mlr-tutorial

Julia Schiffner, Bernd Bischl, Michel Lang, Jakob Richter, Zachary M. Jones, Philipp Probst, Florian Pfisterer, Mason Gallo, Dominik Kirchhoff, Tobias Kühn, Janek Thomas, Kira Engelhardt, Teodora Pandeva, Gunnar König, Lars Kotthoff

Contents

1 Basics

2 Learning Tasks

Learning tasks encapsulate the data set and further relevant information about a machine learning problem, for example the name of the target variable for supervised problems.

2.1 Task types and creation

The tasks are organized in a hierarchy, with the generic `Task()` at the top. The following tasks can be instantiated and all inherit from the virtual superclass `Task()`:

- `RegrTask()` for regression problems,
- `ClassifTask()` for binary and multi-class classification problems (`vignette("cost_sensitive_classif")`) with class-dependent costs can be handled as well),
- `SurvTask()` for survival analysis,
- `ClusterTask()` for cluster analysis,
- `MultilabelTask()` for multilabel classification problems,
- `CostSensTask()` for general `vignette("cost_sensitive_classif")` (with example-specific costs).

To create a task, just call `make<TaskType>`, e.g., `makeClassifTask()`. All tasks require an identifier (argument `id`) and a `base::data.frame()` (argument `data`). If no ID is provided it is automatically generated using the variable name of the data. The ID will be later used to name results, for example of `vignette("benchmark_experiments")`, and to annotate plots. Depending on the nature of the learning problem, additional arguments may be required and are discussed in the following sections.

2.1.1 Regression

For supervised learning like regression (as well as classification and survival analysis) we, in addition to `data`, have to specify the name of the `target` variable.

```
data(BostonHousing, package = "mlbench")
regr.task = makeRegrTask(id = "bh", data = BostonHousing, target = "medv")
regr.task
```

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

As you can see, the `Task()` records the type of the learning problem and basic information about the data set, e.g., the types of the features (`base::numeric()` vectors, `base::factors()` or ordered factors), the number of observations, or whether missing values are present.

Creating tasks for classification and survival analysis follows the same scheme, the data type of the target variables included in `data` is simply different. For each of these learning problems some specifics are described below.

2.1.2 Classification

For classification the target column has to be a `factor`.

In the following example we define a classification task for the `mlbench::BreastCancer()` data set and exclude the variable `Id` from all furthercmodel fitting and evaluation.

```
data(BreastCancer, package = "mlbench")
df = BreastCancer
df$Id = NULL
classif.task = makeClassifTask(id = "BreastCancer", data = df, target = "Class")
classif.task
```

```
## Supervised task: BreastCancer
## Type: classif
## Target: Class
## Observations: 699
## Features:
##    numerics     factors     ordered functionals
##           0           4           5           0
## Missings: TRUE
## Has weights: FALSE
## Has blocking: FALSE
## Is spatial: FALSE
## Classes: 2
##    benign malignant
##       458       241
## Positive class: benign
```

In binary classification the two classes are usually referred to as *positive* andc*negative* class with the positive class being the category of greater interest. This is relevant for many `vignette("performance")` like the *true positive rate* or `vignette("roc_analysis")`. Moreover, `mlr`, where possible, permits to set options (like the `setThreshold()` or `makeWeightedClassesWrapper()`) and returns and plots results (like class posterior probabilities) for the positive class only.

`makeClassifTask()` by default selects the first factor level of the target variable as the positive class, in the above example `benign`. Class `malignant` can be manually selected as follows:

```
classif.task = makeClassifTask(id = "BreastCancer", data = df, target = "Class", positive = "malignant")
```

2.1.3 Survival analysis

Survival tasks use two target columns. For left and right censored problems these consist of the survival time and a binary event indicator. For interval censored data the two target columns must be specified in the `"interval2"` format (see `survival::Surv()`).

```
data(lung, package = "survival")
lung$status = (lung$status == 2) # convert to logical
surv.task = makeSurvTask(data = lung, target = c("time", "status"))
surv.task
```

```
## Supervised task: lung
## Type: surv
## Target: time,status
## Events: 165
## Observations: 228
## Features:
##    numerics     factors     ordered functionals
##           8           0           0           0
## Missings: TRUE
## Has weights: FALSE
## Has blocking: FALSE
## Is spatial: FALSE
```

The type of censoring can be specified via the argument `censoring`, which defaults to `"rcens"` for right censored data.

2.1.4 Multilabel classification

In multilabel classification each object can belong to more than one category at the same time.

The `data` are expected to contain as many target columns as there are class labels. The target columns should be logical vectors that indicate which class labels are present. The names of the target columns are taken as class labels and need to be passed to the `target` argument of `makeMultilabelTask](&Task()`.

In the following example we get the data of the yeast data set, extract the label names, and pass them to the `target` argument in `makeMultilabelTask](&Task()`.

```
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 functionals
##         103           0           0           0
## Missings: FALSE
## Has weights: FALSE
```

```
## Has blocking: FALSE
## Is spatial: 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
```

See also the tutorial page `vignette("multilabel")`.

2.1.5 Cluster analysis

As cluster analysis is unsupervised, the only mandatory argument to construct a cluster analysis task is the `data`. Below we create a learning task from the data set `datasets::mtcars()`.

```
data(mtcars, package = "datasets")
cluster.task = makeClusterTask(data = mtcars)
cluster.task
```

```
## Unsupervised task: mtcars
## Type: cluster
## Observations: 32
## Features:
##    numerics     factors     ordered functionals
##          11           0           0           0
## Missings: FALSE
## Has weights: FALSE
## Has blocking: FALSE
## Is spatial: FALSE
```

2.1.6 Cost-sensitive classification

The standard objective in classification is to obtain a high prediction accuracy, i.e., to minimize the number of errors. All types of misclassification errors are thereby deemed equally severe. However, in many applications different kinds of errors cause different costs.

In case of *class-dependent costs*, that solely depend on the actual and predicted class labels, it is sufficient to create an ordinary `ClassifTask()`.

In order to handle *example-specific costs* it is necessary to generate a `CostSensTask()`. In this scenario, each example (x, y) is associated with an individual cost vector of length K with K denoting the number of classes. The k-th component indicates the cost of assigning x to class k. Naturally, it is assumed that the cost of the intended class label y is minimal.

As the cost vector contains all relevant information about the intended class y, only the feature values x and a `cost` matrix, which contains the cost vectors for all examples in the data set, are required to create the `CostSensTask()`.

In the following example we use the `datasets::iris()` data and an artificial cost matrix (which is generated as proposed by Beygelzimer et al., 2005):

```
df = iris
cost = matrix(runif(150 * 3, 0, 2000), 150) * (1 - diag(3))[df$Species,]
df$Species = NULL
```

```
costsens.task = makeCostSensTask(data = df, cost = cost)
costsens.task
```

```
## Supervised task: df
## Type: costsens
## Observations: 150
## Features:
##    numerics     factors     ordered functionals
##           4           0           0           0
## Missings: FALSE
## Has blocking: FALSE
## Is spatial: FALSE
## Classes: 3
## y1, y2, y3
```

For more details see the page `vignette("cost_sensitive_classif")`.

2.2 Further settings

The `Task()` help page also lists several other arguments to describe further details of the learning problem.

For example, we could include a `blocking` factor in the task. This would indicate that some observations "belong together" and should not be separated when splitting the data into training and test sets for `vignette("resampling")`.

Another option is to assign `weights` to observations. These can simply indicate observation frequencies or result from the sampling scheme used to collect the data. Note that you should use this option only if the weights really belong to the task. If you plan to train some learning algorithms with different weights on the same `Task()`, `mlr` offers several other ways to set observation or class weights (for supervised classification). See for example the tutorial page about `vignette("train")` or function `makeWeightedClassesWrapper()`.

2.3 Accessing a learning task

We provide many operators to access the elements stored in a `Task()`. The most important ones are listed in the documentation of `Task()` and `getTaskData()`.

To access the `TaskDesc()` that contains basic information about the task you can use:

```
getTaskDesc(classif.task)
```

```
## $id
## [1] "BreastCancer"
##
## $type
## [1] "classif"
##
## $target
## [1] "Class"
##
## $size
## [1] 699
##
## $n.feat
##    numerics     factors     ordered functionals
##           0           4           5           0
```

```
##
## $has.missings
## [1] TRUE
##
## $has.weights
## [1] FALSE
##
## $has.blocking
## [1] FALSE
##
## $is.spatial
## [1] FALSE
##
## $class.levels
## [1] "benign"    "malignant"
##
## $positive
## [1] "malignant"
##
## $negative
## [1] "benign"
##
## $class.distribution
##
##    benign malignant
##       458       241
##
## attr(,"class")
## [1] "ClassifTaskDesc"    "SupervisedTaskDesc" "TaskDesc"
```

Note that `TaskDesc()` have slightly different elements for different types of `Task()`s. Frequently required elements can also be accessed directly.

```
## Get the ID
getTaskId(classif.task)
```

```
## [1] "BreastCancer"
```

```
## Get the type of task
getTaskType(classif.task)
```

```
## [1] "classif"
```

```
## Get the names of the target columns
getTaskTargetNames(classif.task)
```

```
## [1] "Class"
```

```
## Get the number of observations
getTaskSize(classif.task)
```

```
## [1] 699
```

```
## Get the number of input variables
getTaskNFeats(classif.task)
```

```
## [1] 9
```

```
## Get the class levels in classif.task
getTaskClassLevels(classif.task)
```

```
## [1] "benign"    "malignant"
```

Moreover, `mlr` provides several functions to extract data from a `Task()`.

```
## Accessing the data set in classif.task
str(getTaskData(classif.task))
```

```
## 'data.frame':    699 obs. of  10 variables:
##  $ Cl.thickness   : Ord.factor w/ 10 levels "1"<"2"<"3"<"4"<..: 5 5 3 6 4 8 1 2 2 4 ...
##  $ Cell.size      : Ord.factor w/ 10 levels "1"<"2"<"3"<"4"<..: 1 4 1 8 1 10 1 1 1 2 ...
##  $ Cell.shape     : Ord.factor w/ 10 levels "1"<"2"<"3"<"4"<..: 1 4 1 8 1 10 1 2 1 1 ...
##  $ Marg.adhesion  : Ord.factor w/ 10 levels "1"<"2"<"3"<"4"<..: 1 5 1 1 3 8 1 1 1 1 ...
##  $ Epith.c.size   : Ord.factor w/ 10 levels "1"<"2"<"3"<"4"<..: 2 7 2 3 2 7 2 2 2 2 ...
##  $ Bare.nuclei    : Factor w/ 10 levels "1","2","3","4",..: 1 10 2 4 1 10 10 1 1 1 ...
##  $ Bl.cromatin    : Factor w/ 10 levels "1","2","3","4",..: 3 3 3 3 3 9 3 3 1 2 ...
##  $ Normal.nucleoli: Factor w/ 10 levels "1","2","3","4",..: 1 2 1 7 1 7 1 1 1 1 ...
##  $ Mitoses        : Factor w/ 9 levels "1","2","3","4",..: 1 1 1 1 1 1 1 1 5 1 ...
##  $ Class          : Factor w/ 2 levels "benign","malignant": 1 1 1 1 1 2 1 1 1 1 ...
```

```
## Get the names of the input variables in cluster.task
getTaskFeatureNames(cluster.task)
```

```
##  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
## [11] "carb"
```

```
## Get the values of the target variables in surv.task
head(getTaskTargets(surv.task))
```

```
##   time status
## 1  306   TRUE
## 2  455   TRUE
## 3 1010  FALSE
## 4  210   TRUE
## 5  883   TRUE
## 6 1022  FALSE
```

```
## Get the cost matrix in costsens.task
head(getTaskCosts(costsens.task))
```

```
## NULL
```

Note that `getTaskData()` offers many options for converting the data set into a convenient format. This especially comes in handy when you integrate a new learner from another **R** package into `mlr`. In this regard function `getTaskFormula()` is also useful.

2.4 Modifying a learning task

`mlr` provides several functions to alter an existing `Task()`, which is often more convenient than creating a new `Task()` from scratch. Here are some examples.

```
## Select observations and/or features
cluster.task = subsetTask(cluster.task, subset = 4:17)

## It may happen, especially after selecting observations, that features are constant.
```

```
## These should be removed.
removeConstantFeatures(cluster.task)
```

```
## Removing 1 columns: am
```

```
## Unsupervised task: mtcars
## Type: cluster
## Observations: 14
## Features:
##    numerics     factors     ordered functionals
##          10           0           0           0
## Missings: FALSE
## Has weights: FALSE
## Has blocking: FALSE
## Is spatial: FALSE
```

```
## Remove selected features
dropFeatures(surv.task, c("meal.cal", "wt.loss"))
```

```
## Supervised task: lung
## Type: surv
## Target: time,status
## Events: 165
## Observations: 228
## Features:
##    numerics     factors     ordered functionals
##           6           0           0           0
## Missings: TRUE
## Has weights: FALSE
## Has blocking: FALSE
## Is spatial: FALSE
```

```
## Standardize numerical features
task = normalizeFeatures(cluster.task, method = "range")
summary(getTaskData(task))
```

```
##       mpg              cyl              disp              hp
##  Min.   :0.0000   Min.   :0.0000   Min.   :0.0000   Min.   :0.0000
##  1st Qu.:0.3161   1st Qu.:0.5000   1st Qu.:0.1242   1st Qu.:0.2801
##  Median :0.5107   Median :1.0000   Median :0.4076   Median :0.6311
##  Mean   :0.4872   Mean   :0.7143   Mean   :0.4430   Mean   :0.5308
##  3rd Qu.:0.6196   3rd Qu.:1.0000   3rd Qu.:0.6618   3rd Qu.:0.7473
##  Max.   :1.0000   Max.   :1.0000   Max.   :1.0000   Max.   :1.0000
##       drat              wt              qsec              vs
##  Min.   :0.0000   Min.   :0.0000   Min.   :0.0000   Min.   :0.0000
##  1st Qu.:0.2672   1st Qu.:0.1275   1st Qu.:0.2302   1st Qu.:0.0000
##  Median :0.3060   Median :0.1605   Median :0.3045   Median :0.0000
##  Mean   :0.4544   Mean   :0.3268   Mean   :0.3752   Mean   :0.4286
##  3rd Qu.:0.7026   3rd Qu.:0.3727   3rd Qu.:0.4908   3rd Qu.:1.0000
##  Max.   :1.0000   Max.   :1.0000   Max.   :1.0000   Max.   :1.0000
##        am           gear             carb
##  Min.   :0.5   Min.   :0.0000   Min.   :0.0000
##  1st Qu.:0.5   1st Qu.:0.0000   1st Qu.:0.3333
##  Median :0.5   Median :0.0000   Median :0.6667
##  Mean   :0.5   Mean   :0.2857   Mean   :0.6429
##  3rd Qu.:0.5   3rd Qu.:0.7500   3rd Qu.:1.0000
```

```
##  Max.   :0.5   Max.   :1.0000   Max.   :1.0000
```

For more functions and more detailed explanations have a look at the data preprocessing (`vignette("preproc")`) page.

2.5 Example tasks and convenience functions

For your convenience `mlr` provides pre-defined `Task()`s for each type of learning problem. These are also used throughout this tutorial in order to get shorter and more readable code. A list of all `Task()`s can be found in the Appendix (`vignette("example_tasks")`).

Moreover, `mlr`'s function `convertMLBenchObjToTask()` can generate `Task()`s from the data sets and data generating functions in package `mlbench::mlbench()`.

3 Learners

The following classes provide a unified interface to all popular machine learning methods in **R**: (cost-sensitive) classification, regression, survival analysis, and clustering. Many are already integrated in `mlr`, others are not, but the package is specifically designed to make extensions simple.

Section integrated learners (`vignette("integrated_learners")`) shows the already implemented machine learning methods and their properties. If your favorite method is missing, either open an issue or take a look at how to integrate a learning method yourself (`vignette("create_learners")`). This basic introduction demonstrates how to use already implemented learners.

3.1 Constructing a learner

A learner in `mlr` is generated by calling `makeLearner()`. In the constructor you need to specify which learning method you want to use. Moreover, you can:

- Set hyperparameters.
- Control the output for later prediction, e.g., for classification whether you want a factor of predicted class labels or probabilities.
- Set an ID to name the object (some methods will later use this ID to name results or annotate plots).

```
## Classification tree, set it up for predicting probabilities
classif.lrn = makeLearner("classif.randomForest", predict.type = "prob", fix.factors.prediction = TRUE)

## Regression gradient boosting machine, specify hyperparameters via a list
regr.lrn = makeLearner("regr.gbm", par.vals = list(n.trees = 500, interaction.depth = 3))

## Cox proportional hazards model with custom name
surv.lrn = makeLearner("surv.coxph", id = "cph")

## K-means with 5 clusters
cluster.lrn = makeLearner("cluster.kmeans", centers = 5)

## Multilabel Random Ferns classification algorithm
multilabel.lrn = makeLearner("multilabel.rFerns")
```

The first argument specifies which algorithm to use. The naming convention is `classif.<R_method_name>` for classification methods, `regr.<R_method_name>` for regression methods, `surv.<R_method_name>` for survival analysis, `cluster.<R_method_name>` for clustering methods, and `multilabel.<R_method_name>` for multilabel classification.

Hyperparameter values can be specified either via the `...` argument or as a `list` via `par.vals`.

Occasionally, `factor` features may cause problems when fewer levels are present in the test data set than in the training data. By setting `fix.factors.prediction = TRUE` these are avoided by adding a factor level for missing data in the test data set.

Let's have a look at two of the learners created above.

```
classif.lrn
```

```
## Learner classif.randomForest from package randomForest
## Type: classif
## Name: Random Forest; Short name: rf
## Class: classif.randomForest
## Properties: twoclass,multiclass,numerics,factors,ordered,prob,class.weights,oobpreds,featimp
## Predict-Type: prob
## Hyperparameters:
```

```
surv.lrn
```

```
## Learner cph from package survival
## Type: surv
## Name: Cox Proportional Hazard Model; Short name: coxph
## Class: surv.coxph
## Properties: numerics,factors,weights
## Predict-Type: response
## Hyperparameters:
```

All generated learners are objects of class Learner (`makeLearner()`). This class contains the properties of the method, e.g., which types of features it can handle, what kind of output is possible during prediction, and whether multi-class problems, observations weights or missing values are supported.

As you might have noticed, there is currently no special learner class for cost-sensitive classification. For ordinary misclassification costs you can use standard classification methods. For example-dependent costs there are several ways to generate cost-sensitive learners from ordinary regression and classification learners. This is explained in greater detail in the section about cost-sensitive classification (`vignette("cost_sensitive_classif")`).

3.2 Accessing a learner

The Learner (`makeLearner()`) object is a `list` and the following elements contain information regarding the hyperparameters and the type of prediction.

```
## Get the configured hyperparameter settings that deviate from the defaults
cluster.lrn$par.vals
```

```
## $centers
## [1] 5
```

```
## Get the set of hyperparameters
classif.lrn$par.set
```

```
##                     Type  len   Def   Constr Req Tunable Trafo
## ntree            integer    -   500 1 to Inf   -    TRUE     -
## mtry             integer    -     - 1 to Inf   -    TRUE     -
## replace          logical    -  TRUE        -   -    TRUE     -
## classwt     numericvector <NA>     - 0 to Inf   -    TRUE     -
## cutoff      numericvector <NA>     -   0 to 1   -    TRUE     -
## strata           untyped    -     -        -   -   FALSE     -
```

```
## sampsize    integervector <NA>     - 1 to Inf   -    TRUE     -
## nodesize          integer    -     1 1 to Inf   -    TRUE     -
## maxnodes          integer    -     - 1 to Inf   -    TRUE     -
## importance        logical    - FALSE        -   -    TRUE     -
## localImp          logical    - FALSE        -   -    TRUE     -
## proximity         logical    - FALSE        -   -   FALSE     -
## oob.prox          logical    -     -        -   Y   FALSE     -
## norm.votes        logical    -  TRUE        -   -   FALSE     -
## do.trace          logical    - FALSE        -   -   FALSE     -
## keep.forest       logical    -  TRUE        -   -   FALSE     -
## keep.inbag        logical    - FALSE        -   -   FALSE     -
```

```
## Get the type of prediction
regr.lrn$predict.type
```

```
## [1] "response"
```

Slot `$par.set` is an object of class `ParamSet` (`ParamHelpers::makeParamSet()`). It contains, among others, the type of hyperparameters (e.g., numeric, logical), potential default values and the range of allowed values.

Moreover, `mlr` provides function `getHyperPars()` or its alternative `getLearnerParVals()` to access the current hyperparameter setting of a Learner (`makeLearner()`) and `getParamSet()` to get a description of all possible settings. These are particularly useful in case of wrapped Learner (`makeLearner()`)s, for example if a learner is fused with a feature selection strategy, and both, the learner as well the feature selection method, have hyperparameters. For details see the section on wrapped learners.

```
## Get current hyperparameter settings
getHyperPars(cluster.lrn)
```

```
## $centers
## [1] 5
```

```
## Get a description of all possible hyperparameter settings
getParamSet(classif.lrn)
```

```
##                     Type  len   Def   Constr Req Tunable Trafo
## ntree            integer    -   500 1 to Inf   -    TRUE     -
## mtry             integer    -     - 1 to Inf   -    TRUE     -
## replace          logical    -  TRUE        -   -    TRUE     -
## classwt     numericvector <NA>     - 0 to Inf   -    TRUE     -
## cutoff      numericvector <NA>     -   0 to 1   -    TRUE     -
## strata           untyped    -     -        -   -   FALSE     -
## sampsize    integervector <NA>     - 1 to Inf   -    TRUE     -
## nodesize          integer    -     1 1 to Inf   -    TRUE     -
## maxnodes          integer    -     - 1 to Inf   -    TRUE     -
## importance        logical    - FALSE        -   -    TRUE     -
## localImp          logical    - FALSE        -   -    TRUE     -
## proximity         logical    - FALSE        -   -   FALSE     -
## oob.prox          logical    -     -        -   Y   FALSE     -
## norm.votes        logical    -  TRUE        -   -   FALSE     -
## do.trace          logical    - FALSE        -   -   FALSE     -
## keep.forest       logical    -  TRUE        -   -   FALSE     -
## keep.inbag        logical    - FALSE        -   -   FALSE     -
```

We can also use `getParamSet()` or its alias `getLearnerParamSet()` to get a quick overview about the available hyperparameters and defaults of a learning method without explicitly constructing it (by calling `makeLearner()`).

```
getParamSet("classif.randomForest")
```

```
##                       Type  len   Def   Constr Req Tunable Trafo
## ntree              integer    -   500 1 to Inf   -    TRUE     -
## mtry               integer    -     - 1 to Inf   -    TRUE     -
## replace            logical    -  TRUE        -   -    TRUE     -
## classwt     numericvector <NA>     - 0 to Inf   -    TRUE     -
## cutoff      numericvector <NA>     -   0 to 1   -    TRUE     -
## strata             untyped    -     -        -   -   FALSE     -
## sampsize    integervector <NA>     - 1 to Inf   -    TRUE     -
## nodesize           integer    -     1 1 to Inf   -    TRUE     -
## maxnodes           integer    -     - 1 to Inf   -    TRUE     -
## importance         logical    - FALSE        -   -    TRUE     -
## localImp           logical    - FALSE        -   -    TRUE     -
## proximity          logical    - FALSE        -   -   FALSE     -
## oob.prox           logical    -     -        -   Y   FALSE     -
## norm.votes         logical    -  TRUE        -   -   FALSE     -
## do.trace           logical    - FALSE        -   -   FALSE     -
## keep.forest        logical    -  TRUE        -   -   FALSE     -
## keep.inbag         logical    - FALSE        -   -   FALSE     -
```

Functions for accessing a Learner's meta information are available in `mlr`. We can use `getLearnerId()`, `getLearnerShortName()` and `getLearnerType()` to get Learner's ID, short name and type, respectively. Moreover, in order to show the required packages for the Learner, one can call `getLearnerPackages()`.

```
## Get object's id
getLearnerId(surv.lrn)
```

```
## [1] "cph"
```

```
## Get the short name
getLearnerShortName(classif.lrn)
```

```
## [1] "rf"
```

```
## Get the type of the learner
getLearnerType(multilabel.lrn)
```

```
## [1] "multilabel"
```

```
## Get required packages
getLearnerPackages(cluster.lrn)
```

```
## [1] "stats" "clue"
```

3.3 Modifying a learner

There are also some functions that enable you to change certain aspects of a Learner (`makeLearner()`) without needing to create a new Learner (`makeLearner()`) from scratch. Here are some examples.

```
## Change the ID
surv.lrn = setLearnerId(surv.lrn, "CoxModel")
surv.lrn
```

```
## Learner CoxModel from package survival
## Type: surv
## Name: Cox Proportional Hazard Model; Short name: coxph
## Class: surv.coxph
```

```
## Properties: numerics,factors,weights
## Predict-Type: response
## Hyperparameters:
```

```
## Change the prediction type, predict a factor with class labels instead of probabilities
classif.lrn = setPredictType(classif.lrn, "response")

## Change hyperparameter values
cluster.lrn = setHyperPars(cluster.lrn, centers = 4)

## Go back to default hyperparameter values
regr.lrn = removeHyperPars(regr.lrn, c("n.trees", "interaction.depth"))
```

3.4 Listing learners

A list of all learners integrated in `mlr` and their respective properties is shown in the Appendix (`vignette("integrated_learners")`).

If you would like a list of available learners, maybe only with certain properties or suitable for a certain learning `Task()` use function `listLearners()`.

```
## List everything in mlr
lrns = listLearners()
head(lrns[c("class", "package")])
```

```
##                 class      package
## 1         classif.ada    ada,rpart
## 2  classif.adaboostm1        RWeka
## 3 classif.bartMachine  bartMachine
## 4    classif.binomial        stats
## 5  classif.blackboost mboost,party
## 6    classif.boosting adabag,rpart
```

```
## List classifiers that can output probabilities
lrns = listLearners("classif", properties = "prob")
head(lrns[c("class", "package")])
```

```
##                 class      package
## 1         classif.ada    ada,rpart
## 2  classif.adaboostm1        RWeka
## 3 classif.bartMachine  bartMachine
## 4    classif.binomial        stats
## 5  classif.blackboost mboost,party
## 6    classif.boosting adabag,rpart
```

```
## List classifiers that can be applied to iris (i.e., multiclass) and output probabilities
lrns = listLearners(iris.task, properties = "prob")
head(lrns[c("class", "package")])
```

```
##                class      package
## 1 classif.adaboostm1        RWeka
## 2   classif.boosting adabag,rpart
## 3        classif.C50          C50
## 4    classif.cforest        party
## 5      classif.ctree        party
## 6   classif.cvglmnet       glmnet
```

```
## The calls above return character vectors, but you can also create learner objects
head(listLearners("cluster", create = TRUE), 2)
```

```
## [[1]]
## Learner cluster.cmeans from package e1071,clue
## Type: cluster
## Name: Fuzzy C-Means Clustering; Short name: cmeans
## Class: cluster.cmeans
## Properties: numerics,prob
## Predict-Type: response
## Hyperparameters: centers=2
##
##
## [[2]]
## Learner cluster.Cobweb from package RWeka
## Type: cluster
## Name: Cobweb Clustering Algorithm; Short name: cobweb
## Class: cluster.Cobweb
## Properties: numerics
## Predict-Type: response
## Hyperparameters:
```

4 Predicting Outcomes for New Data

Predicting the target values for new observations is implemented the same way as most of the other predict methods in **R**. In general, all you need to do is call `predict` (`predict.WrappedModel()`) on the object returned by `train()` and pass the data you want predictions for.

There are two ways to pass the data:

- Either pass the `Task()` via the `task` argument or
- pass a `data.frame` via the `newdata` argument.

The first way is preferable if you want predictions for data already included in a `Task()`.

Just as `train()`, the `predict` (`predict.WrappedModel()`) function has a `subset` argument, so you can set aside different portions of the data in `Task()` for training and prediction (more advanced methods for splitting the data in train and test set are described in the section on resampling (`vignette("resampling")`)).

In the following example we fit a gradient boosting machine (`gbm::gbm()`) to every second observation of the `BostonHousing` (`mlbench::BostonHousing()`) data set and make predictions on the remaining data in `bh.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)
mod = train(lrn, bh.task, subset = train.set)

task.pred = predict(mod, task = bh.task, subset = test.set)
task.pred
```

```
## Prediction: 253 observations
## predict.type: response
## threshold:
```

```
## time: 0.00
##    id truth response
## 2   2  21.6 22.23395
## 4   4  33.4 23.29479
## 6   6  28.7 22.32199
## 8   8  27.1 22.14655
## 10 10  18.9 22.14655
## 12 12  18.9 22.14655
## ... (#rows: 253, #cols: 3)
```

The second way is useful if you want to predict data not included in the `Task()`.

Here we cluster the `iris` data set without the target variable. All observations with an odd index are included in the `Task()` and used for training. Predictions are made for the remaining observations.

```
n = nrow(iris)
iris.train = iris[seq(1, n, by = 2), -5]
iris.test = iris[seq(2, n, by = 2), -5]
task = makeClusterTask(data = iris.train)
mod = train("cluster.kmeans", task)

newdata.pred = predict(mod, newdata = iris.test)
newdata.pred
```

```
## Prediction: 75 observations
## predict.type: response
## threshold:
## time: 0.00
##    response
## 2         1
## 4         1
## 6         1
## 8         1
## 10        1
## 12        1
## ... (#rows: 75, #cols: 1)
```

Note that for supervised learning you do not have to remove the target columns from the data. These columns are automatically removed prior to calling the underlying `predict` method of the learner.

4.1 Accessing the prediction

Function `predict()` returns a named `list` of class `Prediction()`. Its most important element is `$data` which is a `data.frame` that contains columns with the true values of the target variable (in case of supervised learning problems) and the predictions. Use `as.data.frame` (`Prediction()`) for direct access.

In the following the predictions on the `BostonHousing` (`mlbench::BostonHousing()`) and the `iris` (`datasets::iris()`) data sets are shown. As you may recall, the predictions in the first case were made from a `Task()` and in the second case from a `data.frame`.

```
## Result of predict with data passed via task argument
head(as.data.frame(task.pred))
```

```
##    id truth response
## 2   2  21.6 22.23395
## 4   4  33.4 23.29479
## 6   6  28.7 22.32199
```

```
## 8   8  27.1 22.14655
## 10 10  18.9 22.14655
## 12 12  18.9 22.14655
```

```
## Result of predict with data passed via newdata argument
head(as.data.frame(newdata.pred))
```

```
##    response
## 2         1
## 4         1
## 6         1
## 8         1
## 10        1
## 12        1
```

As you can see when predicting from a `Task()`, the resulting `data.frame` contains an additional column, called `id`, which tells us which element in the original data set the prediction corresponds to.

A direct way to access the true and predicted values of the target variable(s) is provided by functions `getPredictionTruth` (`getPredictionResponse()`) and [`getPredictionResponse()`].

```
head(getPredictionTruth(task.pred))
```

```
## [1] 21.6 33.4 28.7 27.1 18.9 18.9
```

```
head(getPredictionResponse(task.pred))
```

```
## [1] 22.23395 23.29479 22.32199 22.14655 22.14655 22.14655
```

4.1.1 Regression: Extracting standard errors

Some learners provide standard errors for predictions, which can be accessed in `mlr`. An overview is given by calling the function `listLearners()` and setting `properties = "se"`. By assigning `FALSE` to `check.packages` learners from packages which are not installed will be included in the overview.

```
listLearners("regr", check.packages = FALSE, properties = "se")[c("class", "name")]
```

```
##          class
## 1   regr.bcart
## 2     regr.bgp
## 3  regr.bgpllm
## 4     regr.blm
## 5    regr.btgp
## 6 regr.btgpllm
##                                                                      name
## 1                                                              Bayesian CART
## 2                                                 Bayesian Gaussian Process
## 3       Bayesian Gaussian Process with jumps to the Limiting Linear Model
## 4                                                     Bayesian Linear Model
## 5                                       Bayesian Treed Gaussian Process
## 6 Bayesian Treed Gaussian Process with jumps to the Limiting Linear Model
## ... (#rows: 16, #cols: 2)
```

In this example we train a linear regression model (`stats::lm()`) on the `BostonHousing` (`bh.task()`) dataset. In order to calculate standard errors set the `predict.type` to `"se"`:

```
## Create learner and specify predict.type
lrn.lm = makeLearner("regr.lm", predict.type = 'se')
```

```
mod.lm = train(lrn.lm, bh.task, subset = train.set)
task.pred.lm = predict(mod.lm, task = bh.task, subset = test.set)
task.pred.lm
```

```
## Prediction: 253 observations
## predict.type: se
## threshold:
## time: 0.00
##    id truth response        se
## 2   2  21.6 24.83734 0.7501615
## 4   4  33.4 28.38206 0.8742590
## 6   6  28.7 25.16725 0.8652139
## 8   8  27.1 19.38145 1.1963265
## 10 10  18.9 18.66449 1.1793944
## 12 12  18.9 21.25802 1.0727918
## ... (#rows: 253, #cols: 4)
```

The standard errors can then be extracted using `getPredictionSE()`.

```
head(getPredictionSE(task.pred.lm))
```

```
## [1] 0.7501615 0.8742590 0.8652139 1.1963265 1.1793944 1.0727918
```

4.1.2 Classification and clustering: Extracting probabilities

The predicted probabilities can be extracted from the `Prediction()` using function `getPredictionProbabilities()`. Here is another cluster analysis example. We use fuzzy c-means clustering (`e1071::cmeans()`) on the `mtcars` (`datasets::mtcars()`) data set.

```
lrn = makeLearner("cluster.cmeans", predict.type = "prob")
mod = train(lrn, mtcars.task)

pred = predict(mod, task = mtcars.task)
head(getPredictionProbabilities(pred))
```

```
##                           1           2
## Mazda RX4         0.9795917 0.020408274
## Mazda RX4 Wag     0.9796319 0.020368055
## Datsun 710        0.9926610 0.007338952
## Hornet 4 Drive    0.5428997 0.457100326
## Hornet Sportabout 0.0187042 0.981295796
## Valiant           0.7574491 0.242550890
```

For *classification problems* there are some more things worth mentioning. By default, class labels are predicted.

```
## Linear discriminant analysis on the iris data set
mod = train("classif.lda", task = iris.task)

pred = predict(mod, task = iris.task)
pred
```

```
## 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
## ... (#rows: 150, #cols: 3)
```

In order to get predicted posterior probabilities we have to create a Learner (`makeLearner()`) with the appropriate `predict.type`.

```
lrn = makeLearner("classif.rpart", predict.type = "prob")
mod = train(lrn, iris.task)

pred = predict(mod, newdata = iris)
head(as.data.frame(pred))
```

```
##    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
```

In addition to the probabilities, class labels are predicted by choosing the class with the maximum probability and breaking ties at random.

As mentioned above, the predicted posterior probabilities can be accessed via the `getPredictionProbabilities()` function.

```
head(getPredictionProbabilities(pred))
```

```
##   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
```

4.1.3 Classification: Confusion matrix

A confusion matrix can be obtained by calling `calculateConfusionMatrix()`. The columns represent predicted and the rows true class labels.

```
calculateConfusionMatrix(pred)
```

```
##             predicted
## true         setosa versicolor virginica -err.-
##   setosa         50          0         0      0
##   versicolor      0         49         1      1
##   virginica       0          5        45      5
##   -err.-          0          5         1      6
```

You can see the number of correctly classified observations on the diagonal of the matrix. Misclassified observations are on the off-diagonal. The total number of errors for single (true and predicted) classes is

shown in the `-err.-` row and column, respectively.

To get relative frequencies additional to the absolute numbers we can set `relative = TRUE`.

```
conf.matrix = calculateConfusionMatrix(pred, relative = TRUE)
conf.matrix
```

```
## Relative confusion matrix (normalized by row/column):
##             predicted
## true         setosa    versicolor virginica -err.-
##   setosa     1.00/1.00 0.00/0.00  0.00/0.00 0.00
##   versicolor 0.00/0.00 0.98/0.91  0.02/0.02 0.02
##   virginica  0.00/0.00 0.10/0.09  0.90/0.98 0.10
##   -err.-          0.00      0.09       0.02 0.04
##
##
## Absolute confusion matrix:
##             predicted
## true         setosa versicolor virginica -err.-
##   setosa         50          0         0      0
##   versicolor      0         49         1      1
##   virginica       0          5        45      5
##   -err.-          0          5         1      6
```

It is possible to normalize by either row or column, therefore every element of the above relative confusion matrix contains two values. The first is the relative frequency grouped by row (the true label) and the second value grouped by column (the predicted label).

If you want to access the relative values directly you can do this through the `$relative.row` and `$relative.col` members of the returned object `conf.matrix`. For more details see the `ConfusionMatrix()` documentation page.

```
conf.matrix$relative.row
```

```
##            setosa versicolor virginica -err-
## setosa          1       0.00      0.00  0.00
## versicolor      0       0.98      0.02  0.02
## virginica       0       0.10      0.90  0.10
```

Finally, we can also add the absolute number of observations for each predicted and true class label to the matrix (both absolute and relative) by setting `sums = TRUE`.

```
calculateConfusionMatrix(pred, relative = TRUE, sums = TRUE)
```

```
## Relative confusion matrix (normalized by row/column):
##             predicted
## true         setosa    versicolor virginica -err.-    -n-
##   setosa     1.00/1.00 0.00/0.00  0.00/0.00 0.00      50
##   versicolor 0.00/0.00 0.98/0.91  0.02/0.02 0.02      54
##   virginica  0.00/0.00 0.10/0.09  0.90/0.98 0.10      46
##   -err.-          0.00      0.09       0.02 0.04      <NA>
##   -n-        50        50         50        <NA>      150
##
##
## Absolute confusion matrix:
##            setosa versicolor virginica -err.- -n-
## setosa         50          0         0      0  50
## versicolor      0         49         1      1  50
```

```
## virginica       0          5        45      5  50
## -err.-          0          5         1      6  NA
## -n-            50         54        46     NA 150
```

4.2 Classification: Adjusting the decision threshold

We can set the threshold value that is used to map the predicted posterior probabilities to class labels. Note that for this purpose we need to create a Learner (`makeLearner()`) that predicts probabilities. For binary classification, the threshold determines when the *positive* class is predicted. The default is 0.5. Now, we set the threshold for the positive class to 0.9 (that is, an example is assigned to the positive class if its posterior probability exceeds 0.9). Which of the two classes is the positive one can be seen by accessing the `Task()`. To illustrate binary classification, we use the `Sonar` (`mlbench::Sonar()`) data set from the `mlbench` package.

```
lrn = makeLearner("classif.rpart", predict.type = "prob")
mod = train(lrn, task = sonar.task)

## Label of the positive class
getTaskDesc(sonar.task)$positive
```

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

```
## Default threshold
pred1 = predict(mod, sonar.task)
pred1$threshold
```

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

```
## Set the threshold value for the positive class
pred2 = setThreshold(pred1, 0.9)
pred2$threshold
```

```
##   M   R
## 0.9 0.1
```

```
pred2
```

```
## Prediction: 208 observations
## predict.type: prob
## threshold: M=0.90,R=0.10
## time: 0.00
##   id truth    prob.M    prob.R response
## 1  1     R 0.1060606 0.8939394        R
## 2  2     R 0.7333333 0.2666667        R
## 3  3     R 0.0000000 1.0000000        R
## 4  4     R 0.1060606 0.8939394        R
## 5  5     R 0.9250000 0.0750000        M
## 6  6     R 0.0000000 1.0000000        R
## ... (#rows: 208, #cols: 5)
```

```
## We can also set the effect in the confusion matrix
calculateConfusionMatrix(pred1)
```

```
##         predicted
## true      M  R -err.-
##   M      95 16     16
##   R      10 87     10
```

```
##   -err.- 10 16     26
```

```
calculateConfusionMatrix(pred2)
```

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

Note that in the binary case `getPredictionProbabilities()` by default extracts the posterior probabilities of the positive class only.

```
head(getPredictionProbabilities(pred1))
```

```
## [1] 0.1060606 0.7333333 0.0000000 0.1060606 0.9250000 0.0000000
```

```
## But we can change that, too
head(getPredictionProbabilities(pred1, cl = c("M", "R")))
```

```
##           M         R
## 1 0.1060606 0.8939394
## 2 0.7333333 0.2666667
## 3 0.0000000 1.0000000
## 4 0.1060606 0.8939394
## 5 0.9250000 0.0750000
## 6 0.0000000 1.0000000
```

It works similarly for multiclass classification. The threshold has to be given by a named vector specifying the values by which each probability will be divided. The class with the maximum resulting value is then selected.

```
lrn = makeLearner("classif.rpart", predict.type = "prob")
mod = train(lrn, iris.task)
pred = predict(mod, newdata = iris)
pred$threshold
```

```
##     setosa versicolor  virginica
##  0.3333333  0.3333333  0.3333333
```

```
table(as.data.frame(pred)$response)
```

```
##
##     setosa versicolor  virginica
##         50         54         46
```

```
pred = setThreshold(pred, c(setosa = 0.01, versicolor = 50, virginica = 1))
pred$threshold
```

```
##     setosa versicolor  virginica
##       0.01      50.00       1.00
```

```
table(as.data.frame(pred)$response)
```

```
##
##     setosa versicolor  virginica
##         50          0        100
```

If you are interested in tuning the threshold (vector) have a look at the section about performance curves and threshold tuning (`vignette("roc_analysis")`).

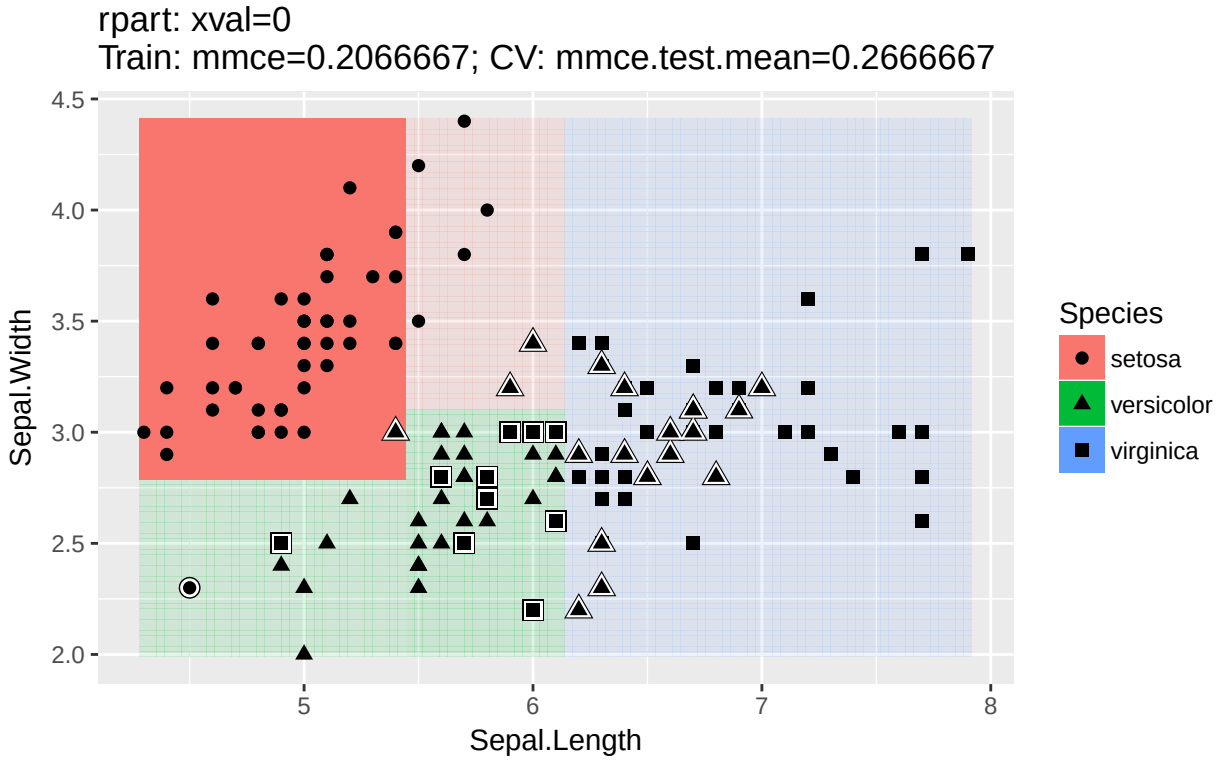
4.3 Visualizing the prediction

The function `plotLearnerPrediction()` allows to visualize predictions, e.g., for teaching purposes or exploring models. It trains the chosen learning method for 1 or 2 selected features and then displays the predictions with `ggplot2::ggplot()`.

For *classification*, we get a scatter plot of 2 features (by default the first 2 in the data set). The type of symbol shows the true class labels of the data points. Symbols with white border indicate misclassified observations. The posterior probabilities (if the learner under consideration supports this) are represented by the background color where higher saturation means larger probabilities.

The plot title displays the ID of the Learner (`makeLearner()`) (in the following example CART), its parameters, its training performance and its cross-validation performance. `mmce` (`vignette("measures")`) stands for *mean misclassification error*, i.e., the error rate. See the sections on performance (`vignette("performance")`) and resampling (`vignette("resampling")`) for further explanations.

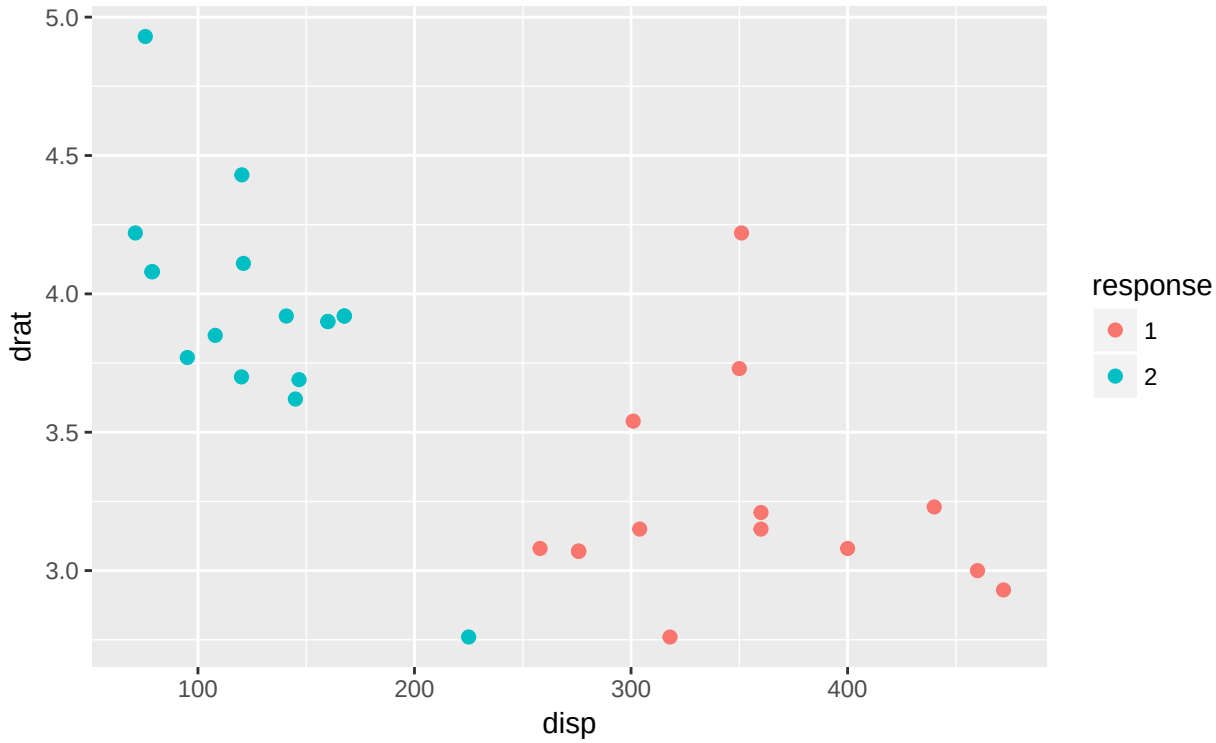
```
lrn = makeLearner("classif.rpart", id = "CART")
plotLearnerPrediction(lrn, task = iris.task)
```



For *clustering* we also get a scatter plot of two selected features. The color of the points indicates the predicted cluster.

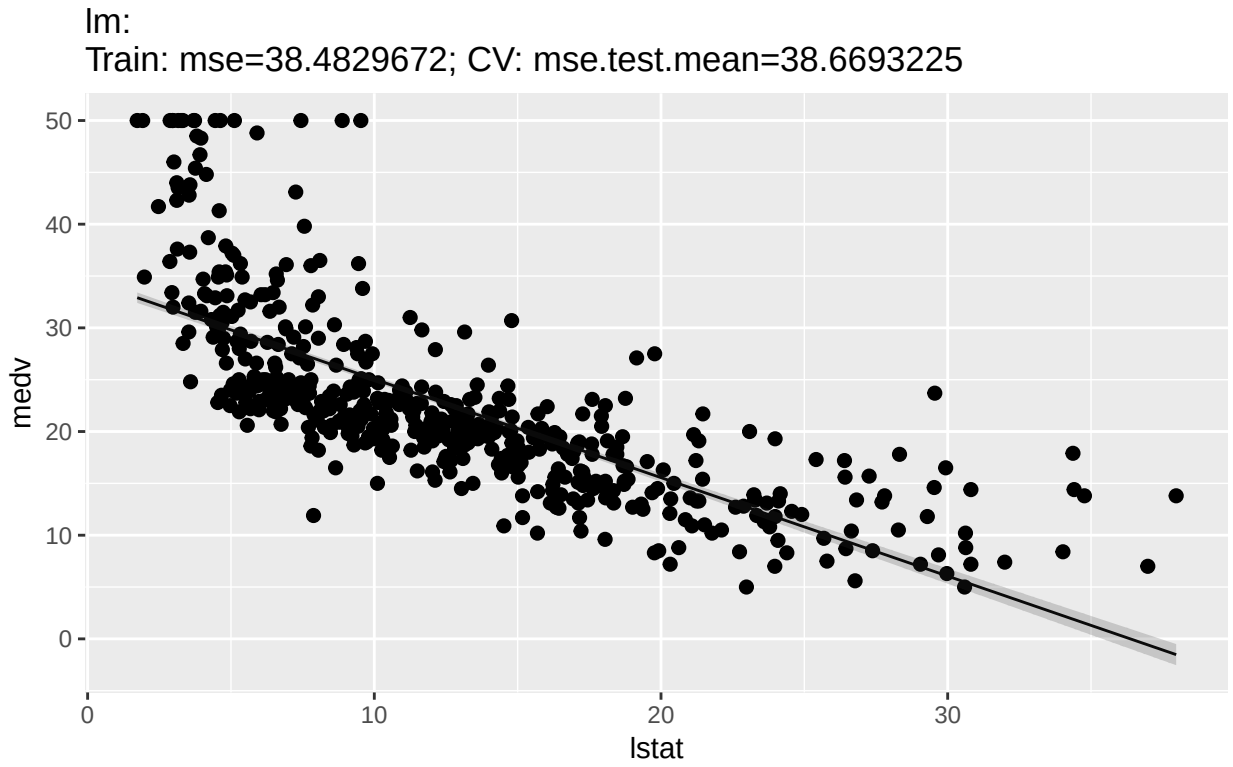
```
lrn = makeLearner("cluster.kmeans")
plotLearnerPrediction(lrn, task = mtcars.task, features = c("disp", "drat"), cv = 0)
```

```
##
## This is package 'modeest' written by P. PONCET.
## For a complete list of functions, use 'library(help = "modeest")' or 'help.start()'.
```



For *regression*, there are two types of plots. The 1D plot shows the target values in relation to a single feature, the regression curve and, if the chosen learner supports this, the estimated standard error.

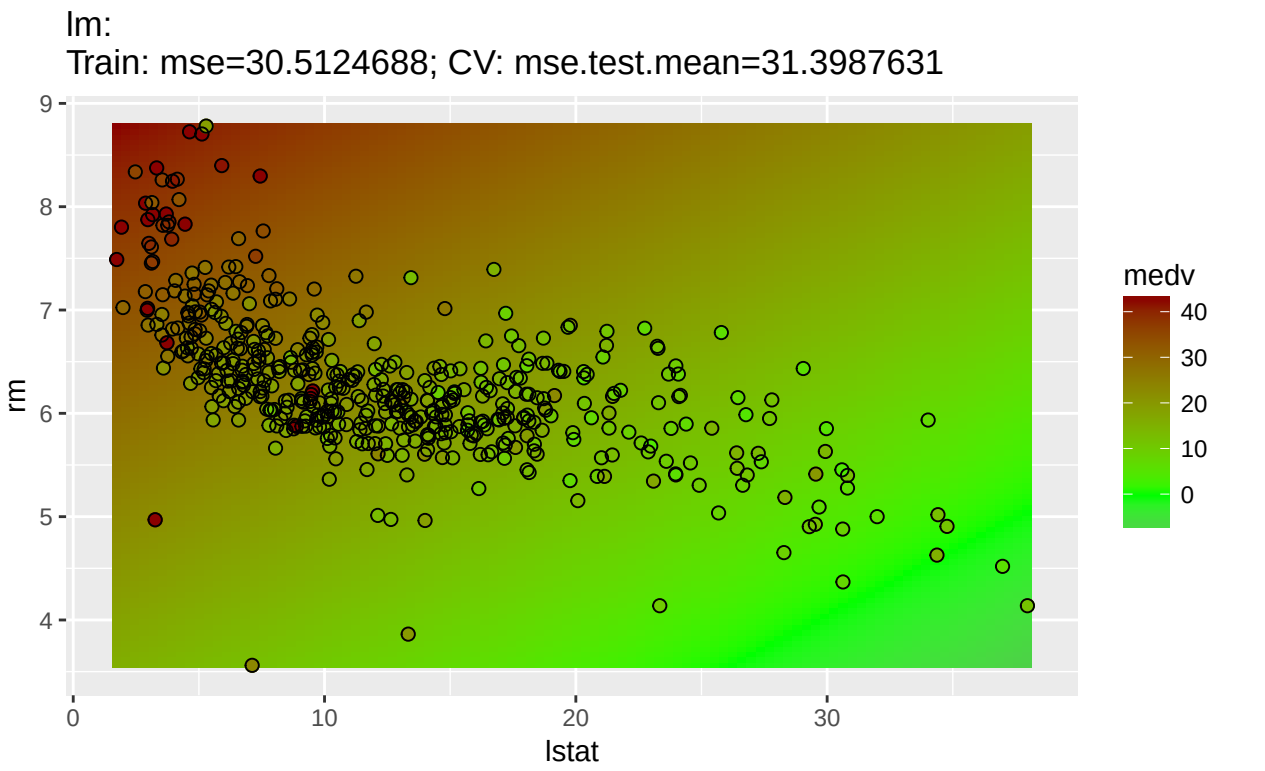
```
plotLearnerPrediction("regr.lm", features = "lstat", task = bh.task)
```



The 2D variant, as in the classification case, generates a scatter plot of 2 features. The fill color of the dots illustrates the value of the target variable `"medv"`, the background colors show the estimated mean. The plot

does not represent the estimated standard error.

```
plotLearnerPrediction("regr.lm", features = c("lstat", "rm"), task = bh.task)
```



5 Evaluating Learner Performance

The quality of the predictions of a model in `mlr` can be assessed with respect to a number of different performance measures. In order to calculate the performance measures, call `performance()` on the object returned by `predict` (`predict.WrappedModel()`) and specify the desired performance measures.

5.1 Available performance measures

`mlr` provides a large number of performance measures for all types of learning problems. Typical performance measures for *classification* are the mean misclassification error (`mmce` (`vignette("measures")`)), accuracy (`acc` (`vignette("measures")`)) or measures based on ROC analysis (`vignette("roc_analysis")`). For *regression* the mean of squared errors (`mse` (`vignette("measures")`)) or mean of absolute errors (`mae` (`vignette("measures")`)) are usually considered. For *clustering* tasks, measures such as the Dunn index (`dunn` (`vignette("measures")`)) are provided, while for *survival* predictions, the Concordance Index (`cindex` (`vignette("measures")`)) is supported, and for *cost-sensitive* predictions the misclassification penalty (`mcp` (`vignette("measures")`)) and others. It is also possible to access the time to train the learner (`timetrain` (`vignette("measures")`)), the time to compute the prediction (`timepredict` (`vignette("measures")`)) and their sum (`timeboth` (`vignette("measures")`)) as performance measures.

To see which performance measures are implemented, have a look at the table of performance measures (`vignette("measures")`) and the `measures()` documentation page.

If you want to implement an additional measure or include a measure with non-standard misclassification costs, see the section on creating custom measures (`vignette("create_measure")`).

5.2 Listing measures

The properties and requirements of the individual measures are shown in the table of performance measures (`vignette("measures")`).

If you would like a list of available measures with certain properties or suitable for a certain learning `Task()` use the function `listMeasures()`.

```
## Performance measures for classification with multiple classes
listMeasures("classif", properties = "classif.multi")
```

```
##  [1] "kappa"            "multiclass.brier" "multiclass.aunp"
##  [4] "multiclass.aunu"  "qsr"              "ber"
##  [7] "logloss"          "wkappa"           "timeboth"
## [10] "timepredict"      "acc"              "lsr"
## [13] "featperc"         "multiclass.au1p"  "multiclass.au1u"
## [16] "ssr"              "timetrain"        "mmce"
```

```
## Performance measure suitable for the iris classification task
listMeasures(iris.task)
```

```
##  [1] "kappa"            "multiclass.brier" "multiclass.aunp"
##  [4] "multiclass.aunu"  "qsr"              "ber"
##  [7] "logloss"          "wkappa"           "timeboth"
## [10] "timepredict"      "acc"              "lsr"
## [13] "featperc"         "multiclass.au1p"  "multiclass.au1u"
## [16] "ssr"              "timetrain"        "mmce"
```

For convenience there exists a default measure for each type of learning problem, which is calculated if nothing else is specified. As defaults we chose the most commonly used measures for the respective types, e.g., the mean squared error (`mse` (`vignette("measures")`)) for regression and the misclassification rate (`mmce` (`vignette("measures")`)) for classification. The help page of function `getDefaultMeasure()` lists all defaults for all types of learning problems. The function itself returns the default measure for a given task type, `Task()` or `Learner()`.

```
## Get default measure for iris.task
getDefaultMeasure(iris.task)
```

```
## Name: Mean misclassification error
## Performance measure: mmce
## Properties: classif,classif.multi,req.pred,req.truth
## Minimize: TRUE
## Best: 0; Worst: 1
## Aggregated by: test.mean
## Arguments:
## Note: Defined as: mean(response != truth)
```

```
## Get the default measure for linear regression
getDefaultMeasure(makeLearner("regr.lm"))
```

```
## Name: Mean of squared errors
## Performance measure: mse
## Properties: regr,req.pred,req.truth
## Minimize: TRUE
## Best: 0; Worst: Inf
## Aggregated by: test.mean
## Arguments:
## Note: Defined as: mean((response - truth)^2)
```

5.3 Calculate performance measures

In the following example we fit a gradient boosting machine (`gbm::gbm()`) on a subset of the `BostonHousing` (`mlbench::BostonHousing()`) data set and calculate the default measure mean squared error (`mse` (`vignette("measures")`)) on the remaining observations.

```
n = getTaskSize(bh.task)
lrn = makeLearner("regr.gbm", n.trees = 1000)
mod = train(lrn, task = bh.task, subset = seq(1, n, 2))
pred = predict(mod, task = bh.task, subset = seq(2, n, 2))

performance(pred)
```

```
##      mse
## 42.78716
```

The following code computes the median of squared errors (`medse` (`vignette("measures")`)) instead.

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

```
##    medse
## 9.012037
```

Of course, we can also calculate multiple performance measures at once by simply passing a list of measures which can also include your own measure (`vignette("create_measure")`).

Calculate the mean squared error, median squared error and mean absolute error (`mae` (`vignette("measures")`)).

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

```
##       mse     medse       mae
## 42.787158  9.012037  4.545393
```

For the other types of learning problems and measures, calculating the performance basically works in the same way.

5.3.1 Requirements of performance measures

Note that in order to calculate some performance measures it is required that you pass the `Task()` or the fitted model (`makeWrappedModel()`) in addition to the `Prediction()`.

For example in order to assess the time needed for training (`timetrain` (`vignette("measures")`)), the fitted model has to be passed.

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

```
## timetrain
##     0.515
```

For many performance measures in cluster analysis the `Task()` is required.

```
lrn = makeLearner("cluster.kmeans", centers = 3)
mod = train(lrn, mtcars.task)
pred = predict(mod, task = mtcars.task)

## Calculate the Dunn index
performance(pred, measures = dunn, task = mtcars.task)
```

```
##      dunn
## 0.2278991
```

Moreover, some measures require a certain type of prediction. For example in binary classification in order to calculate the AUC (`auc` (`vignette("measures")`)) – the area under the ROC (receiver operating characteristic) curve – we have to make sure that posterior probabilities are predicted. For more information on ROC analysis, see the section on ROC analysis (`vignette("roc_analysis")`).

```
lrn = makeLearner("classif.rpart", predict.type = "prob")
mod = train(lrn, task = sonar.task)
pred = predict(mod, task = sonar.task)

performance(pred, measures = auc)
```

```
##       auc
## 0.9224018
```

Also bear in mind that many of the performance measures that are available for classification, e.g., the false positive rate (`fpr` (`vignette("measures")`)), are only suitable for binary problems.

5.4 Access a performance measure

Performance measures in `mlr` are objects of class `Measure` (`makeMeasure()`). If you are interested in the properties or requirements of a single measure you can access it directly. See the help page of `Measure` (`makeMeasure()`) for information on the individual slots.

```
## Mean misclassification error
str(mmce)
```

```
## List of 10
##  $ id        : chr "mmce"
##  $ minimize  : logi TRUE
##  $ properties: chr [1:4] "classif" "classif.multi" "req.pred" "req.truth"
##  $ fun       :function (task, model, pred, feats, extra.args)
##   ..- attr(*, "srcref")=Class 'srcref'  atomic [1:8] 464 9 466 3 9 3 26997 26999
##   .. .. ..- attr(*, "srcfile")=Classes 'srcfilealias', 'srcfile' <environment: 0x560e400a9d18>
##  $ extra.args: list()
##  $ best      : num 0
##  $ worst     : num 1
##  $ name      : chr "Mean misclassification error"
##  $ note      : chr "Defined as: mean(response != truth)"
##  $ aggr      :List of 4
##   ..$ id        : chr "test.mean"
##   ..$ name      : chr "Test mean"
##   ..$ fun       :function (task, perf.test, perf.train, measure, group, pred)
##   .. ..- attr(*, "srcref")=Class 'srcref'  atomic [1:8] 43 9 43 83 9 83 19052 19052
##   .. .. .. ..- attr(*, "srcfile")=Classes 'srcfilealias', 'srcfile' <environment: 0x560e400a4d68>
##   ..$ properties: chr "req.test"
##   ..- attr(*, "class")= chr "Aggregation"
##  - attr(*, "class")= chr "Measure"
```

5.5 Binary classification

For binary classification specialized techniques exist to analyze the performance.

5.5.1 Plot performance versus threshold

As you may recall (see the previous section on making predictions (`vignette("predict")`)) in binary classification we can adjust the threshold used to map probabilities to class labels. Helpful in this regard is are the functions `generateThreshVsPerfData()` and `plotThreshVsPerf()`, which generate and plot, respectively, the learner performance versus the threshold.

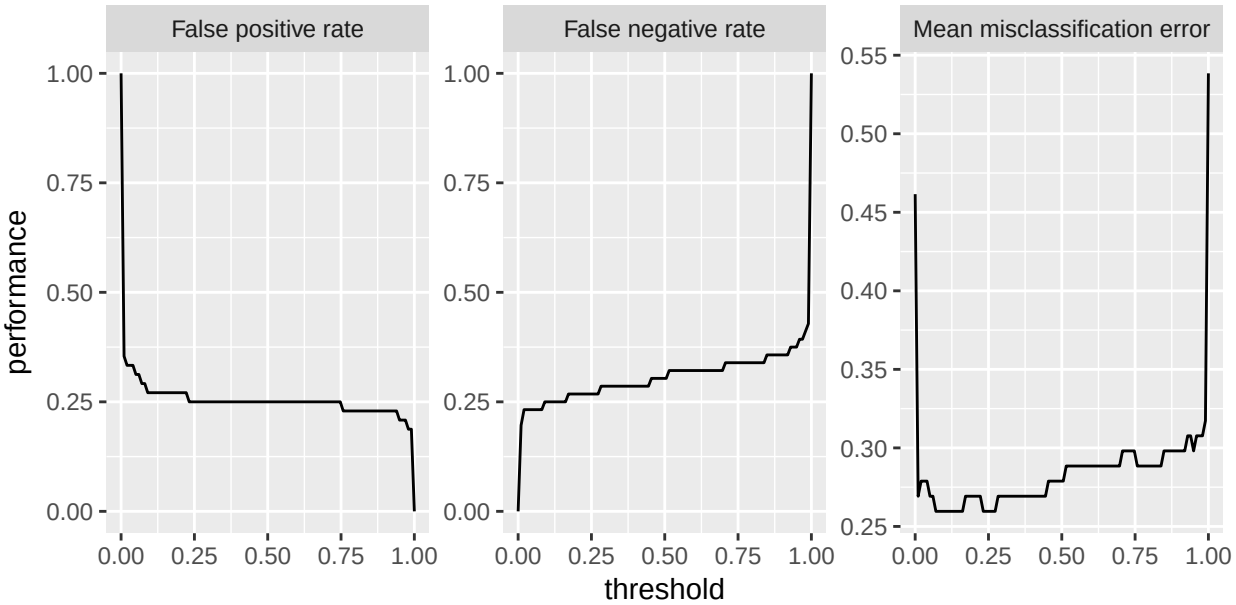
For more performance plots and automatic threshold tuning see `vignette("roc_analysis")`.

In the following example we consider the `mlbench::Sonar()` data set and plot the false positive rate (`fpr` (`vignette("measures")`)), the false negative rate (`fnr` (`vignette("measures")`)) as well as the misclassification rate (`mmce` (`vignette("measures")`)) for all possible threshold values.

```
lrn = makeLearner("classif.lda", predict.type = "prob")
n = getTaskSize(sonar.task)
mod = train(lrn, task = sonar.task, subset = seq(1, n, by = 2))
pred = predict(mod, task = sonar.task, subset = seq(2, n, by = 2))

## Performance for the default threshold 0.5
performance(pred, measures = list(fpr, fnr, mmce))
```

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

```
## Plot false negative and positive rates as well as the error rate versus the threshold
d = generateThreshVsPerfData(pred, measures = list(fpr, fnr, mmce))
plotThreshVsPerf(d)
```



There is an experimental `ggvis` plotting function `plotThreshVsPerfGGVIS()` which performs similarly to `plotThreshVsPerf()` but instead of creating facetted subplots to visualize multiple learners and/or multiple measures, one of them is mapped to an interactive sidebar which selects what to display.

```
plotThreshVsPerfGGVIS(d)
```

5.5.2 ROC measures

For binary classification a large number of specialized measures exist, which can be nicely formatted into one matrix, see for example the receiver operating characteristic page on wikipedia.

We can generate a similiar table with the `calculateROCMeasures()` function.

```
r = calculateROCMeasures(pred)
r
```

```
##     predicted
## true M         R
##    M 39        17        tpr: 0.7  fnr: 0.3
##    R 12        36        fpr: 0.25 tnr: 0.75
##      ppv: 0.76 for: 0.32 lrp: 2.79 acc: 0.72
##      fdr: 0.24 npv: 0.68 lrm: 0.4  dor: 6.88
##
##
## Abbreviations:
## tpr - True positive rate (Sensitivity, Recall)
## fpr - False positive rate (Fall-out)
## fnr - False negative rate (Miss rate)
## tnr - True negative rate (Specificity)
## ppv - Positive predictive value (Precision)
## for - False omission rate
## lrp - Positive likelihood ratio (LR+)
## fdr - False discovery rate
## npv - Negative predictive value
## acc - Accuracy
## lrm - Negative likelihood ratio (LR-)
## dor - Diagnostic odds ratio
```

The top left 2×2 matrix is the confusion matrix (`vignette("predict")`), which shows the relative frequency of correctly and incorrectly classified observations. Below and to the right a large number of performance measures that can be inferred from the confusion matrix are added. By default some additional info about the measures is printed. You can turn this off using the `abbreviations` argument of the `print` (`calculateROCMeasures()`) method: `print(r, abbreviations = FALSE)`.

6 Resampling

Resampling strategies are usually used to assess the performance of a learning algorithm: The entire data set is (repeatedly) split into training sets D^{*b} and test sets $D \setminus D^{*b}$, $b = 1, \ldots, B$. The learner is trained on each training set, predictions are made on the corresponding test set (sometimes on the training set as well) and the performance measure $S(D^{*b}, D \setminus D^{*b})$ is calculated. Then the B individual performance values are aggregated, most often by calculating the mean. There exist various different resampling strategies, for example cross-validation and bootstrap, to mention just two popular approaches.

If you want to read up on further details, the paper Resampling Strategies for Model Assessment and Selection by Simon is probably not a bad choice. Bernd has also published a paper Resampling methods for meta-model validation with recommendations for evolutionary computation which contains detailed descriptions and lots of statistical background information on resampling methods.

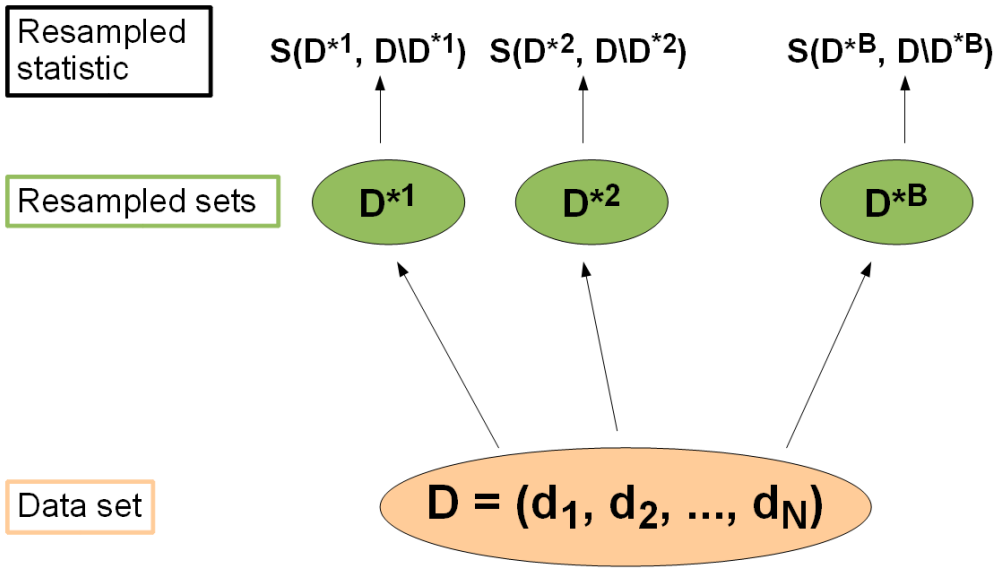


Figure 1: Resampling Figure

6.1 Defining the resampling strategy

In `mlr` the resampling strategy can be defined via function `makeResampleDesc()`. It requires a string that specifies the resampling method and, depending on the selected strategy, further information like the number of iterations. The supported resampling strategies are:

- Cross-validation (`"CV"`),
- Leave-one-out cross-validation (`"LOO"`),
- Repeated cross-validation (`"RepCV"`),
- Out-of-bag bootstrap and other variants like *b632* (`"Bootstrap"`),
- Subsampling, also called Monte-Carlo cross-validation (`"Subsample"`),
- Holdout (training/test) (`"Holdout"`).

For example if you want to use 3-fold cross-validation type:

```
## 3-fold cross-validation
rdesc = makeResampleDesc("CV", iters = 3)
rdesc
```

```
## Resample description: cross-validation with 3 iterations.
## Predict: test
## Stratification: FALSE
```

For holdout estimation use:

```
## Holdout estimation
rdesc = makeResampleDesc("Holdout")
rdesc
```

```
## Resample description: holdout with 0.67 split rate.
## Predict: test
## Stratification: FALSE
```

In order to save you some typing `mlr` contains some pre-defined resample descriptions for very common

strategies like holdout (`hout` (`makeResampleDesc()`)) as well as cross-validation with different numbers of folds (e.g., `cv5` (`makeResampleDesc()`) or `cv10` (`makeResampleDesc()`)).

```
hout
```

```
## Resample description: holdout with 0.67 split rate.
## Predict: test
## Stratification: FALSE
```

```
cv3
```

```
## Resample description: cross-validation with 3 iterations.
## Predict: test
## Stratification: FALSE
```

6.2 Performing the resampling

Function `resample()` evaluates a Learner (`makeLearner()`) on a given machine learning `Task()` using the selected resampling strategy (`makeResampleDesc()`).

As a first example, the performance of linear regression (`stats::lm()`) on the `BostonHousing` (`mlbench::BostonHousing()`) data set is calculated using *3-fold cross-validation*.

Generally, for *K-fold cross-validation* the data set D is partitioned into K subsets of (approximately) equal size. In the b-th of the K iterations, the b-th subset is used for testing, while the union of the remaining parts forms the training set.

As usual, you can either pass a Learner (`makeLearner()`) object to `resample()` or, as done here, provide the class name `"regr.lm"` of the learner. Since no performance measure is specified the default for regression learners (mean squared error, `mse` (`vignette("measures")`)) is calculated.

```
## Specify the resampling strategy (3-fold cross-validation)
rdesc = makeResampleDesc("CV", iters = 3)

## Calculate the performance
r = resample("regr.lm", bh.task, rdesc)
```

```
## Resampling: cross-validation
## Measures:             mse
## [Resample] iter 1:    24.3463204
## [Resample] iter 2:    24.2083583
## [Resample] iter 3:    22.8403257
##
## Aggregated Result: mse.test.mean=23.7983348
##
```

```
r
```

```
## Resample Result
## Task: BostonHousing-example
## Learner: regr.lm
## Aggr perf: mse.test.mean=23.7983348
## Runtime: 0.0362196
```

The result `r` is an object of class `resample()` result. It contains performance results for the learner and some additional information like the runtime, predicted values, and optionally the models fitted in single resampling iterations.

```
## Peak into r
names(r)
```

```
##  [1] "learner.id"     "task.id"        "task.desc"      "measures.train"
##  [5] "measures.test"  "aggr"           "pred"           "models"
##  [9] "err.msgs"       "err.dumps"      "extract"        "runtime"
```

```
r$aggr
```

```
## mse.test.mean
##      23.79833
```

```
r$measures.test
```

```
##   iter      mse
## 1    1 24.34632
## 2    2 24.20836
## 3    3 22.84033
```

`r$measures.test` gives the performance on each of the 3 test data sets. `r$aggr` shows the aggregated performance value. Its name `"mse.test.mean"` indicates the performance measure, `mse` (`vignette("measures")`), and the method, `test.mean` (`aggregations()`), used to aggregate the 3 individual performances. `test.mean` (`aggregations()`) is the default aggregation scheme for most performance measures and, as the name implies, takes the mean over the performances on the test data sets.

Resampling in `mlr` works the same way for all types of learning problems and learners. Below is a classification example where a classification tree (rpart) (`rpart::rpart()`) is evaluated on the `Sonar` (`mlbench::sonar()`) data set by subsampling with 5 iterations.

In each subsampling iteration the data set D is randomly partitioned into a training and a test set according to a given percentage, e.g., 2/3 training and 1/3 test set. If there is just one iteration, the strategy is commonly called *holdout* or *test sample estimation*.

You can calculate several measures at once by passing a `list` of Measures (`makeMeasure()`)s to `resample()`. Below, the error rate (`mmce` (`vignette("measures")`)), false positive and false negative rates (`fpr` (`vignette("measures")`), `fnr` (`vignette("measures")`)), and the time it takes to train the learner (`timetrain` (`vignette("measures")`)) are estimated by *subsampling* with 5 iterations.

```
## Subsampling with 5 iterations and default split ratio 2/3
rdesc = makeResampleDesc("Subsample", iters = 5)

## Subsampling with 5 iterations and 4/5 training data
rdesc = makeResampleDesc("Subsample", iters = 5, split = 4/5)

## Classification tree with information splitting criterion
lrn = makeLearner("classif.rpart", parms = list(split = "information"))

## Calculate the performance measures
r = resample(lrn, sonar.task, rdesc, measures = list(mmce, fpr, fnr, timetrain))
```

```
## Resampling: subsampling
## Measures:             mmce        fpr         fnr         timetrain
## [Resample] iter 1:    0.2142857   0.1500000   0.2727273   0.0140000
## [Resample] iter 2:    0.1904762   0.1500000   0.2272727   0.0250000
```

```
## [Resample] iter 3:    0.3571429   0.5263158   0.2173913   0.0150000
## [Resample] iter 4:    0.2142857   0.2000000   0.2272727   0.0140000
## [Resample] iter 5:    0.2380952   0.2500000   0.2333333   0.0140000
##
## Aggregated Result: mmce.test.mean=0.2428571,fpr.test.mean=0.2552632,fnr.test.mean=0.2355995,timetrai
##
```

```
r
```

```
## Resample Result
## Task: Sonar_example
## Learner: classif.rpart
## Aggr perf: mmce.test.mean=0.2428571,fpr.test.mean=0.2552632,fnr.test.mean=0.2355995,timetrain.test.m
## Runtime: 0.145206
```

If you want to add further measures afterwards, use `addRRMeasure()`.

```
## Add balanced error rate (ber) and time used to predict
addRRMeasure(r, list(ber, timepredict))
```

```
## Resample Result
## Task: Sonar_example
## Learner: classif.rpart
## Aggr perf: mmce.test.mean=0.2428571,fpr.test.mean=0.2552632,fnr.test.mean=0.2355995,timetrain.test.m
## Runtime: 0.145206
```

By default, `resample()` prints progress messages and intermediate results. You can turn this off by setting `show.info = FALSE`, as done in the code chunk below. (If you are interested in suppressing these messages permanently have a look at the tutorial page about configuring mlr (`vignette("configureMlr")`).)

In the above example, the Learner (`makeLearner()`) was explicitly constructed. For convenience you can also specify the learner as a string and pass any learner parameters via the `...` argument of `resample()`.

```
r = resample("classif.rpart", parms = list(split = "information"), sonar.task, rdesc,
  measures = list(mmce, fpr, fnr, timetrain), show.info = FALSE)

r
```

```
## Resample Result
## Task: Sonar_example
## Learner: classif.rpart
## Aggr perf: mmce.test.mean=0.2761905,fpr.test.mean=0.3101520,fnr.test.mean=0.2786567,timetrain.test.m
## Runtime: 0.150011
```

6.3 Accessing resample results

Apart from the learner performance you can extract further information from the resample results, for example predicted values or the models fitted in individual resample iterations.

6.3.1 Predictions

Per default, the `resample()` result contains the predictions made during the resampling. If you do not want to keep them, e.g., in order to conserve memory, set `keep.pred = FALSE` when calling `resample()`.

The predictions are stored in slot `$pred` of the resampling result, which can also be accessed by function `getRRPredictions()`.

```
r$pred
```

```
## Resampled Prediction for:
## Resample description: subsampling with 5 iterations and 0.80 split rate.
## Predict: test
## Stratification: FALSE
## predict.type: response
## threshold:
## time (mean): 0.00
##    id truth response iter  set
## 1 169     M        M    1 test
## 2  74     R        M    1 test
## 3  64     R        R    1 test
## 4 111     M        M    1 test
## 5 129     M        M    1 test
## 6  14     R        M    1 test
## ... (#rows: 210, #cols: 5)
```

```
pred = getRRPredictions(r)
pred
```

```
## Resampled Prediction for:
## Resample description: subsampling with 5 iterations and 0.80 split rate.
## Predict: test
## Stratification: FALSE
## predict.type: response
## threshold:
## time (mean): 0.00
##    id truth response iter  set
## 1 169     M        M    1 test
## 2  74     R        M    1 test
## 3  64     R        R    1 test
## 4 111     M        M    1 test
## 5 129     M        M    1 test
## 6  14     R        M    1 test
## ... (#rows: 210, #cols: 5)
```

`pred` is an object of class `resample()` Prediction. Just as a `Prediction()` object (see the tutorial page on making predictions (`vignette("predict")`)) it has an element `$data` which is a `data.frame` that contains the predictions and in the case of a supervised learning problem the true values of the target variable(s). You can use `as.data.frame` (`Prediction()` to directly access the `$data` slot. Moreover, all getter functions for `Prediction()` objects like `getPredictionResponse()` or `getPredictionProbabilities()` are applicable.

```
head(as.data.frame(pred))
```

```
##    id truth response iter  set
## 1 169     M        M    1 test
## 2  74     R        M    1 test
## 3  64     R        R    1 test
## 4 111     M        M    1 test
## 5 129     M        M    1 test
## 6  14     R        M    1 test
```

```
head(getPredictionTruth(pred))
```

```
## [1] M R R M M R
## Levels: M R
```

```
head(getPredictionResponse(pred))
```

```
## [1] M M R M M M
## Levels: M R
```

The columns `iter` and `set` in the `data.frame` indicate the resampling iteration and the data set (`train` or `test`) for which the prediction was made.

By default, predictions are made for the test sets only. If predictions for the training set are required, set `predict = "train"` (for predictions on the train set only) or `predict = "both"` (for predictions on both train and test sets) in `makeResampleDesc()`. In any case, this is necessary for some bootstrap methods (*b632* and *b632+*) and some examples are shown later on.

Below, we use simple Holdout, i.e., split the data once into a training and test set, as resampling strategy and make predictions on both sets.

```
## Make predictions on both training and test sets
rdesc = makeResampleDesc("Holdout", predict = "both")

r = resample("classif.lda", iris.task, rdesc, show.info = FALSE)
r
```

```
## Resample Result
## Task: iris_example
## Learner: classif.lda
## Aggr perf: mmce.test.mean=0.0400000
## Runtime: 0.0117452
```

```
r$measures.train
```

```
##   iter mmce
## 1    1 0.02
```

(Please note that nonetheless the misclassification rate `r$aggr` is estimated on the test data only. How to calculate performance measures on the training sets is shown below.)

A second function to extract predictions from resample results is `getRRPredictionList()` which returns a `list` of predictions split by data set (train/test) and resampling iteration.

```
predList = getRRPredictionList(r)
predList
```

```
## $train
## $train$`1`
## Prediction: 100 observations
## predict.type: response
## threshold:
## time: 0.00
##      id      truth   response
## 57   57 versicolor versicolor
## 142 142  virginica  virginica
## 56   56 versicolor versicolor
## 104 104  virginica  virginica
## 71   71 versicolor versicolor
```

```
## 133 133  virginica  virginica
## ... (#rows: 100, #cols: 3)
##
##
## $test
## $test$`1`
## Prediction: 50 observations
## predict.type: response
## threshold:
## time: 0.00
##      id      truth   response
## 23   23     setosa     setosa
## 38   38     setosa     setosa
## 100 100 versicolor versicolor
## 11   11     setosa     setosa
## 143 143  virginica  virginica
## 103 103  virginica  virginica
## ... (#rows: 50, #cols: 3)
```

6.3.2 Learner models

In each resampling iteration a Learner (`makeLearner()`) is fitted on the respective training set. By default, the resulting `WrappedModel` (`makeWrappedModel()`)s are not included in the `resample()` result and slot `$models` is empty. In order to keep them, set `models = TRUE` when calling `resample()`, as in the following survival analysis example.

```
## 3-fold cross-validation
rdesc = makeResampleDesc("CV", iters = 3)

r = resample("surv.coxph", lung.task, rdesc, show.info = FALSE, models = TRUE)
r$models
```

```
## [[1]]
## Model for learner.id=surv.coxph; learner.class=surv.coxph
## Trained on: task.id = lung-example; obs = 112; features = 8
## Hyperparameters:
##
## [[2]]
## Model for learner.id=surv.coxph; learner.class=surv.coxph
## Trained on: task.id = lung-example; obs = 111; features = 8
## Hyperparameters:
##
## [[3]]
## Model for learner.id=surv.coxph; learner.class=surv.coxph
## Trained on: task.id = lung-example; obs = 111; features = 8
## Hyperparameters:
```

6.3.3 The extract option

Keeping complete fitted models can be memory-intensive if these objects are large or the number of resampling iterations is high. Alternatively, you can use the `extract` argument of `resample()` to retain only the information you need. To this end you need to pass a function to `extract` which is applied to each `WrappedModel` (`makeWrappedModel()`) object fitted in each resampling iteration.

Below, we cluster the `datasets::mtcars()` data using the k-means algorithm with $k = 3$ and keep only the cluster centers.

```
## 3-fold cross-validation
rdesc = makeResampleDesc("CV", iters = 3)

## Extract the compute cluster centers
r = resample("cluster.kmeans", mtcars.task, rdesc, show.info = FALSE,
  centers = 3, extract = function(x) getLearnerModel(x)$centers)
r$extract
```

```
## [[1]]
##        mpg      cyl     disp  hp     drat       wt     qsec        vs
## 1 14.55714 8.000000 355.9429 205 3.142857 4.189286 16.93143 0.0000000
## 2 24.19231 4.769231 138.4538  97 3.848462 2.601769 18.69846 0.7692308
## 3 15.00000 8.000000 301.0000 335 3.540000 3.570000 14.60000 0.0000000
##          am     gear     carb
## 1 0.0000000 3.000000 3.428571
## 2 0.6153846 4.076923 2.230769
## 3 1.0000000 5.000000 8.000000
##
## [[2]]
##        mpg      cyl     disp        hp     drat       wt     qsec
## 1 24.32727 4.545455 119.5000  94.81818 4.040909 2.464545 18.31818
## 2 16.26000 7.600000 279.7200 153.00000 2.962000 3.585000 17.99800
## 3 14.91667 8.000000 390.6667 228.16667 3.320000 4.116500 16.49500
##          vs        am     gear     carb
## 1 0.7272727 0.7272727 4.090909 2.363636
## 2 0.2000000 0.0000000 3.000000 2.200000
## 3 0.0000000 0.3333333 3.666667 4.000000
##
## [[3]]
##     mpg cyl     disp       hp     drat       wt     qsec   vs        am
## 1 24.14 4.8 128.2000 101.1000 3.944000 2.631800 18.84600 0.80 0.6000000
## 2 17.05 7.5 278.4000 155.0000 3.092500 3.625000 18.03500 0.25 0.0000000
## 3 15.20 8.0 388.7143 221.2857 3.374286 4.090571 16.43714 0.00 0.1428571
##       gear     carb
## 1 4.000000 2.500000
## 2 3.000000 2.250000
## 3 3.285714 3.428571
```

As a second example, we extract the variable importances from fitted regression trees using function `getFeatureImportance()`. (For more detailed information on this topic see the feature selection (`vignette("feature_selection")`) page.)

```
## Extract the variable importance in a regression tree
r = resample("regr.rpart", bh.task, rdesc, show.info = FALSE, extract = getFeatureImportance)
r$extract
```

```
## [[1]]
## FeatureImportance:
## Task: BostonHousing-example
##
## Learner: regr.rpart
## Measure: NA
## Contrast: NA
```

```
## Aggregation: function (x)  x
## Replace: NA
## Number of Monte-Carlo iterations: NA
## Local: FALSE
##       crim       zn    indus chas      nox      rm      age      dis
## 1 1624.607 1154.658 3027.455    0 3589.014 15167.3 3244.105 3460.386
##        rad      tax  ptratio        b    lstat
## 1 723.4146 3743.818 2195.438 490.4226 10754.31
##
## [[2]]
## FeatureImportance:
## Task: BostonHousing-example
##
## Learner: regr.rpart
## Measure: NA
## Contrast: NA
## Aggregation: function (x)  x
## Replace: NA
## Number of Monte-Carlo iterations: NA
## Local: FALSE
##       crim       zn    indus chas      nox       rm      age      dis
## 1 6492.919 128.6643 7812.346    0 7275.538 13318.33 6918.349 1278.763
##        rad      tax  ptratio b    lstat
## 1 200.4173 819.9258 1248.917 0 16321.67
##
## [[3]]
## FeatureImportance:
## Task: BostonHousing-example
##
## Learner: regr.rpart
## Measure: NA
## Contrast: NA
## Aggregation: function (x)  x
## Replace: NA
## Number of Monte-Carlo iterations: NA
## Local: FALSE
##       crim       zn    indus chas      nox       rm      age      dis
## 1 3611.777 6387.452 9189.073    0 8505.042 14691.76 7030.925 2200.776
##        rad      tax  ptratio        b    lstat
## 1 2866.666 2742.564 329.0868 662.2103 16749.87
```

6.4 Stratification and blocking

- *Stratification* with respect to a categorical variable makes sure that all its values are present in each training and test set in approximately the same proportion as in the original data set. Stratification is possible with regard to categorical target variables (and thus for supervised classification and survival analysis) or categorical explanatory variables.
- *Blocking* refers to the situation that subsets of observations belong together and must not be separated during resampling. Hence, for one train/test set pair the entire block is either in the training set or in the test set.

6.4.1 Stratification with respect to the target variable(s)

For classification, it is usually desirable to have the same proportion of the classes in all of the partitions of the original data set. This is particularly useful in the case of imbalanced classes and small data sets. Otherwise, it may happen that observations of less frequent classes are missing in some of the training sets which can decrease the performance of the learner, or lead to model crashes. In order to conduct stratified resampling, set `stratify = TRUE` in `makeResampleDesc()`.

```
## 3-fold cross-validation
rdesc = makeResampleDesc("CV", iters = 3, stratify = TRUE)

r = resample("classif.lda", iris.task, rdesc, show.info = FALSE)
r
```

```
## Resample Result
## Task: iris_example
## Learner: classif.lda
## Aggr perf: mmce.test.mean=0.0200053
## Runtime: 0.0239542
```

Stratification is also available for survival tasks. Here the stratification balances the censoring rate.

6.4.2 Stratification with respect to explanatory variables

Sometimes it is required to also stratify on the input data, e.g., to ensure that all subgroups are represented in all training and test sets. To stratify on the input columns, specify `factor` columns of your task data via `stratify.cols`.

```
rdesc = makeResampleDesc("CV", iters = 3, stratify.cols = "chas")

r = resample("regr.rpart", bh.task, rdesc, show.info = FALSE)
r
```

```
## Resample Result
## Task: BostonHousing-example
## Learner: regr.rpart
## Aggr perf: mse.test.mean=24.0319858
## Runtime: 0.0371411
```

6.4.3 Blocking

If some observations "belong together" and must not be separated when splitting the data into training and test sets for resampling, you can supply this information via a `blocking factor` when creating the task (`vignette("task")`).

```
## 5 blocks containing 30 observations each
task = makeClassifTask(data = iris, target = "Species", blocking = factor(rep(1:5, each = 30)))
task
```

```
## Supervised task: iris
## Type: classif
## Target: Species
## Observations: 150
## Features:
##    numerics     factors     ordered functionals
##           4           0           0           0
```

```
## Missings: FALSE
## Has weights: FALSE
## Has blocking: TRUE
## Is spatial: FALSE
## Classes: 3
##     setosa versicolor  virginica
##         50         50         50
## Positive class: NA
```

6.5 Resample descriptions and resample instances

As already mentioned, you can specify a resampling strategy using function `makeResampleDesc()`.

```
rdesc = makeResampleDesc("CV", iters = 3)
rdesc
```

```
## Resample description: cross-validation with 3 iterations.
## Predict: test
## Stratification: FALSE
```

```
str(rdesc)
```

```
## List of 4
##  $ id      : chr "cross-validation"
##  $ iters   : int 3
##  $ predict : chr "test"
##  $ stratify: logi FALSE
##  - attr(*, "class")= chr [1:2] "CVDesc" "ResampleDesc"
```

```
str(makeResampleDesc("Subsample", stratify.cols = "chas"))
```

```
## List of 6
##  $ split        : num 0.667
##  $ id           : chr "subsampling"
##  $ iters        : int 30
##  $ predict      : chr "test"
##  $ stratify     : logi FALSE
##  $ stratify.cols: chr "chas"
##  - attr(*, "class")= chr [1:2] "SubsampleDesc" "ResampleDesc"
```

The result `rdesc` inherits from class `ResampleDesc` (`makeResampleDesc()`) (short for resample description) and, in principle, contains all necessary information about the resampling strategy including the number of iterations, the proportion of training and test sets, stratification variables, etc.

Given either the size of the data set at hand or the `Task()`, function `makeResampleInstance()` draws the training and test sets according to the `ResampleDesc` (`makeResampleDesc()`).

```
## Create a resample instance based an a task
rin = makeResampleInstance(rdesc, iris.task)
rin
```

```
## Resample instance for 150 cases.
## Resample description: cross-validation with 3 iterations.
## Predict: test
## Stratification: FALSE
```

```
str(rin)
```

```
## List of 5
##  $ desc      :List of 4
##   ..$ id      : chr "cross-validation"
##   ..$ iters   : int 3
##   ..$ predict : chr "test"
##   ..$ stratify: logi FALSE
##   ..- attr(*, "class")= chr [1:2] "CVDesc" "ResampleDesc"
##  $ size      : int 150
##  $ train.inds:List of 3
##   ..$ : int [1:100] 34 103 55 117 135 120 97 75 124 100 ...
##   ..$ : int [1:100] 34 24 114 103 55 117 135 65 15 51 ...
##   ..$ : int [1:100] 24 114 120 97 75 124 100 41 15 51 ...
##  $ test.inds :List of 3
##   ..$ : int [1:50] 1 8 10 11 12 13 15 16 21 22 ...
##   ..$ : int [1:50] 4 5 7 17 18 27 31 32 38 41 ...
##   ..$ : int [1:50] 2 3 6 9 14 19 20 25 28 30 ...
##  $ group     : Factor w/ 0 levels:
##  - attr(*, "class")= chr "ResampleInstance"
```

```
## Create a resample instance given the size of the data set
rin = makeResampleInstance(rdesc, size = nrow(iris))
str(rin)
```

```
## List of 5
##  $ desc      :List of 4
##   ..$ id      : chr "cross-validation"
##   ..$ iters   : int 3
##   ..$ predict : chr "test"
##   ..$ stratify: logi FALSE
##   ..- attr(*, "class")= chr [1:2] "CVDesc" "ResampleDesc"
##  $ size      : int 150
##  $ train.inds:List of 3
##   ..$ : int [1:100] 146 134 12 37 122 89 49 100 16 46 ...
##   ..$ : int [1:100] 138 95 134 6 136 67 65 43 122 49 ...
##   ..$ : int [1:100] 138 146 95 12 6 136 67 65 37 43 ...
##  $ test.inds :List of 3
##   ..$ : int [1:50] 2 3 6 8 9 11 14 20 24 26 ...
##   ..$ : int [1:50] 5 7 10 12 16 21 22 27 29 31 ...
##   ..$ : int [1:50] 1 4 13 15 17 18 19 23 25 28 ...
##  $ group     : Factor w/ 0 levels:
##  - attr(*, "class")= chr "ResampleInstance"
```

```
## Access the indices of the training observations in iteration 3
rin$train.inds[[3]]
```

```
##   [1] 138 146  95  12   6 136  67  65  37  43  89 100  36  16  52  47   2
##  [18]  73  60  61 130 148   5 132 115  29 128 147 103 141  94   9  51  24
##  [35]  44  50 144 120  77  32 114  75 118 123 133  78 108 135 110  72  55
##  [52]  82 139 119   7 117 129  66  22  27  58  63 105  31  10  64  96  20
##  [69] 101 109  59  56  74 142  14  26  98  54   8  38 102  68  86 126  79
##  [86]   3  91  33  35  21  57  99  83  41  85 111 125  48 143  11
```

The result `rin` inherits from class `ResampleInstance` (`makeResampleInstance()`) and contains `lists` of index vectors for the train and test sets.

If a `ResampleDesc` (`makeResampleDesc()`) is passed to `resample()`, it is instantiated internally. Naturally, it is also possible to pass a `ResampleInstance` (`makeResampleInstance()`) directly.

While the separation between resample descriptions, resample instances, and the `resample()` function itself seems overly complicated, it has several advantages:

- Resample instances readily allow for paired experiments, that is comparing the performance of several learners on exactly the same training and test sets. This is particularly useful if you want to add another method to a comparison experiment you already did. Moreover, you can store the resample instance along with your data in order to be able to reproduce your results later on.

```
rdesc = makeResampleDesc("CV", iters = 3)
rin = makeResampleInstance(rdesc, task = iris.task)

## Calculate the performance of two learners based on the same resample instance
r.lda = resample("classif.lda", iris.task, rin, show.info = FALSE)
r.rpart = resample("classif.rpart", iris.task, rin, show.info = FALSE)
r.lda$aggr
```

```
## mmce.test.mean
##           0.02
```

```
r.rpart$aggr
```

```
## mmce.test.mean
##     0.05333333
```

- In order to add further resampling methods you can simply derive from the `ResampleDesc` (`makeResampleDesc()`) and `ResampleInstance` (`makeResampleInstance()`) classes, but you do neither have to touch `resample()` nor any further methods that use the resampling strategy.

Usually, when calling `makeResampleInstance()` the train and test index sets are drawn randomly. Mainly for *holdout* (*test sample*) *estimation* you might want full control about the training and tests set and specify them manually. This can be done using function `makeFixedHoldoutInstance()`.

```
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
```

6.6 Aggregating performance values

In each resampling iteration $b = 1, \ldots, B$ we get performance values $S(D^{*b}, D \setminus D^{*b})$ (for each measure we wish to calculate), which are then aggregated to an overall performance.

For the great majority of common resampling strategies (like holdout, cross-validation, subsampling) performance values are calculated on the test data sets only and for most measures aggregated by taking the mean (`test.mean(aggregations())`).

Each performance `Measure` (`makeMeasure()`) in `mlr` has a corresponding default aggregation method which is stored in slot `$aggr`. The default aggregation for most measures is `test.mean(aggregations())`. One exception is the root mean square error (`rmse` (`vignette("measures")`)).

```
## Mean misclassification error
mmce$aggr
```

```
## Aggregation function: test.mean
```

```
mmce$aggr$fun
```

```
## function(task, perf.test, perf.train, measure, group, pred) mean(perf.test)
## <bytecode: 0x560e44a168c8>
## <environment: namespace:mlr>
```

```
## Root mean square error
rmse$aggr
```

```
## Aggregation function: test.rmse
```

```
rmse$aggr$fun
```

```
## function(task, perf.test, perf.train, measure, group, pred) sqrt(mean(perf.test^2))
## <bytecode: 0x560e47541410>
## <environment: namespace:mlr>
```

You can change the aggregation method of a `Measure` (`makeMeasure()`) via function `setAggregation()`. All available aggregation schemes are listed on the `aggregations()` documentation page.

6.6.1 Example: One measure with different aggregations

The aggregation schemes `test.median` (`aggregations()`), `test.min` (`aggregations()`), and `text.max` (`aggregations()`) compute the median, minimum, and maximum of the performance values on the test sets.

```
mseTestMedian = setAggregation(mse, test.median)
mseTestMin = setAggregation(mse, test.min)
mseTestMax = setAggregation(mse, test.max)

mseTestMedian
```

```
## Name: Mean of squared errors
## Performance measure: mse
## Properties: regr,req.pred,req.truth
## Minimize: TRUE
## Best: 0; Worst: Inf
## Aggregated by: test.median
## Arguments:
## Note: Defined as: mean((response - truth)^2)
```

```
rdesc = makeResampleDesc("CV", iters = 3)
r = resample("regr.lm", bh.task, rdesc, measures = list(mse, mseTestMedian, mseTestMin, mseTestMax))
```

```
## Resampling: cross-validation
## Measures:             mse       mse       mse       mse
## [Resample] iter 1:    23.821507423.821507423.821507423.8215074
## [Resample] iter 2:    20.030927820.030927820.030927820.0309278
## [Resample] iter 3:    27.339803527.339803527.339803527.3398035
##
## Aggregated Result: mse.test.mean=23.7307463,mse.test.median=23.8215074,mse.test.min=20.0309278,mse.te
##
```

```
r
```

```
## Resample Result
## Task: BostonHousing-example
## Learner: regr.lm
## Aggr perf: mse.test.mean=23.7307463,mse.test.median=23.8215074,mse.test.min=20.0309278,mse.test.max=
## Runtime: 0.0411327
```

```
r$aggr
```

```
##   mse.test.mean mse.test.median    mse.test.min    mse.test.max
##        23.73075        23.82151        20.03093        27.33980
```

6.6.2 Example: Calculating the training error

Below we calculate the mean misclassification error (`mmce` (`vignette("measures")`)) on the training and the test data sets. Note that we have to set `predict = "both"` when calling `makeResampleDesc()` in order to get predictions on both training and test sets.

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

```
## Resampling: cross-validation
## Measures:             mmce.train   mmce.test
## [Resample] iter 1:    0.0400000    0.0200000
## [Resample] iter 2:    0.0400000    0.0600000
## [Resample] iter 3:    0.0000000    0.1200000
##
## Aggregated Result: mmce.test.mean=0.0666667,mmce.train.mean=0.0266667
##
```

```
r$measures.train
```

```
##   iter mmce mmce
## 1    1 0.04 0.04
## 2    2 0.04 0.04
## 3    3 0.00 0.00
```

```
r$aggr
```

```
##  mmce.test.mean mmce.train.mean
##      0.06666667      0.02666667
```

6.6.3 Example: Bootstrap

In *out-of-bag bootstrap estimation* B new data sets $D^{*1}, \ldots, D^{*B}$ are drawn from the data set D with replacement, each of the same size as D. In the b-th iteration, D^{*b} forms the training set, while the remaining elements from D, i.e., $D \setminus D^{*b}$, form the test set.

The *b632* and *b632+* variants calculate a convex combination of the training performance and the out-of-bag bootstrap performance and thus require predictions on the training sets and an appropriate aggregation strategy.

```
## Use bootstrap as resampling strategy and predict on both train and test sets
rdesc = makeResampleDesc("Bootstrap", predict = "both", iters = 10)

## Set aggregation schemes for b632 and b632+ bootstrap
mmceB632 = setAggregation(mmce, b632)
mmceB632plus = setAggregation(mmce, b632plus)

mmceB632
```

```
## Name: Mean misclassification error
## Performance measure: mmce
## Properties: classif,classif.multi,req.pred,req.truth
## Minimize: TRUE
## Best: 0; Worst: 1
## Aggregated by: b632
## Arguments:
## Note: Defined as: mean(response != truth)
```

```
r = resample("classif.rpart", iris.task, rdesc, measures = list(mmce, mmceB632, mmceB632plus),
  show.info = FALSE)
head(r$measures.train)
```

```
##   iter       mmce       mmce       mmce
## 1    1 0.05333333 0.05333333 0.05333333
## 2    2 0.01333333 0.01333333 0.01333333
## 3    3 0.04666667 0.04666667 0.04666667
## 4    4 0.04000000 0.04000000 0.04000000
## 5    5 0.02666667 0.02666667 0.02666667
## 6    6 0.04666667 0.04666667 0.04666667
```

```
## Compare misclassification rates for out-of-bag, b632, and b632+ bootstrap
r$aggr
```

```
## mmce.test.mean      mmce.b632  mmce.b632plus
##     0.06683768     0.05303608     0.05414469
```

6.7 Convenience functions

The functionality described on this page allows for much control and flexibility. However, when quickly trying out some learners, it can get tedious to type all the code for defining the resampling strategy, setting the aggregation scheme and so on. As mentioned above, `mlr` includes some pre-defined resample description objects for frequently used strategies like, e.g., 5-fold cross-validation (`cv5` (`makeResampleDesc()`)). Moreover, `mlr` provides special functions for the most common resampling methods, for example `holdout` (`resample()`), `crossval` (`resample()`), or `bootstrapB632` (`resample()`).

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

```
## Resampling: cross-validation
## Measures:             mmce      ber
## [Resample] iter 1:    0.0200000 0.0208333
## [Resample] iter 2:    0.0000000 0.0000000
## [Resample] iter 3:    0.0400000 0.0393519
##
```

```
## Aggregated Result: mmce.test.mean=0.0200000,ber.test.mean=0.0200617
##
## Resample Result
## Task: iris_example
## Learner: classif.lda
## Aggr perf: mmce.test.mean=0.0200000,ber.test.mean=0.0200617
## Runtime: 0.0301597
```

```
bootstrapB632plus("regr.lm", bh.task, iters = 3, measures = list(mse, mae))
```

```
## Resampling: OOB bootstrapping
## Measures:             mse.train   mae.train   mse.test    mae.test
## [Resample] iter 1:    19.4242311  3.2529975   35.0709375  3.8405890
## [Resample] iter 2:    22.1983068  3.3328420   22.1964153  3.3298815
## [Resample] iter 3:    21.3800268  3.2207697   22.4788296  3.4005261
##
## Aggregated Result: mse.b632plus=24.6474199,mae.b632plus=3.4341243
##
## Resample Result
## Task: BostonHousing-example
## Learner: regr.lm
## Aggr perf: mse.b632plus=24.6474199,mae.b632plus=3.4341243
## Runtime: 0.0501359
```

7 Tuning Hyperparameters

Many machine learning algorithms have hyperparameters that need to be set. If selected by the user they can be specified as explained on the tutorial page on learners (`vignette("learners")`) – simply pass them to `makeLearner()`. Often suitable parameter values are not obvious and it is preferable to tune the hyperparameters, that is automatically identify values that lead to the best performance.

In order to tune a machine learning algorithm, you have to specify:

- the search space
- the optimization algorithm (aka tuning method)
- an evaluation method, i.e., a resampling strategy and a performance measure

An example of the search space could be searching values of the `C` parameter for `kernlab::ksvm()`:

```
## ex: create a search space for the C hyperparameter from 0.01 to 0.1
ps = makeParamSet(
  makeNumericParam("C", lower = 0.01, upper = 0.1)
)
```

An example of the optimization algorithm could be performing random search on the space:

```
## ex: random search with 100 iterations
ctrl = makeTuneControlRandom(maxit = 100L)
```

An example of an evaluation method could be 3-fold CV using accuracy as the performance measure:

```
rdesc = makeResampleDesc("CV", iters = 3L)
measure = acc
```

The evaluation method is already covered in detail in evaluation of learning methods (`vignette("performance")`) and resampling (`vignette("resample")`).

In this tutorial, we show how to specify the search space and optimization algorithm, how to do the tuning and how to access the tuning result, and how to visualize the hyperparameter tuning effects through several examples.

Throughout this section we consider classification examples. For the other types of learning problems, you can follow the same process analogously.

We use the iris classification task (`iris.task()`) for illustration and tune the hyperparameters of an SVM (function `kernlab::ksvm()`) from the `kernlab` package) with a radial basis kernel. The following examples tune the cost parameter `C` and the RBF kernel parameter `sigma` of the `kernlab::ksvm()`) function.

7.1 Specifying the search space

We first must define a space to search when tuning our learner. For example, maybe we want to tune several specific values of a hyperparameter or perhaps we want to define a space from 10^{-10} to 10^{10} and let the optimization algorithm decide which points to choose.

In order to define a search space, we create a `ParamSet` (`ParamHelpers::makeParamSet()`) object, which describes the parameter space we wish to search. This is done via the function `ParamHelpers::makeParamSet()`.

For example, we could define a search space with just the values 0.5, 1.0, 1.5, 2.0 for both `C` and `gamma`. Notice how we name each parameter as it's defined in the `kernlab` package:

```
discrete_ps = makeParamSet(
  makeDiscreteParam("C", values = c(0.5, 1.0, 1.5, 2.0)),
  makeDiscreteParam("sigma", values = c(0.5, 1.0, 1.5, 2.0))
)
print(discrete_ps)
```

```
##           Type len Def      Constr Req Tunable Trafo
## C     discrete   -   - 0.5,1,1.5,2   -    TRUE     -
## sigma discrete   -   - 0.5,1,1.5,2   -    TRUE     -
```

We could also define a continuous search space (using `makeNumericParam` (`ParamHelpers::makeNumericParam()`) instead of `makeDiscreteParam` (`ParamHelpers::makeDiscreteParam()`)) from 10^{-10} to 10^{10} for both parameters through the use of the `trafo` argument (trafo is short for transformation). Transformations work like this: All optimizers basically see the parameters on their original scale (from -10 to 10 in this case) and produce values on this scale during the search. Right before they are passed to the learning algorithm, the transformation function is applied.

Notice this time we use `makeNumericParam` (`ParamHelpers::makeNumericParam()`):

```
num_ps = makeParamSet(
  makeNumericParam("C", lower = -10, upper = 10, trafo = function(x) 10^x),
  makeNumericParam("sigma", lower = -10, upper = 10, trafo = function(x) 10^x)
)
```

Many other parameters can be created, check out the examples in `ParamHelpers::makeParamSet()`.

In order to standardize your workflow across several packages, whenever parameters in the underlying **R** functions should be passed in a `list` structure, `mlr` tries to give you direct access to each parameter and get rid of the list structure!

This is the case with the `kpar` argument of `kernlab::ksvm()`) which is a list of kernel parameters like `sigma`. This allows us to interface with learners from different packages in the same way when defining parameters to tune!

7.2 Specifying the optimization algorithm

Now that we have specified the search space, we need to choose an optimization algorithm for our parameters to pass to the `kernlab::ksvm()`) learner. Optimization algorithms are considered `TuneControl()` objects in `mlr`.

A grid search is one of the standard – albeit slow – ways to choose an appropriate set of parameters from a given search space.

In the case of `discrete_ps` above, since we have manually specified the values, grid search will simply be the cross product. We create the grid search object using the defaults, noting that we will have $4 \times 4 = 16$ combinations in the case of `discrete_ps`:

```
ctrl = makeTuneControlGrid()
```

In the case of `num_ps` above, since we have only specified the upper and lower bounds for the search space, grid search will create a grid using equally-sized steps. By default, grid search will span the space in 10 equal-sized steps. The number of steps can be changed with the `resolution` argument. Here we change to 15 equal-sized steps in the space defined within the `ParamSet` (`ParamHelpers::makeParamSet()`) object. For `num_ps`, this means 15 steps in the form of `10 ^ seq(-10, 10, length.out = 15)`:

```
ctrl = makeTuneControlGrid(resolution = 15L)
```

Many other types of optimization algorithms are available. Check out `TuneControl()` for some examples.

Since grid search is normally too slow in practice, we'll also examine random search. In the case of `discrete_ps`, random search will randomly choose from the specified values. The `maxit` argument controls the amount of iterations.

```
ctrl = makeTuneControlRandom(maxit = 10L)
```

In the case of `num_ps`, random search will randomly choose points within the space according to the specified bounds. Perhaps in this case we would want to increase the amount of iterations to ensure we adequately cover the space:

```
ctrl = makeTuneControlRandom(maxit = 200L)
```

7.3 Performing the tuning

Now that we have specified a search space and the optimization algorithm, it's time to perform the tuning. We will need to define a resampling strategy and make note of our performance measure.

We will use 3-fold cross-validation to assess the quality of a specific parameter setting. For this we need to create a resampling description just like in the resampling (`vignette("resample")`) part of the tutorial.

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

Finally, by combining all the previous pieces, we can tune the SVM parameters by calling `tuneParams()`. We will use `discrete_ps` with grid search:

```
discrete_ps = makeParamSet(
  makeDiscreteParam("C", values = c(0.5, 1.0, 1.5, 2.0)),
  makeDiscreteParam("sigma", values = c(0.5, 1.0, 1.5, 2.0))
)
```

```
ctrl = makeTuneControlGrid()
rdesc = makeResampleDesc("CV", iters = 3L)
res = tuneParams("classif.ksvm", task = iris.task, resampling = rdesc,
  par.set = discrete_ps, control = ctrl)
```

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

```
##           Type len Def      Constr Req Tunable Trafo
## C     discrete   -   - 0.5,1,1.5,2   -    TRUE     -
## sigma discrete   -   - 0.5,1,1.5,2   -    TRUE     -
```

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

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

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

```
## [Tune-y] 1: mmce.test.mean=0.0533333; time: 0.0 min
```

```
## [Tune-x] 2: C=1; sigma=0.5
```

```
## [Tune-y] 2: mmce.test.mean=0.0466667; time: 0.0 min
```

```
## [Tune-x] 3: C=1.5; sigma=0.5
```

```
## [Tune-y] 3: mmce.test.mean=0.0400000; time: 0.0 min
```

```
## [Tune-x] 4: C=2; sigma=0.5
```

```
## [Tune-y] 4: mmce.test.mean=0.0400000; time: 0.0 min
```

```
## [Tune-x] 5: C=0.5; sigma=1
```

```
## [Tune-y] 5: mmce.test.mean=0.0533333; time: 0.0 min
```

```
## [Tune-x] 6: C=1; sigma=1
```

```
## [Tune-y] 6: mmce.test.mean=0.0533333; time: 0.0 min
```

```
## [Tune-x] 7: C=1.5; sigma=1
```

```
## [Tune-y] 7: mmce.test.mean=0.0600000; time: 0.0 min
```

```
## [Tune-x] 8: C=2; sigma=1
```

```
## [Tune-y] 8: mmce.test.mean=0.0600000; time: 0.0 min
```

```
## [Tune-x] 9: C=0.5; sigma=1.5
```

```
## [Tune-y] 9: mmce.test.mean=0.0533333; time: 0.0 min
```

```
## [Tune-x] 10: C=1; sigma=1.5
```

```
## [Tune-y] 10: mmce.test.mean=0.0666667; time: 0.0 min
```

```
## [Tune-x] 11: C=1.5; sigma=1.5
```

```
## [Tune-y] 11: mmce.test.mean=0.0600000; time: 0.0 min
```

```
## [Tune-x] 12: C=2; sigma=1.5
```

```
## [Tune-y] 12: mmce.test.mean=0.0533333; time: 0.0 min
```

```
## [Tune-x] 13: C=0.5; sigma=2
```

```
## [Tune-y] 13: mmce.test.mean=0.0600000; time: 0.0 min
```

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

```
## [Tune-y] 14: mmce.test.mean=0.0533333; time: 0.0 min
## [Tune-x] 15: C=1.5; sigma=2
## [Tune-y] 15: mmce.test.mean=0.0533333; time: 0.0 min
## [Tune-x] 16: C=2; sigma=2
## [Tune-y] 16: mmce.test.mean=0.0466667; time: 0.0 min
## [Tune] Result: C=2; sigma=0.5 : mmce.test.mean=0.0400000
```

```
res
```

```
## Tune result:
## Op. pars: C=2; sigma=0.5
## mmce.test.mean=0.0400000
```

`tuneParams()` simply performs the cross-validation for every element of the cross-product and selects the parameter setting with the best mean performance. As no performance measure was specified, by default the error rate (`mmce` (`vignette("measures")`)) is used.

Note that each measure (`makeMeasure()`) "knows" if it is minimized or maximized during tuning.

```
## error rate
mmce$minimize
```

```
## [1] TRUE
```

```
## accuracy
acc$minimize
```

```
## [1] FALSE
```

Of course, you can pass other measures and also a `list` of measures to `tuneParams()`. In the latter case the first measure is optimized during tuning, the others are simply evaluated. If you are interested in optimizing several measures simultaneously have a look at Advanced Tuning (`vignette("advanced_tune")`).

In the example below we calculate the accuracy (`acc` (`vignette("measures")`)) instead of the error rate. We use function `setAggregation()`, as described on the resampling (`vignette("resample")`) page, to additionally obtain the standard deviation of the accuracy. We also use random search with 100 iterations on the `num_set` we defined above and set `show.info` to `FALSE` to hide the output for all 100 iterations:

```
num_ps = makeParamSet(
  makeNumericParam("C", lower = -10, upper = 10, trafo = function(x) 10^x),
  makeNumericParam("sigma", lower = -10, upper = 10, trafo = function(x) 10^x)
)
ctrl = makeTuneControlRandom(maxit = 100L)
res = tuneParams("classif.ksvm", task = iris.task, resampling = rdesc, par.set = num_ps,
  control = ctrl, measures = list(acc, setAggregation(acc, test.sd)), show.info = FALSE)
res
```

```
## Tune result:
## Op. pars: C=4.54e+06; sigma=2.82e-06
## acc.test.mean=0.9733333,acc.test.sd=0.0115470
```

7.4 Accessing the tuning result

The result object `TuneResult()` allows you to access the best found settings `$x` and their estimated performance `$y`.

```
res$x
```

```
## $C
## [1] 4536265
##
## $sigma
## [1] 2.817605e-06
```

```
res$y
```

```
## acc.test.mean   acc.test.sd
##    0.97333333    0.01154701
```

We can generate a Learner (`makeLearner()`) with optimal hyperparameter settings as follows:

```
lrn = setHyperPars(makeLearner("classif.ksvm"), par.vals = res$x)
lrn
```

```
## Learner classif.ksvm from package kernlab
## Type: classif
## Name: Support Vector Machines; Short name: ksvm
## Class: classif.ksvm
## Properties: twoclass,multiclass,numerics,factors,prob,class.weights
## Predict-Type: response
## Hyperparameters: fit=FALSE,C=4.54e+06,sigma=2.82e-06
```

Then you can proceed as usual. Here we refit and predict the learner on the complete iris (`datasets::iris()`) data set:

```
m = train(lrn, iris.task)
predict(m, 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
## ... (#rows: 150, #cols: 3)
```

But what if you wanted to inspect the other points on the search path, not just the optimal?

7.5 Investigating hyperparameter tuning effects

We can inspect all points evaluated during the search by using `generateHyperParsEffectData()`:

```
generateHyperParsEffectData(res)
```

```
## HyperParsEffectData:
## Hyperparameters: C,sigma
## Measures: acc.test.mean,acc.test.sd
## Optimizer: TuneControlRandom
## Nested CV Used: FALSE
```

```
## Snapshot of data:
##           C      sigma acc.test.mean acc.test.sd iteration exec.time
## 1  8.783712  7.4083866     0.2800000   0.0400000         1     0.047
## 2 -2.714250 -1.4224080     0.2666667   0.0305505         2     0.045
## 3  7.632347  6.7656452     0.2800000   0.0400000         3     0.043
## 4 -4.353174  4.9882368     0.2666667   0.0305505         4     0.043
## 5  8.904523  7.1379858     0.2800000   0.0400000         5     0.058
## 6 -8.980262  0.1525309     0.2666667   0.0305505         6     0.045
```

Note that the result of `generateHyperParsEffectData()` contains the parameter values *on the original scale*. In order to get the *transformed* parameter values instead, use the `trafo` argument:

```
generateHyperParsEffectData(res, trafo = TRUE)
```

```
## HyperParsEffectData:
## Hyperparameters: C,sigma
## Measures: acc.test.mean,acc.test.sd
## Optimizer: TuneControlRandom
## Nested CV Used: FALSE
## Snapshot of data:
##              C        sigma acc.test.mean acc.test.sd iteration exec.time
## 1 6.077323e+08 2.560865e+07     0.2800000   0.0400000         1     0.047
## 2 1.930855e-03 3.780872e-02     0.2666667   0.0305505         2     0.045
## 3 4.288915e+07 5.829686e+06     0.2800000   0.0400000         3     0.043
## 4 4.434306e-05 9.732778e+04     0.2666667   0.0305505         4     0.043
## 5 8.026445e+08 1.373997e+07     0.2800000   0.0400000         5     0.058
## 6 1.046497e-09 1.420793e+00     0.2666667   0.0305505         6     0.045
```

Note that we can also generate performance on the train data along with the validation/test data, as discussed on the resampling (`vignette("resample")`) tutorial page:

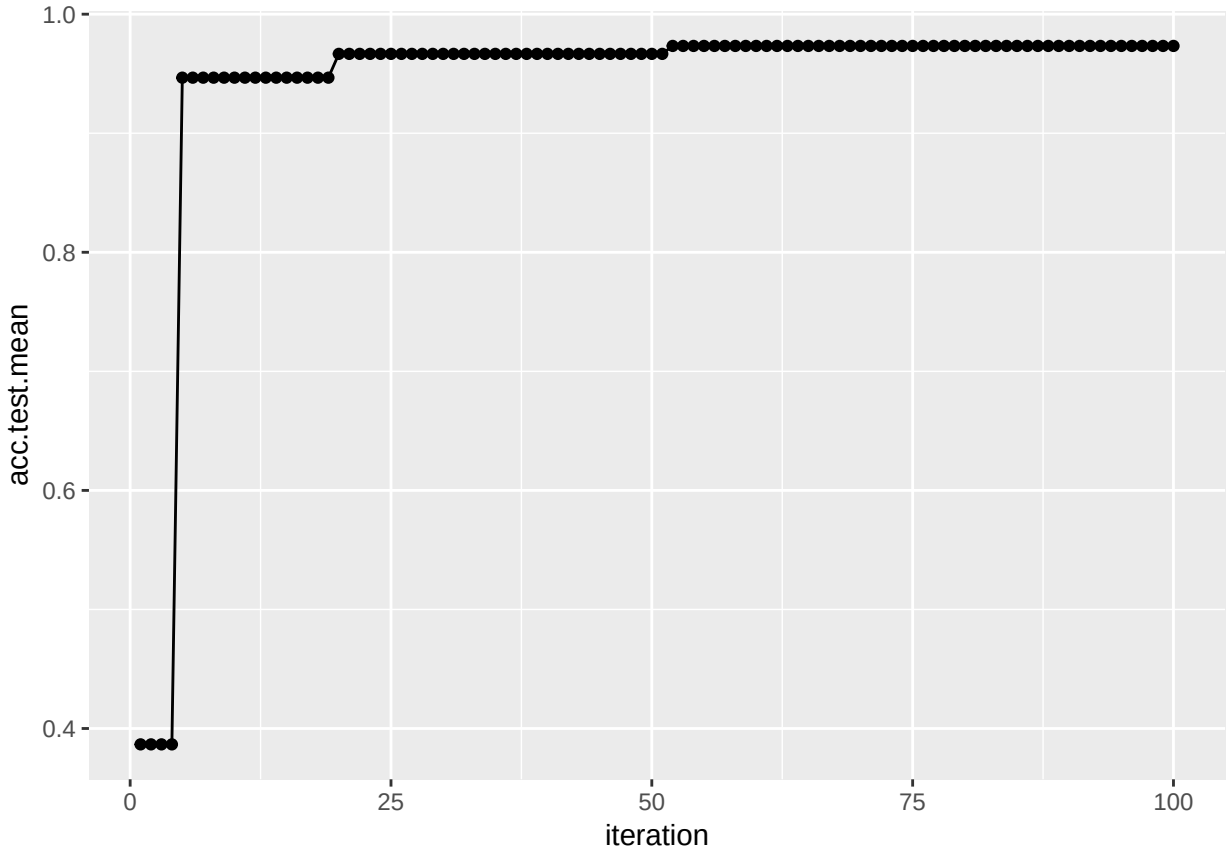
```
rdesc2 = makeResampleDesc("Holdout", predict = "both")
res2 = tuneParams("classif.ksvm", task = iris.task, resampling = rdesc2, par.set = num_ps,
  control = ctrl, measures = list(acc, setAggregation(acc, train.mean)), show.info = FALSE)
generateHyperParsEffectData(res2)
```

```
## HyperParsEffectData:
## Hyperparameters: C,sigma
## Measures: acc.test.mean,acc.train.mean
## Optimizer: TuneControlRandom
## Nested CV Used: FALSE
## Snapshot of data:
##            C        sigma acc.test.mean acc.train.mean iteration exec.time
## 1  8.9665079 -8.779976391          0.86           0.90         1     0.024
## 2 -3.8779510  3.904006165          0.22           0.39         2     0.022
## 3 -0.7284543  4.256760618          0.22           0.39         3     0.025
## 4  2.7639028 -5.833319025          0.22           0.39         4     0.032
## 5  4.5708740  0.006970754          0.92           1.00         5     0.023
## 6 -2.9305088  3.334401296          0.22           0.39         6     0.026
```

We can also easily visualize the points evaluated by using `plotHyperParsEffect()`. In the example below, we plot the performance over iterations, using the `res` from the previous section but instead with 2 performance measures:

```
res = tuneParams("classif.ksvm", task = iris.task, resampling = rdesc, par.set = num_ps,
  control = ctrl, measures = list(acc, mmce), show.info = FALSE)
data = generateHyperParsEffectData(res)
```

```
plotHyperParsEffect(data, x = "iteration", y = "acc.test.mean",
  plot.type = "line")
```



Note that by default, we only plot the current global optima. This can be changed with the `global.only` argument.

For an in-depth exploration of generating hyperparameter tuning effects and plotting the data, check out Hyperparameter Tuning Effects (`vignette("hyperpar_tuning_effects")`).

7.6 Further comments

- Tuning works for all other tasks like regression, survival analysis and so on in a completely similar fashion.
- In longer running tuning experiments it is very annoying if the computation stops due to numerical or other errors. Have a look at `on.learner.error` in `configureMlr()` as well as the examples given in section configure mlr (`vignette("configureMlr")`) of this tutorial. You might also want to inform yourself about `impute.val` in `TuneControl()`.
- As we continually optimize over the same data during tuning, the estimated performance value might be optimistically biased. A clean approach to ensure unbiased performance estimation is nested resampling (`vignette("nested_resampling")`), where we embed the whole model selection process into an outer resampling loop.

8 Benchmark Experiments

In a benchmark experiment different learning methods are applied to one or several data sets with the aim to compare and rank the algorithms with respect to one or more performance measures.

In `mlr` a benchmark experiment can be conducted by calling function `benchmark()` on a `base::list()` of `makeLearner()`s and a `base::list()` of `Task()`s. `benchmark()` basically executes `resample()` for each combination of `makeLearner()` and `Task()`. You can specify an individual resampling strategy for each `Task()` and select one or multiple performance measures to be calculated.

8.1 Conducting benchmark experiments

We start with a small example. Two learners, `MASS::lda()` and a `rpart::rpart()`, are applied to one classification problem (`sonar.task()`). As resampling strategy we choose `"Holdout"`. The performance is thus calculated on a single randomly sampled test data set.

In the example below we create a resample description (`makeResampleDesc()`), which is automatically instantiated by `benchmark()`. The instantiation is done only once per `Task()`, i.e., the same training and test sets are used for all learners. It is also possible to directly pass a `makeResampleInstance()`.

If you would like to use a *fixed test data set* instead of a randomly selected one, you can create a suitable `makeResampleInstance()` through function `makeFixedHoldoutInstance()`.

```
## Two learners to be compared
lrns = list(makeLearner("classif.lda"), makeLearner("classif.rpart"))

## Choose the resampling strategy
rdesc = makeResampleDesc("Holdout")

## Conduct the benchmark experiment
bmr = benchmark(lrns, sonar.task, rdesc)
```

```
## Task: Sonar_example, Learner: classif.lda
## Resampling: holdout
## Measures:             mmce
## [Resample] iter 1:    0.2571429
##
## Aggregated Result: mmce.test.mean=0.2571429
##
## Task: Sonar_example, Learner: classif.rpart
## Resampling: holdout
## Measures:             mmce
## [Resample] iter 1:    0.2714286
##
## Aggregated Result: mmce.test.mean=0.2714286
##
```

```
bmr
```

```
##         task.id    learner.id mmce.test.mean
## 1 Sonar_example   classif.lda      0.2571429
## 2 Sonar_example classif.rpart      0.2714286
```

For convenience, if you don't want to pass any additional arguments to `makeLearner()`, you don't need to generate the `makeLearner()`s explicitly, but it's sufficient to provide the learner name. In the above example we could also have written:

```
## Vector of strings
lrns = c("classif.lda", "classif.rpart")

## A mixed list of Learner objects and strings works, too
lrns = list(makeLearner("classif.lda", predict.type = "prob"), "classif.rpart")

bmr = benchmark(lrns, sonar.task, rdesc)
```

```
## Task: Sonar_example, Learner: classif.lda
## Resampling: holdout
## Measures:             mmce
## [Resample] iter 1:    0.3000000
##
## Aggregated Result: mmce.test.mean=0.3000000
##
## Task: Sonar_example, Learner: classif.rpart
## Resampling: holdout
## Measures:             mmce
## [Resample] iter 1:    0.3142857
##
## Aggregated Result: mmce.test.mean=0.3142857
##
```

```
bmr
```

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

In the printed summary table every row corresponds to one pair of `Task()` and `makeLearner()`. The entries show the mean misclassification error (`vignette("measures")`), the default performance measure for classification, on the test data set.

The result `bmr` is an object of class `BenchmarkResult()`. Basically, it contains a `base::list()` of lists of `ResampleResult()` objects, first ordered by `Task()` and then by `makeLearner()`.

8.1.1 Making experiments reproducible

Typically, we would want our experiment results to be reproducible. `mlr` obeys the `set.seed` function, so make sure to use `set.seed` at the beginning of your script if you would like your results to be reproducible.

Note that if you are using parallel computing, you may need to adjust how you call `set.seed` depending on your usecase. One possibility is to use `set.seed(123, "L'Ecuyer")` in order to ensure the results are

reproducible for each child process. See the examples in `parallel::mclapply()` for more information on reproducibility and parallel computing.

8.2 Accessing benchmark results

`mlr` provides several accessor functions, named `getBMR<WhatToExtract>`, that permit to retrieve information for further analyses. This includes for example the performances or predictions of the learning algorithms under consideration.

8.2.1 Learner performances

Let's have a look at the benchmark result above. `getBMRPerformances()` returns individual performances in resampling runs, while `getBMRAggrPerformances()` gives the aggregated values.

```
getBMRPerformances(bmr)
```

```
## $Sonar_example
## $Sonar_example$classif.lda
##   iter mmce
## 1    1  0.3
##
## $Sonar_example$classif.rpart
##   iter      mmce
## 1    1 0.3142857
```

```
getBMRAggrPerformances(bmr)
```

```
## $Sonar_example
## $Sonar_example$classif.lda
## mmce.test.mean
##            0.3
##
## $Sonar_example$classif.rpart
## mmce.test.mean
##      0.3142857
```

Since we used holdout as resampling strategy, individual and aggregated performance values coincide.

By default, nearly all "getter" functions return a nested `base::list()`, with the first level indicating the task and the second level indicating the learner. If only a single learner or, as in our case a single task is considered, setting `drop = TRUE` simplifies the result to a flat `base::list()`.

```
getBMRPerformances(bmr, drop = TRUE)
```

```
## $classif.lda
##   iter mmce
## 1    1  0.3
##
## $classif.rpart
##   iter      mmce
## 1    1 0.3142857
```

Often it is more convenient to work with `base::data.frame()`s. You can easily convert the result structure by setting `as.df = TRUE`.

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

```
##         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
```

8.2.2 Predictions

Per default, the `BenchmarkResult()` contains the learner predictions. If you do not want to keep them, e.g., to conserve memory, set `keep.pred = FALSE` when calling `benchmark()`.

You can access the predictions using function `getBMRPredictions()`. Per default, you get a nested `base::list()` of `ResamplePrediction()`objects. As above, you can use the `drop` or `as.df` options to simplify the result.

```
getBMRPredictions(bmr)
```

```
## $Sonar_example
## $Sonar_example$classif.lda
## Resampled Prediction for:
## Resample description: holdout with 0.67 split rate.
## Predict: test
## Stratification: FALSE
## predict.type: prob
## threshold: M=0.50,R=0.50
## time (mean): 0.00
##    id truth       prob.M    prob.R response iter  set
## 1 110     M 1.030021e-02 0.9896998        R    1 test
## 2  87     R 7.021385e-01 0.2978615        M    1 test
## 3 128     M 1.830008e-03 0.9981700        R    1 test
## 4  38     R 1.741571e-04 0.9998258        R    1 test
## 5  65     R 8.870874e-06 0.9999911        R    1 test
## 6  12     R 2.029072e-01 0.7970928        R    1 test
## ... (#rows: 70, #cols: 7)
##
## $Sonar_example$classif.rpart
## Resampled Prediction for:
## Resample description: holdout with 0.67 split rate.
## Predict: test
## Stratification: FALSE
## predict.type: response
## threshold:
## time (mean): 0.00
##    id truth response iter  set
## 1 110     M        R    1 test
## 2  87     R        R    1 test
## 3 128     M        R    1 test
## 4  38     R        R    1 test
## 5  65     R        M    1 test
```

```
## 6  12     R        R    1 test
## ... (#rows: 70, #cols: 5)
```

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

```
##         task.id  learner.id  id truth       prob.M    prob.R response iter
## 1 Sonar_example classif.lda 110     M 1.030021e-02 0.9896998        R    1
## 2 Sonar_example classif.lda  87     R 7.021385e-01 0.2978615        M    1
## 3 Sonar_example classif.lda 128     M 1.830008e-03 0.9981700        R    1
## 4 Sonar_example classif.lda  38     R 1.741571e-04 0.9998258        R    1
## 5 Sonar_example classif.lda  65     R 8.870874e-06 0.9999911        R    1
## 6 Sonar_example classif.lda  12     R 2.029072e-01 0.7970928        R    1
##    set
## 1 test
## 2 test
## 3 test
## 4 test
## 5 test
## 6 test
```

It is also easily possible to access results for certain learners or tasks via their IDs. For this purpose many "getter" functions have a `learner.ids` and a `task.ids` argument.

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

```
##         task.id    learner.id  id truth response iter  set
## 1 Sonar_example classif.rpart 110     M        R    1 test
## 2 Sonar_example classif.rpart  87     R        R    1 test
## 3 Sonar_example classif.rpart 128     M        R    1 test
## 4 Sonar_example classif.rpart  38     R        R    1 test
## 5 Sonar_example classif.rpart  65     R        M    1 test
## 6 Sonar_example classif.rpart  12     R        R    1 test
```

If you don't like the default IDs, you can set the IDs of learners and tasks via the `id` option of `makeLearner()` and `makeTask()`. Moreover, you can conveniently change the ID of a `makeLearner()` via function `setLearnerid()`.

8.2.3 IDs

The IDs of all `makeLearner()`s, `Task()`s and Measure's (`makeMeasure()`) in a benchmark experiment can be retrieved as follows:

```
getBMRTaskIds(bmr)
```

```
## [1] "Sonar_example"
```

```
getBMRLearnerIds(bmr)
```

```
## [1] "classif.lda"   "classif.rpart"
```

```
getBMRMeasureIds(bmr)
```

```
## [1] "mmce"
```

8.2.4 Fitted models

Per default the `BenchmarkResult()` also contains the fitted models for all learners on all tasks. If you do not want to keep them set `models = FALSE` when calling `benchmark()`. The fitted models can be retrieved by function `getBMRModels()`. It returns a (possibly nested) `base::list()` of `WrappedModel` (`makeWrappedModel()`) objects.

```
getBMRModels(bmr)
```

```
## $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:
##
##
## $Sonar_example$classif.rpart
## $Sonar_example$classif.rpart[[1]]
## Model for learner.id=classif.rpart; learner.class=classif.rpart
## Trained on: task.id = Sonar_example; obs = 138; features = 60
## Hyperparameters: xval=0
```

```
getBMRModels(bmr, drop = TRUE)
```

```
## $classif.lda
## $classif.lda[[1]]
## Model for learner.id=classif.lda; learner.class=classif.lda
## Trained on: task.id = Sonar_example; obs = 138; features = 60
## Hyperparameters:
##
##
## $classif.rpart
## $classif.rpart[[1]]
## Model for learner.id=classif.rpart; learner.class=classif.rpart
## Trained on: task.id = Sonar_example; obs = 138; features = 60
## Hyperparameters: xval=0
```

```
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:
```

8.2.5 Learners and measures

Moreover, you can extract the employed `makeLearner()`s and Measure's (`makeMeasure()`).

```
getBMRLearners(bmr)
```

```
## $classif.lda
## Learner classif.lda from package MASS
## Type: classif
```

```
## Name: Linear Discriminant Analysis; Short name: lda
## Class: classif.lda
## Properties: twoclass,multiclass,numerics,factors,prob
## Predict-Type: prob
## Hyperparameters:
##
##
## $classif.rpart
## Learner classif.rpart from package rpart
## Type: classif
## Name: Decision Tree; Short name: rpart
## Class: classif.rpart
## Properties: twoclass,multiclass,missings,numerics,factors,ordered,prob,weights,featimp
## Predict-Type: response
## Hyperparameters: xval=0
```

```
getBMRMeasures(bmr)
```

```
## [[1]]
## Name: Mean misclassification error
## Performance measure: mmce
## Properties: classif,classif.multi,req.pred,req.truth
## Minimize: TRUE
## Best: 0; Worst: 1
## Aggregated by: test.mean
## Arguments:
## Note: Defined as: mean(response != truth)
```

8.3 Merging benchmark results

Sometimes after completing a benchmark experiment it turns out that you want to extend it by another `makeLearner()` or another `Task()`. In this case you can perform an additional benchmark experiment and then use function `mergeBenchmarkResults()` to combine the results to a single `BenchmarkResult()` object that can be accessed and analyzed as usual.

For example in the benchmark experiment above we applied `MASS::lda()` and `rpart::rpart()` to the `sonar.task()`. We now perform a second experiment using a `randomForest::randomForest()` and quadratic discriminant analysis `MASS::qda()` and merge the results.

```
## First benchmark result
bmr
```

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

```
## Benchmark experiment for the additional learners
lrns2 = list(makeLearner("classif.randomForest"), makeLearner("classif.qda"))
bmr2 = benchmark(lrns2, sonar.task, rdesc, show.info = FALSE)
bmr2
```

```
##         task.id           learner.id mmce.test.mean
## 1 Sonar_example classif.randomForest      0.1714286
## 2 Sonar_example          classif.qda      0.4000000
```

```
## Merge the results
mergeBenchmarkResults(list(bmr, bmr2))
```

```
##         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      0.1714286
## 4 Sonar_example          classif.qda      0.4000000
```

Note that in the above examples in each case a resample description (`makeResampleDesc()`) was passed to the `benchmark()` function. For this reason `MASS::lda()` and `rpart::rpart()` were most likely evaluated on a different training/test set pair than `randomForest::randomForest()` and `MASS::qda()`.

Differing training/test set pairs across learners pose an additional source of variation in the results, which can make it harder to detect actual performance differences between learners. Therefore, if you suspect that you will have to extend your benchmark experiment by another `makeLearner()` later on it's probably easiest to work with `makeResampleInstance()`s from the start. These can be stored and used for any additional experiments.

Alternatively, if you used a resample description in the first benchmark experiment you could also extract the `makeResampleInstance()`s from the `BenchmarkResult()` `bmr` and pass these to all further `benchmark()` calls.

```
rin = getBMRPredictions(bmr)[[1]][[1]]$instance
rin
```

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

```
## Benchmark experiment for the additional random forest
bmr3 = benchmark(lrns2, sonar.task, rin, show.info = FALSE)
bmr3
```

```
##         task.id           learner.id mmce.test.mean
## 1 Sonar_example classif.randomForest      0.1571429
## 2 Sonar_example          classif.qda      0.4857143
```

```
## Merge the results
mergeBenchmarkResults(list(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      0.4857143
```

8.4 Benchmark analysis and visualization

`mlr` offers several ways to analyze the results of a benchmark experiment. This includes visualization, ranking of learning algorithms and hypothesis tests to assess performance differences between learners.

In order to demonstrate the functionality we conduct a slightly larger benchmark experiment with three learning algorithms that are applied to five classification tasks.

8.4.1 Example: Comparing lda, rpart and random Forest

We consider `MASS::lda()`, classification trees `rpart::rpart()`, and random forests `randomForest::randomForest()`. Since the default learner IDs are a little long, we choose shorter names in the **R** code below.

We use five classification tasks. Three are already provided by `mlr`, two more data sets are taken from package `mlbench::mlbench()` and converted to `Task()`s by function `convertMLBenchObjToTask()`.

For all tasks 10-fold cross-validation is chosen as resampling strategy. This is achieved by passing a single resample description (`makeResampleDesc()`) to `benchmark()`, which is then instantiated automatically once for each `Task()`. This way, the same instance is used for all learners applied to a single task.

It is also possible to choose a different resampling strategy for each `Task()` by passing a `base::list()` of the same length as the number of tasks that can contain both resample descriptions (`makeResampleDesc()`) and resample instances (`makeResampleInstance()`).

We use the mean misclassification error `mmce` (`vignette("measures")`) as primary performance measure, but also calculate the balanced error rate `ber` and the training time `timetrain`.

```
## Create a list of learners
lrns = list(
  makeLearner("classif.lda", id = "lda"),
  makeLearner("classif.rpart", id = "rpart"),
  makeLearner("classif.randomForest", id = "randomForest")
)

## Get additional Tasks from package mlbench
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)
rdesc = makeResampleDesc("CV", iters = 10)
meas = list(mmce, ber, timetrain)
bmr = benchmark(lrns, tasks, rdesc, meas, show.info = FALSE)
bmr
```

```
##                       task.id   learner.id mmce.test.mean ber.test.mean
## 1                iris_example          lda     0.02000000    0.02222222
## 2                iris_example        rpart     0.08000000    0.07555556
## 3                iris_example randomForest     0.05333333    0.05250000
## 4             mlbench.ringnorm          lda     0.35000000    0.34605671
## 5             mlbench.ringnorm        rpart     0.17333333    0.17313632
## 6             mlbench.ringnorm randomForest     0.05833333    0.05806121
## 7             mlbench.waveform          lda     0.19000000    0.18257244
## 8             mlbench.waveform        rpart     0.28833333    0.28765247
## 9             mlbench.waveform randomForest     0.16500000    0.16306057
## 10 PimaIndiansDiabetes_example          lda     0.22778537    0.27148893
## 11 PimaIndiansDiabetes_example        rpart     0.25133288    0.28967870
## 12 PimaIndiansDiabetes_example randomForest     0.23685919    0.27543146
## 13               Sonar_example          lda     0.24619048    0.23986694
## 14               Sonar_example        rpart     0.30785714    0.31153361
## 15               Sonar_example randomForest     0.17785714    0.17442696
##    timetrain.test.mean
## 1               0.0036
## 2               0.0054
## 3               0.0372
## 4               0.0075
## 5               0.0118
## 6               0.3292
## 7               0.0076
## 8               0.0122
```

```
## 9               0.3719
## 10              0.0039
## 11              0.0082
## 12              0.4017
## 13              0.0134
## 14              0.0140
## 15              0.2500
```

From the aggregated performance values we can see that for the iris- and PimaIndiansDiabetes-example linear discriminant analysis (`MASS::lda()`) performs well while for all other tasks the `randomForest::randomForest()` seems superior. Training takes longer for the `randomForest::randomForest()` than for the other learners.

In order to draw any conclusions from the average performances at least their variability has to be taken into account or, preferably, the distribution of performance values across resampling iterations.

The individual performances on the 10 folds for every task, learner, and measure are retrieved below.

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

```
##        task.id learner.id iter      mmce       ber timetrain
## 1 iris_example        lda    1 0.0000000 0.0000000     0.003
## 2 iris_example        lda    2 0.1333333 0.1666667     0.002
## 3 iris_example        lda    3 0.0000000 0.0000000     0.003
## 4 iris_example        lda    4 0.0000000 0.0000000     0.004
## 5 iris_example        lda    5 0.0000000 0.0000000     0.005
## 6 iris_example        lda    6 0.0000000 0.0000000     0.005
```

A closer look at the result reveals that the `randomForest::randomForest()` outperforms the classification tree (`rpart::rpart()`) in every instance, while linear discriminant analysis (`MASS::lda()`) performs better than `rpart::rpart()` most of the time. Additionally `MASS::lda()` sometimes even beats the `randomForest::randomForest()`. With increasing size of such `benchmark()` experiments, those tables become almost unreadable and hard to comprehend.

`mlr` features some plotting functions to visualize results of benchmark experiments that you might find useful. Moreover, `mlr` offers statistical hypothesis tests to assess performance differences between learners.

8.4.2 Integrated plots

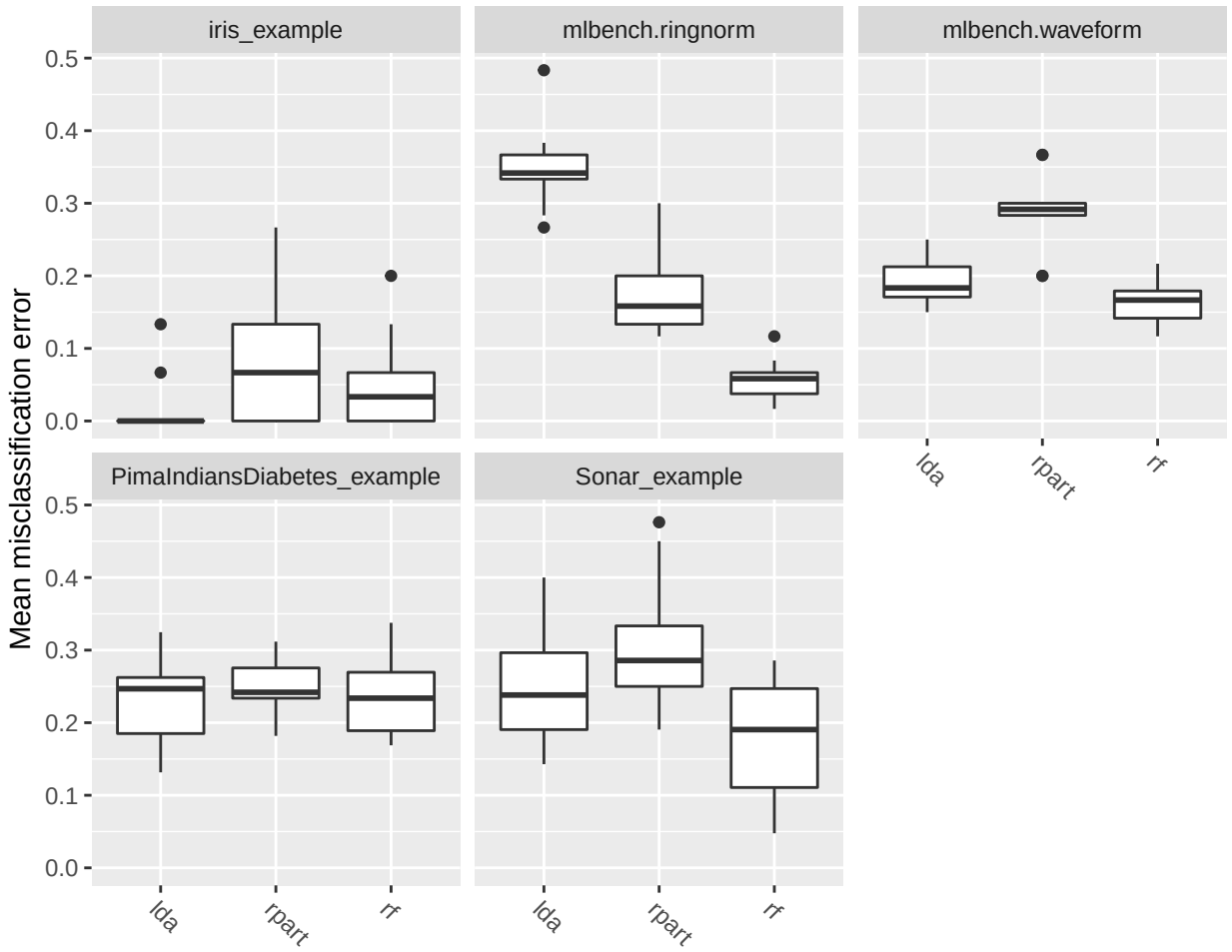
Plots are generated using `ggplot2::ggplot2()`. Further customization, such as renaming plot elements or changing colors, is easily possible.

8.4.2.1 Visualizing performances

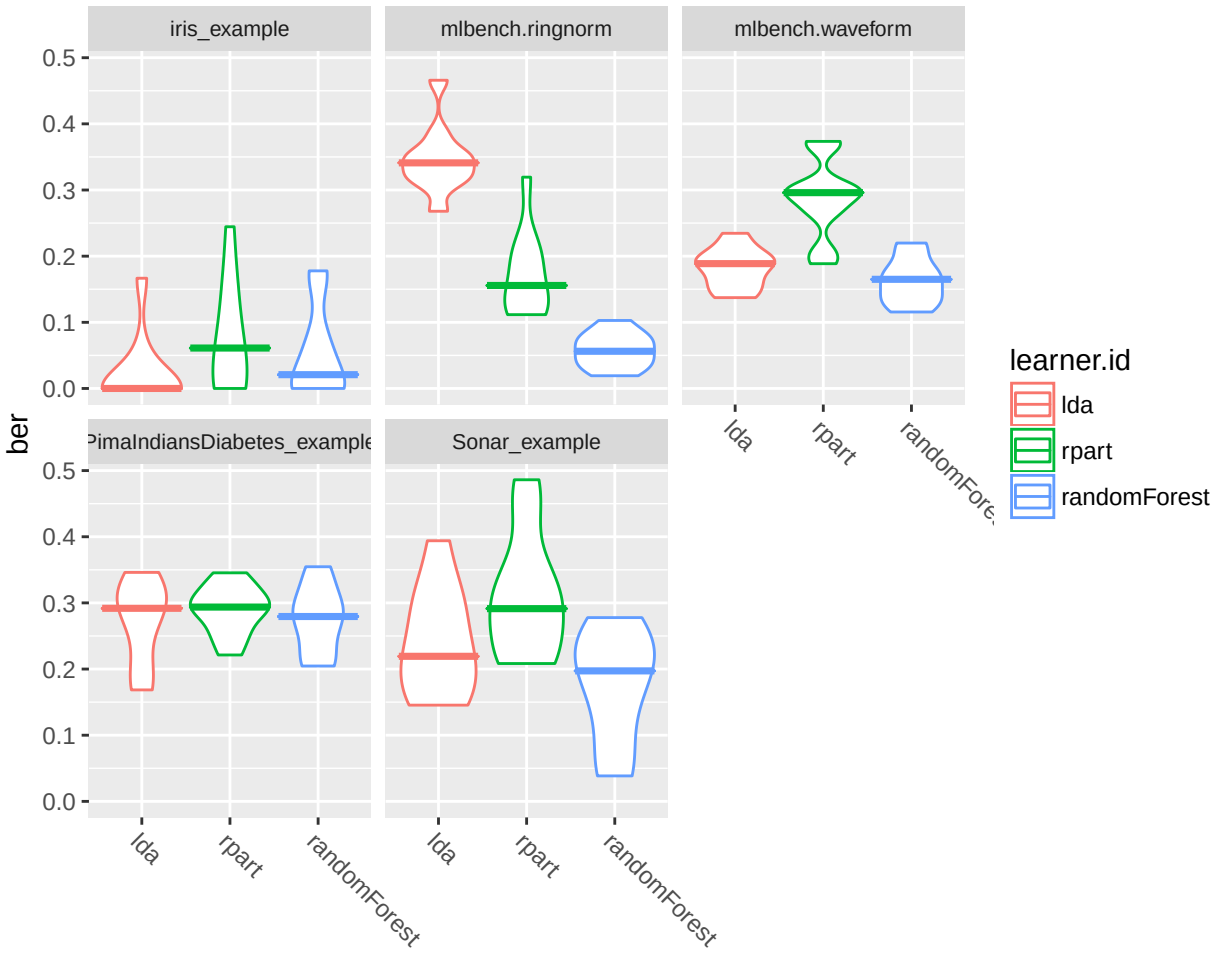
`plotBMRBoxplots()` creates box or violin plots which show the distribution of performance values across resampling iterations for one performance measure and for all learners and tasks (and thus visualize the output of `getBMRPerformances()`).

Below are both variants, box and violin plots. The first plot shows the `mmce` and the second plot the `ber` (`vignette("measures")`). Moreover, in the second plot we color the boxes according to the `learner.id`s.

```
plotBMRBoxplots(bmr, measure = mmce)
```




```
plotBMRBoxplots(bmr, measure = ber, style = "violin", pretty.names = FALSE) +
  aes(color = learner.id) +
  theme(strip.text.x = element_text(size = 8))
```



Note that by default the measure `name`s and the learner `short.name`s are used as axis labels.

```
mmce$name
```

```
## [1] "Mean misclassification error"
```

```
mmce$id
```

```
## [1] "mmce"
```

```
getBMRLearnerIds(bmr)
```

```
## [1] "lda"          "rpart"        "randomForest"
```

```
getBMRLearnerShortNames(bmr)
```

```
## [1] "lda"   "rpart" "rf"
```

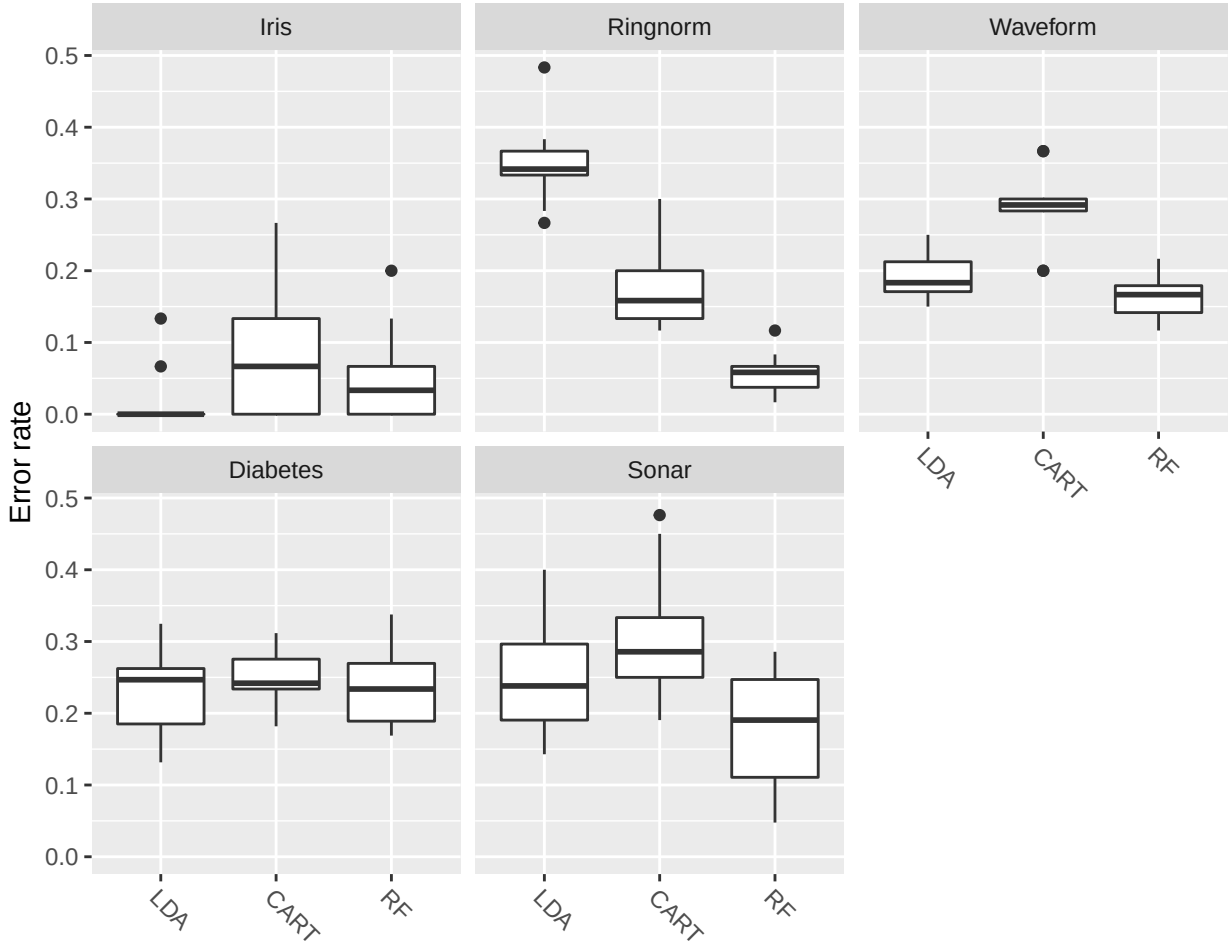
If you prefer the `id`s like, e.g., mmce and ber set `pretty.names = FALSE` (as done for the second plot). Of course you can also use the `ggplot2::ggplot2()` functionality like the `ggplot2::labs()` function to choose completely different labels.

One question which comes up quite often is how to change the panel headers (which default to the `Task()` IDs) and the learner names on the x-axis. For example looking at the above plots we would like to remove the "example" suffixes and the "mlbench" prefixes from the panel headers. Moreover, we want uppercase learner labels. Currently, the probably simplest solution is to change the factor levels of the plotted data as shown below.

```
plt = plotBMRBoxplots(bmr, measure = mmce)
head(plt$data)
```

```
##        task.id learner.id iter      mmce       ber timetrain
## 1 iris_example        lda    1 0.0000000 0.0000000     0.003
## 2 iris_example        lda    2 0.1333333 0.1666667     0.002
## 3 iris_example        lda    3 0.0000000 0.0000000     0.003
## 4 iris_example        lda    4 0.0000000 0.0000000     0.004
## 5 iris_example        lda    5 0.0000000 0.0000000     0.005
## 6 iris_example        lda    6 0.0000000 0.0000000     0.005
```

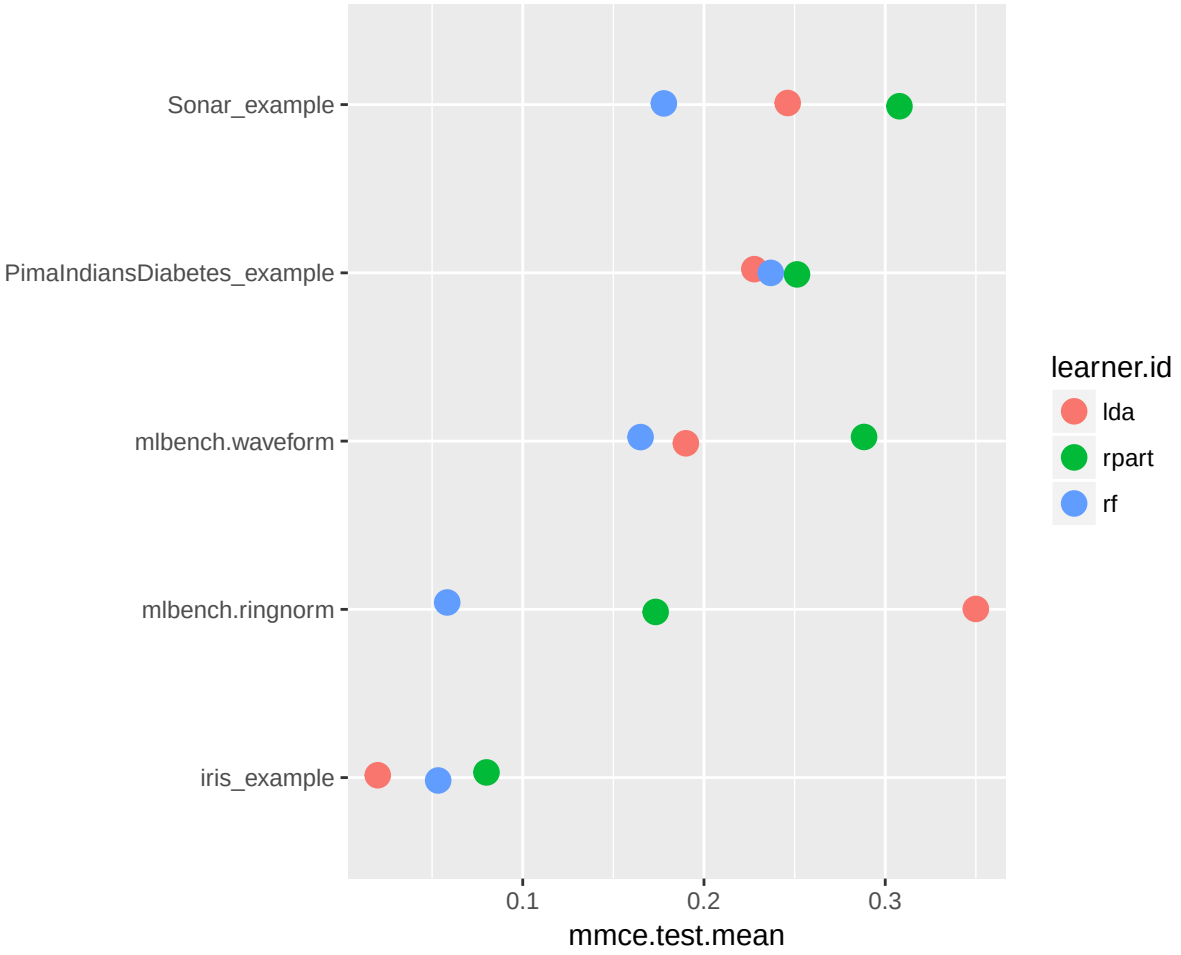
```
levels(plt$data$task.id) = c("Iris", "Ringnorm", "Waveform", "Diabetes", "Sonar")
levels(plt$data$learner.id) = c("LDA", "CART", "RF")

plt + ylab("Error rate")
```



8.4.2.2 Visualizing aggregated performances

The aggregated performance values (resulting from `getBMRAggrPerformances()`) can be visualized by function `plotBMRSummary()`. This plot draws one line for each task on which the aggregated values of one performance measure for all learners are displayed. By default, the first measure in the `base::list()` of Measure's (`makeMeasure()`) passed to `benchmark()` is used, in our example `vignette("measures")`. Moreover, a small vertical jitter is added to prevent overplotting.

```
plotBMRSummary(bmr)
```



8.4.2.3 Calculating and visualizing ranks

Additional to the absolute performance, relative performance, i.e., ranking the learners is usually of interest and might provide valuable additional insight.

Function `convertBMRToRankMatrix()` calculates ranks based on aggregated learner performances of one measure. We choose the mean misclassification error (`vignette("measures")`). The rank structure can be visualized by `plotBMRRanksAsBarChart()`.

```
m = convertBMRToRankMatrix(bmr, mmce)
m
```

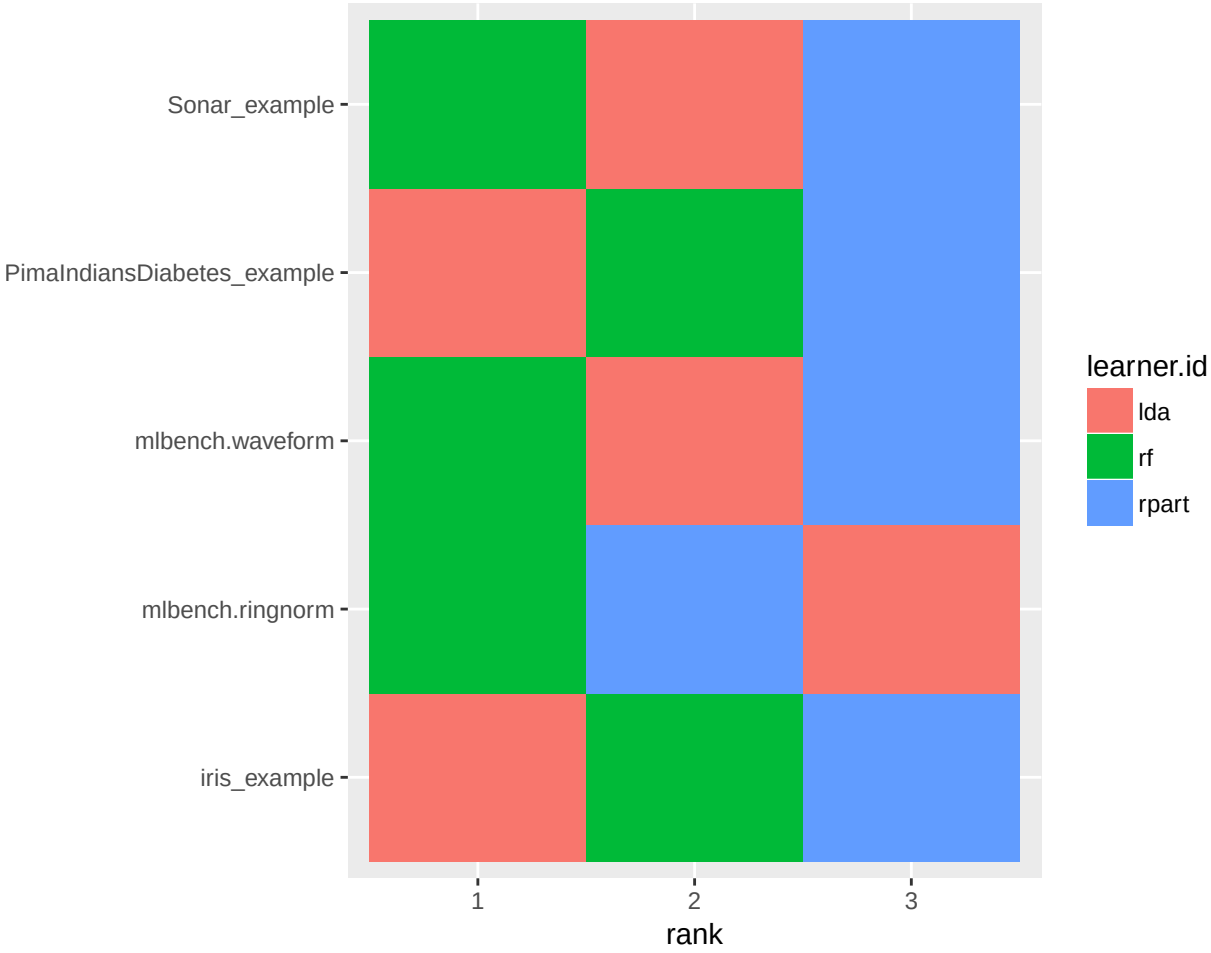
```
##              iris_example mlbench.ringnorm mlbench.waveform
## lda                     1                3                2
## rpart                   3                2                3
## randomForest            2                1                1
##              PimaIndiansDiabetes_example Sonar_example
## lda                                    1             2
## rpart                                  3             3
## randomForest                           2             1
```

Methods with best performance, i.e., with lowest `vignette("measures")`, are assigned the lowest rank. Linear discriminant analysis (`MASS::lda()`) is best for the iris and PimaIndiansDiabetes-examples while the

`randomForest::randomForest()` shows best results on the remaining tasks.

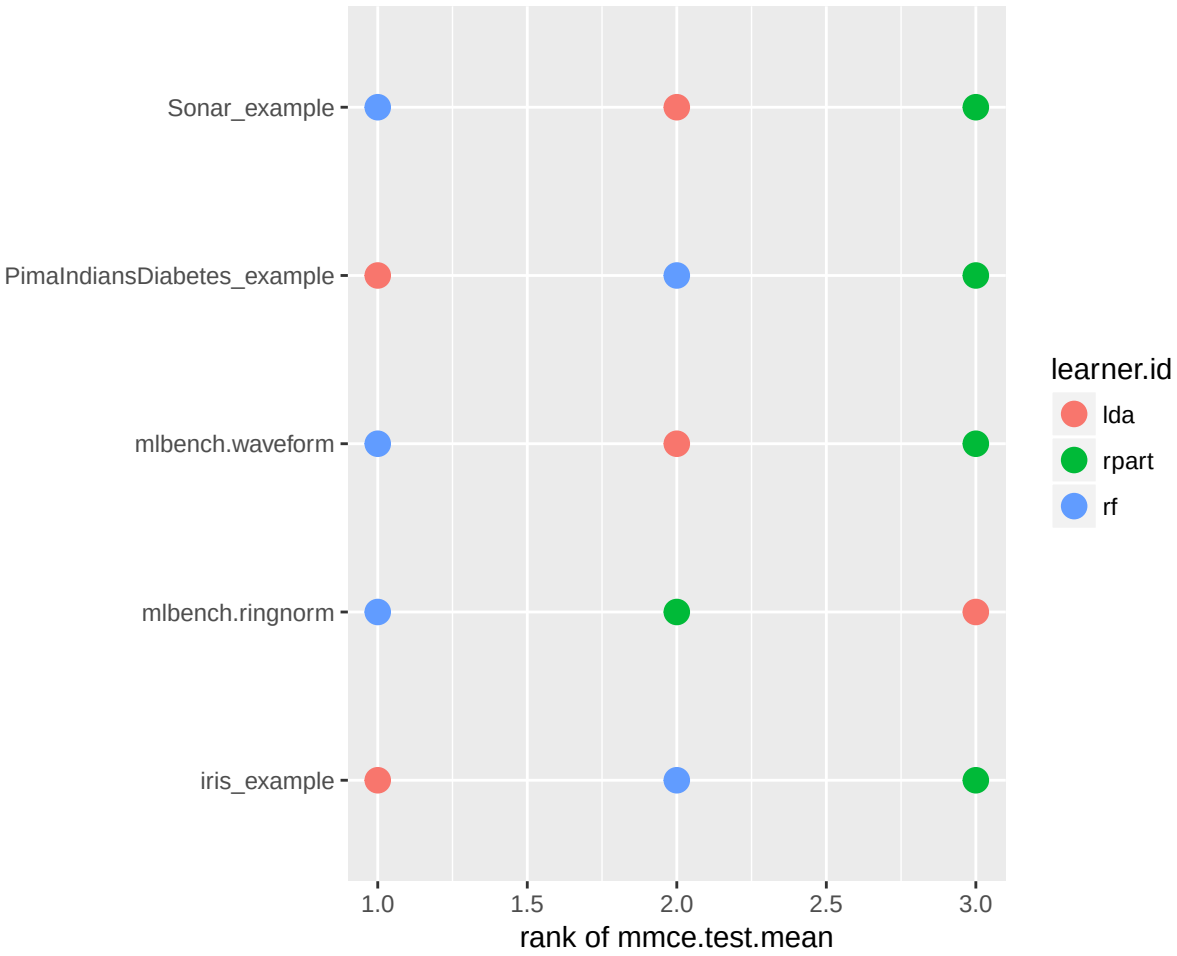
`plotBMRRanksAsBarChart()` with option `pos = "tile"` shows a corresponding heat map. The ranks are displayed on the x-axis and the learners are color-coded.

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



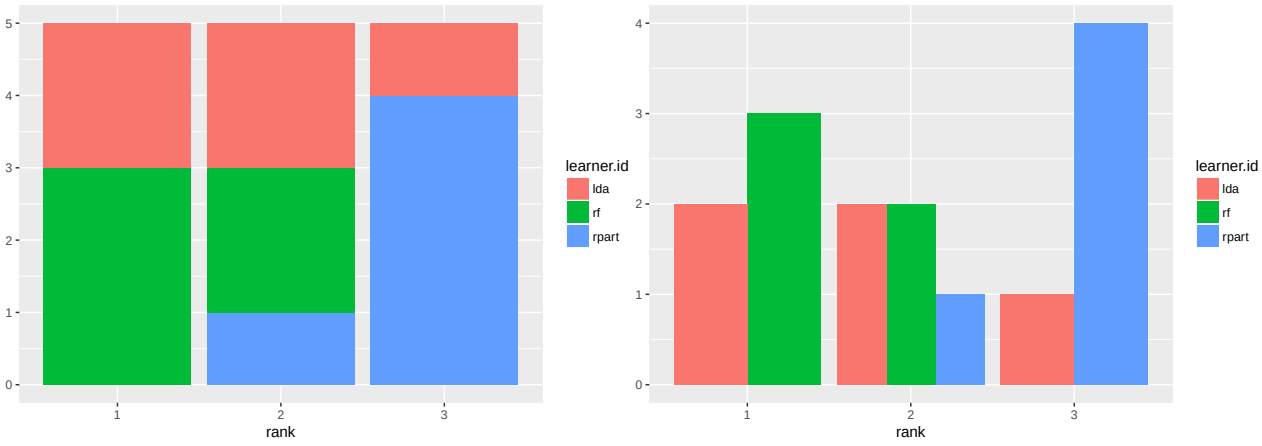
A similar plot can also be obtained via `plotBMRSummary()`. With option `trafo = "rank"` the ranks are displayed instead of the aggregated performances.

```
plotBMRSummary(bmr, trafo = "rank", jitter = 0)
```



Alternatively, you can draw stacked bar charts (the default) or bar charts with juxtaposed bars (`pos = "dodge"`) that are better suited to compare the frequencies of learners within and across ranks.

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



8.4.3 Comparing learners using hypothesis tests

Many researchers feel the need to display an algorithm's superiority by employing some sort of hypothesis testing. As non-parametric tests seem better suited for such benchmark results the tests provided in `mlr` are the **Overall Friedman test** and the **Friedman-Nemenyi post hoc test**.

While the ad hoc `friedmanTestBMR()` based on `stats::friedman.test()` is testing the hypothesis whether there is a significant difference between the employed learners, the post hoc `friedmanPostHocTestBMR()` tests for significant differences between all pairs of learners. *Non parametric* tests often do have less power then their *parametric* counterparts but less assumptions about underlying distributions have to be made. This often means many **data sets** are needed in order to be able to show significant differences at reasonable significance levels.

In our example, we want to compare the three learners on the selected data sets. First we might we want to test the hypothesis whether there is a difference between the learners.

```
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
```

In order to keep the computation time for this tutorial small, the `makeLearner()`s are only evaluated on five tasks. This also means that we operate on a relatively low significance level $\alpha = 0.1$. As we can reject the null hypothesis of the Friedman test at a reasonable significance level we might now want to test where these differences lie exactly.

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

```
##
##  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
```

At this level of significance, we can reject the null hypothesis that there exists no performance difference between the decision tree (`rpart::rpart()`) and the `randomForest::randomForest()`.

8.4.4 Critical differences diagram

In order to visualize differently performing learners, a critical differences diagramcan be plotted, using either the Nemenyi test (`test = "nemenyi"`) or the Bonferroni-Dunn test (`test = "bd"`).

The mean rank of learners is displayed on the x-axis.

- Choosing `test = "nemenyi"` compares all pairs of `makeLearner()`s to each other, thus the output are groups of not significantly different learners. The diagram connects all groups of learners where the mean ranks do not differ by more than the critical differences. Learners that are not connected by a bar are significantly different, and the learner(s) with the lower mean rank can be considered "better" at the chosen significance level.

- Choosing `test = "bd"` performs a *pairwise comparison with a baseline*. An interval which extends by the given *critical difference* in both directions is drawn around the `makeLearner()` chosen as baseline, though only comparisons with the baseline are possible. All learners within the interval are not significantly different, while the baseline can be considered better or worse than a given learner which is outside of the interval.

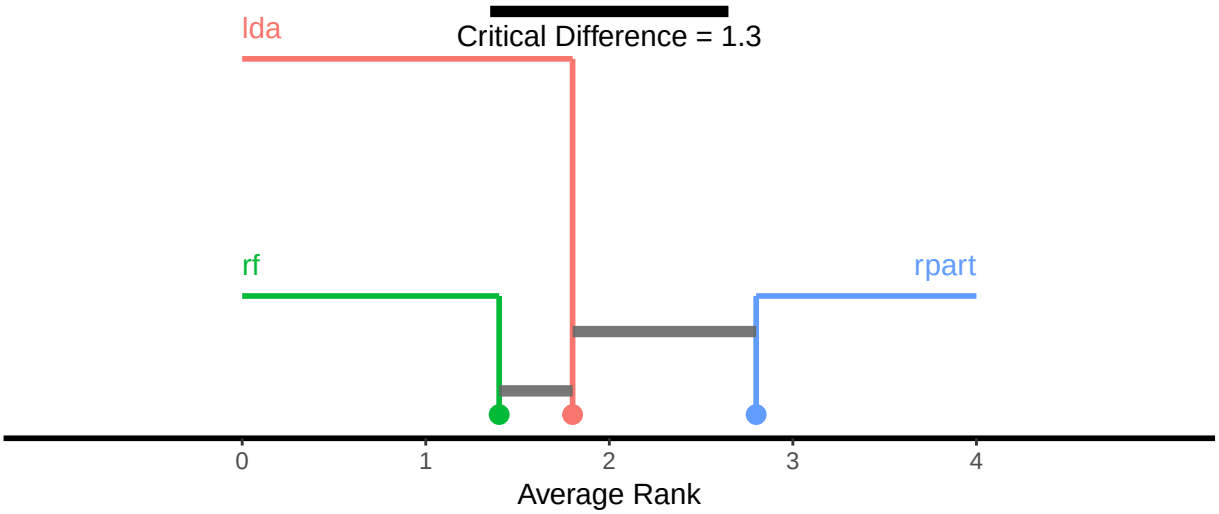
The critical difference CD is calculated by

$$CD = q_\alpha \cdot \sqrt{\frac{k(k+1)}{6N}},$$

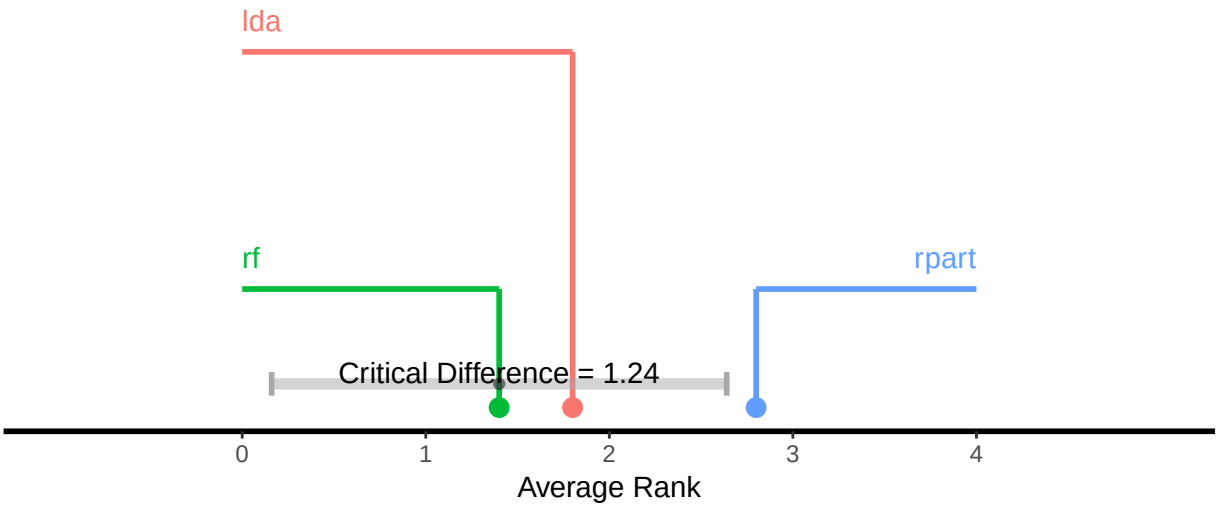
where N denotes the number of tasks, k is the number of learners, and q_α comes from the studentized range statistic divided by $\sqrt{2}$. For details see Demsar (2006).

Function `generateCritDifferencesData()` does all necessary calculations while function `plotCritDifferences()` draws the plot. See the tutorial page about `vignette("visualization")` for details on data generation and plotting functions.

```
## Nemenyi test
g = generateCritDifferencesData(bmr, p.value = 0.1, test = "nemenyi")
plotCritDifferences(g) + coord_cartesian(xlim = c(-1,5), ylim = c(0,2))
```




```
## Bonferroni-Dunn test
g = generateCritDifferencesData(bmr, p.value = 0.1, test = "bd", baseline = "randomForest")
plotCritDifferences(g) + coord_cartesian(xlim = c(-1,5), ylim = c(0,2))
```

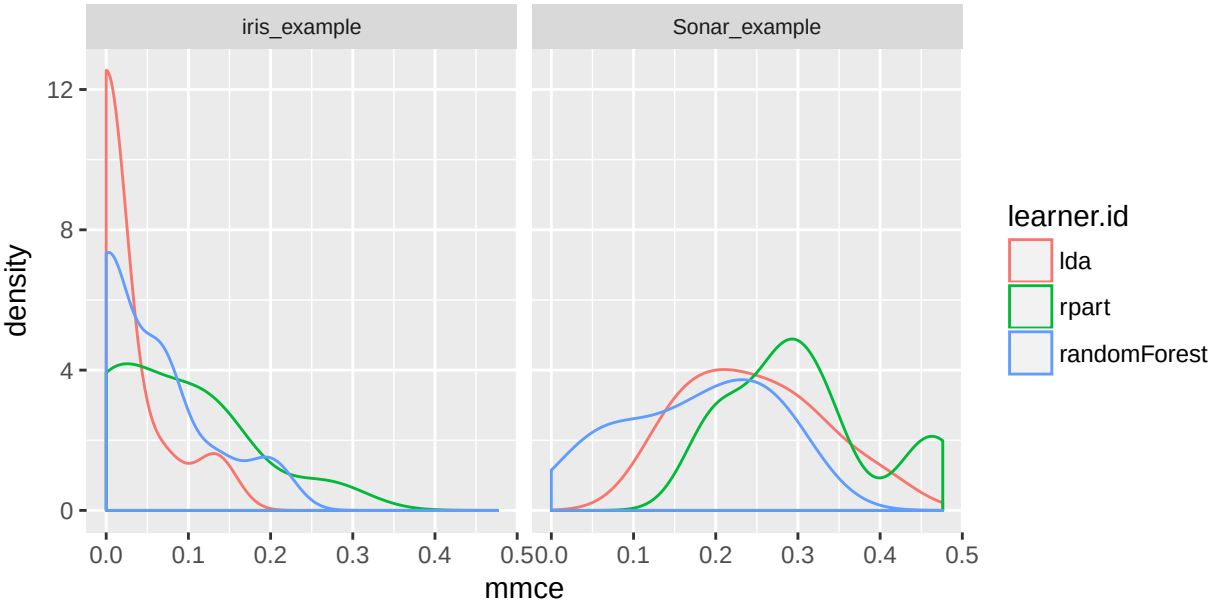


8.4.5 Custom plots

You can easily generate your own visualizations by customizing the `ggplot2::ggplot()` objects returned by the plots above, retrieve the data from the `ggplot2::ggplot()` objects and use them as basis for your own plots, or rely on the `base::data.frame()`s returned by `getBMRPerformances()` or `getBMRAggrPerformances()`. Here are some examples.

Instead of boxplots (as in `plotBMRBoxplots()`) we could create density plots to show the performance values resulting from individual resampling iterations.

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

## Density plots for two tasks
qplot(mmce, colour = learner.id, facets = . ~ task.id,
  data = perf[perf$task.id %in% c("iris_example", "Sonar_example"),], geom = "density") +
  theme(strip.text.x = element_text(size = 8))
```



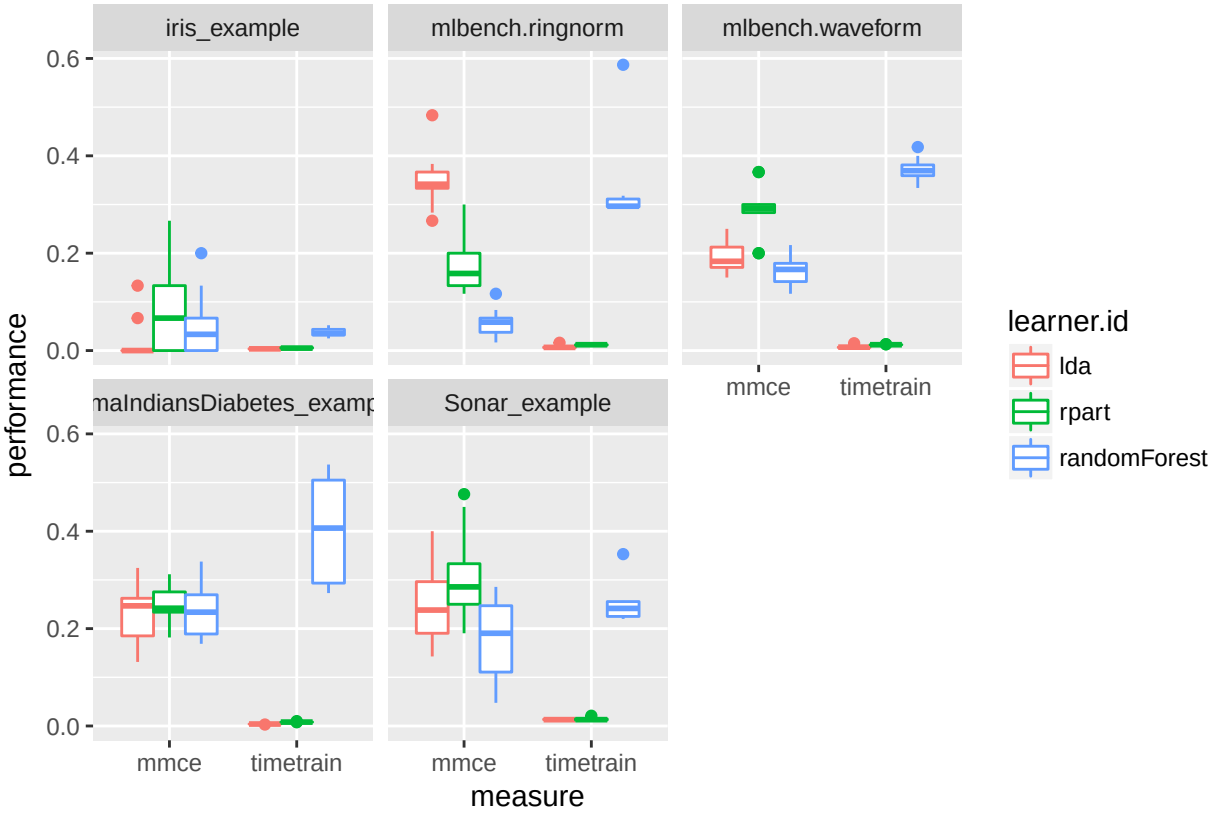
In order to plot multiple performance measures in parallel, `perf` is reshaped to long format. Below we

generate grouped boxplots showing the error rate and the training time (`vignette("measures")`).

```
## Compare mmce and timetrain
df = reshape2::melt(perf, id.vars = c("task.id", "learner.id", "iter"))
df = df[df$variable != "ber",]
head(df)
```

```
##        task.id learner.id iter variable     value
## 1 iris_example        lda    1     mmce 0.0000000
## 2 iris_example        lda    2     mmce 0.1333333
## 3 iris_example        lda    3     mmce 0.0000000
## 4 iris_example        lda    4     mmce 0.0000000
## 5 iris_example        lda    5     mmce 0.0000000
## 6 iris_example        lda    6     mmce 0.0000000
```

```
qplot(variable, value, data = df, colour = learner.id, geom = "boxplot",
  xlab = "measure", ylab = "performance") +
  facet_wrap(~ task.id, nrow = 2)
```



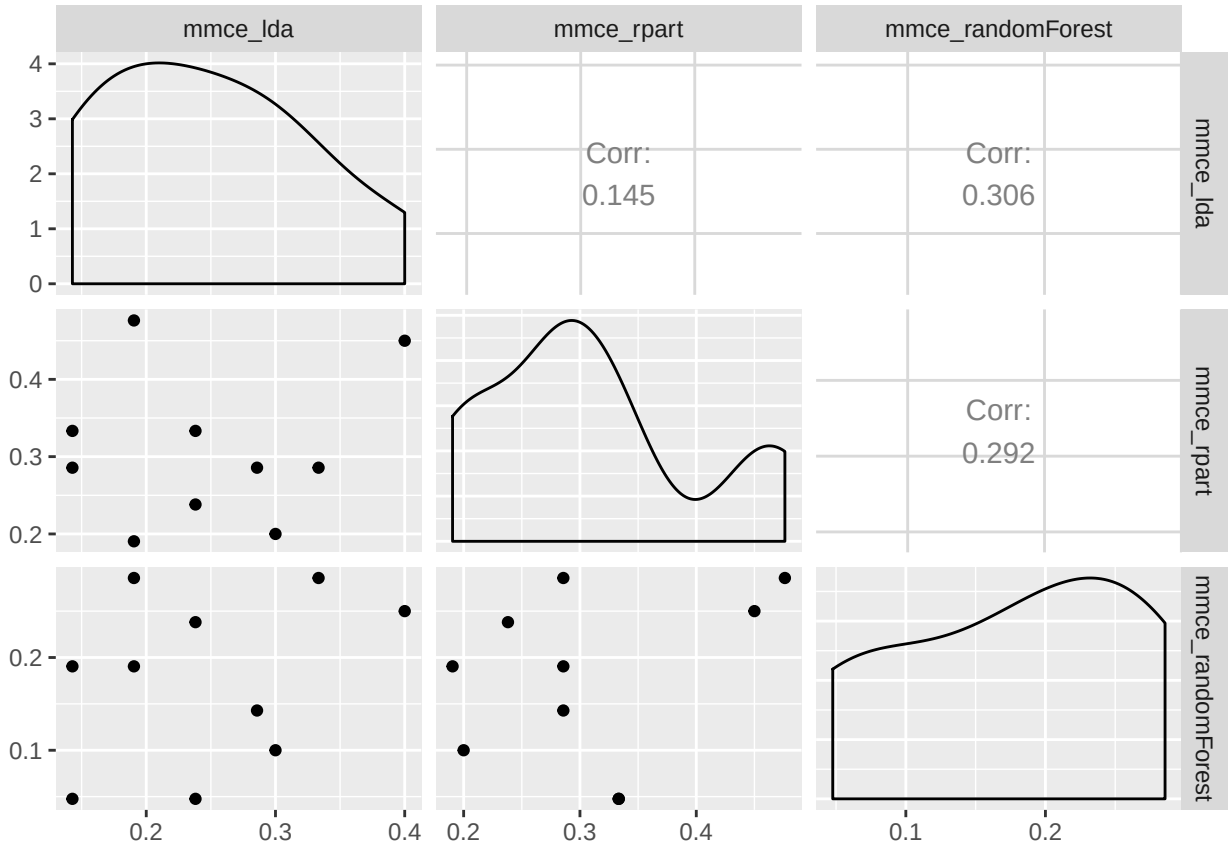
It might also be useful to assess if learner performances in single resampling iterations, i.e., in one fold, are related. This might help to gain further insight, for example by having a closer look at train and test sets from iterations where one learner performs exceptionally well while another one is fairly bad. Moreover, this might be useful for the construction of ensembles of learning algorithms. Below, function `GGally::ggpairs()` from package `GGally::GGally()` is used to generate a scatterplot matrix of mean misclassification errors (`vignette("measures")`) on the `mlbench::Sonar()` data set.

```
perf = getBMRPerformances(bmr, task.id = "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 0.2857143  0.2857143        0.14285714
## 2 Sonar_example    2 0.2380952  0.2380952        0.23809524
## 3 Sonar_example    3 0.3333333  0.2857143        0.28571429
## 4 Sonar_example    4 0.2380952  0.3333333        0.04761905
## 5 Sonar_example    5 0.1428571  0.2857143        0.19047619
## 6 Sonar_example    6 0.4000000  0.4500000        0.25000000
```

```
GGally::ggpairs(df, 3:5)
```



8.5 Further comments

- Note that for supervised classification `mlr` offers some more plots that operate on `BenchmarkResult()` objects and allow you to compare the performance of learning algorithms. See for example the tutorial page on `vignette("roc_analysis")` and functions `generateThreshVsPerfData()`, `plotROCCurves()`, and `plotViperCharts()` as well as the page about `vignette("classifier_calibration")` and function `generateCalibrationData()`.
- In the examples shown in this section we applied "raw" learning algorithms, but often things are more complicated. At the very least, many learners have hyperparameters that need to be tuned to get sensible results. Reliable performance estimates can be obtained by `vignette("nested_resampling")`, i.e., by doing the tuning in an inner resampling loop while estimating the performance in an outer loop. Moreover, you might want to combine learners with pre-processing steps like imputation, scaling, outlier removal, dimensionality reduction or feature selection and so on. All this can be easily done using `mlr`'s wrapper functionality. The general principle is explained in the section about `vignette("wrapper")`

in the Advanced part of this tutorial. There are also several sections devoted to common pre-processing steps.
- Benchmark experiments can very quickly become computationally demanding. `mlr` offers some possibilities for `vignette("parallelization")`.

9 Parallelization

R by default does not make use of parallelization. With the integration of `parallelMap()` into `mlr`, it becomes easy to activate the parallel computing capabilities already supported by `mlr`. `parallelMap()` works with all major parallelization backends: local multicore execution using `parallel()`, socket and MPI clusters using `snow()`, makeshift SSH-clusters using `BatchJobs()` and high performance computing clusters (managed by a scheduler like SLURM, Torque/PBS, SGE or LSF) also using `BatchJobs()`.

All you have to do is select a backend by calling one of the `parallelStart*` (`parallelMap::parallelStart()`) functions. The first loop `mlr` encounters which is marked as parallel executable will be automatically parallelized. It is good practice to call `parallelStop` (`parallelMap::parallelStop()`) at the end of your script.

```
library("parallelMap")
parallelStartSocket(2)
```

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

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

```
## Exporting objects to slaves for mode socket: .mlr.slave.options

## Resampling: cross-validation

## Measures:             mmce

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

##

## Aggregated Result: mmce.test.mean=0.0200000

##
```

```
parallelStop()
```

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

On Linux or Mac OS X, you may want to use `parallelStartMulticore` (`parallelMap::parallelStart()`) instead.

9.1 Parallelization levels

We offer different parallelization levels for fine grained control over the parallelization. E.g., if you do not want to parallelize the `benchmark()` function because it has only very few iterations but want to parallelize the resampling (`resample()`) of each learner instead, you can specifically pass the `level "mlr.resample"` to the `parallelStart*` (`parallelMap::parallelStart()`) function. Currently the following levels are supported:

```
parallelGetRegisteredLevels()
```

```
## mlr: mlr.benchmark, mlr.resample, mlr.selectFeatures, mlr.tuneParams, mlr.ensemble
```

For further details please see the `parallelization()` documentation page.

9.2 Custom learners and parallelization

If you have implemented a custom learner yourself (`vignette("create_learner")`), locally, you currently need to export this to the slave. So if you see an error after calling, e.g., a parallelized version of `resample()` like this:

```
no applicable method for 'trainLearner' applied to an object of class <my_new_learner>
```

simply add the following line somewhere after calling `parallelMap::parallelStart()`.

```
parallelExport("trainLearner.<my_new_learner>", "predictLearner.<my_new_learner>")
```

9.3 The end

For further details, consult the parallelMap tutorial and help (`?parallelMap()`).

10 Visualization

10.1 Generation and plotting functions

`mlr`'s visualization capabilities rely on *generation functions* which generate data for plots, and *plotting functions* which plot this output using either `ggplot2::ggplot2()` or `ggvis::ggvis()` (the latter being currently experimental).

This separation allows users to easily make custom visualizations by taking advantage of the generation functions. The only data transformation that is handled inside plotting functions is reshaping. The reshaped data is also accessible by calling the plotting functions and then extracting the data from the `ggplot2::ggplot()` object.

The functions are named accordingly.

- Names of generation functions start with `generate` and are followed by a title-case description of their `FunctionPurpose`, followed by `Data`, i.e., `generateFunctionPurposeData`. These functions output objects of class `FunctionPurposeData`.
- Plotting functions are prefixed by `plot` followed by their purpose, i.e., `plotFunctionPurpose`.
- `ggvis::ggvis()` plotting functions have an additional suffix `GGVIS`, i.e., `plotFunctionPurposeGGVIS`.

10.1.1 Some examples

In the example below we create a plot of classifier performance as function of the decision threshold for the binary classification problem `sonar.task`. The generation function `generateThreshVsPerfData()` creates an object of class `ThreshVsPerfData` which contains the data for the plot in slot `$data`.

```
lrn = makeLearner("classif.lda", predict.type = "prob")
n = getTaskSize(sonar.task)
mod = train(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))

class(d)
```

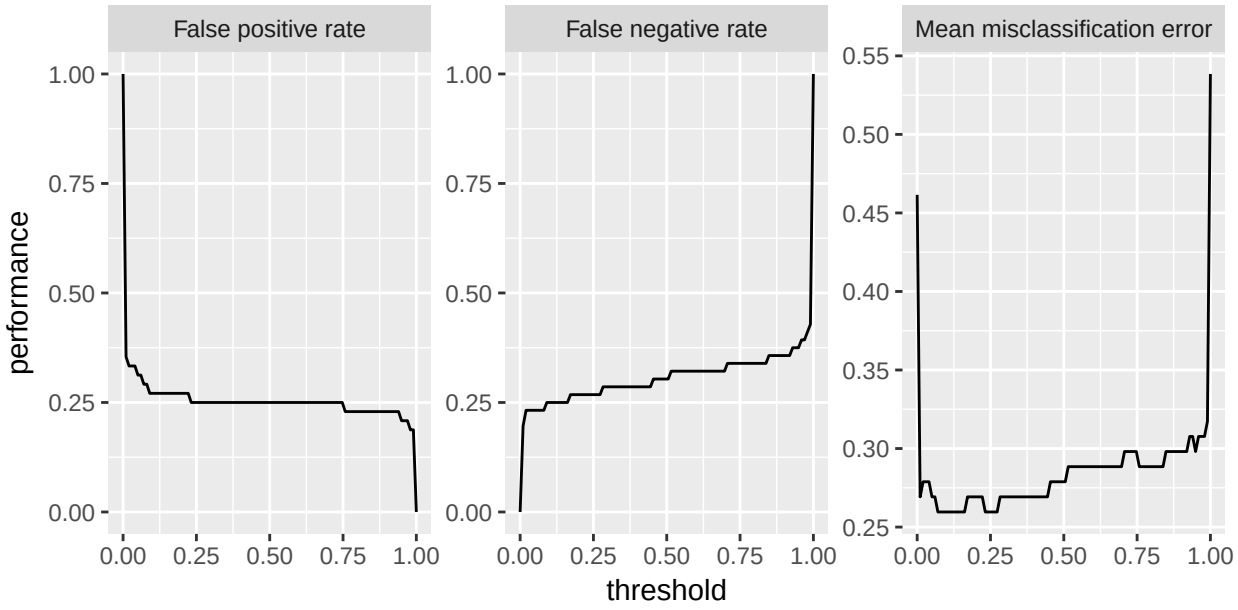
```
## [1] "ThreshVsPerfData"
```

```
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
```

For plotting we can use the built-in `mlr` function `plotThreshVsPerf()`.

```
plotThreshVsPerf(d)
```



Note that by default the `Measure` `name`s are used to annotate the panels.

```
fpr$name
```

```
## [1] "False positive rate"
```

```
fpr$id
```

```
## [1] "fpr"
```

This does not only apply to `plotThreshVsPerf()`, but to other plot functions that show performance measures as well, for example `plotLearningCurve()`. You can use the `id`s instead of the names by setting `pretty.names = FALSE`.

10.1.2 Customizing plots

As mentioned above it is easily possible to customize the built-in plots or making your own visualizations from scratch based on the generated data.

What will probably come up most often is changing labels and annotations. Generally, this can be done by manipulating the `ggplot2::ggplot()` object, in this example the object returned by `plotThreshVsPerf()`, using the usual `ggplot2::ggplot2()` functions like `ggplot2::labs()` or `ggplot2::labeller()`. Moreover, you can change the underlying data, either `d$data` (resulting from `generateThreshVsPerfData()` or the possibly reshaped data contained in the `ggplot2::ggplot()` object (resulting from `plotThreshVsPerf()`, most often by renaming columns or factor levels.

Below are two examples of how to alter the axis and panel labels of the above plot.

Imagine you want to change the order of the panels and also are not satisfied with the panel names, for example you find that "Mean misclassification error" is too long and you prefer "Error rate" instead. Moreover, you want the error rate to be displayed first.

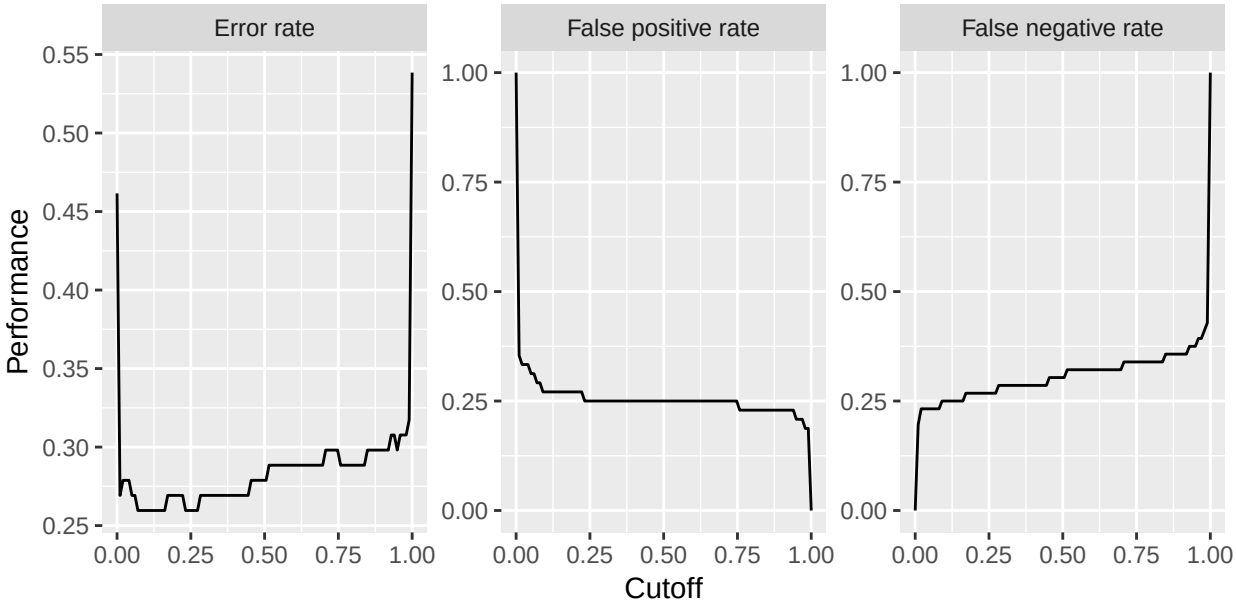
```
plt = plotThreshVsPerf(d, pretty.names = FALSE)

## Reshaped version of the underlying data d
head(plt$data)
```

```
##    threshold measure performance
## 1 0.00000000     fpr   1.0000000
## 2 0.01010101     fpr   0.3541667
## 3 0.02020202     fpr   0.3333333
## 4 0.03030303     fpr   0.3333333
## 5 0.04040404     fpr   0.3333333
## 6 0.05050505     fpr   0.3125000
```

```
levels(plt$data$measure)
```

```
## [1] "fpr"  "fnr"  "mmce"
```

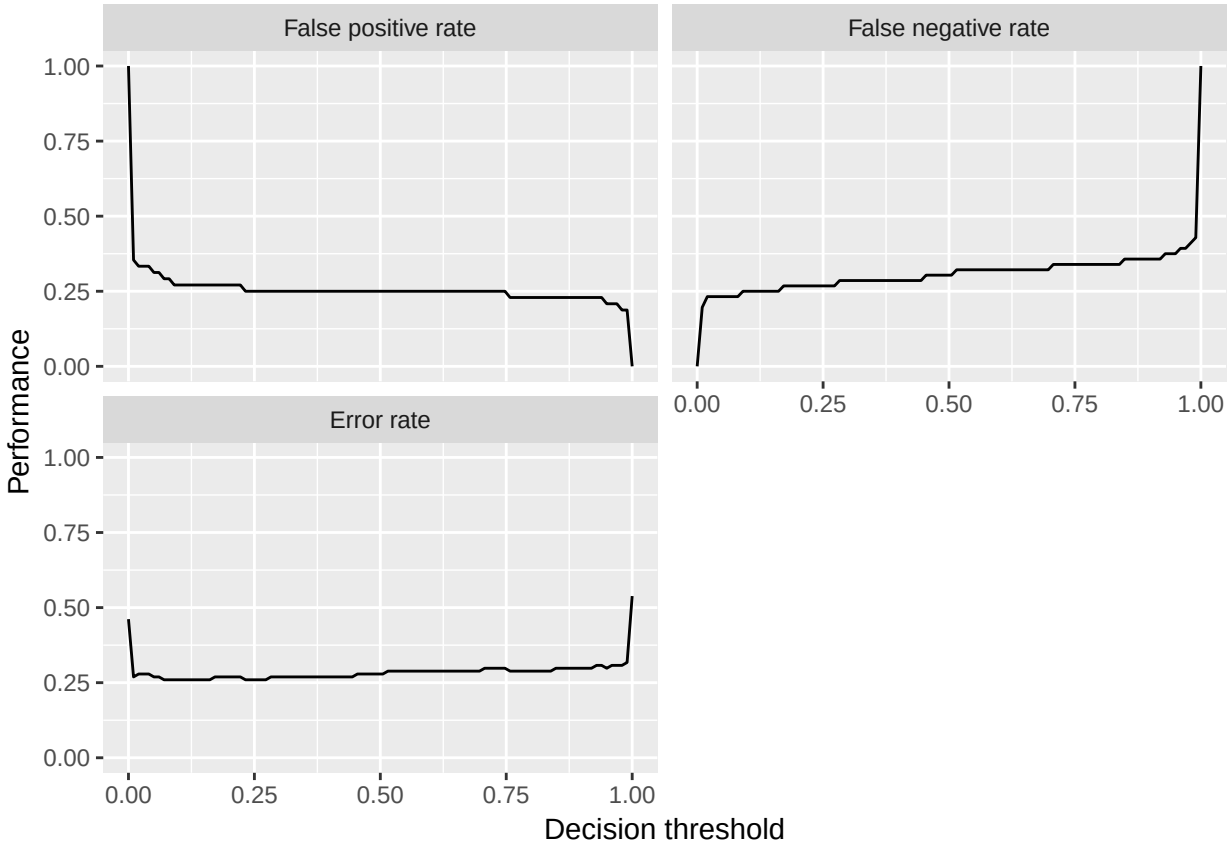
```
## Rename and reorder factor levels
plt$data$measure = factor(plt$data$measure, levels = c("mmce", "fpr", "fnr"),
  labels = c("Error rate", "False positive rate", "False negative rate"))
plt = plt + xlab("Cutoff") + ylab("Performance")
plt
```



Using the `ggplot2::labeller()` function requires calling `ggplot2::facet_wrap()` (or `ggplot2::facet_grid()`), which can be useful if you want to change how the panels are positioned (number of rows and columns) or influence the axis limits.

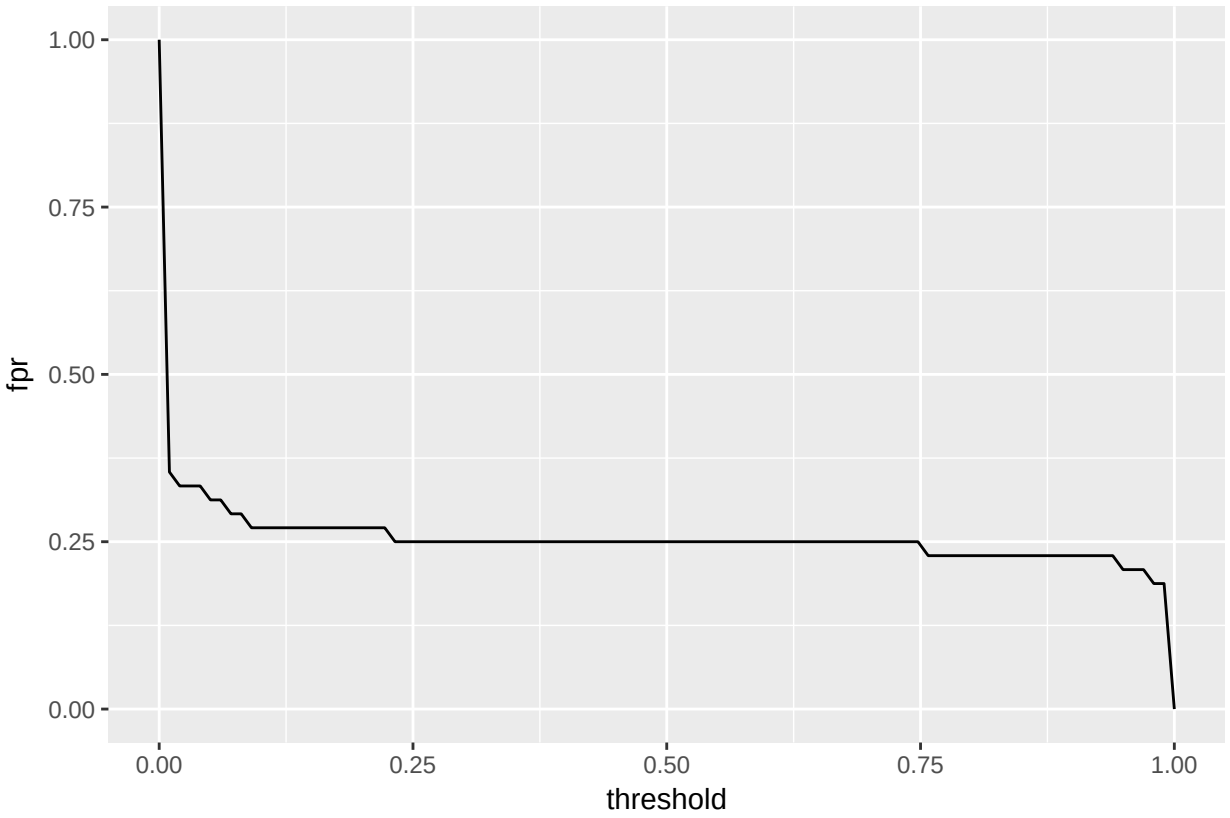
```
plt = plotThreshVsPerf(d, pretty.names = FALSE)

measure_names = c(
  fpr = "False positive rate",
  fnr = "False negative rate",
```

```
  mmce = "Error rate"
)
## Manipulate the measure names via the labeller function and
## arrange the panels in two columns and choose common axis limits for all panels
plt = plt + facet_wrap( ~ measure, labeller = labeller(measure = measure_names), ncol = 2)
plt = plt + xlab("Decision threshold") + ylab("Performance")
plt
```



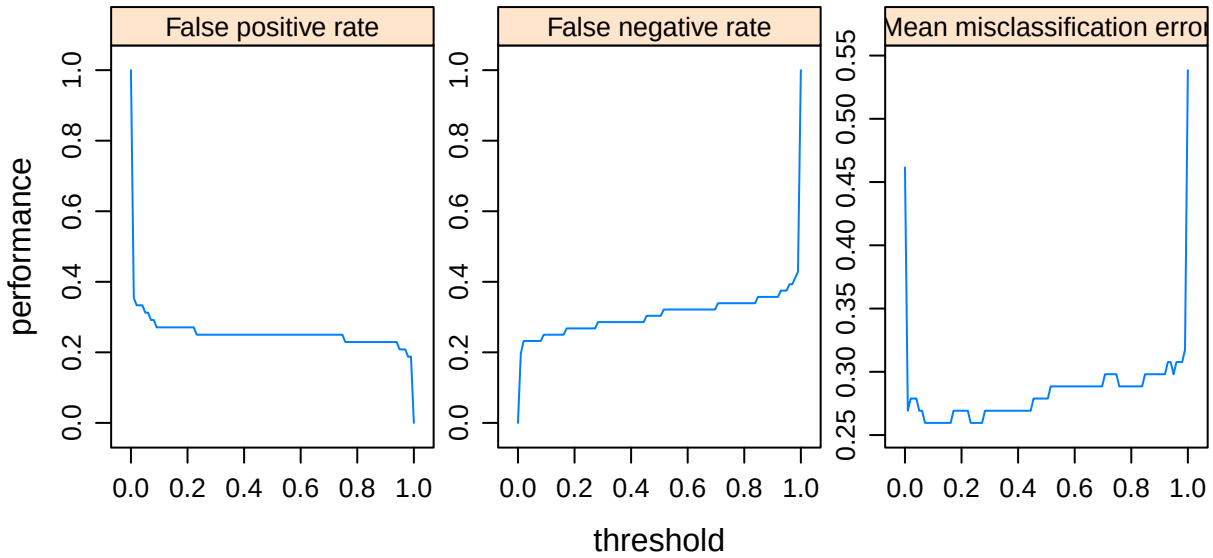
Instead of using the built-in function `plotThreshVsPerf()` we could also manually create the plot based on the output of `generateThreshVsPerfData()`: In this case to plot only one measure.

```
ggplot(d$data, aes(threshold, fpr)) + geom_line()
```



The decoupling of generation and plotting functions is especially practical if you prefer traditional `graphics::graphics()` or `lattice::lattice()`. Here is a `lattice::lattice()` plot which gives a result similar to that of `plotThreshVsPerf()`.

```
lattice::xyplot(fpr + fnr + mmce ~ threshold, data = d$data, type = "l", ylab = "performance",
  outer = TRUE, scales = list(relation = "free"),
  strip = strip.custom(factor.levels = sapply(d$measures, function(x) x$name),
    par.strip.text = list(cex = 0.8)))
```



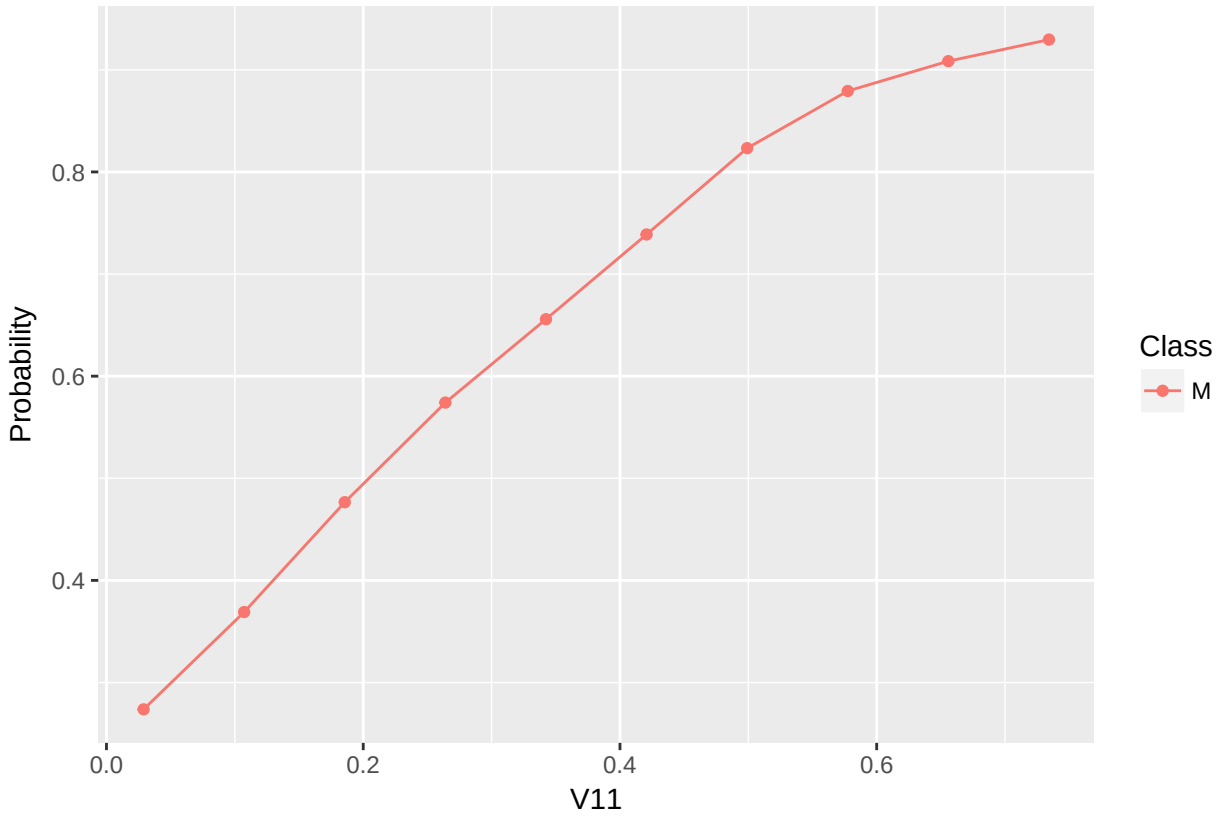
Let's conclude with a brief look on a second example. Here we use `plotPartialDependence()` but extract the data from the `ggplot2::ggplot()` object `plt` and use it to create a traditional `graphics::plot()`,

additional to the ggplot2::ggplot() plot.

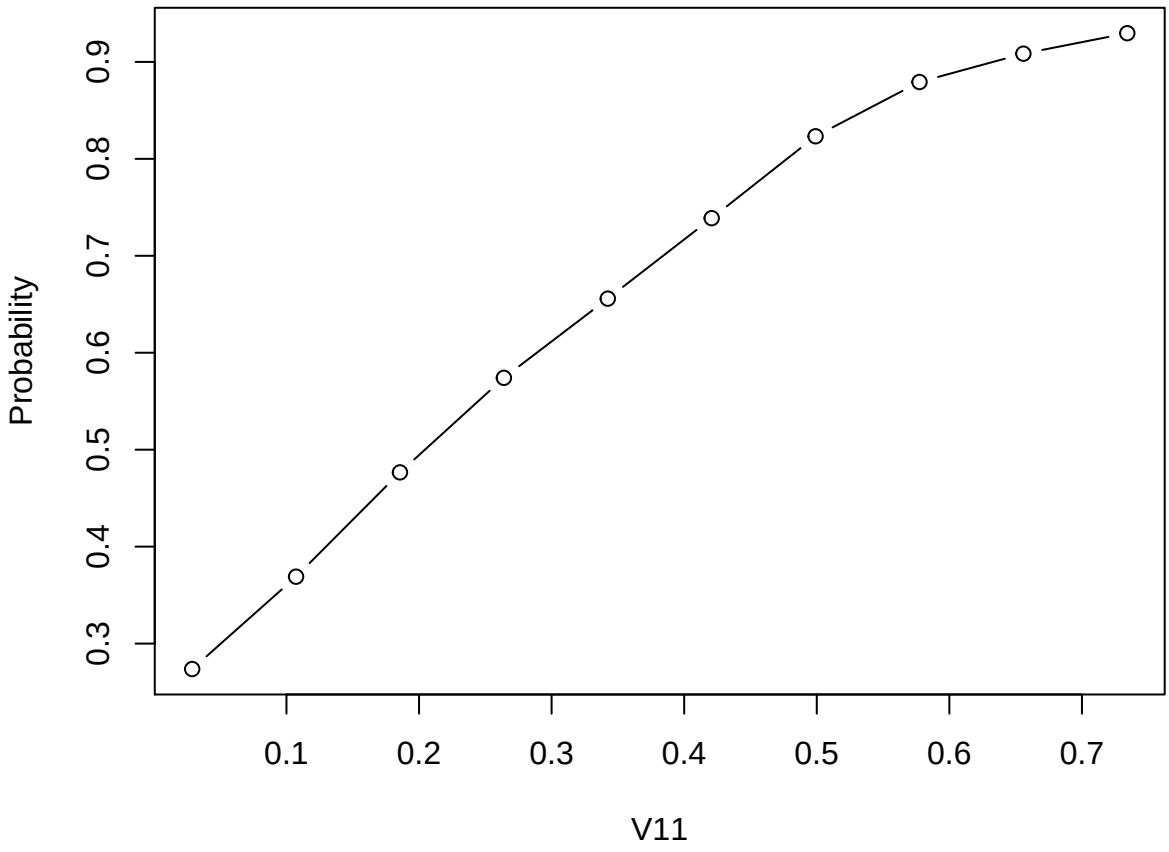
```
sonar = getTaskData(sonar.task)
pd = generatePartialDependenceData(mod, sonar, "V11")
plt = plotPartialDependence(pd)
head(plt$data)
```

```
##   Class Probability Feature     Value
## 1     M   0.2737158     V11 0.0289000
## 2     M   0.3689970     V11 0.1072667
## 3     M   0.4765742     V11 0.1856333
## 4     M   0.5741233     V11 0.2640000
## 5     M   0.6557857     V11 0.3423667
## 6     M   0.7387962     V11 0.4207333
```

```
plt
```




```
plot(Probability ~ Value, data = plt$data, type = "b", xlab = plt$data$Feature[1])
```



10.2 Available generation and plotting functions

Below the currently available generation and plotting functions are listed and tutorial pages that provide in depth descriptions of the listed functions are referenced.

Note that some plots, e.g., `plotTuneMultiCritResult()` are not mentioned here since they lack a generation function. Both `plotThreshVsPerf()` and `plotROCCurves()` operate on the result of `generateThreshVsPerfData()`. Functions `plotPartialDependence()` and `plotPartialDependenceGGVIS()` can be applied to the results of both `generatePartialDependenceData()` and `generateFunctionalANOVAData()`.

The `ggvis::ggvis()` functions are experimental and are subject to change, though they should work. Most generate interactive `shiny::shiny()` applications, that automatically start and run locally.

generation function	ggplot2 plotting function	ggvis plotting function	tutorial pages
`generateThreshVsPerfData()`	`plotThreshVsPerf()`	`plotThreshVsPerfGGVIS()`	`vignette("performance")`
	`plotROCCurves()`	–	`vignette("roc_analysis")`
`generateCritDifferencesData()`	`plotCritDifferences()`	–	`vignette("benchmark_experiments")`
`generateHyperParsEffectData()`	`plotHyperParsEffect()`	–	`vignette("tune")`, `vignette("hyperparameter_tuning_effects")`
`generateFilterValuesData()`	`plotFilterValues()`	`plotFilterValuesGGVIS()`	`vignette("feature_selection")`
`generateLearningCurveData()`	`plotLearningCurve()`	`plotLearningCurveGGVIS()`	`vignette("learning_curve_analysis")`
`generatePartialDependenceData()`	`plotPartialDependence()`	`plotPartialDependenceGGVIS()`	`vignette("partial_dependence")`
`generateFunctionalANOVAData()`			
`generateCalibrationData()`	`plotCalibration()`	–	`vignette("classifier_calibration")`