptratio, lstat
## mse.test.mean=19.5
```

Again, we can access specific learners:

```r
feats <- getBMRFeatSelResults(res, learner.ids = "regr.lm.featsel")
feats <- feats[["BostonHousing-example"]][["regr.lm.featsel"]]
```

The list `feats` now has information on the outer resampling iterations where for each element in the list, `$x` and `$y` access the features and resampled performance of the test set.

```r
feats[[1]]$x
```

```
## [1] "crim"    "nox"     "rm"      "dis"     "rad"     "tax"     "ptratio"
## [8] "b"       "lstat"
```

```r
feats[[1]]$y
```

```
## mse.test.mean
##      22.23291
```

The optimization paths indicating which features were selected during the feature selection search along with the corresponding test performances can be extracted using `$opt.path`.

```
opt.paths <- lapply(feats, function(x) as.data.frame(x$opt.path))
head(opt.paths[[1]])
```

```
##   crim zn indus chas nox rm age dis rad tax ptratio b lstat mse.test.mean
## 1    0  0     0    0   0  0   0   0   0   0       0 0     0      86.40099
## 2    1  0     0    0   0  0   0   0   0   0       0 0     0      71.58366
## 3    0  1     0    0   0  0   0   0   0   0       0 0     0      74.62474
## 4    0  0     1    0   0  0   0   0   0   0       0 0     0      66.05012
## 5    0  0     0    1   0  0   0   0   0   0       0 0     0      85.55271
## 6    0  0     0    0   1  0   0   0   0   0       0 0     0      73.41056
##   dob eol error.message exec.time
## 1   1   2          <NA>     0.011
## 2   2   2          <NA>     0.015
## 3   2   2          <NA>     0.018
## 4   2   2          <NA>     0.018
## 5   2   2          <NA>     0.017
## 6   2   2          <NA>     0.015
```

The function `analyzeFeatSelResult` gives a clearer overview of what feature has been added in sequence with the improvement in performance.

```
analyzeFeatSelResult(feats[[1]])
```

```
## Features         : 9
## Performance      : mse.test.mean=22.2
## crim, nox, rm, dis, rad, tax, ptratio, b, lstat
##
## Path to optimum:
## - Features:    0  Init   :                       Perf = 86.401  Diff: NA  *
## - Features:    1  Add    : lstat                 Perf = 40.011  Diff: 46.39  *
## - Features:    2  Add    : ptratio               Perf = 31.236  Diff: 8.7748  *
## - Features:    3  Add    : rm                    Perf = 25.631  Diff: 5.6047  *
## - Features:    4  Add    : b                     Perf = 24.864  Diff: 0.7676  *
## - Features:    5  Add    : dis                   Perf = 24.429  Diff: 0.43506  *
## - Features:    6  Add    : nox                   Perf = 22.707  Diff: 1.7219  *
## - Features:    7  Add    : crim                  Perf = 22.471  Diff: 0.23618  *
## - Features:    8  Add    : rad                   Perf = 22.377  Diff: 0.094093  *
## - Features:    9  Add    : tax                   Perf = 22.233  Diff: 0.14379  *
##
## Stopped, because no improving feature was found.
```

Example 3: One task, two learners, feature filtering with tuning

This is an example where we use feature filtering and tune a parameter in the filtering method.

```
# feature filtering with tuning in inner loop
lrn <- makeFilterWrapper(learner = "regr.lm", fw.method = "chi.squared")
ps <- makeParamSet(makeDiscreteParam("fw.abs", values = seq_len(getTaskNFeats(bh.task))) )
ctrl <- makeTuneControlGrid()
inner <- makeResampleDesc("CV", iters = 2)
```

```
lrn <- makeTuneWrapper(lrn, resampling = inner, par.set = ps, control = ctrl, show.info = FALSE)

# outer loop
lrns <- list(makeLearner("regr.rpart"), lrn)
outer <- makeResampleDesc("Subsample", iters = 3)
res <- benchmark(lrns, tasks = bh.task, resamplings = outer, show.info = FALSE)
res
```

```
##                 task.id             learner.id mse.test.mean
## 1 BostonHousing-example             regr.rpart      24.68150
## 2 BostonHousing-example regr.lm.filtered.tuned      25.22963
```

Cost-sensitive classification

Classifiers are biased to predict well the *majority* class. In regular classification problems, all misclassification errors are taken to be equal. If we take the example of a medical test used to identify patients with tumor (positive class) and patients without tumor (negative class), there should be a higher cost associated with a *false negative* (FN) than a *false positive* (FP). In the former case, we miss the tumor and the patient potentially dies as a result of not undergoing treatment while in the latter case, the patient is, in addition of giving him/her quite a scare, sent to carry out additional tests. The FN error therefore should be associated a higher cost than the FP error.

The confusion matrix for a binary classification problem with classes `+` and `-` is given below:

Actual/Predicted	+	-
+	TP	FN
-	FP	TN

where the usual metrics are:

where the usual metrics are:

sensitivity/recall = $\frac{TP}{TP+FN}$, specificity = $\frac{TN}{FP+TN}$, and precision = $\frac{TP}{TP+FP}$.

When calculating *classification costs*, we associate a cost with misclassification, i.e. $C(i|j)$ is the cost of misclassifying class j as class i. As indicated above, in the medical example, a FN is associated with higher cost than a FP:

$C(-|+) > C(+|-)$.

The associated cost matrix therefore is given as:

Actual/Predicted	+	-
+	$C(+\|+)$	$C(-\|+)$
-	$C(+\|-)$	$C(-\|-)$

The cost matrix can then be used for cost-sensitive evaluation of classifiers and cost-sensitive classification where the objective is to minimize expected costs. The cost associated with predicting the correct class is of course lowest; usually the diagonal is calculated to be zero or the matrix is rescaled such as the diagonal is made up of zeros. Some algorithms can utilize a cost matrix directly; an example is `rpart`. Alternatively, we can use the cost matrix to calculate a threshold when predicting class labels or carry out rebalancing wherein less costly classes are given higher importance in training. The rebalancing is achieved either through *weights*

(if the learner supports them) or through *oversampling/undersampling* We start with examples in **binary classification** and then we move to **multi-class**.

Binary classification examples In mlr, we use the ordinary `ClassifTask` for classification as in previous parts of this tutorial. The `CostSensTask` is used when the costs are *example-dependent*; in our case the costs are *class-dependent*. In the example below, we use the `GermanCredit` data to create a classification task and then remove constant features using the appropriate function.

```
data(GermanCredit, package = "caret")
credit.task <- makeClassifTask(data = GermanCredit, target = "Class")
credit.task <- removeConstantFeatures(credit.task)
```

```
## Removing 2 columns: Purpose.Vacation,Personal.Female.Single
```

```
credit.task
```

```
## Supervised task: GermanCredit
## Type: classif
## Target: Class
## Observations: 1000
## Features:
## numerics  factors  ordered
##       59        0        0
## Missings: FALSE
## Has weights: FALSE
## Has blocking: FALSE
## Classes: 2
##  Bad Good
##  300  700
## Positive class: Bad
```

By default, bad is the positive class, as indicated when printing `credit.task`. We then create the cost matrix (note that this needs to be calculated based on application).

```
costs <- matrix(c(0,1,5,0), 2)
colnames(costs) <- rownames(costs) <- getTaskClassLevels(credit.task)
```

We first create a logistic regression learner to fit the data and predict posterior probabilities. The classes are then predicted from those probabilities with a default threshold of 0.5. Note that `multinom` fits multinomial log-linear models via neural networks.

```
lrn <- makeLearner("classif.multinom", predict.type = "prob", trace = FALSE)
mod <- mlr::train(lrn, credit.task)
pred <- predict(mod, task = credit.task)
pred
```

```
## Prediction: 1000 observations
## predict.type: prob
## threshold: Bad=0.50,Good=0.50
## time: 0.01
##   id truth   prob.Bad prob.Good response
```

```
## 1  1  Good 0.03525092 0.9647491     Good
## 2  2   Bad 0.63222363 0.3677764      Bad
## 3  3  Good 0.02807414 0.9719259     Good
## 4  4  Good 0.25182703 0.7481730     Good
## 5  5   Bad 0.75193275 0.2480673      Bad
## 6  6  Good 0.26230149 0.7376985     Good
```

According to the cost matrix, we should predict class `Good` only if we are very sure that the class label is in fact `Good`. We need to increase the threshold for class `Good` and decrease the threshold for class `Bad`.

Theoretical thresholding

The cost matrix can be used to calculate a theoretical thresholding for the *positive class* Elkan (2001), as follows:

$t^* = \frac{C(+|-)-C(-|-)}{C(+|-)-C(+|+)+C(-|+)-C(-|-)}$

With the diagonal being zeros, the formula simplifies quite a bit.

```
th <- costs[2,1]/(costs[2,1]+costs[1,2])
th
```

```
## [1] 0.1666667
```

We can change the threshold before training in `makeLearner` using the argument `predict.threshold = th` where `th` is the desired threshold or, after predidction by calling `setThreshold` on the object returned by `predict`.

```
pred.th <- setThreshold(pred, th)
pred.th
```

```
## Prediction: 1000 observations
## predict.type: prob
## threshold: Bad=0.17,Good=0.83
## time: 0.01
##   id truth   prob.Bad prob.Good response
## 1  1  Good 0.03525092 0.9647491     Good
## 2  2   Bad 0.63222363 0.3677764      Bad
## 3  3  Good 0.02807414 0.9719259     Good
## 4  4  Good 0.25182703 0.7481730      Bad
## 5  5   Bad 0.75193275 0.2480673      Bad
## 6  6  Good 0.26230149 0.7376985      Bad
```

Next, we wish to calculate the cost associated with our predictions averaged over the entire data set. Recall that the objective is to choose a model which minimize the costs. To calculate the averaged costs, we need to create a new performance measure using `makeCostMeasure`.

```
credit.costs <- makeCostMeasure(id = "credit.costs", name = "Credit costs", costs = costs, task = credi
credit.costs
```

```
## Name: Credit costs
## Performance measure: credit.costs
## Properties: classif,classif.multi,req.pred,req.truth,predtype.response,predtype.prob
## Minimize: TRUE
```

```
## Best: 0; Worst: 5
## Aggregated by: test.mean
## Note:
```

We now calculate the performance based on our new `credit.costs` measire and the misclassification error rate on both prediction objects `pred` and `pred.th`:

```
performance(pred, measures = list(credit.costs, mmce))
```

```
## credit.costs         mmce
##        0.774        0.214
```

```
performance(pred.th, measures = list(credit.costs, mmce))
```

```
## credit.costs         mmce
##        0.478        0.346
```

No resampling was carried out and therefore the same data (all of it) was used for training and prediction leading to potentially overly optimistic performance values. Below, we fit again a logistic regression model using 3-fold cross-validation to obtain less biased predictions. Note that we create a `ResampleInstance` to specify training/test sets so that we get comparable performance values when we try different cost-sensitive methods on the `credit.task` data.

```
rin <- makeResampleInstance("CV", iters = 3, task = credit.task)
lrn <- makeLearner("classif.multinom", predict.type = "prob", predict.threshold = th, trace = FALSE)
r <- resample(lrn, credit.task, rin, measures = list(credit.costs, mmce), show.info = FALSE)
r
```

```
## Resample Result
## Task: GermanCredit
## Learner: classif.multinom
## credit.costs.aggr: 0.56
## credit.costs.mean: 0.56
## credit.costs.sd: 0.05
## mmce.aggr: 0.36
## mmce.mean: 0.36
## mmce.sd: 0.07
## Runtime: 0.167706
```

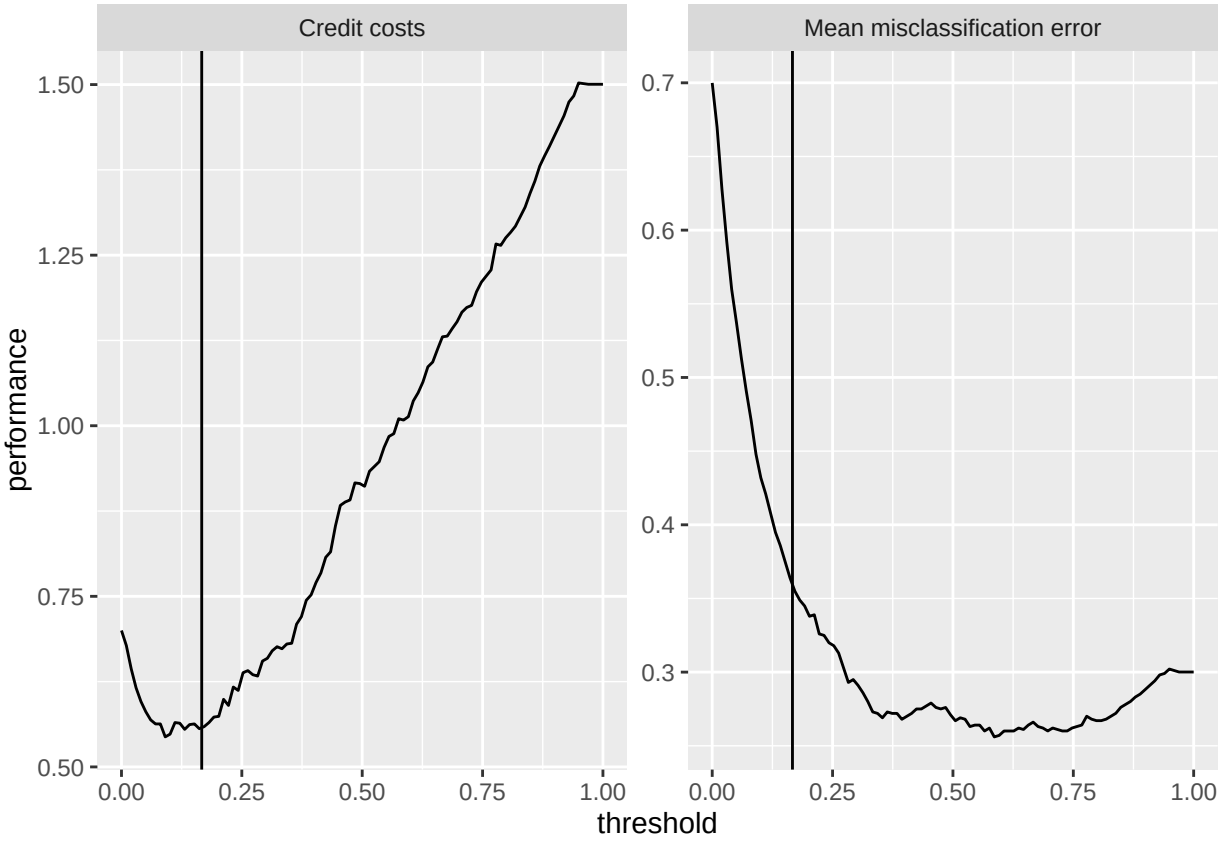
To get the cross-validated performance measures using the default threshold, we use the `performance` function as above:

```
pred.th <- setThreshold(r$pred, 0.5)
performance(pred.th, measures = list(credit.costs, mmce))
```

```
## credit.costs         mmce
##    0.9142586    0.2700215
```

We can visualize the performance measures (costs, mmce, etc) vs threshold in [0,1] for the positive class using `plotThreshVsPerf` using data from `generateThreshVsPerfData`. The latter requires a prediction object (from `predict`, `resample`, `benchmark`).

```
d <- generateThreshVsPerfData(r, measures = list(credit.costs, mmce))
plotThreshVsPerf(d, mark.th = th)
```



From the plots, we observe that the theoretical threshold (indicated by the vertical line) is a little large.

Empirical thresholding

Theoretical thresholding is reliable when the predicted posterior probabilities are correct; empirical thresholding is useful when the probabilities are order-correct. In empirical thresholding, the cost-optimal threshold is selected for a given learner using training data. The function `tuneThreshold` is used to determine the optimal threshold; this is used with a resampling strategy and not on the entire data set, to avoid overfitting. The function `tuneThreshold` returns the optimal threshold and performance for the specified measure (below, we choose to minimize `credit.costs`).

```
lrn <- makeLearner("classif.multinom", predict.type = "prob", trace = FALSE)

r <- resample(lrn, credit.task, rin, measures = list(credit.costs, mmce), show.info = FALSE)
r
```

```
## Resample Result
## Task: GermanCredit
## Learner: classif.multinom
## credit.costs.aggr: 0.91
## credit.costs.mean: 0.91
## credit.costs.sd: 0.23
## mmce.aggr: 0.27
## mmce.mean: 0.27
```

```
## mmce.sd: 0.02
## Runtime: 0.144895
```

```
tune.res <- tuneThreshold(pred = r$pred, measure = credit.costs)
tune.res
```

```
## $th
## [1] 0.09789179
##
## $perf
## credit.costs
##    0.5379871
```

Rebalancing

To minimize average costs, observations associated with the less costly class should be given higher importance during training. One way of achieving this is through class weighting. The learner must be able to support observation or class weights to use this method. Alternative methods, over- and undersampling, are considered later in this section.

Weighting

```
# check which mlr learners support weights
listLearners("classif", properties = "weights")[c("class", "package")] #obs weights
```

```
## Warning in listLearners.character("classif", properties = "weights"): The following learners could n
## classif.ada,classif.bartMachine,classif.bdk,classif.blackboost,classif.boosting,classif.bst,classif.
## Check ?learners to see which packages you need or install mlr with all suggestions.
```

```
##              class package
## 1   classif.avNNet    nnet
## 2 classif.binomial   stats
## 3      classif.gbm     gbm
## 4   classif.logreg   stats
## 5 classif.multinom    nnet
## 6     classif.nnet    nnet
## 7   classif.probit   stats
## 8    classif.rpart   rpart
## 9  classif.xgboost xgboost
```

```
listLearners("classif", properties = "class.weights")[c("class", "package")] #class weights
```

```
## Warning in listLearners.character("classif", properties = "class.weights"): The following learners c
## classif.ada,classif.bartMachine,classif.bdk,classif.blackboost,classif.boosting,classif.bst,classif.
## Check ?learners to see which packages you need or install mlr with all suggestions.
```

```
##                  class      package
## 1         classif.ksvm      kernlab
## 2 classif.randomForest randomForest
## 3          classif.svm        e1071
```

According to Elkan 2001, the proportion of observations in the positive class is multiplied by

$\frac{1-t}{t}\frac{t_0}{1-t_0}$,

where t and t_0 are the target and original thresholds, respectively. If $t_0 = 0.5$, the second factor is equal to 1. Alternatively, the proportion of observations in the negative class can be multiplied by the inverse. The *theoretical weights* can be a function of the theoretical threshold if the target threshold equals the theoretical value, t^*.

```
# theoretical weights for positive class corresponding to theoretical threshold
w <- (1-th)/th
w
```

```
## [1] 5
```

In binary classification, if we choose to set a weight for the positive class then the negative class receives a weight of 1 automatically. The mlr package offers a `makeWeightedClassWrapper` which allows to assign class weights to a learner using the argument `wcw.weight`. If the learner supports observation weights then these are internally generated during training or resampling. For available class weight support, the weights are simply passed on to the appropriate learner parameter.

```
lrn <- makeLearner("classif.multinom", trace = FALSE)
lrn <- makeWeightedClassesWrapper(lrn, wcw.weight = w)
lrn
```

```
## Learner weightedclasses.classif.multinom from package nnet
## Type: classif
## Name: ; Short name:
## Class: WeightedClassesWrapper
## Properties: twoclass,multiclass,numerics,factors,prob
## Predict-Type: response
## Hyperparameters: trace=FALSE,wcw.weight=5
```

We use the resampling instance created above (`rin`) so that we can compare the performance of *rebalancing* against *thresholding*.

```
r <- resample(lrn, credit.task, rin, measures = list(credit.costs, mmce), show.info = FALSE)
r
```

```
## Resample Result
## Task: GermanCredit
## Learner: weightedclasses.classif.multinom
## credit.costs.aggr: 0.55
## credit.costs.mean: 0.55
## credit.costs.sd: 0.08
## mmce.aggr: 0.35
## mmce.mean: 0.35
## mmce.sd: 0.06
## Runtime: 0.185865
```

Now, the `classif.multinom` learner supports observation weights. If we were to choose a learner that supports class weights like, for instance, `classif.ksvm`, we can pass them directly or using the `makeWeightedClassesWrapper`:

```
# directly
lrn <- makeLearner("classif.ksvm", class.weights = c(Bad = w, Good = 1))
# using wrapper
lrn <- makeWeightedClassesWrapper("classif.ksvm", wcw.weight = w)

r <- resample(lrn, credit.task, rin, measures = list(credit.costs, mmce), show.info = FALSE)
r
```

```
## Resample Result
## Task: GermanCredit
## Learner: weightedclasses.classif.ksvm
## credit.costs.aggr: 0.61
## credit.costs.mean: 0.61
## credit.costs.sd: 0.10
## mmce.aggr: 0.33
## mmce.mean: 0.33
## mmce.sd: 0.05
## Runtime: 0.303572
```

Just like with the theoretical threshold, the theoretical weights may not always be suitable. We can tune the `wcw.weight` parameter like any other parameter using `tuneParams`. Using the threshold weight however can help us narrow down the search.

```
lrn <- makeLearner("classif.multinom", trace = FALSE)
lrn <- makeWeightedClassesWrapper(lrn)
ps <- makeParamSet(makeDiscreteParam("wcw.weight", values = seq(4, 12, 0.5) ))

ctrl <- makeTuneControlGrid()

tune.res <- tuneParams(lrn, credit.task, rin, measures = list(credit.costs, mmce), ps, ctrl, show.info =

as.data.frame(tune.res$opt.path)[1:3]
```

```
##    wcw.weight credit.costs.test.mean mmce.test.mean
## 1           4              0.5730611      0.3289247
## 2         4.5              0.5560591      0.3399268
## 3           5              0.5510570      0.3509288
## 4         5.5              0.5570660      0.3609298
## 5           6              0.5570690      0.3729328
## 6         6.5              0.5640641      0.3839318
## 7           7              0.5640521      0.3919279
## 8         7.5              0.5640431      0.3999269
## 9           8              0.5700371      0.4099249
## 10        8.5              0.5730311      0.4169229
## 11          9              0.5650231      0.4169229
## 12        9.5              0.5720211      0.4279249
## 13         10              0.5730161      0.4329239
## 14       10.5              0.5609981      0.4329179
## 15         11              0.5570121      0.4409200
## 16       11.5              0.5540061      0.4419180
## 17         12              0.5509971      0.4429130
```

Over- and undersampling

Not all learners support observations or class weights and so in those cases we *change* the training data by over- or undersampling. With the theoretical weighting, we calculated that the positive class in the German credit data should receive a weight of 5 (or, equivalently, the negative class should receive a weight of 1/5). This can be achieved by the function `oversample` (`undersample`) giving it a `rate` of 5 (1/5).

```
credit.task.over <- oversample(credit.task, rate = 5, cl = "Bad")
lrn <- makeLearner("classif.multinom", trace = FALSE)
mod <- train(lrn, credit.task.over)
```

We trained our learner on the changed data in `credit.task.over` but our predictions and the calculated traiining performance needs to be assessed using the original data in `credit.task`.

```
pred <- predict(mod, task = credit.task)
performance(pred, measures = list(credit.costs, mmce))
```

```
## credit.costs         mmce
##        0.441        0.325
```

Of course the usual strategy for more accurate performance measures is resampled performances but calling `resample` with `credit.task.over` does not work since predictions need to be based on the original data. To overcome this issue, we create a wrapped learner with `makeOversampleWrapper`.

```
lrn <- makeLearner("classif.multinom", trace = FALSE)
lrn <- makeOversampleWrapper(lrn, osw.rate = w, osw.cl = "Bad")
lrn
```

```
## Learner classif.multinom.oversampled from package mlr,nnet
## Type: classif
## Name: ; Short name:
## Class: OversampleWrapper
## Properties: numerics,factors,weights,prob,twoclass,multiclass
## Predict-Type: response
## Hyperparameters: trace=FALSE,osw.rate=5,osw.cl=<character>
```

```
r <- resample(lrn, credit.task, rin, measures = list(credit.costs, mmce), show.info = FALSE)
r
```

```
## Resample Result
## Task: GermanCredit
## Learner: classif.multinom.oversampled
## credit.costs.aggr: 0.58
## credit.costs.mean: 0.58
## credit.costs.sd: 0.05
## mmce.aggr: 0.36
## mmce.mean: 0.36
## mmce.sd: 0.07
## Runtime: 0.300809
```

We might want to tune the `rate` parameter to minimize the costs. To do so we create a wrapped learner as above and tune `osw.rate` using `tuneParams`.

```
lrn <- makeLearner("classif.multinom", trace = FALSE)
lrn <- makeOversampleWrapper(lrn, osw.cl = "Bad")
ps <- makeParamSet(makeDiscreteParam("osw.rate", seq(3, 7 , 0.25)))
ctrl <- makeTuneControlGrid()
tune.res <- tuneParams(lrn, credit.task, rin, measures = list(credit.costs, mmce), control = ctrl, show
tune.res
```

```
## Tune result:
## Op. pars: osw.rate=5.75
## credit.costs.test.mean=0.543,mmce.test.mean=0.355
```

Multi-class examples It is possible to calculate thresholds and rebalancing weights in a manner similar to that of the binary case if the cost matrix has a special strcuture where $C(k|l) = C(l)$ for $k = 1, ..., K$ and $k \neq l$. In $C(k|l)$, k is the *predicted class* and l is the *true label*. The above structure implies that the cost of misclassifying an observation is independent of the predicted class label. For our multi-class examples below, we use threshold and rebalancing using the `waveform` data and an associated cost matrix that does *not* conform to the special structure.

Theoretical thresholding

```
df <- mlbench::mlbench.waveform(500)
wf.task <- makeClassifTask(id = "waveform", data = as.data.frame(df), target = "classes")

costs <- matrix(c(0, 5, 10, 30, 0 ,8, 80, 4, 0), 3)
colnames(costs) <- rownames(costs) <- getTaskClassLevels(wf.task)

wf.costs <- makeCostMeasure(id = "wf.costs", name = "waveform costs", costs = costs, task = wf.task, be
```

Here, we calculate a vector of threshold values as long as the number of classes K. The threshold vector is calculated as:

$$\text{threshold} = \frac{1}{\text{(average costs of true classes)}}$$

The threshold value is chosen to have the artificial structure mentioned previously, i.e. $C(k, l) = C(l)$ for any $k \neq l$ and we choose $C(l)$ to be the average of the true classes. For example, for class 1, the average would be $110/2 = 55$ and hence the threshold value is $1/55$.

Once we have the threshold vector, we divide the predicted probabilities by the threshold vector to obtain adjusted probabilities. The class with the highest adjusted probability is predicted. We set the threshold using `setThreshold`, as before.

```
lrn <- makeLearner("classif.rpart", predict.type = "prob")
rin <- makeResampleInstance("CV", iters = 3, task = wf.task)
r <- resample(lrn, wf.task, rin, measures = list(wf.costs, mmce), show.info = FALSE)
r
```

```
## Resample Result
## Task: waveform
## Learner: classif.rpart
## wf.costs.aggr: 6.51
## wf.costs.mean: 6.51
## wf.costs.sd: 2.61
## mmce.aggr: 0.27
## mmce.mean: 0.27
```

```
## mmce.sd: 0.04
## Runtime: 0.045521
```

```
avg <- rowSums(costs)/2
th <- 1/avg
# important: the threshold vector needs to have names which correspond to the class labels
names(th) = getTaskClassLevels(wf.task)
th
```

```
##          1          2          3
## 0.01818182 0.22222222 0.11111111
```

```
pred.th <- setThreshold(r$pred, threshold = th)
performance(pred.th, measures = list(wf.costs, mmce))
```

```
##  wf.costs      mmce
## 4.7038333 0.3161749
```

It seems that the data on class probabilities found in `pred.th$data` and `r$pred$data` show the original predicted probabilities but it appears that in `pred.th` the class label is predicted using the adjusted probabilities so all is good.

Empirical thresholding

Once again, we can tune the threshold using `tuneThreshold`. The function returns values that lie in [0,1] and sum up to 1 (the scaling does not change the predicted class labels).

```
tune.res <- tuneThreshold(pred = r$pred, measure = wf.costs)
tune.res
```

```
## $th
##          1          2          3
## 0.08036383 0.22789042 0.69174576
##
## $perf
## [1] 3.736707
```

```
# compare with the standaridized theoretical vector
th/sum(th)
```

```
##          1          2          3
## 0.05172414 0.63218391 0.31609195
```

Rebalancing: weighting

With weights, the function `makeWeightedClassesWrapper` is used just like in the binary case but for multi-class, we also need to specify the length of the vector corresponding to the number of classes. The weight vector can be tuned using `tuneParams`.

```
lrn <- makeLearner("classif.multinom", trace = FALSE)
lrn <- makeWeightedClassesWrapper(lrn)

ps <- makeParamSet(makeNumericVectorParam("wcw.weight", len = 3, lower = 0, upper = 1))
ctrl <- makeTuneControlRandom()

tune.res <- tuneParams(lrn, wf.task, resampling = rin, par.set = ps, measures = list(wf.costs, mmce), c
tune.res
```

```
## Tune result:
## Op. pars: wcw.weight=0.409,0.0593,0.0104
## wf.costs.test.mean= 2.3,mmce.test.mean=0.162
```

Note that `oversample` and `undersample` are for *binary* classification only.

Imbalanced classification problems

In imbalanced classification problems, the smaller classes tend to be ignored as noise in classifiers thus failing to predict them when given new data. The mlr package offers various correction methods to tackle this problem. The methods are divided into sampling and cost-based approaches.

Sampling-based approaches The idea is to adjust the proportion of the classes to increase the weight of the minority class observations within the model.

1. **Undersampling** : randomly chosen cases of the majority class are eliminated while all of the minority class cases are kept.
2. **Oversampling** : additional cases (copies, artifical observations) of the minority class are generated to increase their effect on the classifier while all majority class cases are kept.
3. **Hybrid** : mixture of over- and undersampling strategies.

The above methods directly access the data which means that the sampling is done as part of the *preprocessing* and may be used with every appropriate classifier. Note that the mlr package supports the first 2 approaches currently.

Simple over- and undersampling

In *undersampling*, the majority class is effected while in *oversampling*, copies of the minority class are generated. In oversampling, the minority cases are considered at least once when fitting the model while exact copies are generated by random sampling with repetitions.

We first look at some simulated data for a binary problem with classes `A` and `B`.

```
data.imbal.train <- rbind(
  data.frame(x = rnorm(100, mean = 1), class = "A"),
  data.frame(x = rnorm(5000, mean = 2 ), class = "B")
)

task <- makeClassifTask(data = data.imbal.train, target = "class")
task.over <- oversample(task, rate = 8)
task.under <- undersample(task, rate = 1/8)

table(getTaskTargets(task))
```

```
##
##    A    B
##  100 5000
```

```
table(getTaskTargets(task.under))
```

```
##
##   A   B
## 100 625
```

```
table(getTaskTargets(task.over))
```

```
##
##    A    B
##  800 5000
```

In `undersample` the rate is in [0,1]; here, `rate = 1/8` implies that the number of majority class observations are reduced to 1/8th of their original size. In `oversample`, rate ≥ 1; here, `rate = 8` means the minority class observations have been increased to 8 times of their original size. We compare the performance on each task.

```
lrn <- makeLearner("classif.rpart", predict.type = "prob")
mod <- train(lrn, task)
mod.over <- train(lrn, task.over)
mod.under <- train(lrn, task.under)
data.imbal.test <- rbind(
  data.frame(x = rnorm(10, mean = 1), class = "A"),
  data.frame(x = rnorm(500, mean = 2 ), class = "B")
)

performance(predict(mod, newdata = data.imbal.test), measures = list(mmce, ber, auc))
```

```
##       mmce        ber        auc
## 0.01960784 0.50000000 0.50000000
```

```
performance(predict(mod.over, newdata = data.imbal.test), measures = list(mmce, ber, auc))
```

```
##       mmce        ber        auc
## 0.03333333 0.36000000 0.70910000
```

```
performance(predict(mod.under, newdata = data.imbal.test), measures = list(mmce, ber, auc))
```

```
##       mmce        ber        auc
## 0.05882353 0.42200000 0.65970000
```

Note that since `mmce` evaluates the overall accuracy of the predictions, the balanced error rate (`ber`) which gives the average of the errors in each class and the area under the ROC curver (`auc`) may be more suitable performance measures.

```
getConfMatrix(predict(mod, newdata = data.imbal.test))
```

```
##        predicted
## true    A   B -SUM-
##   A     0  10    10
##   B     0 500     0
##   -SUM- 0  10    10
```

```
getConfMatrix(predict(mod.under, newdata = data.imbal.test))
```

```
##        predicted
## true     A   B -SUM-
##   A      2   8     8
##   B     22 478    22
##   -SUM- 22   8    30
```

```
getConfMatrix(predict(mod.over, newdata = data.imbal.test))
```

```
##        predicted
## true     A   B -SUM-
##   A      3   7     7
##   B     10 490    10
##   -SUM- 10   7    17
```

Over- and undersampling wrappers

The over- and undersampling may be dont via the use of wrapped learners as well thus keeping the task unmodified.

```
lrn.over <- makeOversampleWrapper(lrn, osw.rate = 8)
lrn.under <- makeUndersampleWrapper(lrn, usw.rate = 1/8)
mod <- train(lrn, task)
mod.over <- train(lrn.over, task)
mod.under <- train(lrn.under, task)


performance(predict(mod, newdata = data.imbal.test), measures = list(mmce, ber, auc))
```

```
##       mmce        ber        auc
## 0.01960784 0.50000000 0.50000000
```

```
performance(predict(mod.over, newdata = data.imbal.test), measures = list(mmce, ber, auc))
```

```
##       mmce        ber        auc
## 0.01960784 0.50000000 0.65300000
```

```
performance(predict(mod.under, newdata = data.imbal.test), measures = list(mmce, ber, auc))
```

```
##       mmce        ber        auc
## 0.01960784 0.50000000 0.50000000
```

Extensions to oversampling

1. SMOTE (Synthetic Minority Oversampling Technique)

As mentioned previously, in oversampling, copies of minority class observations are created; however, this may lead to overfitting. *SMOTE* constructs the "new" cases by randomly choosing an observation and interpolating it with randomly chosen next neighbors (this number can be set in the function) such that an artificial "new" observation is created. Both numeric and factor features are handled within *SMOTE*. Again, we can either modify the task or use a wrapped learner.

```
#  modify a task
task.smote <- smote(task, rate = 8, nn = 5)
table(getTaskTargets(task))
```

```
##
##    A    B
##  100 5000
```

```
# wrap a learner
lrn.smote <- makeSMOTEWrapper(lrn, sw.rate = 8, sw.nn = 5 )
mod.smote <- train(lrn.smote, task)
performance(predict(mod.smote, newdata = data.imbal.test), measures = list(mmce, ber, auc))
```

```
##       mmce        ber        auc
## 0.04117647 0.41300000 0.65410000
```

2. Overbagging

A second approach is supported in the mlr package for oversampling which is combined with bagging. For each bagging iteration, minority classes are oversampled using `obw.rate`. The majority class cases are either all taken into account with `obw.maxcl = "all"` or bootstrapped with replacement to increase variability between training data sets during iterations using `obw.maxcl = "boot"`.

To create the wrapped learner for bagging + oversampling wrapped learner, we use the function `makeOverBaggingWrapper` in a similar way as the `makeBaggingWrapper`. The number of iters or fitted models is set through the argument `iters`.

```
# first make the base learner
lrn <- makeLearner("classif.rpart", predict.type = "response") # needs to be response for wrapper
obw.lrn <- makeOverBaggingWrapper(lrn, obw.rate = 8, obw.iters = 3)
```

Prediction works as follows: for classification, we do majority voting to create a discrete label and probabilities are predicted by considering the proportions of all predicted labels. The benefits of overbagging are strongly dependent on the learner specified. For instance, overbagging with the *random forest* as the learning algorithm may be of little use since the learner is already a strong, bagged learner. This is shown in the next example. We train on a wrapped decision tree (`rpart`) and a random forest learner with overbagging and compare the performance.

```
lrn <- setPredictType(lrn, "prob")
set.seed(34)
rin <- makeResampleInstance("CV", iters = 5, task = task)
r1 <- resample(learner = lrn, task = task, resampling = rin, show.info = FALSE, measures = list(mmce, b
r1$aggr
```

```
## mmce.test.mean  ber.test.mean  auc.test.mean
##     0.01960784     0.50000000     0.50000000
```

```
obw.lrn <- setPredictType(obw.lrn, predict.type = "prob")
r2 <- resample(learner = obw.lrn, task = task, resampling = rin, show.info = FALSE, measures = list(mmc
r2$aggr
```

```
## mmce.test.mean  ber.test.mean  auc.test.mean
##     0.03137255     0.48989320     0.53833973
```

Now with *random forest*:

```
lrn <- makeLearner("classif.randomForest")
obw.lrn <- makeOverBaggingWrapper(lrn, obw.rate = 8, obw.iters = 3)

lrn <- setPredictType(lrn, "prob")
r1 <- resample(learner = lrn, task = task, resampling = rin, show.info = FALSE, measures = list(mmce, b
r1$aggr
```

```
## mmce.test.mean  ber.test.mean  auc.test.mean
##     0.03647059     0.48804907     0.59139620
```

```
# does this bagging step improve performance?
obw.lrn <- setPredictType(obw.lrn, predict.type = "prob")
r2 <- resample(learner = obw.lrn, task = task, resampling = rin, measures = list(mmce, ber, auc), show.
r2$aggr
```

```
## mmce.test.mean  ber.test.mean  auc.test.mean
##     0.04196078     0.47163079     0.53280816
```

Cost-based approaches We saw applications of weighted class wrappers (through `makeWeightedClassesWrapper`) in the previous section under *cost-sensitive classification*.

ROC analysis and performance curves

Threshold values control how predicted posterior probabilities are converted into class labels. *Receiver operating characterstic* (ROC) curves plot the true positive rate (TPR) which is equivalent to the sensitivity (or recall) on the vertical axis against the false positive rate (FPR) which is equivalent to 1-specificity (SPC) on the horizontal axis for all possible threshold values.

TPR = $\frac{TP}{TP+FN}$ and FPR = 1- SPC, where SPC = $\frac{TN}{TN+FP}$

The ROC curves are useful in:

- assessing performance visualization;
- determining an optimal decision threshold for given class prior probabilities and misclassification costs (helps with cost-sensitive classification?);
- identifying regions where one classifier outperforms another;
- obtaining calibrated estimates of the posterior probabilities.

For more information: Fawcett (2004), Fawcett (2006), Flach (ICML 2004).

Often, in the absence of background knowledge, there is uncertainty about the class priors or misclassification errors at the time of predictions (difficult to quantify costs, or cost variability with time). A good classifier would therefore be required to work well over a range of decision thresholds; the area under the ROC curve (AUC) gives a scalar measure to compare and select classifiers. We have already used `auc` as a measure in the previous section on *imbalanced classification problems*. The `auc` measure is offered by the `ROCR` package and a generalization to multi-class problems is offered by `pROC` through `multiclass.auc`. The latter builds multiple ROC curves to compute the multi-class AUC.

The following methods are available in the `mlr` package for plotting ROC curves and other performance curves:

1. `plotROCCurves`: using the result of `generateThreshVsPerfData`, choose any two measures to plot the performance curve.
2. Use the function `asROCRPrediction` to generate a `ROCR` prediction object which can then be used with `ROCR`'s `performance` function which, in turn, creates performance objects that are then used with `plot` for visualization.
3. ViperCharts is a web-based platform for visual performance evaluation of classification, prediction, and information retrieval algorithms. Using `plotViperCharts` with a prediction object, mlr provides an interface to ViperCharts.

We show some examples of the three methods available in mlr next. Note that to use the above, we need learners that are capable of predicting probabilities.

```
## Warning in listLearners.character("classif", properties = c("twoclass", : The following learners coul
## classif.ada,classif.bartMachine,classif.bdk,classif.blackboost,classif.boosting,classif.bst,classif.
## Check ?learners to see which packages you need or install mlr with all suggestions.
```

```
##                    class    type      package short.name
## 1         classif.avNNet classif         nnet     avNNet
## 2       classif.binomial classif        stats   binomial
## 3            classif.gbm classif          gbm        gbm
## 4            classif.IBk classif        RWeka        ibk
## 5            classif.J48 classif        RWeka        j48
## 6           classif.JRip classif        RWeka       jrip
## 7           classif.ksvm classif      kernlab       ksvm
## 8            classif.lda classif         MASS        lda
## 9         classif.logreg classif        stats     logreg
## 10     classif.multinom classif         nnet   multinom
## 11   classif.naiveBayes classif        e1071     nbayes
## 12          classif.nnet classif         nnet       nnet
## 13          classif.OneR classif        RWeka       oner
## 14          classif.PART classif        RWeka       part
## 15   classif.plsdaCaret classif        caret plsdacaret
## 16        classif.probit classif        stats     probit
## 17           classif.qda classif         MASS        qda
## 18 classif.randomForest classif randomForest         rf
## 19         classif.rpart classif        rpart      rpart
## 20           classif.svm classif        e1071        svm
## 21       classif.xgboost classif      xgboost    xgboost
##                                                 name numerics factors
## 1                                    Neural Network     TRUE    TRUE
```

```
## 2                                Binomial Regression     TRUE    TRUE
## 3                            Gradient Boosting Machine     TRUE    TRUE
## 4                                 k-Nearest Neighbours     TRUE    TRUE
## 5                                   J48 Decision Trees     TRUE    TRUE
## 6                             Propositional Rule Learner     TRUE    TRUE
## 7                               Support Vector Machines     TRUE    TRUE
## 8                           Linear Discriminant Analysis     TRUE    TRUE
## 9                                    Logistic Regression     TRUE    TRUE
## 10                               Multinomial Regression     TRUE    TRUE
## 11                                           Naive Bayes     TRUE    TRUE
## 12                                        Neural Network     TRUE    TRUE
## 13                                        1-R Classifier     TRUE    TRUE
## 14                                     PART Decision Lists     TRUE    TRUE
## 15 Partial Least Squares (PLS) Discriminant Analysis     TRUE   FALSE
## 16                                     Probit Regression     TRUE    TRUE
## 17                      Quadratic Discriminant Analysis     TRUE    TRUE
## 18                                       Random Forest     TRUE    TRUE
## 19                                       Decision Tree     TRUE    TRUE
## 20                     Support Vector Machines (libsvm)     TRUE    TRUE
## 21                             eXtreme Gradient Boosting     TRUE    TRUE
##    ordered missings weights prob oneclass twoclass multiclass
## 1    FALSE    FALSE    TRUE TRUE    FALSE     TRUE       TRUE
## 2    FALSE    FALSE    TRUE TRUE    FALSE     TRUE      FALSE
## 3    FALSE     TRUE    TRUE TRUE    FALSE     TRUE       TRUE
## 4    FALSE    FALSE   FALSE TRUE    FALSE     TRUE       TRUE
## 5    FALSE     TRUE   FALSE TRUE    FALSE     TRUE       TRUE
## 6    FALSE     TRUE   FALSE TRUE    FALSE     TRUE       TRUE
## 7    FALSE    FALSE   FALSE TRUE    FALSE     TRUE       TRUE
## 8    FALSE    FALSE   FALSE TRUE    FALSE     TRUE       TRUE
## 9    FALSE    FALSE    TRUE TRUE    FALSE     TRUE      FALSE
## 10   FALSE    FALSE    TRUE TRUE    FALSE     TRUE       TRUE
## 11   FALSE     TRUE   FALSE TRUE    FALSE     TRUE       TRUE
## 12   FALSE    FALSE    TRUE TRUE    FALSE     TRUE       TRUE
## 13   FALSE     TRUE   FALSE TRUE    FALSE     TRUE       TRUE
## 14   FALSE     TRUE   FALSE TRUE    FALSE     TRUE       TRUE
## 15   FALSE    FALSE   FALSE TRUE    FALSE     TRUE      FALSE
## 16   FALSE    FALSE    TRUE TRUE    FALSE     TRUE      FALSE
## 17   FALSE    FALSE   FALSE TRUE    FALSE     TRUE       TRUE
## 18    TRUE    FALSE   FALSE TRUE    FALSE     TRUE       TRUE
## 19    TRUE     TRUE    TRUE TRUE    FALSE     TRUE       TRUE
## 20   FALSE    FALSE   FALSE TRUE    FALSE     TRUE       TRUE
## 21   FALSE    FALSE    TRUE TRUE    FALSE     TRUE       TRUE
##    class.weights
## 1          FALSE
## 2          FALSE
## 3          FALSE
## 4          FALSE
## 5          FALSE
## 6          FALSE
## 7           TRUE
## 8          FALSE
## 9          FALSE
## 10         FALSE
## 11         FALSE
```

```
## 12         FALSE
## 13         FALSE
## 14         FALSE
## 15         FALSE
## 16         FALSE
## 17         FALSE
## 18          TRUE
## 19         FALSE
## 20          TRUE
## 21         FALSE
##
## 1
## 2
## 3                                          Note on param 'distribution': gbm will select 'bernoulli'
## 4
## 5
## 6
## 7                                                                                  Kernel paramete
## 8
## 9
## 10
## 11
## 12
## 13
## 14
## 15
## 16
## 17
## 18 Note that the rf can freeze the R process if trained on a task with 1 feature which is constant.
## 19
## 20
## 21
```

Using plotROCCurves As we have already seen, the function `generateThreshVsPerfData` generates data on threshold vs performance for 2-class classification that can be used for plotting. The function takes in an object which is a list of `Prediction`, `ResampleResult` or `BenchmarkResult`. The resulting object can then be used with `plotROCCurves` for plotting using `ggplot2`.

Example 1: single predictions

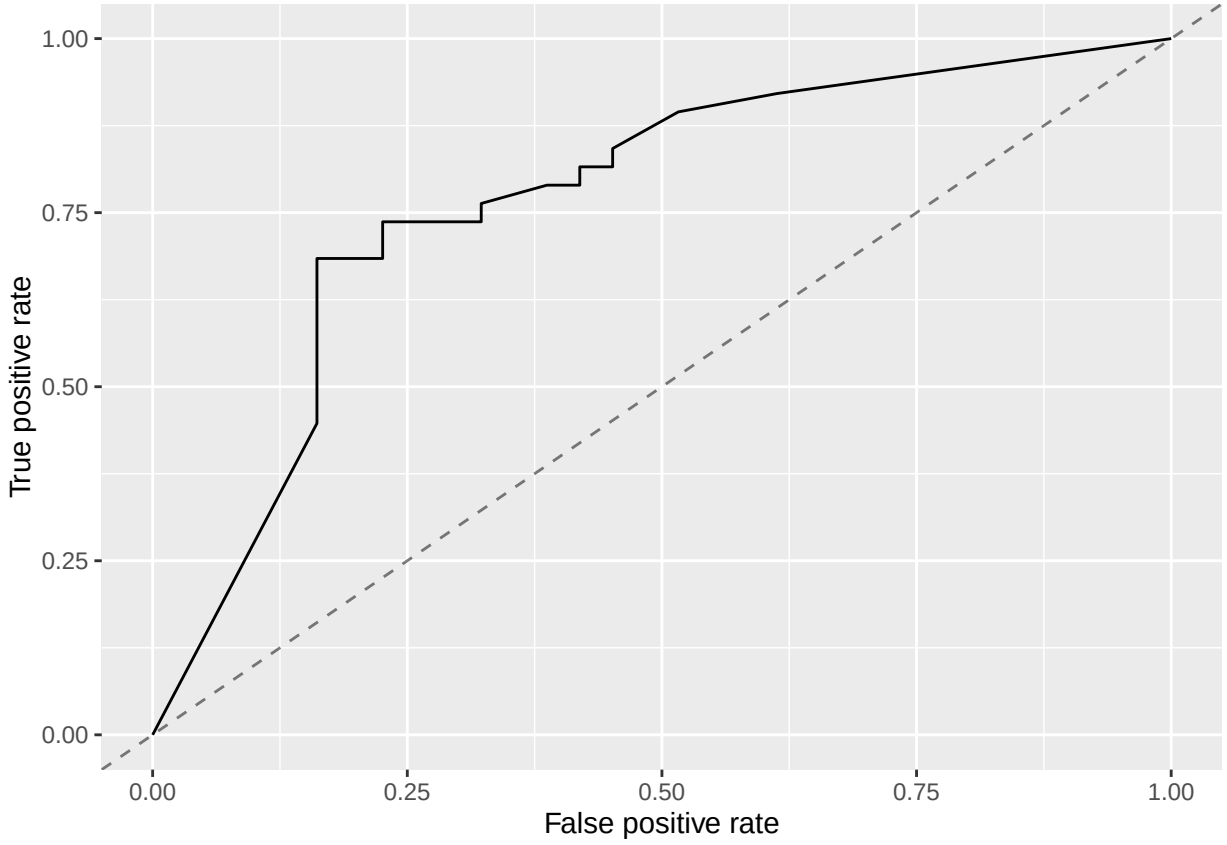
```
n <- getTaskSize(sonar.task)
train.set <- sample(n, size = round(2/3 * n))
test.set <- setdiff(seq_len(n), train.set)

# linear discriminant analysis learner
lrn1 <- makeLearner("classif.lda", predict.type = "prob")
mod1 <- train(lrn1, sonar.task, subset = train.set)
pred1 <- predict(mod1, task = sonar.task, subset = test.set)
```

Since the objective is to plot ROC curves, we generate data on FPR and TPR. We also compute the `mmce`.

```
df <- generateThreshVsPerfData(pred1, measures = list(fpr, tpr, mmce))
```

We now use `df` with `plotROCCurves` which, by default, plots the first 2 measures passed to `generateThreshVsPerfData` and a dashed diagonal indicating the performance of a random classifier (this can be turned off with `diagonal = FALSE`.

```
plotROCCurves(df)
```

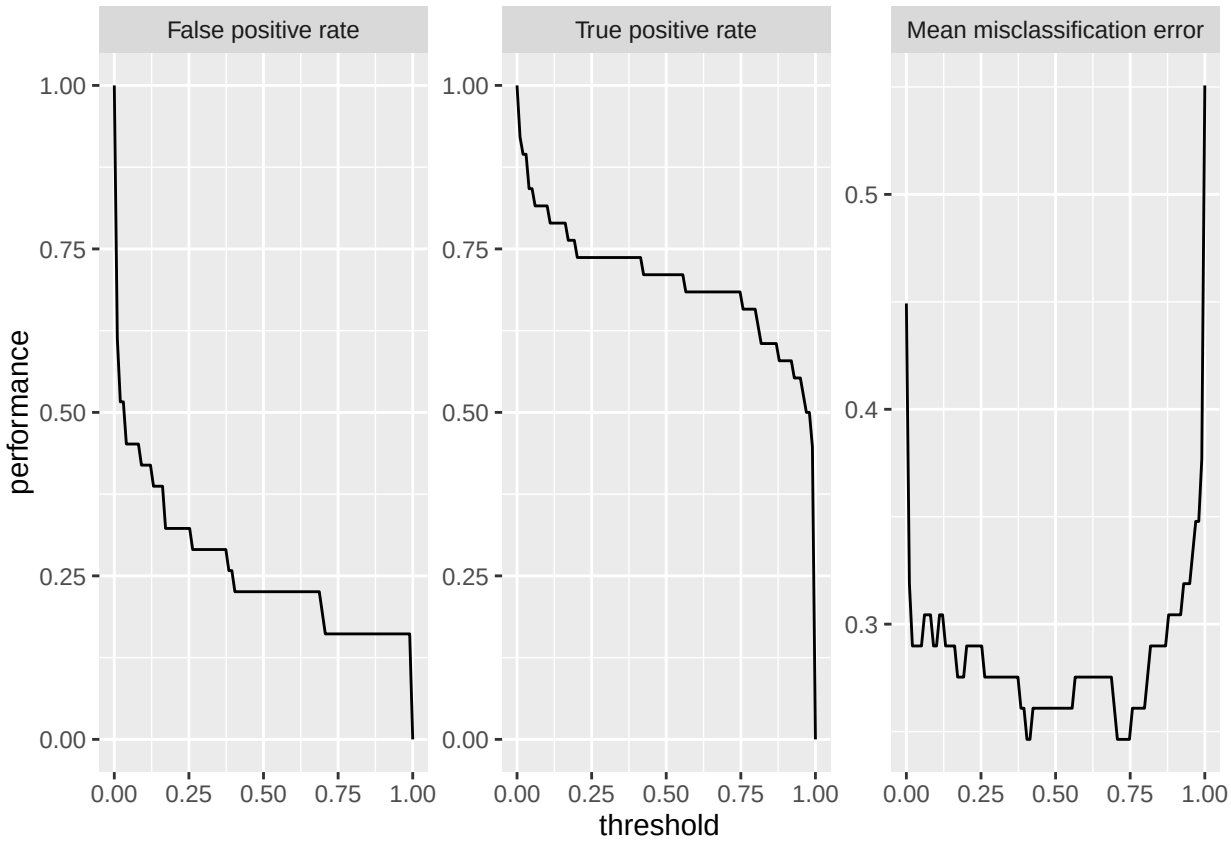



```
# corresponding auc:
performance(pred1, auc)
```

```
##       auc
## 0.7971138
```

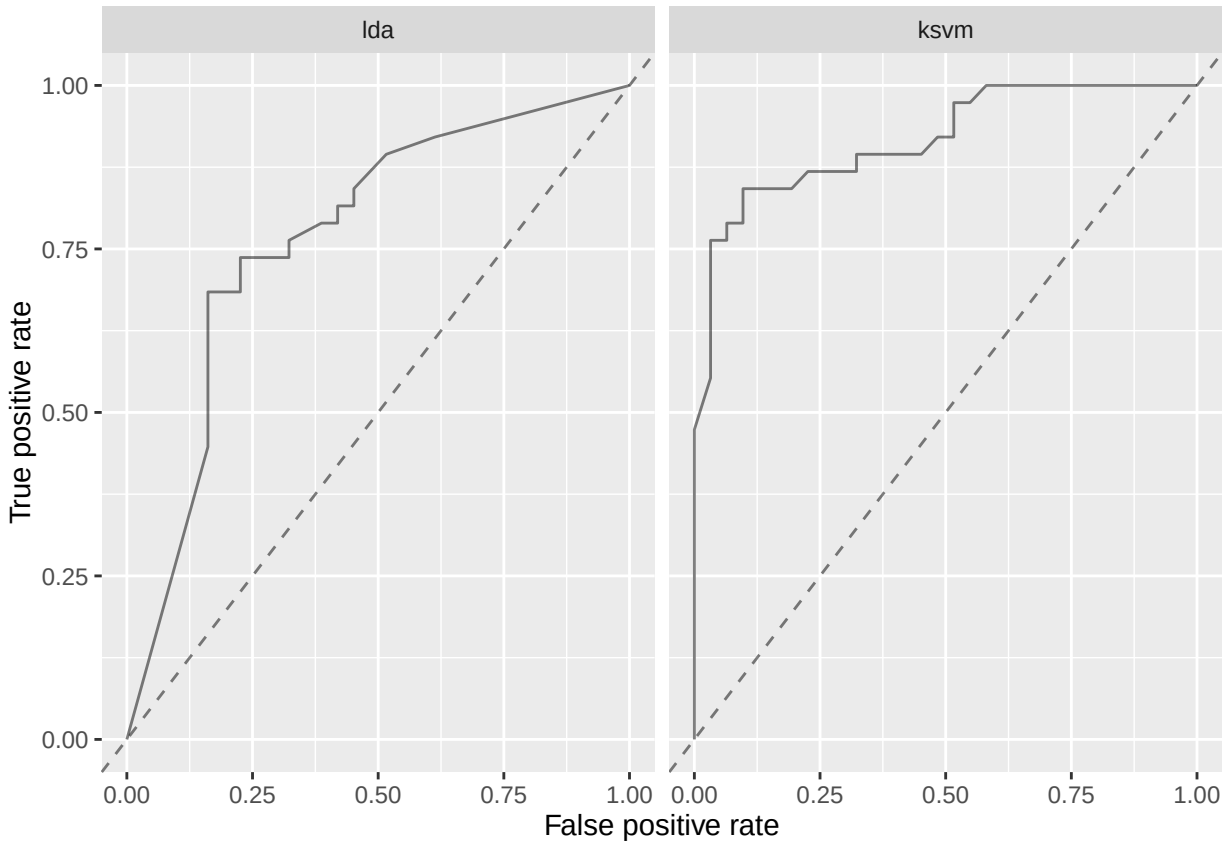
The function `plotROCCurves` plots a pair of performance measures against each other. The individual performance measures against the threshold value may be plotted by calling `plotThreshVsPerf(df)`:

```
plotThreshVsPerf(df)
```



We now try a support vector machine with RBF kerner (`ksvm`) on the `sonar.task` and compare the ROC curve between the `lda` and `ksvm` fitted models.

```
lrn2 <- makeLearner("classif.ksvm", predict.type = "prob")
mod2 <- train(lrn2, sonar.task, subset = train.set)
pred2 <- predict(mod2, task = sonar.task, subset = test.set)

# generateThreshVsPerfData generated using a list of predictions to compare performance side-by-side

df <- generateThreshVsPerfData(list(lda = pred1, ksvm = pred2), measures = list(fpr, tpr, mmce))
plotROCCurves(df)
```

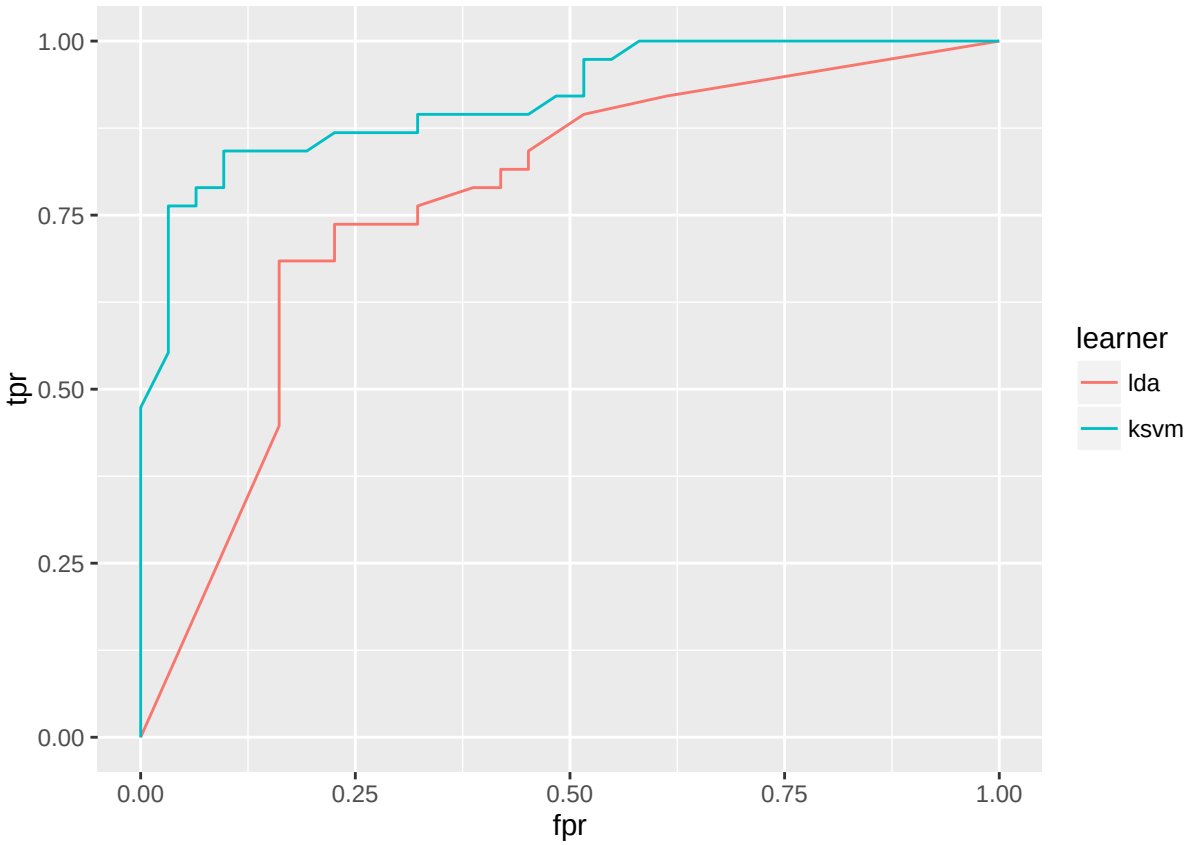



```
performance(pred2, auc)
```

```
##       auc
## 0.9168081
```

The `df` object contains the element `$data` from which we can generate custom plots like, for example, superimposed ROC curves which clearly depict the AUC.

```
qplot(x = fpr, y = tpr, data = df$data, color = learner, geom = "path")
```



With `plotROCCurves` we can plot other performance measures by passing the various measures as arguments in `measures` in `generateThreshVsPerfData`. We then choose the ones to be plotted via the `measures` argument in `plotROCCurves`.

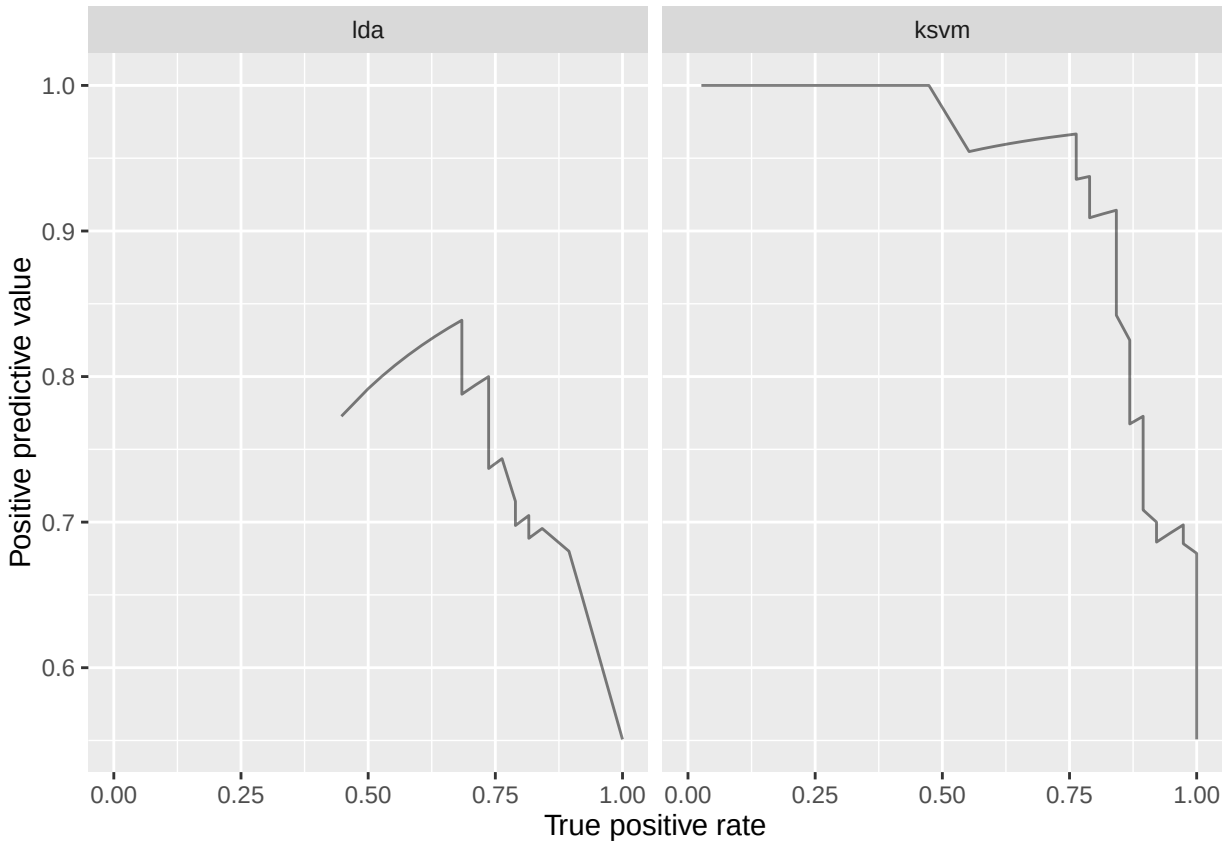
```
df <- generateThreshVsPerfData(list(lda = pred1, ksvm = pred2), measures = list(tpr, ppv, tnr))
```

Note that TPR = sensitivity = recall, TNR = specificity. The *positive predictive value* (PPV) is:

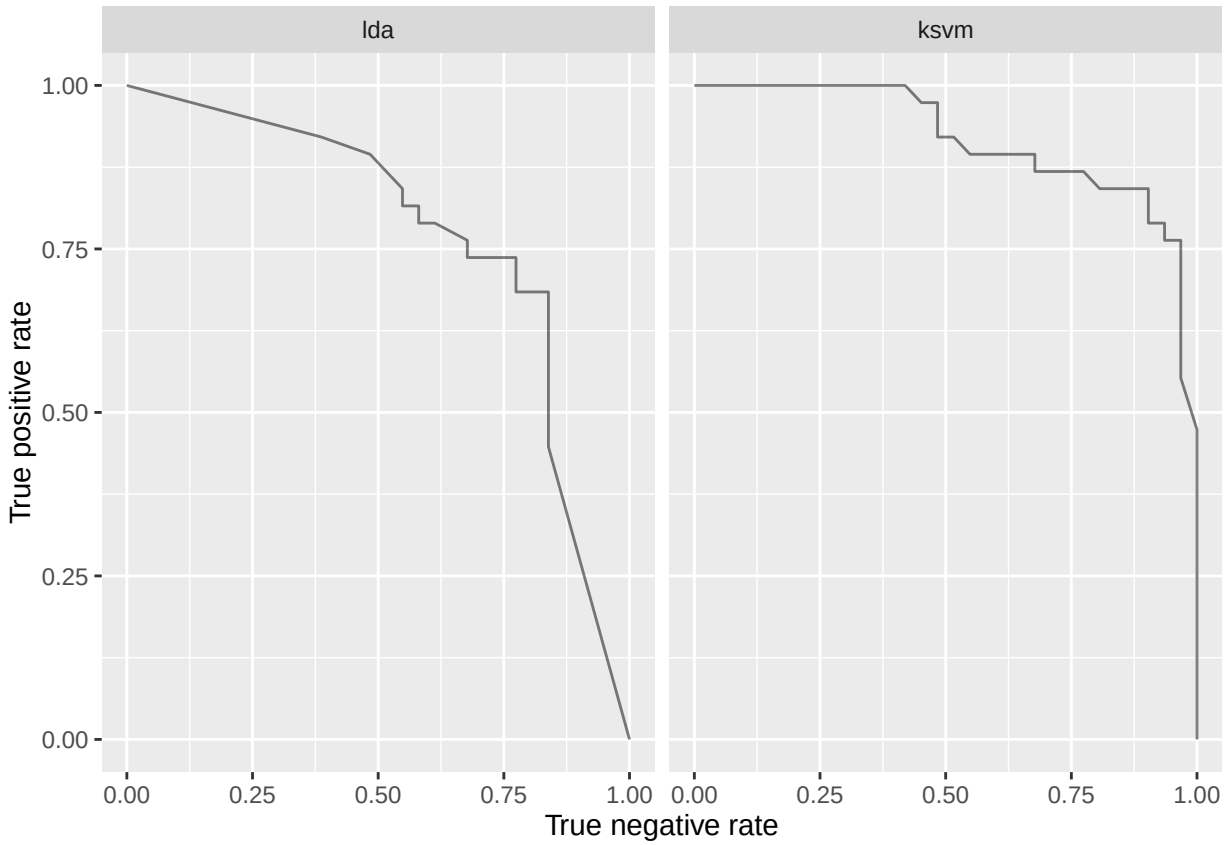
PPV = precision = $\frac{TP}{TP+FP}$.

```
# precision/recall plot
plotROCCurves(df, measures = list(tpr, ppv), diagonal = FALSE)
```

```
## Warning: Removed 1 rows containing missing values (geom_path).
```




```
# sensitivity/specificity plot
plotROCCurves(df, measures = list(tnr, tpr), diagonal = FALSE)
```



Example 2: benchmark experiment

We can combine some of the steps above using `benchmark` and, in addition, tune some of the parameters in the learners. We use the `sonar.task` again to fit `lda` and `ksvm` learners. The cost of constraint violation (or regularization parameter), `C` is tuned via a tune wrapper. We want a good performance over the whole threshold range so we set out to *maximize* the AUC.

```
# tune wrapper for ksvm
rdesc.inner <- makeResampleDesc("Holdout")
ms <- list(auc, mmce)
ps <- makeParamSet(
  makeDiscreteParam("C", 2^(-1:1))
)
ctrl <- makeTuneControlGrid()
lrn2 <- makeTuneWrapper(lrn2, resampling = rdesc.inner, measures = ms, par.set = ps, control = ctrl, sh
```

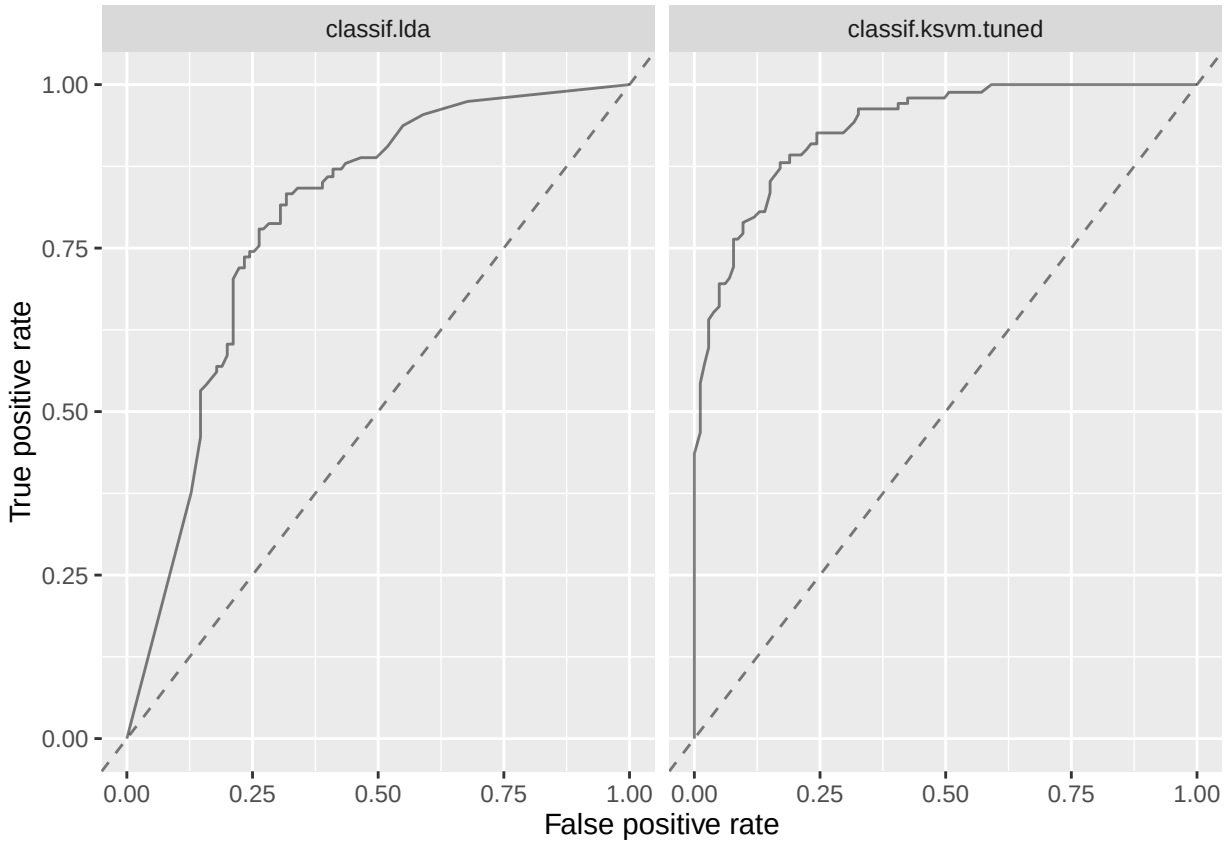
The becnhmark experiment is conducted with the `lda` learner and the tuned `ksvm` learner with an outer resampling strategy.

```
# the benchmark experiment
lrns <- list(lrn1, lrn2)
rdesc.outer <- makeResampleDesc("CV", iters = 5)

bmr <- benchmark(lrns, tasks = sonar.task, resamplings = rdesc.outer, measures = ms, show.info = FALSE)
bmr
```

```
##         task.id         learner.id auc.test.mean mmce.test.mean
```

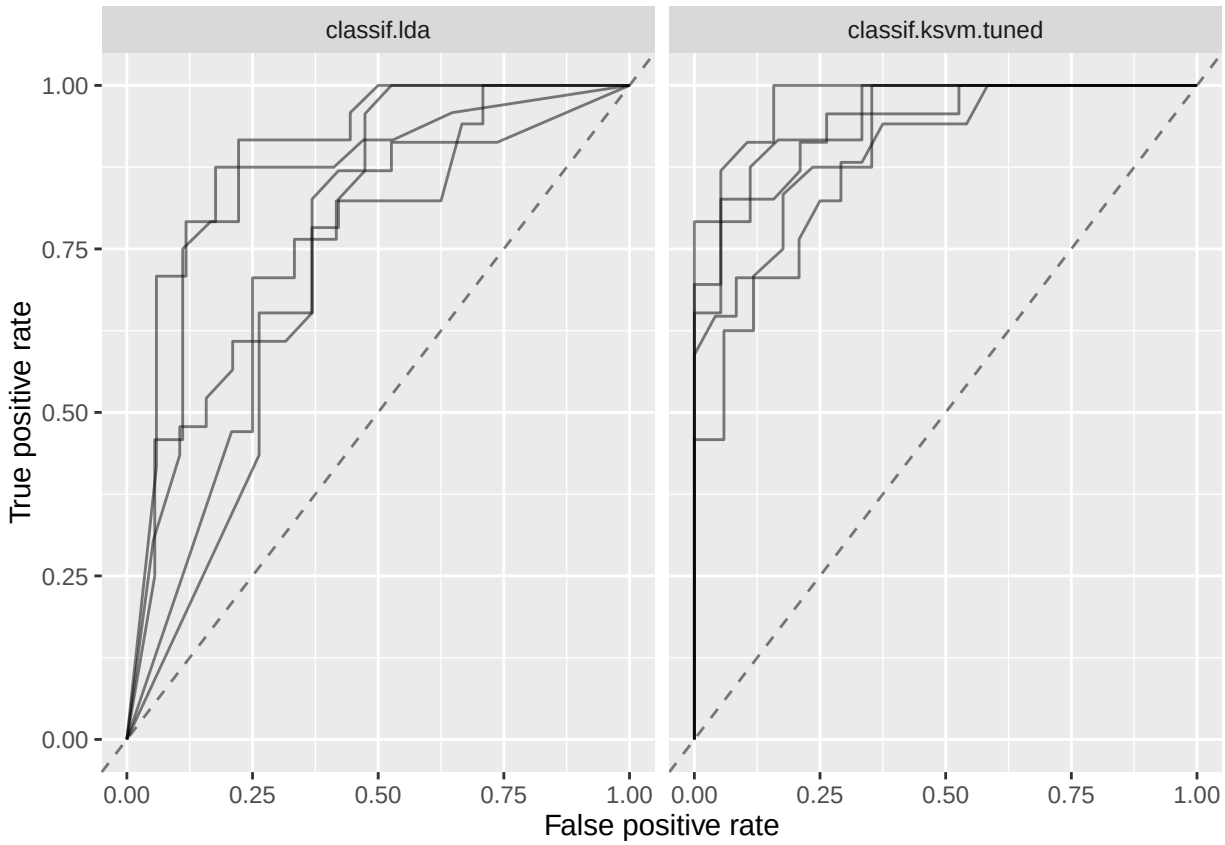
```
## 1 Sonar-example        classif.lda     0.7925862      0.2545877
## 2 Sonar-example classif.ksvm.tuned     0.9335838      0.1543554
```

```
# the ROC curves
df <- generateThreshVsPerfData(bmr, measures = list(fpr, tpr, mmce))
plotROCCurves(df)
```

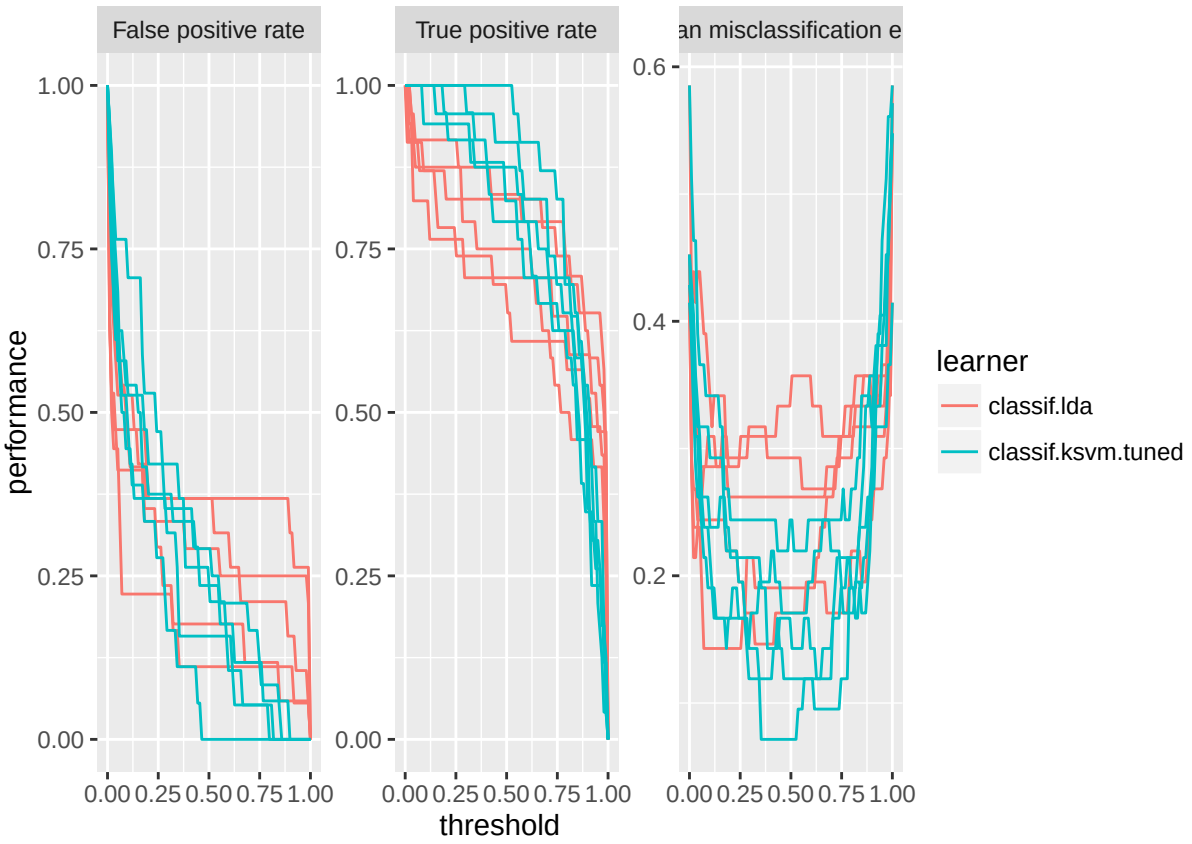


The object returned from `generateThreshVsPerf` calculates aggregated performances according to the resampling strategy. The resulting ROC curves are threshold-averaged. If we want to keep the performances from the individual iterations, we can set `aggregate = FALSE`.

```
# performance measure from individual iterations
df <- generateThreshVsPerfData(bmr, measures = list(fpr, tpr, mmce), aggregate = FALSE)
plotROCCurves(df)
```

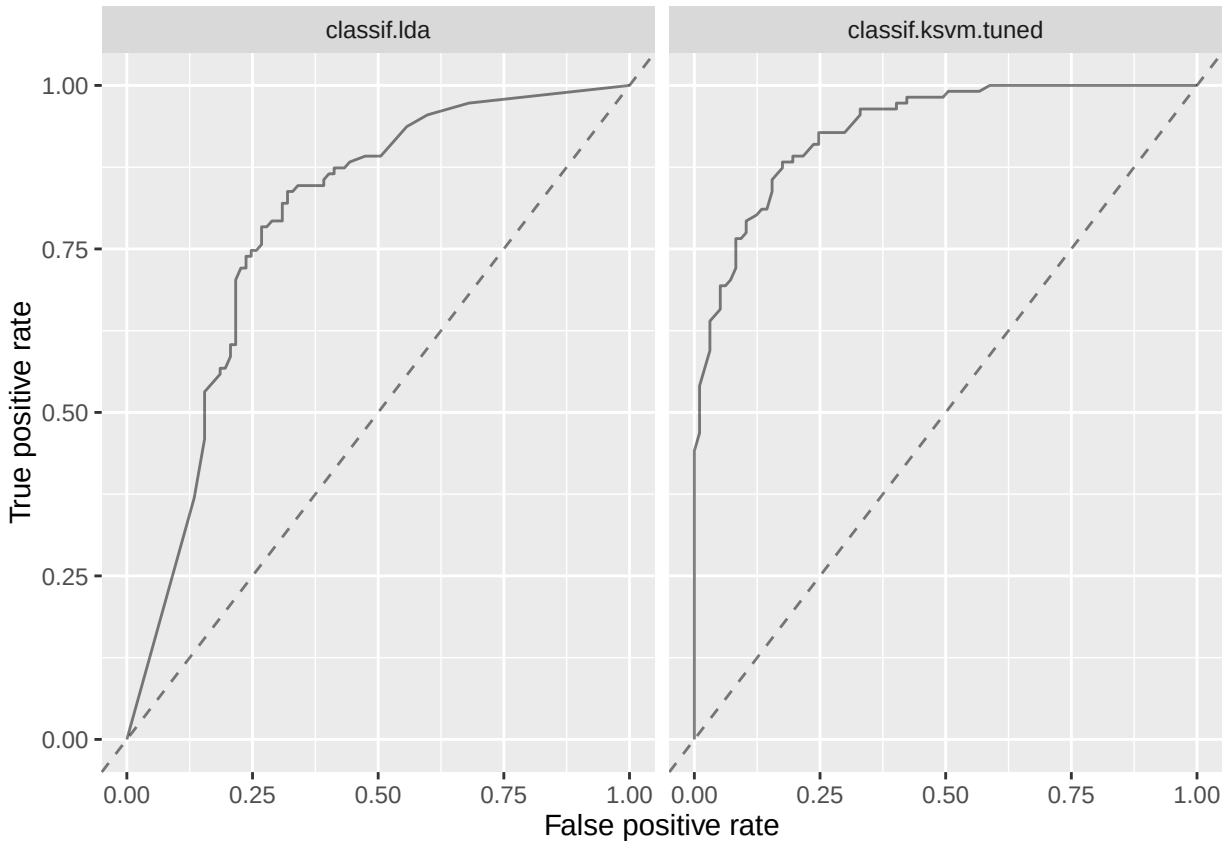



```
plotThreshVsPerf(df)
```



The 5 test folds may also be merged and draw a single ROC curve instead of averaging them as done above. Averaging methods are preferred as they take into account variability which is needed to compare classifier performance.

```
preds <- getBMRPredictions(bmr)[[1]]

# merging is achieved by changing the class attribute to Prediction
preds2 <- lapply(preds, function(x) {class(x) = "Prediction"; return(x)})

df <- generateThreshVsPerfData(preds2, measures = list(fpr, tpr, mmce))
plotROCCurves(df)
```

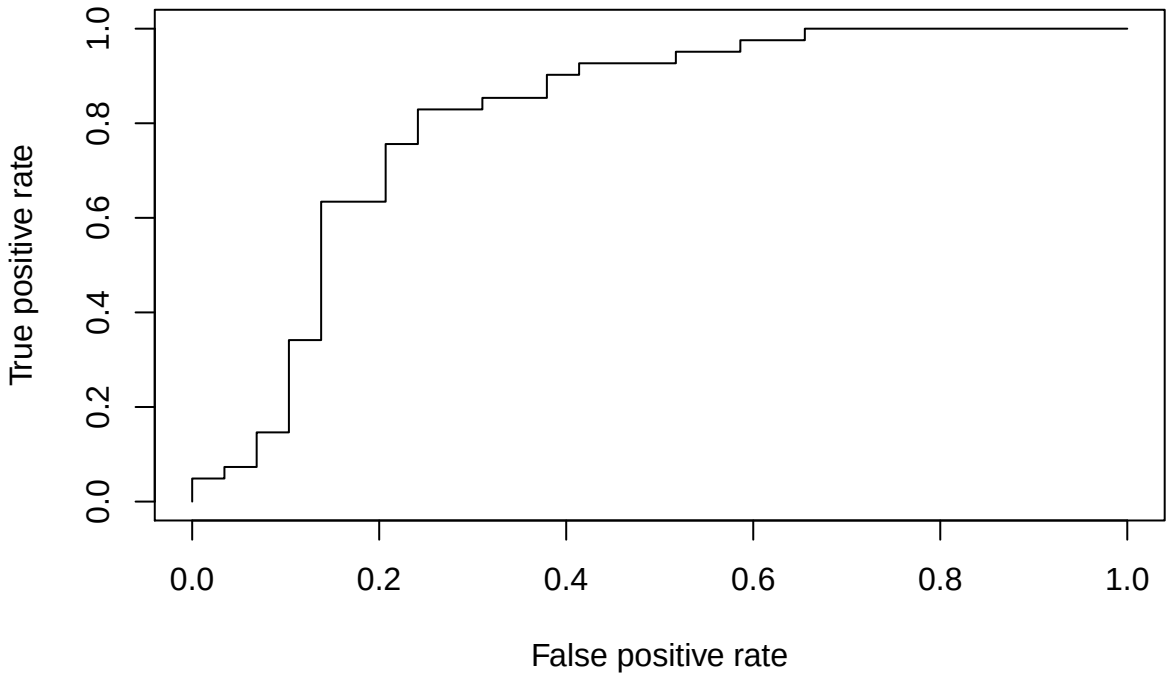


Using asROCRPrediction As mentioned before, the mlr package provides an interface with the `ROCR` package to draw performance plots. To create the plots, we need a ROCR prediction object which we use with `ROCR::performance` to calculate one or more performance measures and finally, we use `ROCR::plot` to generate the performance plot.

Example 1: single predictions

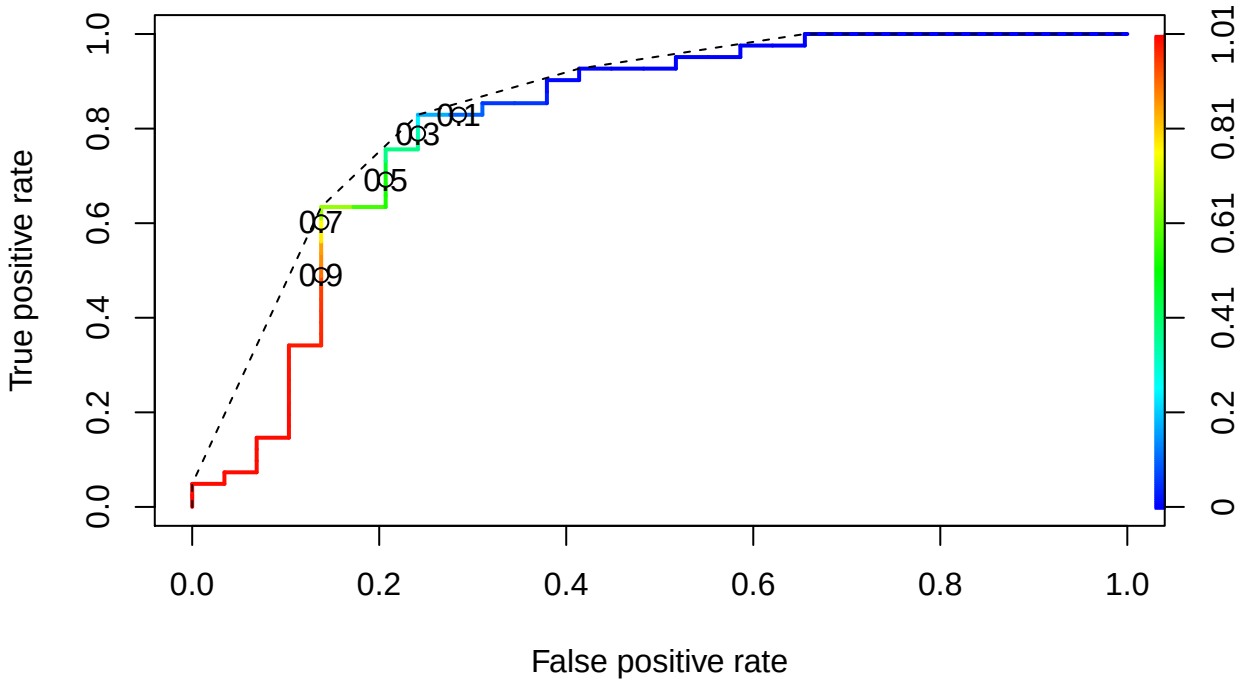
We revisit the `lda` learner trained on the `sonar.task`.

```
n <- getTaskSize(sonar.task)
train.set <- sample(n, size = (2/3 * n))
test.set <- setdiff(seq_len(n), train.set)

lrn1 <- makeLearner("classif.lda", predict.type = "prob")
mod1 <- train(lrn1, sonar.task, subset = train.set)
pred1 <- predict(mod1, task = sonar.task, subset = test.set)

# convert pred1 to ROCR prediction
ROCRpred1 <- asROCRPrediction(pred1)
# calculate perf measures
ROCRperf1 <- ROCR::performance(ROCRpred1, "tpr", "fpr")
# generate plots
ROCR::plot(ROCRperf1)
```



The advantages of using the ROCR interface lie in the graphical properties available: we can create an ROC curve which is color-coded according to threshold, print selected threshold values onto the curve, and superimpose the convex hull. The convex hull of a set of points in ROC space is a piecewise linear curve connecting a selection of points such that all other points lie below it. The curve is *convex* because each line segment has a slope that it not steeper than the previous one. The curve is drawn with a black, dashed line in the plot below.

```
ROCR::plot(ROCRperf1, colorize = TRUE, print.cutoffs.at = seq(0.1, 0.9, 0.2), lwd = 2)

ch <- ROCR::performance(ROCRpred1, "rch")
ROCR::plot(ch, add = TRUE, lty = 2)
```



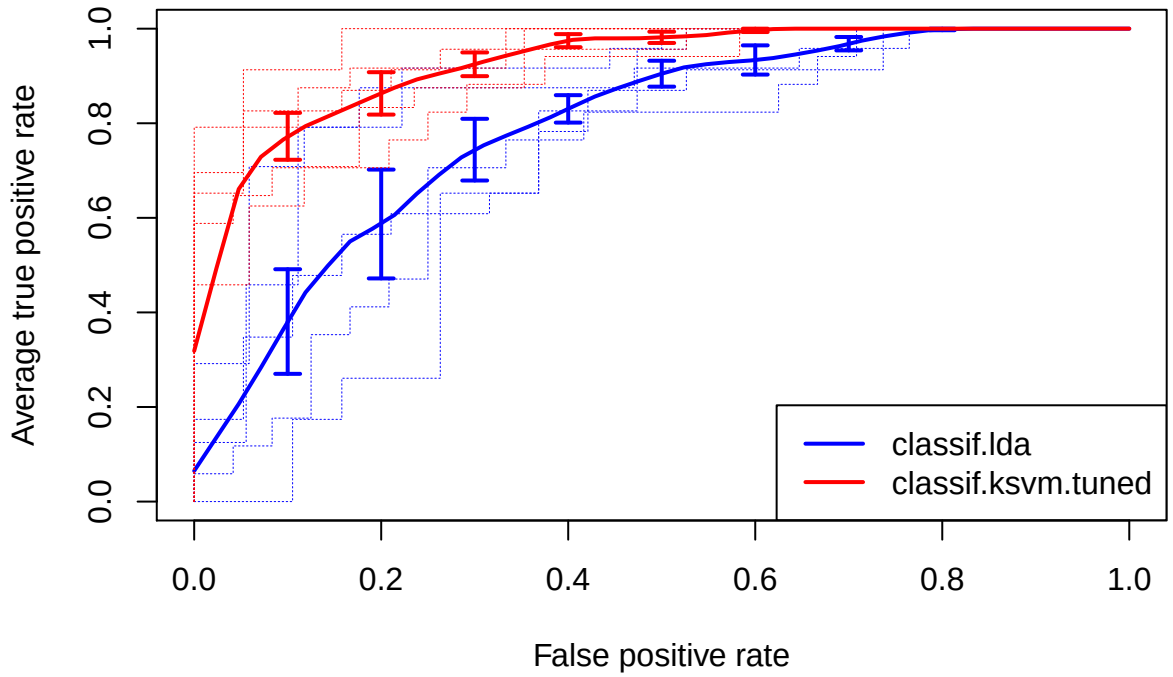
Example 2: benchmark experiments

Using the same benchmark experiments conducted above, we show the functionality of performance plots with ROCR.

```
preds <- getBMRPredictions(bmr)[[1]]
ROCRpreds <- lapply(preds, asROCRPrediction)
ROCRperfs <- lapply(ROCRpreds, function(x) ROCR::performance(x, "tpr", "fpr"))
```

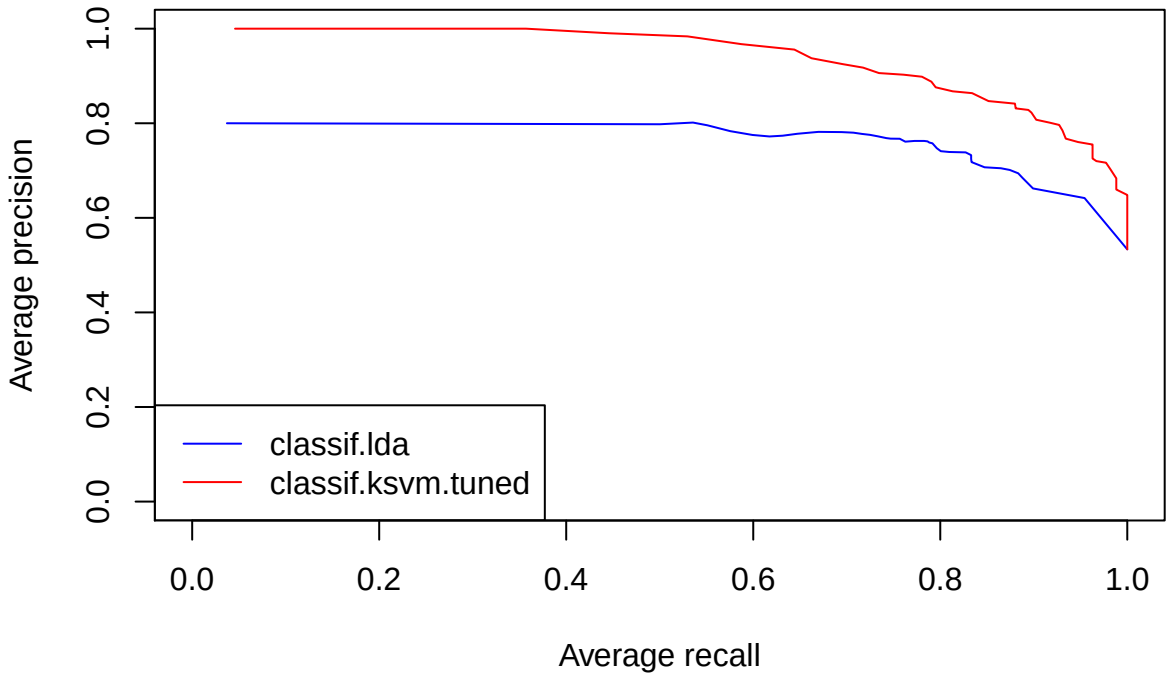
The resampling strategy used in the benchmark experiments was a 5-fold, cross-validation so we obtain 5 different models for each classifier trained. We now plot the ROC curves for each iteration (dashed), the averaged ROC (solid), as well as standard error bars for selected fpr. Note that horizontal averaging is also possible as well as threshold averaging (see later plot). As seen previously, different plot commands can be used in sequence to add to the same figure by selecting `add = TRUE`. Note that the function `plotCI` used below is used internally to plot the error bars.

```
# averaged learner 1
plot(ROCRperfs[[1]], col = "blue", avg = "vertical", spread.estimate = "stderror", show.spread.at = seq
# individual perf 5-fold learner 1
plot(ROCRperfs[[1]], col = "blue", lty = 2, lwd = .25, add = TRUE)

# averaged learner 2
plot(ROCRperfs[[2]], col = "red", avg = "vertical", spread.estimate = "stderror", show.spread.at = seq(0
# individual perf 5-fold learner 2
plot(ROCRperfs[[2]], col = "red", lty = 2, lwd = .25, add = TRUE)

legend("bottomright", legend = getBMRLearnerIds(bmr), lty = 1, lwd = 2, col = c("blue", "red"))
```



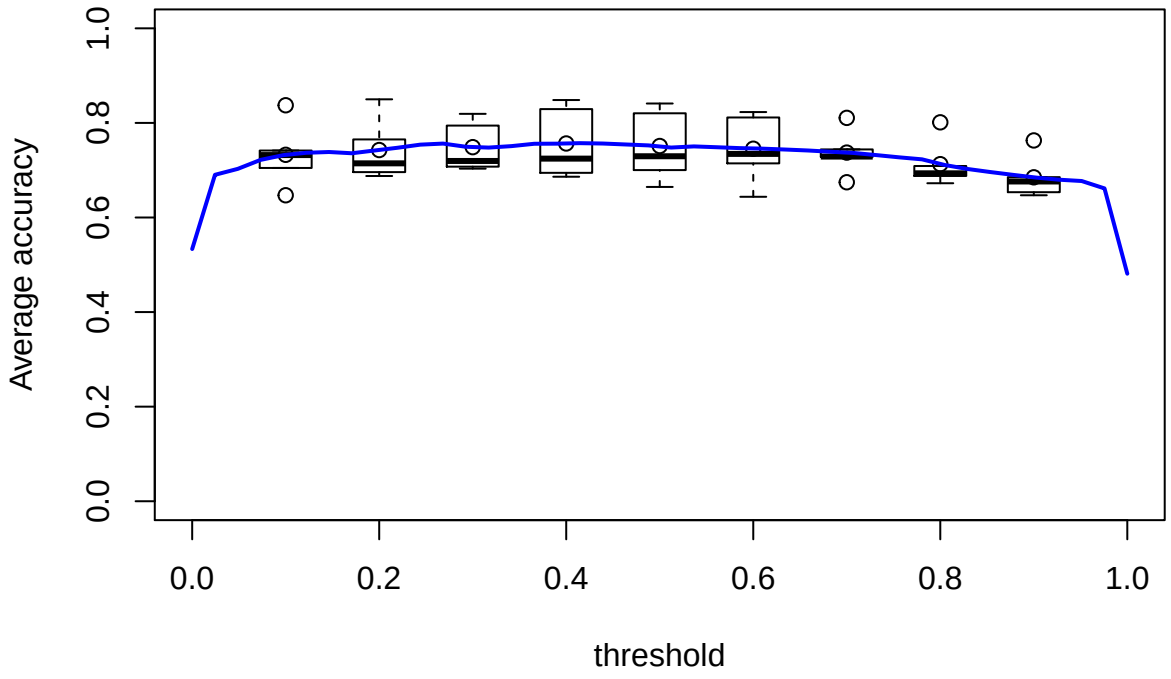
It is possible to plot other performance measures. In the next plot, we generate a precision/recall evaluation plot. Here, we obtain a threshold-averaged curve by setting `avg = "threshold"` in `plot`. Note that the ids for the precision and recall performance measures as defined in ROCR are given as `prec` and `rec`, respectively.

```
preds <- getBMRPredictions(bmr)[[1]]
ROCRpreds <- lapply(preds, asROCRPrediction)

ROCRperfs <- lapply(ROCRpreds, function(x) ROCR::performance(x, "prec", "rec"))

plot(ROCRperfs[[1]], col = "blue", avg = "threshold")
plot(ROCRperfs[[2]], col = "red", avg = "threshold", add = TRUE)
legend("bottomleft", legend = getBMRLearnerIds(bmr), lty = 1, col = c("blue", "red"))
```



It may be of use to plot a single performance measure against the threshold values; this is achieved by calling `ROCR::performance` with only one performance measure (e.g. `acc`). The following plot shows the vertically averaged curve of accuracy vs threshold with the vertical variation around the curve visualized as boxplots.

```
preds <- getBMRPredictions(bmr)[[1]]
ROCRpreds <- lapply(preds, asROCRPrediction)

ROCRperfacc <- lapply(ROCRpreds, function(x) ROCR::performance(x, "acc"))

plot(ROCRperfacc[[1]], avg = "vertical", spread.estimate = "boxplot", show.spread.at = seq(.1, .9, .1),
```



Viper charts Using `mlr::plotViperCharts` with an object of class `Prediction`, `ResampleResult` or `BenchmarkResult` gererates a url which hosts the ViperCharts curves. The argument `chart` takes a character which represents the first chart to display in focus in browser. All other charts can be displayed by clicking on the browser page menu. The default chart is `rocc`. If `browse = TRUE`, the url opens in the default browser.

```
z = plotViperCharts(bmr, chart = "rocc", browse = FALSE)
```

Click here for the plot.

Multilabel classification

In multilabel classification, there exist multiple targets that can be assigned to each observation vs just one target in multiclass classification. An example is the following: predict topics that are relevant for a document. For instance, a text might be about any of religion, politics, finance or education at the same time or none of these. Two approaches exist to dealing with multilabel problems:

1. *Problem transformation* : transform the multilabel classification problem to binary or multiclass problems.
2. *Algorithm adaptation* : adapt multiclass algorithms so that they can be directly applied to the problem.

Creating a task We need to create a task using data in the right format. The data frame needs to consist of the features and a logical vector indicating whether each label is present in the observation or not. Once we have the data in the right format, we can use `makeMultilabelTask`. In what follows, we use the *yeast* data which has 2417 observations, 14 target labels and 103 features. We recreate the multilabel classification `yeast.task` already included in mlr by extracting the data, labels and feeding them in`makeMultilabelTask`.

```
yeast <- getTaskData(yeast.task)
labels <- colnames(yeast)[1:14]
yeast.task <- makeMultilabelTask(id = "multi", data = yeast, target = labels)
yeast.task
```

```
## Supervised task: multi
## Type: multilabel
## Target: label1,label2,label3,label4,label5,label6,label7,label8,label9,label10,label11,label12,label
## Observations: 2417
## Features:
## numerics  factors  ordered
##      103        0        0
## Missings: FALSE
## Has weights: FALSE
## Has blocking: FALSE
## Classes: 14
##  label1  label2  label3  label4  label5  label6  label7  label8  label9
##     762    1038     983     862     722     597     428     480     178
## label10 label11 label12 label13 label14
##     253     289    1816    1799      34
```

Constructing a learner 1. Problem transformation: binary relevance method

To use this method we create a classification learner the usual way and then use the *binary relevance method* to convert the multilabel classification problem into simple binary classifications for each target on which the

binary learner is applied. In mlr, any learner which supports binary classification can be converted into a binary relevance wrapper.

```
lrn <- makeLearner("classif.rpart", predict.type = "prob")
multilabel.lrn1 <- makeMultilabelBinaryRelevanceWrapper(lrn)
multilabel.lrn1
```

```
## Learner multilabel.classif.rpart from package rpart
## Type: multilabel
## Name: ; Short name:
## Class: MultilabelBinaryRelevanceWrapper
## Properties: numerics,factors,ordered,missings,weights,prob,twoclass,multiclass
## Predict-Type: prob
## Hyperparameters: xval=0
```

2. Agorithm adaptation

Currently, the only available algorithm adaptation method in `R` which can be used directly to the multilabel classification task is the *random ferns* multilabel algorithm from package `rFerns`.

```
multilabel.lrn2 <- makeLearner("multilabel.rFerns")
multilabel.lrn2
```

```
## Learner multilabel.rFerns from package rFerns
## Type: multilabel
## Name: Random ferns; Short name: rFerns
## Class: multilabel.rFerns
## Properties: numerics,factors,ordered
## Predict-Type: response
## Hyperparameters:
```

Train, predict, performance Once we have the task and the learner, training, predicting, and performance evaluation follow from the previous sections of this document.

```
mod1 <- mlr::train(multilabel.lrn1, task = yeast.task)
# can also pass subsets + weights
mod1 <- mlr::train(multilabel.lrn1, task = yeast.task, subset = 1:1500, weights = rep(1/1500, 1500))

mod2 <- mlr::train(multilabel.lrn2, yeast.task, subset = 1:100)
mod2
```

```
## Model for learner.id=multilabel.rFerns; learner.class=multilabel.rFerns
## Trained on: task.id = multi; obs = 100; features = 103
## Hyperparameters:
```

```
pred1 <- predict(mod1, task = yeast.task, subset = 1:10) # or
pred1 <- predict(mod1, newdata = yeast[1501:1600, ])
names(as.data.frame(pred1))
```

```
##  [1] "truth.label1"     "truth.label2"     "truth.label3"
##  [4] "truth.label4"     "truth.label5"     "truth.label6"
```

```
##  [7] "truth.label7"     "truth.label8"     "truth.label9"
## [10] "truth.label10"    "truth.label11"    "truth.label12"
## [13] "truth.label13"    "truth.label14"    "prob.label1"
## [16] "prob.label2"      "prob.label3"      "prob.label4"
## [19] "prob.label5"      "prob.label6"      "prob.label7"
## [22] "prob.label8"      "prob.label9"      "prob.label10"
## [25] "prob.label11"     "prob.label12"     "prob.label13"
## [28] "prob.label14"     "response.label1"  "response.label2"
## [31] "response.label3"  "response.label4"  "response.label5"
## [34] "response.label6"  "response.label7"  "response.label8"
## [37] "response.label9"  "response.label10" "response.label11"
## [40] "response.label12" "response.label13" "response.label14"
```

```
pred2 <- predict(mod2, task = yeast.task)
names(as.data.frame(pred2))
```

```
##  [1] "id"               "truth.label1"     "truth.label2"
##  [4] "truth.label3"     "truth.label4"     "truth.label5"
##  [7] "truth.label6"     "truth.label7"     "truth.label8"
## [10] "truth.label9"     "truth.label10"    "truth.label11"
## [13] "truth.label12"    "truth.label13"    "truth.label14"
## [16] "response.label1"  "response.label2"  "response.label3"
## [19] "response.label4"  "response.label5"  "response.label6"
## [22] "response.label7"  "response.label8"  "response.label9"
## [25] "response.label10" "response.label11" "response.label12"
## [28] "response.label13" "response.label14"
```

The prediction object gives us true and predicted values and, depending on the `predict.type` of the learner, we can also get probabilities for each class label. The `get` functions `getPrediction`(Truth, Response, Probabilites) can be used with the prediciton object to extract the associated information.

For performance evaluation, we use the `performance` function with the prediction object and we optionally define the measures we want to evaluate. The default measure for multilabel classification is the *Hamming loss* (`hamloss`). The Hamming loss is the fraction of labels that are incorrectly predicted and therefore, the smaller the Hamming loss, the better the performance.

```
# available measures
listMeasures("multilabel")
```

```
## [1] "timepredict" "featperc"    "timeboth"    "timetrain"   "hamloss"
```

```
performance(pred1)
```

```
##   hamloss
## 0.2257143
```

```
performance(pred2, measures = list(hamloss, timepredict))
```

```
##     hamloss timepredict
##   0.7040014   0.0410000
```

Resampling In the above examples we did not specify a resampling strategy. As usual, we prefer to evaluate the performance of a learning algorithm when trained with resampling. The function `resample` may be used as in other parts of this tutorial.

```
rdesc <- makeResampleDesc("CV", stratify = FALSE, iters = 3)
r1 <- resample(multilabel.lrn1, yeast.task, rdesc, show.info = FALSE)
r1
```

```
## Resample Result
## Task: multi
## Learner: multilabel.classif.rpart
## hamloss.aggr: 0.22
## hamloss.mean: 0.22
## hamloss.sd: 0.00
## Runtime: 4.16422
```

```
r2 <- resample(multilabel.lrn2, yeast.task, rdesc, show.info = FALSE)
r2
```

```
## Resample Result
## Task: multi
## Learner: multilabel.rFerns
## hamloss.aggr: 0.47
## hamloss.mean: 0.47
## hamloss.sd: 0.00
## Runtime: 0.35233
```

Binary performance We can also calculate binary performance measures such as the `mmce`, `auc`, and `acc` for each label using the prediction object returned from the model based on the binary relevance wrapper. The function to use here is `getMultilabelBinaryPerformances`. Note that to claclulate `auc`, we need predicted probabilities.

```
# on prediction object returned from predict
getMultilabelBinaryPerformances(pred1, measures = list(mmce, acc, auc))
```

```
##         mmce.test.mean acc.test.mean auc.test.mean
## label1            0.25          0.75     0.6321925
## label2            0.36          0.64     0.6547917
## label3            0.32          0.68     0.7118227
## label4            0.31          0.69     0.6764835
## label5            0.27          0.73     0.6676923
## label6            0.30          0.70     0.6417739
## label7            0.19          0.81     0.5968750
## label8            0.27          0.73     0.5164474
## label9            0.11          0.89     0.4688458
## label10           0.14          0.86     0.3996463
## label11           0.15          0.85     0.5000000
## label12           0.24          0.76     0.5330667
## label13           0.25          0.75     0.5938610
## label14           0.00          1.00            NA
```

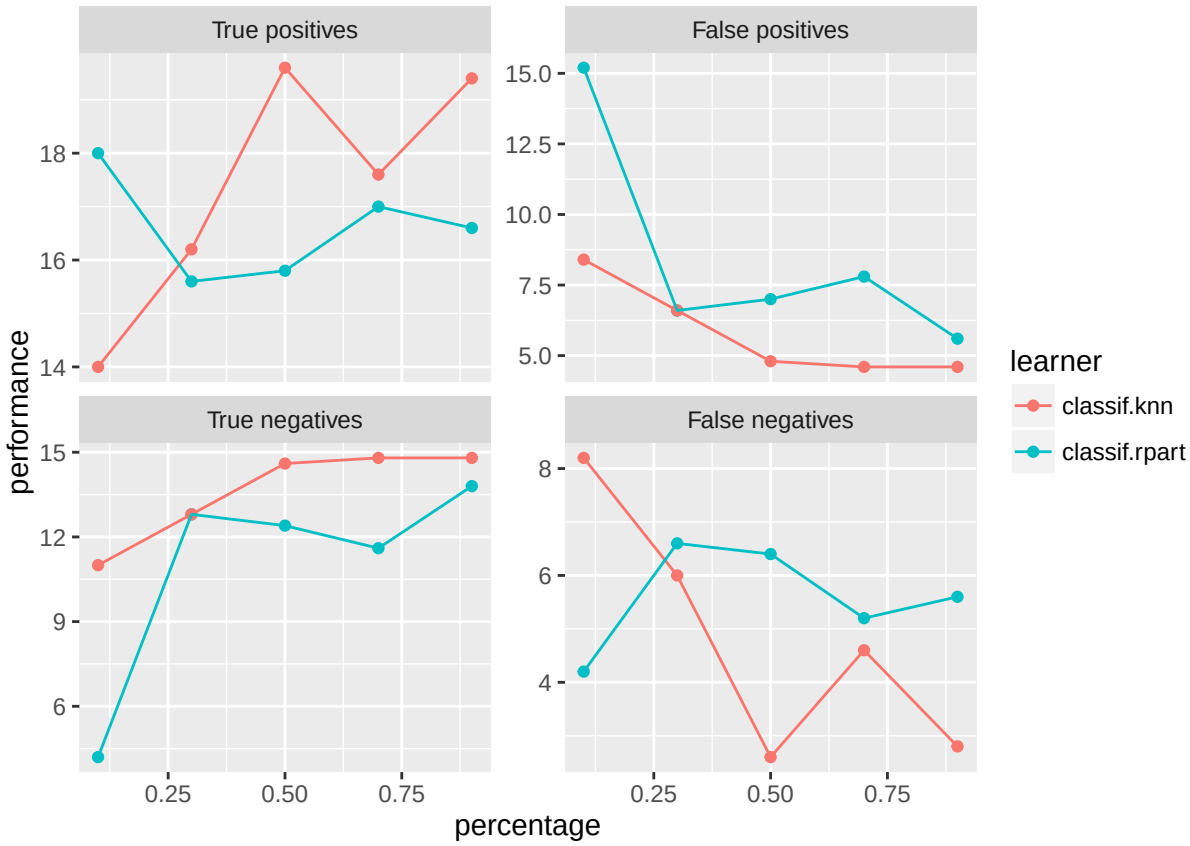
```r
# on prediction object returned from resample
getMultilabelBinaryPerformances(r1$pred, measures = list(mmce, acc, auc))
```

```
##         mmce.test.mean acc.test.mean auc.test.mean
## label1       0.26313612     0.7368639     0.6932480
## label2       0.39801407     0.6019859     0.6111557
## label3       0.34671080     0.6532892     0.6565029
## label4       0.29499379     0.7050062     0.7127271
## label5       0.24906909     0.7509309     0.6524673
## label6       0.24741415     0.7525859     0.6125854
## label7       0.19445594     0.8055441     0.6169299
## label8       0.21845263     0.7815474     0.5900286
## label9       0.07529996     0.9247000     0.4693106
## label10      0.12908564     0.8709144     0.5716001
## label11      0.14935871     0.8506413     0.5807116
## label12      0.26644601     0.7335540     0.4894188
## label13      0.29002896     0.7099710     0.5037921
## label14      0.01406703     0.9859330     0.4603626
```

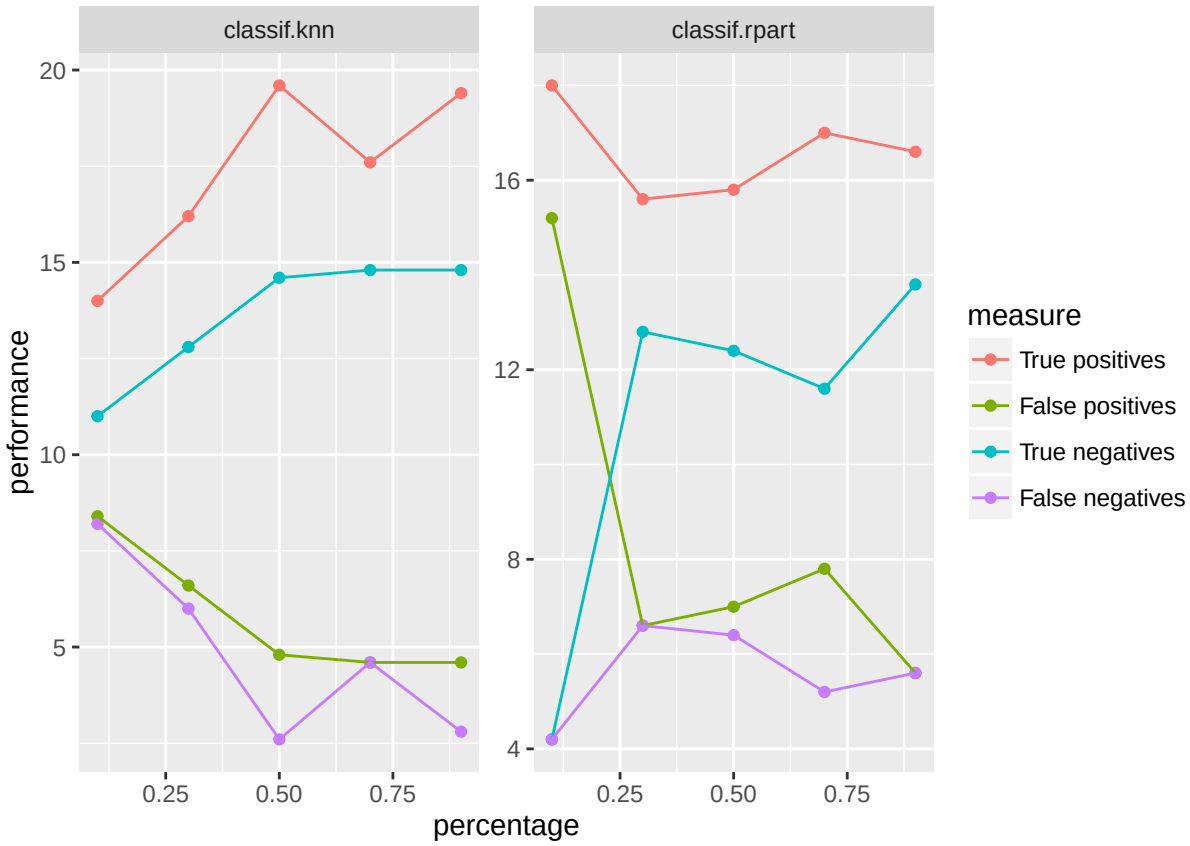
Learning curves

The mlr package has functions used to generate learning curves and visualize them to improve the performance of a learner. Learning curves usually depict the training error and cross-validation (test) error with increasing data size. These curves are useful in the sense that they can be indicative of a high bias or high variance model. The function `generateLearningCurveData` controls the size of the training observations (through `percs`, the vector of percentages to be drawn from the training split), trains the learner on a single percentage of the training set and evaluates the chosen (aggregated) performance measures on the *complete* test set. The complete test set is specified in the resampling strategy (default is `Holdout`). This is repeated for all values in the percentage vector. Note that for each percentage value, randomly selected observations are drawn from the training set which can result in noisy performance measures. The noise can be reduced by increasing the number of iterations in `resampling`.

```r
r <- generateLearningCurveData(
  learners = list("classif.rpart", "classif.knn"),
  task = sonar.task,
  percs = seq(0.1, 1, by = 0.2),
  measures = list(tp, fp, tn, fn),
  resampling = makeResampleDesc("CV", iters = 5),
  show.info = FALSE
)

# facetted plots created for each measure, learners mapped to color
plotLearningCurve(r, facet = "measure")
```

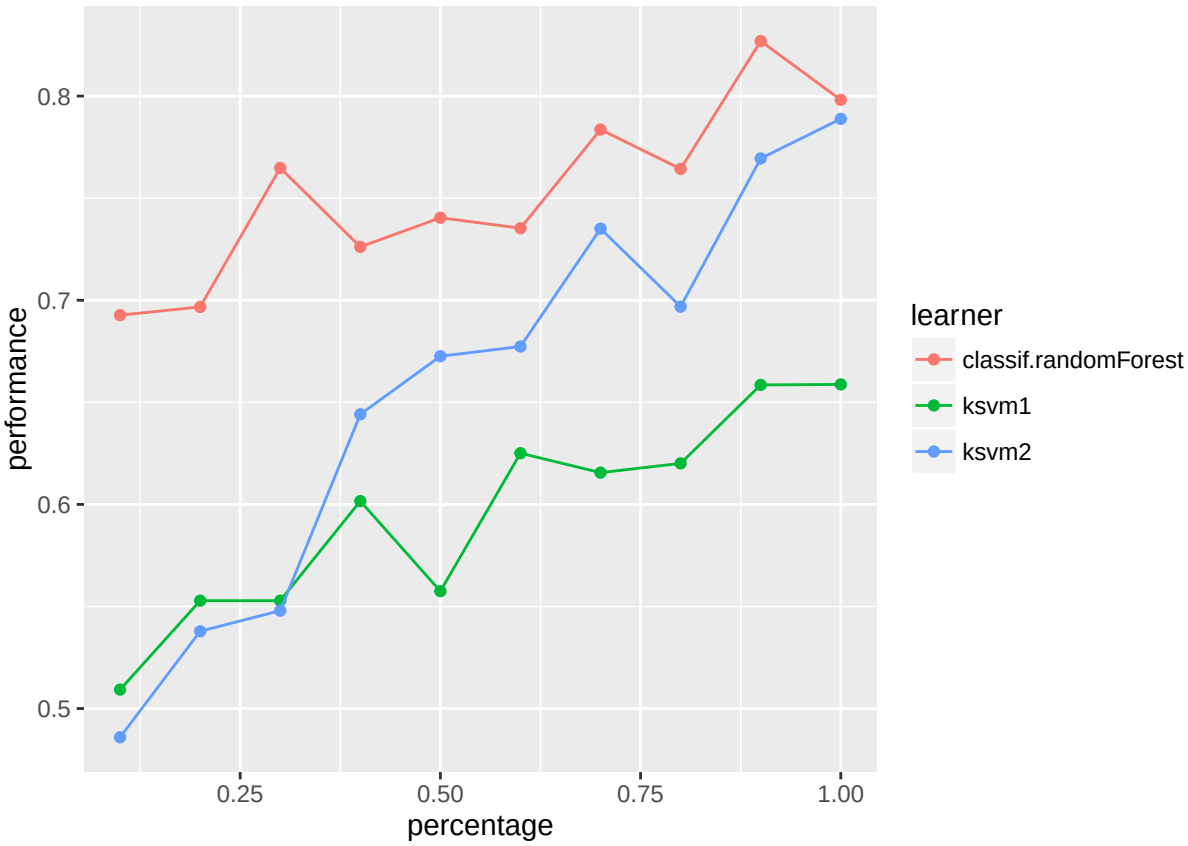



```
# facetted plots created for each learner, measures mapped to color
plotLearningCurve(r, facet = "learner")
```



We can create more complicated learners and pass them on the `generateLearningCurveData` function, as shown in the example below.

```
lrns <- list(
  makeLearner("classif.ksvm", id = "ksvm1", sigma = 0.2, C = 2),
  makeLearner("classif.ksvm", id = "ksvm2", sigma = 0.1, C = 1),
  "classif.randomForest"
)

rin <- makeResampleDesc("CV", iters = 5)

lc <- generateLearningCurveData(learners = lrns, task = sonar.task, percs = seq(.1, 1, by = 0.1), measu

plotLearningCurve(lc)
```



The following is experimental `plotLearningCurveGGVIS(r, interaction = "measure")`

Partial prediction plots

Introduction Machine learning algorithms use available features to make predictions but it is not always obvious how those features are used. Using the mlr package, we can use available functions which aim to estimate the depedence of a learned function on a subset of the feature space. Our response, Y is described as:

$Y = f(X) + \epsilon,$

where $f(X)$ is some unknown function of the feature space, $f(X)$ and ϵ is a random error term independent of X with mean zero. A learner gives some estimate $\hat{f}(X)$ to give $\hat{Y}$ which denotes predictions on Y. Essentially, after the most relevant variables have been identified, we also want to understand the nature of the dependence of the approximation $\hat{f}(X)$ on their joint values. For a feature space that is highly dimensional, $\hat{f}$ may be uninterpretable. Through partial dependence plots, we can more easily visualize how $\hat{f}$ uses the features to make predictions.

Suppose X is our feature space; we partition the space into two sets, X_s and X_c where $X = X_c \cup X_s$ and X_s is the subset of features which are of interest. The partial dependence of f on X_s is:

$f_s = \mathbb{E}_{X_c}[f(X_s, X_c)] = \int f(X_s, X_c)\, dP(X_c);$

this is the averaged value of f when X_s is fixed and X_c varies over its marginal distribution, $dP(X_c)$. Since neither f nor $dP(X_c)$ are known, we need to use an estimator to compute $\hat{f}_s$. Note that each subset of predictors s, has its own partial dependence function, f_s. The estimator function is:

$\hat{f}_s = \frac{1}{N} \sum_{i=1}^{N} \hat{f}(X_s, X_{ci}),$

where $\{_{X_{c1},\cdots,X_{cN}}\}$ represent the different values of X_c that are observed in the training data. The partial dependence functions defined represent the effect of X_s on $f(X)$ after accounting for the (av- erage) effects of the other variables X_c on $f(X)$.

The conditional expectation of an observation i can be estimated using the estimator function above (without the averaging). This has the advantage of discovering features which would otherwise be made unclear through the averaging. The partial prediction function, the individual conditional expectation as well as the partial derivatives of the above with respect to the features are computed. The partial derivatives can be useful in the following context. Suppose the estimate $\hat{f}_{X_s}$ is an additive function because of the lack of interactions between X_s and other features X_c. Then:

$\hat{f}_{X_s,X_c} = g(X_s) + h(X_c)$

and if $\frac{\partial \hat{f}_{X_s}}{\partial X_s} = g'$, we can imply that $\hat{f}$ does not depend on X_c. If there exists variation in estimated partial derivative then this could mean that there is a region of interaction between X_s and X_c in $\hat{f}$.

More information can be found here.

Generating partial predictions To use the function `generatePartialPredictionData`, we need an object from `train`, input data (which can be a `data.frame` or a `task`), and a features character vector for which we want to want to calculate the partial prediction of `\hat{f}`. We need a feature grid for every element of the character vector features passed. The default is a uniform grid of length 10 (this may be set by `gridsize`) from the empirical min to the empirical max (these may also be set through `fmin` and `fmax`). Resampling is also available for the feature data (in the `features` vector) through the `bootstrap` or `resample` methods.

```
lrn.classif <- makeLearner("classif.ksvm", predict.type = "prob")
fit.classif <- train(lrn.classif, iris.task)
pd <- generatePartialPredictionData(fit.classif, input = iris.task, features = "Petal.Width")
pd$data[1:5, ]
```

```
##    Class Probability Petal.Width
## 1 setosa  0.11358157    2.500000
## 2 setosa  0.09956561    2.233333
## 3 setosa  0.09525302    1.966667
## 4 setosa  0.09953924    1.700000
## 5 setosa  0.12232060    1.433333
```

```
pd$data[11:15, ]
```

```
##         Class Probability Petal.Width
## 11 versicolor   0.1384642    2.500000
## 12 versicolor   0.1351334    2.233333
## 13 versicolor   0.1825441    1.966667
## 14 versicolor   0.3752083    1.700000
## 15 versicolor   0.5717253    1.433333
```

Suppose now we pass two features in the `features` vector. If `interaction = FALSE` (default) then X_s is assumed to be unidimensional and partial predictions are generated for each feature separately (`NA` is shown for the feature that has not been used for the prediction). If `interaction = TRUE`, then the individual feature grids are combined using the Cartesian product and the estimator produces a partial prediction for every combination of unique feature values.

```
# interaction = FALSE
pd.lst <- generatePartialPredictionData(fit.classif, iris.task, c("Petal.Width", "Petal.Length"), intera
head(pd.lst$data)
```

```
##    Class Probability Petal.Width Petal.Length
## 1 setosa  0.11358157    2.500000           NA
## 2 setosa  0.09956561    2.233333           NA
## 3 setosa  0.09525302    1.966667           NA
## 4 setosa  0.09953924    1.700000           NA
## 5 setosa  0.12232060    1.433333           NA
## 6 setosa  0.18778786    1.166667           NA
```

```
tail(pd.lst$data)
```

```
##        Class Probability Petal.Width Petal.Length
## 55 virginica   0.3740572          NA     4.277778
## 56 virginica   0.2780015          NA     3.622222
## 57 virginica   0.2160617          NA     2.966667
## 58 virginica   0.2329795          NA     2.311111
## 59 virginica   0.2825701          NA     1.655556
## 60 virginica   0.3105751          NA     1.000000
```

```
# interaction = TRUE
pd.int <- generatePartialPredictionData(fit.classif, iris.task, c("Petal.Width", "Petal.Length"), intera
pd.int
```

```
## PartialPredictionData
## Task: iris-example
## Features: Petal.Width, Petal.Length
## Target: setosa, versicolor, virginica
## Derivative: FALSE
## Interaction: TRUE
## Individual: FALSE
##    Class Probability Petal.Width Petal.Length
## 1 setosa   0.1414267    2.500000          6.9
## 2 setosa   0.1267246    2.233333          6.9
## 3 setosa   0.1218138    1.966667          6.9
## 4 setosa   0.1264862    1.700000          6.9
## 5 setosa   0.1413398    1.433333          6.9
## 6 setosa   0.1652182    1.166667          6.9
```

The object `pd.int` creates a feature grid of 10×10 for every combination of feature values and estimates probabilities corresponding to each unique combination for each class in the target variable. From the definition of the estimator function, the mean prediction is returned which, here, are represented by the mean class probabilites. It is possible to return other summaries of the predictions using the `fun` argument. We show this using a `regression` task.

```
lrn.regr <- makeLearner("regr.ksvm")
fit.regr <- train(lrn.regr, bh.task)
pd.regr <- generatePartialPredictionData(fit.regr, bh.task, "lstat", fun = median)
pd.regr
```

```
## PartialPredictionData
## Task: BostonHousing-example
## Features: lstat
## Target: medv
## Derivative: FALSE
## Interaction: FALSE
## Individual: FALSE
##       medv    lstat
## 1 18.79492 37.97000
## 2 18.56123 33.94333
## 3 18.56444 29.91667
## 4 18.67220 25.89000
## 5 19.17552 21.86333
## 6 19.59268 17.83667
```

Note that the function argument must return a numeric vector of length 1 or 3.

```
pd.ci <- generatePartialPredictionData(fit.regr, bh.task, "lstat", fun = function(x) quantile(x, c(.25,
pd.ci
```

```
## PartialPredictionData
## Task: BostonHousing-example
## Features: lstat
## Target: medv
## Derivative: FALSE
## Interaction: FALSE
## Individual: FALSE
##       medv    lstat    lower    upper
## 1 18.79492 37.97000 15.96955 20.81787
## 2 18.56123 33.94333 14.78636 20.79412
## 3 18.56444 29.91667 14.06848 20.96023
## 4 18.67220 25.89000 14.10915 21.45855
## 5 19.17552 21.86333 14.99129 22.16446
## 6 19.59268 17.83667 16.51673 22.93814
```

```
pd.classif <- generatePartialPredictionData(fit.classif, iris.task, "Petal.Length", fun = median)
pd.classif
```

```
## PartialPredictionData
## Task: iris-example
## Features: Petal.Length
## Target: setosa, versicolor, virginica
## Derivative: FALSE
## Interaction: FALSE
## Individual: FALSE
##    Class Probability Petal.Length
## 1 setosa  0.12438690     6.900000
## 2 setosa  0.06566829     6.244444
## 3 setosa  0.03347497     5.588889
## 4 setosa  0.02296467     4.933333
## 5 setosa  0.03181992     4.277778
## 6 setosa  0.07856785     3.622222
```

Using the argument `individual = TRUE`, we obtain te conditional expectation of the learned function at observation i. This is shown with a regression task below. The resulting object contains N predictions for each point in the feature grid (so, for the `bh.task`, we have 506 obs times the 10 points in the gird therefore 5060 predictions).

```
pd.ind.regr <- generatePartialPredictionData(fit.regr, bh.task, "lstat", individual = TRUE)
pd.ind.regr
```

```
## PartialPredictionData
## Task: BostonHousing-example
## Features: lstat
## Target: medv
## Derivative: FALSE
## Interaction: FALSE
## Individual: TRUE
## Predictions centered: FALSE
##       medv    lstat idx
## 1 20.34921 37.97000   1
## 2 20.34549 33.94333   1
## 3 20.56191 29.91667   1
## 4 20.97720 25.89000   1
## 5 21.55322 21.86333   1
## 6 22.27515 17.83667   1
```

The `idx` column in the `$data` element gives the index of the observation. In classification tasks, `idx` gives the index corresponding to the observation and the class target label.

```
pd.ind.classif <- generatePartialPredictionData(fit.classif, iris.task, "Petal.Length", individual = TRU
pd.ind.classif
```

```
## PartialPredictionData
## Task: iris-example
## Features: Petal.Length
## Target: setosa, versicolor, virginica
## Derivative: FALSE
## Interaction: FALSE
## Individual: TRUE
## Predictions centered: FALSE
##    Class Probability Petal.Length      idx
## 1 setosa   0.2374042          6.9 1.setosa
## 2 setosa   0.2364840          6.9 2.setosa
## 3 setosa   0.2373213          6.9 3.setosa
## 4 setosa   0.2372784          6.9 4.setosa
## 5 setosa   0.2375749          6.9 5.setosa
## 6 setosa   0.2375047          6.9 6.setosa
```

If `individual = TRUE`, through the argument `center`, we can pass a fixed value for each feature in the `features` vector. This fixed value is used to center the data by subtracting it from each individual prediction made across the prediction grid.

```
iris <- getTaskData(iris.task)
pd.ind.classif <- generatePartialPredictionData(fit.classif, iris.task, "Petal.Length", individual = TRU
```

Partial derivatives can also be computed for individual and aggregate partial predictions. This is restricted to one feature at a time.

```
pd.regr.der <- generatePartialPredictionData(fit.regr, bh.task, "lstat", derivative = TRUE)
head(pd.regr.der$data)
```

```
##          medv    lstat
## 1  0.16062360 37.97000
## 2  0.08256021 33.94333
## 3 -0.02705605 29.91667
## 4 -0.15298603 25.89000
## 5 -0.27724716 21.86333
## 6 -0.37997720 17.83667
```

```
pd.regr.der.ind <- generatePartialPredictionData(fit.regr, bh.task, "lstat", individual = TRUE, derivati
head(pd.regr.der.ind$data)
```

```
##          medv    lstat idx
## 1  0.02708051 37.97000   1
## 2 -0.02641034 33.94333   1
## 3 -0.08005067 29.91667   1
## 4 -0.12444486 25.89000   1
## 5 -0.16101276 21.86333   1
## 6 -0.19830894 17.83667   1
```

```
pd.classif.der <- generatePartialPredictionData(fit.classif, iris.task, "Petal.Width", derivative = TRU
head(pd.classif.der$data)
```

```
##    Class  Probability Petal.Width
## 1 setosa  0.073533032    2.500000
## 2 setosa  0.033271496    2.233333
## 3 setosa  0.000115817    1.966667
## 4 setosa -0.037954111    1.700000
## 5 setosa -0.149220904    1.433333
## 6 setosa -0.349755594    1.166667
```

```
pd.classif.der.ind <- generatePartialPredictionData(fit.classif, iris.task, "Petal.Width", derivative =
head(pd.classif.der.ind$data)
```

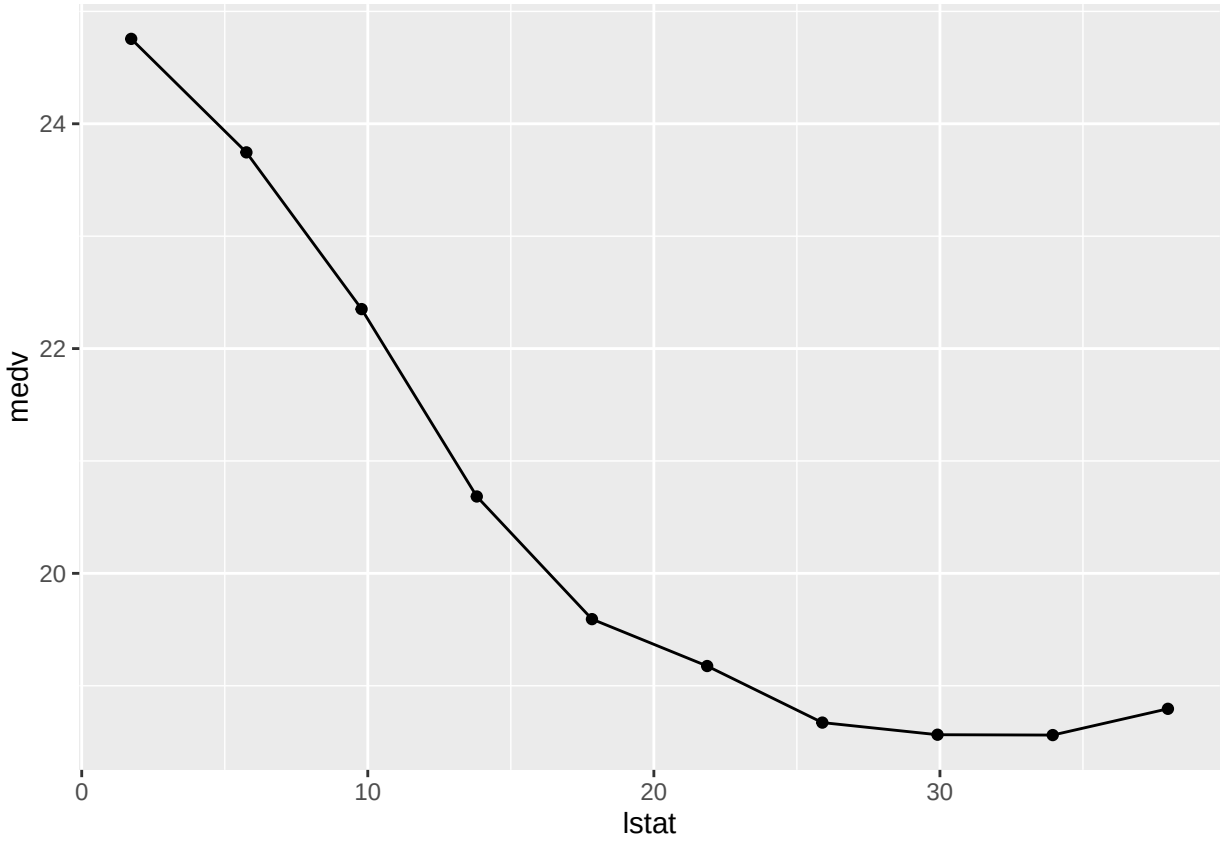
```
##    Class   Probability Petal.Width      idx
## 1 setosa -0.0001653354         2.5 1.setosa
## 2 setosa  0.0075935447         2.5 2.setosa
## 3 setosa  0.0009903833         2.5 3.setosa
## 4 setosa  0.0033948994         2.5 4.setosa
## 5 setosa -0.0011711145         2.5 5.setosa
## 6 setosa -0.0001492683         2.5 6.setosa
```

Plotting partial predictions The resulting object from `generatePartialPredictionData` can be visualized with the functions `plotPartialPrediction` and `plotPartialPredictionGGVIS`.

Regression: single feature

The result is a line plot showing the dependence on the target on the chosen feature. The markers on the line correspond to the points in the prediction plot.

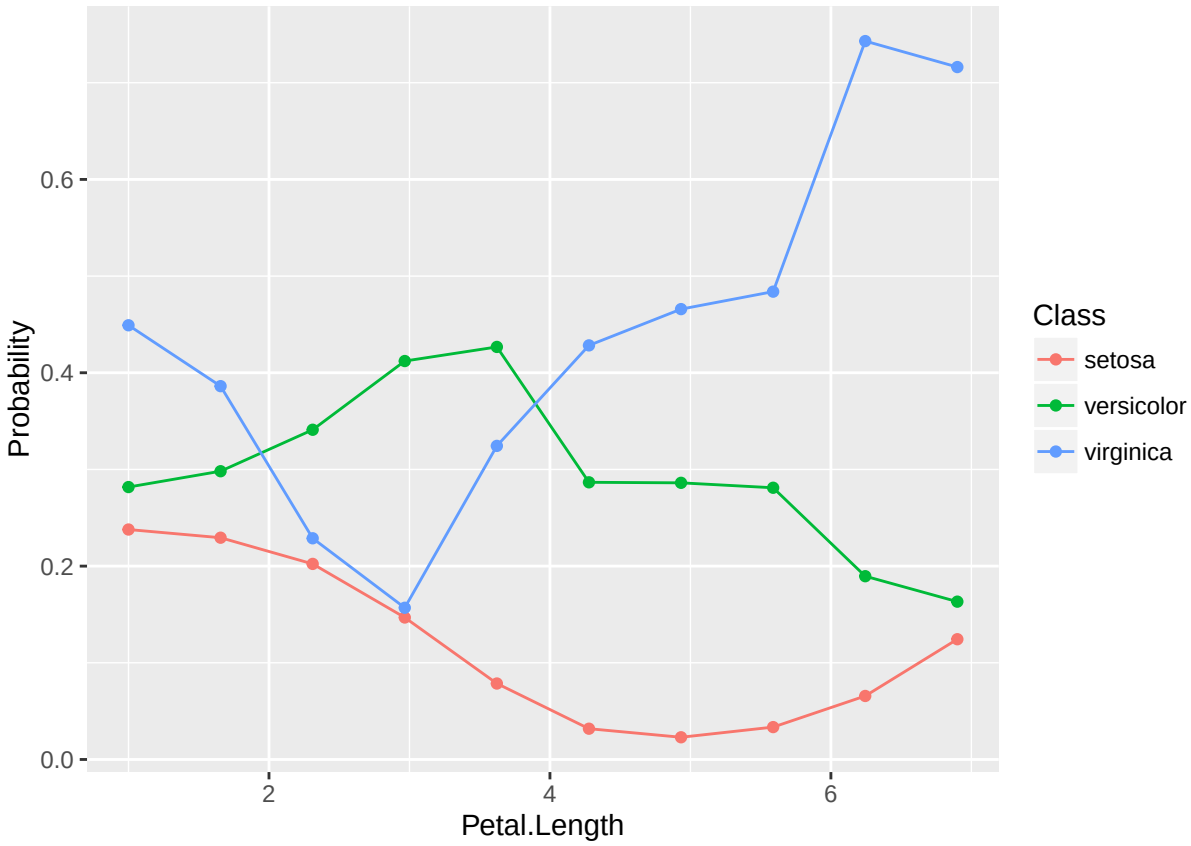
```
plotPartialPrediction(pd.regr)
```



Classification: single feature

In classification, we have a line plot of the class probability against the selected feature for each class in the target variable.

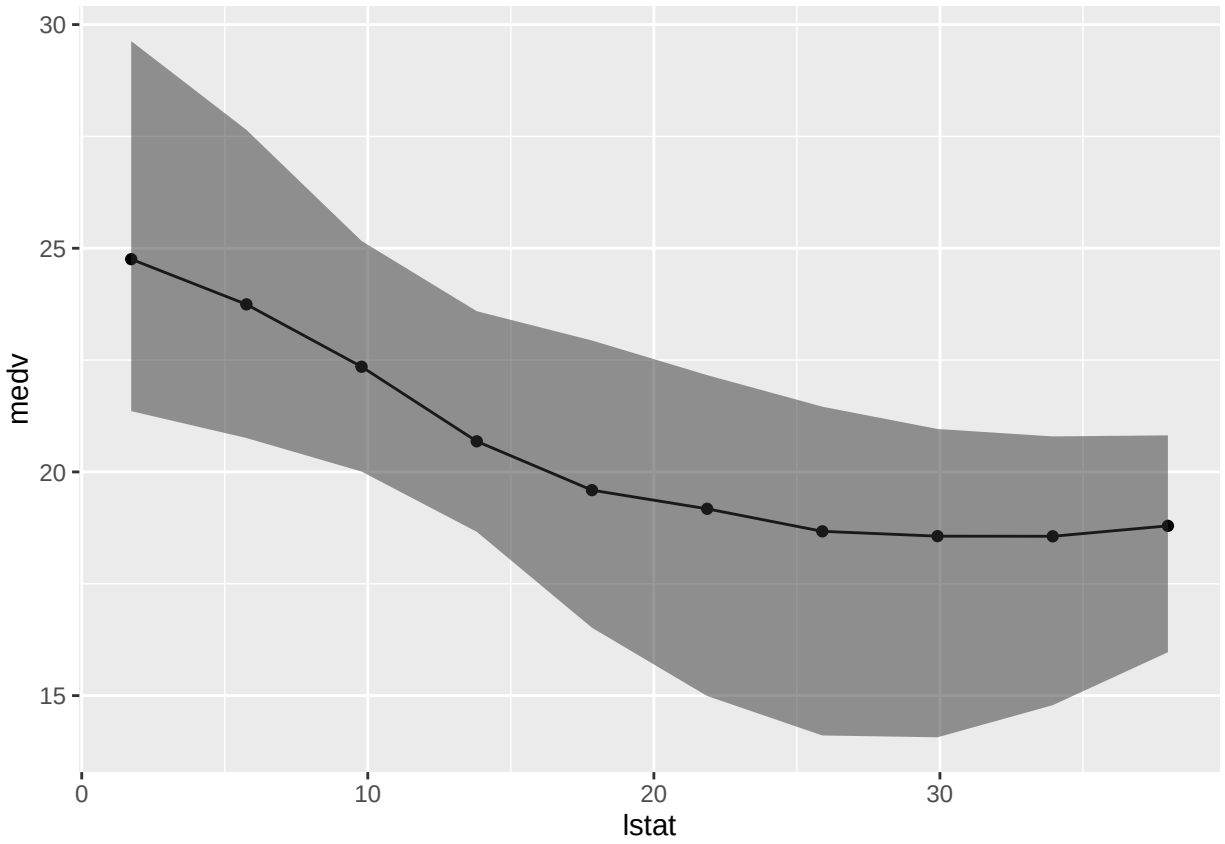
```
plotPartialPrediction(pd.classif)
```



Regression: single feature, `fun` enabled

With the `fun` argument used, the bounds are automatically shown with a grey region. This is also true when the learner has `predict.type = "se"`.

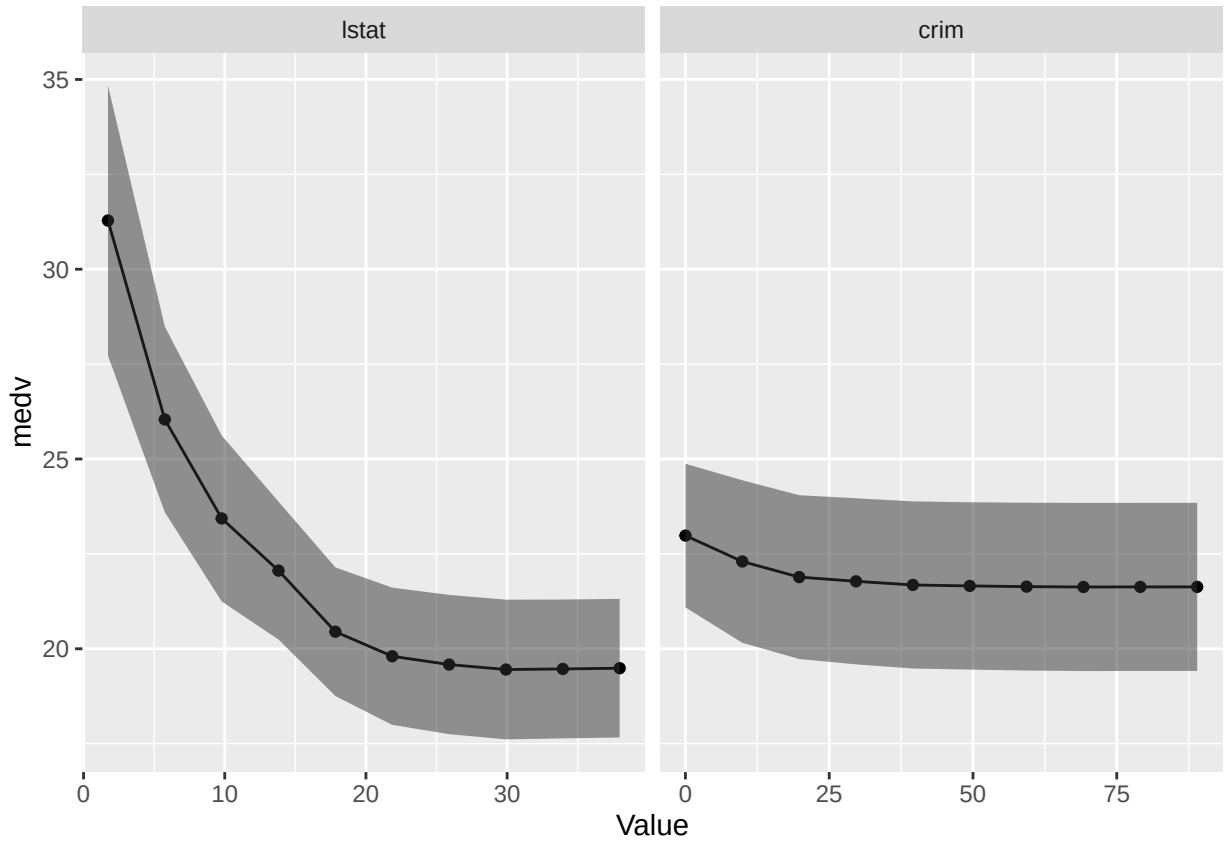
```
plotPartialPrediction(pd.ci)
```



Regression: multiple features, `fun` enabled

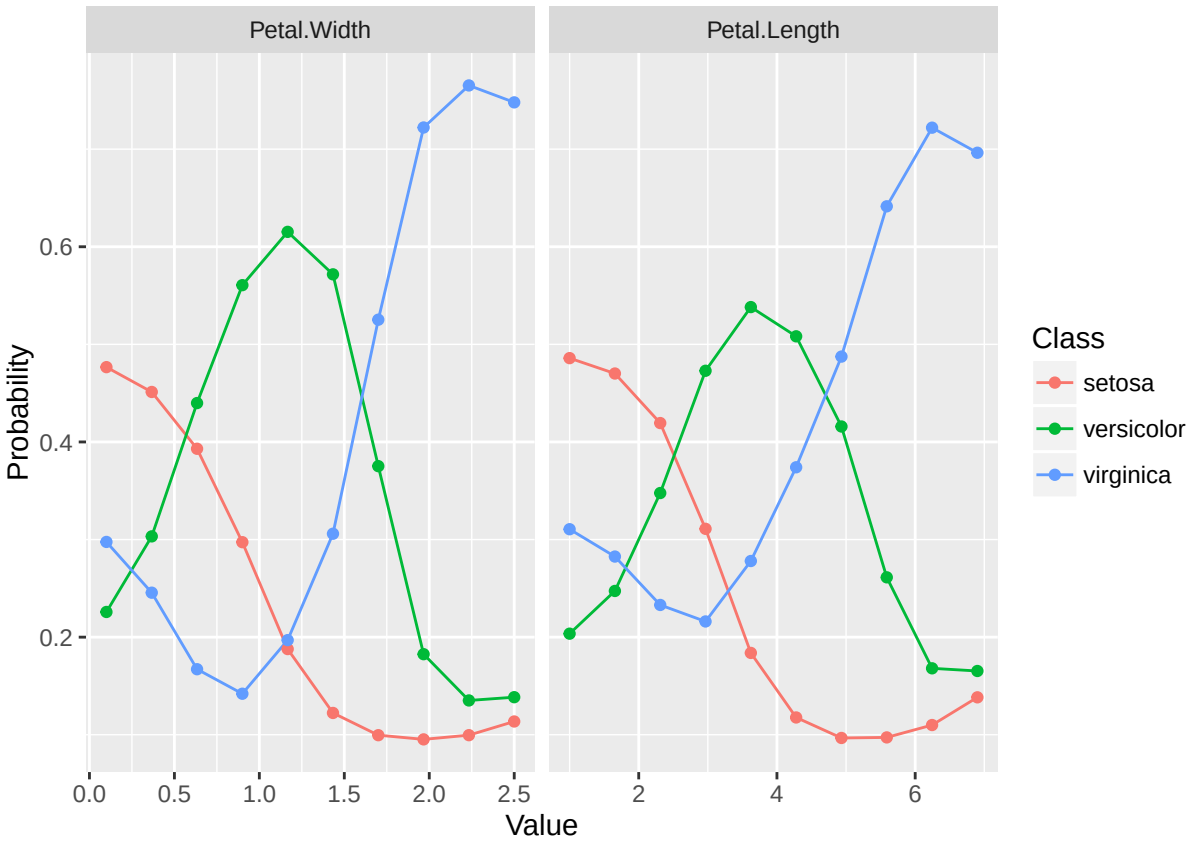
When multiple features are passed in the features vector but with `interaction = FALSE`, we obtain facetted plots to show the dependence of target on each feature.

```
fit.se <- train(makeLearner("regr.randomForest", predict.type = "se"), bh.task)
pd.se <- generatePartialPredictionData(fit.se, bh.task, c("lstat", "crim"))
plotPartialPrediction(pd.se)
```



Classification: multiple features

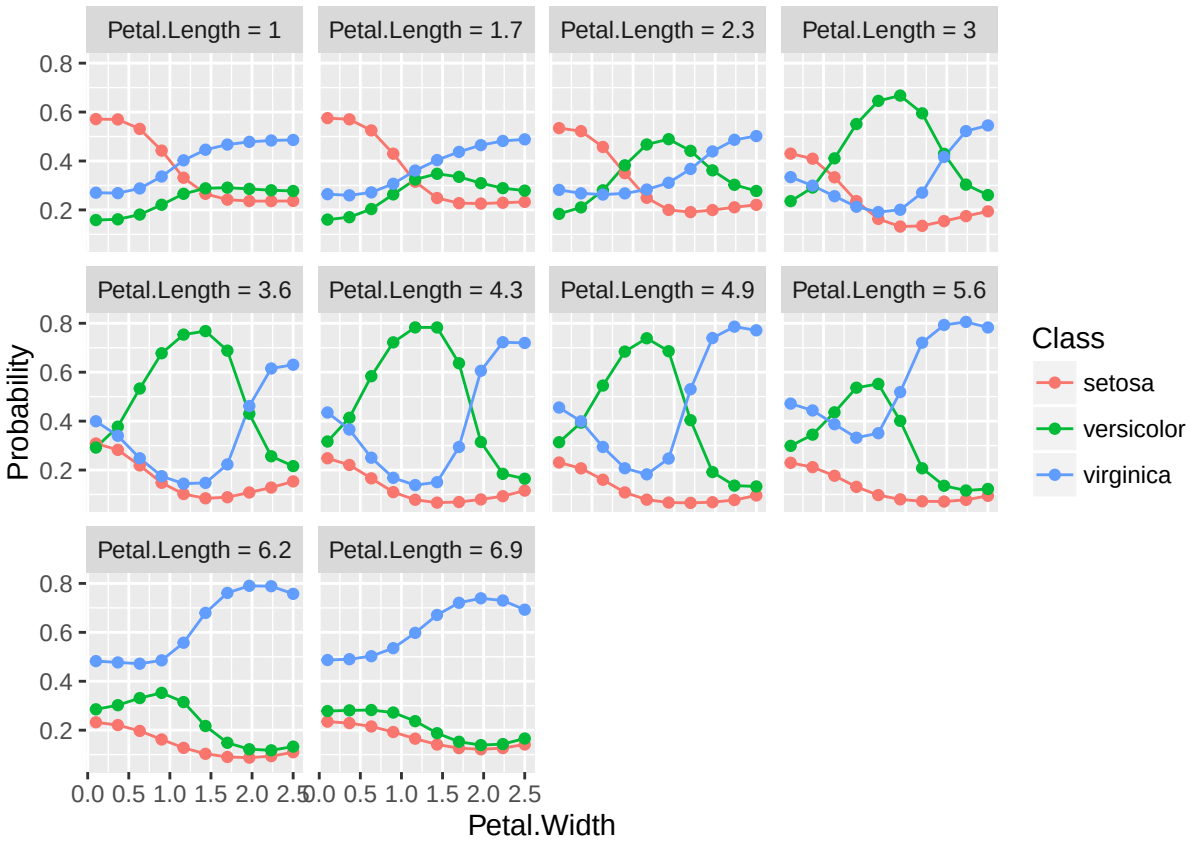
```
plotPartialPrediction(pd.lst)
```



Classification example: `interaction = TRUE`

When `interaction = TRUE`, one variable is used for facetting and for each value in the grid for the chosen variable, we get a partial prediction plot against the variable not used in facetting.

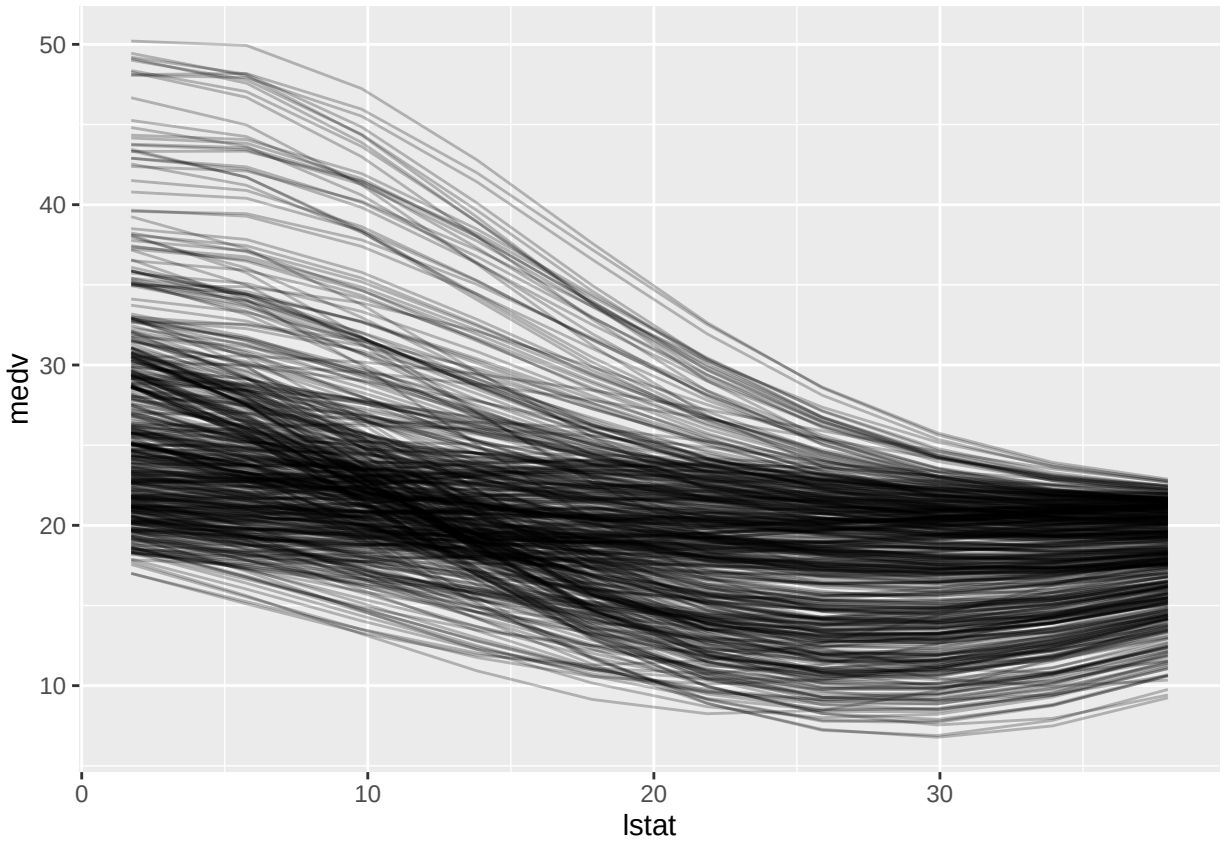
```
plotPartialPrediction(pd.int, facet = "Petal.Length")
```



Individual predictions

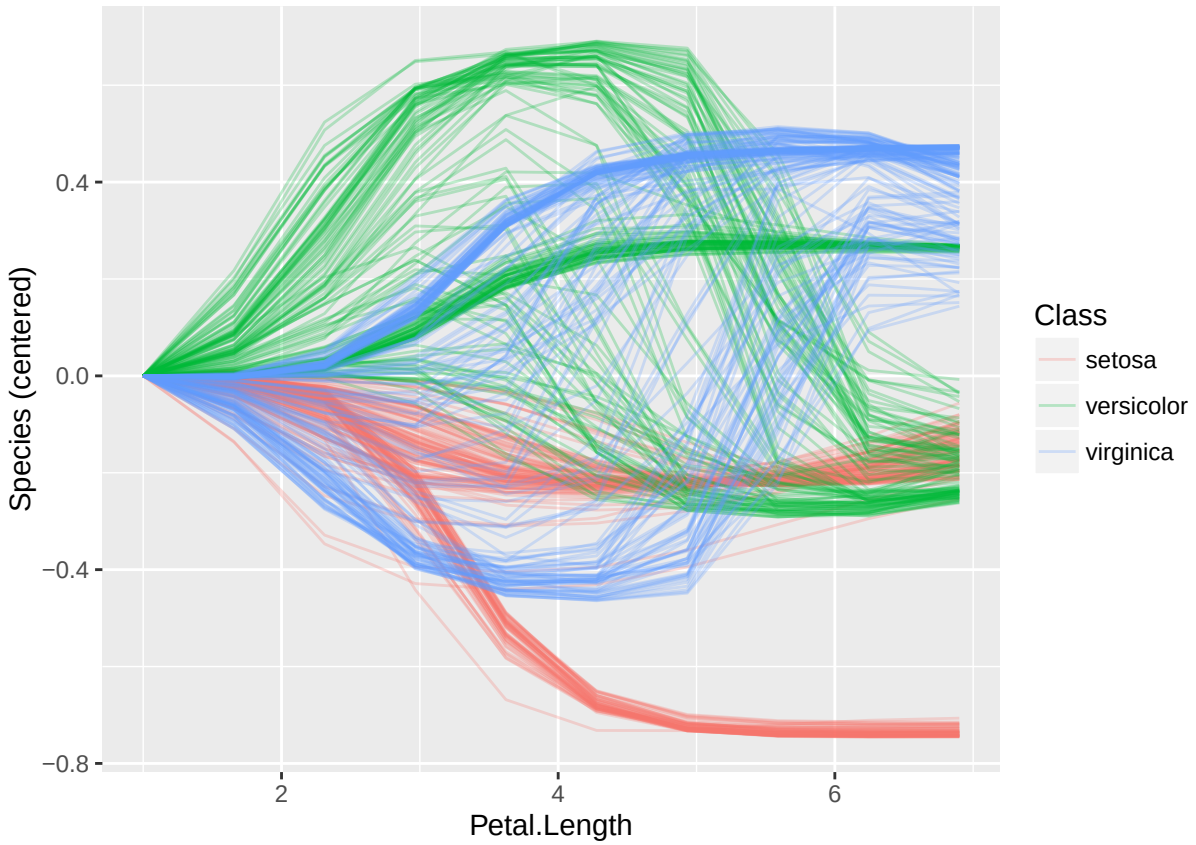
With `individual = TRUE`, we obtain all individual conditional expectation curves.

```
plotPartialPrediction(pd.ind.regr)
```



When the data is centered (by some given fixed value of X_s), we obtain a fixed intercept, as shown below.

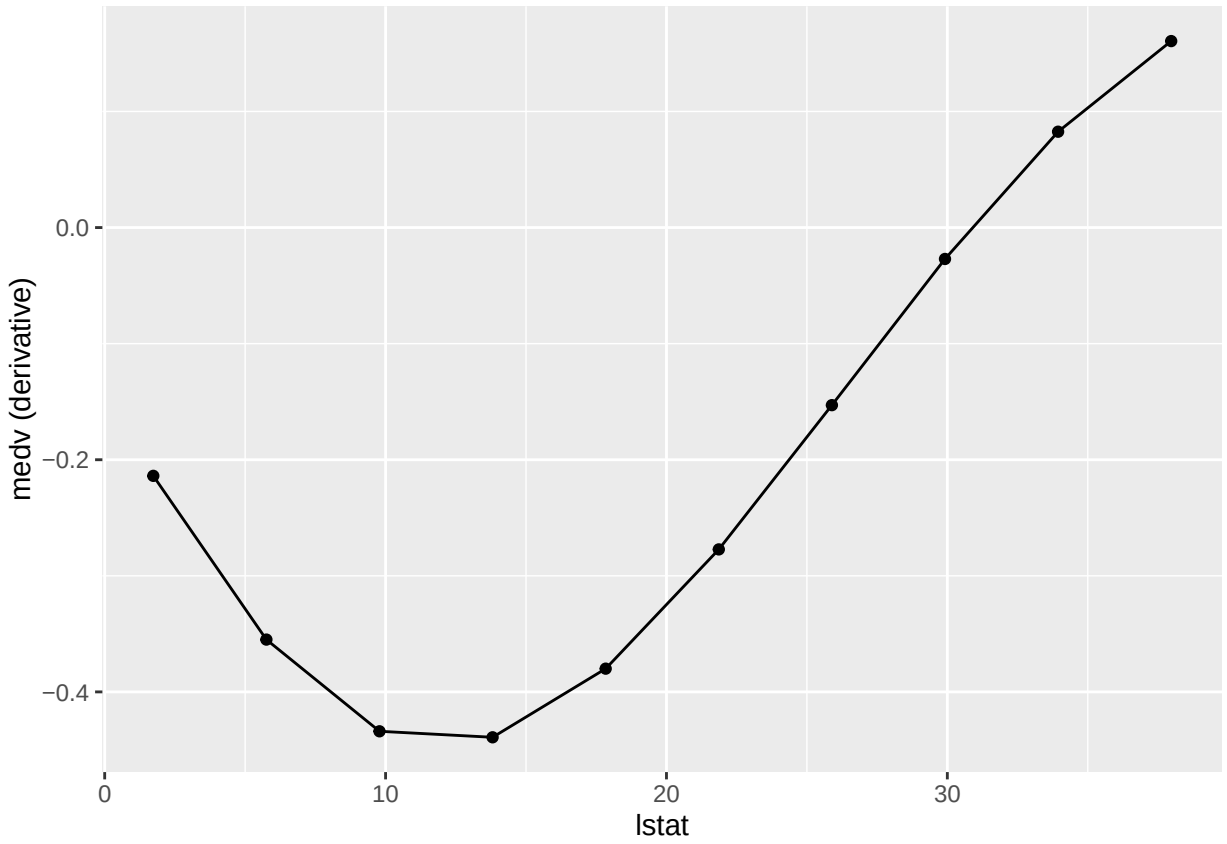
```
plotPartialPrediction(pd.ind.classif)
```



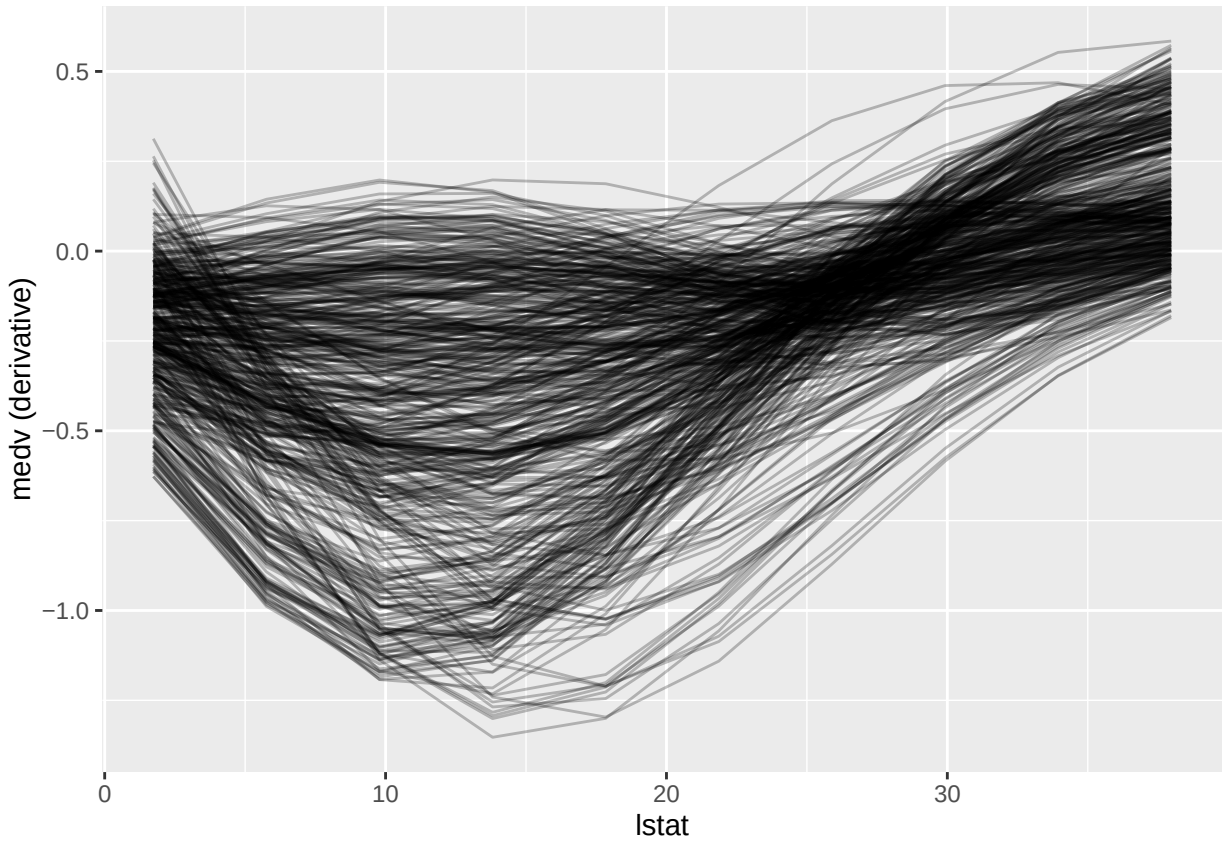
Plotting derivatives

This works the same way: we use the returned object when `derivative = TRUE` to return the derivatives. What we are looking for here is variation in the derivative with the selected feature. If no variation exists, the result would suggest that the model is additive in the selected feature and no interaction occurs with other predictors.

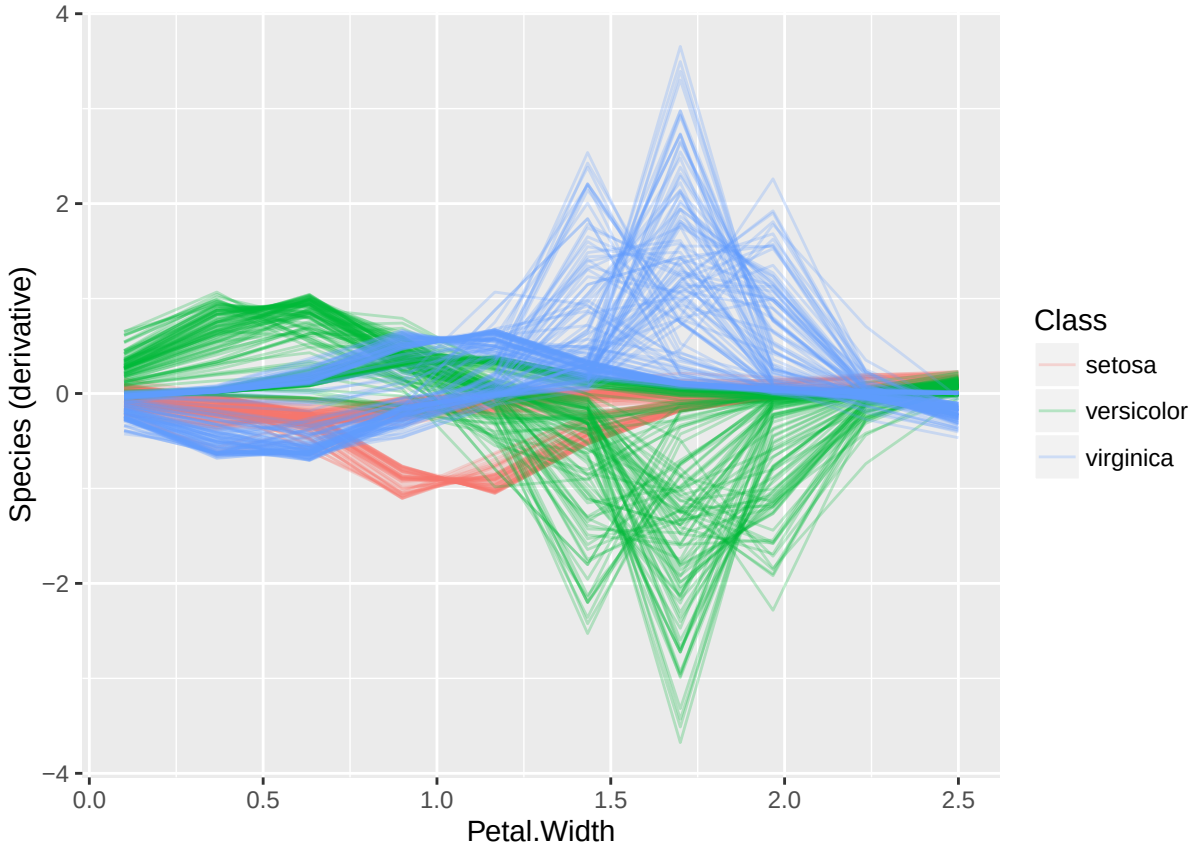
```
plotPartialPrediction(pd.regr.der)
```




```
plotPartialPrediction(pd.regr.der.ind)
```




```
plotPartialPrediction(pd.classif.der.ind)
```



These plots can identify regions where there may be interactions with the selected features, which can then further be investigated.

Classifier calibration

A calibrated classifiers is one where the predicted probability of a class closely matches the frequency of that class. The function `generateCalibrationData` takes as an input an object or a list of objects returned from `predict`, `resample`, or an object returned from `benchmark`. The objects must be from a binary or multiclass classification task with learners that support `predict.type = "prob"`. The function groups data points with similar predicted probabilites for each class. The object has elements `$data`, `$proportion`, and `$task`. The `$proportion` element is a `data.frame` with `Learner`, `bin`, `Class`, and `Proportion` columns. The `Proportion` column gives the proportion of observations from class `Class` for each bin among all observations with posterior probabilities of that class. For example, for observations that are predicted to have class "A" with probability (0,0.1], what is the proportion of said observations which have class "A"? The bins are calculated according to the `breaks` or `groups` argument (default is `breaks = "Sturges"`).

```
lrn <- makeLearner("classif.rpart", predict.type = "prob")
mod <- train(lrn, task = sonar.task)
pred <- predict(mod, task = sonar.task)
cal <- generateCalibrationData(pred)
cal$proportion
```

```
##      Learner       bin Class Proportion
## 1 prediction   [0,0.1]     M  0.0000000
## 2 prediction (0.1,0.2]     M  0.1060606
## 3 prediction (0.2,0.3]     M  0.2727273
```

```
## 4 prediction (0.4,0.5]     M  0.4615385
## 5 prediction (0.5,0.6]     M  0.0000000
## 6 prediction (0.7,0.8]     M  0.7333333
## 7 prediction (0.8,0.9]     M  0.0000000
## 8 prediction   (0.9,1]     M  0.9333333
```

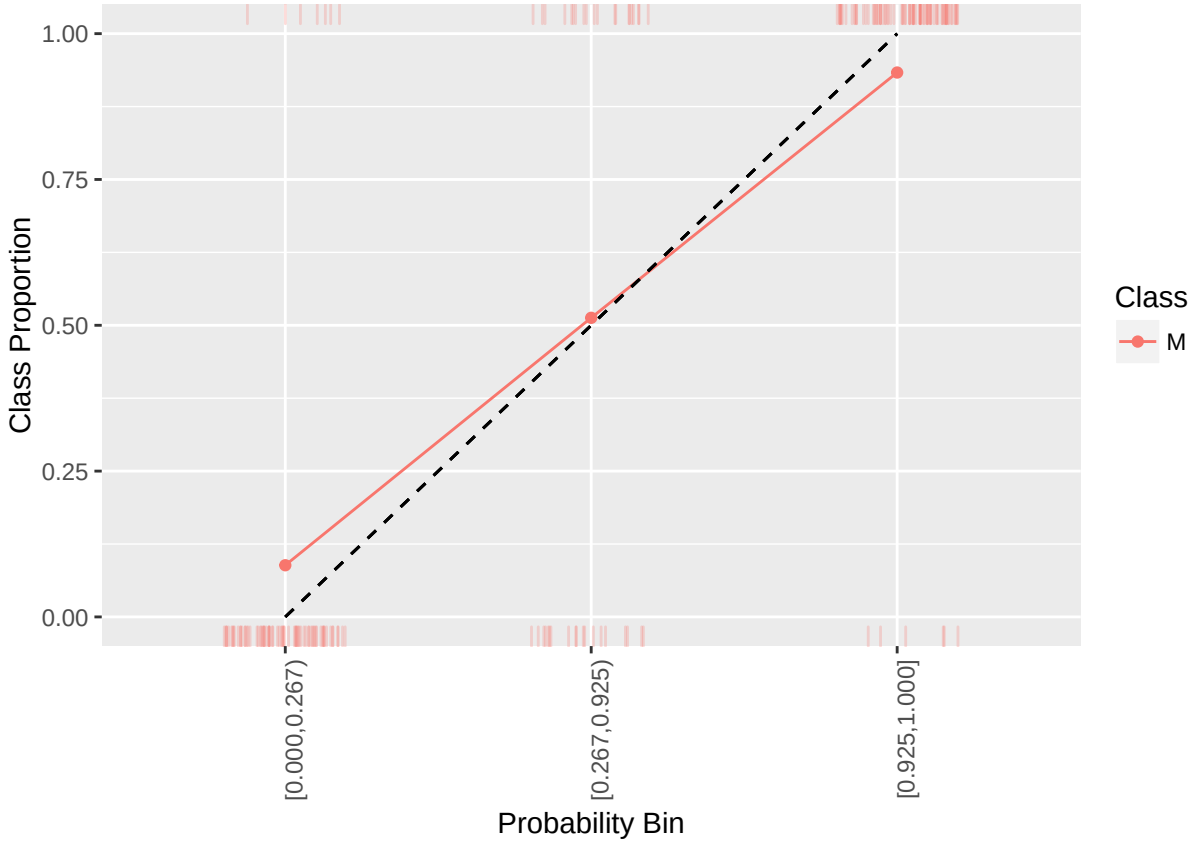
With `groups = n` where `n` is the number of groups, `cut2` is used to create bins with approximately equal number of observations in each group.

```
cal <- generateCalibrationData(pred, groups = 3)
cal$proportion
```

```
##      Learner           bin Class Proportion
## 1 prediction [0.000,0.267)     M 0.08860759
## 2 prediction [0.267,0.925)     M 0.51282051
## 3 prediction [0.925,1.000]     M 0.93333333
```

The function `plotCalibration` uses the object returned from `generateCalibrationData` to plot the proportion of each class in each bin against the bins.

```
plotCalibration(cal)
```

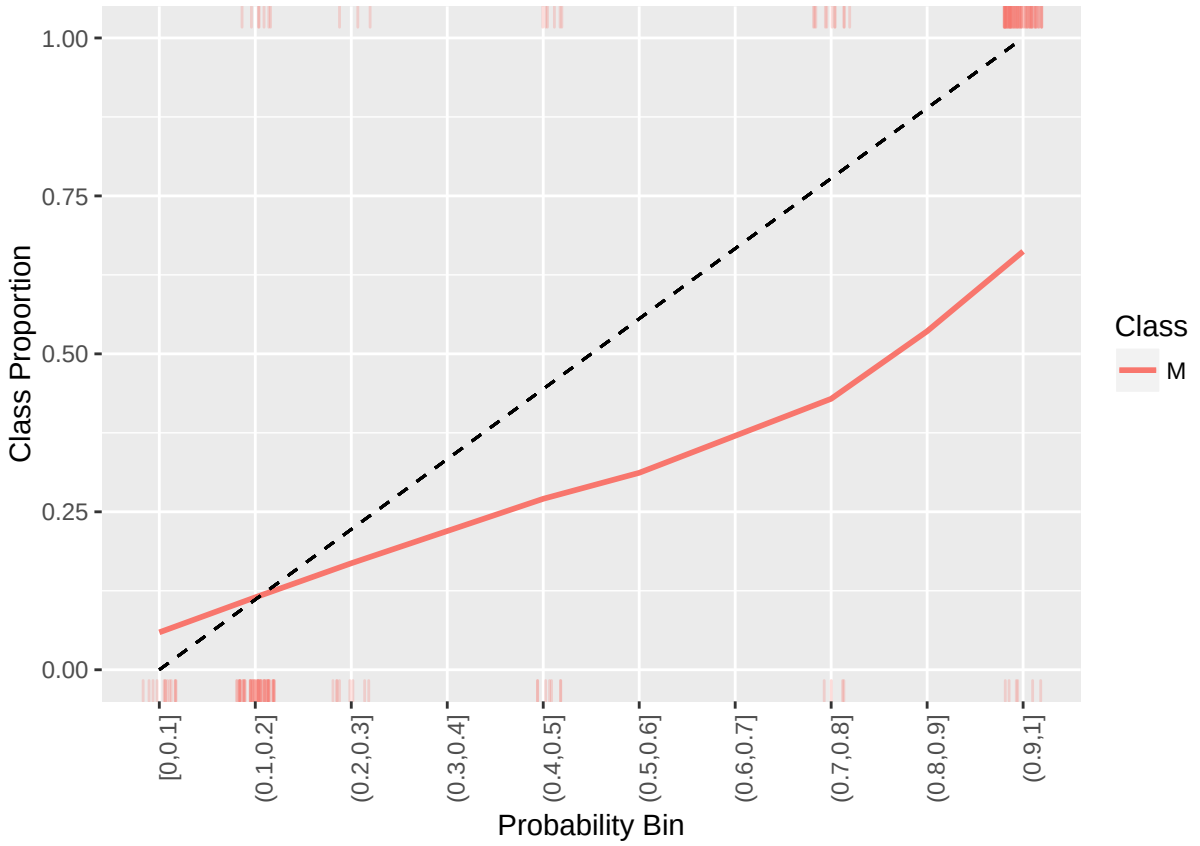


The function also plots a reference line representing the perfect classifier. A rug plot (this draws ticks for each value) is also shown at the top and bottom. At the top, the ticks represent the "positive" class in each bin while at the bottom, the ticks represent the "negative" class. A perfect classifier would have all the positive

classes at the top right (correct class predicted with high probability) and all the negative classes at the bottom left.

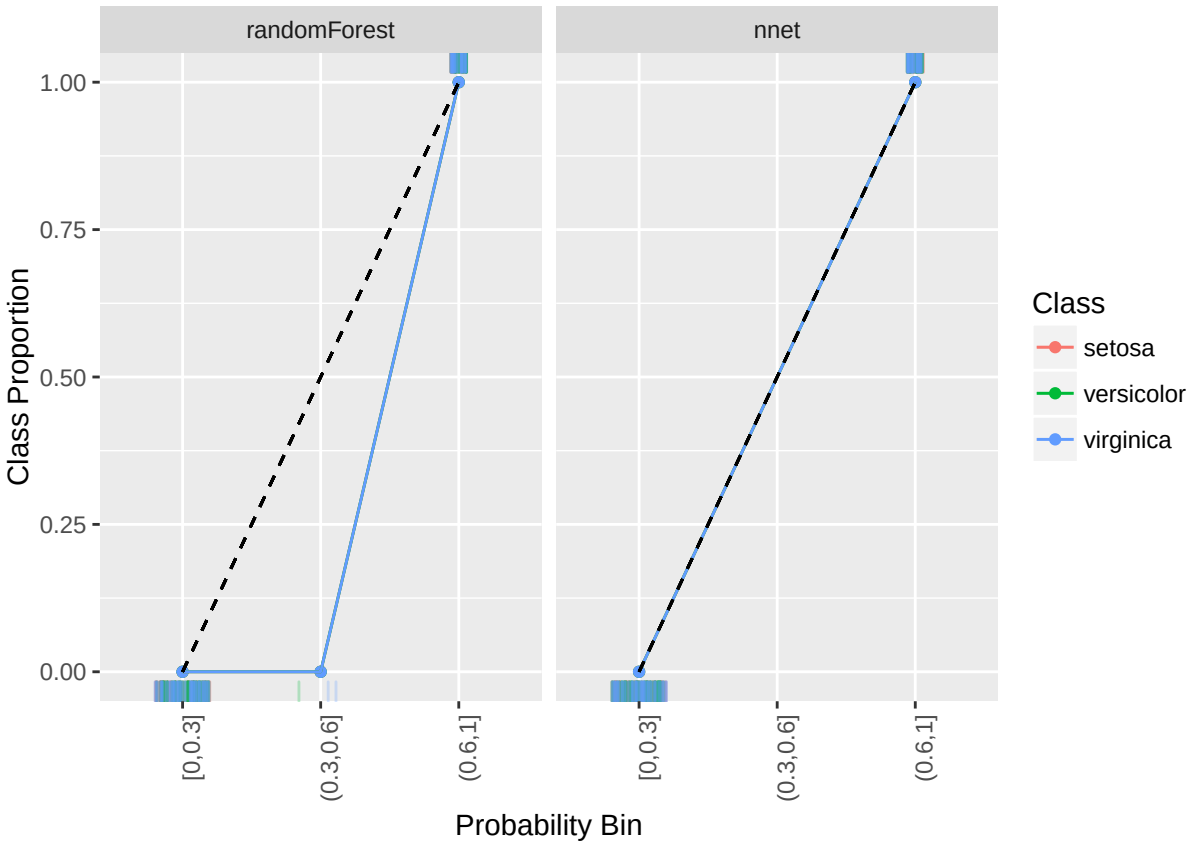
In the plot above, we only had the three bins but, in situations where we have multiple bins, because of the discretization of the probabilities, it might be best to use `smooth = TRUE` which replaces the estimated proportions with a loess smoother.

```
cal <- generateCalibrationData(pred)
plotCalibration(cal, smooth = TRUE)
```



While the plots were generated for a binary classification task, they do generalize to mutliclass classification examples. We show this with our favorite `iris.task`.

```
lrns <- list(
  makeLearner("classif.randomForest", predict.type = "prob"),
  makeLearner("classif.nnet", predict.type = "prob", trace = FALSE)
)

mod <- lapply(lrns, train, task = iris.task)
pred <- lapply(mod, predict, task = iris.task)
names(pred) = c("randomForest", "nnet")
cal <- generateCalibrationData(pred, breaks = c(0, .3, .6, 1))
plotCalibration(cal)
```


randomForest
nnet
1.00
0.75
0.50
0.25
0.00
Class Proportion
[0,0.3]
(0.3,0.6]
(0.6,1]
Probability Bin
Class
setosa
versicolor
virginica