# 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 Machine Learning in R: mlr Tutorial

This document provides an in-depth introduction to Machine Learning in R: mlr, a framework for machine learning experiments in **R**.

In this tutorial, we focus on basic functions and applications. More detailed technical information can be found in the manual pages which are regularly updated and reflect the documentation of the current package version on CRAN.

Offline versions of this tutorial are also available for download:

- current mlr release on CRAN
- the mlr devel version on GitHub

The tutorial aims to walkthrough basic data analysis tasks step by step. We will use simple examples from classification, regression, cluster and survival analysis to illustrate the main features of the package.

Enjoy reading!

## 1.1 Quick start

Here we show the mlr workflow to train, make predictions, and evaluate a learner on a classification problem. We walk through 5 basic steps that work on any learning problem or method supported by mlr.

```
library(mlr)
data(iris)

## 1) Define the task
## Specify the type of analysis (e.g. classification) and provide
    data and response variable
```

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

## 2) Define the learner
## Choose a specific algorithm (e.g. linear discriminant analysis)
lrn = makeLearner("classif.lda")

n = nrow(iris)
train.set = sample(n, size = 2/3*n)
test.set = setdiff(1:n, train.set)

## 3) Fit the model
## Train the learner on the task using a random subset of the data
    as training set
model = train(lrn, task, subset = train.set)

## 4) Make predictions
## Predict values of the response variable for new observations by
    the trained model
## using the other part of the data as test set
pred = predict(model, task = task, subset = test.set)

## 5) Evaluate the learner
## Calculate the mean misclassification error and accuracy
performance(pred, measures = list(mmce, acc))
#> mmce  acc
#> 0.04 0.96
```

# 2 Basics

## 2.1 Learning Tasks

Learning tasks allow us to encapsulate the data set and specify information about a machine learning task.

### 2.1.1 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 inherit from the virtual superclass Task:

- RegrTask for regression problems
- ClassifTask for binary and multi-class classification problems
- SurvTask for survival analysis
- ClusterTask for cluster analysis

- MultilabelTask for multilabel classification problems
- CostSensTask for general cost-sensitive classification (with example-specific costs)

To create a task, just call `make<TaskType>`, e.g., makeClassifTask. All tasks have an identifier (argument `id`) and a 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 label results (for example 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.

#### 2.1.1.1 Regression

For supervised learning like regression (as well as classification and survival analysis), `data` and `target` must be specified.

```
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 (numeric vectors, 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.

#### 2.1.1.2 Classification

For classification, the target column must be a factor.

In the following example we define a classification task for the BreastCancer data set and exclude the variable `Id` from all further model 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* and *negative* class with the positive class being the category of greater interest. This is relevant for many performance measures like the *true positive rate* or ROC curves. Moreover, mlr allows the user to set options like the decision threshold or class weights 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, `benign` in the example above. Class `malignant` can be manually selected as follows:

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

#### 2.1.1.3 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 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.

#### 2.1.1.4 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.

In the following example, we extract the label names and pass them to the `target` argument in makeMultilabelTask.

```
yeast = getTaskData(yeast.task)

labels = colnames(yeast)[1:14]
yeast.task = makeMultilabelTask(id = "multi", data = yeast, target =
    labels)
yeast.task
#> Supervised task: multi
#> Type: multilabel
#> Target:
    label1,label2,label3,label4,label5,label6,label7,label8,label9,label10,label11,label12,label
#> Observations: 2417
#> Features:
#>    numerics     factors     ordered 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 on multilabel classification.

#### 2.1.1.5 Cluster analysis

As cluster analysis is unsupervised, the only mandatory argument to construct a cluster analysis task is the `data`. We create a learning task from mtcars below:

```
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
```

#### 2.1.1.6 Cost-sensitive classification

The standard objective in classification is to obtain a high prediction accuracy, i.e., to minimize the number of errors our classifier makes. This implies that all types of errors are treated equally. In many applications, such as healthcare or finance, different kinds of errors incur asymmetric costs.

For *class-dependent costs* that solely depend on the actual and predicted class labels, use ClassifTask.

For *example-specific costs*, use 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 iris data and an artificial cost matrix (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 about cost-sensitive classification.

### 2.1.2 Further settings

The Task help page also lists several other arguments to provide further specifications 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 resampling.

Another option is to assign **`weights`** to observations. These can 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 the training tutorial or makeWeightedClassesWrapper.

### 2.1.3 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 task description 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 task descriptions have slightly different elements for different types of Tasks. 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"
```

For convenience, 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))
#>      y1        y2         y3
#> [1,]  0 1589.5664  674.44434
#> [2,]  0 1173.4364  828.40682
#> [3,]  0  942.7611 1095.33713
#> [4,]  0 1049.5562  477.82496
#> [5,]  0 1121.8899   90.85237
#> [6,]  0 1819.9830  841.06686
```

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.

### 2.1.4 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 detailed explanations of common preprocessing tasks, see data preprocessing tutorial.

### 2.1.5 Example tasks and convenience functions

For your convenience, mlr provides pre-defined Tasks for each type of learning problem. These are also used throughout this tutorial. A list of all Tasks can be found in the Appendix.

Use mlr's function convertMLBenchObjToTask to generate Tasks from the data sets and data generating functions in package mlbench.

## 2.2 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, and mlr is specifically designed to make extensions simple.

See the integrated learners page to reference already implemented machine learning methods and their properties. If your favorite method is missing, either open an issue or integrate a learning method yourself.

This basic introduction demonstrates how to use already implemented learners.

### 2.2.1 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. You may also:

- Set hyperparameters
- Control the prediction output type, 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 label 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. We can avoid this by setting `fix.factors.prediction = TRUE` to add 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. 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.

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 cost-sensitive classification tutorial.

### 2.2.2 Accessing a learner

The Learner 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 containing the type of hyperparameters (e.g., numeric, logical), potential default values and the range of allowed values.

mlr provides function getHyperPars or its alternative getLearnerParVals to access the current hyperparameter setting of a Learner and getParamSet to get a description of all possible settings.

These are particularly useful with wrapped Learners, such as a learner fused with a feature selection strategy, where both the learner and feature selection strategy have hyperparameters. For details see the wrapped learners tutorial.

```
### 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 show the required packages for a Learner, use 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"
```

### 2.2.3 Modifying a learner

We also provide functions that enable you to change certain aspects of a Learner without needing to create a new Learner 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"))
```

### 2.2.4 Listing learners

See the Appendix for a list of all learners integrated in mlr along with their respective properties.

If you would like a list of available learners with certain properties or suitable for a particular 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:
```

## 2.3 Training a Learner

Training a learner means fitting a model to a given data set. In mlr this can be done by calling function train on a Learner and a suitable Task.

We start with a classification example and perform a linear discriminant analysis on the iris data set.

```
### Generate the task
task = makeClassifTask(data = iris, target = "Species")

### Generate the learner
lrn = makeLearner("classif.lda")

### Train the learner
mod = train(lrn, task)
mod
#> Model for learner.id=classif.lda; learner.class=classif.lda
#> Trained on: task.id = iris; obs = 150; features = 4
#> Hyperparameters:
```

In the above example creating the Learner explicitly is not absolutely necessary. As a general rule, you have to generate the Learner yourself if you want to change any defaults, e.g., setting hyperparameter values or altering the predict type. Otherwise, train and many other functions also accept the class name of the learner and call makeLearner internally with default settings.

```
mod = train("classif.lda", task)
mod
#> Model for learner.id=classif.lda; learner.class=classif.lda
#> Trained on: task.id = iris; obs = 150; features = 4
#> Hyperparameters:
```

Training a learner works the same way for every type of learning problem. Below is a survival analysis example where a Cox proportional hazards model is fitted to the lung data set. Note that we use the corresponding lung.task provided by mlr. All available Tasks are listed in the Appendix.

```
mod = train("surv.coxph", lung.task)
mod
#> Model for learner.id=surv.coxph; learner.class=surv.coxph
#> Trained on: task.id = lung-example; obs = 167; features = 8
#> Hyperparameters:
```

### 2.3.1 Accessing learner models

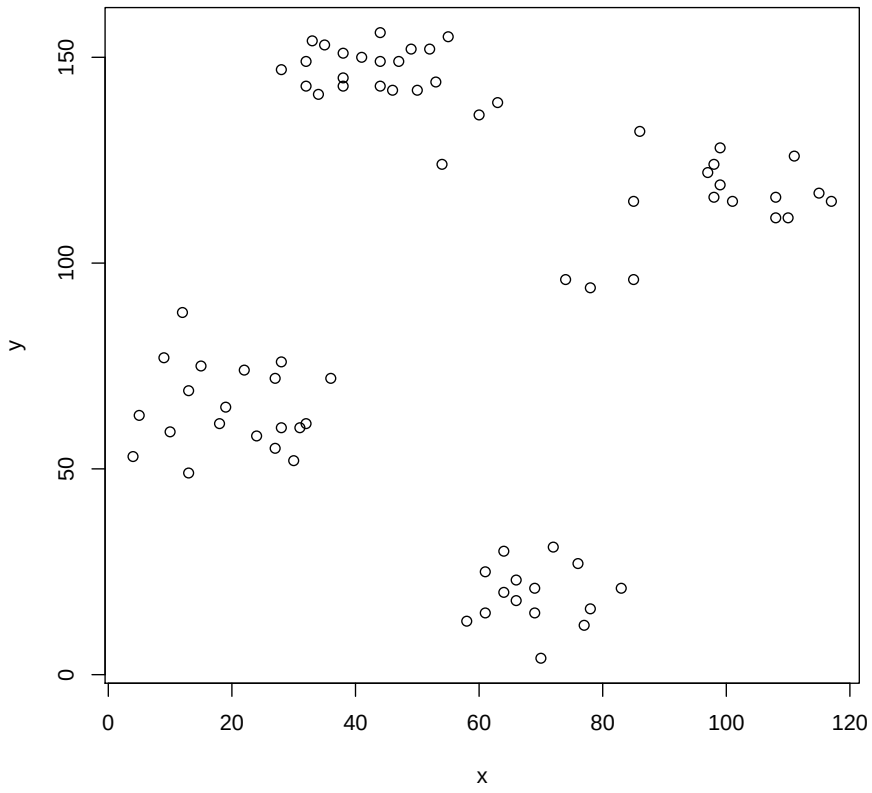
Function train returns an object of class WrappedModel, which encapsulates the fitted model, i.e., the output of the underlying **R** learning method. Additionally,

it contains some information about the Learner, the Task, the features and observations used for training, and the training time. A WrappedModel can subsequently be used to make a prediction for new observations.

The fitted model in slot `$learner.model` of the WrappedModel object can be accessed using function getLearnerModel.

In the following example we cluster the Ruspini data set (which has four groups and two features) by $K$-means with $K = 4$ and extract the output of the underlying kmeans function.

```
data(ruspini, package = "cluster")
plot(y ~ x, ruspini)
```




```
### Generate the task
ruspini.task = makeClusterTask(data = ruspini)

### Generate the learner
```

```
lrn = makeLearner("cluster.kmeans", centers = 4)

### Train the learner
mod = train(lrn, ruspini.task)
mod
#> Model for learner.id=cluster.kmeans; learner.class=cluster.kmeans
#> Trained on: task.id = ruspini; obs = 75; features = 2
#> Hyperparameters: centers=4

### Peak into mod
names(mod)
#> [1] "learner"       "learner.model" "task.desc"     "subset"
#> [5] "features"      "factor.levels" "time"          "dump"

mod$learner
#> Learner cluster.kmeans from package stats,clue
#> Type: cluster
#> Name: K-Means; Short name: kmeans
#> Class: cluster.kmeans
#> Properties: numerics,prob
#> Predict-Type: response
#> Hyperparameters: centers=4

mod$features
#> [1] "x" "y"

mod$time
#> [1] 0.001

### Extract the fitted model
getLearnerModel(mod)
#> K-means clustering with 4 clusters of sizes 23, 17, 15, 20
#>
#> Cluster means:
#>          x        y
#> 1 43.91304 146.0435
#> 2 98.17647 114.8824
#> 3 68.93333  19.4000
#> 4 20.15000  64.9500
#>
#> Clustering vector:
#>  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
   23 24 25
#>  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  1  1
    1  1  1
#> 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
```

```
   48 49 50
#>  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  2  2  2  2
     2  2  2
#> 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72
    73 74 75
#>  2  2  2  2  2  2  2  2  2  2  3  3  3  3  3  3  3  3  3  3  3
     3  3  3
#>
#> Within cluster sum of squares by cluster:
#> [1] 3176.783 4558.235 1456.533 3689.500
#>  (between_SS / total_SS =  94.7 %)
#>
#> Available components:
#>
#> [1] "cluster"      "centers"      "totss"        "withinss"
#> [5] "tot.withinss" "betweenss"    "size"         "iter"
#> [9] "ifault"
```

### 2.3.2 Further options and comments

By default, the whole data set in the Task is used for training. The **subset** argument of train takes a logical or integer vector that indicates which observations to use, for example if you want to split your data into a training and a test set or if you want to fit separate models to different subgroups in the data.

Below we fit a linear regression model to the BostonHousing data set (bh.task) and randomly select 1/3 of the data set for training.

```
### Get the number of observations
n = getTaskSize(bh.task)

### Use 1/3 of the observations for training
train.set = sample(n, size = n/3)

### Train the learner
mod = train("regr.lm", bh.task, subset = train.set)
mod
#> Model for learner.id=regr.lm; learner.class=regr.lm
#> Trained on: task.id = BostonHousing-example; obs = 168; features
    = 13
#> Hyperparameters:
```

Note, for later, that all standard resampling strategies are supported. Therefore you usually do not have to subset the data yourself.

Moreover, if the learner supports this, you can specify observation **weights** that reflect the relevance of observations in the training process. Weights can

be useful in many regards, for example to express the reliability of the training observations, reduce the influence of outliers or, if the data were collected over a longer time period, increase the influence of recent data. In supervised classification weights can be used to incorporate misclassification costs or account for class imbalance.

For example in the BreastCancer data set class `benign` is almost twice as frequent as class `malignant`. In order to grant both classes equal importance in training the classifier we can weight the examples according to the inverse class frequencies in the data set as shown in the following **R** code.

```
### Calculate the observation weights
target = getTaskTargets(bc.task)
tab = as.numeric(table(target))
w = 1/tab[target]

train("classif.rpart", task = bc.task, weights = w)
#> Model for learner.id=classif.rpart; learner.class=classif.rpart
#> Trained on: task.id = BreastCancer_example; obs = 683; features =
    9
#> Hyperparameters: xval=0
```

Note, for later, that mlr offers much more functionality to deal with imbalanced classification problems.

As another side remark for more advanced readers: By varying the weights in the calls to train, you could also implement your own variant of a general boosting type algorithm on arbitrary mlr base learners.

As you may recall, it is also possible to set observation weights when creating the Task. As a general rule, you should specify them in make*Task if the weights really "belong" to the task and always should be used. Otherwise, pass them to train. The weights in train take precedence over the weights in Task.

## 2.4 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 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 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).

In the following example we fit a gradient boosting machine to every second observation of the 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: 0.10
#>    id truth response
#> 2   2  21.6 22.28539
#> 4   4  33.4 23.33968
#> 6   6  28.7 22.40896
#> 8   8  27.1 22.12750
#> 10 10  18.9 22.12750
#> 12 12  18.9 22.12750
#> ... (#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         2
#> 4         2
#> 6         2
#> 8         2
#> 10        2
#> 12        2
#> ... (#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.

### 2.4.1 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 for direct access.

In the following the predictions on the BostonHousing and the 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 22.28539
#> 4   4  33.4 23.33968
#> 6   6  28.7 22.40896
#> 8   8  27.1 22.12750
#> 10 10  18.9 22.12750
#> 12 12  18.9 22.12750

### Result of predict with data passed via newdata argument
head(as.data.frame(newdata.pred))
#>    response
#> 2         2
#> 4         2
#> 6         2
#> 8         2
#> 10        2
#> 12        2
```

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 and getPredictionResponse.

```
head(getPredictionTruth(task.pred))
#> [1] 21.6 33.4 28.7 27.1 18.9 18.9

head(getPredictionResponse(task.pred))
#> [1] 22.28539 23.33968 22.40896 22.12750 22.12750 22.12750
```

#### 2.4.1.1 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.bgplm
#> 4     regr.blm
#> 5    regr.btgp
#> 6 regr.btgplm
#>

    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 on the Boston Housing 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
```

#### 2.4.1.2 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 on the 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         0.97959529 0.020404714
#> Mazda RX4 Wag     0.97963550 0.020364495
#> Datsun 710        0.99265984 0.007340164
#> Hornet 4 Drive    0.54292079 0.457079211
#> Hornet Sportabout 0.01870622 0.981293776
#> Valiant           0.75746556 0.242534444
```

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 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
```

#### 2.4.1.3 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
```

### 2.4.2 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 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 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: 0.01
#>   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.

### 2.4.3 Visualizing the prediction

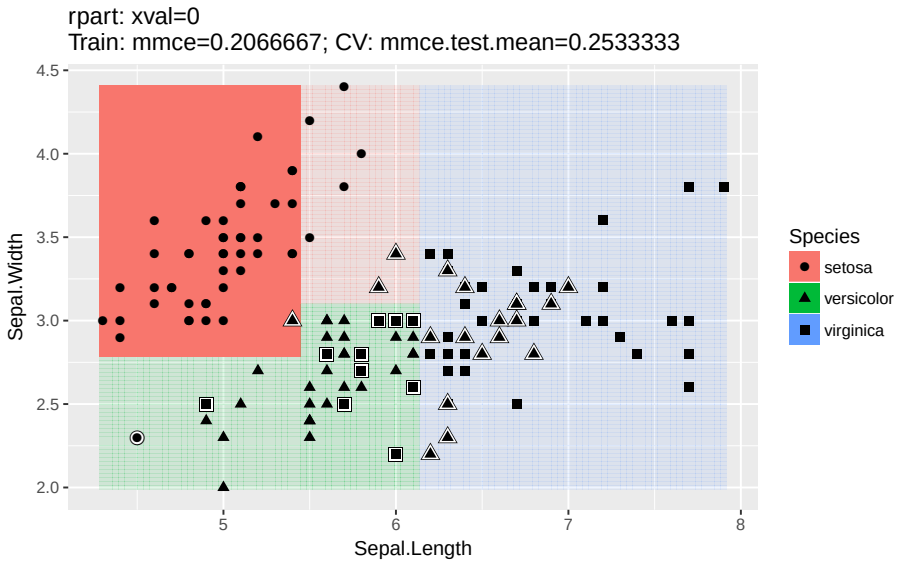
The function plotLearnerPrediction allows to visualize predictions, e.g., for teaching purposes or exploring models. It trains the chosen learning method for 1 or 2 selected features and then displays the predictions with ggplot.

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.

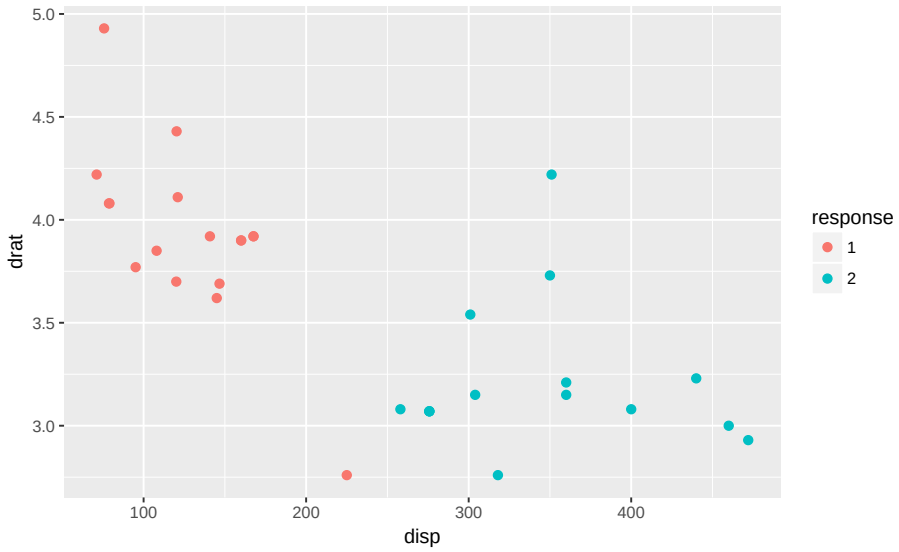
The plot title displays the ID of the Learner (in the following example CART), its parameters, its training performance and its cross-validation performance. mmce stands for *mean misclassification error*, i.e., the error rate. See the sections on performance and resampling for further explanations.

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



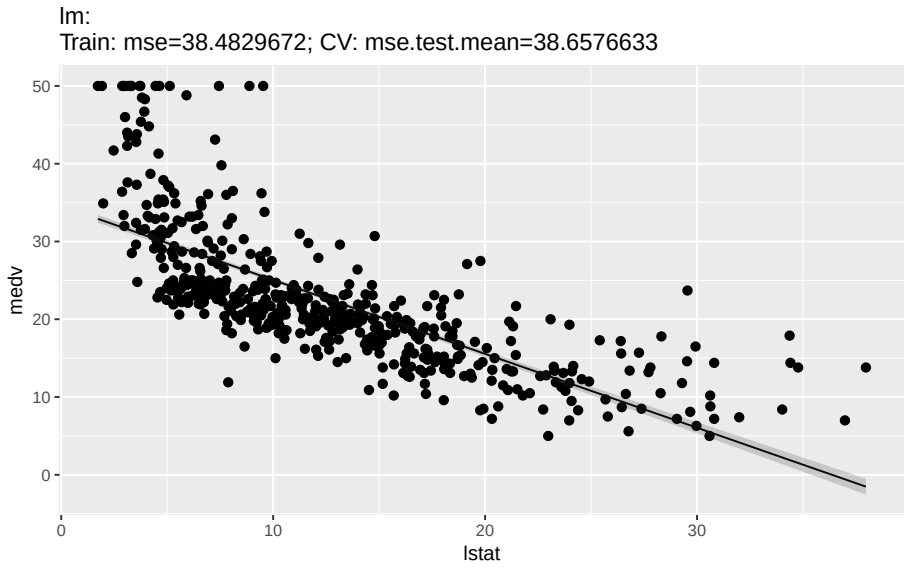

For *clustering* we also get a scatter plot of two selected features. The color of the points indicates the predicted cluster.

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




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.

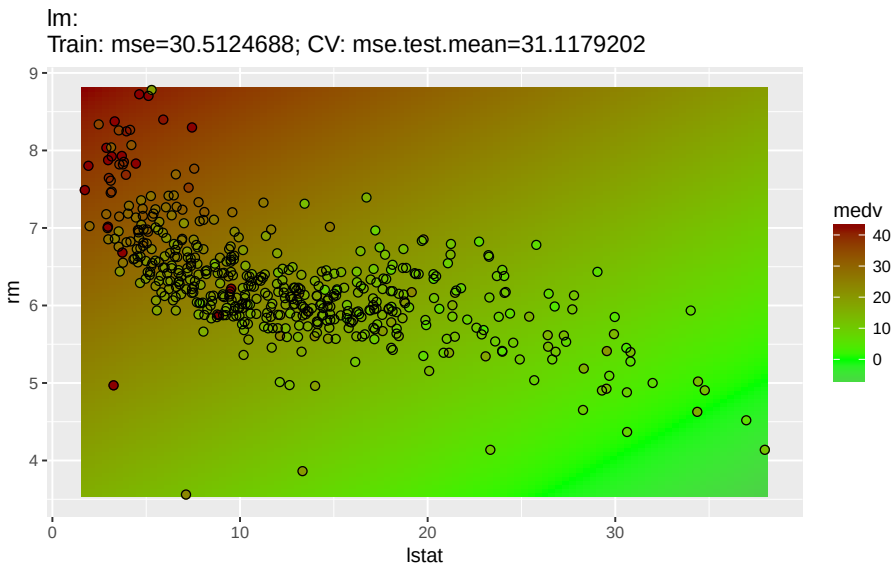
```
plotLearnerPrediction("regr.lm", features = "lstat", task = bh.task)
```




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)
```




## 2.5 Data Preprocessing

Data preprocessing refers to any transformation of the data done before applying a learning algorithm. This comprises for example finding and resolving inconsistencies, imputation of missing values, identifying, removing or replacing outliers, discretizing numerical data or generating numerical dummy variables for categorical data, any kind of transformation like standardization of predictors or Box-Cox, dimensionality reduction and feature extraction and/or selection.

mlr offers several options for data preprocessing. Some of the following simple methods to change a Task (or data.frame) were already mentioned on the page about learning tasks:

- capLargeValues: Convert large/infinite numeric values.
- createDummyFeatures: Generate dummy variables for factor features.
- dropFeatures: Remove selected features.
- joinClassLevels: Only for classification: Merge existing classes to new, larger classes.
- mergeSmallFactorLevels: Merge infrequent levels of factor features.
- normalizeFeatures: Normalize features by different methods, e.g., standardization or scaling to a certain range.

- removeConstantFeatures: Remove constant features.
- subsetTask: Remove observations and/or features from a Task.

Moreover, there are tutorial pages devoted to

- Feature selection and
- Imputation of missing values.

### 2.5.1 Fusing learners with preprocessing

mlr's wrapper functionality permits to combine learners with preprocessing steps. This means that the preprocessing "belongs" to the learner and is done any time the learner is trained or predictions are made.

This is, on the one hand, very practical. You don't need to change any data or learning Tasks and it's quite easy to combine different learners with different preprocessing steps.

On the other hand this helps to avoid a common mistake in evaluating the performance of a learner with preprocessing: Preprocessing is often seen as completely independent of the later applied learning algorithms. When estimating the performance of the a learner, e.g., by cross-validation all preprocessing is done beforehand on the full data set and only training/predicting the learner is done on the train/test sets. Depending on what exactly is done as preprocessing this can lead to overoptimistic results. For example if imputation by the mean is done on the whole data set before evaluating the learner performance you are using information from the test data during training, which can cause overoptimistic performance results.

To clarify things one should distinguish between *data-dependent* and *data-independent* preprocessing steps: Data-dependent steps in some way learn from the data and give different results when applied to different data sets. Data-independent steps always lead to the same results. Clearly, correcting errors in the data or removing data columns like Ids that should not be used for learning, is data-independent. Imputation of missing values by the mean, as mentioned above, is data-dependent. Imputation by a fixed constant, however, is not.

To get a honest estimate of learner performance combined with preprocessing, all data-dependent preprocessing steps must be included in the resampling. This is automatically done when fusing a learner with preprocessing.

To this end mlr provides two wrappers:

- makePreprocWrapperCaret is an interface to all preprocessing options offered by caret's preProcess function.

- makePreprocWrapper permits to write your own custom preprocessing methods by defining the actions to be taken before training and before prediction.

As mentioned above the specified preprocessing steps then "belong" to the wrapped Learner. In contrast to the preprocessing options listed above like normalizeFeatures

- the Task itself remains unchanged,
- the preprocessing is not done globally, i.e., for the whole data set, but for every pair of training/test data sets in, e.g., resampling,
- any parameters controlling the preprocessing as, e.g., the percentage of outliers to be removed can be tuned together with the base learner parameters.

We start with some examples for makePreprocWrapperCaret.

### 2.5.2 Preprocessing with makePreprocWrapperCaret

makePreprocWrapperCaret is an interface to caret's preProcess function that provides many different options like imputation of missing values, data transformations as scaling the features to a certain range or Box-Cox and dimensionality reduction via Independent or Principal Component Analysis. For all possible options see the help page of function preProcess.

Note that the usage of makePreprocWrapperCaret is slightly different than that of preProcess.

- makePreprocWrapperCaret takes (almost) the same formal arguments as preProcess, but their names are prefixed by `ppc.`.
- The only exception: makePreprocWrapperCaret does not have a `method` argument. Instead all preprocessing options that would be passed to preProcess's `method` argument are given as individual logical parameters to makePreprocWrapperCaret.

For example the following call to preProcess

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

with `x` being a matrix or data.frame would thus translate into

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

where `learner` is a mlr Learner or the name of a learner class like `"classif.lda"`.

If you enable multiple preprocessing options (like knn imputation and principal component analysis above) these are executed in a certain order detailed on the help page of function preProcess.

In the following we show an example where principal components analysis (PCA) is used for dimensionality reduction. This should never be applied blindly, but can be beneficial with learners that get problems with high dimensionality or those that can profit from rotating the data.

We consider the sonar.task, which poses a binary classification problem with 208 observations and 60 features.

```
sonar.task
#> Supervised task: Sonar_example
#> Type: classif
#> Target: Class
#> Observations: 208
#> Features:
#>    numerics     factors     ordered functionals
#>          60           0           0           0
#> Missings: FALSE
#> Has weights: FALSE
#> Has blocking: FALSE
#> Is spatial: FALSE
#> Classes: 2
#>   M   R
#> 111  97
#> Positive class: M
```

Below we fuse quadratic discriminant analysis from package MASS with a principal components preprocessing step. The threshold is set to 0.9, i.e., the principal components necessary to explain a cumulative percentage of 90% of the total variance are kept. The data are automatically standardized prior to PCA.

```
lrn = makePreprocWrapperCaret("classif.qda", ppc.pca = TRUE,
    ppc.thresh = 0.9)
lrn
#> Learner classif.qda.preproc from package MASS
#> Type: classif
#> Name: ; Short name:
#> Class: PreprocWrapperCaret
#> Properties: twoclass,multiclass,numerics,factors,prob
#> Predict-Type: response
#> Hyperparameters:
    ppc.BoxCox=FALSE,ppc.YeoJohnson=FALSE,ppc.expoTrans=FALSE,ppc.center=TRUE,ppc.scale=TRUE,ppc
```

The wrapped learner is trained on the sonar.task. By inspecting the underlying qda model, we see that the first 22 principal components have been used for training.

```
mod = train(lrn, sonar.task)
mod
#> Model for learner.id=classif.qda.preproc;
    learner.class=PreprocWrapperCaret
#> Trained on: task.id = Sonar_example; obs = 208; features = 60
#> Hyperparameters:
    ppc.BoxCox=FALSE,ppc.YeoJohnson=FALSE,ppc.expoTrans=FALSE,ppc.center=TRUE,ppc.scale=TRUE,ppc

getLearnerModel(mod)
#> Model for learner.id=classif.qda; learner.class=classif.qda
#> Trained on: task.id = Sonar_example; obs = 208; features = 22
#> Hyperparameters:

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

```
#> M -0.01192247  0.006098658  0.01263492 -0.001224809
#> R  0.01364323 -0.006978877 -0.01445851  0.001401586
```

Below the performances of qda with and without PCA preprocessing are compared in a benchmark experiment. Note that we use stratified resampling to prevent errors in qda due to a too small number of observations from either class.

```
rin = makeResampleInstance("CV", iters = 3, stratify = TRUE, task =
    sonar.task)
res = benchmark(list("classif.qda", lrn), sonar.task, rin, show.info
    = FALSE)
res
#>         task.id          learner.id mmce.test.mean
#> 1 Sonar_example         classif.qda      0.3941339
#> 2 Sonar_example classif.qda.preproc      0.2643202
```

PCA preprocessing in this case turns out to be really beneficial for the performance of Quadratic Discriminant Analysis.

#### 2.5.2.1 Joint tuning of preprocessing options and learner parameters

Let's see if we can optimize this a bit. The threshold value of 0.9 above was chosen arbitrarily and led to 22 out of 60 principal components. But maybe a lower or higher number of principal components should be used. Moreover, qda has several options that control how the class covariance matrices or class probabilities are estimated.

Those preprocessing and learner parameters can be tuned jointly. Before doing this let's first get an overview of all the parameters of the wrapped learner using function getParamSet.

```
getParamSet(lrn)
#>                      Type len     Def                    Constr
    Req
#> ppc.BoxCox        logical   -   FALSE                         -
      -
#> ppc.YeoJohnson    logical   -   FALSE                         -
      -
#> ppc.expoTrans     logical   -   FALSE                         -
      -
#> ppc.center        logical   -    TRUE                         -
      -
#> ppc.scale         logical   -    TRUE                         -
      -
#> ppc.range         logical   -   FALSE                         -
      -
```

```
#> ppc.knnImpute     logical   -   FALSE                          -
      -
#> ppc.bagImpute     logical   -   FALSE                          -
      -
#> ppc.medianImpute  logical   -   FALSE                          -
      -
#> ppc.pca           logical   -   FALSE                          -
      -
#> ppc.ica           logical   -   FALSE                          -
      -
#> ppc.spatialSign   logical   -   FALSE                          -
      -
#> ppc.corr          logical   -   FALSE                          -
      -
#> ppc.zv            logical   -   FALSE                          -
      -
#> ppc.nzv           logical   -   FALSE                          -
      -
#> ppc.thresh        numeric   -    0.95                  0 to Inf
      -
#> ppc.pcaComp       integer   -       -                  1 to Inf
      -
#> ppc.na.remove     logical   -    TRUE                          -
      -
#> ppc.k             integer   -       5                  1 to Inf
      -
#> ppc.fudge         numeric   -     0.2                  0 to Inf
      -
#> ppc.numUnique     integer   -       3                  1 to Inf
      -
#> ppc.n.comp        integer   -       -                  1 to Inf
      -
#> ppc.cutoff        numeric   -     0.9                    0 to 1
      -
#> ppc.freqCut       numeric   -      19                  1 to Inf
      -
#> ppc.uniqueCut     numeric   -      10                  0 to Inf
      -
#> method           discrete   -  moment           moment,mle,mve,t
      -
#> nu                numeric   -       5                  2 to Inf
      Y
#> predict.method   discrete   - plug-in plug-in,predictive,debiased
      -
#>                 Tunable Trafo
#> ppc.BoxCox         TRUE     -
```

```
#> ppc.YeoJohnson      TRUE     -
#> ppc.expoTrans       TRUE     -
#> ppc.center          TRUE     -
#> ppc.scale           TRUE     -
#> ppc.range           TRUE     -
#> ppc.knnImpute       TRUE     -
#> ppc.bagImpute       TRUE     -
#> ppc.medianImpute    TRUE     -
#> ppc.pca             TRUE     -
#> ppc.ica             TRUE     -
#> ppc.spatialSign     TRUE     -
#> ppc.corr            TRUE     -
#> ppc.zv              TRUE     -
#> ppc.nzv             TRUE     -
#> ppc.thresh          TRUE     -
#> ppc.pcaComp         TRUE     -
#> ppc.na.remove       TRUE     -
#> ppc.k               TRUE     -
#> ppc.fudge           TRUE     -
#> ppc.numUnique       TRUE     -
#> ppc.n.comp          TRUE     -
#> ppc.cutoff          TRUE     -
#> ppc.freqCut         TRUE     -
#> ppc.uniqueCut       TRUE     -
#> method              TRUE     -
#> nu                  TRUE     -
#> predict.method      TRUE     -
```

The parameters prefixed by `ppc.` belong to preprocessing. `method`, `nu` and `predict.method` are qda parameters.

Instead of tuning the PCA threshold (`ppc.thresh`) we tune the number of principal components (`ppc.pcaComp`) directly. Moreover, for qda we try two different ways to estimate the posterior probabilities (parameter `predict.method`): the usual plug-in estimates and unbiased estimates.

We perform a grid search and set the resolution to 10. This is for demonstration. You might want to use a finer resolution.

```
ps = makeParamSet(
  makeIntegerParam("ppc.pcaComp", lower = 1, upper =
      getTaskNFeats(sonar.task)),
  makeDiscreteParam("predict.method", values = c("plug-in",
      "debiased"))
)
ctrl = makeTuneControlGrid(resolution = 10)
res = tuneParams(lrn, sonar.task, rin, par.set = ps, control = ctrl,
    show.info = FALSE)
```

```
res
#> Tune result:
#> Op. pars: ppc.pcaComp=8; predict.method=plug-in
#> mmce.test.mean=0.1920635

as.data.frame(res$opt.path)[1:3]
#>    ppc.pcaComp predict.method mmce.test.mean
#> 1            1        plug-in      0.4757074
#> 2            8        plug-in      0.1920635
#> 3           14        plug-in      0.2162871
#> 4           21        plug-in      0.2643202
#> 5           27        plug-in      0.2454106
#> 6           34        plug-in      0.2645273
#> 7           40        plug-in      0.2742581
#> 8           47        plug-in      0.3173223
#> 9           53        plug-in      0.3512767
#> 10          60        plug-in      0.3941339
#> 11           1       debiased      0.5336094
#> 12           8       debiased      0.2450656
#> 13          14       debiased      0.2403037
#> 14          21       debiased      0.2546584
#> 15          27       debiased      0.3075224
#> 16          34       debiased      0.3172533
#> 17          40       debiased      0.3125604
#> 18          47       debiased      0.2979986
#> 19          53       debiased      0.3079365
#> 20          60       debiased      0.3654244
```

There seems to be a preference for a lower number of principal components (<27) for both `"plug-in"` and `"debiased"` with `"plug-in"` achieving slightly lower error rates.

### 2.5.3 Writing a custom preprocessing wrapper

If the options offered by makePreprocWrapperCaret are not enough, you can write your own preprocessing wrapper using function makePreprocWrapper.

As described in the tutorial section about wrapped learners wrappers are implemented using a *train* and a *predict* method. In case of preprocessing wrappers these methods specify how to transform the data before training and before prediction and are *completely user-defined*.

Below we show how to create a preprocessing wrapper that centers and scales the data before training/predicting. Some learning methods as, e.g., k nearest neighbors, support vector machines or neural networks usually require scaled features. Many, but not all, have a built-in scaling option where the training

data set is scaled before model fitting and the test data set is scaled accordingly, that is by using the scaling parameters from the training stage, before making predictions. In the following we show how to add a scaling option to a Learner by coupling it with function scale.

Note that we chose this simple example for demonstration. Centering/scaling the data is also possible with makePreprocWrapperCaret.

#### 2.5.3.1 Specifying the train function

The *train* function has to be a function with the following arguments:

- **data** is a data.frame with columns for all features and the target variable.
- target is a string and denotes the name of the target variable in **data**.
- **args** is a list of further arguments and parameters that influence the preprocessing.

It must return a list with elements **$data** and **$control**, where **$data** is the preprocessed data set and **$control** stores all information required to preprocess the data before prediction.

The *train* function for the scaling example is given below. It calls scale on the numerical features and returns the scaled training data and the corresponding scaling parameters.

**args** contains the center and **scale** arguments of function scale and slot **$control** stores the scaling parameters to be used in the prediction stage.

Regarding the latter note that the center and **scale** arguments of scale can be either a logical value or a numeric vector of length equal to the number of the numeric columns in **data**, respectively. If a logical value was passed to **args** we store the column means and standard deviations/ root mean squares in the $center and **$scale** slots of the returned **$control** object.

```
trainfun = function(data, target, args = list(center, scale)) {
  ## Identify numerical features
  cns = colnames(data)
  nums = setdiff(cns[sapply(data, is.numeric)], target)
  ## Extract numerical features from the data set and call scale
  x = as.matrix(data[, nums, drop = FALSE])
  x = scale(x, center = args$center, scale = args$scale)
  ## Store the scaling parameters in control
  ## These are needed to preprocess the data before prediction
  control = args
  if (is.logical(control$center) && control$center)
    control$center = attr(x, "scaled:center")
  if (is.logical(control$scale) && control$scale)
    control$scale = attr(x, "scaled:scale")
```

```
  ## Recombine the data
  data = data[, setdiff(cns, nums), drop = FALSE]
  data = cbind(data, as.data.frame(x))
  return(list(data = data, control = control))
}
```

#### 2.5.3.2 Specifying the predict function

The *predict* function has the following arguments:

- `data` is a data.frame containing *only* feature values (as for prediction the target values naturally are not known).
- `target` is a string indicating the name of the target variable.
- `args` are the `args` that were passed to the *train* function.
- `control` is the object returned by the *train* function.

It returns the preprocessed data.

In our scaling example the *predict* function scales the numerical features using the parameters from the training stage stored in `control`.

```
predictfun = function(data, target, args, control) {
  ## Identify numerical features
  cns = colnames(data)
  nums = cns[sapply(data, is.numeric)]
  ## Extract numerical features from the data set and call scale
  x = as.matrix(data[, nums, drop = FALSE])
  x = scale(x, center = control$center, scale = control$scale)
  ## Recombine the data
  data = data[, setdiff(cns, nums), drop = FALSE]
  data = cbind(data, as.data.frame(x))
  return(data)
}
```

#### 2.5.3.3 Creating the preprocessing wrapper

Below we create a preprocessing wrapper with a regression neural network (which itself does not have a scaling option) as base learner.

The *train* and *predict* functions defined above are passed to makePreprocWrapper via the `train` and `predict` arguments. `par.vals` is a list of parameter values that is relayed to the `args` argument of the *train* function.

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

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

Let's compare the cross-validated mean squared error (mse) on the Boston Housing data set with and without scaling.

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

r = resample(lrn, bh.task, resampling = rdesc, show.info = FALSE)
r
#> Resample Result
#> Task: BostonHousing-example
#> Learner: regr.nnet.preproc
#> Aggr perf: mse.test.mean=20.6204314
#> Runtime: 0.163055

lrn = makeLearner("regr.nnet", trace = FALSE, decay = 1e-02)
r = resample(lrn, bh.task, resampling = rdesc, show.info = FALSE)
r
#> Resample Result
#> Task: BostonHousing-example
#> Learner: regr.nnet
#> Aggr perf: mse.test.mean=55.0639088
#> Runtime: 0.120283
```

#### 2.5.3.4 Joint tuning of preprocessing and learner parameters

Often it's not clear which preprocessing options work best with a certain learning algorithm. As already shown for the number of principal components in makePreprocWrapperCaret we can tune them easily together with other hyperparameters of the learner.

In our scaling example we can try if nnet works best with both centering and scaling the data or if it's better to omit one of the two operations or do no preprocessing at all. In order to tune `center` and `scale` we have to add appropriate LearnerParams to the parameter set of the wrapped learner.

As mentioned above scale allows for numeric and logical `center` and `scale` arguments. As we want to use the latter option we declare `center` and `scale` as logical learner parameters.

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

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

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

Now we do a simple grid search for the `decay` parameter of nnet and the `center` and `scale` parameters.

```
rdesc = makeResampleDesc("Holdout")
ps = makeParamSet(
  makeDiscreteParam("decay", c(0, 0.05, 0.1)),
  makeLogicalParam("center"),
  makeLogicalParam("scale")
)
ctrl = makeTuneControlGrid()
```

```
res = tuneParams(lrn, bh.task, rdesc, par.set = ps, control = ctrl,
    show.info = FALSE)

res
#> Tune result:
#> Op. pars: decay=0.05; center=FALSE; scale=TRUE
#> mse.test.mean=14.8430579

as.data.frame(res$opt.path)
#>    decay center scale mse.test.mean dob eol error.message
    exec.time
#> 1      0   TRUE  TRUE      49.38128   1  NA          <NA>
    0.045
#> 2   0.05   TRUE  TRUE      20.64761   2  NA          <NA>
    0.053
#> 3    0.1   TRUE  TRUE      22.42986   3  NA          <NA>
    0.053
#> 4      0  FALSE  TRUE      96.25474   4  NA          <NA>
    0.029
#> 5   0.05  FALSE  TRUE      14.84306   5  NA          <NA>
    0.058
#> 6    0.1  FALSE  TRUE      16.65383   6  NA          <NA>
    0.050
#> 7      0   TRUE FALSE      40.51518   7  NA          <NA>
    0.054
#> 8   0.05   TRUE FALSE      68.00069   8  NA          <NA>
    0.049
#> 9    0.1   TRUE FALSE      55.42210   9  NA          <NA>
    0.051
#> 10     0  FALSE FALSE      96.25474  10  NA          <NA>
    0.029
#> 11  0.05  FALSE FALSE      56.25758  11  NA          <NA>
    0.051
#> 12   0.1  FALSE FALSE      42.85529  12  NA          <NA>
    0.061
```

#### 2.5.3.5 Preprocessing wrapper functions

If you have written a preprocessing wrapper that you might want to use from time to time it's a good idea to encapsulate it in an own function as shown below. If you think your preprocessing method is something others might want to use as well and should be integrated into mlr just contact us.

```
makePreprocWrapperScale = function(learner, center = TRUE, scale =
    TRUE) {
  trainfun = function(data, target, args = list(center, scale)) {
```

```
    cns = colnames(data)
    nums = setdiff(cns[sapply(data, is.numeric)], target)
    x = as.matrix(data[, nums, drop = FALSE])
    x = scale(x, center = args$center, scale = args$scale)
    control = args
    if (is.logical(control$center) && control$center)
      control$center = attr(x, "scaled:center")
    if (is.logical(control$scale) && control$scale)
      control$scale = attr(x, "scaled:scale")
    data = data[, setdiff(cns, nums), drop = FALSE]
    data = cbind(data, as.data.frame(x))
    return(list(data = data, control = control))
  }
  predictfun = function(data, target, args, control) {
    cns = colnames(data)
    nums = cns[sapply(data, is.numeric)]
    x = as.matrix(data[, nums, drop = FALSE])
    x = scale(x, center = control$center, scale = control$scale)
    data = data[, setdiff(cns, nums), drop = FALSE]
    data = cbind(data, as.data.frame(x))
    return(data)
  }
  makePreprocWrapper(
    learner,
    train = trainfun,
    predict = predictfun,
    par.set = makeParamSet(
      makeLogicalLearnerParam("center"),
      makeLogicalLearnerParam("scale")
    ),
    par.vals = list(center = center, scale = scale)
  )
}

lrn = makePreprocWrapperScale("classif.lda")
train(lrn, iris.task)
#> Model for learner.id=classif.lda.preproc;
    learner.class=PreprocWrapper
#> Trained on: task.id = iris_example; obs = 150; features = 4
#> Hyperparameters: center=TRUE,scale=TRUE
```

## 2.6 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 and specify the desired performance measures.

### 2.6.1 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), accuracy (acc) or measures based on ROC analysis. For *regression* the mean of squared errors (mse) or mean of absolute errors (mae) are usually considered. For *clustering* tasks, measures such as the Dunn index (dunn) are provided, while for *survival* predictions, the Concordance Index (cindex) is supported, and for *cost-sensitive* predictions the misclassification penalty (mcp) and others. It is also possible to access the time to train the learner (timetrain), the time to compute the prediction (timepredict) and their sum (timeboth) as performance measures.

To see which performance measures are implemented, have a look at the table of performance 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.

### 2.6.2 Listing measures

The properties and requirements of the individual measures are shown in the table of performance 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] "featperc"         "mmce"             "lsr"
#>  [4] "qsr"              "timeboth"         "multiclass.aunp"
#>  [7] "timetrain"        "multiclass.aunu"  "ber"
#> [10] "timepredict"      "multiclass.brier" "ssr"
#> [13] "acc"              "logloss"          "wkappa"
#> [16] "multiclass.au1p"  "multiclass.au1u"  "kappa"
### Performance measure suitable for the iris classification task
listMeasures(iris.task)
#>  [1] "featperc"         "mmce"             "lsr"
#>  [4] "qsr"              "timeboth"         "multiclass.aunp"
#>  [7] "timetrain"        "multiclass.aunu"  "ber"
#> [10] "timepredict"      "multiclass.brier" "ssr"
#> [13] "acc"              "logloss"          "wkappa"
#> [16] "multiclass.au1p"  "multiclass.au1u"  "kappa"
```

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) for regression and the misclassification rate (mmce) 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)
```

### 2.6.3 Calculate performance measures

In the following example we fit a gradient boosting machine on a subset of the BostonHousing data set and calculate the default measure mean squared error (mse) 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
#> 42.68414
```

The following code computes the median of squared errors (medse) instead.

```
performance(pred, measures = medse)
#>    medse
#> 9.134965
```

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.

Calculate the mean squared error, median squared error and mean absolute error (mae).

```
performance(pred, measures = list(mse, medse, mae))
#>       mse     medse       mae
#> 42.684141  9.134965  4.536750
```

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

#### 2.6.3.1 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 in addition to the Prediction.

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

```
performance(pred, measures = timetrain, model = mod)
#> timetrain
#>     0.115
```

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
#> 0.1462919
```

Moreover, some measures require a certain type of prediction. For example in binary classification in order to calculate the AUC (auc) – 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.

```
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), are only suitable for binary problems.

### 2.6.4 Access a performance measure

Performance measures in mlr are objects of class Measure. If you are interested in the properties or requirements of a single measure you can access it directly. See the help page of Measure 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)
#>  $ 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)
#>   ..$ properties: chr "req.test"
#>   ..- attr(*, "class")= chr "Aggregation"
#>  - attr(*, "class")= chr "Measure"
```

### 2.6.5 Binary classification

For binary classification specialized techniques exist to analyze the performance.

#### 2.6.5.1 Plot performance versus threshold

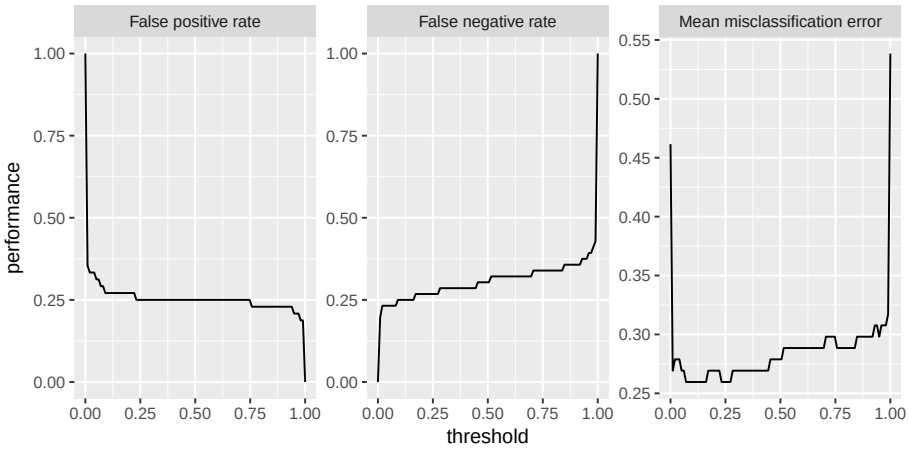
As you may recall (see the previous section on making predictions) in binary classification we can adjust the threshold used to map probabilities to class labels.

Helpful in this regard 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 here.

In the following example we consider the Sonar data set and plot the false positive rate (fpr), the false negative rate (fnr) as well as the misclassification rate (mmce) 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)
```

#### 2.6.5.2 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.

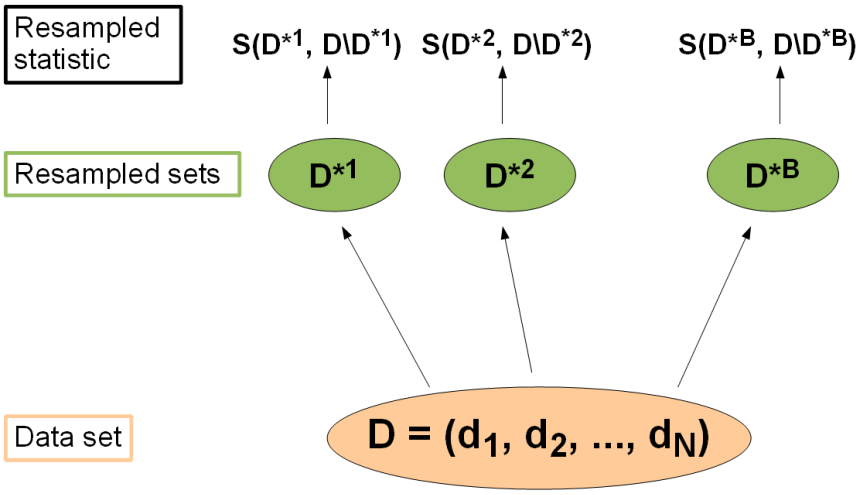
```
r = calculateROCMeasures(pred)
r
#>     predicted
#> true M         R
#>    M 39        17        tpr: 0.7  fnr: 0.3
#>    R 12        36        fpr: 0.25 tnr: 0.75
#>      ppv: 0.76 for: 0.32 lrp: 2.79 acc: 0.72
#>      fdr: 0.24 npv: 0.68 lrm: 0.4  dor: 6.88
#>
#>
#> Abbreviations:
#> tpr - True 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 $2 \times 2$ matrix is the confusion matrix, 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 method: `print(r, abbreviations = FALSE)`.

## 2.7 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.

### 2.7.1 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) as well as cross-validation with different numbers of folds (e.g., cv5 or cv10).

```
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
```

### 2.7.2 Performing the resampling

Function resample evaluates a Learner on a given machine learning Task using the selected resampling strategy.

As a first example, the performance of linear regression on the 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 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) 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:    19.8982806
#> [Resample] iter 2:    31.7803405
#> [Resample] iter 3:    20.9698523
#>
#> Aggregated Result: mse.test.mean=24.2161578
#>

r
#> Resample Result
#> Task: BostonHousing-example
#> Learner: regr.lm
#> Aggr perf: mse.test.mean=24.2161578
#> Runtime: 0.0487487
```

The result `r` is an object of class ResampleResult. 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
#>      24.21616

r$measures.test
#>   iter      mse
#> 1    1 19.89828
#> 2    2 31.78034
#> 3    3 20.96985
```

`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, and the method, test.mean, used to aggregate the 3 individual performances. test.mean 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) is evaluated on the 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 to resample. Below, the error rate (mmce), false positive and false negative rates (fpr, fnr), and the time it takes to train the learner (timetrain) 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:    0.2619048   0.2352941   0.2800000
    0.0210000
#> [Resample] iter 2:    0.2857143   0.2857143   0.2857143
    0.0200000
#> [Resample] iter 3:    0.2619048   0.2857143   0.2380952
    0.0210000
#> [Resample] iter 4:    0.3333333   0.3500000   0.3181818
    0.0210000
#> [Resample] iter 5:    0.3333333   0.1666667   0.4583333
    0.0200000
#>
#> Aggregated Result:
    mmce.test.mean=0.2952381,fpr.test.mean=0.2646779,fnr.test.mean=0.3160649,timetrain.test.mean
#>

r
#> Resample Result
#> Task: Sonar_example
```

```
#> Learner: classif.rpart
#> Aggr perf:
    mmce.test.mean=0.2952381,fpr.test.mean=0.2646779,fnr.test.mean=0.3160649,timetrain.test.mean
#> Runtime: 0.192073
```

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=0.2952381,fpr.test.mean=0.2646779,fnr.test.mean=0.3160649,timetrain.test.mean
#> Runtime: 0.192073
```

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.)

In the above example, the Learner 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=0.3047619,fpr.test.mean=0.2785319,fnr.test.mean=0.3093917,timetrain.test.mean
#> Runtime: 0.199275
```

### 2.7.3 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.

#### 2.7.3.1 Predictions

Per default, the ResampleResult 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): 0.01
#>    id truth response iter  set
#> 1  36     R        M    1 test
#> 2 132     M        R    1 test
#> 3 145     M        R    1 test
#> 4 161     M        R    1 test
#> 5 108     M        M    1 test
#> 6 178     M        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): 0.01
#>    id truth response iter  set
#> 1  36     R        M    1 test
#> 2 132     M        R    1 test
#> 3 145     M        R    1 test
#> 4 161     M        R    1 test
#> 5 108     M        M    1 test
#> 6 178     M        M    1 test
#> ... (#rows: 210, #cols: 5)
```

`pred` is an object of class ResamplePrediction. Just as a Prediction object (see the tutorial page on making predictions) 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 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  36     R        M    1 test
#> 2 132     M        R    1 test
#> 3 145     M        R    1 test
#> 4 161     M        R    1 test
#> 5 108     M        M    1 test
#> 6 178     M        M    1 test

head(getPredictionTruth(pred))
#> [1] R M M M M M
#> Levels: M R

head(getPredictionResponse(pred))
#> [1] M R R R 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=0.0200000
#> Runtime: 0.0200086

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
#> 123 123  virginica  virginica
#> 101 101  virginica  virginica
#> 51   51 versicolor versicolor
#> 45   45     setosa     setosa
#> 46   46     setosa     setosa
#> 3     3     setosa     setosa
#> ... (#rows: 100, #cols: 3)
#>
#>
#> $test
#> $test$`1`
#> Prediction: 50 observations
#> predict.type: response
#> threshold:
#> time: 0.00
#>      id      truth   response
#> 109 109  virginica  virginica
#> 80   80 versicolor versicolor
#> 40   40     setosa     setosa
#> 140 140  virginica  virginica
#> 125 125  virginica  virginica
#> 10   10     setosa     setosa
#> ... (#rows: 50, #cols: 3)
```

#### 2.7.3.2 Learner models

In each resampling iteration a Learner is fitted on the respective training set. By default, the resulting WrappedModels are not included in the ResampleResult

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:
```

#### 2.7.3.3 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 object fitted in each resampling iteration.

Below, we cluster the 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
#> 1 16.23333 7.666667 308.9667 214.00000 3.400000 3.564167 16.37000
#> 2 26.00833 4.333333 113.5917  86.08333 4.040833 2.368583 18.88917
#> 3 13.33333 8.000000 444.0000 198.33333 3.003333 4.839667 17.61667
```

```
#>          vs        am     gear     carb
#> 1 0.1666667 0.3333333 3.666667 3.666667
#> 2 0.8333333 0.6666667 4.083333 1.916667
#> 3 0.0000000 0.0000000 3.000000 3.333333
#>
#> [[2]]
#>       mpg cyl     disp     hp  drat       wt     qsec    vs   am
    gear
#> 1 15.5600   8 326.0400 207.00 3.198 3.830000 16.74600 0.000 0.10
    3.200
#> 2 26.7125   4 102.8875  86.00 4.145 2.179125 19.05375 0.875 0.75
    4.125
#> 3 19.1500   6 174.4000 128.25 3.550 3.136250 17.91000 0.500 0.50
    4.000
#>    carb
#> 1 3.500
#> 2 1.625
#> 3 3.750
#>
#> [[3]]
#>        mpg cyl     disp       hp     drat       wt     qsec
          vs
#> 1 25.25000   4 113.6000  82.5000 3.932500 2.622500 19.17000
    1.0000000
#> 2 15.12000   8 369.8600 201.9000 3.211000 4.098900 17.05300
    0.0000000
#> 3 19.74286   6 183.3143 122.2857 3.585714 3.117143 17.97714
    0.5714286
#>          am     gear     carb
#> 1 0.7500000 4.000000 1.500000
#> 2 0.1000000 3.200000 3.200000
#> 3 0.4285714 3.857143 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 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 3842.839 952.3849 4443.578 90.63669 3772.273 15853.01 3997.275
    3355.651
#>        rad     tax  ptratio b    lstat
#> 1 987.4256 568.177 2860.129 0 11255.66
#>
#> [[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 3246.521 3411.444 5806.613    0 2349.776 10125.04 5692.587
    2108.059
#>        rad     tax  ptratio        b    lstat
#> 1 312.6521 2159.42 1104.839 174.6412 15871.53
#>
#> [[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 3785.852 1649.28 4942.119    0 3989.326 18426.87 2604.239
    350.8401
```

```
#>       rad      tax  ptratio        b    lstat
#> 1 800.798 2907.556 3871.556 491.6297 12505.88
```

### 2.7.4 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.

#### 2.7.4.1 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=0.0132026
#> Runtime: 0.0382466
```

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

#### 2.7.4.2 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=21.2385142
#> Runtime: 0.0612757
```

#### 2.7.4.3 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.

```
### 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
```

### 2.7.5 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 (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.

```
### 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] 88 129 94 109 108 43 72 47 137 39 ...
```

```
#>   ..$ : int [1:100] 129 94 138 83 112 54 29 36 72 137 ...
#>   ..$ : int [1:100] 88 138 109 83 112 108 54 29 36 43 ...
#>  $ test.inds :List of 3
#>   ..$ : int [1:50] 2 5 6 13 14 17 20 21 24 25 ...
#>   ..$ : int [1:50] 3 4 7 8 11 12 22 30 34 35 ...
#>   ..$ : int [1:50] 1 9 10 15 16 18 19 23 27 28 ...
#>  $ 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] 149 58 120 44 148 29 66 46 124 137 ...
#>   ..$ : int [1:100] 51 58 64 148 56 46 124 8 14 137 ...
#>   ..$ : int [1:100] 149 51 120 44 64 56 29 66 8 14 ...
#>  $ test.inds :List of 3
#>   ..$ : int [1:50] 3 8 12 14 17 22 23 24 32 34 ...
#>   ..$ : int [1:50] 1 2 4 6 10 11 13 26 29 30 ...
#>   ..$ : int [1:50] 5 7 9 15 16 18 19 20 21 25 ...
#>  $ group     : Factor w/ 0 levels:
#>  - attr(*, "class")= chr "ResampleInstance"

### Access the indices of the training observations in iteration 3
rin$train.inds[[3]]
#>   [1] 149  51 120  44  64  56  29  66   8  14  83  65  97 114  13
      3 104
#>  [18]  88 130  81  89  23  63 131  92  31  41  78  72 139  67  10
     57  12
#>  [35] 107  74  70 116  36  24  35  93 126 111  75  91  80  85  42
     30  22
#>  [52]   1  69 113  87  26  17 150 119   4 138 129 147  38  99  60
    142  50
#>  [69] 122  40 127  43  96  34 141 106  79 133 145 125 135 108  52
    109  37
#>  [86]  61  84  59  39  82  32  53  94   6  45  86  95   2  68  11
```

The result `rin` inherits from class ResampleInstance and contains lists of index

vectors for the train and test sets.

If a ResampleDesc is passed to resample, it is instantiated internally. Naturally, it is also possible to pass a ResampleInstance 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 and ResampleInstance 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
```

### 2.7.6 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).

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

```
### 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: 0x5478058>
#> <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: 0xc5a5f98>
#> <environment: namespace:mlr>
```

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

#### 2.7.6.1 Example: One measure with different aggregations

The aggregation schemes test.median, test.min, and test.max 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:    28.164474328.164474328.164474328.1644743
#> [Resample] iter 2:    17.593981817.593981817.593981817.5939818
#> [Resample] iter 3:    24.957218724.957218724.957218724.9572187
#>
#> Aggregated Result:
    mse.test.mean=23.5718916,mse.test.median=24.9572187,mse.test.min=17.5939818,mse.test.max=28.
#>

r
#> Resample Result
#> Task: BostonHousing-example
#> Learner: regr.lm
#> Aggr perf:
    mse.test.mean=23.5718916,mse.test.median=24.9572187,mse.test.min=17.5939818,mse.test.max=28.
#> Runtime: 0.0510554

r$aggr
#>   mse.test.mean mse.test.median    mse.test.min    mse.test.max
#>        23.57189        24.95722        17.59398        28.16447
```

#### 2.7.6.2 Example: Calculating the training error

Below we calculate the mean misclassification error (mmce) 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:    0.0200000    0.1000000
#> [Resample] iter 2:    0.0400000    0.0400000
#> [Resample] iter 3:    0.0400000    0.0400000
#>
#> Aggregated Result:
    mmce.test.mean=0.0600000,mmce.train.mean=0.0333333
#>

r$measures.train
#>   iter mmce mmce
#> 1    1 0.02 0.02
#> 2    2 0.04 0.04
#> 3    3 0.04 0.04

r$aggr
#>  mmce.test.mean mmce.train.mean
#>      0.06000000      0.03333333
```

#### 2.7.6.3 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 0.04000000 0.04000000 0.04000000
#> 2    2 0.04000000 0.04000000 0.04000000
#> 3    3 0.01333333 0.01333333 0.01333333
#> 4    4 0.02666667 0.02666667 0.02666667
#> 5    5 0.01333333 0.01333333 0.01333333
#> 6    6 0.02000000 0.02000000 0.02000000

### Compare misclassification rates for out-of-bag, b632, and b632+
    bootstrap
r$aggr
#> mmce.test.mean      mmce.b632  mmce.b632plus
#>     0.05804883     0.04797219     0.04860054
```

### 2.7.7 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). Moreover, mlr provides special functions for the most common resampling methods, for example holdout, crossval, or bootstrapB632.

```
crossval("classif.lda", iris.task, iters = 3, measures = list(mmce,
    ber))
#> Resampling: cross-validation
#> Measures:             mmce      ber
#> [Resample] iter 1:    0.0200000 0.0158730
#> [Resample] iter 2:    0.0400000 0.0415140
#> [Resample] iter 3:    0.0000000 0.0000000
#>
#> Aggregated Result:
    mmce.test.mean=0.0200000,ber.test.mean=0.0191290
#>
#> Resample Result
#> Task: iris_example
#> Learner: classif.lda
#> Aggr perf: mmce.test.mean=0.0200000,ber.test.mean=0.0191290
#> Runtime: 0.0445554
```

```
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:    18.9037446  3.0912153   29.2662169
    3.7698624
#> [Resample] iter 2:    17.9389954  3.0343581   26.3888260
    3.5992878
#> [Resample] iter 3:    20.9092738  3.2640991   23.7739540
    3.6788560
#>
#> Aggregated Result: mse.b632plus=23.9312510,mae.b632plus=3.4912886
#>
#> Resample Result
#> Task: BostonHousing-example
#> Learner: regr.lm
#> Aggr perf: mse.b632plus=23.9312510,mae.b632plus=3.4912886
#> Runtime: 0.0921044
```

## 2.8 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 – 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 SVM:

```
### 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 and resampling.

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 for illustration and tune the hyperparameters of an SVM (function 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 ksvm function.

### 2.8.1 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 object, which describes the parameter space we wish to search. This is done via the function 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 instead of 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:

```
num_ps = makeParamSet(
  makeNumericParam("C", lower = -10, upper = 10, trafo = function(x)
      2^x),
  makeNumericParam("sigma", lower = -10, upper = 10, trafo =
      function(x) 2^x)
)
```

Many other parameters can be created, check out the examples in 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 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!

### 2.8.2 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 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 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)
```

### 2.8.3 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 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=0.0400000; time: 0.0 min
#> [Tune-x] 2: C=1; sigma=0.5
#> [Tune-y] 2: mmce.test.mean=0.0400000; time: 0.0 min
#> [Tune-x] 3: C=1.5; sigma=0.5
#> [Tune-y] 3: mmce.test.mean=0.0466667; time: 0.0 min
#> [Tune-x] 4: C=2; sigma=0.5
#> [Tune-y] 4: mmce.test.mean=0.0466667; time: 0.0 min
#> [Tune-x] 5: C=0.5; sigma=1
#> [Tune-y] 5: mmce.test.mean=0.0400000; time: 0.0 min
#> [Tune-x] 6: C=1; sigma=1
#> [Tune-y] 6: mmce.test.mean=0.0466667; time: 0.0 min
#> [Tune-x] 7: C=1.5; sigma=1
#> [Tune-y] 7: mmce.test.mean=0.0466667; time: 0.0 min
#> [Tune-x] 8: C=2; sigma=1
#> [Tune-y] 8: mmce.test.mean=0.0466667; time: 0.0 min
#> [Tune-x] 9: C=0.5; sigma=1.5
#> [Tune-y] 9: mmce.test.mean=0.0333333; time: 0.0 min
#> [Tune-x] 10: C=1; sigma=1.5
#> [Tune-y] 10: mmce.test.mean=0.0400000; time: 0.0 min
#> [Tune-x] 11: C=1.5; sigma=1.5
#> [Tune-y] 11: mmce.test.mean=0.0400000; time: 0.0 min
#> [Tune-x] 12: C=2; sigma=1.5
#> [Tune-y] 12: mmce.test.mean=0.0466667; time: 0.0 min
#> [Tune-x] 13: C=0.5; sigma=2
#> [Tune-y] 13: mmce.test.mean=0.0400000; time: 0.0 min
#> [Tune-x] 14: C=1; sigma=2
#> [Tune-y] 14: mmce.test.mean=0.0333333; time: 0.0 min
#> [Tune-x] 15: C=1.5; sigma=2
#> [Tune-y] 15: mmce.test.mean=0.0400000; time: 0.0 min
#> [Tune-x] 16: C=2; sigma=2
#> [Tune-y] 16: mmce.test.mean=0.0400000; time: 0.0 min
#> [Tune] Result: C=0.5; sigma=1.5 : mmce.test.mean=0.0333333

res
#> Tune result:
#> Op. pars: C=0.5; sigma=1.5
#> mmce.test.mean=0.0333333
```

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) is used.

Note that each measure "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.

In the example below we calculate the accuracy (acc) instead of the error rate. We use function setAggregation, as described on the resampling 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=95.2; sigma=0.0067
#> acc.test.mean=0.9866667,acc.test.sd=0.0230940
```

### 2.8.4 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] 95.22422
#>
#> $sigma
#> [1] 0.006695534
```

```
res$y
#> acc.test.mean   acc.test.sd
#>    0.98666667    0.02309401
```

We can generate a Learner 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=95.2,sigma=0.0067
```

Then you can proceed as usual. Here we refit and predict the learner on the complete 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?

### 2.8.5 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 -9.9783231  1.0531818     0.2733333  0.02309401         1
    0.071
#> 2 -0.5292817  3.2214785     0.2733333  0.02309401         2
    0.066
#> 3 -0.3544567  4.1644832     0.2733333  0.02309401         3
    0.087
#> 4  0.6341910  7.8640461     0.2866667  0.03055050         4
    0.069
#> 5  5.7640748 -3.3159251     0.9533333  0.03055050         5
    0.067
#> 6 -6.5880397  0.4600323     0.2733333  0.02309401         6
    0.063
```

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:

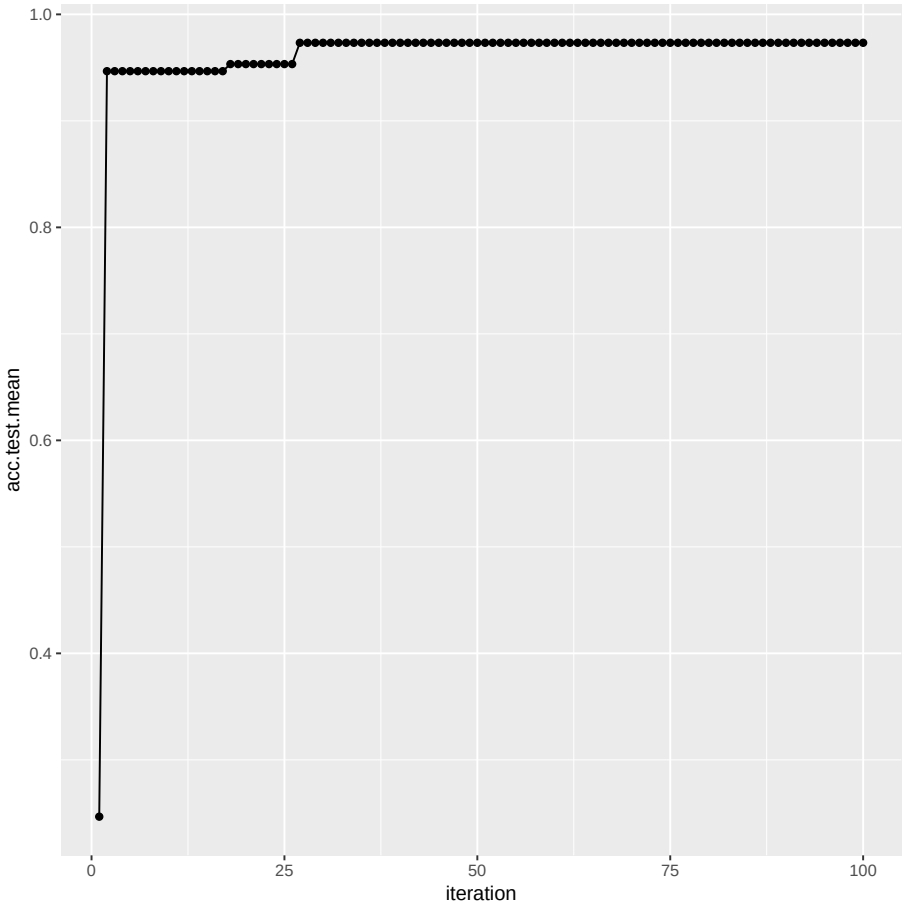
```
generateHyperParsEffectData(res, trafo = TRUE)
#> HyperParsEffectData:
#> Hyperparameters: C,sigma
#> Measures: acc.test.mean,acc.test.sd
#> Optimizer: TuneControlRandom
#> Nested CV Used: FALSE
#> Snapshot of data:
#>              C        sigma acc.test.mean acc.test.sd iteration
    exec.time
#> 1 1.051180e-10 1.130269e+01     0.2733333  0.02309401         1
       0.071
#> 2 2.956095e-01 1.665246e+03     0.2733333  0.02309401         2
       0.066
#> 3 4.421232e-01 1.460438e+04     0.2733333  0.02309401         3
       0.087
#> 4 4.307159e+00 7.312168e+07     0.2866667  0.03055050         4
       0.069
#> 5 5.808644e+05 4.831421e-04     0.9533333  0.03055050         5
       0.067
#> 6 2.582024e-07 2.884246e+00     0.2733333  0.02309401         6
       0.063
```

Note that we can also generate performance on the train data along with the validation/test data, as discussed on the resampling 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  9.457202 -4.0536025          0.98           0.97         1
    0.044
#> 2  9.900523  1.8815923          0.40           1.00         2
    0.035
#> 3  2.363975  5.3202458          0.26           1.00         3
    0.033
#> 4 -1.530251  4.7579424          0.26           0.37         4
    0.033
#> 5 -7.837476  2.4352698          0.26           0.37         5
    0.049
#> 6  8.782931 -0.4143757          0.92           1.00         6
    0.036
```

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.

### 2.8.6 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 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, where we embed the whole model selection process into an outer resampling loop.

## 2.9 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 list of Learners and a list of Tasks. benchmark basically executes resample for each combination of Learner and Task. You can specify an individual resampling strategy for each Task and select one or multiple performance measures to be calculated.

### 2.9.1 Conducting benchmark experiments

We start with a small example. Two learners, linear discriminant analysis (lda) and a classification tree (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 (ResampleDesc), 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 ResampleInstance.

If you would like to use a *fixed test data set* instead of a randomly selected one, you can create a suitable ResampleInstance 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:    0.3000000
#>
```

```
#> Aggregated Result: mmce.test.mean=0.3000000
#>
#> Task: Sonar_example, Learner: classif.rpart
#> Resampling: holdout
#> Measures:             mmce
#> [Resample] iter 1:    0.2857143
#>
#> Aggregated Result: mmce.test.mean=0.2857143
#>

bmr
#>         task.id    learner.id mmce.test.mean
#> 1 Sonar_example   classif.lda      0.3000000
#> 2 Sonar_example classif.rpart      0.2857143
```

For convenience, if you don't want to pass any additional arguments to makeLearner, you don't need to generate the Learners 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:    0.2571429
#>
#> 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
```

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

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

#### 2.9.1.1 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 mclapply for more information on reproducibility and parallel computing.

### 2.9.2 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.

#### 2.9.2.1 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 0.2571429
#>
#> $Sonar_example$classif.rpart
#>   iter      mmce
#> 1    1 0.2714286

getBMRAggrPerformances(bmr)
#> $Sonar_example
#> $Sonar_example$classif.lda
#> mmce.test.mean
#>      0.2571429
```

```
#>
#> $Sonar_example$classif.rpart
#> mmce.test.mean
#>      0.2714286
```

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

By default, nearly all "getter" functions return a nested 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 list.

```
getBMRPerformances(bmr, drop = TRUE)
#> $classif.lda
#>   iter      mmce
#> 1    1 0.2571429
#>
#> $classif.rpart
#>   iter      mmce
#> 1    1 0.2714286
```

Often it is more convenient to work with data.frames. 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 0.2571429
#> 2 Sonar_example classif.rpart    1 0.2714286

getBMRAggrPerformances(bmr, as.df = TRUE)
#>         task.id    learner.id mmce.test.mean
#> 1 Sonar_example   classif.lda      0.2571429
#> 2 Sonar_example classif.rpart      0.2714286
```

#### 2.9.2.2 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 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): 0.01
#>    id truth     prob.M      prob.R response iter  set
#> 1 127     M 0.98673000 0.013270001        M    1 test
#> 2 159     M 0.99659179 0.003408211        M    1 test
#> 3  81     R 0.55436799 0.445632009        M    1 test
#> 4 207     M 0.98660766 0.013392337        M    1 test
#> 5  74     R 0.94120073 0.058799272        M    1 test
#> 6 154     M 0.03862365 0.961376347        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): 0.01
#>    id truth response iter  set
#> 1 127     M        R    1 test
#> 2 159     M        R    1 test
#> 3  81     R        R    1 test
#> 4 207     M        M    1 test
#> 5  74     R        R    1 test
#> 6 154     M        M    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 127     M 0.98673000 0.013270001
           M    1
#> 2 Sonar_example classif.lda 159     M 0.99659179 0.003408211
           M    1
#> 3 Sonar_example classif.lda  81     R 0.55436799 0.445632009
           M    1
#> 4 Sonar_example classif.lda 207     M 0.98660766 0.013392337
           M    1
#> 5 Sonar_example classif.lda  74     R 0.94120073 0.058799272
           M    1
```

```
#> 6 Sonar_example classif.lda 154     M 0.03862365 0.961376347
           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 127     M        R    1 test
#> 2 Sonar_example classif.rpart 159     M        R    1 test
#> 3 Sonar_example classif.rpart  81     R        R    1 test
#> 4 Sonar_example classif.rpart 207     M        M    1 test
#> 5 Sonar_example classif.rpart  74     R        R    1 test
#> 6 Sonar_example classif.rpart 154     M        M    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 make*Task. Moreover, you can conveniently change the ID of a Learner via function setLearnerId.

#### 2.9.2.3 IDs

The IDs of all Learners, Tasks and Measures 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"
```

#### 2.9.2.4 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) list of WrappedModel 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:
```

#### 2.9.2.5 Learners and measures

Moreover, you can extract the employed Learners and Measures.

```
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)
```

### 2.9.3 Merging benchmark results

Sometimes after completing a benchmark experiment it turns out that you want to extend it by another Learner 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 lda and rpart to the sonar.task. We now perform a second experiment using a random forest and quadratic discriminant analysis (qda) and merge the results.

```
### First benchmark result
bmr
#>         task.id    learner.id mmce.test.mean
#> 1 Sonar_example   classif.lda      0.2571429
```

```
#> 2 Sonar_example classif.rpart      0.2714286

### 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      0.1428571
#> 2 Sonar_example          classif.qda      0.2714286

### Merge the results
mergeBenchmarkResults(list(bmr, bmr2))
#>         task.id           learner.id mmce.test.mean
#> 1 Sonar_example          classif.lda      0.2571429
#> 2 Sonar_example        classif.rpart      0.2714286
#> 3 Sonar_example classif.randomForest      0.1428571
#> 4 Sonar_example          classif.qda      0.2714286
```

Note that in the above examples in each case a resample description was passed to the benchmark function. For this reason lda and rpart were most likely evaluated on a different training/test set pair than random forest and 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 Learner later on it's probably easiest to work with ResampleInstances 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 ResampleInstances 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      0.2000000
#> 2 Sonar_example          classif.qda      0.5142857
```

```
### Merge the results
mergeBenchmarkResults(list(bmr, bmr3))
#>         task.id           learner.id mmce.test.mean
#> 1 Sonar_example          classif.lda      0.2571429
#> 2 Sonar_example        classif.rpart      0.2714286
#> 3 Sonar_example classif.randomForest      0.2000000
#> 4 Sonar_example          classif.qda      0.5142857
```

### 2.9.4 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.

#### 2.9.4.1 Example: Comparing lda, rpart and random Forest

We consider linear discriminant analysis (lda), classification trees (rpart), and random forests (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 and converted to Tasks by function convertMLBenchObjToTask.

For all tasks 10-fold cross-validation is chosen as resampling strategy. This is achieved by passing a single resample description 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 list of the same length as the number of tasks that can contain both resample descriptions and resample instances.

We use the mean misclassification error mmce 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               0.0033
#> 2               0.0069
#> 3               0.0531
```

```
#> 4               0.0089
#> 5               0.0374
#> 6               0.3567
#> 7               0.0088
#> 8               0.0134
#> 9               0.4538
#> 10              0.0046
#> 11              0.0098
#> 12              0.4133
#> 13              0.0172
#> 14              0.0167
#> 15              0.2776
```

From the aggregated performance values we can see that for the iris- and PimaIndiansDiabetes-example linear discriminant analysis performs well while for all other tasks the random forest seems superior. Training takes longer for the random forest 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     0.004
#> 2 iris_example        lda    2 0.1333333 0.1666667     0.003
#> 3 iris_example        lda    3 0.0000000 0.0000000     0.003
#> 4 iris_example        lda    4 0.0000000 0.0000000     0.003
#> 5 iris_example        lda    5 0.0000000 0.0000000     0.003
#> 6 iris_example        lda    6 0.0000000 0.0000000     0.003
```

A closer look at the result reveals that the random forest outperforms the classification tree in every instance, while linear discriminant analysis performs better than rpart most of the time. Additionally lda sometimes even beats the random forest. 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.

#### 2.9.4.2 Integrated plots

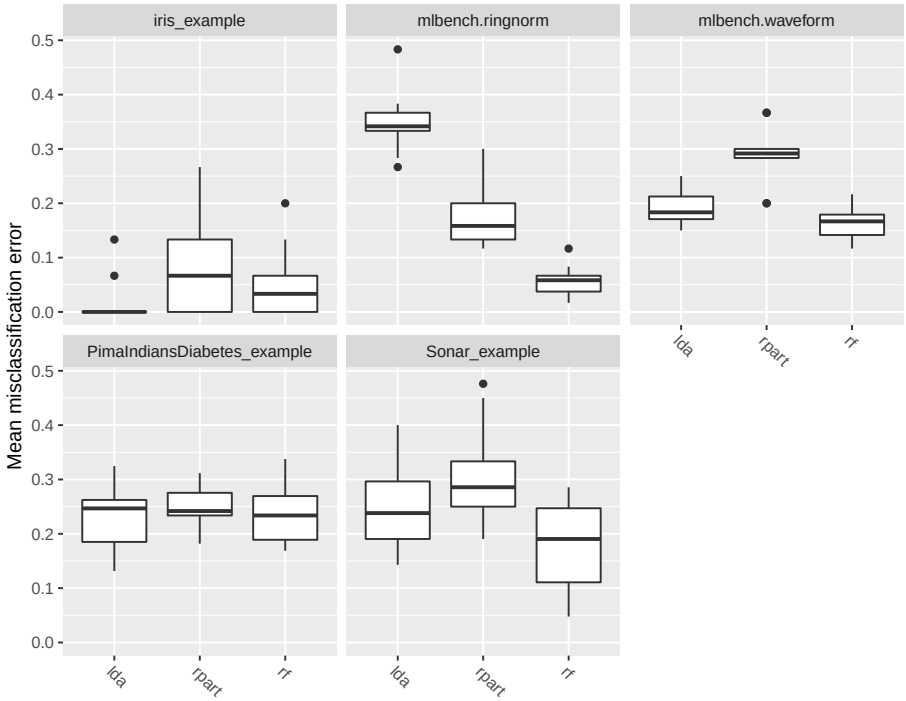
Plots are generated using ggplot2. Further customization, such as renaming plot elements or changing colors, is easily possible.

#### 2.9.4.2.1 Visualizing performances

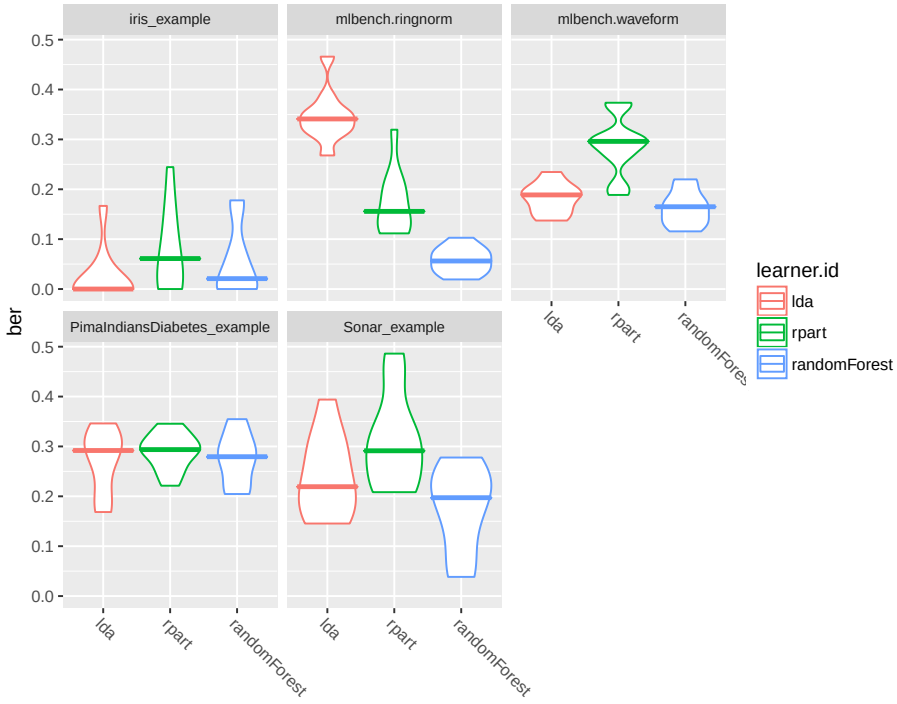
plotBMRBoxplots creates box or violin plots which show the distribution of performance values across resampling iterations for one performance measure and for all learners and tasks (and thus visualize the output of getBMRPerformances).

Below are both variants, box and violin plots. The first plot shows the mmce and the second plot the balanced error rate (ber). 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.

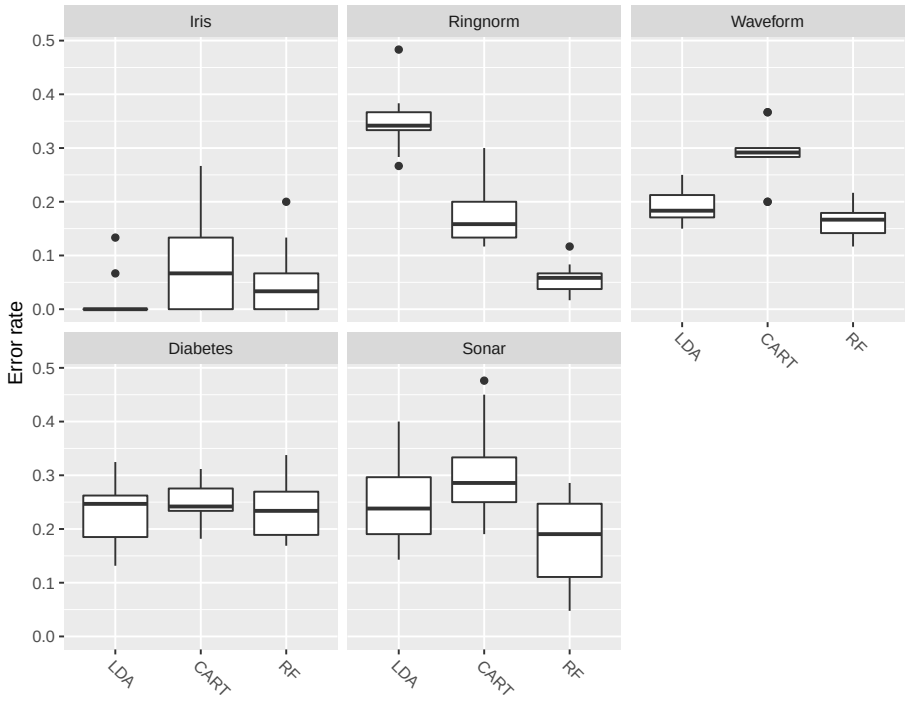
```
mmce$name
#> [1] "Mean misclassification error"

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 functionality like the ylab 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     0.004
#> 2 iris_example        lda    2 0.1333333 0.1666667     0.003
#> 3 iris_example        lda    3 0.0000000 0.0000000     0.003
#> 4 iris_example        lda    4 0.0000000 0.0000000     0.003
#> 5 iris_example        lda    5 0.0000000 0.0000000     0.003
#> 6 iris_example        lda    6 0.0000000 0.0000000     0.003

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

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

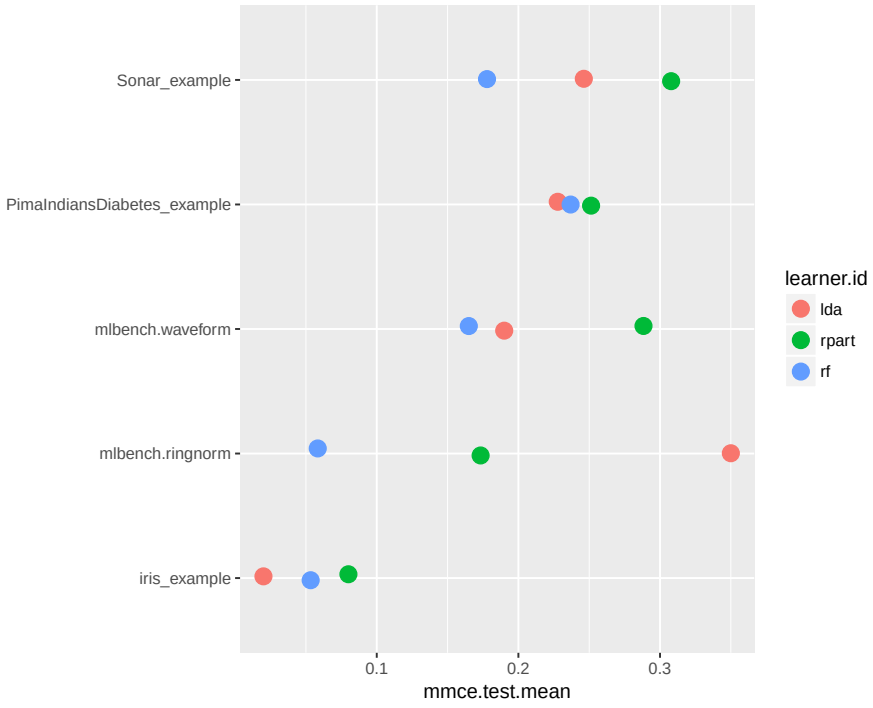



#### 2.9.4.2.2 Visualizing aggregated performances

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 list of Measures passed to benchmark is used, in our example mmce. Moreover, a small vertical jitter is added to prevent overplotting.

```
plotBMRSummary(bmr)
```




#### 2.9.4.2.3 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 (mmce). The rank structure can be visualized by plotBMRRanksAsBarChart.

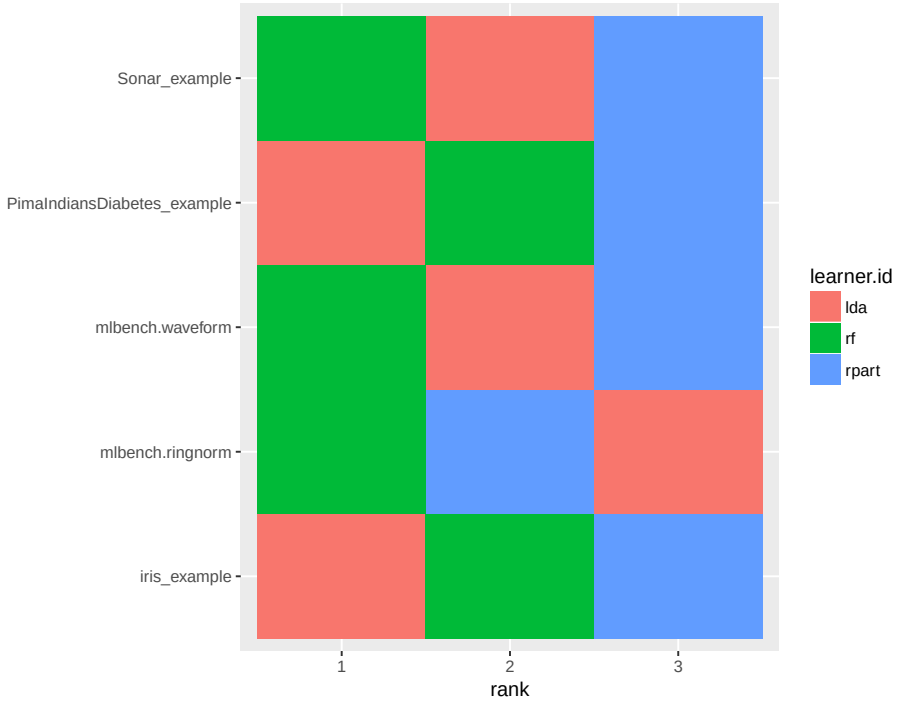
```
m = convertBMRToRankMatrix(bmr, mmce)
m
#>              iris_example mlbench.ringnorm mlbench.waveform
#> lda                     1                3                2
#> rpart                   3                2                3
#> randomForest            2                1                1
#>              PimaIndiansDiabetes_example Sonar_example
```

```
#> lda                    1          2
#> rpart                  3          3
#> randomForest           2          1
```

Methods with best performance, i.e., with lowest mmce, are assigned the lowest rank. Linear discriminant analysis is best for the iris and PimaIndiansDiabetes-examples while the random forest shows best results on the remaining tasks.

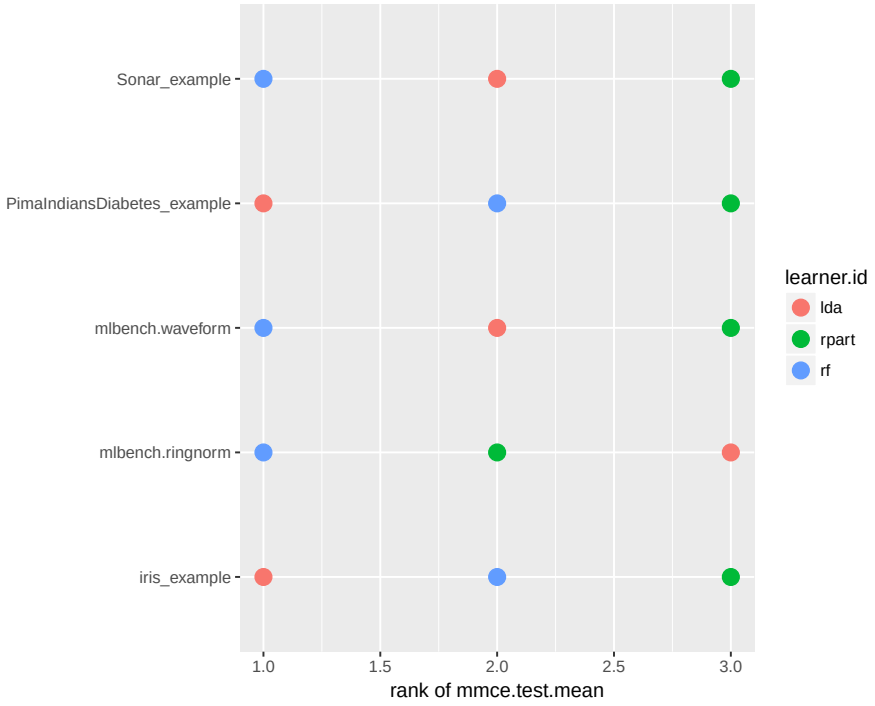
plotBMRRanksAsBarChart with option `pos = "tile"` shows a corresponding heat map. The ranks are displayed on the x-axis and the learners are color-coded.

```
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")
```

#### 2.9.4.3 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 Friedman test based on friedman.test from the stats package is testing the hypothesis whether there is a significant difference between the employed learners, the post hoc Friedman-Nemenyi test 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 Learners 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) and the random Forest.

#### 2.9.4.4 Critical differences diagram

In order to visualize differently performing learners, a critical differences diagram can 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 Learners 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 Learner 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.

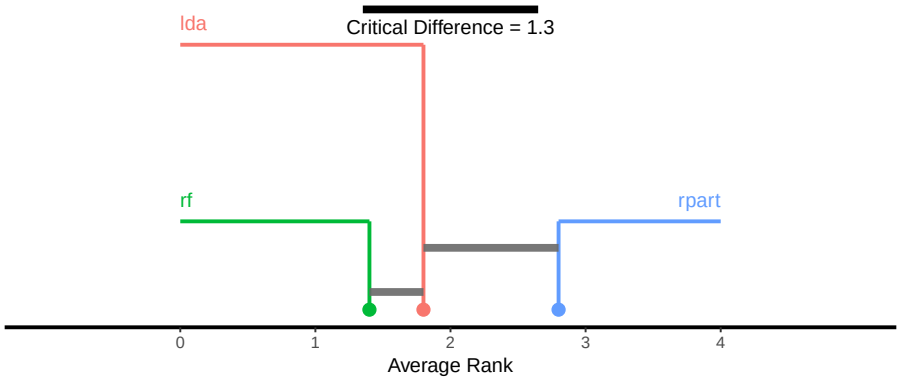
The critical difference $CD$ is calculated by

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

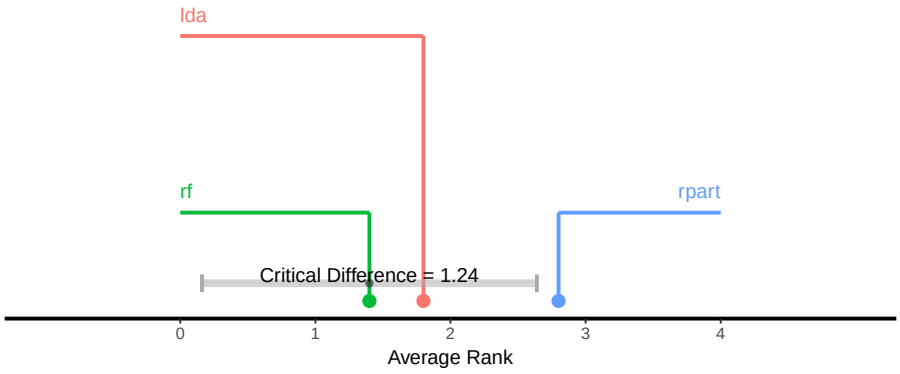
where $N$ denotes the number of tasks, $k$ is the number of learners, and $q_\alpha$ comes from the studentized range statistic divided by $\sqrt{2}$. For details see Demsar (2006).

Function generateCritDifferencesData does all necessary calculations while function plotCritDifferences draws the plot. See the tutorial page about 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))
```

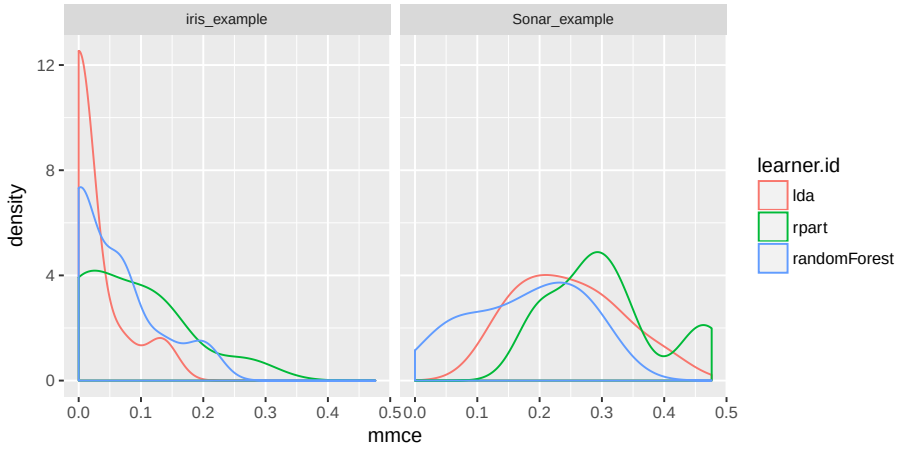



#### 2.9.4.5 Custom plots

You can easily generate your own visualizations by customizing the ggplot objects returned by the plots above, retrieve the data from the ggplot objects and use them as basis for your own plots, or rely on the data.frames returned by getBMRPerformances or getBMRAggrPerformances. Here are some examples.

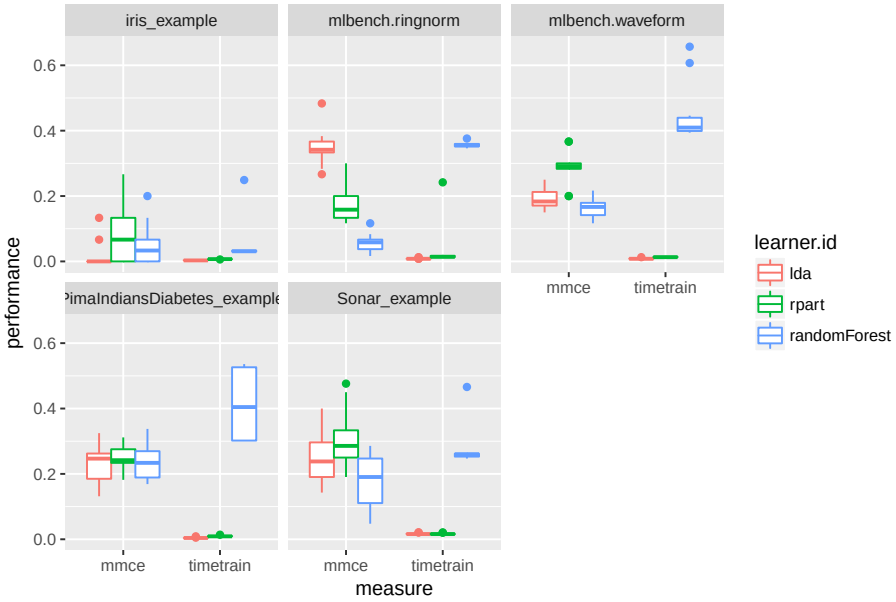
Instead of boxplots (as in plotBMRBoxplots) we could create density plots to show the performance values resulting from individual resampling iterations.

```
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))
```



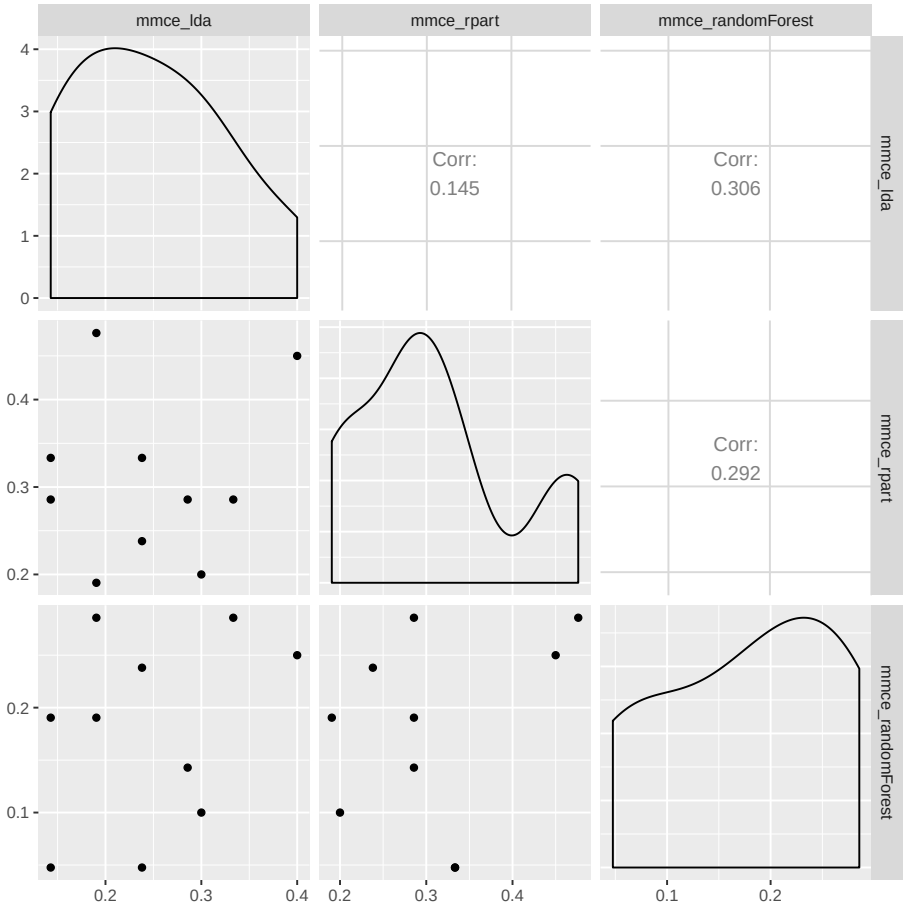

In order to plot multiple performance measures in parallel, `perf` is reshaped to long format. Below we generate grouped boxplots showing the error rate (mmce) and the training time timetrain.

```
### 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 ggpairs from package GGally is used to generate a scatterplot matrix of mean misclassification errors (mmce) on the 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)
```




### 2.9.5 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 ROC curves and functions generateThreshVsPerfData, plotROCCurves, and plotViperCharts as well as the page about 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 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, dimen-

sionality 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 wrapped learners 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 parallelization.

## 2.10 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* functions. The first loop mlr encounters which is marked as parallel executable will be automatically parallelized. It is good practice to call 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 instead.

### 2.10.1 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 of each learner instead, you can specifically pass the `level "mlr.resample"` to the 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.

### 2.10.2 Custom learners and parallelization

If you have implemented a custom learner yourself, 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 parallelStart.

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

### 2.10.3 The end

For further details, consult the parallelMap tutorial and help.

## 2.11 Visualization

### 2.11.1 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 or 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 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 plotting functions have an additional suffix `GGVIS`, i.e., `plotFunctionPurposeGGVIS`.

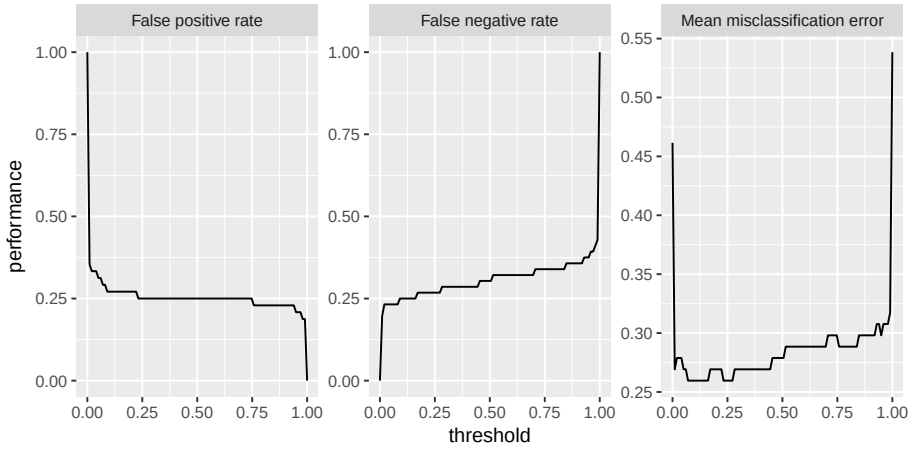
#### 2.11.1.1 Some examples

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`.

#### 2.11.1.2 Customizing plots

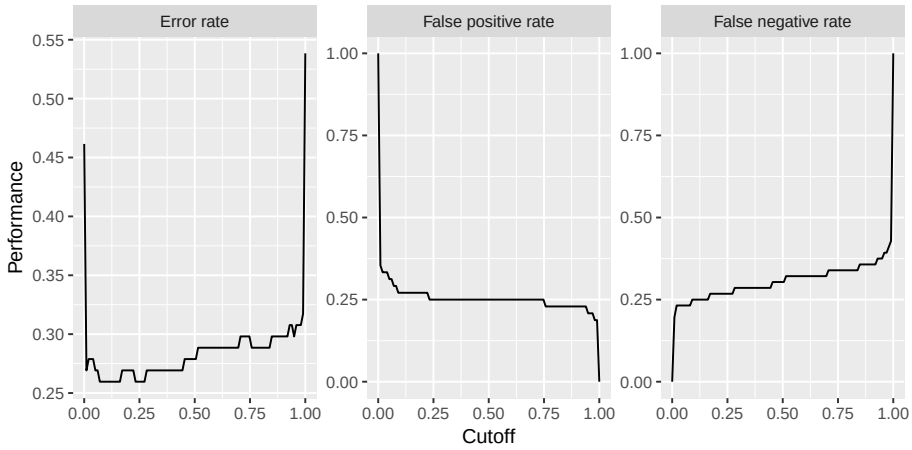
As mentioned above it is easily possible to customize the built-in plots or making your own visualizations from scratch based on the generated data.

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

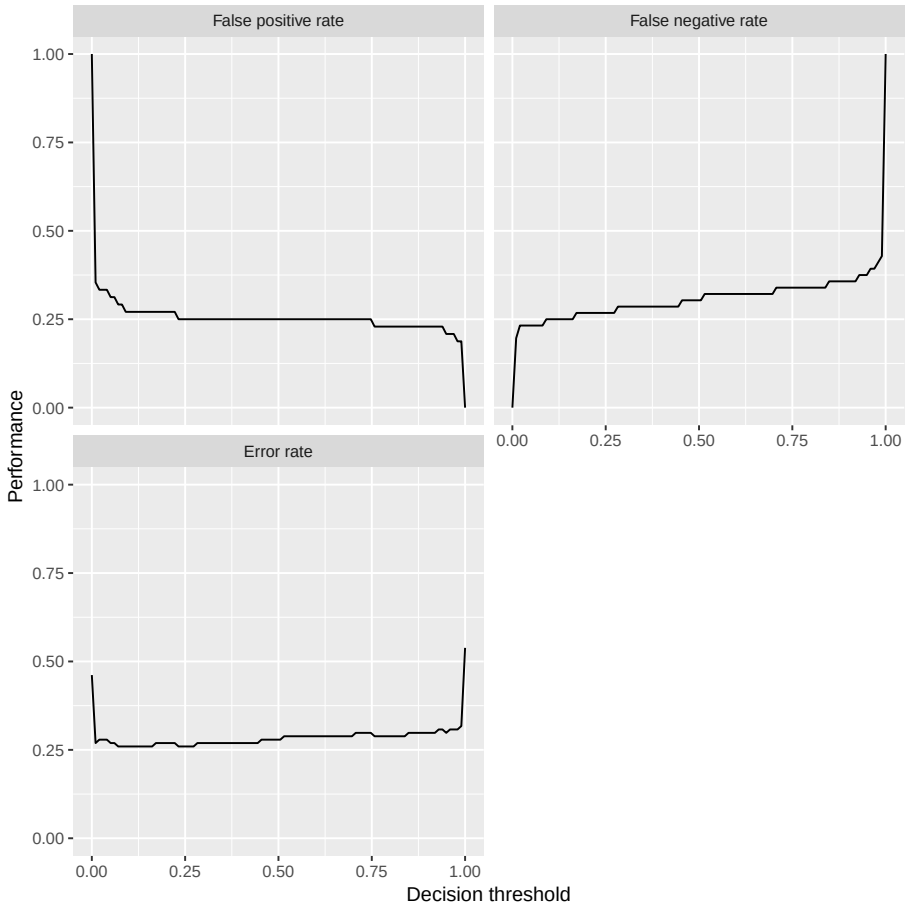
```
plt = plotThreshVsPerf(d, pretty.names = FALSE)

### 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 labeller function requires calling facet_wrap (or 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.

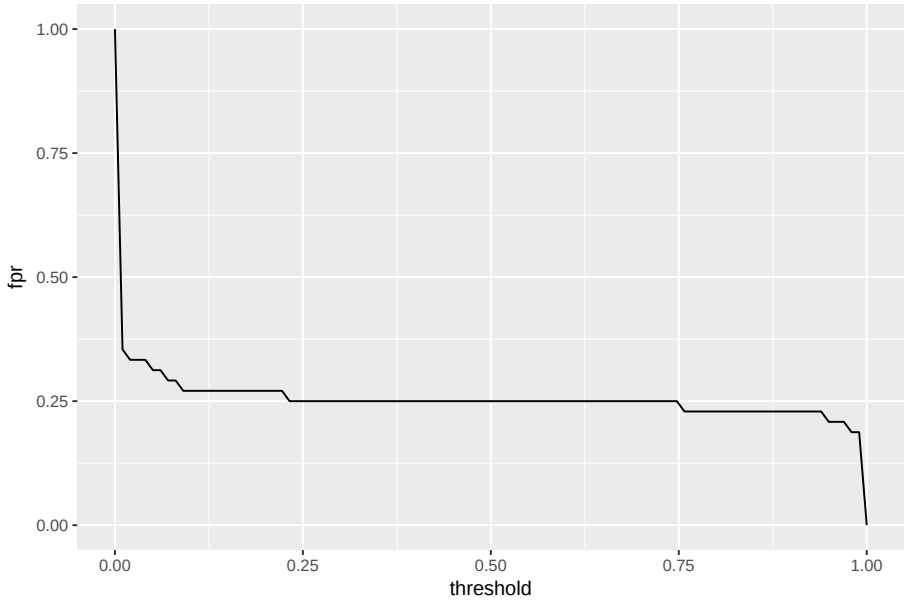
```
plt = plotThreshVsPerf(d, pretty.names = FALSE)

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
```



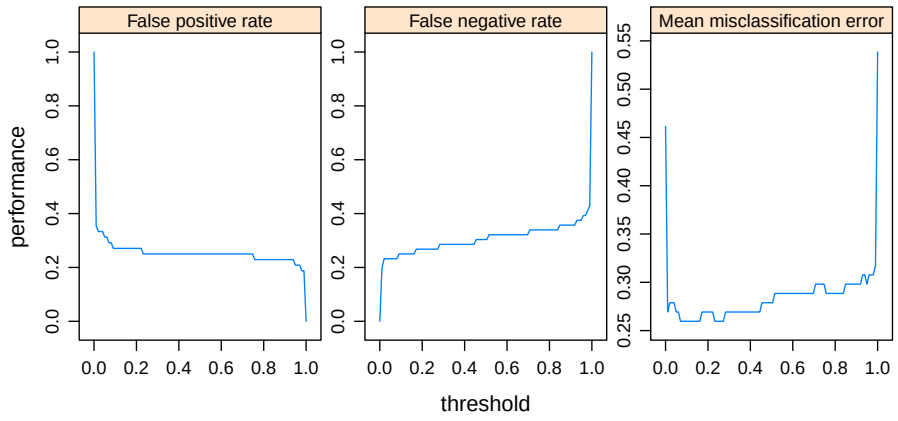

Instead of using the built-in function plotThreshVsPerf we could also manually create the plot based on the output of generateThreshVsPerfData: in this case to plot only one measure.

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



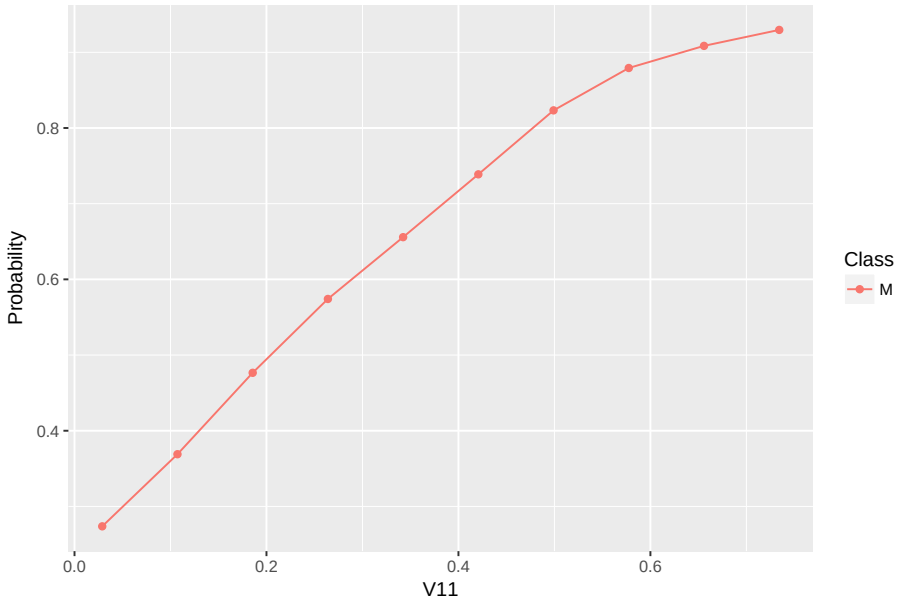

The decoupling of generation and plotting functions is especially practical if you prefer traditional graphics or lattice. Here is a lattice plot which gives a result similar to that of plotThreshVsPerf.

```
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)))
```

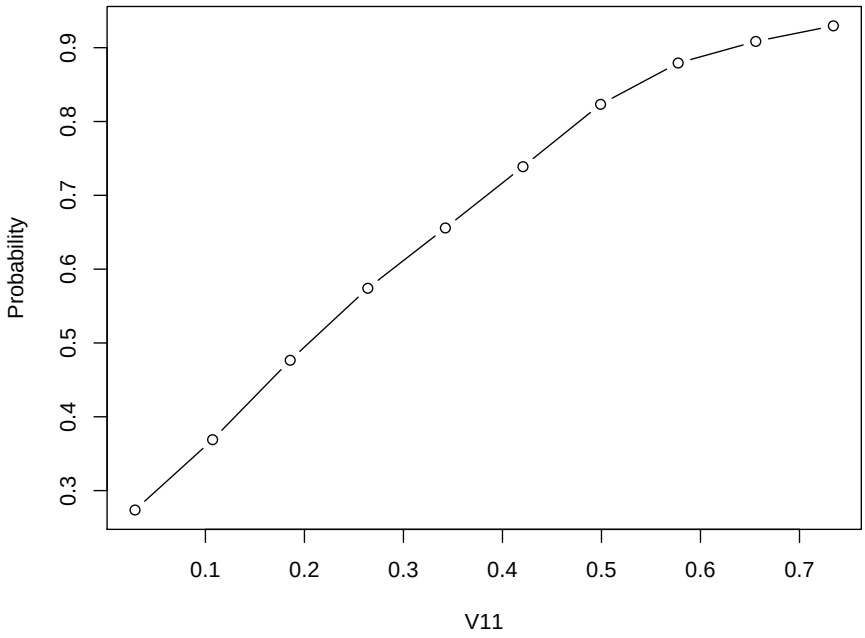


Let's conclude with a brief look on a second example. Here we use plotPartialDependence but extract the data from the ggplot object `plt`and use it to create a traditional graphics::plot, additional to the ggplot2 plot.

```
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])
```




### 2.11.2 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 functions are experimental and are subject to change, though they should work. Most generate interactive shiny applications, that automatically start and run locally.

Performance evaluation and ROC analysis:

- generateThreshVsPerfData
- plotThresVsPerf, plotThreshVsPerfGGVIS
- plotROCCurves

Benchmark experiments:

- generateCritDifferencesData
- plotCritDifferences

Tuning and hyperparameter tuning effects:

- generateHyperParsEffectData
- plotHyperParsEffect

Feature selection:

- generateFilterValuesData
- plotFilterValues
- plotFilterValuesGGVIS

Learning curves:

- generatePartialDependenceData, generateFunctionalANOVAData
- plotPartialDependence
- plotPartialDependenceGGVIS

Classifier calibration:

- generateCalibrationData
- plotCalibration

# 3 Advanced

## 3.1 Configuring mlr

mlr is designed to make usage errors due to typos or invalid parameter values as unlikely as possible. Occasionally, you might want to break those barriers and get full access, for example to reduce the amount of output on the console or to turn off checks. For all available options simply refer to the documentation of configureMlr. In the following we show some common use cases.

Generally, function configureMlr permits to set options globally for your current **R** session.

It is also possible to set options locally.

- All options referring to the behavior of learners (these are all options except `show.info`) can be set for an individual learner via the `config` argument of makeLearner. The local precedes the global configuration.
- Some functions like resample, benchmark, selectFeatures, tuneParams, and tuneParamsMultiCrit have a `show.info` flag that controls if progress messages are shown. The default value of `show.info` can be set by configureMlr.

### 3.1.1 Example: Reducing the output on the console

You are bothered by all the output on the console like in this example?

```
rdesc = makeResampleDesc("Holdout")
r = resample("classif.multinom", iris.task, rdesc)
#> Resampling: holdout
#> Measures:             mmce
#> # weights:  18 (10 variable)
#> initial  value 109.861229
#> iter  10 value 12.256619
#> iter  20 value 3.638740
#> iter  30 value 3.228628
#> iter  40 value 2.951100
#> iter  50 value 2.806521
#> iter  60 value 2.739076
#> iter  70 value 2.522206
#> iter  80 value 2.485225
#> iter  90 value 2.381397
#> iter 100 value 2.360602
#> final  value 2.360602
#> stopped after 100 iterations
#> [Resample] iter 1:    0.0200000
#>
#> Aggregated Result: mmce.test.mean=0.0200000
#>
```

You can suppress the output for this Learner and this resample call as follows:

```
lrn = makeLearner("classif.multinom", config =
    list(show.learner.output = FALSE))
r = resample(lrn, iris.task, rdesc, show.info = FALSE)
```

(Note that multinom has a `trace` switch that can alternatively be used to turn off the progress messages.)

To globally suppress the output for all subsequent learners and calls to resample, benchmark etc. do the following:

```
configureMlr(show.learner.output = FALSE, show.info = FALSE)
r = resample("classif.multinom", iris.task, rdesc)
```

### 3.1.2 Accessing and resetting the configuration

Function getMlrOptions returns a list with the current configuration.

```
getMlrOptions()
#> $show.info
#> [1] FALSE
#>
#> $on.learner.error
#> [1] "stop"
#>
#> $on.learner.warning
#> [1] "warn"
#>
#> $on.par.without.desc
#> [1] "stop"
#>
#> $on.par.out.of.bounds
#> [1] "stop"
#>
#> $on.measure.not.applicable
#> [1] "stop"
#>
#> $show.learner.output
#> [1] FALSE
#>
#> $on.error.dump
#> [1] FALSE
```

To restore the default configuration call configureMlr with an empty argument list.

```
configureMlr()
```

```
getMlrOptions()
#> $show.info
#> [1] TRUE
#>
#> $on.learner.error
#> [1] "stop"
#>
#> $on.learner.warning
#> [1] "warn"
```

```
#>
#> $on.par.without.desc
#> [1] "stop"
#>
#> $on.par.out.of.bounds
#> [1] "stop"
#>
#> $on.measure.not.applicable
#> [1] "stop"
#>
#> $show.learner.output
#> [1] TRUE
#>
#> $on.error.dump
#> [1] FALSE
```

### 3.1.3 Example: Turning off parameter checking

It might happen that you want to set a parameter of a Learner, but the parameter is not registered in the learner's parameter set yet. In this case you might want to contact us or open an issue as well! But until the problem is fixed you can turn off mlr's parameter checking. The parameter setting will then be passed to the underlying function without further ado.

```
### Support Vector Machine with linear kernel and new parameter
    'newParam'
lrn = makeLearner("classif.ksvm", kernel = "vanilladot", newParam =
    3)
#> Error in setHyperPars2.Learner(learner, insert(par.vals, args)):
    classif.ksvm: Setting parameter newParam without available
    description object!
#> Did you mean one of these hyperparameters instead: degree scaled
    kernel
#> You can switch off this check by using configureMlr!

### Turn off parameter checking completely
configureMlr(on.par.without.desc = "quiet")
lrn = makeLearner("classif.ksvm", kernel = "vanilladot", newParam =
    3)
train(lrn, iris.task)
#>  Setting default kernel parameters
#> Model for learner.id=classif.ksvm; learner.class=classif.ksvm
#> Trained on: task.id = iris_example; obs = 150; features = 4
#> Hyperparameters: fit=FALSE,kernel=vanilladot,newParam=3
```

```
### Option "quiet" also masks typos
lrn = makeLearner("classif.ksvm", kernl = "vanilladot")
train(lrn, iris.task)
#> Model for learner.id=classif.ksvm; learner.class=classif.ksvm
#> Trained on: task.id = iris_example; obs = 150; features = 4
#> Hyperparameters: fit=FALSE,kernl=vanilladot

### Alternatively turn off parameter checking, but still see warnings
configureMlr(on.par.without.desc = "warn")
lrn = makeLearner("classif.ksvm", kernl = "vanilladot", newParam = 3)
#> Warning in setHyperPars2.Learner(learner, insert(par.vals,
    args)): classif.ksvm: Setting parameter kernl without available
    description object!
#> Did you mean one of these hyperparameters instead: kernel nu
    degree
#> You can switch off this check by using configureMlr!
#> Warning in setHyperPars2.Learner(learner, insert(par.vals,
    args)): classif.ksvm: Setting parameter newParam without
    available description object!
#> Did you mean one of these hyperparameters instead: degree scaled
    kernel
#> You can switch off this check by using configureMlr!

train(lrn, iris.task)
#> Model for learner.id=classif.ksvm; learner.class=classif.ksvm
#> Trained on: task.id = iris_example; obs = 150; features = 4
#> Hyperparameters: fit=FALSE,kernl=vanilladot,newParam=3
```

### 3.1.4 Example: Handling errors in a learning method

If a learning method throws an error the default behavior of mlr is to generate an exception as well. However, in some situations, for example if you conduct a larger benchmark study with multiple data sets and learners, you usually don't want the whole experiment stopped due to one error. You can prevent this using the `on.learner.error` option of configureMlr.

```
### This call gives an error caused by the low number of
    observations in class "virginica"
train("classif.qda", task = iris.task, subset = 1:104)
#> Error in qda.default(x, grouping, ...): some group is too small
    for 'qda'

### Get a warning instead of an error
configureMlr(on.learner.error = "warn")
mod = train("classif.qda", task = iris.task, subset = 1:104)
```

```
#> Warning in train("classif.qda", task = iris.task, subset =
    1:104): Could not train learner classif.qda: Error in
    qda.default(x, grouping, ...) :
#>   some group is too small for 'qda'

mod
#> Model for learner.id=classif.qda; learner.class=classif.qda
#> Trained on: task.id = iris_example; obs = 104; features = 4
#> Hyperparameters:
#> Training failed: Error in qda.default(x, grouping, ...) :
#>   some group is too small for 'qda'
#>
#> Training failed: Error in qda.default(x, grouping, ...) :
#>   some group is too small for 'qda'

### mod is an object of class FailureModel
isFailureModel(mod)
#> [1] TRUE

### Retrieve the error message
getFailureModelMsg(mod)
#> [1] "Error in qda.default(x, grouping, ...) : \n  some group is
    too small for 'qda'\n"

### predict and performance return NA's
pred = predict(mod, iris.task)
pred
#> Prediction: 150 observations
#> predict.type: response
#> threshold:
#> time: NA
#>   id  truth response
#> 1  1 setosa     <NA>
#> 2  2 setosa     <NA>
#> 3  3 setosa     <NA>
#> 4  4 setosa     <NA>
#> 5  5 setosa     <NA>
#> 6  6 setosa     <NA>
#> ... (#rows: 150, #cols: 3)

performance(pred)
#> mmce
#>   NA
```

If `on.learner.error = "warn"` a warning is issued instead of an exception and an object of class FailureModel is created. You can extract the error message

using function getFailureModelMsg. All further steps like prediction and performance calculation work and return `NA's`.

## 3.2 Wrapper

Wrappers can be employed to extend integrated learners with new functionality. The broad scope of operations and methods which are implemented as wrappers underline the flexibility of the wrapping approach:

- Data preprocessing
- Imputation
- Bagging
- Tuning
- Feature selection
- Cost-sensitive classification
- Over- and undersampling for imbalanced classification problems
- Multiclass extension for binary-class learners
- Multilabel classification

All these operations and methods have a few things in common: First, they all wrap around mlr learners and they return a new learner. Therefore learners can be wrapped multiple times. Second, they are implemented using a *train* (pre-model hook) and *predict* (post-model hook) method.

### 3.2.1 Example: Bagging wrapper

In this section we exemplary describe the bagging wrapper to create a random forest which supports weights. To achieve that we combine several decision trees from the rpart package to create our own custom random forest.

First, we create a weighted toy task.

```
data(iris)
task = makeClassifTask(data = iris, target = "Species", weights =
    as.integer(iris$Species))
```

Next, we use makeBaggingWrapper to create the base learners and the bagged learner. We choose to set equivalents of `ntree` (100 base learners) and `mtry` (proportion of randomly selected features).

```
base.lrn = makeLearner("classif.rpart")
wrapped.lrn = makeBaggingWrapper(base.lrn, bw.iters = 100, bw.feats
    = 0.5)
print(wrapped.lrn)
#> Learner classif.rpart.bagged from package rpart
```

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

As we can see in the output, the wrapped learner inherited all properties from the base learner, especially the "weights" attribute is still present. We can use this newly constructed learner like all base learners, i.e. we can use it in train, benchmark, resample, etc.

```
benchmark(tasks = task, learners = list(base.lrn, wrapped.lrn))
#> Task: iris, Learner: classif.rpart
#> Resampling: cross-validation
#> Measures:             mmce
#> [Resample] iter 1:    0.2000000
#> [Resample] iter 2:    0.0666667
#> [Resample] iter 3:    0.0000000
#> [Resample] iter 4:    0.0666667
#> [Resample] iter 5:    0.0000000
#> [Resample] iter 6:    0.0666667
#> [Resample] iter 7:    0.0000000
#> [Resample] iter 8:    0.1333333
#> [Resample] iter 9:    0.0666667
#> [Resample] iter 10:   0.0666667
#>
#> Aggregated Result: mmce.test.mean=0.0666667
#>
#> Task: iris, Learner: classif.rpart.bagged
#> Resampling: cross-validation
#> Measures:             mmce
#> [Resample] iter 1:    0.2000000
#> [Resample] iter 2:    0.0666667
#> [Resample] iter 3:    0.0000000
#> [Resample] iter 4:    0.0666667
#> [Resample] iter 5:    0.0000000
#> [Resample] iter 6:    0.0666667
#> [Resample] iter 7:    0.0000000
#> [Resample] iter 8:    0.1333333
#> [Resample] iter 9:    0.0000000
#> [Resample] iter 10:   0.0666667
#>
#> Aggregated Result: mmce.test.mean=0.0600000
#>
```

```
#>   task.id           learner.id mmce.test.mean
#> 1    iris        classif.rpart     0.06666667
#> 2    iris classif.rpart.bagged     0.06000000
```

That far we are quite happy with our new learner. But we hope for a better performance by tuning some hyperparameters of both the decision trees and bagging wrapper. Let's have a look at the available hyperparameters of the fused learner:

```
getParamSet(wrapped.lrn)
#>                     Type len   Def   Constr Req Tunable Trafo
#> bw.iters         integer   -    10 1 to Inf   -    TRUE     -
#> bw.replace       logical   -  TRUE        -   -    TRUE     -
#> bw.size          numeric   -     -   0 to 1   -    TRUE     -
#> bw.feats         numeric   - 0.667   0 to 1   -    TRUE     -
#> minsplit         integer   -    20 1 to Inf   -    TRUE     -
#> minbucket        integer   -     - 1 to Inf   -    TRUE     -
#> cp               numeric   -  0.01   0 to 1   -    TRUE     -
#> maxcompete       integer   -     4 0 to Inf   -    TRUE     -
#> maxsurrogate     integer   -     5 0 to Inf   -    TRUE     -
#> usesurrogate    discrete   -     2    0,1,2   -    TRUE     -
#> surrogatestyle  discrete   -     0      0,1   -    TRUE     -
#> maxdepth         integer   -    30  1 to 30   -    TRUE     -
#> xval             integer   -    10 0 to Inf   -   FALSE     -
#> parms            untyped   -     -        -   -    TRUE     -
```

We choose to tune the parameters `minsplit` and `bw.feats` for the mmce using a random search in a 3-fold CV:

```
ctrl = makeTuneControlRandom(maxit = 10)
rdesc = makeResampleDesc("CV", iters = 3)
par.set = makeParamSet(
  makeIntegerParam("minsplit", lower = 1, upper = 10),
  makeNumericParam("bw.feats", lower = 0.25, upper = 1)
)
tuned.lrn = makeTuneWrapper(wrapped.lrn, rdesc, mmce, par.set, ctrl)
print(tuned.lrn)
#> Learner classif.rpart.bagged.tuned from package rpart
#> Type: classif
#> Name: ; Short name:
#> Class: TuneWrapper
#> Properties:
    numerics,factors,ordered,missings,weights,prob,twoclass,multiclass,featimp
#> Predict-Type: response
#> Hyperparameters: xval=0,bw.iters=100,bw.feats=0.5
```

Calling the train method of the newly constructed learner performs the following steps:

1. The tuning wrapper sets parameters for the underlying model in slot `$next.learner` and calls its train method.
2. Next learner is the bagging wrapper. The passed down argument `bw.feats` is used in the bagging wrapper training function, the argument `minsplit` gets passed down to `$next.learner`. The base wrapper function calls the base learner `bw.iters` times and stores the resulting models.
3. The bagged models are evaluated using the mean mmce (default aggregation for this performance measure) and new parameters are selected using the tuning method.
4. This is repeated until the tuner terminates. Output is a tuned bagged learner.

```
lrn = train(tuned.lrn, task = task)
#> [Tune] Started tuning learner classif.rpart.bagged for parameter
    set:
#>             Type len Def    Constr Req Tunable Trafo
#> minsplit integer   -   -   1 to 10   -    TRUE     -
#> bw.feats numeric   -   - 0.25 to 1   -    TRUE     -
#> With control class: TuneControlRandom
#> Imputation value: 1
#> [Tune-x] 1: minsplit=5; bw.feats=0.935
#> [Tune-y] 1: mmce.test.mean=0.0466667; time: 0.1 min
#> [Tune-x] 2: minsplit=9; bw.feats=0.675
#> [Tune-y] 2: mmce.test.mean=0.0466667; time: 0.1 min
#> [Tune-x] 3: minsplit=2; bw.feats=0.847
#> [Tune-y] 3: mmce.test.mean=0.0466667; time: 0.1 min
#> [Tune-x] 4: minsplit=4; bw.feats=0.761
#> [Tune-y] 4: mmce.test.mean=0.0466667; time: 0.1 min
#> [Tune-x] 5: minsplit=6; bw.feats=0.338
#> [Tune-y] 5: mmce.test.mean=0.0866667; time: 0.1 min
#> [Tune-x] 6: minsplit=1; bw.feats=0.637
#> [Tune-y] 6: mmce.test.mean=0.0466667; time: 0.1 min
#> [Tune-x] 7: minsplit=1; bw.feats=0.998
#> [Tune-y] 7: mmce.test.mean=0.0466667; time: 0.1 min
#> [Tune-x] 8: minsplit=4; bw.feats=0.698
#> [Tune-y] 8: mmce.test.mean=0.0466667; time: 0.1 min
#> [Tune-x] 9: minsplit=3; bw.feats=0.836
#> [Tune-y] 9: mmce.test.mean=0.0466667; time: 0.1 min
#> [Tune-x] 10: minsplit=10; bw.feats=0.529
#> [Tune-y] 10: mmce.test.mean=0.0533333; time: 0.1 min
#> [Tune] Result: minsplit=1; bw.feats=0.998 :
    mmce.test.mean=0.0466667
print(lrn)
#> Model for learner.id=classif.rpart.bagged.tuned;
    learner.class=TuneWrapper
```

```
#> Trained on: task.id = iris; obs = 150; features = 4
#> Hyperparameters: xval=0,bw.iters=100,bw.feats=0.5
```

## 3.3 Imputation of Missing Values

mlr provides several imputation methods which are listed on the help page imputations. These include standard techniques as imputation by a constant value (like a fixed constant, the mean, median or mode) and random numbers (either from the empirical distribution of the feature under consideration or a certain distribution family). Moreover, missing values in one feature can be replaced based on the other features by predictions from any supervised Learner integrated into mlr.

If your favourite option is not implemented in mlr yet, you can easily create your own imputation method.

Also note that some of the learning algorithms included in mlr can deal with missing values in a sensible way, i.e., other than simply deleting observations with missing values. Those Learners have the property `"missings"` and thus can be identified using listLearners.

```
### Regression learners that can deal with missing values
listLearners("regr", properties = "missings")[c("class", "package")]
#>              class      package
#> 1 regr.bartMachine  bartMachine
#> 2  regr.blackboost mboost,party
#> 3     regr.cforest        party
#> 4       regr.ctree        party
#> 5      regr.cubist       Cubist
#> 6 regr.featureless          mlr
#> ... (#rows: 14, #cols: 2)
```

See also the list of integrated learners in the Appendix.

### 3.3.1 Imputation and reimputation

Imputation can be done by function impute. You can specify an imputation method for each feature individually or for classes of features like numerics or factors. Moreover, you can generate dummy variables that indicate which values are missing, also either for classes of features or for individual features. These allow to identify the patterns and reasons for missing data and permit to treat imputed and observed values differently in a subsequent analysis.

Let's have a look at the airquality data set.

```
data(airquality)
summary(airquality)
#>      Ozone           Solar.R          Wind             Temp
#>  Min.   :  1.00   Min.   :  7.0   Min.   : 1.700   Min.   :56.00
#>  1st Qu.: 18.00   1st Qu.:115.8   1st Qu.: 7.400   1st Qu.:72.00
#>  Median : 31.50   Median :205.0   Median : 9.700   Median :79.00
#>  Mean   : 42.13   Mean   :185.9   Mean   : 9.958   Mean   :77.88
#>  3rd Qu.: 63.25   3rd Qu.:258.8   3rd Qu.:11.500   3rd Qu.:85.00
#>  Max.   :168.00   Max.   :334.0   Max.   :20.700   Max.   :97.00
#>  NA's   :37       NA's   :7
#>      Month            Day
#>  Min.   :5.000   Min.   : 1.0
#>  1st Qu.:6.000   1st Qu.: 8.0
#>  Median :7.000   Median :16.0
#>  Mean   :6.993   Mean   :15.8
#>  3rd Qu.:8.000   3rd Qu.:23.0
#>  Max.   :9.000   Max.   :31.0
#>
```

There are 37 `NA's` in variable `Ozone` (ozone pollution) and 7 `NA's` in variable `Solar.R` (solar radiation). For demonstration purposes we insert artificial `NA's` in column `Wind` (wind speed) and coerce it into a factor.

```
airq = airquality
ind = sample(nrow(airq), 10)
airq$Wind[ind] = NA
airq$Wind = cut(airq$Wind, c(0,8,16,24))
summary(airq)
#>      Ozone           Solar.R          Wind         Temp
#>  Min.   :  1.00   Min.   :  7.0   (0,8]  :51   Min.   :56.00
#>  1st Qu.: 18.00   1st Qu.:115.8   (8,16] :86   1st Qu.:72.00
#>  Median : 31.50   Median :205.0   (16,24]: 6   Median :79.00
#>  Mean   : 42.13   Mean   :185.9   NA's   :10   Mean   :77.88
#>  3rd Qu.: 63.25   3rd Qu.:258.8                3rd Qu.:85.00
#>  Max.   :168.00   Max.   :334.0                Max.   :97.00
#>  NA's   :37       NA's   :7
#>      Month            Day
#>  Min.   :5.000   Min.   : 1.0
#>  1st Qu.:6.000   1st Qu.: 8.0
#>  Median :7.000   Median :16.0
#>  Mean   :6.993   Mean   :15.8
#>  3rd Qu.:8.000   3rd Qu.:23.0
#>  Max.   :9.000   Max.   :31.0
#>
```

If you want to impute `NA's` in all integer features (these include `Ozone` and

Solar.R) by the mean, in all factor features (Wind) by the mode and additionally generate dummy variables for all integer features, you can do this as follows:

```
imp = impute(airq, classes = list(integer = imputeMean(), factor =
    imputeMode()),
  dummy.classes = "integer")
```

impute returns a list where slot $data contains the imputed data set. Per default, the dummy variables are factors with levels "TRUE" and "FALSE". It is also possible to create numeric zero-one indicator variables.

```
head(imp$data, 10)
#>       Ozone  Solar.R    Wind Temp Month Day Ozone.dummy
    Solar.R.dummy
#> 1  41.00000 190.0000   (0,8]   67     5   1       FALSE
    FALSE
#> 2  36.00000 118.0000   (0,8]   72     5   2       FALSE
    FALSE
#> 3  12.00000 149.0000  (8,16]   74     5   3       FALSE
    FALSE
#> 4  18.00000 313.0000  (8,16]   62     5   4       FALSE
    FALSE
#> 5  42.12931 185.9315  (8,16]   56     5   5        TRUE
    TRUE
#> 6  28.00000 185.9315  (8,16]   66     5   6       FALSE
    TRUE
#> 7  23.00000 299.0000  (8,16]   65     5   7       FALSE
    FALSE
#> 8  19.00000  99.0000  (8,16]   59     5   8       FALSE
    FALSE
#> 9   8.00000  19.0000 (16,24]   61     5   9       FALSE
    FALSE
#> 10 42.12931 194.0000  (8,16]   69     5  10        TRUE
    FALSE
```

Slot $desc is an ImputationDesc object that stores all relevant information about the imputation. For the current example this includes the means and the mode computed on the non-missing data.

```
imp$desc
#> Imputation description
#> Target:
#> Features: 6; Imputed: 6
#> impute.new.levels: TRUE
#> recode.factor.levels: TRUE
#> dummy.type: factor
```

The imputation description shows the name of the target variable (not present), the number of features and the number of imputed features. Note that the latter number refers to the features for which an imputation method was specified (five integers plus one factor) and not to the features actually containing `NA's`. `dummy.type` indicates that the dummy variables are factors. For details on `impute.new.levels` and `recode.factor.levels` see the help page of function impute.

Let's have a look at another example involving a target variable. A possible learning task associated with the airquality data is to predict the ozone pollution based on the meteorological features. Since we do not want to use columns `Day` and `Month` we remove them.

```
airq = subset(airq, select = 1:4)
```

The first 100 observations are used as training data set.

```
airq.train = airq[1:100,]
airq.test = airq[-c(1:100),]
```

In case of a supervised learning problem you need to pass the name of the target variable to impute. This prevents imputation and creation of a dummy variable for the target variable itself and makes sure that the target variable is not used to impute the features.

In contrast to the example above we specify imputation methods for individual features instead of classes of features.

Missing values in `Solar.R` are imputed by random numbers drawn from the empirical distribution of the non-missing observations.

Function imputeLearner allows to use all supervised learning algorithms integrated into mlr for imputation. The type of the Learner (`regr`, `classif`) must correspond to the class of the feature to be imputed. The missing values in `Wind` are replaced by the predictions of a classification tree (rpart). Per default, all available columns in `airq.train` except the target variable (`Ozone`) and the variable to be imputed (`Wind`) are used as features in the classification tree, here `Solar.R` and `Temp`. You can also select manually which columns to use. Note that rpart can deal with missing feature values, therefore the `NA's` in column `Solar.R` do not pose a problem.

```
imp = impute(airq.train, target = "Ozone", cols = list(Solar.R =
    imputeHist(),
  Wind = imputeLearner("classif.rpart")), dummy.cols = c("Solar.R",
      "Wind"))
summary(imp$data)
#>      Ozone           Solar.R           Wind         Temp
#>  Min.   :  1.00   Min.   :  7.00   (0,8]  :34   Min.   :56.00
#>  1st Qu.: 16.00   1st Qu.: 98.75   (8,16] :61   1st Qu.:69.00
```

```
#>  Median : 34.00   Median :221.50   (16,24]: 5   Median :79.50
#>  Mean   : 41.59   Mean   :191.54                Mean   :76.87
#>  3rd Qu.: 63.00   3rd Qu.:274.25                3rd Qu.:84.00
#>  Max.   :135.00   Max.   :334.00                Max.   :93.00
#>  NA's   :31
#>  Solar.R.dummy Wind.dummy
#>  FALSE:93      FALSE:92
#>  TRUE : 7      TRUE : 8
#>
#>
#>
#>
#>

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

The ImputationDesc object can be used by function reimpute to impute the test data set the same way as the training data.

```
airq.test.imp = reimpute(airq.test, imp$desc)
head(airq.test.imp)
#>   Ozone Solar.R   Wind Temp Solar.R.dummy Wind.dummy
#> 1   110     207  (0,8]   90         FALSE      FALSE
#> 2    NA     222 (8,16]   92         FALSE      FALSE
#> 3    NA     137 (8,16]   86         FALSE      FALSE
#> 4    44     192 (8,16]   86         FALSE      FALSE
#> 5    28     273 (8,16]   82         FALSE      FALSE
#> 6    65     157 (8,16]   80         FALSE      FALSE
```

Especially when evaluating a machine learning method by some resampling technique you might want that impute/reimpute are called automatically each time before training/prediction. This can be achieved by creating an imputation wrapper.

### 3.3.2 Fusing a learner with imputation

You can couple a Learner with imputation by function makeImputeWrapper which basically has the same formal arguments as impute. Like in the example above we impute `Solar.R` by random numbers from its empirical distribution,

Wind by the predictions of a classification tree and generate dummy variables for both features.

```
lrn = makeImputeWrapper("regr.lm", cols = list(Solar.R =
    imputeHist(),
  Wind = imputeLearner("classif.rpart")), dummy.cols = c("Solar.R",
      "Wind"))
lrn
#> Learner regr.lm.imputed from package stats
#> Type: regr
#> Name: ; Short name:
#> Class: ImputeWrapper
#> Properties: numerics,factors,se,weights,missings
#> Predict-Type: response
#> Hyperparameters:
```

Before training the resulting Learner, impute is applied to the training set. Before prediction reimpute is called on the test set and the ImputationDesc object from the training stage.

We again aim to predict the ozone pollution from the meteorological variables. In order to create the Task we need to delete observations with missing values in the target variable.

```
airq = subset(airq, subset = !is.na(airq$Ozone))
task = makeRegrTask(data = airq, target = "Ozone")
```

In the following the 3-fold cross-validated mean squared error is calculated.

```
rdesc = makeResampleDesc("CV", iters = 3)
r = resample(lrn, task, resampling = rdesc, show.info = FALSE,
    models = TRUE)
r$aggr
#> mse.test.mean
#>      524.3392
```

```
lapply(r$models, getLearnerModel, more.unwrap = TRUE)
#> [[1]]
#>
#> Call:
#> stats::lm(formula = f, data = d)
#>
#> Coefficients:
#>       (Intercept)            Solar.R         Wind(8,16]
#>         -117.0954            0.0853            -27.6763
#>       Wind(16,24]              Temp  Solar.R.dummyTRUE
#>           -9.0988            2.0505            -27.4152
#>    Wind.dummyTRUE
```

```
#>            2.2535
#>
#>
#> [[2]]
#>
#> Call:
#> stats::lm(formula = f, data = d)
#>
#> Coefficients:
#>       (Intercept)            Solar.R         Wind(8,16]
#>         -94.84542            0.03936           -16.26255
#>       Wind(16,24]               Temp  Solar.R.dummyTRUE
#>           -7.00707            1.79513           -11.08578
#>    Wind.dummyTRUE
#>           -0.68340
#>
#>
#> [[3]]
#>
#> Call:
#> stats::lm(formula = f, data = d)
#>
#> Coefficients:
#>       (Intercept)            Solar.R         Wind(8,16]
#>         -57.30438            0.07426           -30.70737
#>       Wind(16,24]               Temp  Solar.R.dummyTRUE
#>           -18.25055            1.35898            -2.16654
#>    Wind.dummyTRUE
#>           -5.56400
```

A second possibility to fuse a learner with imputation is provided by makePreprocWrapperCaret, which is an interface to caret's preProcess function. preProcess only works for numeric features and offers imputation by k-nearest neighbors, bagged trees, and by the median.

## 3.4 Generic Bagging

One reason why random forests perform so well is that they are using bagging as a technique to gain more stability. But why do you want to limit yourself to the classifiers already implemented in well known random forests when it is really easy to build your own with mlr?

Just bag an mlr learner already makeBaggingWrapper.

As in a random forest, we need a Learner which is trained on a subset of the data during each iteration of the bagging process. The subsets are chosen according

to the parameters given to makeBaggingWrapper:

- `bw.iters` On how many subsets (samples) do we want to train our Learner?
- `bw.replace` Sample with replacement (also known as *bootstrapping*)?
- `bw.size` Percentage size of the samples. If `bw.replace = TRUE`, `bw.size = 1` is the default. This does not mean that one sample will contain all the observations as observations will occur multiple times in each sample.
- `bw.feats` Percentage size of randomly selected features for each iteration.

Of course we also need a Learner which we have to pass to makeBaggingWrapper.

```
lrn = makeLearner("classif.rpart")
bag.lrn = makeBaggingWrapper(lrn, bw.iters = 50, bw.replace = TRUE,
    bw.size = 0.8, bw.feats = 3/4)
```

Now we can compare the performance with and without bagging. First let's try it without bagging:

```
rdesc = makeResampleDesc("CV", iters = 10)
r = resample(learner = lrn, task = sonar.task, resampling = rdesc,
    show.info = FALSE)
r$aggr
#> mmce.test.mean
#>      0.2735714
```

And now with bagging:

```
rdesc = makeResampleDesc("CV", iters = 10)
result = resample(learner = bag.lrn, task = sonar.task, resampling =
    rdesc, show.info = FALSE)
result$aggr
#> mmce.test.mean
#>      0.2069048
```

Training more learners takes more time, but can outperform pure learners on noisy data with many features.

### 3.4.1 Changing the type of prediction

In case of a *classification* problem the predicted class labels are determined by majority voting over the predictions of the individual models. Additionally, posterior probabilities can be estimated as the relative proportions of the predicted class labels. For this purpose you have to change the predict type of the *bagging learner* as follows.

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

Note that it is not relevant if the *base learner* itself can predict probabilities and that for this reason the predict type of the *base learner* always has to be `"response"`.

For *regression* the mean value across predictions is computed. Moreover, the standard deviation across predictions is estimated if the predict type of the bagging learner is changed to `"se"`. Below, we give a small example for regression.

```
n = getTaskSize(bh.task)
train.inds = seq(1, n, 3)
test.inds  = setdiff(1:n, train.inds)
lrn = makeLearner("regr.rpart")
bag.lrn = makeBaggingWrapper(lrn)
bag.lrn = setPredictType(bag.lrn, predict.type = "se")
mod = train(learner = bag.lrn, task = bh.task, subset = train.inds)
```

With function getLearnerModel, you can access the models fitted in the individual iterations.

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

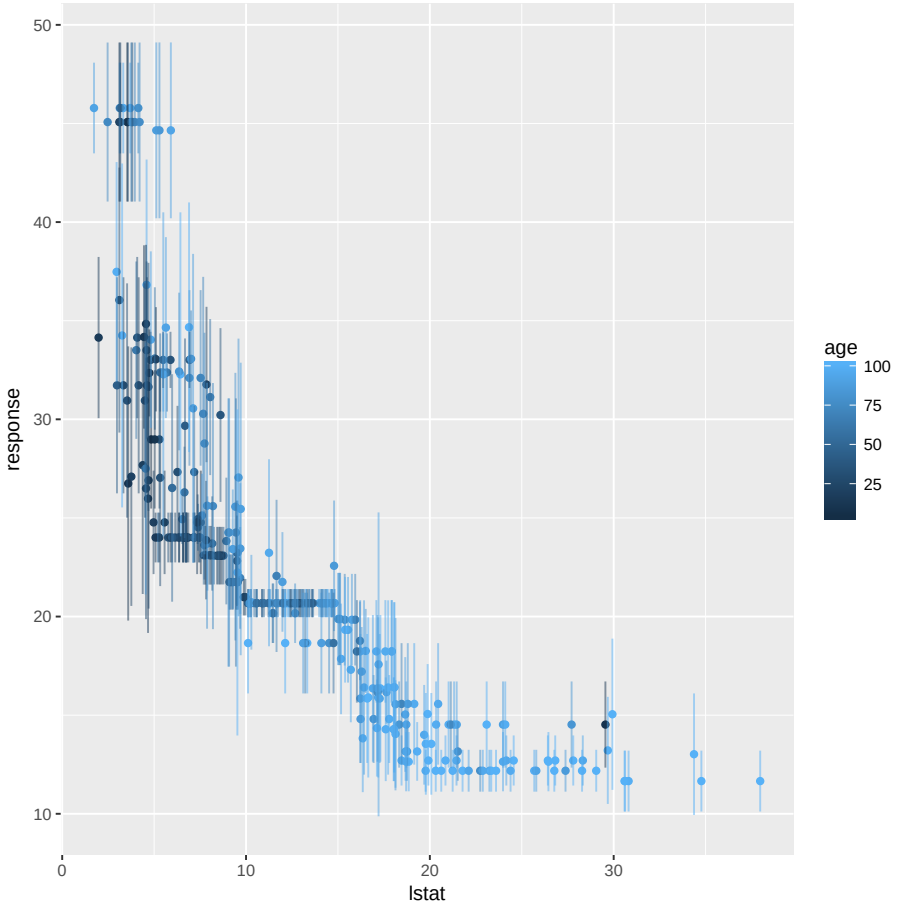
Predict the response and calculate the standard deviation:

```
pred = predict(mod, task = bh.task, subset = test.inds)
head(as.data.frame(pred))
#>   id truth response       se
#> 2  2  21.6 21.74909 1.397291
#> 3  3  34.7 33.50682 4.496894
#> 5  5  36.2 32.37504 1.982321
#> 6  6  28.7 24.00791 1.262131
#> 8  8  27.1 15.57187 3.079712
#> 9  9  16.5 15.04702 3.833359
```

In the column labelled `se` the standard deviation for each prediction is given.

Let's visualise this a bit using ggplot2. Here we plot the percentage of lower status of the population (`lstat`) against the prediction.

```
library("ggplot2")
library("reshape2")
data = cbind(as.data.frame(pred), getTaskData(bh.task, subset =
    test.inds))
g = ggplot(data, aes(x = lstat, y = response, ymin = response-se,
    ymax = response+se, col = age))
g + geom_point() + geom_linerange(alpha=0.5)
```



## 3.5 Advanced Tuning

### 3.5.1 Iterated F-Racing for mixed spaces and dependencies

The package supports a larger number of tuning algorithms, which can all be looked up and selected via TuneControl. One of the cooler algorithms is iterated F-racing from the irace package (technical description here). This not only works for arbitrary parameter types (numeric, integer, discrete, logical), but also for so-called dependent / hierarchical parameters:

```
ps = makeParamSet(
  makeNumericParam("C", lower = -12, upper = 12, trafo = function(x)
      2^x),
  makeDiscreteParam("kernel", values = c("vanilladot", "polydot",
      "rbfdot")),
  makeNumericParam("sigma", lower = -12, upper = 12, trafo =
      function(x) 2^x,
    requires = quote(kernel == "rbfdot")),
  makeIntegerParam("degree", lower = 2L, upper = 5L,
    requires = quote(kernel == "polydot"))
)
ctrl = makeTuneControlIrace(maxExperiments = 200L)
rdesc = makeResampleDesc("Holdout")
res = tuneParams("classif.ksvm", iris.task, rdesc, par.set = ps,
    control = ctrl, show.info = FALSE)
print(head(as.data.frame(res$opt.path)))
#>            C     kernel     sigma degree mmce.test.mean dob eol
#> 1 -2.8894525    polydot        NA      4           0.06   1  NA
#> 2 -2.6793542 vanilladot        NA     NA           0.04   1  NA
#> 3 -0.7855061     rbfdot 11.049783     NA           0.68   1  NA
#> 4  1.7678978    polydot        NA      5           0.14   1  NA
#> 5  9.7729840 vanilladot        NA     NA           0.04   1  NA
#> 6 -2.7930352     rbfdot -7.198476     NA           0.36   1  NA
#>   error.message exec.time
#> 1          <NA>     0.122
#> 2          <NA>     1.815
#> 3          <NA>     0.107
#> 4          <NA>     0.037
#> 5          <NA>     0.020
#> 6          <NA>     0.038
```

See how we made the kernel parameters like `sigma` and `degree` dependent on the `kernel` selection parameters? This approach allows you to tune parameters of multiple kernels at once, efficiently concentrating on the ones which work best for your given data set.

### 3.5.2 Tuning across whole model spaces with ModelMultiplexer

We can now take the following example even one step further. If we use the ModelMultiplexer we can tune over different model classes at once, just as we did with the SVM kernels above.

```
base.learners = list(
  makeLearner("classif.ksvm"),
  makeLearner("classif.randomForest")
)
lrn = makeModelMultiplexer(base.learners)
```

Function makeModelMultiplexerParamSet offers a simple way to construct a parameter set for tuning: The parameter names are prefixed automatically and the `requires` element is set, too, to make all parameters subordinate to `selected.learner`.

```
ps = makeModelMultiplexerParamSet(lrn,
  makeNumericParam("sigma", lower = -12, upper = 12, trafo =
      function(x) 2^x),
  makeIntegerParam("ntree", lower = 1L, upper = 500L)
)
print(ps)
#>                               Type len Def
#> selected.learner           discrete   -   -
#> classif.ksvm.sigma          numeric   -   -
#> classif.randomForest.ntree  integer   -   -
#>                                                        Constr Req
    Tunable
#> selected.learner           classif.ksvm,classif.randomForest   -
       TRUE
#> classif.ksvm.sigma                                  -12 to 12   Y
       TRUE
#> classif.randomForest.ntree                           1 to 500   Y
       TRUE
#>                               Trafo
#> selected.learner                  -
#> classif.ksvm.sigma                Y
#> classif.randomForest.ntree        -

rdesc = makeResampleDesc("CV", iters = 2L)
ctrl = makeTuneControlIrace(maxExperiments = 200L)
res = tuneParams(lrn, iris.task, rdesc, par.set = ps, control =
    ctrl, show.info = FALSE)
print(head(as.data.frame(res$opt.path)))
#>       selected.learner classif.ksvm.sigma
    classif.randomForest.ntree
```

```
#> 1 classif.randomForest                NA
    273
#> 2         classif.ksvm          10.53605
    NA
#> 3         classif.ksvm         -11.79057
    NA
#> 4         classif.ksvm          10.42478
    NA
#> 5 classif.randomForest                NA
    394
#> 6         classif.ksvm          11.02356
    NA
#>   mmce.test.mean dob eol error.message exec.time
#> 1     0.04666667   1  NA          <NA>     1.011
#> 2     0.68000000   1  NA          <NA>     0.049
#> 3     0.52666667   1  NA          <NA>     0.052
#> 4     0.68000000   1  NA          <NA>     0.048
#> 5     0.04666667   1  NA          <NA>     0.087
#> 6     0.68000000   1  NA          <NA>     0.052
```

### 3.5.3 Multi-criteria evaluation and optimization

During tuning you might want to optimize multiple, potentially conflicting, performance measures simultaneously.

In the following example we aim to minimize both, the false positive and the false negative rates (fpr and fnr). We again tune the hyperparameters of an SVM (function ksvm) with a radial basis kernel and use the sonar classification task for illustration. As search strategy we choose a random search.

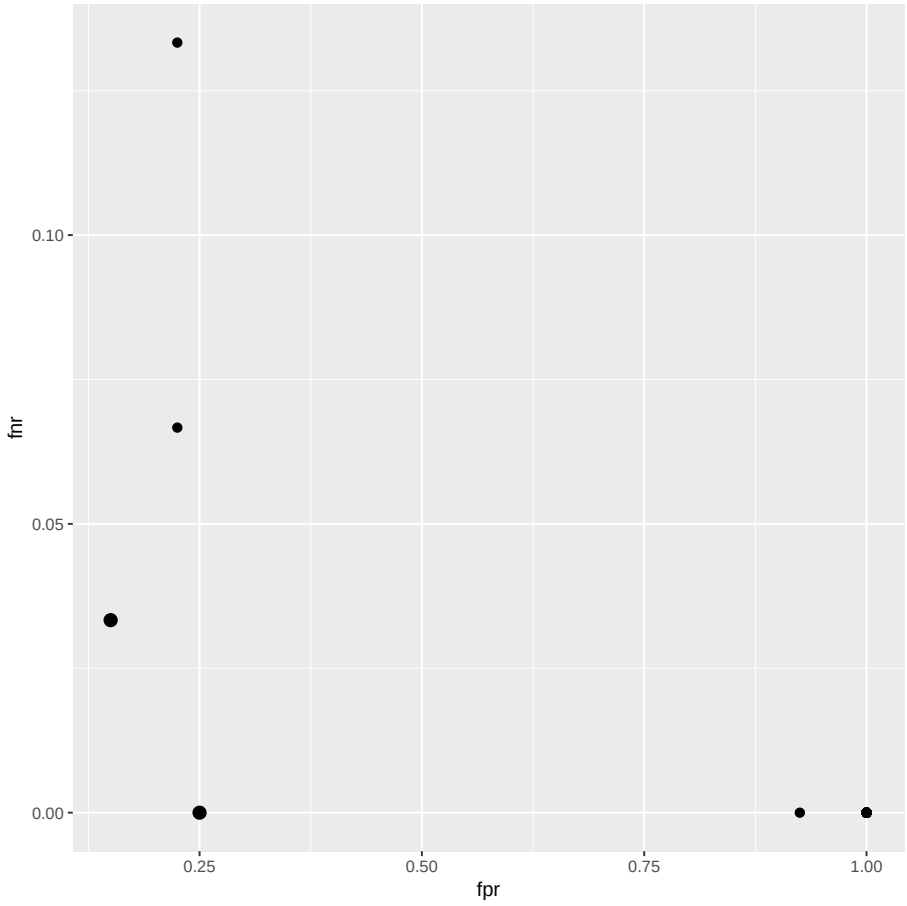
For all available multi-criteria tuning algorithms see TuneMultiCritControl.

```
ps = makeParamSet(
  makeNumericParam("C", lower = -12, upper = 12, trafo = function(x)
      2^x),
  makeNumericParam("sigma", lower = -12, upper = 12, trafo =
      function(x) 2^x)
)
ctrl = makeTuneMultiCritControlRandom(maxit = 30L)
rdesc = makeResampleDesc("Holdout")
res = tuneParamsMultiCrit("classif.ksvm", task = sonar.task,
    resampling = rdesc, par.set = ps,
  measures = list(fpr, fnr), control = ctrl, show.info = FALSE)
res
#> Tune multicrit result:
#> Points on front: 2
```

```
head(as.data.frame(trafoOptPath(res$opt.path)))
#>              C        sigma fpr.test.mean fnr.test.mean dob eol
#> 1 2.837139e-02  0.004605846          1.00    0.00000000   1  NA
#> 2 8.161350e+00 10.073402485          1.00    0.00000000   2  NA
#> 3 2.947371e+03  0.023696559          0.15    0.03333333   3  NA
#> 4 5.020557e-01  0.279973960          1.00    0.00000000   4  NA
#> 5 8.642356e+01 47.600399172          1.00    0.00000000   5  NA
#> 6 3.661447e-04  0.715765529          1.00    0.00000000   6  NA
#>   error.message exec.time
#> 1          <NA>     0.068
#> 2          <NA>     0.045
#> 3          <NA>     0.046
#> 4          <NA>     0.041
#> 5          <NA>     0.051
#> 6          <NA>     0.046
```

The results can be visualized with function plotTuneMultiCritResult. The plot shows the false positive and false negative rates for all parameter settings evaluated during tuning. Points on the Pareto front are slightly increased.

```
plotTuneMultiCritResult(res)
```




## 3.6 Feature Selection

Often, data sets include a large number of features. The technique of extracting a subset of relevant features is called feature selection. Feature selection can enhance the interpretability of the model, speed up the learning process and improve the learner performance. There exist different approaches to identify the relevant features. mlr supports *filter* and *wrapper methods*.

### 3.6.1 Filter methods

Filter methods assign an importance value to each feature. Based on these values the features can be ranked and a feature subset can be selected.

#### 3.6.1.1 Calculating the feature importance

Different methods for calculating the feature importance are built into mlr's function generateFilterValuesData (getFilterValues has been deprecated in favor of generateFilterValuesData.). Currently, classification, regression and survival analysis tasks are supported. A table showing all available methods can be found here.

Function generateFilterValuesData requires the Task and a character string specifying the filter method.

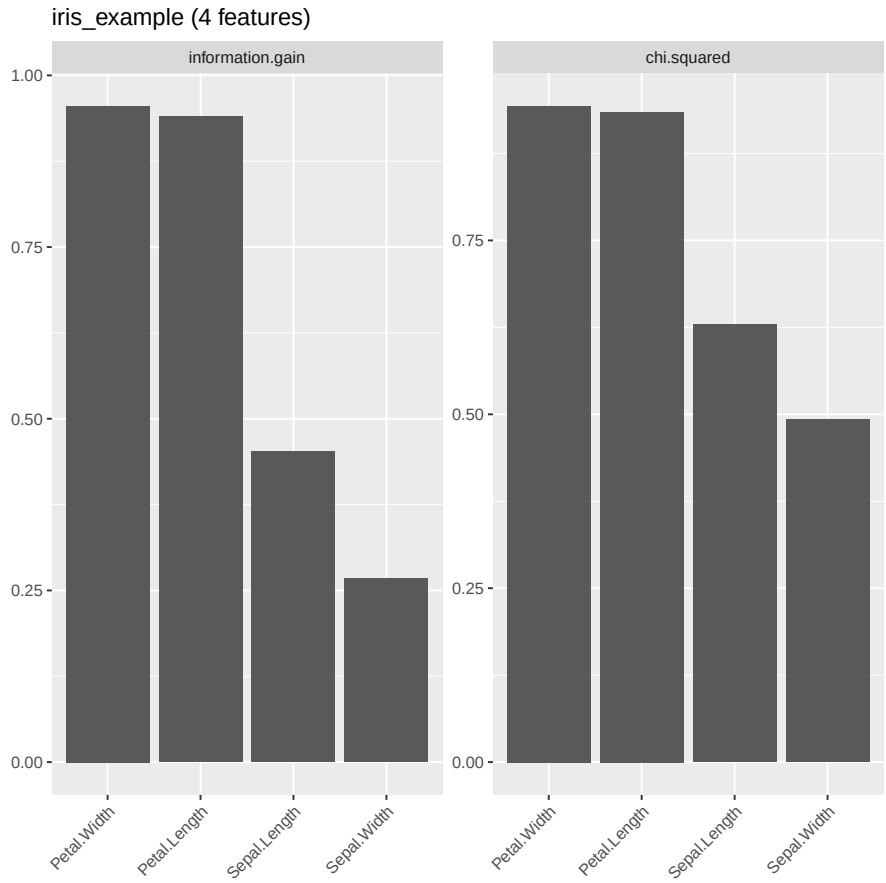
```
fv = generateFilterValuesData(iris.task, method = "information.gain")
fv
#> FilterValues:
#> Task: iris_example
#>           name    type information.gain
#> 1 Sepal.Length numeric        0.4521286
#> 2  Sepal.Width numeric        0.2672750
#> 3 Petal.Length numeric        0.9402853
#> 4  Petal.Width numeric        0.9554360
```

`fv` is a FilterValues object and `fv$data` contains a data.frame that gives the importance values for all features. Optionally, a vector of filter methods can be passed.

```
fv2 = generateFilterValuesData(iris.task, method =
    c("information.gain", "chi.squared"))
fv2$data
#>           name    type information.gain chi.squared
#> 1 Sepal.Length numeric        0.4521286   0.6288067
#> 2  Sepal.Width numeric        0.2672750   0.4922162
#> 3 Petal.Length numeric        0.9402853   0.9346311
#> 4  Petal.Width numeric        0.9554360   0.9432359
```

A bar plot of importance values for the individual features can be obtained using function plotFilterValues.

```
plotFilterValues(fv2)
```




By default plotFilterValues will create facetted subplots if multiple filter methods are passed as input to generateFilterValuesData.

There is also an experimental ggvis plotting function, plotFilterValuesGGVIS. This takes the same arguments as plotFilterValues and produces a shiny application that allows the interactive selection of the displayed filter method, the number of features selected, and the sorting method (e.g., ascending or descending).

```
plotFilterValuesGGVIS(fv2)
```

According to the `"information.gain"` measure, `Petal.Width` and `Petal.Length` contain the most information about the target variable `Species`.

#### 3.6.1.2 Selecting a feature subset

With mlr's function filterFeatures you can create a new Task by leaving out features of lower importance.

There are several ways to select a feature subset based on feature importance values:

- Keep a certain *absolute number* (`abs`) of features with highest importance.
- Keep a certain *percentage* (`perc`) of features with highest importance.
- Keep all features whose importance exceeds a certain *threshold value* (`threshold`).

Function filterFeatures supports these three methods as shown in the following example. Moreover, you can either specify the `method` for calculating the feature importance or you can use previously computed importance values via argument `fval`.

```
### Keep the 2 most important features
filtered.task = filterFeatures(iris.task, method =
    "information.gain", abs = 2)

### Keep the 25% most important features
filtered.task = filterFeatures(iris.task, fval = fv, perc = 0.25)

### Keep all features with importance greater than 0.5
filtered.task = filterFeatures(iris.task, fval = fv, threshold = 0.5)
filtered.task
#> Supervised task: iris_example
#> Type: classif
#> Target: Species
#> Observations: 150
#> Features:
#>    numerics     factors     ordered functionals
#>           2           0           0           0
#> Missings: FALSE
#> Has weights: FALSE
#> Has blocking: FALSE
#> Is spatial: FALSE
#> Classes: 3
#>     setosa versicolor  virginica
#>         50         50         50
#> Positive class: NA
```

#### 3.6.1.3 Fuse a learner with a filter method

Often feature selection based on a filter method is part of the data preprocessing and in a subsequent step a learning method is applied to the filtered data. In a proper experimental setup you might want to automate the selection of the features so that it can be part of the validation method of your choice. A

Learner can be fused with a filter method by function makeFilterWrapper. The resulting Learner has the additional class attribute FilterWrapper.

In the following example we calculate the 10-fold cross-validated error rate (mmce) of the k nearest neighbor classifier with preceding feature selection on the iris data set. We use `"information.gain"` as importance measure and select the 2 features with highest importance. In each resampling iteration feature selection is carried out on the corresponding training data set before fitting the learner.

```
lrn = makeFilterWrapper(learner = "classif.fnn", fw.method =
    "information.gain", fw.abs = 2)
rdesc = makeResampleDesc("CV", iters = 10)
r = resample(learner = lrn, task = iris.task, resampling = rdesc,
    show.info = FALSE, models = TRUE)
r$aggr
#> mmce.test.mean
#>           0.04
```

You may want to know which features have been used. Luckily, we have called resample with the argument `models = TRUE`, which means that `r$models` contains a list of models fitted in the individual resampling iterations. In order to access the selected feature subsets we can call getFilteredFeatures on each model.

```
sfeats = sapply(r$models, getFilteredFeatures)
table(sfeats)
#> sfeats
#> Petal.Length  Petal.Width
#>           10           10
```

The selection of features seems to be very stable. The features `Sepal.Length` and `Sepal.Width` did not make it into a single fold.

#### 3.6.1.4 Tuning the size of the feature subset

In the above examples the number/percentage of features to select or the threshold value have been arbitrarily chosen. If filtering is a preprocessing step before applying a learning method optimal values with regard to the learner performance can be found by tuning.

In the following regression example we consider the BostonHousing data set. We use a linear regression model and determine the optimal percentage value for feature selection such that the 3-fold cross-validated mean squared error of the learner is minimal. As search strategy for tuning a grid search is used.

```
lrn = makeFilterWrapper(learner = "regr.lm", fw.method =
    "chi.squared")
```

```
ps = makeParamSet(makeDiscreteParam("fw.perc", values = seq(0.2,
    0.5, 0.05)))
rdesc = makeResampleDesc("CV", iters = 3)
res = tuneParams(lrn, task = bh.task, resampling = rdesc, par.set =
    ps,
  control = makeTuneControlGrid())
#> [Tune] Started tuning learner regr.lm.filtered for parameter set:
#>             Type len Def                          Constr Req
    Tunable Trafo
#> fw.perc discrete   -   - 0.2,0.25,0.3,0.35,0.4,0.45,0.5   -
    TRUE     -
#> With control class: TuneControlGrid
#> Imputation value: Inf
#> [Tune-x] 1: fw.perc=0.2
#> [Tune-y] 1: mse.test.mean=40.5957841; time: 0.0 min
#> [Tune-x] 2: fw.perc=0.25
#> [Tune-y] 2: mse.test.mean=40.5957841; time: 0.0 min
#> [Tune-x] 3: fw.perc=0.3
#> [Tune-y] 3: mse.test.mean=37.0559228; time: 0.0 min
#> [Tune-x] 4: fw.perc=0.35
#> [Tune-y] 4: mse.test.mean=35.8371232; time: 0.0 min
#> [Tune-x] 5: fw.perc=0.4
#> [Tune-y] 5: mse.test.mean=35.8371232; time: 0.0 min
#> [Tune-x] 6: fw.perc=0.45
#> [Tune-y] 6: mse.test.mean=27.3995509; time: 0.0 min
#> [Tune-x] 7: fw.perc=0.5
#> [Tune-y] 7: mse.test.mean=27.3995509; time: 0.0 min
#> [Tune] Result: fw.perc=0.5 : mse.test.mean=27.3995509
res
#> Tune result:
#> Op. pars: fw.perc=0.5
#> mse.test.mean=27.3995509
```

The performance of all percentage values visited during tuning is:

```
as.data.frame(res$opt.path)
#>   fw.perc mse.test.mean dob eol error.message exec.time
#> 1     0.2      40.59578   1  NA          <NA>     0.319
#> 2    0.25      40.59578   2  NA          <NA>     0.289
#> 3     0.3      37.05592   3  NA          <NA>     0.215
#> 4    0.35      35.83712   4  NA          <NA>     0.203
#> 5     0.4      35.83712   5  NA          <NA>     0.210
#> 6    0.45      27.39955   6  NA          <NA>     0.213
#> 7     0.5      27.39955   7  NA          <NA>     0.217
```

The optimal percentage and the corresponding performance can be accessed as follows:

```
res$x
#> $fw.perc
#> [1] 0.5
res$y
#> mse.test.mean
#>      27.39955
```

After tuning we can generate a new wrapped learner with the optimal percentage value for further use.

```
lrn = makeFilterWrapper(learner = "regr.lm", fw.method =
    "chi.squared", fw.perc = res$x$fw.perc)
mod = train(lrn, bh.task)
mod
#> Model for learner.id=regr.lm.filtered; learner.class=FilterWrapper
#> Trained on: task.id = BostonHousing-example; obs = 506; features
    = 13
#> Hyperparameters: fw.method=chi.squared,fw.perc=0.5

getFilteredFeatures(mod)
#> [1] "crim"  "zn"    "rm"    "dis"   "rad"   "lstat"
```

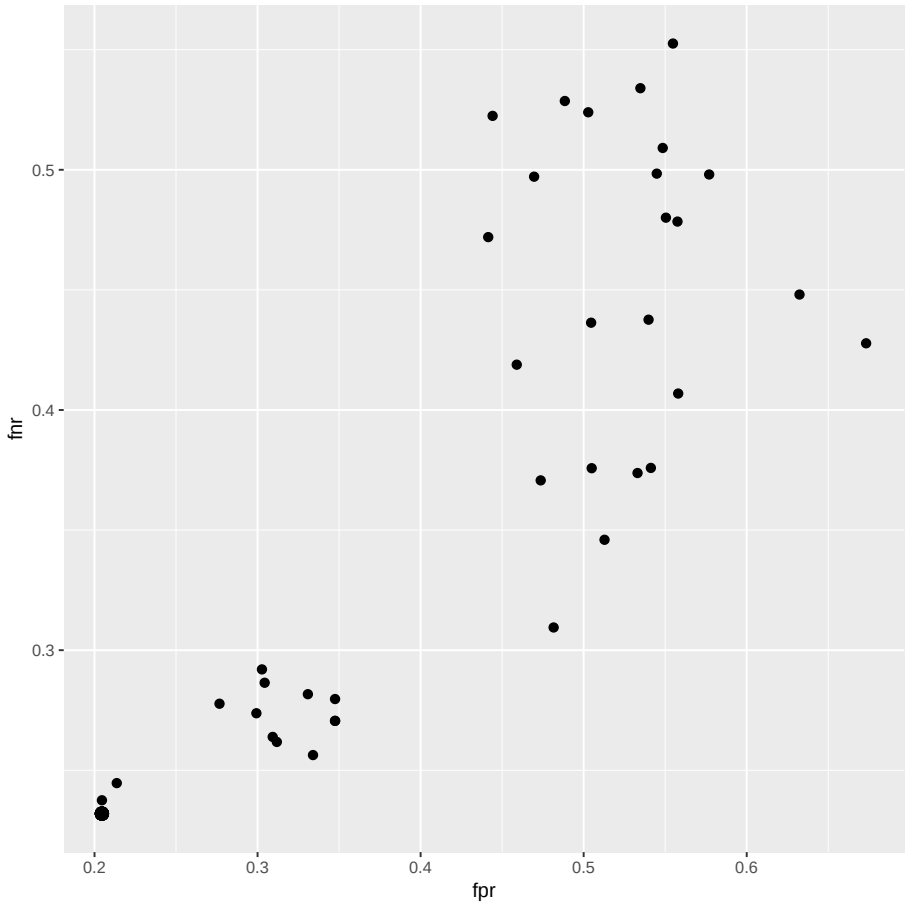
Here is another example using multi-criteria tuning. We consider linear discriminant analysis with precedent feature selection based on the Chi-squared statistic of independence (`"chi.squared"`) on the Sonar data set and tune the threshold value. During tuning both, the false positive and the false negative rate (fpr and fnr), are minimized. As search strategy we choose a random search (see makeTuneMultiCritControlRandom).

```
lrn = makeFilterWrapper(learner = "classif.lda", fw.method =
    "chi.squared")
ps = makeParamSet(makeNumericParam("fw.threshold", lower = 0.1,
    upper = 0.9))
rdesc = makeResampleDesc("CV", iters = 10)
res = tuneParamsMultiCrit(lrn, task = sonar.task, resampling =
    rdesc, par.set = ps,
  measures = list(fpr, fnr), control =
      makeTuneMultiCritControlRandom(maxit = 50L),
  show.info = FALSE)
res
#> Tune multicrit result:
#> Points on front: 13
head(as.data.frame(res$opt.path))
#>   fw.threshold fpr.test.mean fnr.test.mean dob eol error.message
    exec.time
#> 1    0.4892321     0.3092818     0.2639033   1  NA          <NA>
       1.852
```

```
#> 2    0.2481696    0.2045499    0.2319697  2  NA          <NA>
       1.809
#> 3    0.7691875    0.5128000    0.3459740  3  NA          <NA>
       1.749
#> 4    0.1470133    0.2045499    0.2319697  4  NA          <NA>
       1.860
#> 5    0.5958241    0.5028216    0.5239538  5  NA          <NA>
       1.729
#> 6    0.6892421    0.6323959    0.4480808  6  NA          <NA>
       1.758
```

The results can be visualized with function plotTuneMultiCritResult. The plot shows the false positive and false negative rates for all parameter values visited during tuning. The size of the points on the Pareto front is slightly increased.

```
plotTuneMultiCritResult(res)
```



### 3.6.2 Wrapper methods

Wrapper methods use the performance of a learning algorithm to assess the usefulness of a feature set. In order to select a feature subset a learner is trained repeatedly on different feature subsets and the subset which leads to the best learner performance is chosen.

In order to use the wrapper approach we have to decide:

- How to assess the performance: This involves choosing a performance measure that serves as feature selection criterion and a resampling strategy.
- Which learning method to use.
- How to search the space of possible feature subsets.

The search strategy is defined by functions following the naming convention `makeFeatSelControl<search_strategy`. The following search strategies are available:

- Exhaustive search (makeFeatSelControlExhaustive),
- Genetic algorithm (makeFeatSelControlGA),
- Random search (makeFeatSelControlRandom),
- Deterministic forward or backward search (makeFeatSelControlSequential).

#### 3.6.2.1 Select a feature subset

Feature selection can be conducted with function selectFeatures.

In the following example we perform a random search on the Wisconsin Prognostic Breast Cancer data set. As learning method we use the Cox proportional hazards model. The performance is assessed by the holdout estimate of the concordance index (cindex).

```
### Specify the search strategy
ctrl = makeFeatSelControlRandom(maxit = 20L)
ctrl
#> FeatSel control: FeatSelControlRandom
#> Same resampling instance: TRUE
#> Imputation value: <worst>
#> Max. features: <not used>
#> Max. iterations: 20
#> Tune threshold: FALSE
#> Further arguments: prob=0.5
```

`ctrl` is a FeatSelControl object that contains information about the search strategy and potential parameter values.

```
### Resample description
rdesc = makeResampleDesc("Holdout")

### Select features
sfeats = selectFeatures(learner = "surv.coxph", task = wpbc.task,
    resampling = rdesc,
  control = ctrl, show.info = FALSE)
sfeats
#> FeatSel result:
#> Features (17): mean_radius, mean_area, mean_smoothness,
    mean_concavepoints, mean_symmetry, mean_fractaldim, SE_texture,
    SE_perimeter, SE_smoothness, SE_compactness, SE_concavity,
    SE_concavepoints, worst_area, worst_compactness,
    worst_concavepoints, tsize, pnodes
#> cindex.test.mean=0.7137990
```

`sfeats`is a FeatSelResult object. The selected features and the corresponding performance can be accessed as follows:

```
sfeats$x
#>  [1] "mean_radius"         "mean_area"           "mean_smoothness"
#>  [4] "mean_concavepoints"  "mean_symmetry"       "mean_fractaldim"
#>  [7] "SE_texture"          "SE_perimeter"        "SE_smoothness"
#> [10] "SE_compactness"      "SE_concavity"
    "SE_concavepoints"
#> [13] "worst_area"          "worst_compactness"
    "worst_concavepoints"
#> [16] "tsize"               "pnodes"
sfeats$y
#> cindex.test.mean
#>         0.713799
```

In a second example we fit a simple linear regression model to the BostonHousing data set and use a sequential search to find a feature set that minimizes the mean squared error (mse). `method = "sfs"` indicates that we want to conduct a sequential forward search where features are added to the model until the performance cannot be improved anymore. See the documentation page makeFeatSelControlSequential for other available sequential search methods. The search is stopped if the improvement is smaller than `alpha = 0.02`.

```
### Specify the search strategy
ctrl = makeFeatSelControlSequential(method = "sfs", alpha = 0.02)

### Select features
rdesc = makeResampleDesc("CV", iters = 10)
sfeats = selectFeatures(learner = "regr.lm", task = bh.task,
    resampling = rdesc, control = ctrl,
```

```
  show.info = FALSE)
sfeats
#> FeatSel result:
#> Features (11): crim, zn, chas, nox, rm, dis, rad, tax, ptratio,
    b, lstat
#> mse.test.mean=23.7319094
```

Further information about the sequential feature selection process can be obtained by function analyzeFeatSelResult.

```
analyzeFeatSelResult(sfeats)
#> Features         : 11
#> Performance      : mse.test.mean=23.7319094
#> crim, zn, chas, nox, rm, dis, rad, tax, ptratio, b, lstat
#>
#> Path to optimum:
#> - Features:    0  Init   :                       Perf = 84.831
    Diff: NA  *
#> - Features:    1  Add    : lstat                 Perf = 38.894
    Diff: 45.936  *
#> - Features:    2  Add    : rm                    Perf = 31.279
    Diff: 7.6156  *
#> - Features:    3  Add    : ptratio               Perf = 28.108
    Diff: 3.1703  *
#> - Features:    4  Add    : dis                   Perf = 27.48
    Diff: 0.62813  *
#> - Features:    5  Add    : nox                   Perf = 26.079
    Diff: 1.4008  *
#> - Features:    6  Add    : b                     Perf = 25.563
    Diff: 0.51594  *
#> - Features:    7  Add    : chas                  Perf = 25.132
    Diff: 0.43097  *
#> - Features:    8  Add    : zn                    Perf = 24.792
    Diff: 0.34018  *
#> - Features:    9  Add    : rad                   Perf = 24.599
    Diff: 0.19327  *
#> - Features:   10  Add    : tax                   Perf = 24.082
    Diff: 0.51706  *
#> - Features:   11  Add    : crim                  Perf = 23.732
    Diff: 0.35  *
#>
#> Stopped, because no improving feature was found.
```

#### 3.6.2.2 Fuse a learner with feature selection

A Learner can be fused with a feature selection strategy (i.e., a search strategy, a performance measure and a resampling strategy) by function makeFeatSelWrapper. During training features are selected according to the specified selection scheme. Then, the learner is trained on the selected feature subset.

```
rdesc = makeResampleDesc("CV", iters = 3)
lrn = makeFeatSelWrapper("surv.coxph", resampling = rdesc,
  control = makeFeatSelControlRandom(maxit = 10), show.info = FALSE)
mod = train(lrn, task = wpbc.task)
mod
#> Model for learner.id=surv.coxph.featsel;
    learner.class=FeatSelWrapper
#> Trained on: task.id = wpbc-example; obs = 194; features = 32
#> Hyperparameters:
```

The result of the feature selection can be extracted by function getFeatSelResult.

```
sfeats = getFeatSelResult(mod)
sfeats
#> FeatSel result:
#> Features (19): mean_radius, mean_texture, mean_perimeter,
    mean_area, mean_smoothness, mean_compactness,
    mean_concavepoints, mean_fractaldim, SE_compactness,
    SE_concavity, SE_concavepoints, SE_symmetry, worst_texture,
    worst_perimeter, worst_area, worst_concavepoints,
    worst_symmetry, tsize, pnodes
#> cindex.test.mean=0.6308754
```

The selected features are:

```
sfeats$x
#>  [1] "mean_radius"         "mean_texture"        "mean_perimeter"
#>  [4] "mean_area"           "mean_smoothness"
    "mean_compactness"
#>  [7] "mean_concavepoints"  "mean_fractaldim"     "SE_compactness"
#> [10] "SE_concavity"        "SE_concavepoints"    "SE_symmetry"
#> [13] "worst_texture"       "worst_perimeter"     "worst_area"
#> [16] "worst_concavepoints" "worst_symmetry"      "tsize"
#> [19] "pnodes"
```

The 5-fold cross-validated performance of the learner specified above can be computed as follows:

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

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

```
r$aggr
#> cindex.test.mean
#>         0.632357
```

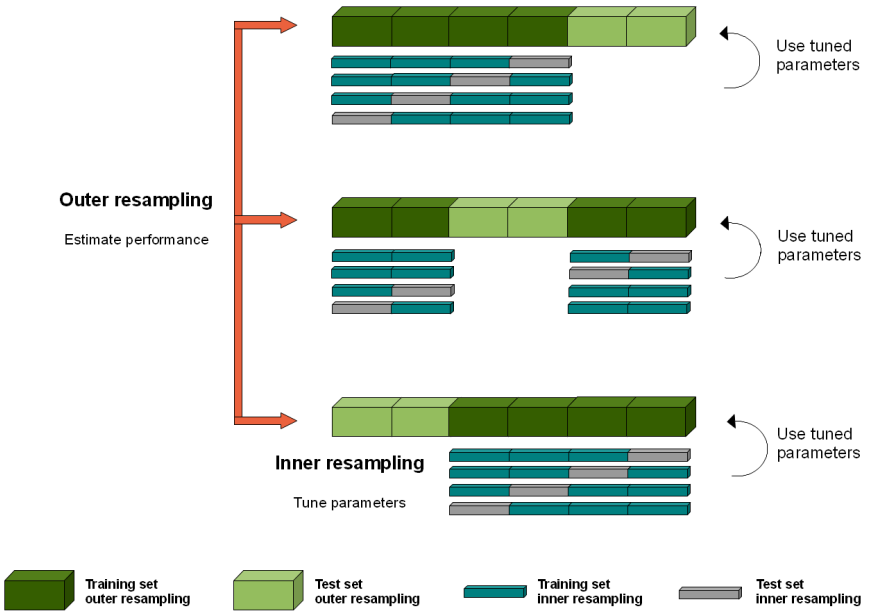
The selected feature sets in the individual resampling iterations can be extracted as follows:

```
lapply(r$models, getFeatSelResult)
#> [[1]]
#> FeatSel result:
#> Features (18): mean_texture, mean_area, mean_smoothness,
    mean_compactness, mean_concavity, mean_symmetry, SE_radius,
    SE_compactness, SE_concavity, SE_concavepoints, SE_fractaldim,
    worst_radius, worst_smoothness, worst_compactness,
    worst_concavity, worst_symmetry, tsize, pnodes
#> cindex.test.mean=0.6599048
#>
#> [[2]]
#> FeatSel result:
#> Features (12): mean_area, mean_compactness, mean_symmetry,
    mean_fractaldim, SE_perimeter, SE_area, SE_concavity,
    SE_symmetry, worst_texture, worst_smoothness, worst_fractaldim,
    tsize
#> cindex.test.mean=0.6524759
#>
#> [[3]]
#> FeatSel result:
#> Features (14): mean_compactness, mean_symmetry, mean_fractaldim,
    SE_radius, SE_perimeter, SE_smoothness, SE_concavity,
    SE_concavepoints, SE_fractaldim, worst_concavity,
    worst_concavepoints, worst_symmetry, worst_fractaldim, pnodes
#> cindex.test.mean=0.6074810
#>
#> [[4]]
#> FeatSel result:
#> Features (18): mean_radius, mean_texture, mean_perimeter,
    mean_compactness, mean_concavity, SE_texture, SE_area,
    SE_smoothness, SE_concavity, SE_symmetry, SE_fractaldim,
    worst_radius, worst_compactness, worst_concavepoints,
    worst_symmetry, worst_fractaldim, tsize, pnodes
#> cindex.test.mean=0.6526122
#>
#> [[5]]
#> FeatSel result:
#> Features (14): mean_radius, mean_texture, mean_compactness,
    mean_concavepoints, mean_symmetry, SE_texture, SE_compactness,
```

```
    SE_symmetry, SE_fractaldim, worst_radius, worst_smoothness,
    worst_compactness, worst_concavity, pnodes
#> cindex.test.mean=0.6262729
```

## 3.7 Nested Resampling

In order to obtain honest performance estimates for a learner all parts of the model building like preprocessing and model selection steps should be included in the resampling, i.e., repeated for every pair of training/test data. For steps that themselves require resampling like parameter tuning or feature selection (via the wrapper approach) this results in two nested resampling loops.




The graphic above illustrates nested resampling for parameter tuning with 3-fold cross-validation in the outer and 4-fold cross-validation in the inner loop.

In the outer resampling loop, we have three pairs of training/test sets. On each of these outer training sets parameter tuning is done, thereby executing the inner resampling loop. This way, we get one set of selected hyperparameters for each outer training set. Then the learner is fitted on each outer training set using the corresponding selected hyperparameters and its performance is evaluated on the outer test sets.

In mlr, you can get nested resampling for free without programming any looping by using the wrapper functionality. This works as follows:

1. Generate a wrapped Learner via function makeTuneWrapper or makeFeatSelWrapper. Specify the inner resampling strategy using their `resampling` argument.
2. Call function resample (see also the section about resampling) and pass the outer resampling strategy to its `resampling` argument.

You can freely combine different inner and outer resampling strategies.

The outer strategy can be a resample description (ResampleDesc) or a resample instance (ResampleInstance). A common setup is prediction and performance evaluation on a fixed outer test set. This can be achieved by using function makeFixedHoldoutInstance to generate the outer ResampleInstance.

The inner resampling strategy should preferably be a ResampleDesc, as the sizes of the outer training sets might differ. Per default, the inner resample description is instantiated once for every outer training set. This way during tuning/feature selection all parameter or feature sets are compared on the same inner training/test sets to reduce variance. You can also turn this off using the `same.resampling.instance` argument of makeTuneControl* or makeFeatSelControl*.

Nested resampling is computationally expensive. For this reason in the examples shown below we use relatively small search spaces and a low number of resampling iterations. In practice, you normally have to increase both. As this is computationally intensive you might want to have a look at section parallelization.

### 3.7.1 Tuning

As you might recall from the tutorial page about tuning, you need to define a search space by function makeParamSet, a search strategy by makeTuneControl*, and a method to evaluate hyperparameter settings (i.e., the inner resampling strategy and a performance measure).

Below is a classification example. We evaluate the performance of a support vector machine (ksvm) with tuned cost parameter `C` and RBF kernel parameter `sigma`. We use 3-fold cross-validation in the outer and subsampling with 2 iterations in the inner loop. For tuning a grid search is used to find the hyperparameters with lowest error rate (mmce is the default measure for classification). The wrapped Learner is generated by calling makeTuneWrapper.

Note that in practice the parameter set should be larger. A common recommendation is `2^(-12:12)` for both `C` and `sigma`.

```
### Tuning in inner resampling loop
ps = makeParamSet(
  makeDiscreteParam("C", values = 2^(-2:2)),
  makeDiscreteParam("sigma", values = 2^(-2:2))
```

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

### Outer resampling loop
outer = makeResampleDesc("CV", iters = 3)
r = resample(lrn, iris.task, resampling = outer, extract =
    getTuneResult, show.info = FALSE)

r
#> Resample Result
#> Task: iris_example
#> Learner: classif.ksvm.tuned
#> Aggr perf: mmce.test.mean=0.0533333
#> Runtime: 3.79262
```

You can obtain the error rates on the 3 outer test sets by:

```
r$measures.test
#>   iter mmce
#> 1    1 0.02
#> 2    2 0.06
#> 3    3 0.08
```

#### 3.7.1.1 Accessing the tuning result

We have kept the results of the tuning for further evaluations. For example one might want to find out, if the best obtained configurations vary for the different outer splits. As storing entire models may be expensive (but possible by setting `models = TRUE`) we used the `extract` option of resample. Function getTuneResult returns, among other things, the optimal hyperparameter values and the optimization path for each iteration of the outer resampling loop. Note that the performance values shown when printing `r$extract` are the aggregated performances resulting from inner resampling on the outer training set for the best hyperparameter configurations (not to be confused with `r$measures.test` shown above).

```
r$extract
#> [[1]]
#> Tune result:
#> Op. pars: C=2; sigma=0.25
#> mmce.test.mean=0.0147059
#>
#> [[2]]
```

```
#> Tune result:
#> Op. pars: C=4; sigma=0.25
#> mmce.test.mean=0.0000000
#>
#> [[3]]
#> Tune result:
#> Op. pars: C=4; sigma=0.25
#> mmce.test.mean=0.0735294

names(r$extract[[1]])
#> [1] "learner"   "control"   "x"         "y"         "threshold"
    "opt.path"
```

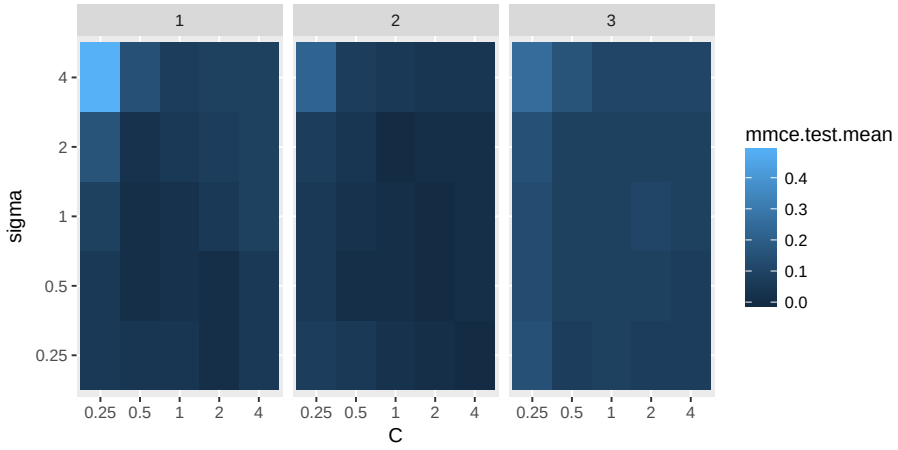
We can compare the optimal parameter settings obtained in the 3 resampling iterations. As you can see, the optimal configuration usually depends on the data. You may be able to identify a *range* of parameter settings that achieve good performance though, e.g., the values for `C` should be at least 1 and the values for `sigma` should be between 0 and 1.

With function getNestedTuneResultsOptPathDf you can extract the optimization paths for the 3 outer cross-validation iterations for further inspection and analysis. These are stacked in one data.frame with column `iter` indicating the resampling iteration.

```
opt.paths = getNestedTuneResultsOptPathDf(r)
head(opt.paths, 10)
#>       C sigma mmce.test.mean dob eol error.message exec.time iter
#> 1  0.25  0.25     0.05882353   1  NA          <NA>     0.063    1
#> 2   0.5  0.25     0.04411765   2  NA          <NA>     0.051    1
#> 3     1  0.25     0.04411765   3  NA          <NA>     0.052    1
#> 4     2  0.25     0.01470588   4  NA          <NA>     0.040    1
#> 5     4  0.25     0.05882353   5  NA          <NA>     0.041    1
#> 6  0.25   0.5     0.05882353   6  NA          <NA>     0.039    1
#> 7   0.5   0.5     0.01470588   7  NA          <NA>     0.039    1
#> 8     1   0.5     0.02941176   8  NA          <NA>     0.039    1
#> 9     2   0.5     0.01470588   9  NA          <NA>     0.040    1
#> 10    4   0.5     0.05882353  10  NA          <NA>     0.039    1
```

Below we visualize the `opt.paths` for the 3 outer resampling iterations.

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




Another useful function is getNestedTuneResultsX, which extracts the best found hyperparameter settings for each outer resampling iteration.

```
getNestedTuneResultsX(r)
#>   C sigma
#> 1 2  0.25
#> 2 4  0.25
#> 3 4  0.25
```

### 3.7.2 Feature selection

As you might recall from the section about feature selection, mlr supports the filter and the wrapper approach.

#### 3.7.2.1 Wrapper methods

Wrapper methods use the performance of a learning algorithm to assess the usefulness of a feature set. In order to select a feature subset a learner is trained repeatedly on different feature subsets and the subset which leads to the best learner performance is chosen.

For feature selection in the inner resampling loop, you need to choose a search strategy (function makeFeatSelControl*), a performance measure and the inner resampling strategy. Then use function makeFeatSelWrapper to bind everything together.

Below we use sequential forward selection with linear regression on the BostonHousing data set (bh.task).

```
### Feature selection in inner resampling loop
inner = makeResampleDesc("CV", iters = 3)
```

```
lrn = makeFeatSelWrapper("regr.lm", resampling = inner,
  control = makeFeatSelControlSequential(method = "sfs"), show.info
      = FALSE)

### Outer resampling loop
outer = makeResampleDesc("Subsample", iters = 2)
r = resample(learner = lrn, task = bh.task, resampling = outer,
    extract = getFeatSelResult,
  show.info = FALSE)

r
#> Resample Result
#> Task: BostonHousing-example
#> Learner: regr.lm.featsel
#> Aggr perf: mse.test.mean=31.6991293
#> Runtime: 8.9968

r$measures.test
#>   iter      mse
#> 1    1 35.08611
#> 2    2 28.31215
```

#### 3.7.2.1.1 Accessing the selected features

The result of the feature selection can be extracted by function getFeatSelResult. It is also possible to keep whole models by setting `models = TRUE` when calling resample.

```
r$extract
#> [[1]]
#> FeatSel result:
#> Features (10): crim, zn, indus, nox, rm, dis, rad, tax, ptratio,
    lstat
#> mse.test.mean=20.1593896
#>
#> [[2]]
#> FeatSel result:
#> Features (9): zn, nox, rm, dis, rad, tax, ptratio, b, lstat
#> mse.test.mean=22.5966796

### Selected features in the first outer resampling iteration
r$extract[[1]]$x
#>  [1] "crim"    "zn"      "indus"   "nox"     "rm"      "dis"
    "rad"
#>  [8] "tax"     "ptratio" "lstat"
```

```
### Resampled performance of the selected feature subset on the
    first inner training set
r$extract[[1]]$y
#> mse.test.mean
#>      20.15939
```

As for tuning, you can extract the optimization paths. The resulting data.frames contain, among others, binary columns for all features, indicating if they were included in the linear regression model, and the corresponding performances.

```
opt.paths = lapply(r$extract, function(x) as.data.frame(x$opt.path))
head(opt.paths[[1]])
#>   crim zn indus chas nox rm age dis rad tax ptratio b lstat
    mse.test.mean
#> 1    0  0     0    0   0  0   0   0   0   0       0 0     0
    80.33019
#> 2    1  0     0    0   0  0   0   0   0   0       0 0     0
    65.95316
#> 3    0  1     0    0   0  0   0   0   0   0       0 0     0
    69.15417
#> 4    0  0     1    0   0  0   0   0   0   0       0 0     0
    55.75473
#> 5    0  0     0    1   0  0   0   0   0   0       0 0     0
    80.48765
#> 6    0  0     0    0   1  0   0   0   0   0       0 0     0
    63.06724
#>   dob eol error.message exec.time
#> 1   1   2          <NA>     0.033
#> 2   2   2          <NA>     0.055
#> 3   2   2          <NA>     0.054
#> 4   2   2          <NA>     0.054
#> 5   2   2          <NA>     0.058
#> 6   2   2          <NA>     0.054
```

An easy-to-read version of the optimization path for sequential feature selection can be obtained with function analyzeFeatSelResult.

```
analyzeFeatSelResult(r$extract[[1]])
#> Features         : 10
#> Performance      : mse.test.mean=20.1593896
#> crim, zn, indus, nox, rm, dis, rad, tax, ptratio, lstat
#>
#> Path to optimum:
#> - Features:    0  Init   :                       Perf = 80.33
    Diff: NA  *
#> - Features:    1  Add    : lstat                 Perf = 36.451
    Diff: 43.879  *
```

```
#> - Features:    2  Add    : rm                    Perf = 27.289
    Diff: 9.1623  *
#> - Features:    3  Add    : ptratio               Perf = 24.004
    Diff: 3.2849  *
#> - Features:    4  Add    : nox                   Perf = 23.513
    Diff: 0.49082  *
#> - Features:    5  Add    : dis                   Perf = 21.49
    Diff: 2.023  *
#> - Features:    6  Add    : crim                  Perf = 21.12
    Diff: 0.37008  *
#> - Features:    7  Add    : indus                 Perf = 20.82
    Diff: 0.29994  *
#> - Features:    8  Add    : rad                   Perf = 20.609
    Diff: 0.21054  *
#> - Features:    9  Add    : tax                   Perf = 20.209
    Diff: 0.40059  *
#> - Features:   10  Add    : zn                    Perf = 20.159
    Diff: 0.049441  *
#>
#> Stopped, because no improving feature was found.
```

#### 3.7.2.2 Filter methods with tuning

Filter methods assign an importance value to each feature. Based on these values you can select a feature subset by either keeping all features with importance higher than a certain threshold or by keeping a fixed number or percentage of the highest ranking features. Often, neither the theshold nor the number or percentage of features is known in advance and thus tuning is necessary.

In the example below the threshold value (`fw.threshold`) is tuned in the inner resampling loop. For this purpose the base Learner `"regr.lm"` is wrapped two times. First, makeFilterWrapper is used to fuse linear regression with a feature filtering preprocessing step. Then a tuning step is added by makeTuneWrapper.

```
### Tuning of the percentage of selected filters in the inner loop
lrn = makeFilterWrapper(learner = "regr.lm", fw.method =
    "chi.squared")
ps = makeParamSet(makeDiscreteParam("fw.threshold", values = seq(0,
    1, 0.2)))
ctrl = makeTuneControlGrid()
inner = makeResampleDesc("CV", iters = 3)
lrn = makeTuneWrapper(lrn, resampling = inner, par.set = ps, control
    = ctrl, show.info = FALSE)

### Outer resampling loop
outer = makeResampleDesc("CV", iters = 3)
```

```
r = resample(learner = lrn, task = bh.task, resampling = outer,
    models = TRUE, show.info = FALSE)
r
#> Resample Result
#> Task: BostonHousing-example
#> Learner: regr.lm.filtered.tuned
#> Aggr perf: mse.test.mean=25.3915155
#> Runtime: 4.09566
```

#### 3.7.2.2.1 Accessing the selected features and optimal percentage

In the above example we kept the complete models.

Below are some examples that show how to extract information from the models.

```
r$models
#> [[1]]
#> Model for learner.id=regr.lm.filtered.tuned;
    learner.class=TuneWrapper
#> Trained on: task.id = BostonHousing-example; obs = 337; features
    = 13
#> Hyperparameters: fw.method=chi.squared
#>
#> [[2]]
#> Model for learner.id=regr.lm.filtered.tuned;
    learner.class=TuneWrapper
#> Trained on: task.id = BostonHousing-example; obs = 338; features
    = 13
#> Hyperparameters: fw.method=chi.squared
#>
#> [[3]]
#> Model for learner.id=regr.lm.filtered.tuned;
    learner.class=TuneWrapper
#> Trained on: task.id = BostonHousing-example; obs = 337; features
    = 13
#> Hyperparameters: fw.method=chi.squared
```

The result of the feature selection can be extracted by function getFilteredFeatures. Almost always all 13 features are selected.

```
lapply(r$models, function(x)
    getFilteredFeatures(x$learner.model$next.model))
#> [[1]]
#>  [1] "crim"    "zn"      "indus"   "chas"    "nox"     "rm"
    "age"
#>  [8] "dis"     "rad"     "tax"     "ptratio" "b"       "lstat"
#>
```

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

Below the tune results and optimization paths are accessed.

```
res = lapply(r$models, getTuneResult)
res
#> [[1]]
#> Tune result:
#> Op. pars: fw.threshold=0
#> mse.test.mean=24.8915992
#>
#> [[2]]
#> Tune result:
#> Op. pars: fw.threshold=0.4
#> mse.test.mean=27.1812625
#>
#> [[3]]
#> Tune result:
#> Op. pars: fw.threshold=0
#> mse.test.mean=19.7012695

opt.paths = lapply(res, function(x) as.data.frame(x$opt.path))
opt.paths[[1]]
#>   fw.threshold mse.test.mean dob eol error.message exec.time
#> 1            0      24.89160   1  NA          <NA>     0.226
#> 2          0.2      25.18817   2  NA          <NA>     0.218
#> 3          0.4      25.18817   3  NA          <NA>     0.207
#> 4          0.6      32.15930   4  NA          <NA>     0.210
#> 5          0.8      90.89848   5  NA          <NA>     0.185
#> 6            1      90.89848   6  NA          <NA>     0.198
```

### 3.7.3 Benchmark experiments

In a benchmark experiment multiple learners are compared on one or several tasks (see also the section about benchmarking). Nested resampling in benchmark experiments is achieved the same way as in resampling:

- First, use makeTuneWrapper or makeFeatSelWrapper to generate wrapped Learners with the inner resampling strategies of your choice.
- Second, call benchmark and specify the outer resampling strategies for all tasks.

The inner resampling strategies should be resample descriptions. You can use different inner resampling strategies for different wrapped learners. For example it might be practical to do fewer subsampling or bootstrap iterations for slower learners.

If you have larger benchmark experiments you might want to have a look at the section about parallelization.

As mentioned in the section about benchmark experiments you can also use different resampling strategies for different learning tasks by passing a list of resampling descriptions or instances to benchmark.

We will see three examples to show different benchmark settings:

1. Two data sets + two classification algorithms + tuning
2. One data set + two regression algorithms + feature selection
3. One data set + two regression algorithms + feature filtering + tuning

#### 3.7.3.1 Example 1: Two tasks, two learners, tuning

Below is a benchmark experiment with two data sets, iris and sonar, and two Learners, ksvm and kknn, that are both tuned.

As inner resampling strategies we use holdout for ksvm and subsampling with 3 iterations for kknn. As outer resampling strategies we take holdout for the iris and bootstrap with 2 iterations for the sonar data (sonar.task). We consider the accuracy (acc), which is used as tuning criterion, and also calculate the balanced error rate (ber).

```
### List of learning tasks
tasks = list(iris.task, sonar.task)

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

### Tune k-nearest neighbor in inner resampling loop
```

```
ps = makeParamSet(makeDiscreteParam("k", 3:5))
ctrl = makeTuneControlGrid()
inner = makeResampleDesc("Subsample", iters = 3)
lrn2 = makeTuneWrapper("classif.kknn", resampling = inner, par.set =
    ps, control = ctrl,
  show.info = FALSE)
#> Loading required package: kknn
#>
#> Attaching package: 'kknn'
#> The following object is masked from 'package:caret':
#>
#>     contr.dummy

### Learners
lrns = list(lrn1, lrn2)

### Outer resampling loop
outer = list(makeResampleDesc("Holdout"),
    makeResampleDesc("Bootstrap", iters = 2))
res = benchmark(lrns, tasks, outer, measures = list(acc, ber),
    show.info = FALSE)
res
#>         task.id         learner.id acc.test.mean ber.test.mean
#> 1  iris_example classif.ksvm.tuned     0.9400000    0.05882353
#> 2  iris_example classif.kknn.tuned     0.9200000    0.08683473
#> 3 Sonar_example classif.ksvm.tuned     0.5289307    0.50000000
#> 4 Sonar_example classif.kknn.tuned     0.8077080    0.19549714
```

The print method for the BenchmarkResult shows the aggregated performances from the outer resampling loop.

As you might recall, mlr offers several accessor function to extract information from the benchmark result. These are listed on the help page of BenchmarkResult and many examples are shown on the tutorial page about benchmark experiments.

The performance values in individual outer resampling runs can be obtained by getBMRPerformances. Note that, since we used different outer resampling strategies for the two tasks, the number of rows per task differ.

```
getBMRPerformances(res, as.df = TRUE)
#>         task.id         learner.id iter       acc        ber
#> 1  iris_example classif.ksvm.tuned    1 0.9400000 0.05882353
#> 2  iris_example classif.kknn.tuned    1 0.9200000 0.08683473
#> 3 Sonar_example classif.ksvm.tuned    1 0.5373134 0.50000000
#> 4 Sonar_example classif.ksvm.tuned    2 0.5205479 0.50000000
#> 5 Sonar_example classif.kknn.tuned    1 0.8208955 0.18234767
#> 6 Sonar_example classif.kknn.tuned    2 0.7945205 0.20864662
```

The results from the parameter tuning can be obtained through function getBMRTuneResults.

```
getBMRTuneResults(res)
#> $iris_example
#> $iris_example$classif.ksvm.tuned
#> $iris_example$classif.ksvm.tuned[[1]]
#> Tune result:
#> Op. pars: C=0.5; sigma=0.5
#> mmce.test.mean=0.0588235
#>
#>
#> $iris_example$classif.kknn.tuned
#> $iris_example$classif.kknn.tuned[[1]]
#> Tune result:
#> Op. pars: k=3
#> mmce.test.mean=0.0490196
#>
#>
#>
#> $Sonar_example
#> $Sonar_example$classif.ksvm.tuned
#> $Sonar_example$classif.ksvm.tuned[[1]]
#> Tune result:
#> Op. pars: C=1; sigma=2
#> mmce.test.mean=0.3428571
#>
#> $Sonar_example$classif.ksvm.tuned[[2]]
#> Tune result:
#> Op. pars: C=2; sigma=0.5
#> mmce.test.mean=0.2000000
#>
#>
#> $Sonar_example$classif.kknn.tuned
#> $Sonar_example$classif.kknn.tuned[[1]]
#> Tune result:
#> Op. pars: k=4
#> mmce.test.mean=0.1095238
#>
#> $Sonar_example$classif.kknn.tuned[[2]]
#> Tune result:
#> Op. pars: k=3
#> mmce.test.mean=0.0666667
```

As for several other accessor functions a clearer representation as data.frame can be achieved by setting `as.df = TRUE`.

```
getBMRTuneResults(res, as.df = TRUE)
#>         task.id         learner.id iter   C sigma mmce.test.mean
    k
#> 1  iris_example classif.ksvm.tuned    1 0.5   0.5     0.05882353
    NA
#> 2  iris_example classif.kknn.tuned    1  NA    NA     0.04901961
    3
#> 3 Sonar_example classif.ksvm.tuned    1 1.0   2.0     0.34285714
    NA
#> 4 Sonar_example classif.ksvm.tuned    2 2.0   0.5     0.20000000
    NA
#> 5 Sonar_example classif.kknn.tuned    1  NA    NA     0.10952381
    4
#> 6 Sonar_example classif.kknn.tuned    2  NA    NA     0.06666667
    3
```

It is also possible to extract the tuning results for individual tasks and learners and, as shown in earlier examples, inspect the optimization path.

```
tune.res = getBMRTuneResults(res, task.ids = "Sonar_example",
    learner.ids = "classif.ksvm.tuned",
  as.df = TRUE)
tune.res
#>         task.id         learner.id iter C sigma mmce.test.mean
#> 1 Sonar_example classif.ksvm.tuned    1 1   2.0      0.3428571
#> 2 Sonar_example classif.ksvm.tuned    2 2   0.5      0.2000000

getNestedTuneResultsOptPathDf(res$results[["Sonar_example"]][["classif.ksvm.tuned"]])
```

#### 3.7.3.2 Example 2: One task, two learners, feature selection

Let's see how we can do feature selection in a benchmark experiment:

```
### Feature selection in inner resampling loop
ctrl = makeFeatSelControlSequential(method = "sfs")
inner = makeResampleDesc("Subsample", iters = 2)
lrn = makeFeatSelWrapper("regr.lm", resampling = inner, control =
    ctrl, show.info = FALSE)

### Learners
lrns = list("regr.rpart", lrn)

### Outer resampling loop
outer = makeResampleDesc("Subsample", iters = 2)
res = benchmark(tasks = bh.task, learners = lrns, resampling =
    outer, show.info = FALSE)
```

```
res
#>                  task.id      learner.id mse.test.mean
#> 1 BostonHousing-example      regr.rpart      25.86232
#> 2 BostonHousing-example regr.lm.featsel      25.07465
```

The selected features can be extracted by function getBMRFeatSelResults. By default, a nested list, with the first level indicating the task and the second level indicating the learner, is returned. If only a single learner or, as in our case, a single task is considered, setting `drop = TRUE` simplifies the result to a flat list.

```
getBMRFeatSelResults(res)
#> $`BostonHousing-example`
#> $`BostonHousing-example`$regr.rpart
#> NULL
#>
#> $`BostonHousing-example`$regr.lm.featsel
#> $`BostonHousing-example`$regr.lm.featsel[[1]]
#> FeatSel result:
#> Features (8): crim, zn, chas, nox, rm, dis, ptratio, lstat
#> mse.test.mean=26.7257383
#>
#> $`BostonHousing-example`$regr.lm.featsel[[2]]
#> FeatSel result:
#> Features (10): crim, zn, nox, rm, dis, rad, tax, ptratio, b, lstat
#> mse.test.mean=24.2874464
getBMRFeatSelResults(res, drop = TRUE)
#> $regr.rpart
#> NULL
#>
#> $regr.lm.featsel
#> $regr.lm.featsel[[1]]
#> FeatSel result:
#> Features (8): crim, zn, chas, nox, rm, dis, ptratio, lstat
#> mse.test.mean=26.7257383
#>
#> $regr.lm.featsel[[2]]
#> FeatSel result:
#> Features (10): crim, zn, nox, rm, dis, rad, tax, ptratio, b, lstat
#> mse.test.mean=24.2874464
```

You can access results for individual learners and tasks and inspect them further.

```
feats = getBMRFeatSelResults(res, learner.id = "regr.lm.featsel",
    drop = TRUE)

### Selected features in the first outer resampling iteration
```

```
feats[[1]]$x
#> [1] "crim"    "zn"      "chas"    "nox"     "rm"      "dis"
    "ptratio"
#> [8] "lstat"

### Resampled performance of the selected feature subset on the
    first inner training set
feats[[1]]$y
#> mse.test.mean
#>      26.72574
```

As for tuning, you can extract the optimization paths. The resulting data.frames contain, among others, binary columns for all features, indicating if they were included in the linear regression model, and the corresponding performances. analyzeFeatSelResult gives a clearer overview.

```
opt.paths = lapply(feats, function(x) as.data.frame(x$opt.path))
head(opt.paths[[1]])
#>   crim zn indus chas nox rm age dis rad tax ptratio b lstat
    mse.test.mean
#> 1    0  0     0    0   0  0   0   0   0   0       0 0     0
    90.16159
#> 2    1  0     0    0   0  0   0   0   0   0       0 0     0
    82.85880
#> 3    0  1     0    0   0  0   0   0   0   0       0 0     0
    79.55202
#> 4    0  0     1    0   0  0   0   0   0   0       0 0     0
    70.02071
#> 5    0  0     0    1   0  0   0   0   0   0       0 0     0
    86.93409
#> 6    0  0     0    0   1  0   0   0   0   0       0 0     0
    76.32457
#>   dob eol error.message exec.time
#> 1   1   2          <NA>     0.015
#> 2   2   2          <NA>     0.025
#> 3   2   2          <NA>     0.026
#> 4   2   2          <NA>     0.027
#> 5   2   2          <NA>     0.028
#> 6   2   2          <NA>     0.025

analyzeFeatSelResult(feats[[1]])
#> Features         : 8
#> Performance      : mse.test.mean=26.7257383
#> crim, zn, chas, nox, rm, dis, ptratio, lstat
#>
#> Path to optimum:
```

```
#> - Features:    0  Init   :                       Perf = 90.162
    Diff: NA  *
#> - Features:    1  Add    : lstat                  Perf = 42.646
    Diff: 47.515  *
#> - Features:    2  Add    : ptratio                Perf = 34.52
    Diff: 8.1263  *
#> - Features:    3  Add    : rm                     Perf = 30.454
    Diff: 4.066  *
#> - Features:    4  Add    : dis                    Perf = 29.405
    Diff: 1.0495  *
#> - Features:    5  Add    : nox                    Perf = 28.059
    Diff: 1.3454  *
#> - Features:    6  Add    : chas                   Perf = 27.334
    Diff: 0.72499  *
#> - Features:    7  Add    : zn                     Perf = 26.901
    Diff: 0.43296  *
#> - Features:    8  Add    : crim                   Perf = 26.726
    Diff: 0.17558  *
#>
#> Stopped, because no improving feature was found.
```

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

Here is a minimal example for feature filtering with tuning of the feature subset size.

```
### Feature filtering with tuning in the inner resampling loop
lrn = makeFilterWrapper(learner = "regr.lm", fw.method =
    "chi.squared")
ps = makeParamSet(makeDiscreteParam("fw.abs", values =
    seq_len(getTaskNFeats(bh.task))))
ctrl = makeTuneControlGrid()
inner = makeResampleDesc("CV", iter = 2)
lrn = makeTuneWrapper(lrn, resampling = inner, par.set = ps, control
    = ctrl,
  show.info = FALSE)

### Learners
lrns = list("regr.rpart", lrn)

### Outer resampling loop
outer = makeResampleDesc("Subsample", iter = 3)
res = benchmark(tasks = bh.task, learners = lrns, resampling =
    outer, show.info = FALSE)
```

```
res
#>                 task.id             learner.id mse.test.mean
#> 1 BostonHousing-example             regr.rpart      22.11687
#> 2 BostonHousing-example regr.lm.filtered.tuned      23.76666
```

```
### Performances on individual outer test data sets
getBMRPerformances(res, as.df = TRUE)
#>                 task.id             learner.id iter      mse
#> 1 BostonHousing-example             regr.rpart    1 23.55486
#> 2 BostonHousing-example             regr.rpart    2 20.03453
#> 3 BostonHousing-example             regr.rpart    3 22.76121
#> 4 BostonHousing-example regr.lm.filtered.tuned    1 27.51086
#> 5 BostonHousing-example regr.lm.filtered.tuned    2 24.87820
#> 6 BostonHousing-example regr.lm.filtered.tuned    3 18.91091
```

## 3.8 Cost-Sensitive Classification

In *regular classification* the aim is to minimize the misclassification rate and thus all types of misclassification errors are deemed equally severe. A more general setting is *cost-sensitive classification* where the costs caused by different kinds of errors are not assumed to be equal and the objective is to minimize the expected costs.

In case of *class-dependent costs* the costs depend on the true and predicted class label. The costs $c(k, l)$ for predicting class $k$ if the true label is $l$ are usually organized into a $K \times K$ cost matrix where $K$ is the number of classes. Naturally, it is assumed that the cost of predicting the correct class label $y$ is minimal (that is $c(y, y) \leq c(k, y)$ for all $k = 1, \ldots, K$).

A further generalization of this scenario are *example-dependent misclassification costs* where each example $(x, y)$ is coupled with an individual cost vector of length $K$. Its $k$-th component expresses the cost of assigning $x$ to class $k$. A real-world example is fraud detection where the costs do not only depend on the true and predicted status fraud/non-fraud, but also on the amount of money involved in each case. Naturally, the cost of predicting the true class label $y$ is assumed to be minimum. The true class labels are redundant information, as they can be easily inferred from the cost vectors. Moreover, given the cost vector, the expected costs do not depend on the true class label $y$. The classification problem is therefore completely defined by the feature values $x$ and the corresponding cost vectors.

In the following we show ways to handle cost-sensitive classification problems in mlr. Some of the functionality is currently experimental, and there may be changes in the future.

### 3.8.1 Class-dependent misclassification costs

There are some classification methods that can accomodate misclassification costs directly. One example is rpart.

Alternatively, we can use cost-insensitive methods and manipulate the predictions or the training data in order to take misclassification costs into account. mlr supports *thresholding* and *rebalancing*.

1. **Thresholding**: The thresholds used to turn posterior probabilities into class labels are chosen such that the costs are minimized. This requires a Learner that can predict posterior probabilities. During training the costs are not taken into account.
2. **Rebalancing**: The idea is to change the proportion of the classes in the training data set in order to account for costs during training, either by *weighting* or by *sampling*. Rebalancing does not require that the Learner can predict probabilities.
    i. For *weighting* we need a Learner that supports class weights or observation weights.
    ii. If the Learner cannot deal with weights the proportion of classes can be changed by *over-* and *undersampling*.

We start with binary classification problems and afterwards deal with multi-class problems.

#### 3.8.1.1 Binary classification problems

The positive and negative classes are labeled 1 and $-1$, respectively, and we consider the following cost matrix where the rows indicate true classes and the columns predicted classes:

| true/pred. | $+1$ | $-1$ |
|---|---|---|
| $+1$ | $c(+1,+1)$ | $c(-1,+1)$ |
| $-1$ | $c(+1,-1)$ | $c(-1,-1)$ |

Often, the diagonal entries are zero or the cost matrix is rescaled to achieve zeros in the diagonal (see for example O'Brien et al, 2008).

A well-known cost-sensitive classification problem is posed by the German Credit data set (see also the UCI Machine Learning Repository). The corresponding cost matrix (though Elkan (2001) argues that this matrix is economically unreasonable) is given as:

| true/pred. | Bad | Good |
|---|---|---|
| Bad | 0 | 5 |
| Good | 1 | 0 |

As in the table above, the rows indicate true and the columns predicted classes.

In case of class-dependent costs it is sufficient to generate an ordinary ClassifTask. A CostSensTask is only needed if the costs are example-dependent. In the **R** code below we create the ClassifTask, remove two constant features from the data set and generate the cost matrix. Per default, Bad is the positive class.

```
data(GermanCredit, package = "caret")
credit.task = makeClassifTask(data = GermanCredit, target = "Class")
credit.task = removeConstantFeatures(credit.task)
#> Removing 2 columns: Purpose.Vacation,Personal.Female.Single

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

costs = matrix(c(0, 1, 5, 0), 2)
colnames(costs) = rownames(costs) = getTaskClassLevels(credit.task)
costs
#>      Bad Good
#> Bad    0    5
#> Good   1    0
```

#### 3.8.1.1.1 1. Thresholding

We start by fitting a logistic regression model to the German credit data set and predict posterior probabilities.

```
### Train and predict posterior probabilities
```

```
lrn = makeLearner("classif.multinom", predict.type = "prob", trace =
    FALSE)
mod = train(lrn, credit.task)
pred = predict(mod, task = credit.task)
pred
#> Prediction: 1000 observations
#> predict.type: prob
#> threshold: Bad=0.50,Good=0.50
#> time: 0.01
#>   id truth   prob.Bad prob.Good response
#> 1  1  Good 0.03525092 0.9647491     Good
#> 2  2   Bad 0.63222363 0.3677764      Bad
#> 3  3  Good 0.02807414 0.9719259     Good
#> 4  4  Good 0.25182703 0.7481730     Good
#> 5  5   Bad 0.75193275 0.2480673      Bad
#> 6  6  Good 0.26230149 0.7376985     Good
#> ... (#rows: 1000, #cols: 5)
```

The default thresholds for both classes are 0.5. But according to the cost matrix we should predict class Good only if we are very sure that Good is indeed the correct label. Therefore we should increase the threshold for class Good and decrease the threshold for class Bad.

i. Theoretical thresholding

The theoretical threshold for the *positive* class can be calculated from the cost matrix as

$$t^* = \frac{c(+1,-1) - c(-1,-1)}{c(+1,-1) - c(+1,+1) + c(-1,+1) - c(-1,-1)}.$$

For more details see Elkan (2001).

Below the theoretical threshold for the German credit example is calculated and used to predict class labels. Since the diagonal of the cost matrix is zero the formula given above simplifies accordingly.

```
### Calculate the theoretical threshold for the positive class
th = costs[2,1]/(costs[2,1] + costs[1,2])
th
#> [1] 0.1666667
```

As you may recall you can change thresholds in mlr either before training by using the `predict.threshold` option of makeLearner or after prediction by calling setThreshold on the Prediction object.

As we already have a prediction we use the setThreshold function. It returns an altered Prediction object with class predictions for the theoretical threshold.

```
### Predict class labels according to the theoretical threshold
pred.th = setThreshold(pred, th)
pred.th
#> Prediction: 1000 observations
#> predict.type: prob
#> threshold: Bad=0.17,Good=0.83
#> time: 0.01
#>   id truth   prob.Bad prob.Good response
#> 1  1  Good 0.03525092 0.9647491     Good
#> 2  2   Bad 0.63222363 0.3677764      Bad
#> 3  3  Good 0.02807414 0.9719259     Good
#> 4  4  Good 0.25182703 0.7481730      Bad
#> 5  5   Bad 0.75193275 0.2480673      Bad
#> 6  6  Good 0.26230149 0.7376985      Bad
#> ... (#rows: 1000, #cols: 5)
```

In order to calculate the average costs over the entire data set we first need to create a new performance Measure. This can be done through function makeCostMeasure. It is expected that the rows of the cost matrix indicate true and the columns predicted class labels.

```
credit.costs = makeCostMeasure(id = "credit.costs", name = "Credit
    costs", costs = costs,
  best = 0, worst = 5)
credit.costs
#> Name: Credit costs
#> Performance measure: credit.costs
#> Properties:
    classif,classif.multi,req.pred,req.truth,predtype.response,predtype.prob
#> Minimize: TRUE
#> Best: 0; Worst: 5
#> Aggregated by: test.mean
#> Arguments: <unnamed>=<matrix>, <unnamed>=<function>
#> Note:
```

Then the average costs can be computed by function performance. Below we compare the average costs and the error rate (mmce) of the learning algorithm with both default thresholds 0.5 and theoretical thresholds.

```
### Performance with default thresholds 0.5
performance(pred, measures = list(credit.costs, mmce))
#> credit.costs         mmce
#>        0.774        0.214

### Performance with theoretical thresholds
performance(pred.th, measures = list(credit.costs, mmce))
#> credit.costs         mmce
```

```
#>        0.478        0.346
```

These performance values may be overly optimistic as we used the same data set for training and prediction, and resampling strategies should be preferred. In the **R** code below we make use of the `predict.threshold` argument of makeLearner to set the threshold before doing a 3-fold cross-validation on the credit.task. Note that we create a ResampleInstance (`rin`) that is used throughout the next several code chunks to get comparable performance values.

```
### Cross-validated performance with theoretical thresholds
rin = makeResampleInstance("CV", iters = 3, task = credit.task)
lrn = makeLearner("classif.multinom", predict.type = "prob",
    predict.threshold = th, trace = FALSE)
r = resample(lrn, credit.task, resampling = rin, measures =
    list(credit.costs, mmce), show.info = FALSE)
r
#> Resample Result
#> Task: GermanCredit
#> Learner: classif.multinom
#> Aggr perf:
    credit.costs.test.mean=0.5831280,mmce.test.mean=0.3630397
#> Runtime: 0.206106
```

If we are also interested in the cross-validated performance for the default threshold values we can call setThreshold on the resample prediction `r$pred`.

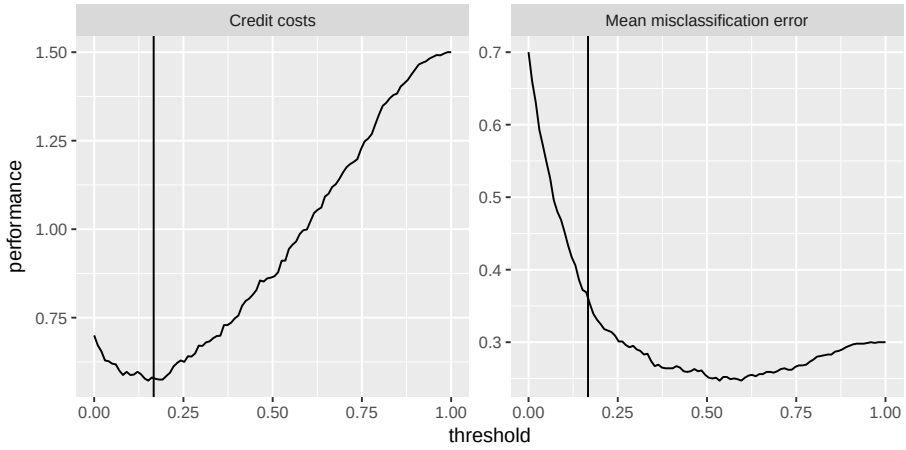
```
### Cross-validated performance with default thresholds
performance(setThreshold(r$pred, 0.5), measures = list(credit.costs,
    mmce))
#> credit.costs         mmce
#>    0.8600427    0.2520215
```

Theoretical thresholding is only reliable if the predicted posterior probabilities are correct. If there is bias the thresholds have to be shifted accordingly.

Useful in this regard is function plotThreshVsPerf that you can use to plot the average costs as well as any other performance measure versus possible threshold values for the positive class in $[0, 1]$. The underlying data is generated by generateThreshVsPerfData.

The following plots show the cross-validated costs and error rate (mmce). The theoretical threshold `th` calculated above is indicated by the vertical line. As you can see from the left-hand plot the theoretical threshold seems a bit large.

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




ii. Empirical thresholding

The idea of *empirical thresholding* (see Sheng and Ling, 2006) is to select cost-optimal threshold values for a given learning method based on the training data. In contrast to *theoretical thresholding* it suffices if the estimated posterior probabilities are order-correct.

In order to determine optimal threshold values you can use mlr's function tuneThreshold. As tuning the threshold on the complete training data set can lead to overfitting, you should use resampling strategies. Below we perform 3-fold cross-validation and use tuneThreshold to calculate threshold values with lowest average costs over the 3 test data sets.

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

### 3-fold cross-validation
r = resample(lrn, credit.task, resampling = rin, measures =
    list(credit.costs, mmce), show.info = FALSE)
r
#> Resample Result
#> Task: GermanCredit
#> Learner: classif.multinom
#> Aggr perf:
    credit.costs.test.mean=0.8600427,mmce.test.mean=0.2520215
#> Runtime: 0.203061

### Tune the threshold based on the predicted probabilities on the 3
    test data sets
tune.res = tuneThreshold(pred = r$pred, measure = credit.costs)
tune.res
#> $th
```

```
#> [1] 0.1874551
#>
#> $perf
#> credit.costs
#>     0.569108
```

tuneThreshold returns the optimal threshold value for the positive class and the corresponding performance. As expected the tuned threshold is smaller than the theoretical threshold.

#### 3.8.1.1.2 2. Rebalancing

In order to minimize the average costs, observations from the less costly class should be given higher importance during training. This can be achieved by *weighting* the classes, provided that the learner under consideration has a 'class weights' or an 'observation weights' argument. To find out which learning methods support either type of weights have a look at the list of integrated learners in the Appendix or use listLearners.

```
### Learners that accept observation weights
listLearners("classif", properties = "weights")[c("class",
    "package")]
#>                class      package
#> 1   classif.binomial        stats
#> 2 classif.blackboost mboost,party
#> 3        classif.C50          C50
#> 4    classif.cforest        party
#> 5      classif.ctree        party
#> 6   classif.cvglmnet       glmnet
#> ... (#rows: 25, #cols: 2)

### Learners that can deal with class weights
listLearners("classif", properties = "class.weights")[c("class",
    "package")]
#>                        class   package
#> 1               classif.ksvm   kernlab
#> 2  classif.LiblineaRL1L2SVC LiblineaR
#> 3 classif.LiblineaRL1LogReg LiblineaR
#> 4  classif.LiblineaRL2L1SVC LiblineaR
#> 5 classif.LiblineaRL2LogReg LiblineaR
#> 6    classif.LiblineaRL2SVC LiblineaR
#> ... (#rows: 9, #cols: 2)
```

Alternatively, *over- and undersampling* techniques can be used.

i. Weighting

Just as *theoretical thresholds*, *theoretical weights* can be calculated from the cost matrix. If $t$ indicates the target threshold and $t_0$ the original threshold for the positive class the proportion of observations in the positive class has to be multiplied by

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

Alternatively, the proportion of observations in the negative class can be multiplied by the inverse. A proof is given by Elkan (2001).

In most cases, the original threshold is $t_0 = 0.5$ and thus the second factor vanishes. If additionally the target threshold $t$ equals the theoretical threshold $t^*$ the proportion of observations in the positive class has to be multiplied by

$$\frac{1-t^*}{t^*} = \frac{c(-1,+1) - c(+1,+1)}{c(+1,-1) - c(-1,-1)}.$$

For the credit example the theoretical threshold corresponds to a weight of 5 for the positive class.

```
### Weight for positive class corresponding to theoretical treshold
w = (1 - th)/th
w
#> [1] 5
```

A unified and convenient way to assign class weights to a Learner (and tune them) is provided by function makeWeightedClassesWrapper. The class weights are specified using argument `wcw.weight`. For learners that support observation weights a suitable weight vector is then generated internally during training or resampling. If the learner can deal with class weights, the weights are basically passed on to the appropriate learner parameter. The advantage of using the wrapper in this case is the unified way to specify the class weights.

Below is an example using learner `"classif.multinom"` (multinom from package nnet) which accepts observation weights. For binary classification problems it is sufficient to specify the weight `w` for the positive class. The negative class then automatically receives weight 1.

```
### Weighted learner
lrn = makeLearner("classif.multinom", trace = FALSE)
lrn = makeWeightedClassesWrapper(lrn, wcw.weight = w)
lrn
#> Learner weightedclasses.classif.multinom from package nnet
#> Type: classif
#> Name: ; Short name:
#> Class: WeightedClassesWrapper
#> Properties: twoclass,multiclass,numerics,factors,prob
#> Predict-Type: response
#> Hyperparameters: trace=FALSE,wcw.weight=5
```

```
r = resample(lrn, credit.task, rin, measures = list(credit.costs,
    mmce), show.info = FALSE)
r
#> Resample Result
#> Task: GermanCredit
#> Learner: weightedclasses.classif.multinom
#> Aggr perf:
    credit.costs.test.mean=0.5851031,mmce.test.mean=0.3530387
#> Runtime: 0.251485
```

For classification methods like `"classif.ksvm"` (the support vector machine ksvm in package kernlab) that support class weights you can pass them directly.

```
lrn = makeLearner("classif.ksvm", class.weights = c(Bad = w, Good =
    1))
```

Or, more conveniently, you can again use makeWeightedClassesWrapper.

```
lrn = makeWeightedClassesWrapper("classif.ksvm", wcw.weight = w)
r = resample(lrn, credit.task, rin, measures = list(credit.costs,
    mmce), show.info = FALSE)
r
#> Resample Result
#> Task: GermanCredit
#> Learner: weightedclasses.classif.ksvm
#> Aggr perf:
    credit.costs.test.mean=0.6520802,mmce.test.mean=0.3360276
#> Runtime: 0.393345
```

Just like the theoretical threshold, the theoretical weights may not always be suitable, therefore you can tune the weight for the positive class as shown in the following example. Calculating the theoretical weight beforehand may help to narrow down the search interval.

```
lrn = makeLearner("classif.multinom", trace = FALSE)
lrn = makeWeightedClassesWrapper(lrn)
ps = makeParamSet(makeDiscreteParam("wcw.weight", seq(4, 12, 0.5)))
ctrl = makeTuneControlGrid()
tune.res = tuneParams(lrn, credit.task, resampling = rin, par.set =
    ps,
  measures = list(credit.costs, mmce), control = ctrl, show.info =
      FALSE)
tune.res
#> Tune result:
#> Op. pars: wcw.weight=7.5
#> credit.costs.test.mean=0.5590681,mmce.test.mean=0.3950118
```

```
as.data.frame(tune.res$opt.path)[1:3]
#>    wcw.weight credit.costs.test.mean mmce.test.mean
#> 1           4              0.5961231      0.3280466
#> 2         4.5              0.5960931      0.3440327
#> 3           5              0.5851031      0.3530387
#> 4         5.5              0.5711010      0.3590327
#> 5           6              0.5761030      0.3720307
#> 6         6.5              0.5851091      0.3850287
#> 7           7              0.5830951      0.3950148
#> 8         7.5              0.5590681      0.3950118
#> 9           8              0.5700671      0.4060108
#> 10        8.5              0.5760671      0.4120108
#> 11          9              0.5780631      0.4180108
#> 12        9.5              0.5790581      0.4230098
#> 13         10              0.5850641      0.4290158
#> 14       10.5              0.5890771      0.4370209
#> 15         11              0.5770891      0.4410249
#> 16       11.5              0.5780901      0.4420259
#> 17         12              0.5770801      0.4450199
```

ii. Over- and undersampling

If the Learner supports neither observation nor class weights the proportions of the classes in the training data can be changed by over- or undersampling.

In the GermanCredit data set the positive class Bad should receive a theoretical weight of `w = (1 - th)/th = 5`. This can be achieved by oversampling class Bad with a `rate` of 5 or by undersampling class Good with a `rate` of 1/5 (using functions oversample or undersample).

```
credit.task.over = oversample(credit.task, rate = w, cl = "Bad")
lrn = makeLearner("classif.multinom", trace = FALSE)
mod = train(lrn, credit.task.over)
pred = predict(mod, task = credit.task)
performance(pred, measures = list(credit.costs, mmce))
#> credit.costs         mmce
#>        0.441        0.325
```

Note that in the above example the learner was trained on the oversampled task `credit.task.over`. In order to get the training performance on the original task predictions were calculated for `credit.task`.

We usually prefer resampled performance values, but simply calling resample on the oversampled task does not work since predictions have to be based on the original task. The solution is to create a wrapped Learner via function makeOversampleWrapper. Internally, oversample is called before training, but predictions are done on the original data.

```
lrn = makeLearner("classif.multinom", trace = FALSE)
lrn = makeOversampleWrapper(lrn, osw.rate = w, osw.cl = "Bad")
lrn
#> Learner classif.multinom.oversampled from package mlr,nnet
#> Type: classif
#> Name: ; Short name:
#> Class: OversampleWrapper
#> Properties: numerics,factors,weights,prob,twoclass,multiclass
#> Predict-Type: response
#> Hyperparameters: trace=FALSE,osw.rate=5,osw.cl=Bad

r = resample(lrn, credit.task, rin, measures = list(credit.costs,
    mmce), show.info = FALSE)
r
#> Resample Result
#> Task: GermanCredit
#> Learner: classif.multinom.oversampled
#> Aggr perf:
    credit.costs.test.mean=0.5681190,mmce.test.mean=0.3360426
#> Runtime: 0.422928
```

Of course, we can also tune the oversampling rate. For this purpose we again have to create an OversampleWrapper. Optimal values for parameter `osw.rate` can be obtained using function tuneParams.

```
lrn = makeLearner("classif.multinom", trace = FALSE)
lrn = makeOversampleWrapper(lrn, osw.cl = "Bad")
ps = makeParamSet(makeDiscreteParam("osw.rate", seq(3, 7, 0.25)))
ctrl = makeTuneControlGrid()
tune.res = tuneParams(lrn, credit.task, rin, par.set = ps, measures
    = list(credit.costs, mmce),
  control = ctrl, show.info = FALSE)
tune.res
#> Tune result:
#> Op. pars: osw.rate=6.75
#> credit.costs.test.mean=0.5700821,mmce.test.mean=0.3860178
```

#### 3.8.1.2 Multi-class problems

We consider the waveform data set from package mlbench and add an artificial cost matrix:

| true/pred. | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 0 | 30 | 80 |
| 2 | 5 | 0 | 4 |
| 3 | 10 | 8 | 0 |

We start by creating the Task, the cost matrix and the corresponding performance measure.

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

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

### Performance measure
wf.costs = makeCostMeasure(id = "wf.costs", name = "Waveform costs",
    costs = costs,
  best = 0, worst = 10)
```

In the multi-class case, both, *thresholding* and *rebalancing* correspond to cost matrices of a certain structure where $c(k, l) = c(l)$ for $k$, $l = 1, \ldots, K$, $k \neq l$. This condition means that the cost of misclassifying an observation is independent of the predicted class label (see Domingos, 1999). Given a cost matrix of this type, theoretical thresholds and weights can be derived in a similar manner as in the binary case. Obviously, the cost matrix given above does not have this special structure.

#### 3.8.1.2.1 1. Thresholding

Given a vector of positive threshold values as long as the number of classes $K$, the predicted probabilities for all classes are adjusted by dividing them by the corresponding threshold value. Then the class with the highest adjusted probability is predicted. This way, as in the binary case, classes with a low threshold are preferred to classes with a larger threshold.

Again this can be done by function setThreshold as shown in the following example (or alternatively by the `predict.threshold` option of makeLearner). Note that the threshold vector needs to have names that correspond to the class labels.

```
lrn = makeLearner("classif.rpart", predict.type = "prob")
rin = makeResampleInstance("CV", iters = 3, task = wf.task)
r = resample(lrn, wf.task, rin, measures = list(wf.costs, mmce),
    show.info = FALSE)
r
#> Resample Result
#> Task: waveform
#> Learner: classif.rpart
#> Aggr perf: wf.costs.test.mean=7.9568814,mmce.test.mean=0.3180386
```

```
#> Runtime: 0.081192

### Calculate thresholds as 1/(average costs of true classes)
th = 2/rowSums(costs)
names(th) = getTaskClassLevels(wf.task)
th
#>          1          2          3
#> 0.01818182 0.22222222 0.11111111

pred.th = setThreshold(r$pred, threshold = th)
performance(pred.th, measures = list(wf.costs, mmce))
#>  wf.costs      mmce
#> 6.1248707 0.4699998
```

The threshold vector `th` in the above example is chosen according to the average costs of the true classes 55, 4.5 and 9. More exactly, `th` corresponds to an artificial cost matrix of the structure mentioned above with off-diagonal elements $c(2,1) = c(3,1) = 55$, $c(1,2) = c(3,2) = 4.5$ and $c(1,3) = c(2,3) = 9$. This threshold vector may be not optimal but leads to smaller total costs on the data set than the default.

ii. Empirical thresholding

As in the binary case it is possible to tune the threshold vector using function tuneThreshold. Since the scaling of the threshold vector does not change the predicted class labels tuneThreshold returns threshold values that lie in [0,1] and sum to unity.

```
tune.res = tuneThreshold(pred = r$pred, measure = wf.costs)
tune.res
#> $th
#>          1          2          3
#> 0.03481266 0.32529865 0.63988869
#>
#> $perf
#> [1] 4.711613
```

For comparison we show the standardized version of the theoretically motivated threshold vector chosen above.

```
th/sum(th)
#>          1          2          3
#> 0.05172414 0.63218391 0.31609195
```

#### 3.8.1.2.2 2. Rebalancing

i. Weighting

In the multi-class case you have to pass a vector of weights as long as the number of classes $K$ to function makeWeightedClassesWrapper. The weight vector can be tuned using function tuneParams.

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

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

tune.res = tuneParams(lrn, wf.task, resampling = rin, par.set = ps,
  measures = list(wf.costs, mmce), control = ctrl, show.info = FALSE)
tune.res
#> Tune result:
#> Op. pars: wcw.weight=0.673,0.0513,0.114
#> wf.costs.test.mean=3.2633889,mmce.test.mean=0.2179496
```

### 3.8.2 Example-dependent misclassification costs

In case of example-dependent costs we have to create a special Task via function makeCostSensTask. For this purpose the feature values $x$ and an $n \times K$ `cost` matrix that contains the cost vectors for all $n$ examples in the data set are required.

We use the iris data and generate an artificial cost matrix (see Beygelzimer et al., 2005).

```
df = iris
cost = matrix(runif(150 * 3, 0, 2000), 150) * (1 -
    diag(3))[df$Species,] + runif(150, 0, 10)
colnames(cost) = levels(iris$Species)
rownames(cost) = rownames(iris)
df$Species = NULL

costsens.task = makeCostSensTask(id = "iris", data = df, cost = cost)
costsens.task
#> Supervised task: iris
#> Type: costsens
#> Observations: 150
#> Features:
#>    numerics     factors     ordered functionals
#>           4           0           0           0
#> Missings: FALSE
#> Has blocking: FALSE
#> Is spatial: FALSE
```

```
#> Classes: 3
#> setosa, versicolor, virginica
```

mlr provides several wrappers to turn regular classification or regression methods into Learners that can deal with example-dependent costs.

- makeCostSensClassifWrapper (wraps a classification Learner): This is a naive approach where the costs are coerced into class labels by choosing the class label with minimum cost for each example. Then a regular classification method is used.
- makeCostSensRegrWrapper (wraps a regression Learner): An individual regression model is fitted for the costs of each class. In the prediction step first the costs are predicted for all classes and then the class with the lowest predicted costs is selected.
- makeCostSensWeightedPairsWrapper (wraps a classification Learner): This is also known as *cost-sensitive one-vs-one* (CS-OVO) and the most sophisticated of the currently supported methods. For each pair of classes, a binary classifier is fitted. For each observation the class label is defined as the element of the pair with minimal costs. During fitting, the observations are weighted with the absolute difference in costs. Prediction is performed by simple voting.

In the following example we use the third method. We create the wrapped Learner and train it on the CostSensTask defined above.

```
lrn = makeLearner("classif.multinom", trace = FALSE)
lrn = makeCostSensWeightedPairsWrapper(lrn)
lrn
#> Learner costsens.classif.multinom from package nnet
#> Type: costsens
#> Name: ; Short name:
#> Class: CostSensWeightedPairsWrapper
#> Properties: twoclass,multiclass,numerics,factors
#> Predict-Type: response
#> Hyperparameters: trace=FALSE

mod = train(lrn, costsens.task)
mod
#> Model for learner.id=costsens.classif.multinom;
    learner.class=CostSensWeightedPairsWrapper
#> Trained on: task.id = iris; obs = 150; features = 4
#> Hyperparameters: trace=FALSE
```

The models corresponding to the individual pairs can be accessed by function getLearnerModel.

```
getLearnerModel(mod)
#> [[1]]
#> Model for learner.id=classif.multinom;
    learner.class=classif.multinom
#> Trained on: task.id = feats; obs = 150; features = 4
#> Hyperparameters: trace=FALSE
#>
#> [[2]]
#> Model for learner.id=classif.multinom;
    learner.class=classif.multinom
#> Trained on: task.id = feats; obs = 150; features = 4
#> Hyperparameters: trace=FALSE
#>
#> [[3]]
#> Model for learner.id=classif.multinom;
    learner.class=classif.multinom
#> Trained on: task.id = feats; obs = 150; features = 4
#> Hyperparameters: trace=FALSE
```

mlr provides some performance measures for example-specific cost-sensitive classification. In the following example we calculate the mean costs of the predicted class labels (meancosts) and the misclassification penalty (mcp). The latter measure is the average difference between the costs caused by the predicted class labels, i.e., meancosts, and the costs resulting from choosing the class with lowest cost for each observation. In order to compute these measures the costs for the test observations are required and therefore the Task has to be passed to performance.

```
pred = predict(mod, task = costsens.task)
pred
#> Prediction: 150 observations
#> predict.type: response
#> threshold:
#> time: 0.05
#>   id response
#> 1  1   setosa
#> 2  2   setosa
#> 3  3   setosa
#> 4  4   setosa
#> 5  5   setosa
#> 6  6   setosa
#> ... (#rows: 150, #cols: 2)

performance(pred, measures = list(meancosts, mcp), task =
    costsens.task)
#> meancosts       mcp
```

```
#>  151.0839  146.2973
```

## 3.9 Imbalanced Classification Problems

In case of *binary classification* strongly imbalanced classes often lead to unsatisfactory results regarding the prediction of new observations, especially for the small class. In this context *imbalanced classes* simply means that the number of observations of one class (usu. positive or majority class) by far exceeds the number of observations of the other class (usu. negative or minority class). This setting can be observed fairly often in practice and in various disciplines like credit scoring, fraud detection, medical diagnostics or churn management.

Most classification methods work best when the number of observations per class are roughly equal. The problem with *imbalanced classes* is that because of the dominance of the majority class classifiers tend to ignore cases of the minority class as noise and therefore predict the majority class far more often. In order to lay more weight on the cases of the minority class, there are numerous correction methods which tackle the *imbalanced classification problem*. These methods can generally be divided into *cost- and sampling-based approaches*. Below all methods supported by mlr are introduced.

### 3.9.1 Sampling-based approaches

The basic idea of *sampling methods* is to simply adjust the proportion of the classes in order to increase the weight of the minority class observations within the model.

The *sampling-based approaches* can be divided further into three different categories:

1. **Undersampling methods**: Elimination of randomly chosen cases of the majority class to decrease their effect on the classifier. All cases of the minority class are kept.
2. **Oversampling methods**: Generation of additional cases (copies, artificial observations) of the minority class to increase their effect on the classifier. All cases of the majority class are kept.
3. **Hybrid methods**: Mixture of under- and oversampling strategies.

All these methods directly access the underlying data and "rearrange" it. In this way the sampling is done as part of the *preprocesssing* and can therefore be combined with every appropriate classifier.

mlr currently supports the first two approaches.

### 3.9.2 (Simple) over- and undersampling

As mentioned above *undersampling* always refers to the majority class, while *oversampling* affects the minority class. By the use of *undersampling*, randomly chosen observations of the majority class are eliminated. Through (simple) *oversampling* all observations of the minority class are considered at least once when fitting the model. In addition, exact copies of minority class cases are created by random sampling with repetitions.

First, let's take a look at the effect for a classification task. Based on a simulated ClassifTask with imbalanced classes two new tasks (`task.over`, `task.under`) are created via mlr functions oversample and undersample, respectively.

```
data.imbal.train = rbind(
  data.frame(x = rnorm(100, mean = 1), class = "A"),
  data.frame(x = rnorm(5000, mean = 2), class = "B")
)
task = makeClassifTask(data = data.imbal.train, target = "class")
task.over = oversample(task, rate = 8)
task.under = undersample(task, rate = 1/8)

table(getTaskTargets(task))
#>
#>    A    B
#>  100 5000

table(getTaskTargets(task.over))
#>
#>    A    B
#>  800 5000

table(getTaskTargets(task.under))
#>
#>   A   B
#> 100 625
```

Please note that the *undersampling rate* has to be between 0 and 1, where 1 means no undersampling and 0.5 implies a reduction of the majority class size to 50 percent. Correspondingly, the *oversampling rate* must be greater or equal to 1, where 1 means no oversampling and 2 would result in doubling the minority class size.

As a result the performance should improve if the model is applied to new data.

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

```
data.imbal.test = rbind(
  data.frame(x = rnorm(10, mean = 1), class = "A"),
  data.frame(x = rnorm(500, mean = 2), class = "B")
)

performance(predict(mod, newdata = data.imbal.test), measures =
    list(mmce, ber, auc))
#>       mmce        ber        auc
#> 0.01960784 0.50000000 0.50000000

performance(predict(mod.over, newdata = data.imbal.test), measures =
    list(mmce, ber, auc))
#>       mmce        ber        auc
#> 0.04509804 0.41500000 0.58500000

performance(predict(mod.under, newdata = data.imbal.test), measures
    = list(mmce, ber, auc))
#>       mmce        ber        auc
#> 0.05098039 0.41800000 0.70550000
```

In this case the *performance measure* has to be considered very carefully. As the *misclassification rate* (mmce) evaluates the overall accuracy of the predictions, the *balanced error rate* (ber) and *area under the ROC Curve* (auc) might be more suitable here, as the misclassifications within each class are separately taken into account.

#### 3.9.2.1 Over- and undersampling wrappers

Alternatively, mlr also offers the integration of over- and undersampling via a wrapper approach. This way over- and undersampling can be applied to already existing learners to extend their functionality.

The example given above is repeated once again, but this time with extended learners instead of modified tasks (see makeOversampleWrapper and makeUndersampleWrapper). Just like before the *undersampling rate* has to be between 0 and 1, while the *oversampling rate* has a lower boundary of 1.

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

performance(predict(mod, newdata = data.imbal.test), measures =
    list(mmce, ber, auc))
#>       mmce        ber        auc
#> 0.01960784 0.50000000 0.50000000
```

```
performance(predict(mod.over, newdata = data.imbal.test), measures =
    list(mmce, ber, auc))
#>       mmce        ber        auc
#> 0.03333333 0.40900000 0.72020000

performance(predict(mod.under, newdata = data.imbal.test), measures
    = list(mmce, ber, auc))
#>       mmce        ber        auc
#> 0.04509804 0.41500000 0.71660000
```

#### 3.9.2.2 Extensions to oversampling

Two extensions to (simple) oversampling are available in mlr.

##### 3.9.2.2.1 1. SMOTE (Synthetic Minority Oversampling Technique)

As the duplicating of the minority class observations can lead to overfitting, within *SMOTE* the "new cases" are constructed in a different way. For each new observation, one randomly chosen minority class observation as well as one of its *randomly chosen next neighbours* are interpolated, so that finally a new *artificial observation* of the minority class is created. The smote function in mlr handles numeric as well as factor features, as the gower distance is used for nearest neighbour calculation. The factor level of the new artificial case is sampled from the given levels of the two input observations.

Analogous to oversampling, *SMOTE preprocessing* is possible via modification of the task.

```
task.smote = smote(task, rate = 8, nn = 5)
table(getTaskTargets(task))
#>
#>    A    B
#>  100 5000

table(getTaskTargets(task.smote))
#>
#>    A    B
#>  800 5000
```

Alternatively, a new wrapped learner can be created via makeSMOTEWrapper.

```
lrn.smote = makeSMOTEWrapper(lrn, sw.rate = 8, sw.nn = 5)
mod.smote = train(lrn.smote, task)
performance(predict(mod.smote, newdata = data.imbal.test), measures
    = list(mmce, ber, auc))
```

```
#>       mmce        ber        auc
#> 0.04509804 0.41500000 0.71660000
```

By default the number of nearest neighbours considered within the algorithm is set to 5.

#### 3.9.2.2.2 2. Overbagging

Another extension of oversampling consists in the combination of sampling with the bagging approach. For each iteration of the bagging process, minority class observations are oversampled with a given rate in `obw.rate`. The majority class cases can either all be taken into account for each iteration (`obw.maxcl = "all"`) or bootstrapped with replacement to increase variability between training data sets during iterations (`obw.maxcl = "boot"`).

The construction of the **Overbagging Wrapper** works similar to makeBaggingWrapper. First an existing mlr learner has to be passed to makeOverBaggingWrapper. The number of iterations or fitted models can be set via `obw.iters`.

```
lrn = makeLearner("classif.rpart", predict.type = "response")
obw.lrn = makeOverBaggingWrapper(lrn, obw.rate = 8, obw.iters = 3)
```

For *binary classification* the prediction is based on majority voting to create a discrete label. Corresponding probabilities are predicted by considering the proportions of all the predicted labels. Please note that the benefit of the sampling process is *highly dependent* on the specific learner as shown in the following example.

First, let's take a look at the tree learner with and without overbagging:

```
lrn = setPredictType(lrn, "prob")
rdesc = makeResampleDesc("CV", iters = 5)
r1 = resample(learner = lrn, task = task, resampling = rdesc,
    show.info = FALSE,
  measures = list(mmce, ber, auc))
r1$aggr
#> mmce.test.mean  ber.test.mean  auc.test.mean
#>     0.01960784     0.50000000     0.50000000

obw.lrn = setPredictType(obw.lrn, "prob")
r2 = resample(learner = obw.lrn, task = task, resampling = rdesc,
    show.info = FALSE,
  measures = list(mmce, ber, auc))
r2$aggr
#> mmce.test.mean  ber.test.mean  auc.test.mean
#>     0.04470588     0.43611719     0.58535862
```

Now let's consider a *random forest* as initial learner:

```
lrn = makeLearner("classif.randomForest")
obw.lrn = makeOverBaggingWrapper(lrn, obw.rate = 8, obw.iters = 3)

lrn = setPredictType(lrn, "prob")
r1 = resample(learner = lrn, task = task, resampling = rdesc,
    show.info = FALSE,
  measures = list(mmce, ber, auc))
r1$aggr
#> mmce.test.mean  ber.test.mean  auc.test.mean
#>     0.03509804     0.46089748     0.58514212

obw.lrn = setPredictType(obw.lrn, "prob")
r2 = resample(learner = obw.lrn, task = task, resampling = rdesc,
    show.info = FALSE,
  measures = list(mmce, ber, auc))
r2$aggr
#> mmce.test.mean  ber.test.mean  auc.test.mean
#>     0.04098039     0.45961754     0.54926842
```

While *overbagging* slighty improves the performance of the *decision tree*, the auc decreases in the second example when additional overbagging is applied. As the *random forest* itself is already a strong learner (and a bagged one as well), a further bagging step isn't very helpful here and usually won't improve the model.

### 3.9.3 Cost-based approaches

In contrast to sampling, *cost-based approaches* usually require particular learners, which can deal with different *class-dependent costs* (Cost-Sensitive Classification).

#### 3.9.3.1 Weighted classes wrapper

Another approach independent of the underlying classifier is to assign the costs as *class weights*, so that each observation receives a weight, depending on the class it belongs to. Similar to the sampling-based approaches, the effect of the minority class observations is thereby increased simply by a higher weight of these instances and vice versa for majority class observations.

In this way every learner which supports weights can be extended through the wrapper approach. If the learner does not have a direct parameter for class weights, but supports observation weights, the weights depending on the class are internally set in the wrapper.

```
lrn = makeLearner("classif.logreg")
wcw.lrn = makeWeightedClassesWrapper(lrn, wcw.weight = 0.01)
```

For binary classification, the single number passed to the classifier corresponds to the weight of the positive / majority class, while the negative / minority class receives a weight of 1. So actually, no real costs are used within this approach, but the cost ratio is taken into account.

If the underlying learner already has a parameter for class weighting (e.g., `class.weights` in `"classif.ksvm"`), the `wcw.weight` is basically passed to the specific class weighting parameter.

```
lrn = makeLearner("classif.ksvm")
wcw.lrn = makeWeightedClassesWrapper(lrn, wcw.weight = 0.01)
```

## 3.10 ROC Analysis and Performance Curves

For binary scoring classifiers a *threshold* (or *cutoff*) value controls how predicted posterior probabilities are converted into class labels. ROC curves and other performance plots serve to visualize and analyse the relationship between one or two performance measures and the threshold.

This page is mainly devoted to *receiver operating characteristic* (ROC) curves that plot the *true positive rate* (sensitivity) on the vertical axis against the *false positive rate* (1 - specificity, fall-out) on the horizontal axis for all possible threshold values. Creating other performance plots like *lift charts* or *precision/recall graphs* works analogously and is shown briefly.

In addition to performance visualization ROC curves are helpful in

- determining an optimal decision threshold for given class prior probabilities and misclassification costs (for alternatives see also the pages about cost-sensitive classification and imbalanced classification problems in this tutorial),
- identifying regions where one classifier outperforms another and building suitable multi-classifier systems,
- obtaining calibrated estimates of the posterior probabilities.

For more information see the tutorials and introductory papers by Fawcett (2004), Fawcett (2006) as well as Flach (ICML 2004).

In many applications as, e.g., diagnostic tests or spam detection, there is uncertainty about the class priors or the misclassification costs at the time of prediction, for example because it's hard to quantify the costs or because costs and class priors vary over time. Under these circumstances the classifier is expected to work well for a whole range of decision thresholds and the area under the ROC curve (AUC) provides a scalar performance measure for comparing and selecting classifiers. mlr provides the AUC for binary classification (auc) and also several generalizations of the AUC to the multi-class case (e.g., multiclass.au1p, multiclass.au1u based on Ferri et al. (2009)).

mlr offers three ways to plot ROC and other performance curves.

1. Function plotROCCurves can, based on the output of generateThreshVsPerfData, plot performance curves for any pair of performance measures available in mlr.
2. mlr offers an interface to package ROCR through function asROCRPrediction.
3. mlr's function plotViperCharts provides an interface to ViperCharts.

With mlr version 2.8 functions `generateROCRCurvesData`, `plotROCRCurves`, and `plotROCRCurvesGGVIS` were deprecated.

Below are some examples that demonstrate the three possible ways. Note that you can only use learners that are capable of predicting probabilities. Have a look at the learner table in the Appendix or run `listLearners("classif", properties = c("twoclass", "prob"))` to get a list of all learners that support this.

### 3.10.1 Performance plots with plotROCCurves

As you might recall generateThreshVsPerfData calculates one or several performance measures for a sequence of decision thresholds from 0 to 1. It provides S3 methods for objects of class Prediction, ResampleResult and BenchmarkResult (resulting from predict, resample or benchmark). plotROCCurves plots the result of generateThreshVsPerfData using ggplot2.

#### 3.10.1.1 Example 1: Single predictions

We consider the Sonar data set from package mlbench, which poses a binary classification problem (sonar.task) and apply linear discriminant analysis.

```
n = getTaskSize(sonar.task)
train.set = sample(n, size = round(2/3 * n))
test.set = setdiff(seq_len(n), train.set)

lrn1 = makeLearner("classif.lda", predict.type = "prob")
mod1 = train(lrn1, sonar.task, subset = train.set)
pred1 = predict(mod1, task = sonar.task, subset = test.set)
```

Since we want to plot ROC curves we calculate the false and true positive rates (fpr and tpr). Additionally, we also compute error rates (mmce).

```
df = generateThreshVsPerfData(pred1, measures = list(fpr, tpr, mmce))
```

generateThreshVsPerfData returns an object of class ThreshVsPerfData which contains the performance values in the `$data` element.

Per default, plotROCCurves plots the performance values of the first two measures passed to generateThreshVsPerfData. The first is shown on the x-axis, the second on the y-axis. Moreover, a diagonal line that represents the performance of a random classifier is added. You can remove the diagonal by setting `diagonal = FALSE`.

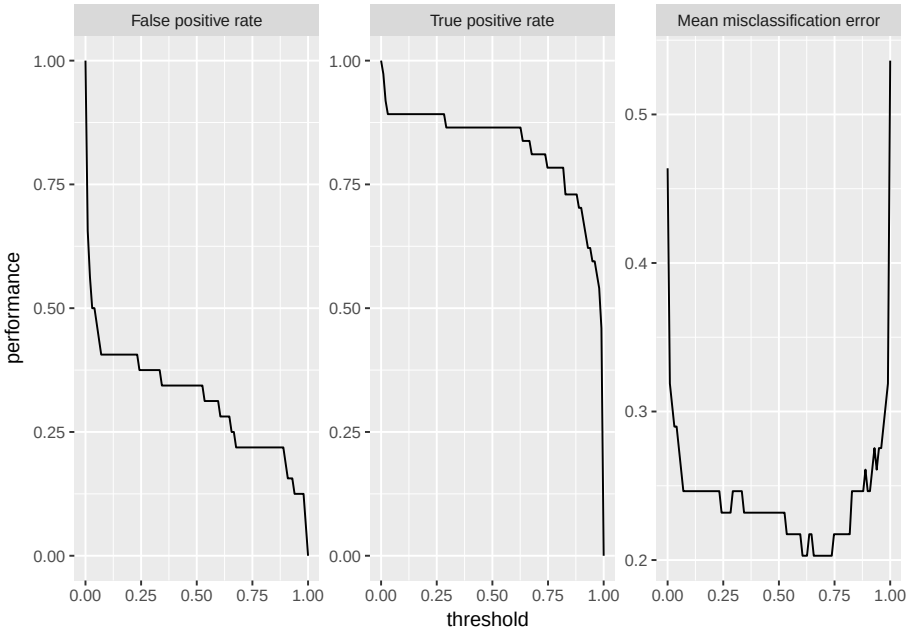
```
plotROCCurves(df)
```

The corresponding area under curve (auc) can be calculated as usual by calling performance.

```
performance(pred1, auc)
#>       auc
#> 0.847973
```

plotROCCurves always requires a pair of performance measures that are plotted against each other. If you want to plot individual measures versus the decision threshold you can use function plotThreshVsPerf.

```
plotThreshVsPerf(df)
```

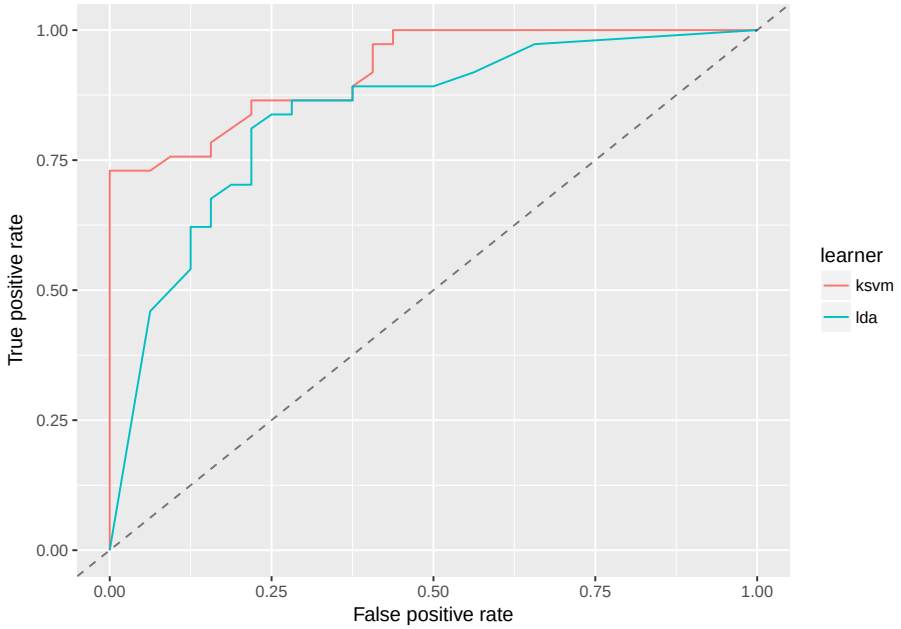



Additional to linear discriminant analysis we try a support vector machine with RBF kernel (ksvm).

```
lrn2 = makeLearner("classif.ksvm", predict.type = "prob")
mod2 = train(lrn2, sonar.task, subset = train.set)
pred2 = predict(mod2, task = sonar.task, subset = test.set)
```

In order to compare the performance of the two learners you might want to display the two corresponding ROC curves in one plot. For this purpose just pass a named list of Predictions to generateThreshVsPerfData.

```
df = generateThreshVsPerfData(list(lda = pred1, ksvm = pred2),
    measures = list(fpr, tpr))
plotROCCurves(df)
```




It's clear from the plot above that ksvm has a slightly higher AUC than lda.

```
performance(pred2, auc)
#>       auc
#> 0.9214527
```

Based on the $data member of df you can easily generate custom plots. Below the curves for the two learners are superposed.

```
qplot(x = fpr, y = tpr, color = learner, data = df$data, geom =
    "path")
```

It is easily possible to generate other performance plots by passing the appropriate performance measures to generateThreshVsPerfData and plotROCCurves. Below, we generate a *precision/recall graph* (precision = positive predictive value = ppv, recall = tpr) and a *sensitivity/specificity plot* (sensitivity = tpr, specificity = tnr).

```
df = generateThreshVsPerfData(list(lda = pred1, ksvm = pred2),
    measures = list(ppv, tpr, tnr))
```

```
### Precision/recall graph
plotROCCurves(df, measures = list(tpr, ppv), diagonal = FALSE)

### Sensitivity/specificity plot
plotROCCurves(df, measures = list(tnr, tpr), diagonal = FALSE)
```

#### 3.10.1.2 Example 2: Benchmark experiment

The analysis in the example above can be improved a little. Instead of writing individual code for training/prediction of each learner, which can become tedious very quickly, we can use function benchmark (see also Benchmark Experiments) and, ideally, the support vector machine should have been tuned.

We again consider the Sonar data set and apply lda as well as ksvm. We first generate a tuning wrapper for ksvm. The cost parameter is tuned on a (for demonstration purposes small) parameter grid. We assume that we are interested in a good performance over the complete threshold range and therefore tune with regard to the auc. The error rate (mmce) for a threshold value of 0.5 is reported as well.

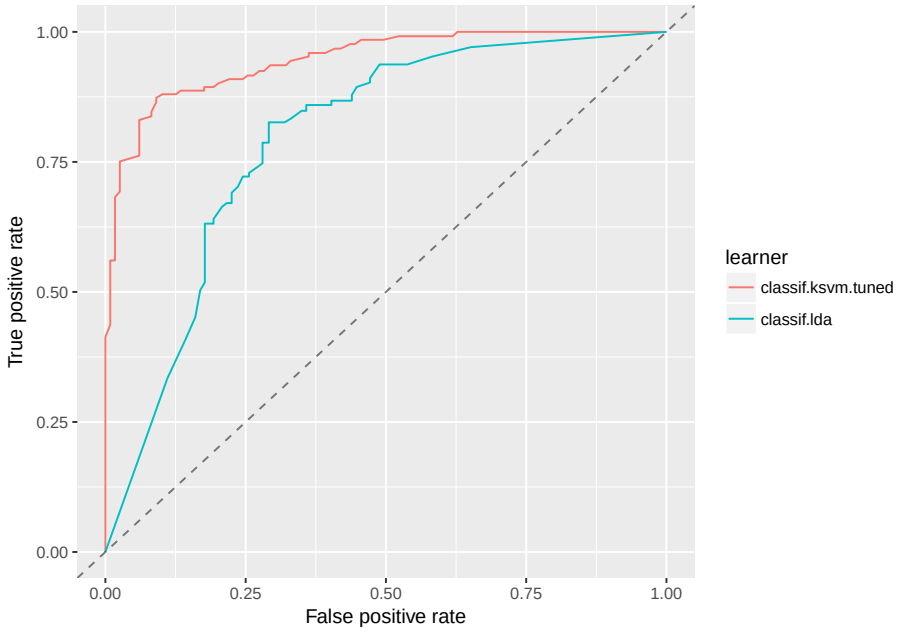
```
### Tune wrapper for ksvm
rdesc.inner = makeResampleDesc("Holdout")
ms = list(auc, mmce)
ps = makeParamSet(
  makeDiscreteParam("C", 2^(-1:1))
)
ctrl = makeTuneControlGrid()
lrn2 = makeTuneWrapper(lrn2, rdesc.inner, ms, ps, ctrl, show.info =
    FALSE)
```

Below the actual benchmark experiment is conducted. As resampling strategy we use 5-fold cross-validation and again calculate the auc as well as the error rate (for a threshold/cutoff value of 0.5).

```
### Benchmark experiment
lrns = list(lrn1, lrn2)
rdesc.outer = makeResampleDesc("CV", iters = 5)

bmr = benchmark(lrns, tasks = sonar.task, resampling = rdesc.outer,
    measures = ms, show.info = FALSE)
bmr
#>         task.id         learner.id auc.test.mean mmce.test.mean
#> 1 Sonar_example        classif.lda     0.7835442      0.2592334
#> 2 Sonar_example classif.ksvm.tuned     0.9454418      0.1390244
```

Calling generateThreshVsPerfData and plotROCCurves on the benchmark result produces a plot with ROC curves for all learners in the experiment.

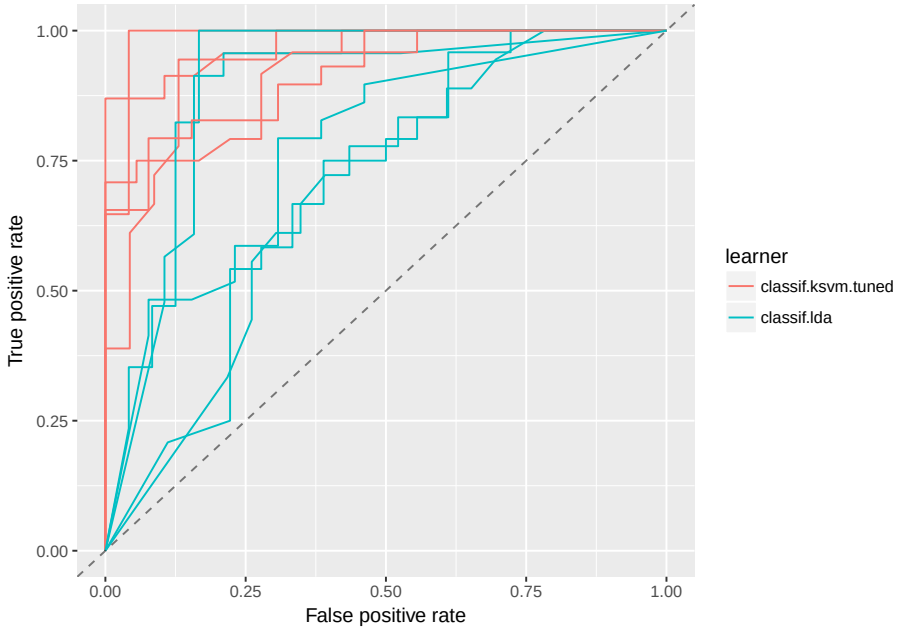
```
df = generateThreshVsPerfData(bmr, measures = list(fpr, tpr, mmce))
plotROCCurves(df)
```




Per default, generateThreshVsPerfData calculates aggregated performances according to the chosen resampling strategy (5-fold cross-validation) and aggregation scheme (test.mean) for each threshold in the sequence. This way we get *threshold-averaged* ROC curves.

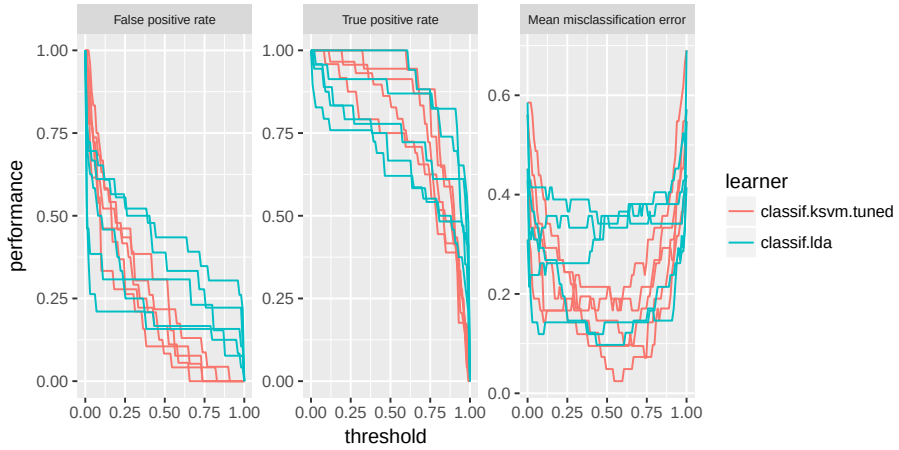
If you want to plot the individual ROC curves for each resample iteration set `aggregate = FALSE`.

```
df = generateThreshVsPerfData(bmr, measures = list(fpr, tpr, mmce),
    aggregate = FALSE)
plotROCCurves(df)
```




The same applies for plotThreshVsPerf.

```
plotThreshVsPerf(df) +
  theme(strip.text.x = element_text(size = 7))
```

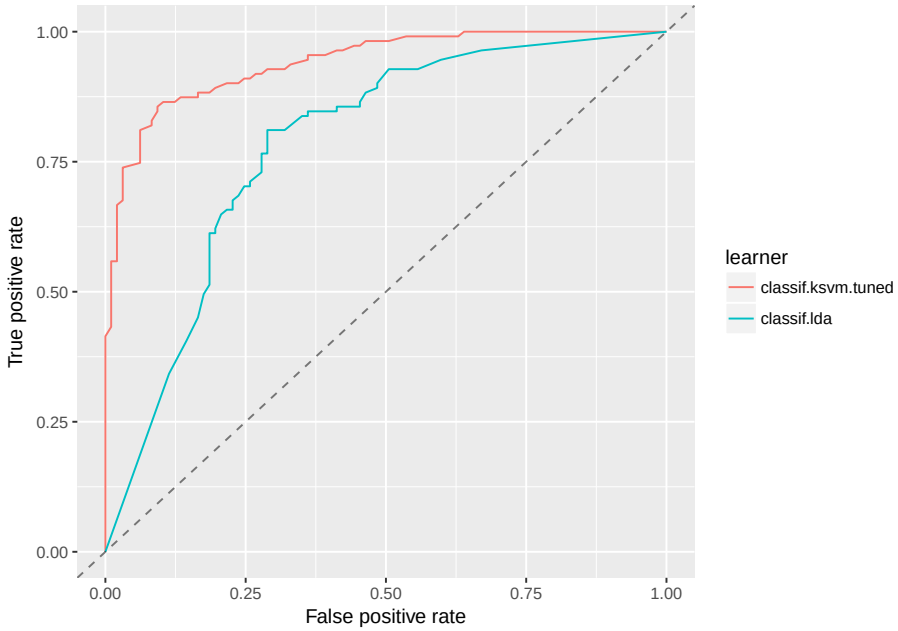



An alternative to averaging is to just merge the 5 test folds and draw a single ROC curve. Merging can be achieved by manually changing the class attribute of the prediction objects from ResamplePrediction to Prediction.

Below, the predictions are extracted from the BenchmarkResult via function getBMRPredictions, the class is changed and the ROC curves are created.

Averaging methods are normally preferred (cp. Fawcett, 2006), as they permit to assess the variability, which is needed to properly compare classifier performance.

```
### Extract predictions
preds = getBMRPredictions(bmr, drop = TRUE)

### Change the class attribute
preds2 = lapply(preds, function(x) {class(x) = "Prediction";
    return(x)})

### Draw ROC curves
df = generateThreshVsPerfData(preds2, measures = list(fpr, tpr,
    mmce))
plotROCCurves(df)
```




Again, you can easily create other standard evaluation plots by passing the appropriate performance measures to generateThreshVsPerfData and plotROCCurves.

### 3.10.2 Performance plots with asROCRPrediction

Drawing performance plots with package ROCR works through three basic commands:

1. ROCR::prediction: Create a ROCR prediction object.
2. ROCR::performance: Calculate one or more performance measures for the given prediction object.
3. ROCR::plot: Generate the performance plot.

mlr's function asROCRPrediction converts an mlr Prediction object to a ROCR prediction object, so you can easily generate performance plots by doing steps 2. and 3. yourself. ROCR's plot method has some nice features which are not (yet) available in plotROCCurves, for example plotting the convex hull of the ROC curves. Some examples are shown below.

#### 3.10.2.1 Example 1: Single predictions (continued)

We go back to out first example where we trained and predicted lda on the sonar classification task.

```
n = getTaskSize(sonar.task)
train.set = sample(n, size = round(2/3 * n))
test.set = setdiff(seq_len(n), train.set)

### Train and predict linear discriminant analysis
lrn1 = makeLearner("classif.lda", predict.type = "prob")
mod1 = train(lrn1, sonar.task, subset = train.set)
pred1 = predict(mod1, task = sonar.task, subset = test.set)
```

Below we use asROCRPrediction to convert the lda prediction, let ROCR calculate the true and false positive rate and plot the ROC curve.

```
### Convert prediction
ROCRpred1 = asROCRPrediction(pred1)

### Calculate true and false positive rate
ROCRperf1 = ROCR::performance(ROCRpred1, "tpr", "fpr")

### Draw ROC curve
ROCR::plot(ROCRperf1)
```

Below is the same ROC curve, but we make use of some more graphical parameters: The ROC curve is color-coded by the threshold and selected threshold values are printed on the curve. Additionally, the convex hull (black broken line) of the ROC curve is drawn.

```
### Draw ROC curve
ROCR::plot(ROCRperf1, colorize = TRUE, print.cutoffs.at = seq(0.1,
    0.9, 0.1), lwd = 2)

### Draw convex hull of ROC curve
```

```
ch = ROCR::performance(ROCRpred1, "rch")
ROCR::plot(ch, add = TRUE, lty = 2)
```

#### 3.10.2.2 Example 2: Benchmark experiments (continued)

We again consider the benchmark experiment conducted earlier. We first extract the predictions by getBMRPredictions and then convert them via function asROCRPrediction.

```
### Extract predictions
preds = getBMRPredictions(bmr, drop = TRUE)

### Convert predictions
ROCRpreds = lapply(preds, asROCRPrediction)

### Calculate true and false positive rate
ROCRperfs = lapply(ROCRpreds, function(x) ROCR::performance(x,
    "tpr", "fpr"))
```

We draw the vertically averaged ROC curves (solid lines) as well as the ROC curves for the individual resampling iterations (broken lines). Moreover, standard error bars are plotted for selected true positive rates (0.1, 0.2, ..., 0.9). See ROCR's plot function for details.

```
### lda average ROC curve
plot(ROCRperfs[[1]], col = "blue", avg = "vertical", spread.estimate
    = "stderror",
  show.spread.at = seq(0.1, 0.8, 0.1), plotCI.col = "blue",
      plotCI.lwd = 2, lwd = 2)
### lda individual ROC curves
plot(ROCRperfs[[1]], col = "blue", lty = 2, lwd = 0.25, add = TRUE)

### ksvm average ROC curve
plot(ROCRperfs[[2]], col = "red", avg = "vertical", spread.estimate
    = "stderror",
  show.spread.at = seq(0.1, 0.6, 0.1), plotCI.col = "red",
      plotCI.lwd = 2, lwd = 2, add = TRUE)
### ksvm individual ROC curves
plot(ROCRperfs[[2]], col = "red", lty = 2, lwd = 0.25, add = TRUE)

legend("bottomright", legend = getBMRLearnerIds(bmr), lty = 1, lwd =
    2, col = c("blue", "red"))
```

In order to create other evaluation plots like *precision/recall graphs* you just have to change the performance measures when calling ROCR::performance. (Note that you have to use the measures provided by ROCR listed here and not mlr's performance measures.)

```
### Extract and convert predictions
preds = getBMRPredictions(bmr, drop = TRUE)
ROCRpreds = lapply(preds, asROCRPrediction)

### Calculate precision and recall
ROCRperfs = lapply(ROCRpreds, function(x) ROCR::performance(x,
    "prec", "rec"))

### Draw performance plot
plot(ROCRperfs[[1]], col = "blue", avg = "threshold")
plot(ROCRperfs[[2]], col = "red", avg = "threshold", add = TRUE)
legend("bottomleft", legend = getBMRLearnerIds(bmr), lty = 1, col =
    c("blue", "red"))
```

If you want to plot a performance measure versus the threshold, specify only one measure when calling ROCR::performance. Below the average accuracy over the 5 cross-validation iterations is plotted against the threshold. Moreover, boxplots for certain threshold values (0.1, 0.2, ..., 0.9) are drawn.

```
### Extract and convert predictions
preds = getBMRPredictions(bmr, drop = TRUE)
ROCRpreds = lapply(preds, asROCRPrediction)

### Calculate accuracy
ROCRperfs = lapply(ROCRpreds, function(x) ROCR::performance(x,
    "acc"))

### Plot accuracy versus threshold
plot(ROCRperfs[[1]], avg = "vertical", spread.estimate = "boxplot",
    lwd = 2, col = "blue",
  show.spread.at = seq(0.1, 0.9, 0.1), ylim = c(0,1), xlab =
      "Threshold")
```

### 3.10.3 Viper charts

mlr also supports ViperCharts for plotting ROC and other performance curves. Like generateThreshVsPerfData it has S3 methods for objects of class Prediction, ResampleResult and BenchmarkResult. Below plots for the benchmark experiment (Example 2) are generated.

```
z = plotViperCharts(bmr, chart = "rocc", browse = FALSE)
```

You can see the plot created this way here. Note that besides ROC curves you get several other plots like lift charts or cost curves. For details, see plotViperCharts.

## 3.11 Multilabel Classification

Multilabel classification is a classification problem where multiple target labels can be assigned to each observation instead of only one like in multiclass classification.

Two different approaches exist for multilabel classification. *Problem transformation methods* try to transform the multilabel classification into binary or multiclass classification problems. *Algorithm adaptation methods* adapt multiclass algorithms so they can be applied directly to the problem.

### 3.11.1 Creating a task

The first thing you have to do for multilabel classification in mlr is to get your data in the right format. You need a data.frame which consists of the features and a logical vector for each label which indicates if the label is present in the observation or not. After that you can create a MultilabelTask like a normal ClassifTask. Instead of one target name you have to specify a vector of targets which correspond to the names of logical variables in the data.frame. In the following example we get the yeast data frame from the already existing yeast.task, extract the 14 label names and create the task again.

```
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
```

### 3.11.2 Constructing a learner

Multilabel classification in mlr can currently be done in two ways:

- Algorithm adaptation methods: Treat the whole problem with a specific algorithm.
- Problem transformation methods: Transform the problem, so that simple binary classification algorithms can be applied.

#### 3.11.2.1 Algorithm adaptation methods

Currently the available algorithm adaptation methods in **R** are the multivariate random forest in the randomForestSRC package and the random ferns multilabel algorithm in the rFerns package. You can create the learner for these algorithms like in multiclass classification problems.

```
lrn.rfsrc = makeLearner("multilabel.randomForestSRC")
lrn.rFerns = makeLearner("multilabel.rFerns")
lrn.rFerns
#> Learner multilabel.rFerns from package rFerns
#> Type: multilabel
#> Name: Random ferns; Short name: rFerns
#> Class: multilabel.rFerns
#> Properties: numerics,factors,ordered
#> Predict-Type: response
#> Hyperparameters:
```

#### 3.11.2.2 Problem transformation methods

For generating a wrapped multilabel learner first create a binary (or multiclass) classification learner with makeLearner. Afterwards apply a function like makeMultilabelBinaryRelevanceWrapper, makeMultilabelClassifierChainsWrapper, makeMultilabelNestedStackingWrapper, makeMultilabelDBRWrapper or makeMultilabelStackingWrapper on the learner to convert it to a learner that uses the respective problem transformation method.

You can also generate a binary relevance learner directly, as you can see in the example.

```
lrn.br = makeLearner("classif.rpart", predict.type = "prob")
lrn.br = makeMultilabelBinaryRelevanceWrapper(lrn.br)
lrn.br
#> Learner multilabel.binaryRelevance.classif.rpart from package
    rpart
#> Type: multilabel
#> Name: ; Short name:
```

```
#> Class: MultilabelBinaryRelevanceWrapper
#> Properties:
    numerics,factors,ordered,missings,weights,prob,twoclass,multiclass
#> Predict-Type: prob
#> Hyperparameters: xval=0

lrn.br2 = makeMultilabelBinaryRelevanceWrapper("classif.rpart")
lrn.br2
#> Learner multilabel.binaryRelevance.classif.rpart from package
    rpart
#> Type: multilabel
#> Name: ; Short name:
#> Class: MultilabelBinaryRelevanceWrapper
#> Properties:
    numerics,factors,ordered,missings,weights,prob,twoclass,multiclass
#> Predict-Type: response
#> Hyperparameters: xval=0
```

The different methods are shortly described in the following.

#### 3.11.2.2.1 Binary relevance

This problem transformation method converts the multilabel problem to binary classification problems for each label and applies a simple binary classificator on these. In mlr this can be done by converting your binary learner to a wrapped binary relevance multilabel learner.

#### 3.11.2.2.2 Classifier chains

Trains consecutively the labels with the input data. The input data in each step is augmented by the already trained labels (with the real observed values). Therefore an order of the labels has to be specified. At prediction time the labels are predicted in the same order as while training. The required labels in the input data are given by the previous done prediction of the respective label.

#### 3.11.2.2.3 Nested stacking

Same as classifier chains, but the labels in the input data are not the real ones, but estimations of the labels obtained by the already trained learners.

#### 3.11.2.2.4 Dependent binary relevance

Each label is trained with the real observed values of all other labels. In prediction phase for a label the other necessary labels are obtained in a previous step by a base learner like the binary relevance method.

#### 3.11.2.2.5 Stacking

Same as the dependent binary relevance method, but in the training phase the labels used as input for each label are obtained by the binary relevance method.

### 3.11.3 Train

You can train a model as usual with a multilabel learner and a multilabel task as input. You can also pass `subset` and `weights` arguments if the learner supports this.

```
mod = train(lrn.br, yeast.task)
mod = train(lrn.br, yeast.task, subset = 1:1500, weights =
    rep(1/1500, 1500))
mod
#> Model for learner.id=multilabel.binaryRelevance.classif.rpart;
    learner.class=MultilabelBinaryRelevanceWrapper
#> Trained on: task.id = multi; obs = 1500; features = 103
#> Hyperparameters: xval=0

mod2 = train(lrn.rfsrc, yeast.task, subset = 1:100)
mod2
#> Model for learner.id=multilabel.randomForestSRC;
    learner.class=multilabel.randomForestSRC
#> Trained on: task.id = multi; obs = 100; features = 103
#> Hyperparameters: na.action=na.impute
```

### 3.11.4 Predict

Prediction can be done as usual in mlr with predict and by passing a trained model and either the task to the `task` argument or some new data to the `newdata` argument. As always you can specify a `subset` of the data which should be predicted.

```
pred = predict(mod, task = yeast.task, subset = 1:10)
pred = predict(mod, newdata = yeast[1501:1600,])
names(as.data.frame(pred))
#>  [1] "truth.label1"     "truth.label2"     "truth.label3"
#>  [4] "truth.label4"     "truth.label5"     "truth.label6"
#>  [7] "truth.label7"     "truth.label8"     "truth.label9"
#> [10] "truth.label10"    "truth.label11"    "truth.label12"
#> [13] "truth.label13"    "truth.label14"    "prob.label1"
#> [16] "prob.label2"      "prob.label3"      "prob.label4"
#> [19] "prob.label5"      "prob.label6"      "prob.label7"
#> [22] "prob.label8"      "prob.label9"      "prob.label10"
```

```
#> [25] "prob.label11"     "prob.label12"     "prob.label13"
#> [28] "prob.label14"     "response.label1"  "response.label2"
#> [31] "response.label3"  "response.label4"  "response.label5"
#> [34] "response.label6"  "response.label7"  "response.label8"
#> [37] "response.label9"  "response.label10" "response.label11"
#> [40] "response.label12" "response.label13" "response.label14"

pred2 = predict(mod2, task = yeast.task)
names(as.data.frame(pred2))
#>  [1] "id"               "truth.label1"     "truth.label2"
#>  [4] "truth.label3"     "truth.label4"     "truth.label5"
#>  [7] "truth.label6"     "truth.label7"     "truth.label8"
#> [10] "truth.label9"     "truth.label10"    "truth.label11"
#> [13] "truth.label12"    "truth.label13"    "truth.label14"
#> [16] "response.label1"  "response.label2"  "response.label3"
#> [19] "response.label4"  "response.label5"  "response.label6"
#> [22] "response.label7"  "response.label8"  "response.label9"
#> [25] "response.label10" "response.label11" "response.label12"
#> [28] "response.label13" "response.label14"
```

Depending on the chosen `predict.type` of the learner you get true and predicted values and possibly probabilities for each class label. These can be extracted by the usual accessor functions getPredictionTruth, getPredictionResponse and getPredictionProbabilities.

### 3.11.5 Performance

The performance of your prediction can be assessed via function performance. You can specify via the `measures` argument which measure(s) to calculate. The default measure for multilabel classification is the Hamming loss (multilabel.hamloss). All available measures for multilabel classification can be shown by listMeasures and found in the table of performance measures and the measures documentation page.

```
performance(pred)
#> multilabel.hamloss
#>          0.2257143

performance(pred2, measures = list(multilabel.subset01,
    multilabel.hamloss, multilabel.acc,
  multilabel.f1, timepredict))
#> multilabel.subset01  multilabel.hamloss      multilabel.acc
#>           0.8642946           0.2043560           0.4639072
#>       multilabel.f1         timepredict
#>           0.5730995           1.8190000
```

```
listMeasures("multilabel")
#>  [1] "featperc"            "multilabel.tpr"
    "multilabel.hamloss"
#>  [4] "multilabel.subset01" "timeboth"            "timetrain"
#>  [7] "timepredict"         "multilabel.ppv"      "multilabel.f1"
#> [10] "multilabel.acc"
```

### 3.11.6 Resampling

For evaluating the overall performance of the learning algorithm you can do some resampling. As usual you have to define a resampling strategy, either via makeResampleDesc or makeResampleInstance. After that you can run the resample function. Below the default measure Hamming loss is calculated.

```
rdesc = makeResampleDesc(method = "CV", stratify = FALSE, iters = 3)
r = resample(learner = lrn.br, task = yeast.task, resampling =
    rdesc, show.info = FALSE)
r
#> Resample Result
#> Task: multi
#> Learner: multilabel.binaryRelevance.classif.rpart
#> Aggr perf: multilabel.hamloss.test.mean=0.2220267
#> Runtime: 6.14522

r = resample(learner = lrn.rFerns, task = yeast.task, resampling =
    rdesc, show.info = FALSE)
r
#> Resample Result
#> Task: multi
#> Learner: multilabel.rFerns
#> Aggr perf: multilabel.hamloss.test.mean=0.4745012
#> Runtime: 0.836878
```

### 3.11.7 Binary performance

If you want to calculate a binary performance measure like, e.g., the accuracy, the mmce or the auc for each label, you can use function getMultilabelBinaryPerformances. You can apply this function to any multilabel prediction, e.g., also on the resample multilabel prediction. For calculating the auc you need predicted probabilities.

```
getMultilabelBinaryPerformances(pred, measures = list(acc, mmce,
    auc))
#>         acc.test.mean mmce.test.mean auc.test.mean
```

```
#> label1          0.75          0.25     0.6321925
#> label2          0.64          0.36     0.6547917
#> label3          0.68          0.32     0.7118227
#> label4          0.69          0.31     0.6764835
#> label5          0.73          0.27     0.6676923
#> label6          0.70          0.30     0.6417739
#> label7          0.81          0.19     0.5968750
#> label8          0.73          0.27     0.5164474
#> label9          0.89          0.11     0.4688458
#> label10         0.86          0.14     0.3996463
#> label11         0.85          0.15     0.5000000
#> label12         0.76          0.24     0.5330667
#> label13         0.75          0.25     0.5938610
#> label14         1.00          0.00            NA

getMultilabelBinaryPerformances(r$pred, measures = list(acc, mmce))
#>         acc.test.mean mmce.test.mean
#> label1     0.69880017      0.3011998
#> label2     0.58543649      0.4145635
#> label3     0.70086885      0.2991312
#> label4     0.71493587      0.2850641
#> label5     0.71162598      0.2883740
#> label6     0.58874638      0.4112536
#> label7     0.53165081      0.4683492
#> label8     0.53496070      0.4650393
#> label9     0.30740588      0.6925941
#> label10    0.44269756      0.5573024
#> label11    0.45097228      0.5490277
#> label12    0.53330575      0.4666942
#> label13    0.53868432      0.4613157
#> label14    0.01696318      0.9830368
```

## 3.12 Learning Curve Analysis

To analyze how the increase of observations in the training set improves the performance of a learner the *learning curve* is an appropriate visual tool. The experiment is conducted with an increasing subsample size and the performance is measured. In the plot the x-axis represents the relative subsample size whereas the y-axis represents the performance.

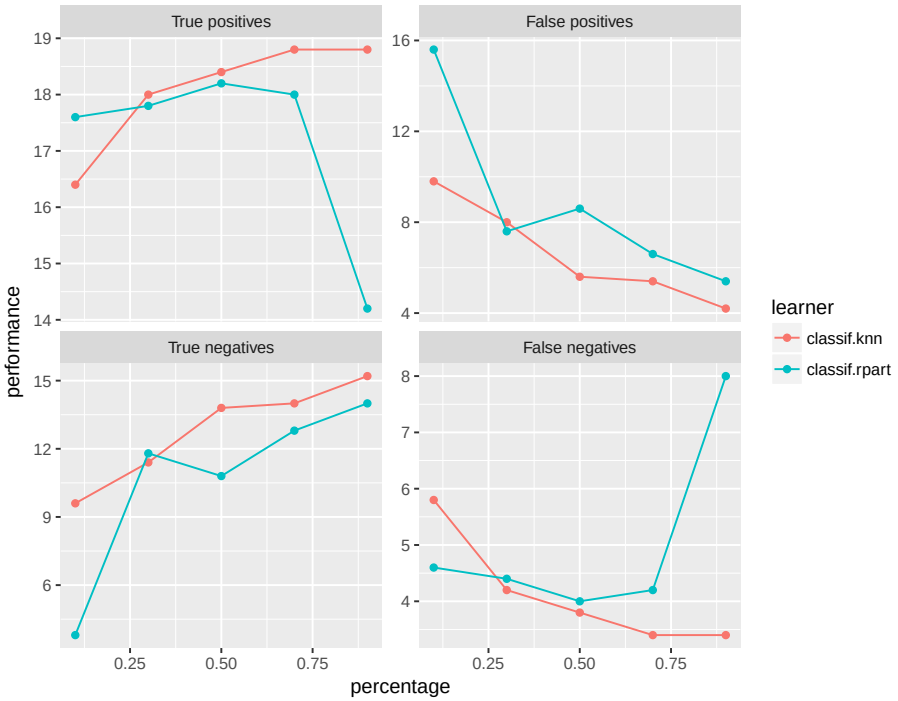
Note that this function internally uses benchmark in combination with makeDownsampleWrapper, so for every run new observations are drawn. Thus the results are noisy. To reduce noise increase the number of resampling iterations. You can define the resampling method in the `resampling` argument of generateLearningCurveData. It is also possible to pass a ResampleInstance (which is a

result of makeResampleInstance) to make resampling consistent for all passed learners and each step of increasing the number of observations.

### 3.12.1 Plotting the learning curve

The mlr function generateLearningCurveData can generate the data for *learning curves* for multiple learners and multiple performance measures at once. With plotLearningCurve the result of generateLearningCurveData can be plotted using ggplot2. plotLearningCurve has an argument `facet` which can be either `"measure"` or `"learner"`. By default `facet = "measure"` and facetted subplots are created for each measure input to generateLearningCurveData. If `facet = "measure"` learners are mapped to color, and vice versa.

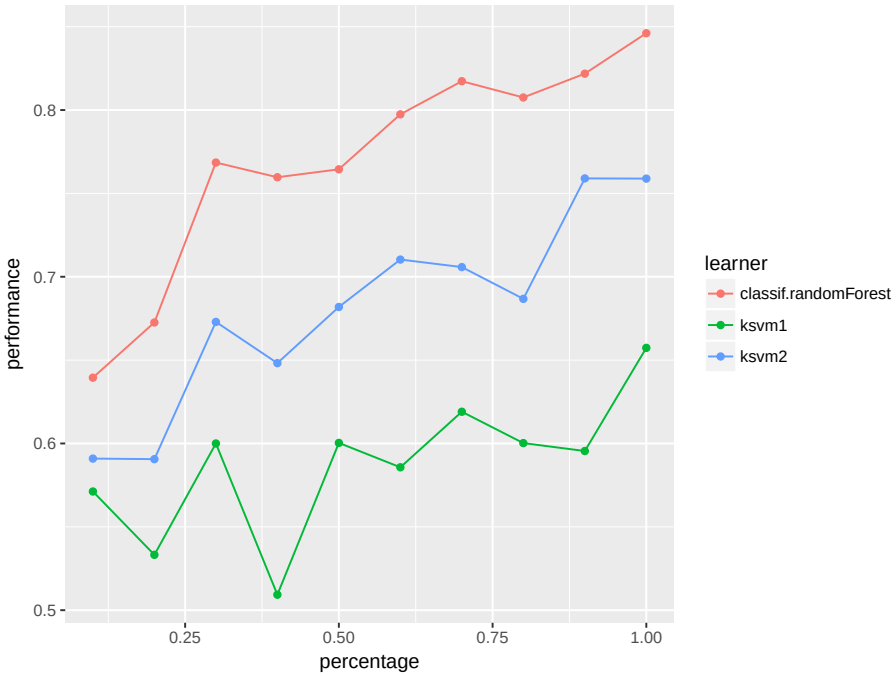
```
r = generateLearningCurveData(
  learners = c("classif.rpart", "classif.knn"),
  task = sonar.task,
  percs = seq(0.1, 1, by = 0.2),
  measures = list(tp, fp, tn, fn),
  resampling = makeResampleDesc(method = "CV", iters = 5),
  show.info = FALSE)
plotLearningCurve(r)
```



What happens in generateLearningCurveData is the following: Each learner will be internally wrapped in a DownsampleWrapper. To measure the performance at the first step of `percs`, say `0.1`, first the data will be split into a *training* and a *test set* according to the given *resampling strategy*. Then a random sample containing 10% of the observations of the *training set* will be drawn and used to train the learner. The performance will be measured on the *complete test set*. These steps will be repeated as defined by the given *resampling method* and for each value of `percs`.

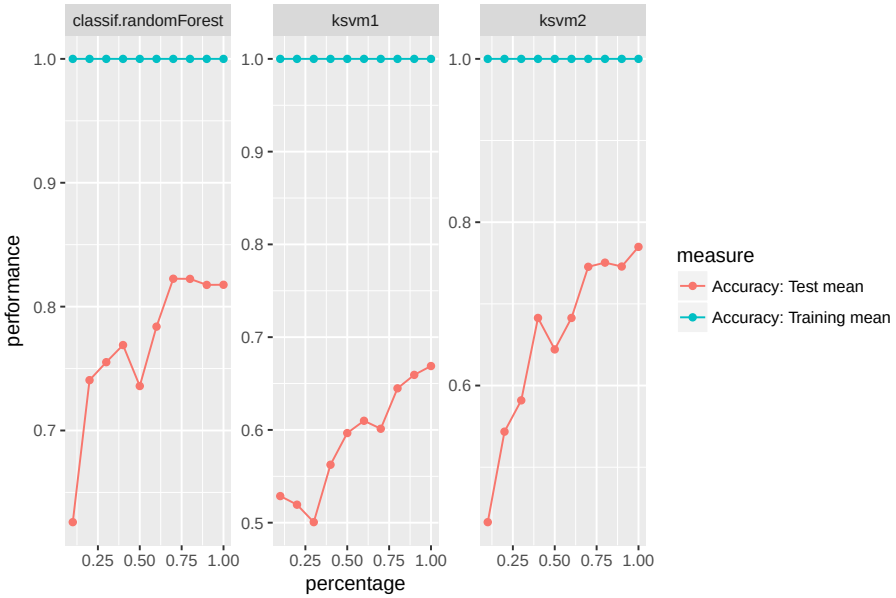
In the first example we simply passed a vector of learner names to [generateLearningCurveData]. As usual, you can also create the learners beforehand and provide a list of Learner objects, or even pass a mixed list of Learner objects and strings. Make sure that all learners have unique `ids`.

```
lrns = list(
  makeLearner(cl = "classif.ksvm", id = "ksvm1", sigma = 0.2, C = 2),
  makeLearner(cl = "classif.ksvm", id = "ksvm2", sigma = 0.1, C = 1),
  "classif.randomForest"
)
rin = makeResampleDesc(method = "CV", iters = 5)
lc = generateLearningCurveData(learners = lrns, task = sonar.task,
  percs = seq(0.1, 1, by = 0.1), measures = acc,
  resampling = rin, show.info = FALSE)
plotLearningCurve(lc)
```



We can display performance on the train set as well as the test set:

```
rin2 = makeResampleDesc(method = "CV", iters = 5, predict = "both")
lc2 = generateLearningCurveData(learners = lrns, task = sonar.task,
  percs = seq(0.1, 1, by = 0.1),
  measures = list(acc, setAggregation(acc, train.mean)), resampling
      = rin2,
  show.info = FALSE)
plotLearningCurve(lc2, facet = "learner")
```




There is also an experimental ggvis plotting function, plotLearningCurveGGVIS. Instead of the `facet` argument to plotLearningCurve there is an argument `interaction` which plays a similar role. As subplots are not available in ggvis, measures or learners are mapped to an interactive sidebar which allows selection of the displayed measures or learners. The other feature is mapped to color.

```
plotLearningCurveGGVIS(lc2, interaction = "measure")
plotLearningCurveGGVIS(lc2, interaction = "learner")
```

## 3.13 Exploring Learner Predictions

Learners use features to learn a prediction function and make predictions, but the effect of those features is often not apparent. mlr can estimate the partial dependence of a learned function on a subset of the feature space using generatePartialDependenceData.

Partial dependence plots reduce the potentially high dimensional function estimated by the learner, and display a marginalized version of this function in a lower dimensional space. For example suppose $Y = f(X) + \epsilon$, where $\mathbb{E}[\epsilon|X] = 0$. With $(X, Y)$ pairs drawn independently from this statistical model, a learner may estimate $\hat{f}$, which, if $X$ is high dimensional, can be uninterpretable. Suppose we want to approximate the relationship between some subset of $X$. We partition $X$ into two sets, $X_s$ and $X_c$ such that $X = X_s \cup X_c$, where $X_s$ is a subset of $X$ of interest.

The partial dependence of $f$ on $X_s$ is

$$f_{X_s} = \mathbb{E}_{X_c} f(X_s, X_c).$$

$X_c$ is integrated out. We use the following estimator:

$$\hat{f}_{X_s} = \frac{1}{N} \sum_{i=1}^{N} \hat{f}(X_s, x_{ic}).$$

The individual conditional expectation of an observation can also be estimated using the above algorithm absent the averaging, giving $\hat{f}^{(i)}_{X_s}$. This allows the discovery of features of $\hat{f}$ that may be obscured by an aggregated summary of $\hat{f}$.

The partial derivative of the partial dependence function, $\frac{\partial \hat{f}_{X_s}}{\partial X_s}$, and the individual conditional expectation function, $\frac{\partial \hat{f}^{(i)}_{X_s}}{\partial X_s}$, can also be computed. For regression and survival tasks the partial derivative of a single feature $X_s$ is the gradient of the partial dependence function, and for classification tasks where the learner can output class probabilities the Jacobian. Note that if the learner produces discontinuous partial dependence (e.g., piecewise constant functions such as decision trees, ensembles of decision trees, etc.) the derivative will be 0 (where the function is not changing) or trending towards positive or negative infinity (at the discontinuities where the derivative is undefined). Plotting the partial dependence function of such learners may give the impression that the function is not discontinuous because the prediction grid is not composed of all discontinuous points in the predictor space. This results in a line interpolating that makes the function appear to be piecewise linear (where the derivative would be defined except at the boundaries of each piece).

The partial derivative can be informative regarding the additivity of the learned function in certain features. If $\hat{f}^{(i)}_{X_s}$ is an additive function in a feature $X_s$, then its partial derivative will not depend on any other features ($X_c$) that may have been used by the learner. Variation in the estimated partial derivative indicates that there is a region of interaction between $X_s$ and $X_c$ in $\hat{f}$. Similarly, instead of using the mean to estimate the expected value of the function at different

values of $X_s$, instead computing the variance can highlight regions of interaction between $X_s$ and $X_c$.

See Goldstein, Kapelner, Bleich, and Pitkin (2014) for more details and their package ICEbox for the original implementation. The algorithm works for any supervised learner with classification, regression, and survival tasks.

### 3.13.1 Generating partial dependences

Our implementation, following mlr's visualization pattern, consists of the above mentioned function generatePartialDependenceData, as well as two visualization functions, plotPartialDependence and plotPartialDependenceGGVIS. The former generates input (objects of class PartialDependenceData) for the latter.

The first step executed by generatePartialDependenceData is to generate a feature grid for every element of the character vector `features` passed. The data are given by the `input` argument, which can be a Task or a data.frame. The feature grid can be generated in several ways. A uniformly spaced grid of length `gridsize` (default 10) from the empirical minimum to the empirical maximum is created by default, but arguments `fmin` and `fmax` may be used to override the empirical default (the lengths of `fmin` and `fmax` must match the length of `features`). Alternatively the feature data can be resampled, either by using a bootstrap or by subsampling.

```
lrn.classif = makeLearner("classif.ksvm", predict.type = "prob")
fit.classif = train(lrn.classif, iris.task)
pd = generatePartialDependenceData(fit.classif, iris.task,
    "Petal.Width")
pd
#> PartialDependenceData
#> Task: iris_example
#> Features: Petal.Width
#> Target: setosa, versicolor, virginica
#> Derivative: FALSE
#> Interaction: FALSE
#> Individual: FALSE
#>    Class Probability Petal.Width
#> 1 setosa   0.4983925   0.1000000
#> 2 setosa   0.4441165   0.3666667
#> 3 setosa   0.3808075   0.6333333
#> 4 setosa   0.3250243   0.9000000
#> 5 setosa   0.2589014   1.1666667
#> 6 setosa   0.1870692   1.4333333
#> ... (#rows: 30, #cols: 3)
```

As noted above, $X_s$ does not have to be unidimensional. If it is not, the `interaction` flag must be set to `TRUE`. Then the individual feature grids are

combined using the Cartesian product, and the estimator above is applied, producing the partial dependence for every combination of unique feature values. If the `interaction` flag is `FALSE` (the default) then by default $X_s$ is assumed unidimensional, and partial dependencies are generated for each feature separately. The resulting output when `interaction = FALSE` has a column for each feature, and `NA` where the feature was not used.

```
pd.lst = generatePartialDependenceData(fit.classif, iris.task,
    c("Petal.Width", "Petal.Length"), FALSE)
head(pd.lst$data)
#>    Class Probability Petal.Width Petal.Length
#> 1 setosa   0.4983925   0.1000000           NA
#> 2 setosa   0.4441165   0.3666667           NA
#> 3 setosa   0.3808075   0.6333333           NA
#> 4 setosa   0.3250243   0.9000000           NA
#> 5 setosa   0.2589014   1.1666667           NA
#> 6 setosa   0.1870692   1.4333333           NA

tail(pd.lst$data)
#>        Class Probability Petal.Width Petal.Length
#> 55 virginica   0.2006336          NA     3.622222
#> 56 virginica   0.3114545          NA     4.277778
#> 57 virginica   0.4404613          NA     4.933333
#> 58 virginica   0.6005358          NA     5.588889
#> 59 virginica   0.7099841          NA     6.244444
#> 60 virginica   0.7242584          NA     6.900000
```

```
pd.int = generatePartialDependenceData(fit.classif, iris.task,
    c("Petal.Width", "Petal.Length"), TRUE)
pd.int
#> PartialDependenceData
#> Task: iris_example
#> Features: Petal.Width, Petal.Length
#> Target: setosa, versicolor, virginica
#> Derivative: FALSE
#> Interaction: TRUE
#> Individual: FALSE
#>    Class Probability Petal.Width Petal.Length
#> 1 setosa   0.6885025   0.1000000            1
#> 2 setosa   0.6824560   0.3666667            1
#> 3 setosa   0.6459476   0.6333333            1
#> 4 setosa   0.5750861   0.9000000            1
#> 5 setosa   0.4745925   1.1666667            1
#> 6 setosa   0.3749285   1.4333333            1
#> ... (#rows: 300, #cols: 4)
```

At each step in the estimation of $\hat{f}_{X_s}$ a set of predictions of length $N$ is generated. By default the mean prediction is used. For classification where `predict.type = "prob"` this entails the mean class probabilities. However, other summaries of the predictions may be used. For regression and survival tasks the function used here must either return one number or three, and, if the latter, the numbers must be sorted lowest to highest. For classification tasks the function must return a number for each level of the target feature.

As noted, the `fun` argument can be a function which returns three numbers (sorted low to high) for a regression task. This allows further exploration of relative feature importance. If a feature is relatively important, the bounds are necessarily tighter because the feature accounts for more of the variance of the predictions, i.e., it is "used" more by the learner. More directly setting `fun = var` identifies regions of interaction between $X_s$ and $X_c$.

```
lrn.regr = makeLearner("regr.ksvm")
fit.regr = train(lrn.regr, bh.task)
pd.regr = generatePartialDependenceData(fit.regr, bh.task, "lstat",
    fun = median)
pd.regr
#> PartialDependenceData
#> Task: BostonHousing-example
#> Features: lstat
#> Target: medv
#> Derivative: FALSE
#> Interaction: FALSE
#> Individual: FALSE
#>       medv     lstat
#> 1 24.69031  1.730000
#> 2 23.72479  5.756667
#> 3 22.34841  9.783333
#> 4 20.78817 13.810000
#> 5 19.76183 17.836667
#> 6 19.33115 21.863333
#> ... (#rows: 10, #cols: 2)
```

```
pd.ci = generatePartialDependenceData(fit.regr, bh.task, "lstat",
  fun = function(x) quantile(x, c(.25, .5, .75)))
pd.ci
#> PartialDependenceData
#> Task: BostonHousing-example
#> Features: lstat
#> Target: medv
#> Derivative: FALSE
#> Interaction: FALSE
#> Individual: FALSE
#>       medv     lstat    lower    upper
```

```
#> 1 24.69031  1.730000 21.36068 29.75615
#> 2 23.72479  5.756667 20.80590 28.02338
#> 3 22.34841  9.783333 20.06507 25.22291
#> 4 20.78817 13.810000 18.55592 23.68100
#> 5 19.76183 17.836667 16.52737 22.98520
#> 6 19.33115 21.863333 15.14425 22.12766
#> ... (#rows: 10, #cols: 4)
```

```
pd.classif = generatePartialDependenceData(fit.classif, iris.task,
    "Petal.Length", fun = median)
pd.classif
#> PartialDependenceData
#> Task: iris_example
#> Features: Petal.Length
#> Target: setosa, versicolor, virginica
#> Derivative: FALSE
#> Interaction: FALSE
#> Individual: FALSE
#>    Class Probability Petal.Length
#> 1 setosa  0.31008788     1.000000
#> 2 setosa  0.24271454     1.655556
#> 3 setosa  0.17126036     2.311111
#> 4 setosa  0.09380787     2.966667
#> 5 setosa  0.04579912     3.622222
#> 6 setosa  0.02455344     4.277778
#> ... (#rows: 30, #cols: 3)
```

In addition to bounds based on a summary of the distribution of the conditional expectation of each observation, learners which can estimate the variance of their predictions can also be used. The argument `bounds` is a numeric vector of length two which is added (so the first number should be negative) to the point prediction to produce a confidence interval for the partial dependence. The default is the .025 and .975 quantiles of the Gaussian distribution.

```
fit.se = train(makeLearner("regr.randomForest", predict.type =
    "se"), bh.task)
pd.se = generatePartialDependenceData(fit.se, bh.task, c("lstat",
    "crim"))
head(pd.se$data)
#>       medv     lstat crim    lower    upper
#> 1 31.45061  1.730000   NA 12.66252 50.23870
#> 2 26.07468  5.756667   NA 14.45816 37.69120
#> 3 23.50402  9.783333   NA 13.53661 33.47144
#> 4 22.09801 13.810000   NA 14.25053 29.94550
#> 5 20.44824 17.836667   NA 12.87242 28.02405
#> 6 19.82650 21.863333   NA 11.85056 27.80244
```

```
tail(pd.se$data)
#>        medv lstat     crim    lower    upper
#> 15 21.85229    NA 39.54849 10.72496 32.97962
#> 16 21.83147    NA 49.43403 10.67659 32.98635
#> 17 21.81572    NA 59.31957 10.63741 32.99402
#> 18 21.79497    NA 69.20512 10.55183 33.03810
#> 19 21.79698    NA 79.09066 10.55663 33.03733
#> 20 21.79698    NA 88.97620 10.55663 33.03733
```

As previously mentioned if the aggregation function is not used, i.e., it is the identity, then the conditional expectation of $\hat{f}_{X_s}^{(i)}$ is estimated. If `individual = TRUE` then generatePartialDependenceData returns $n$ partial dependence estimates made at each point in the prediction grid constructed from the features.

```
pd.ind.regr = generatePartialDependenceData(fit.regr, bh.task,
    "lstat", individual = TRUE)
pd.ind.regr
#> PartialDependenceData
#> Task: BostonHousing-example
#> Features: lstat
#> Target: medv
#> Derivative: FALSE
#> Interaction: FALSE
#> Individual: TRUE
#> Predictions centered: FALSE
#>       medv     lstat idx
#> 1 25.66995  1.730000   1
#> 2 24.71747  5.756667   1
#> 3 23.64157  9.783333   1
#> 4 22.70812 13.810000   1
#> 5 22.00059 17.836667   1
#> 6 21.46195 21.863333   1
#> ... (#rows: 5060, #cols: 3)
```

The resulting output, particularly the element `data` in the returned object, has an additional column `idx` which gives the index of the observation to which the row pertains.

For classification tasks this index references both the class and the observation index.

```
pd.ind.classif = generatePartialDependenceData(fit.classif,
    iris.task, "Petal.Length", individual = TRUE)
pd.ind.classif
#> PartialDependenceData
#> Task: iris_example
```

```
#> Features: Petal.Length
#> Target: setosa, versicolor, virginica
#> Derivative: FALSE
#> Interaction: FALSE
#> Individual: TRUE
#> Predictions centered: FALSE
#>    Class Probability Petal.Length      idx
#> 1 setosa   0.9814053            1 1.setosa
#> 2 setosa   0.9747355            1 2.setosa
#> 3 setosa   0.9815516            1 3.setosa
#> 4 setosa   0.9795761            1 4.setosa
#> 5 setosa   0.9806494            1 5.setosa
#> 6 setosa   0.9758763            1 6.setosa
#> ... (#rows: 4500, #cols: 4)
```

Individual estimates of partial dependence can also be centered by predictions made at all $n$ observations for a particular point in the prediction grid created by the features. This is controlled by the argument `center` which is a list of the same length as the length of the `features` argument and contains the values of the `features` desired.

```
iris = getTaskData(iris.task)
pd.ind.classif = generatePartialDependenceData(fit.classif,
    iris.task, "Petal.Length", individual = TRUE,
  center = list("Petal.Length" = min(iris$Petal.Length)))
```

Partial derivatives can also be computed for individual partial dependence estimates and aggregate partial dependence. This is restricted to a single feature at a time. The derivatives of individual partial dependence estimates can be useful in finding regions of interaction between the feature for which the derivative is estimated and the features excluded.

```
pd.regr.der = generatePartialDependenceData(fit.regr, bh.task,
    "lstat", derivative = TRUE)
head(pd.regr.der$data)
#>         medv     lstat
#> 1 -0.1792626  1.730000
#> 2 -0.3584207  5.756667
#> 3 -0.4557666  9.783333
#> 4 -0.4523905 13.810000
#> 5 -0.3700880 17.836667
#> 6 -0.2471346 21.863333
```

```
pd.regr.der.ind = generatePartialDependenceData(fit.regr, bh.task,
    "lstat", derivative = TRUE,
  individual = TRUE)
head(pd.regr.der.ind$data)
```

```
#>         medv     lstat idx
#> 1 -0.1931323  1.730000   1
#> 2 -0.2656911  5.756667   1
#> 3 -0.2571006  9.783333   1
#> 4 -0.2033080 13.810000   1
#> 5 -0.1511472 17.836667   1
#> 6 -0.1193129 21.863333   1
```

```
pd.classif.der = generatePartialDependenceData(fit.classif,
    iris.task, "Petal.Width", derivative = TRUE)
head(pd.classif.der$data)
#>    Class Probability Petal.Width
#> 1 setosa  -0.1479385   0.1000000
#> 2 setosa  -0.2422728   0.3666667
#> 3 setosa  -0.2189893   0.6333333
#> 4 setosa  -0.2162803   0.9000000
#> 5 setosa  -0.2768042   1.1666667
#> 6 setosa  -0.2394176   1.4333333
```

```
pd.classif.der.ind = generatePartialDependenceData(fit.classif,
    iris.task, "Petal.Width", derivative = TRUE,
  individual = TRUE)
head(pd.classif.der.ind$data)
#>    Class Probability Petal.Width      idx
#> 1 setosa  0.02479474         0.1 1.setosa
#> 2 setosa  0.01710561         0.1 2.setosa
#> 3 setosa  0.01646252         0.1 3.setosa
#> 4 setosa  0.01530718         0.1 4.setosa
#> 5 setosa  0.02608577         0.1 5.setosa
#> 6 setosa  0.03925531         0.1 6.setosa
```

### 3.13.2 Functional ANOVA

Hooker (2004) proposed the decomposition of a learned function $\hat{f}$ as a sum of lower dimensional functions

$$f(\mathbf{x}) = g_0 + \sum_{i=1}^{p} g_i(X_i) + \sum_{i \neq j} g_{ij}(x_{ij}) + \ldots$$

where $p$ is the number of features. generateFunctionalANOVAData estimates the individual $g$ functions using partial dependence. When functions depend only on one feature, they are equivalent to partial dependence, but a $g$ function which depends on more than one feature is the "effect" of only those features: lower dimensional "effects" are removed.

$$\hat{g}_u(X) = \frac{1}{N} \sum_{i=1}^{N} \left( \hat{f}(X) - \sum_{v \subset u} g_v(X) \right)$$

Here $u$ is a subset of $1, \ldots, p$. When $|v| = 1$ $g_v$ can be directly computed by computing the bivariate partial dependence of $\hat{f}$ on $X_u$ and then subtracting off the univariate partial dependences of the features contained in $v$.

Although this decomposition is generalizable to classification it is currently only available for regression tasks.

```
lrn.regr = makeLearner("regr.ksvm")
fit.regr = train(lrn.regr, bh.task)

fa = generateFunctionalANOVAData(fit.regr, bh.task, "lstat", depth =
    1, fun = median)
fa
#> FunctionalANOVAData
#> Task: BostonHousing-example
#> Features: lstat
#> Target: medv
#>
#>
#>   effect     medv     lstat
#> 1  lstat 24.91250  1.730000
#> 2  lstat 23.73349  5.756667
#> 3  lstat 22.35740  9.783333
#> 4  lstat 20.71107 13.810000
#> 5  lstat 19.62082 17.836667
#> 6  lstat 19.04515 21.863333
#> ... (#rows: 10, #cols: 3)

pd.regr = generatePartialDependenceData(fit.regr, bh.task, "lstat",
    fun = median)
pd.regr
#> PartialDependenceData
#> Task: BostonHousing-example
#> Features: lstat
#> Target: medv
#> Derivative: FALSE
#> Interaction: FALSE
#> Individual: FALSE
#>       medv     lstat
#> 1 24.91250  1.730000
#> 2 23.73349  5.756667
#> 3 22.35740  9.783333
```

```
#> 4 20.71107 13.810000
#> 5 19.62082 17.836667
#> 6 19.04515 21.863333
#> ... (#rows: 10, #cols: 2)
```

The `depth` argument is similar to the `interaction` argument in generatePartialDependenceData but instead of specifying whether all of joint "effect" of all the `features` is computed, it determines whether "effects" of all subsets of the features given the specified `depth` are computed. So, for example, with $p$ features and depth 1, the univariate partial dependence is returned. If, instead, `depth = 2`, then all possible bivariate functional ANOVA effects are returned. This is done by computing the univariate partial dependence for each feature and subtracting it from the bivariate partial dependence for each possible pair.

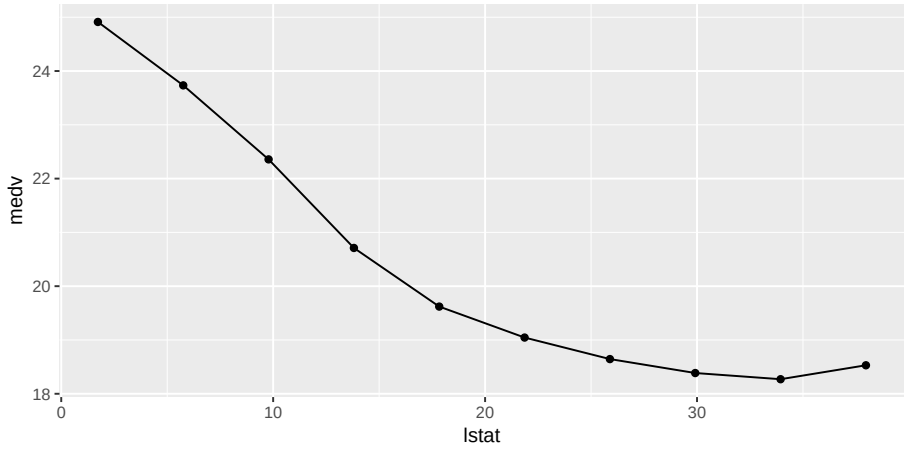
```
fa.bv = generateFunctionalANOVAData(fit.regr, bh.task, c("crim",
    "lstat", "age"),
  depth = 2)
fa.bv
#> FunctionalANOVAData
#> Task: BostonHousing-example
#> Features: crim, lstat, age
#> Target: medv
#>
#>
#>       effect      medv      crim lstat age
#> 1 crim:lstat -22.69831  0.006320  1.73  NA
#> 2 crim:lstat -23.22083  9.891862  1.73  NA
#> 3 crim:lstat -24.84978 19.777404  1.73  NA
#> 4 crim:lstat -26.52861 29.662947  1.73  NA
#> 5 crim:lstat -27.62138 39.548489  1.73  NA
#> 6 crim:lstat -28.21985 49.434031  1.73  NA
#> ... (#rows: 300, #cols: 5)

names(table(fa.bv$data$effect)) ## interaction effects estimated
#> [1] "crim:age"   "crim:lstat" "lstat:age"
```

### 3.13.3 Plotting partial dependences

Results from generatePartialDependenceData and generateFunctionalANOVAData can be visualized with plotPartialDependence and plotPartialDependenceGGVIS.

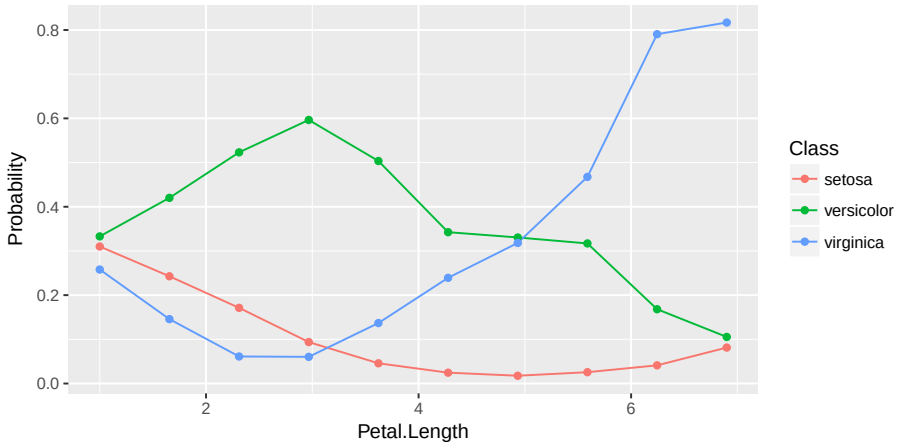
With one feature and a regression task the output is a line plot, with a point for each point in the corresponding feature's grid.

```
plotPartialDependence(pd.regr)
```



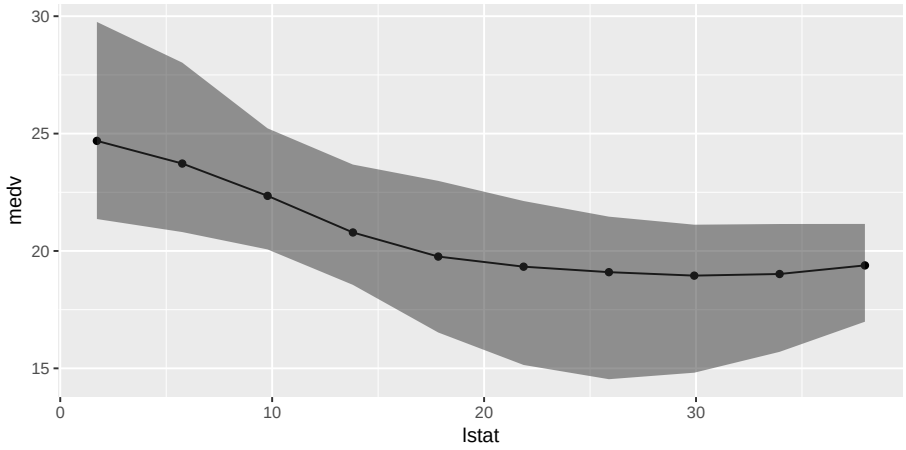

With a classification task, a line is drawn for each class, which gives the estimated partial probability of that class for a particular point in the feature grid.

```
plotPartialDependence(pd.classif)
```



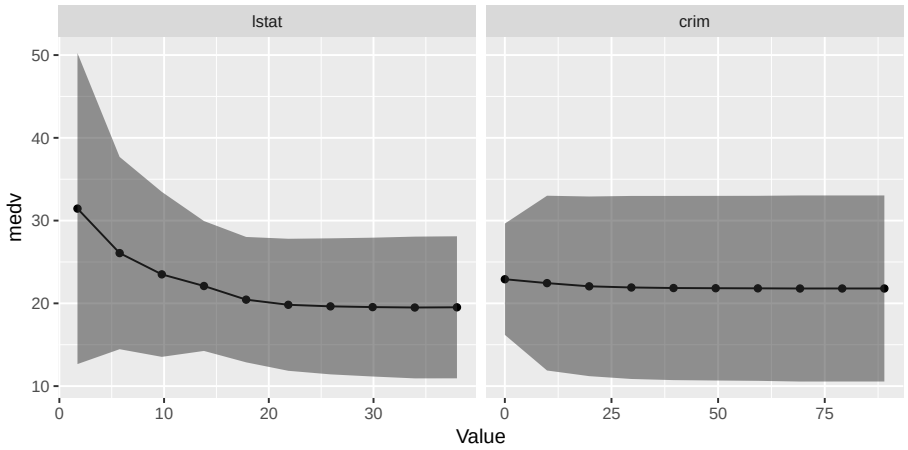

For regression tasks, when the `fun` argument of generatePartialDependenceData is used, the bounds will automatically be displayed using a gray ribbon.

```
plotPartialDependence(pd.ci)
```



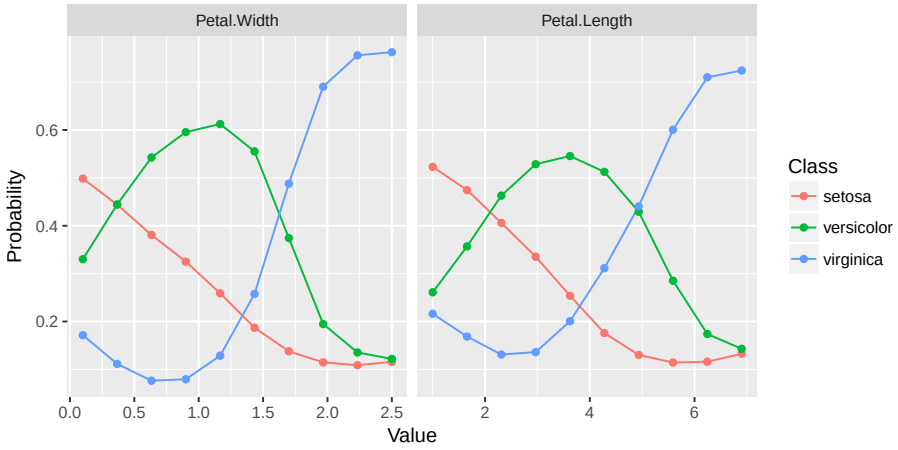

The same goes for plots of partial dependences where the learner has `predict.type = "se"`.

```
plotPartialDependence(pd.se)
```



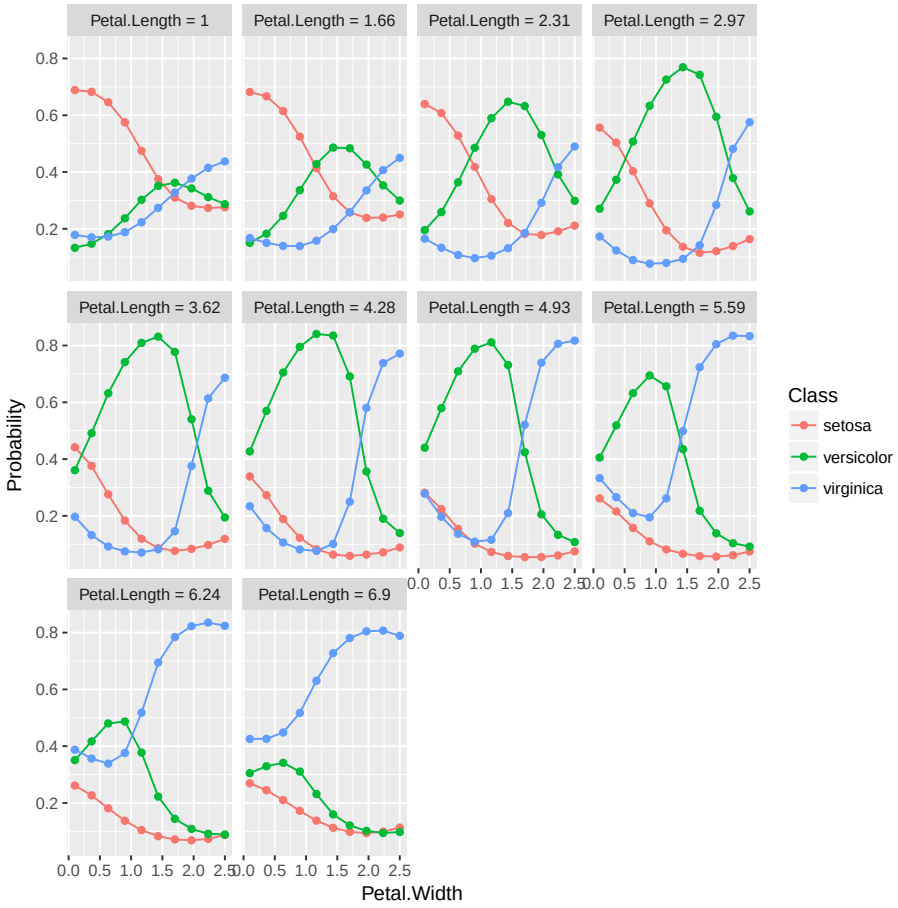

When multiple features are passed to generatePartialDependenceData but `interaction = FALSE`, facetting is used to display each estimated bivariate relationship.

```
plotPartialDependence(pd.lst)
```




When `interaction = TRUE` in the call to generatePartialDependenceData, one variable must be chosen to be used for facetting, and a subplot for each value in the chosen feature's grid is created, wherein the other feature's partial dependences within the facetting feature's value are shown. Note that this type of plot is limited to two features.

```
plotPartialDependence(pd.int, facet = "Petal.Length")
```

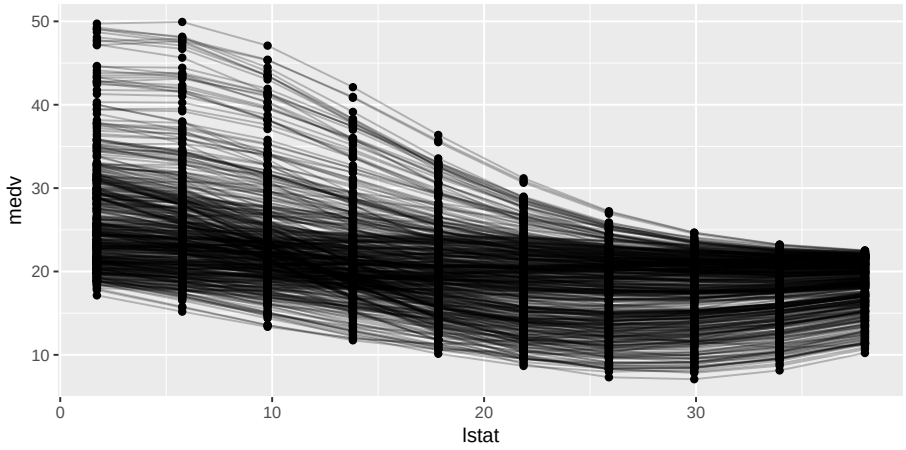



plotPartialDependenceGGVIS can be used similarly, however, since ggvis currently lacks subplotting/facetting capabilities, the argument `interact` maps one feature to an interactive sidebar where the user can select a value of one feature.

```
plotPartialDependenceGGVIS(pd.int, interact = "Petal.Length")
```

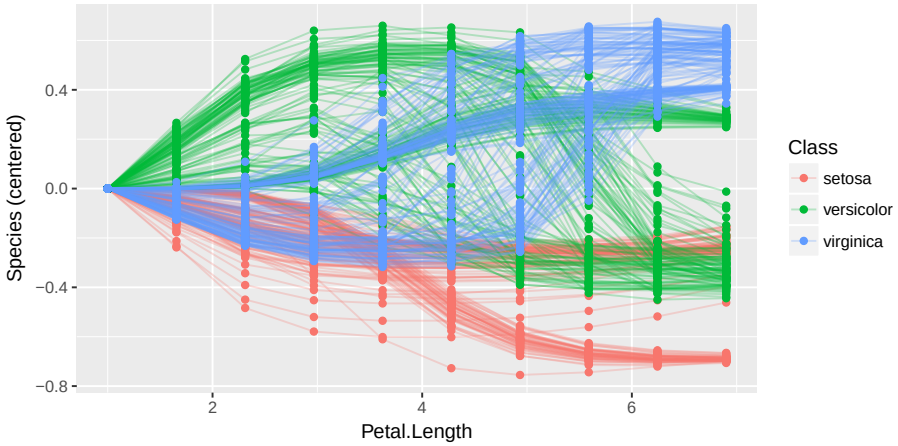
When `individual = TRUE` each individual conditional expectation curve is plotted.

```
plotPartialDependence(pd.ind.regr)
```



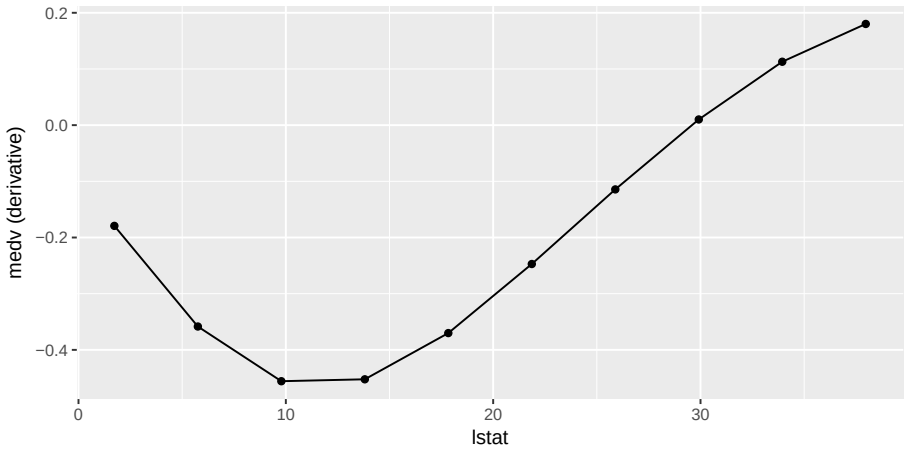

When the individual curves are centered by subtracting the individual conditional expectations estimated at a particular value of $X_s$ this results in a fixed intercept which aids in visualizing variation in predictions made by $\hat{f}_{X_s}^{(i)}$.

```
plotPartialDependence(pd.ind.classif)
```



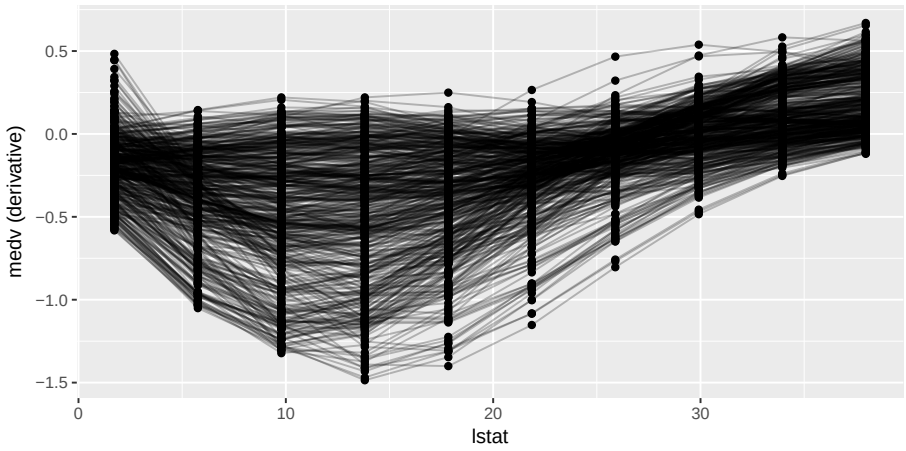

Plotting partial derivative functions works the same as partial dependence. Below are estimates of the derivative of the mean aggregated partial dependence function, and the individual partial dependence functions for a regression and a classification task respectively.

```
plotPartialDependence(pd.regr.der)
```



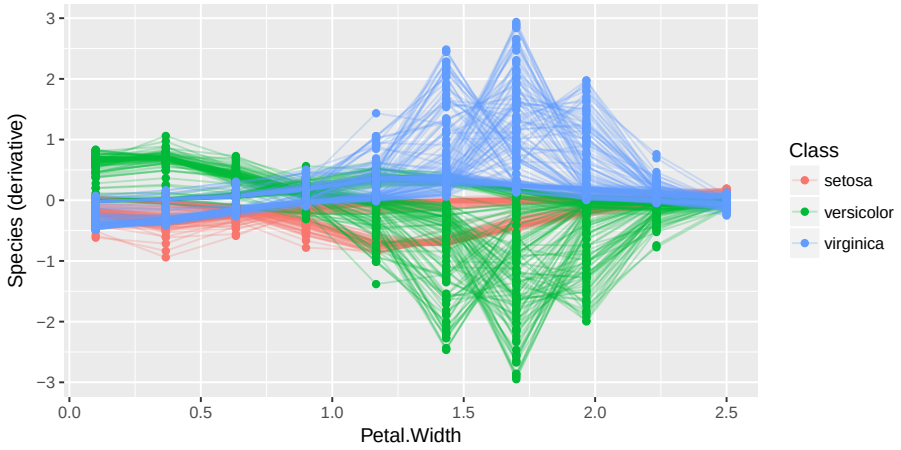

This suggests that $\hat{f}$ is not additive in `lstat` except in the neighborhood of 25.

```
plotPartialDependence(pd.regr.der.ind)
```



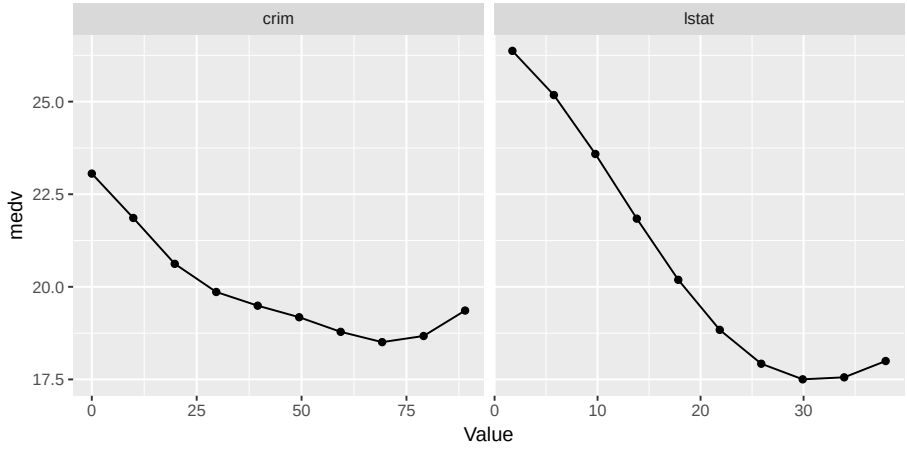

This suggests that `Petal.Width` interacts with some other feature in the neighborhood of $(1.5, 2)$ for classes "virginica" and "versicolor".

```
plotPartialDependence(pd.classif.der.ind)
```



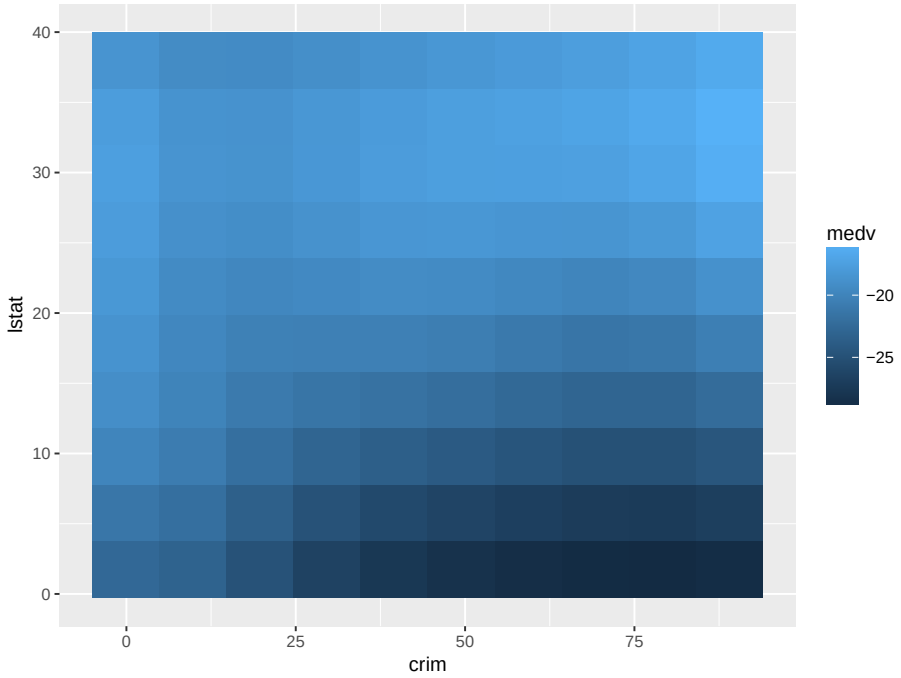

Output from generateFunctionalANOVAData can also be plotted using plotPartialDependence.

```
fa = generateFunctionalANOVAData(fit.regr, bh.task, c("crim",
    "lstat"), depth = 1)
plotPartialDependence(fa)
```




Interactions can often be more easily visualized by using functional ANOVA.

```
fa.bv = generateFunctionalANOVAData(fit.regr, bh.task, c("crim",
    "lstat"), depth = 2)
plotPartialDependence(fa.bv, "tile")
```




## 3.14 Classifier Calibration

A classifier is "calibrated" when the predicted probability of a class matches the expected frequency of that class. mlr can visualize this by plotting estimated class probabilities (which are discretized) against the observed frequency of said class in the data using generateCalibrationData and plotCalibration.

generateCalibrationData takes as input Prediction, ResampleResult, BenchmarkResult, or a named list of Prediction or ResampleResult objects on a classification (multiclass or binary) task with learner(s) that are capable of outputting probabilities (i.e., learners must be constructed with `predict.type = "prob"`). The result is an object of class CalibrationData which has elements `proportion`, `data`, and `task`. `proportion` gives the proportion of observations labelled with a given class for each predicted probability bin (e.g., for observations which are predicted to have class "A" with probability $(0, 0.1]$, what is the proportion of said observations which have class "A"?).

```
lrn = makeLearner("classif.rpart", predict.type = "prob")
mod = train(lrn, task = sonar.task)
pred = predict(mod, task = sonar.task)
cal = generateCalibrationData(pred)
cal$proportion
#>      Learner       bin Class Proportion
```

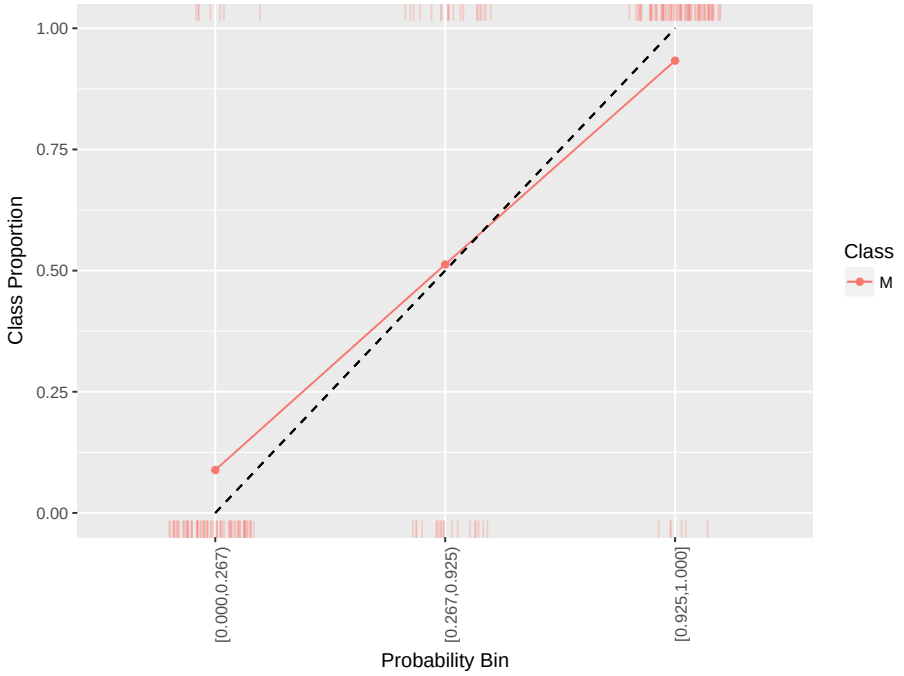
```
#> 1 prediction (0.1,0.2]     M  0.1060606
#> 2 prediction (0.7,0.8]     M  0.7333333
#> 3 prediction   [0,0.1]     M  0.0000000
#> 4 prediction   (0.9,1]     M  0.9333333
#> 5 prediction (0.2,0.3]     M  0.2727273
#> 6 prediction (0.4,0.5]     M  0.4615385
#> 7 prediction (0.8,0.9]     M  0.0000000
#> 8 prediction (0.5,0.6]     M  0.0000000
```

The manner in which the predicted probabilities are discretized is controlled by two arguments: `breaks` and `groups`. By default `breaks = "Sturges"` which uses the Sturges algorithm in hist. This argument can specify other algorithms available in hist, it can be a numeric vector specifying breakpoints for cut, or a single integer specifying the number of bins to create (which are evenly spaced). Alternatively, `groups` can be set to a positive integer value (by default `groups = NULL`) in which case cut2 is used to create bins with an approximately equal number of observations in each bin.

```
cal = generateCalibrationData(pred, groups = 3)
cal$proportion
#>      Learner           bin Class Proportion
#> 1 prediction [0.000,0.267)     M 0.08860759
#> 2 prediction [0.267,0.925)     M 0.51282051
#> 3 prediction [0.925,1.000]     M 0.93333333
```

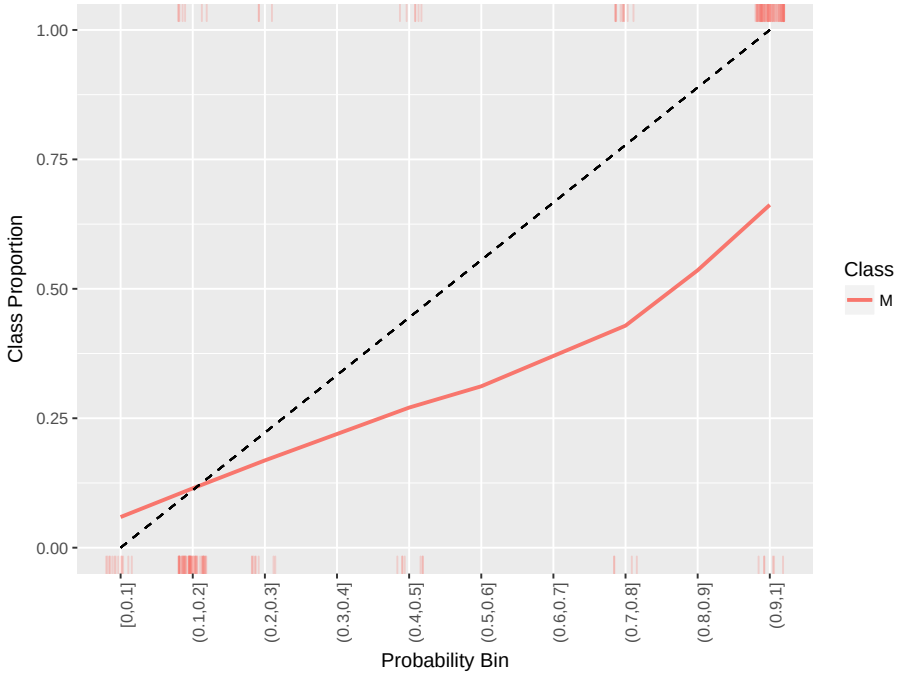
CalibrationData objects can be plotted using plotCalibration. plotCalibration by default plots a reference line which shows perfect calibration and a "rag" plot, which is a rug plot on the top and bottom of the graph, where the top pertains to "positive" cases, where the predicted class matches the observed class, and the bottom pertains to "negative" cases, where the predicted class does not match the observed class. Perfect classifier performance would result in all the positive cases clustering in the top right (i.e., the correct classes are predicted with high probability) and the negative cases clustering in the bottom left.

```
plotCalibration(cal)
```



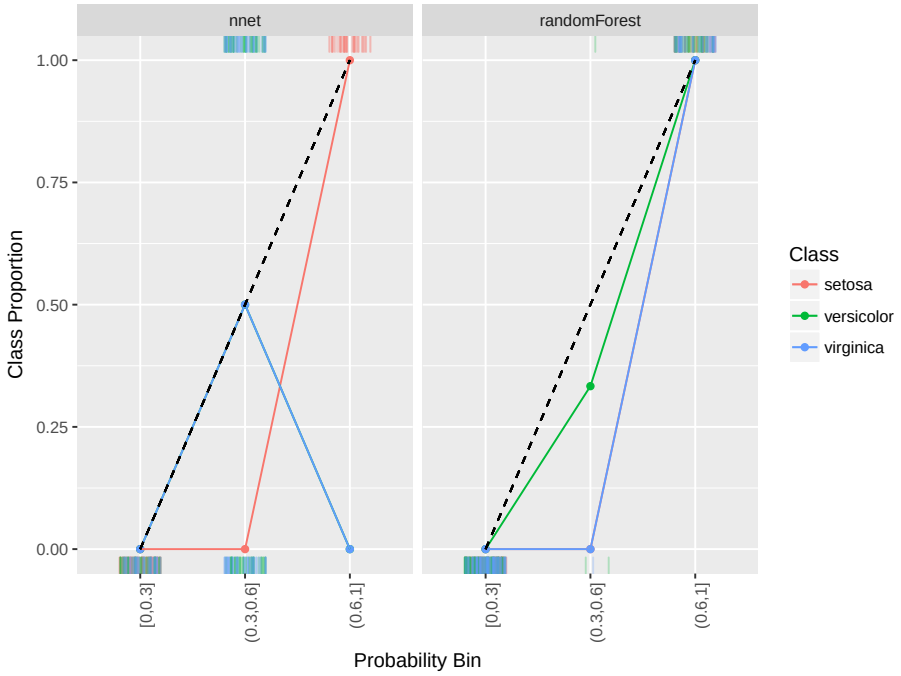

Because of the discretization of the probabilities, sometimes it is advantageous to smooth the calibration plot. Though `smooth = FALSE` by default, setting this option to `TRUE` replaces the estimated proportions with a loess smoother.

```
cal = generateCalibrationData(pred)
plotCalibration(cal, smooth = TRUE)
```




All of the above functionality works with multi-class classification as well.

```
lrns = list(
  makeLearner("classif.randomForest", predict.type = "prob"),
  makeLearner("classif.nnet", predict.type = "prob", trace = FALSE)
)
mod = lapply(lrns, train, task = iris.task)
pred = lapply(mod, predict, task = iris.task)
names(pred) = c("randomForest", "nnet")
cal = generateCalibrationData(pred, breaks = c(0, .3, .6, 1))
plotCalibration(cal)
```




## 3.15 Evaluating Hyperparameter Tuning

As mentioned on the Tuning tutorial page, tuning a machine learning algorithm typically involves:

- the hyperparameter search space:

```
### 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)
)
```

- the optimization algorithm (aka tuning method):

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

- an evaluation method, i.e., a resampling strategy and a performance measure:

```
### ex: 2-fold CV
rdesc = makeResampleDesc("CV", iters = 2L)
```

After tuning, you may want to evaluate the tuning process in order to answer questions such as:

- How does varying the value of a hyperparameter change the performance of the machine learning algorithm?
- What's the relative importance of each hyperparameter?
- How did the optimization algorithm (prematurely) converge?

mlr provides methods to generate and plot the data in order to evaluate the effect of hyperparameter tuning.

### 3.15.1 Generating hyperparameter tuning data

mlr separates the generation of the data from the plotting of the data in case the user wishes to use the data in a custom way downstream.

The generateHyperParsEffectData method takes the tuning result along with 2 additional arguments: `trafo` and `include.diagnostics`. The `trafo` argument will convert the hyperparameter data to be on the transformed scale in case a transformation was used when creating the parameter (as in the case below). The `include.diagnostics` argument will tell mlr whether to include the eol and any error messages from the learner.

Below we perform random search on the `C` parameter for SVM on the famous Pima Indians dataset. We generate the hyperparameter effect data so that the `C` parameter is on the transformed scale and we do not include diagnostic data:

```
ps = makeParamSet(
  makeNumericParam("C", lower = -5, upper = 5, trafo = function(x)
      2^x)
)
ctrl = makeTuneControlRandom(maxit = 100L)
rdesc = makeResampleDesc("CV", iters = 2L)
res = tuneParams("classif.ksvm", task = pid.task, control = ctrl,
  measures = list(acc, mmce), resampling = rdesc, par.set = ps,
      show.info = FALSE)
generateHyperParsEffectData(res, trafo = T, include.diagnostics =
    FALSE)
#> HyperParsEffectData:
#> Hyperparameters: C
#> Measures: acc.test.mean,mmce.test.mean
#> Optimizer: TuneControlRandom
#> Nested CV Used: FALSE
```

```
#> Snapshot of data:
#>            C acc.test.mean mmce.test.mean iteration exec.time
#> 1  0.3770897     0.7695312      0.2304688         1     0.070
#> 2  3.4829323     0.7526042      0.2473958         2     0.070
#> 3  2.2050176     0.7630208      0.2369792         3     0.070
#> 4 24.9285221     0.7070312      0.2929688         4     0.089
#> 5  0.2092395     0.7539062      0.2460938         5     0.072
#> 6  0.1495099     0.7395833      0.2604167         6     0.072
```

As a reminder from the resampling tutorial, if we wanted to generate data on the training set as well as the validation set, we only need to make a few minor changes:

```
ps = makeParamSet(
  makeNumericParam("C", lower = -5, upper = 5, trafo = function(x)
      2^x)
)
ctrl = makeTuneControlRandom(maxit = 100L)
rdesc = makeResampleDesc("CV", iters = 2L, predict = "both")
res = tuneParams("classif.ksvm", task = pid.task, control = ctrl,
  measures = list(acc, setAggregation(acc, train.mean), mmce,
      setAggregation(mmce,
    train.mean)), resampling = rdesc, par.set = ps, show.info =
       FALSE)
generateHyperParsEffectData(res, trafo = T, include.diagnostics =
    FALSE)
#> HyperParsEffectData:
#> Hyperparameters: C
#> Measures:
    acc.test.mean,acc.train.mean,mmce.test.mean,mmce.train.mean
#> Optimizer: TuneControlRandom
#> Nested CV Used: FALSE
#> Snapshot of data:
#>            C acc.test.mean acc.train.mean mmce.test.mean
    mmce.train.mean
#> 1 0.03518875     0.6510417      0.6510417      0.3489583
    0.3489583
#> 2 0.17104229     0.7356771      0.7721354      0.2643229
    0.2278646
#> 3 4.35326556     0.7304688      0.8828125      0.2695312
    0.1171875
#> 4 0.33644238     0.7486979      0.8138021      0.2513021
    0.1861979
#> 5 1.28168692     0.7500000      0.8476562      0.2500000
    0.1523438
#> 6 7.36607693     0.7239583      0.8932292      0.2760417
```

```
    0.1067708
#>   iteration exec.time
#> 1         1     0.103
#> 2         2     0.101
#> 3         3     0.099
#> 4         4     0.109
#> 5         5     0.100
#> 6         6     0.099
```

In the example below, we perform grid search on the `C` parameter for SVM on the Pima Indians dataset using nested cross validation. We generate the hyperparameter effect data so that the `C` parameter is on the untransformed scale and we do not include diagnostic data. As you can see below, nested cross validation is supported without any extra work by the user, allowing the user to obtain an unbiased estimator for the performance.

```
ps = makeParamSet(
  makeNumericParam("C", lower = -5, upper = 5, trafo = function(x)
      2^x)
)
ctrl = makeTuneControlGrid()
rdesc = makeResampleDesc("CV", iters = 2L)
lrn = makeTuneWrapper("classif.ksvm", control = ctrl,
  measures = list(acc, mmce), resampling = rdesc, par.set = ps,
      show.info = FALSE)
res = resample(lrn, task = pid.task, resampling = cv2, extract =
    getTuneResult, show.info = FALSE)
generateHyperParsEffectData(res)
#> HyperParsEffectData:
#> Hyperparameters: C
#> Measures: acc.test.mean,mmce.test.mean
#> Optimizer: TuneControlGrid
#> Nested CV Used: TRUE
#> Snapshot of data:
#>            C acc.test.mean mmce.test.mean iteration exec.time
#> 1 -5.0000000     0.6640625      0.3359375         1     0.050
#> 2 -3.8888889     0.6640625      0.3359375         2     0.050
#> 3 -2.7777778     0.6822917      0.3177083         3     0.049
#> 4 -1.6666667     0.7473958      0.2526042         4     0.050
#> 5 -0.5555556     0.7708333      0.2291667         5     0.058
#> 6  0.5555556     0.7682292      0.2317708         6     0.052
#>   nested_cv_run
#> 1             1
#> 2             1
#> 3             1
#> 4             1
```

```
#> 5             1
#> 6             1
```

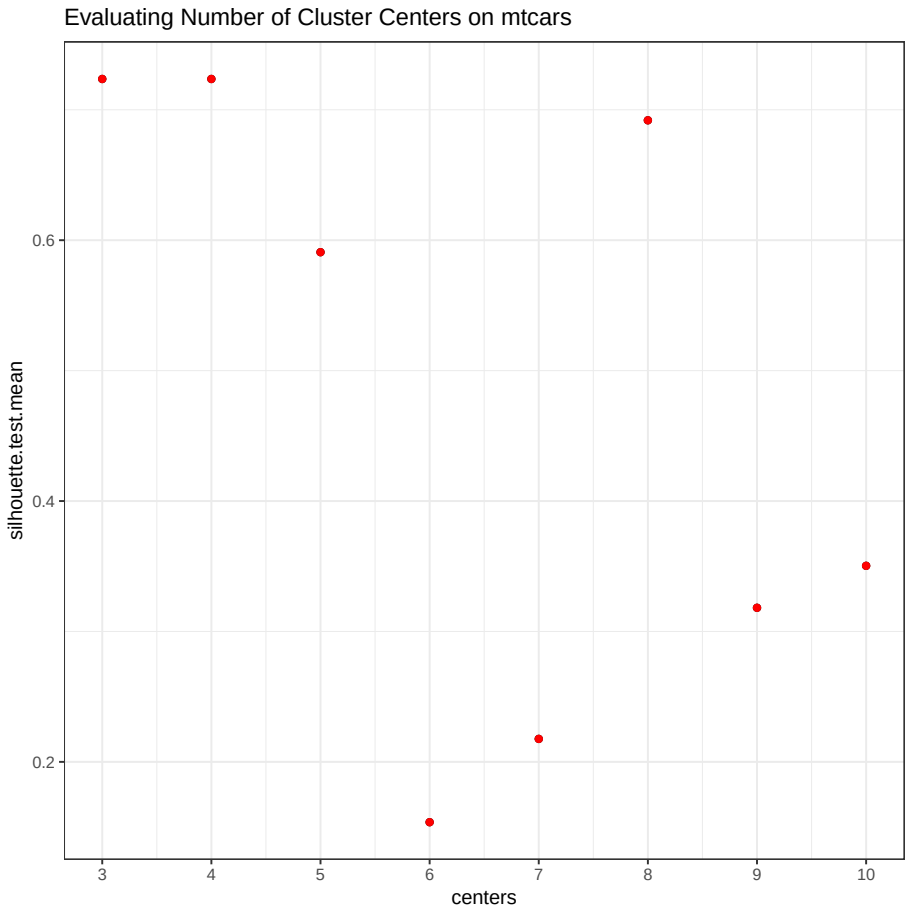
After generating the hyperparameter effect data, the next step is to visualize it. mlr has several methods built-in to visualize the data, meant to support the needs of the researcher and the engineer in industry. The next few sections will walk through the visualization support for several use-cases.

### 3.15.2 Visualizing the effect of a single hyperparameter

In a situation when the user is tuning a single hyperparameter for a learner, the user may wish to plot the performance of the learner against the values of the hyperparameter.

In the example below, we tune the number of clusters against the silhouette score on the Pima dataset. We specify the x-axis with the `x` argument and the y-axis with the `y` argument. If the `plot.type` argument is not specified, mlr will attempt to plot a scatterplot by default. Since plotHyperParsEffect returns a ggplot object, we can easily customize it to our liking!

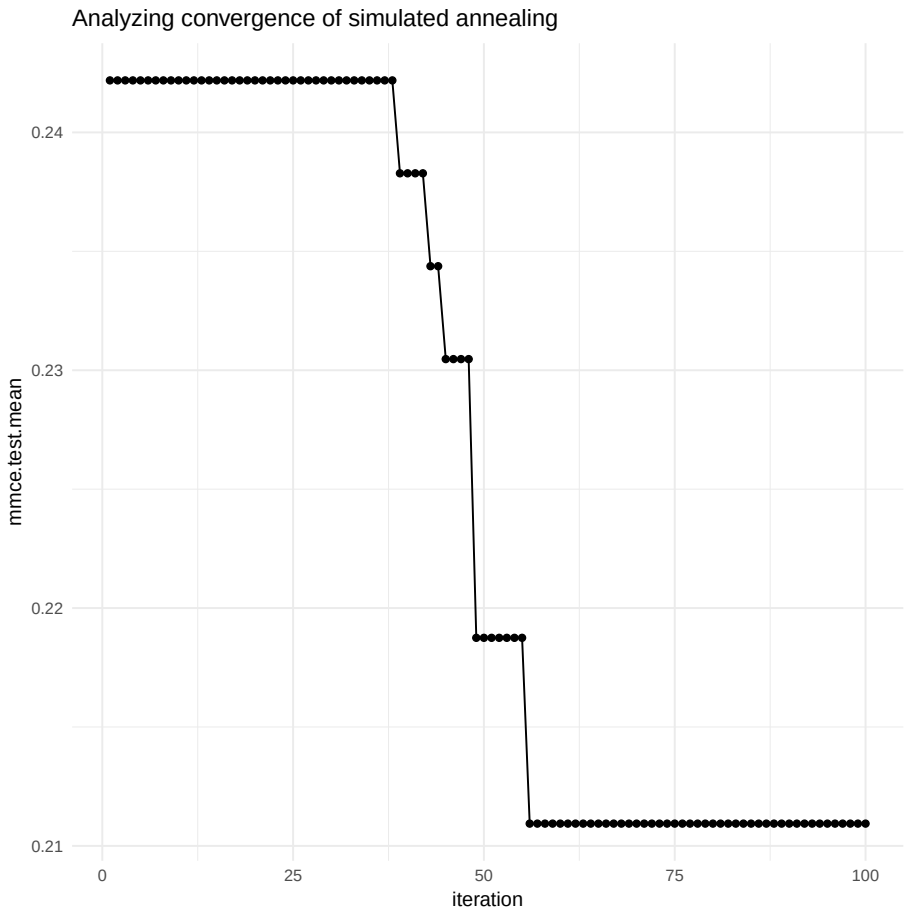
```
ps = makeParamSet(
  makeDiscreteParam("centers", values = 3:10)
)
ctrl = makeTuneControlGrid()
rdesc = makeResampleDesc("Holdout")
res = tuneParams("cluster.kmeans", task = mtcars.task, control =
    ctrl,
  measures = silhouette, resampling = rdesc, par.set = ps, show.info
      = FALSE)
#>
#> This is package 'modeest' written by P. PONCET.
#> For a complete list of functions, use 'library(help = "modeest")'
    or 'help.start()'.
data = generateHyperParsEffectData(res)
plt = plotHyperParsEffect(data, x = "centers", y =
    "silhouette.test.mean")
### add our own touches to the plot
plt + geom_point(colour = "red") +
  ggtitle("Evaluating Number of Cluster Centers on mtcars") +
  scale_x_continuous(breaks = 3:10) +
  theme_bw()
```




In the example below, we tune SVM with the `C` hyperparameter on the Pima dataset. We will use simulated annealing optimizer, so we are interested in seeing if the optimization algorithm actually improves with iterations. By default, mlr only plots improvements to the global optimum.

```
ps = makeParamSet(
  makeNumericParam("C", lower = -5, upper = 5, trafo = function(x)
      2^x)
)
ctrl = makeTuneControlGenSA(budget = 100L)
rdesc = makeResampleDesc("Holdout")
res = tuneParams("classif.ksvm", task = pid.task, control = ctrl,
  resampling = rdesc, par.set = ps, show.info = FALSE)
data = generateHyperParsEffectData(res)
plt = plotHyperParsEffect(data, x = "iteration", y =
    "mmce.test.mean",
  plot.type = "line")
```

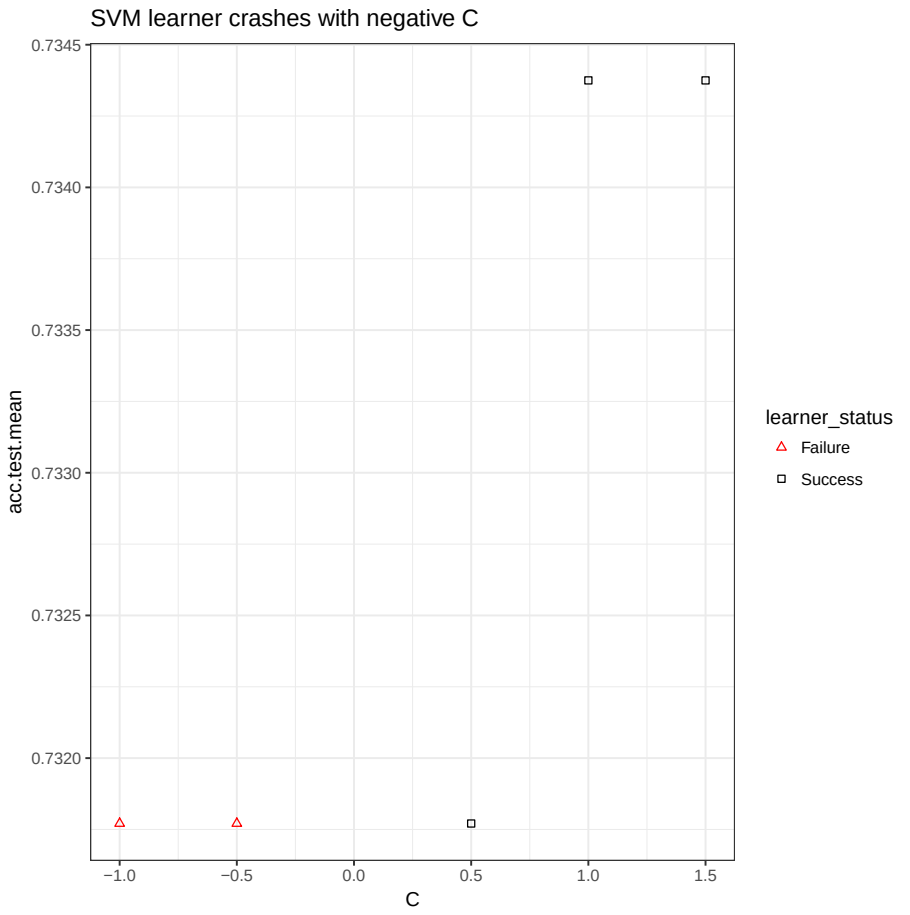
```
plt + ggtitle("Analyzing convergence of simulated annealing") +
  theme_minimal()
```




In the case of a learner crash, mlr will impute the crash with the worst value graphically and indicate the point. In the example below, we give the `C` parameter negative values, which will result in a learner crash for SVM.

```
ps = makeParamSet(
  makeDiscreteParam("C", values = c(-1, -0.5, 0.5, 1, 1.5))
)
ctrl = makeTuneControlGrid()
rdesc = makeResampleDesc("CV", iters = 2L)
res = tuneParams("classif.ksvm", task = pid.task, control = ctrl,
  measures = list(acc, mmce), resampling = rdesc, par.set = ps,
      show.info = FALSE)
data = generateHyperParsEffectData(res)
plt = plotHyperParsEffect(data, x = "C", y = "acc.test.mean")
```

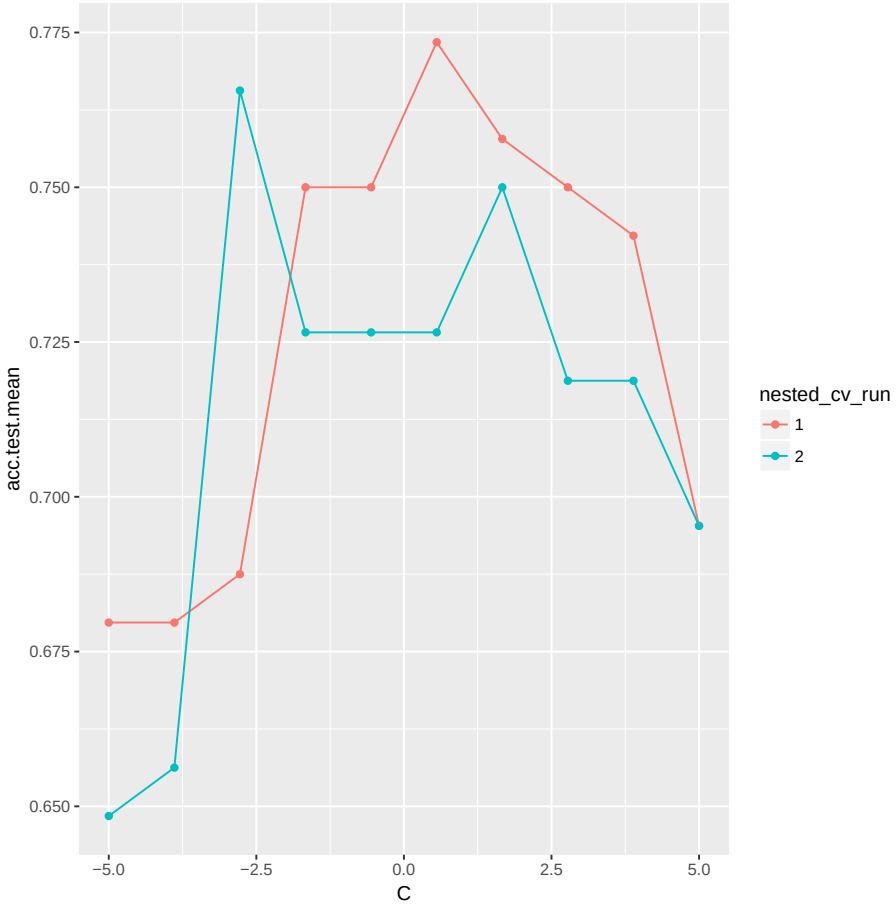
```
plt + ggtitle("SVM learner crashes with negative C") +
  theme_bw()
```




The example below uses nested cross validation with an outer loop of 2 runs. mlr indicates each run within the visualization.

```
ps = makeParamSet(
  makeNumericParam("C", lower = -5, upper = 5, trafo = function(x)
      2^x)
)
ctrl = makeTuneControlGrid()
rdesc = makeResampleDesc("Holdout")
lrn = makeTuneWrapper("classif.ksvm", control = ctrl,
  measures = list(acc, mmce), resampling = rdesc, par.set = ps,
      show.info = FALSE)
res = resample(lrn, task = pid.task, resampling = cv2, extract =
    getTuneResult, show.info = FALSE)
```

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



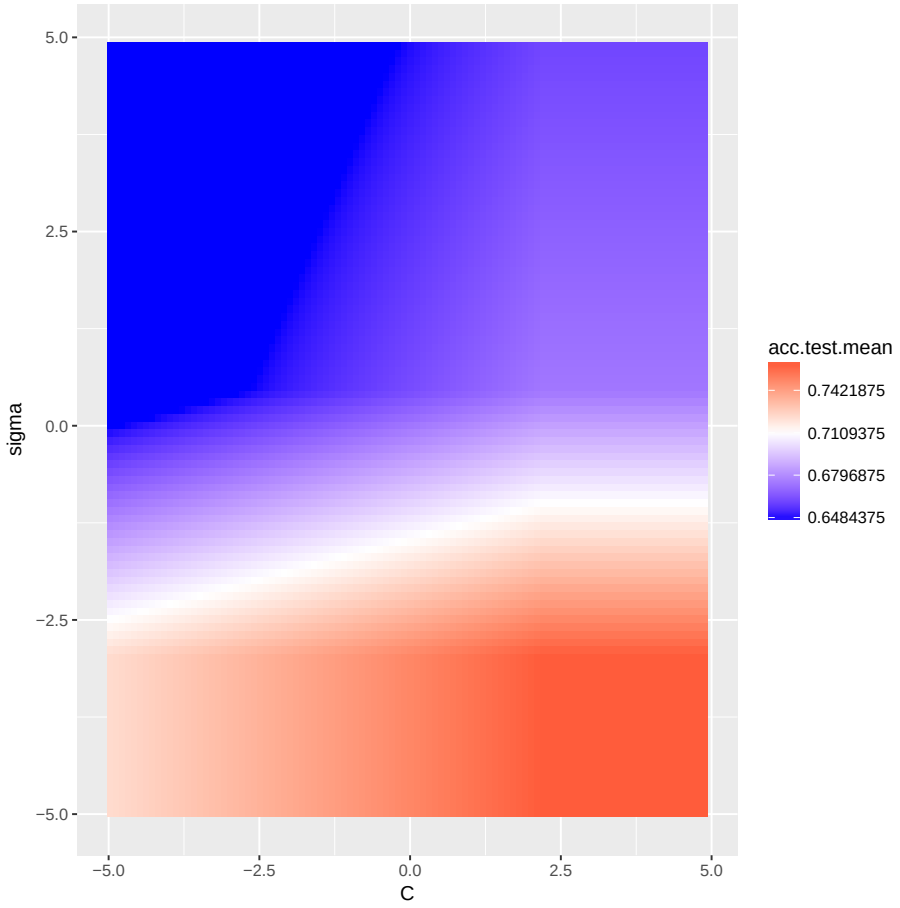

### 3.15.3 Visualizing the effect of 2 hyperparameters

In the case of tuning 2 hyperparameters simultaneously, mlr provides the ability to plot a heatmap and contour plot in addition to a scatterplot or line.

In the example below, we tune the `C` and `sigma` parameters for SVM on the Pima dataset. We use interpolation to produce a regular grid for plotting the heatmap. The `interpolation` argument accepts any regression learner from mlr to perform the interpolation. The `z` argument will be used to fill the heatmap or color lines, depending on the `plot.type` used.

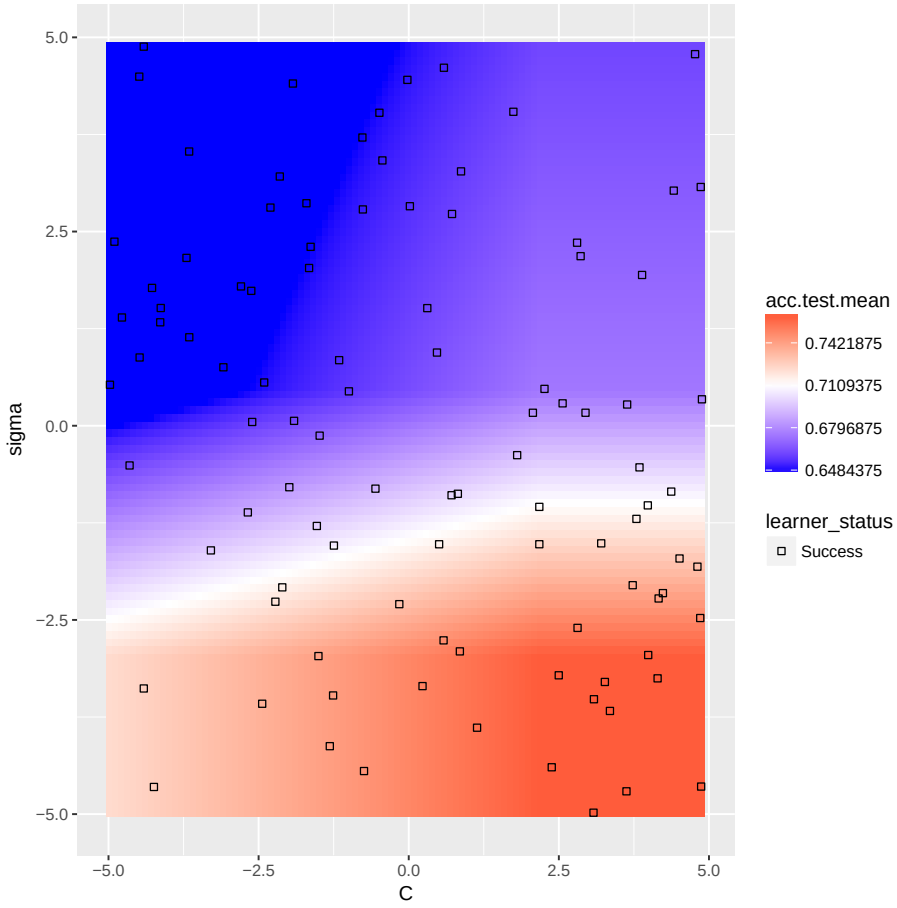
```
ps = makeParamSet(
```

```
  makeNumericParam("C", lower = -5, upper = 5, trafo = function(x)
      2^x),
  makeNumericParam("sigma", lower = -5, upper = 5, trafo =
      function(x) 2^x))
ctrl = makeTuneControlRandom(maxit = 100L)
rdesc = makeResampleDesc("Holdout")
learn = makeLearner("classif.ksvm", par.vals = list(kernel =
    "rbfdot"))
res = tuneParams(learn, task = pid.task, control = ctrl, measures =
    acc,
  resampling = rdesc, par.set = ps, show.info = FALSE)
data = generateHyperParsEffectData(res)
plt = plotHyperParsEffect(data, x = "C", y = "sigma", z =
    "acc.test.mean",
  plot.type = "heatmap", interpolate = "regr.earth")
min_plt = min(data$data$acc.test.mean, na.rm = TRUE)
max_plt = max(data$data$acc.test.mean, na.rm = TRUE)
med_plt = mean(c(min_plt, max_plt))
plt + scale_fill_gradient2(breaks = seq(min_plt, max_plt, length.out
    = 5),
  low = "blue", mid = "white", high = "red", midpoint = med_plt)
```



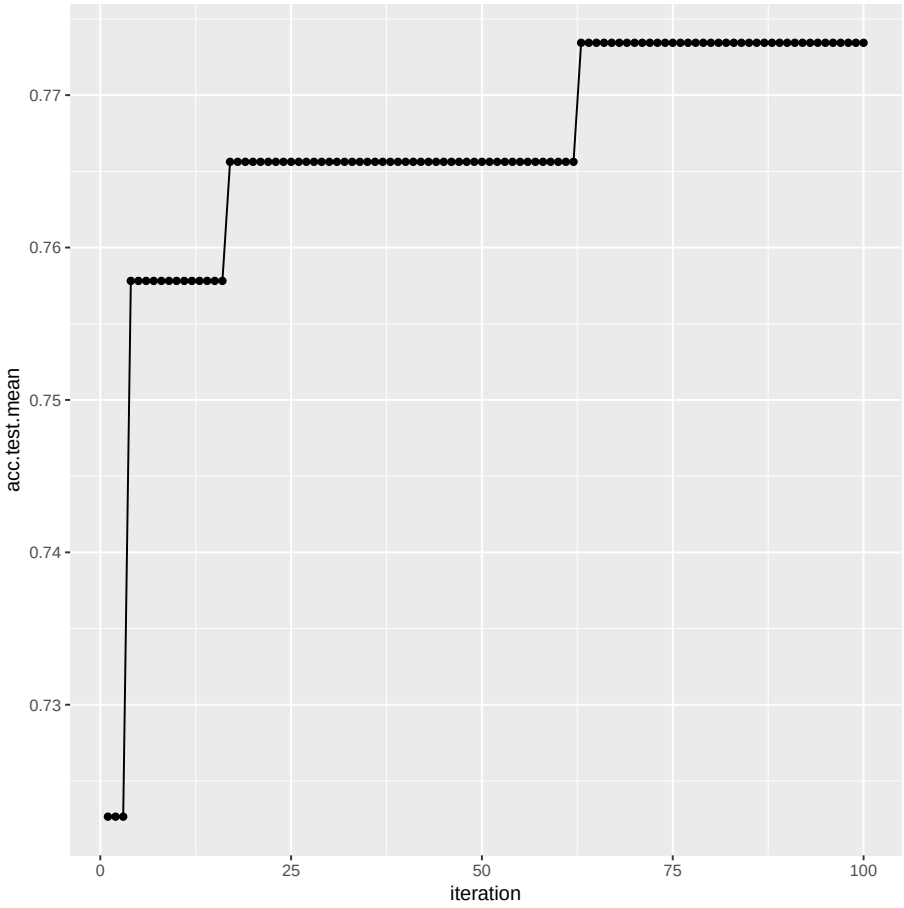

We can use the `show.experiments` argument in order to visualize which points were specifically passed to the learner in the original experiment and which points were interpolated by mlr:

```
plt = plotHyperParsEffect(data, x = "C", y = "sigma", z =
    "acc.test.mean",
  plot.type = "heatmap", interpolate = "regr.earth",
      show.experiments = TRUE)
plt + scale_fill_gradient2(breaks = seq(min_plt, max_plt, length.out
    = 5),
  low = "blue", mid = "white", high = "red", midpoint = med_plt)
```




We can also visualize how long the optimizer takes to reach an optima for the same example:

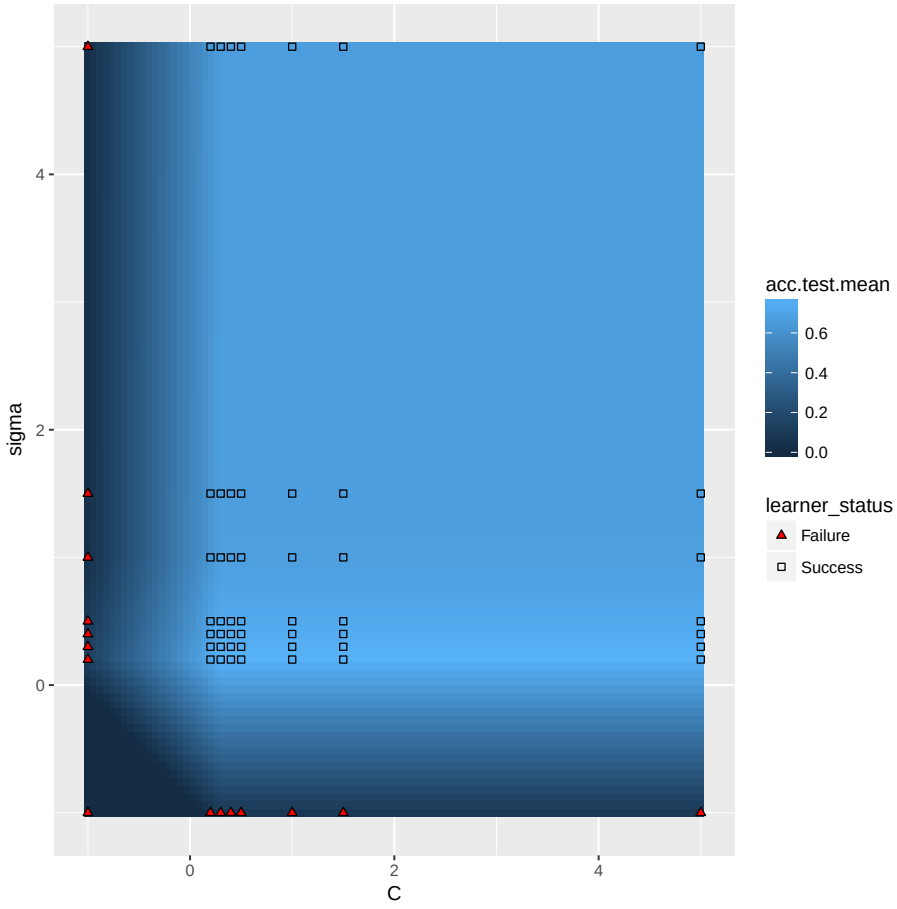
```
plotHyperParsEffect(data, x = "iteration", y = "acc.test.mean",
  plot.type = "line")
```




In the case where we are tuning 2 hyperparameters and we have a learner crash, mlr will indicate the respective points and impute them with the worst value. In the example below, we tune `C` and `sigma`, forcing `C` to be negative for some instances which will crash SVM. We perform interpolation to get a regular grid in order to plot a heatmap. We can see that the interpolation creates axis parallel lines resulting from the learner crashes.

```
ps = makeParamSet(
  makeDiscreteParam("C", values = c(-1, 0.5, 1.5, 1, 0.2, 0.3, 0.4,
      5)),
  makeDiscreteParam("sigma", values = c(-1, 0.5, 1.5, 1, 0.2, 0.3,
      0.4, 5)))
ctrl = makeTuneControlGrid()
rdesc = makeResampleDesc("Holdout")
learn = makeLearner("classif.ksvm", par.vals = list(kernel =
    "rbfdot"))
res = tuneParams(learn, task = pid.task, control = ctrl, measures =
```

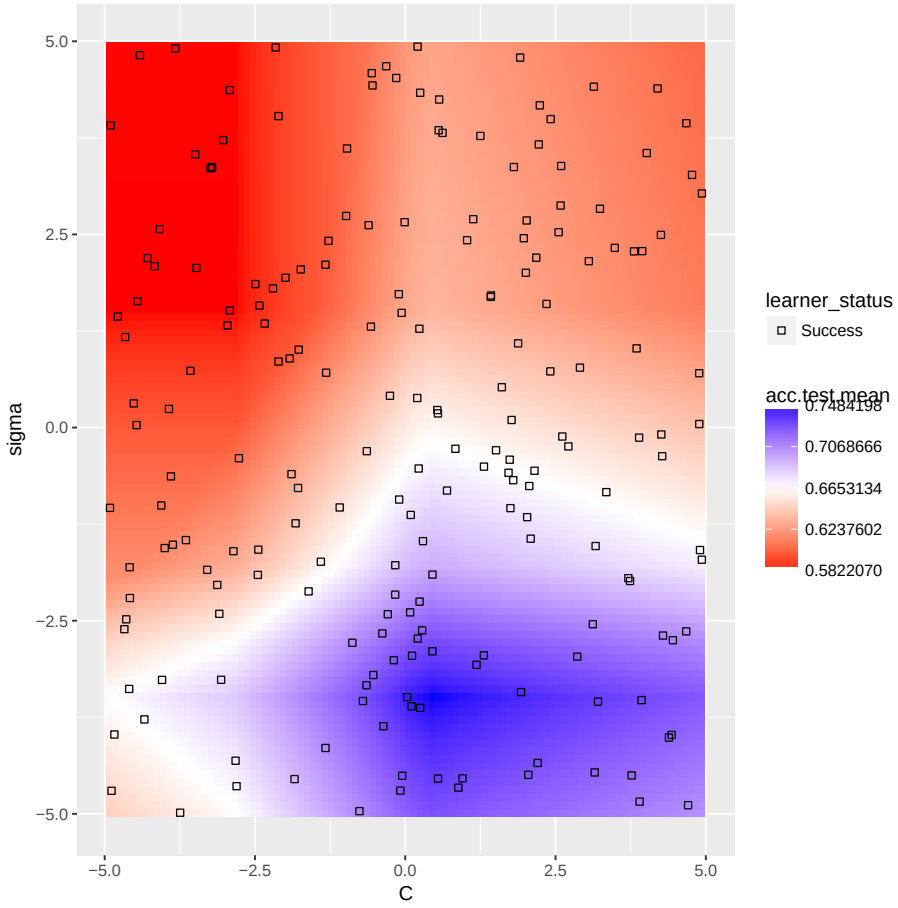
```
    acc,
  resampling = rdesc, par.set = ps, show.info = FALSE)
data = generateHyperParsEffectData(res)
plotHyperParsEffect(data, x = "C", y = "sigma", z = "acc.test.mean",
  plot.type = "heatmap", interpolate = "regr.earth")
```




A slightly more complicated example is using nested cross validation while simultaneously tuning 2 hyperparameters. In order to plot a heatmap in this case, mlr will aggregate each of the nested runs by a user-specified function. The default function is `mean`. As expected, we can still take advantage of interpolation.

```
ps = makeParamSet(
  makeNumericParam("C", lower = -5, upper = 5, trafo = function(x)
      2^x),
  makeNumericParam("sigma", lower = -5, upper = 5, trafo =
      function(x) 2^x))
ctrl = makeTuneControlRandom(maxit = 100)
```

```
rdesc = makeResampleDesc("Holdout")
learn = makeLearner("classif.ksvm", par.vals = list(kernel =
    "rbfdot"))
lrn = makeTuneWrapper(learn, control = ctrl, measures = list(acc,
    mmce),
  resampling = rdesc, par.set = ps, show.info = FALSE)
res = resample(lrn, task = pid.task, resampling = cv2, extract =
    getTuneResult, show.info = FALSE)
data = generateHyperParsEffectData(res)
plt = plotHyperParsEffect(data, x = "C", y = "sigma", z =
    "acc.test.mean",
  plot.type = "heatmap", interpolate = "regr.earth",
      show.experiments = TRUE,
  nested.agg = mean)
min_plt = min(plt$data$acc.test.mean, na.rm = TRUE)
max_plt = max(plt$data$acc.test.mean, na.rm = TRUE)
med_plt = mean(c(min_plt, max_plt))
plt + scale_fill_gradient2(breaks = seq(min_plt, max_plt, length.out
    = 5),
  low = "red", mid = "white", high = "blue", midpoint = med_plt)
```




### 3.15.4 Visualizing the effects of more than 2 hyperparameters

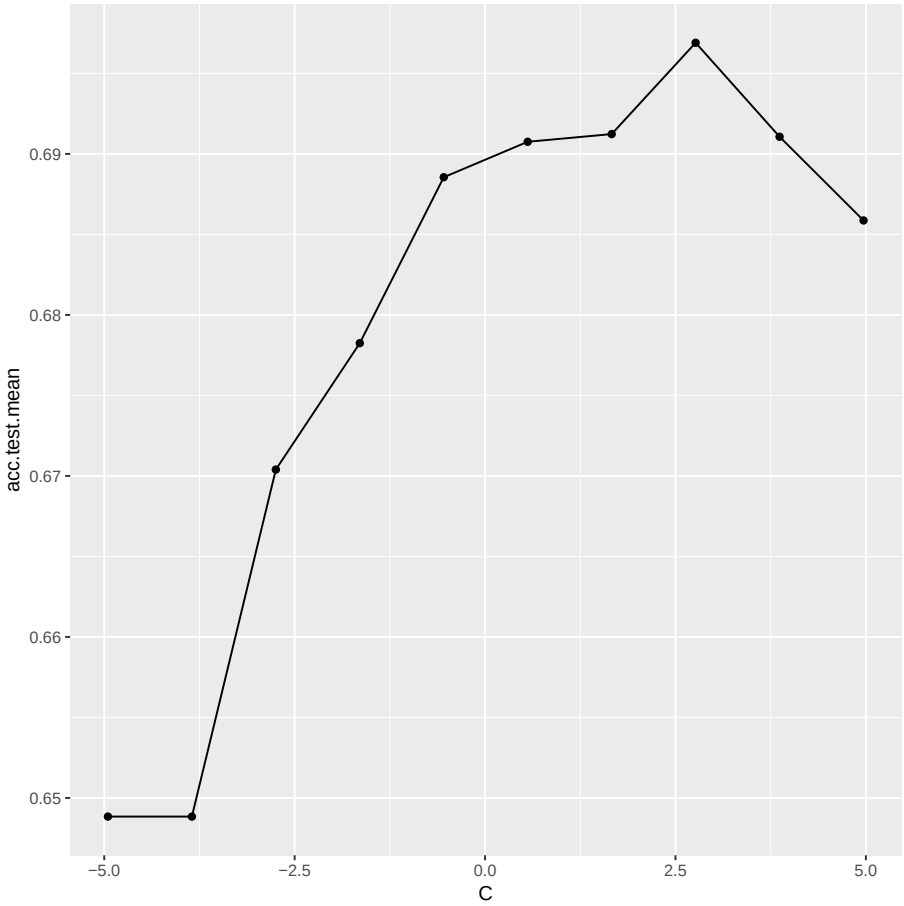
In order to visualize the result when tuning 3 or more hyperparameters simultaneously we can take advantage of partial dependence plots to show how the performance depends on a one- or two-dimensional subset of the hyperparameters. Below we tune three hyperparameters `C`, `sigma`, and `degree` of an SVM with Bessel kernel and set the `partial.dep` flag to `TRUE` to indicate that we intend to calculate partial dependences.

```
ps = makeParamSet(
  makeNumericParam("C", lower = -5, upper = 5, trafo = function(x)
      2^x),
  makeNumericParam("sigma", lower = -5, upper = 5, trafo =
      function(x) 2^x),
  makeDiscreteParam("degree", values = 2:5))
ctrl = makeTuneControlRandom(maxit = 100L)
```

```
rdesc = makeResampleDesc("Holdout", predict = "both")
learn = makeLearner("classif.ksvm", par.vals = list(kernel =
    "besseldot"))
res = tuneParams(learn, task = pid.task, control = ctrl,
  measures = list(acc, setAggregation(acc, train.mean)), resampling
      = rdesc,
  par.set = ps, show.info = FALSE)
data = generateHyperParsEffectData(res, partial.dep = TRUE)
```

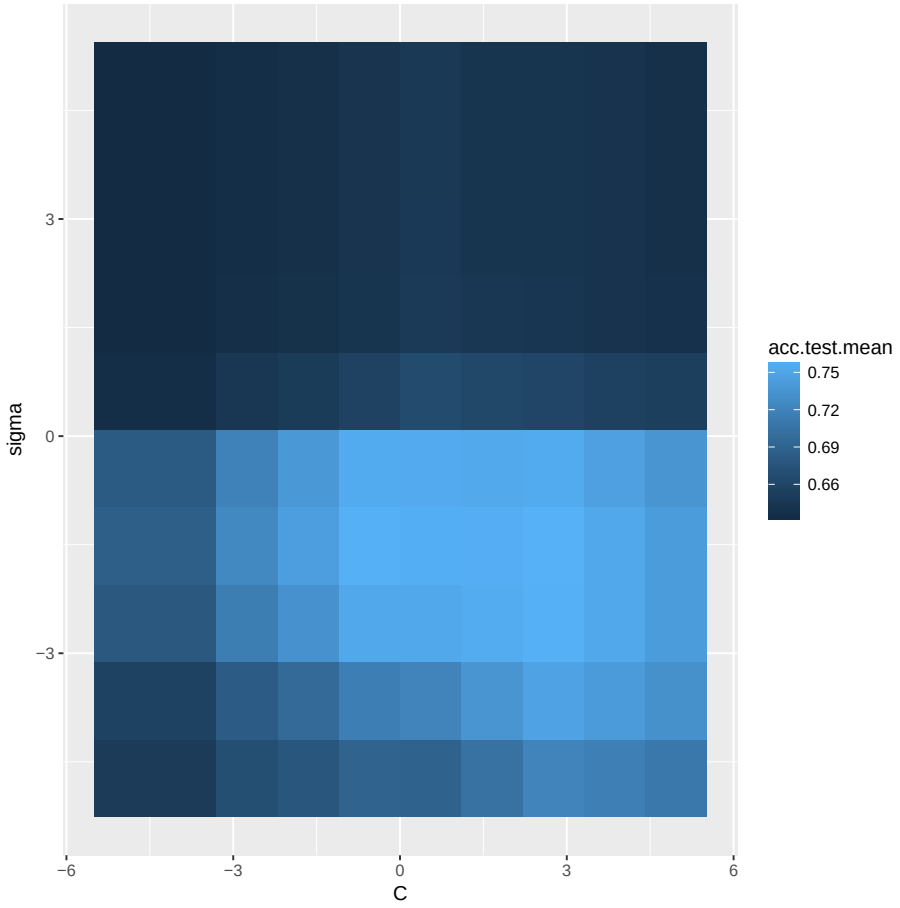
You can generate a plot for a single hyperparameter like `C` as shown below. The `partial.dep.learn` can be any regression Learner in mlr and is used to regress the attained performance values on the values of the 3 hyperparameters visited during tuning. The fitted model serves as basis for calculating partial dependences.

```
plotHyperParsEffect(data, x = "C", y = "acc.test.mean", plot.type =
    "line",
  partial.dep.learn = "regr.randomForest")
```




We can also look at two hyperparameters simultaneously, for example `C` and `sigma`.

```
plotHyperParsEffect(data, x = "C", y = "sigma", z = "acc.test.mean",
  plot.type = "heatmap", partial.dep.learn = "regr.randomForest")
```




# 4 Extend

## 4.1 Out-of-Bag Predictions

Some learners like random forest use bagging. Bagging means that the learner consists of an ensemble of several base learners and each base learner is trained with a different random subsample or bootstrap sample from all observations. A prediction made for an observation in the original data set using only base learners not trained on this particular observation is called out-of-bag (OOB) prediction. These predictions are not prone to overfitting, as each prediction is only made by learners that did not use the observation for training.

To get a list of learners that provide OOB predictions, you can call `listLearners(obj = NA, properties = "oobpreds")`.

```
listLearners(obj = NA, properties = "oobpreds")[c("class",
    "package")]
#>                     class         package
#> 1    classif.randomForest    randomForest
#> 2 classif.randomForestSRC randomForestSRC
#> 3          classif.ranger          ranger
#> 4          classif.rFerns          rFerns
#> 5       regr.randomForest    randomForest
#> 6    regr.randomForestSRC randomForestSRC
#> ... (#rows: 8, #cols: 2)
```

In mlr function getOOBPreds can be used to extract these observations from the trained models. These predictions can be used to evaluate the performance of a given learner like in the following example.

```
lrn = makeLearner("classif.ranger", predict.type = "prob",
    predict.threshold = 0.6)
mod = train(lrn, sonar.task)
oob = getOOBPreds(mod, sonar.task)
oob
#> Prediction: 208 observations
#> predict.type: prob
#> threshold: M=0.60,R=0.40
#> time: NA
#>   id truth    prob.M    prob.R response
#> 1  1     R 0.5373385 0.4626615        R
#> 2  2     R 0.5971972 0.4028028        R
#> 3  3     R 0.5626560 0.4373440        R
#> 4  4     R 0.4319901 0.5680099        R
#> 5  5     R 0.5417589 0.4582411        R
#> 6  6     R 0.4005787 0.5994213        R
#> ... (#rows: 208, #cols: 5)

performance(oob, measures = list(auc, mmce))
#>       auc      mmce
#> 0.9308071 0.1778846
```

As the predictions that are used are out-of-bag, this evaluation strategy is very similar to common resampling strategies like 10-fold cross-validation, but much faster, as only one training instance of the model is required.

## 4.2 Functional Data

Functional data provides information about curves varying over a continuum, such as time. This type of data is often present when analyzing measurements

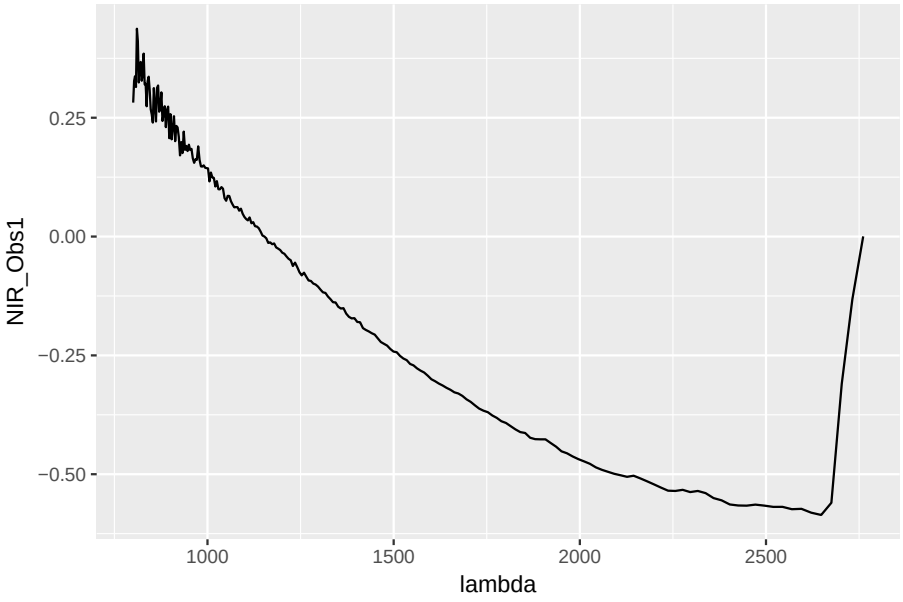
at various time points. Such curves usually are interdependent, which means that the measurement at a point $t_{i+1}$ usually depends on some measurements $t_1, ..., t_i; i \in \mathbb{N}$.

As traditional machine learning techniques usually do not emphasize the interdependence between features, they are often not *well suited* for such tasks, which can lead to poor performance. Functional data analysis on the other hand tries to address this by either using algorithms specifically tailored to functional data, or by transforming the functional covariates into a non time-dependent feature space. For a more in depth introduction to functional data analysis see e.g When the data are functions Ramsay, J.O., 1982.

Each observation of a functional covariate in the data are evaluations of a functional, i.e. measurements of a scalar value at various time points. A single observation might then look like this:

```
## Plot NIR curve for first observarion
library(FDboost)
#> Loading required package: mboost
#> Loading required package: stabs
#>
#> Attaching package: 'stabs'
#> The following object is masked from 'package:mlr':
#>
#>     subsample
#> This is mboost 2.8-1. See 'package?mboost' and 'news(package  =
    "mboost")'
#> for a complete list of changes.
#>
#> Attaching package: 'mboost'
#> The following object is masked from 'package:ggplot2':
#>
#>     %+%
#> This is FDboost 0.3-0.
#>
#> Attaching package: 'FDboost'
#> The following object is masked from 'package:mlr':
#>
#>     plotResiduals
data(fuelSubset)
library(ggplot2)
## NIR_Obs_1 are the measurements for NIR of the first functional
    covariate.
## lambda are the time points, the data was measured at.
df = data.frame("NIR_Obs1" = fuelSubset$NIR[1, ],
                "lambda" = fuelSubset$nir.lambda)
ggplot(df) +
```

```
  geom_line(aes(y = NIR_Obs1, x = lambda))
```




### 4.2.1 How to model functional data

There are two commonly used approaches for analysing functional data.

- Directly analyze the functional data using a learner that is suitable for functional data on a Task. Those learners have the prefixes **classif.fda** and **regr.fda**. For more info on learners see fda learners. For this purpose, the functional data has to be saved as a matrix column in the data.frame used for constructing the Task. For more info on functional tasks consider the folowing section.
- Transform the task into a format suitable for standard **classification** or **regression** learners. This is done by extracting non-temporal/non-functional features from the curves. Non-temporal features do not have any interdependence between each other, similarly to features in traditional machine learning.This is explained in more detail below.

### 4.2.2 Creating a Task that contains functional features

The first step is to get the data in the right format. mlr expects a data.frame which consists of the functional features and the target variable as input. Functional data in contrast to **numeric** data have to be stored as a matrix column

in the data.frame. After that a Task that contains the data in a well-defined format is created. Tasks come in different flavours, such as ClassifTask and RegrTask, which can be used according to the class of the target variable.

In the following example, the data is first stored as matrix columns using the helper function makeFunctionalData for the fuelSubset data from package FDboost.

The data is provided in the following structure:

```
str(fuelSubset)
#> List of 7
#>  $ heatan      : num [1:129] 26.8 27.5 23.8 18.2 17.5 ...
#>  $ h2o         : num [1:129] 2.3 3 2 1.85 2.39 ...
#>  $ nir.lambda  : num [1:231] 800 803 805 808 810 ...
#>  $ NIR         : num [1:129, 1:231] 0.2818 0.2916 -0.0042 -0.034
    -0.1804 ...
#>  $ uvvis.lambda: num [1:134] 250 256 261 267 273 ...
#>  $ UVVIS       : num [1:129, 1:134] 0.145 -1.584 -0.814 -1.311
    -1.373 ...
#>  $ h2o.fit     : num [1:129] 2.58 3.43 1.83 2.03 3.07 ...
```

- **heatan** is the target variable, in this case a numeric value.
- **h2o** is an additional scalar variable.
- **NIR** and **UVVIS** are matricies containing the curve data. Each column corresponds to a single time point the data was sampled at. Each row indicates a single curve. **NIR** was measured at 231 time points, while **UVVIS** was measured at 129 time points.
- **nir.lambda** and **uvvis.lambda** are vectors of length 231 and 129 indicate the time points the data was measured at. Each entry corresponds to one column of **NIR** and **UVVIS** respectively. For now we ignore this additional information in mlr.

Our data already contains functional features as matricies in a list. In order to showcase how such a matrix can be created from arbitrary numeric columns, we transform the list into a data.frame with a set of numeric columns for each matrix. These columns refer to the matrix columns in the list, i.e **UVVIS.1** is the first column of the UVVIS matrix.

```
### Put all values into a data.frame
df = data.frame(fuelSubset[c("heatan", "h2o", "UVVIS", "NIR")])
str(df[, 1:5])
#> 'data.frame':    129 obs. of  5 variables:
#>  $ heatan : num  26.8 27.5 23.8 18.2 17.5 ...
#>  $ h2o    : num  2.3 3 2 1.85 2.39 ...
#>  $ UVVIS.1: num  0.145 -1.584 -0.814 -1.311 -1.373 ...
#>  $ UVVIS.2: num  -0.0111 -2.0467 -1.053 -1.2445 -1.8826 ...
#>  $ UVVIS.3: num  0.0372 -1.5695 -0.9381 -1.0649 -1.4016 ...
```

Before constructing the Task, the data is again reformated so it contains column matricies. This is done by providing a list **fd.features**, that identifies the functional covariates. All columns not mentioned in the list are kept as-is. In our case the column indices 3:136 correspond to the columns of the UVVIS matrix. Alternatively we could also specify the respective column names.

```
## fd.features is a named list, where each name corresponds to the
    name of the
## fuctional feature and the values to the respective column indices
    or column names.
fd.features = list("UVVIS" = 3:136, "NIR" = 137:367)
fdf = makeFunctionalData(df, fd.features = fd.features)
```

makeFunctionalData returns a data.frame, where the functional features are contained as matricies.

```
str(fdf)
#> 'data.frame':    129 obs. of  4 variables:
#>  $ heatan: num  26.8 27.5 23.8 18.2 17.5 ...
#>  $ h2o   : num  2.3 3 2 1.85 2.39 ...
#>  $ UVVIS : num [1:129, 1:134] 0.145 -1.584 -0.814 -1.311 -1.373
    ...
#>   ..- attr(*, "dimnames")=List of 2
#>   .. ..$ : NULL
#>   .. ..$ : chr  "UVVIS.1" "UVVIS.2" "UVVIS.3" "UVVIS.4" ...
#>  $ NIR   : num [1:129, 1:231] 0.2818 0.2916 -0.0042 -0.034
    -0.1804 ...
#>   ..- attr(*, "dimnames")=List of 2
#>   .. ..$ : NULL
#>   .. ..$ : chr  "NIR.1" "NIR.2" "NIR.3" "NIR.4" ...
```

Now with a data.frame containing the functionals as matricies, a Task can be created:

```
## Create a regression task, classification tasks behave analogously
## In this case we use column indices
tsk1 = makeRegrTask("fuelsubset", data = fdf, target = "heatan")
tsk1
#> Supervised task: fuelsubset
#> Type: regr
#> Target: heatan
#> Observations: 129
#> Features:
#>    numerics     factors     ordered functionals
#>           1           0           0           2
#> Missings: FALSE
#> Has weights: FALSE
```

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

### 4.2.3 Constructing a learner

For functional data, learners are constructed using `makeLearner("<classif.<R_method_name>")` or `makeLearner("<regr.<R_method_name>")` depending on the target variable.

Applying learners to a Task works in two ways:

- **Use a learner suitable for functional data**
- For regression:

```
  # The following learners can be used for the task.
  listLearners(tsk1, properties = "functionals")
#>              class                                        name
    short.name
#> 1     regr.FDboost Functional linear array regression boosting
       FDboost
#> 2 regr.featureless                       Featureless regression
    featureless
#>          package type installed numerics factors ordered missings
    weights
#> 1 FDboost,mboost regr      TRUE     TRUE   FALSE   FALSE    FALSE
      FALSE
#> 2            mlr regr      TRUE     TRUE    TRUE    TRUE     TRUE
      FALSE
#>    prob oneclass twoclass multiclass class.weights featimp
    oobpreds
#> 1 FALSE    FALSE    FALSE      FALSE         FALSE   FALSE
    FALSE
#> 2 FALSE    FALSE    FALSE      FALSE         FALSE   FALSE
    FALSE
#>   functionals single.functional    se lcens rcens icens
#> 1        TRUE             FALSE FALSE FALSE FALSE FALSE
#> 2        TRUE             FALSE FALSE FALSE FALSE FALSE
#> ... (#elements: 24)
  # Create a FDboost learner
  fdalrn = makeLearner("regr.FDboost")
```

- Or alternatively for classification:

```
  # knn learner
  knn.lrn = makeLearner("classif.fdausc.knn")
```

- **Use a *standard* learner**: In this case the temporal structure is disregarded, and the functional data treated as simple numeric features.

```
### Decision Tree learner
rpartlrn = makeLearner("classif.rpart")
```

- Alternatively, transform the functional data into a non-temporal/non-functional space by extracting features before training. In this case, a normal regression- or classification-learner can be applied.

This is explained in more detail in the feature extraction section below.

### 4.2.4 Train the learner

The resulting learner can now be trained on the task created in section Creating a Task above.

```
## Train the fdalrn on the constructed task
m = train(learner = fdalrn, task = tsk1)
#> UVVIS is not centered per column, inducing a non-centered effect.
#> NIR is not centered per column, inducing a non-centered effect.
p = predict(m, tsk1)
performance(p, rmse)
#>     rmse
#> 2.181438
```

```
## Alternatively, we can simply resample (3-fold Cross-Validation)
resample(fdalrn, tsk1, resampling = cv3, measures =  mse, show.info
    = FALSE)
#> UVVIS is not centered per column, inducing a non-centered effect.
#> NIR is not centered per column, inducing a non-centered effect.
#> UVVIS is not centered per column, inducing a non-centered effect.
#> NIR is not centered per column, inducing a non-centered effect.
#> UVVIS is not centered per column, inducing a non-centered effect.
#> NIR is not centered per column, inducing a non-centered effect.
#> Warning in bsplines(mf[[i]], knots = args$knots[[i]]$knots,
    boundary.knots
#> = args$knots[[i]]$boundary.knots, : Some 'x' values are beyond
#> 'boundary.knots'; Linear extrapolation used.
#> Resample Result
#> Task: fuelsubset
```

```
#> Learner: regr.FDboost
#> Aggr perf: mse.test.mean=5.9374832
#> Runtime: 2.38408
```

Alternatively, learners that do not specifically treat functional covariates can be applied. In this case the temporal structure is completely disregarded, and all columns are treated as independent.

```
## Train a normal learner on the constructed task.
## Note that we get a message, that functionals have been converted
    to numerics.
rpart.lrn = makeLearner("regr.rpart")
m = train(learner = rpart.lrn, task = tsk1)
#> Functional features have been converted to numerics
m
#> Model for learner.id=regr.rpart; learner.class=regr.rpart
#> Trained on: task.id = fuelsubset; obs = 129; features = 3
#> Hyperparameters: xval=0
```

### 4.2.5 Feature Extraction

In contrast to applying a learner that works on a Task containing functional features, the Task can be converted to a normal Task.md. This works by transforming the functional features into a non-functional domain, e.g by extracting wavelets.

The currently supported preprocessing functions are: * discrete wavelet transform * fast fourier transform * functional principal component analysis * multiresolution feature extraction

In order to do this, we specify methods for each functional feature in the task in a **list**. In this case we simply want to extract the mean from each **UVVIS** functional and the fourier transformed features from each **NIR** functional. Additional arguments can be passed on

```
## feat.methods specifies what to extract from which functional
## from the first functional we extract the fourier transformation,
    from the second the fpca scores
feat.methods = list("UVVIS" = extractFDAFourier(), "NIR" =
    extractFDAFPCA())

## Either create a new task from an existing task
extracted = extractFDAFeatures(tsk1, feat.methods = feat.methods)
extracted
#> $task
#> Supervised task: fuelsubset
#> Type: regr
```

```
#> Target: heatan
#> Observations: 129
#> Features:
#>    numerics     factors     ordered functionals
#>         137           0           0           0
#> Missings: FALSE
#> Has weights: FALSE
#> Has blocking: FALSE
#> Is spatial: FALSE
#>
#> $desc
#> Extraction of features from functional data:
#> Target: heatan
#> Functional Features: 2; Extracted features: 2
```

#### 4.2.5.1 Wavelets

In this case, discrete wavelet feature transformation is applied. We can specify which feature extraction method is used via *method = "wavelets"* and add additional parameters (i.e. the filter and the boundary) in the pars argument. This functions returns a regression task of type regr since the raw data contained temporal structure but the transformed data does not inherit temporal structure anymore. For more informations on wavelets consider the documentation wavelets.

```
### Specify the feature extraction method and generate new task.
### Here, we use the Haar filter:
feat.methods = list("UVVIS" = extractFDAWavelets(filter = "haar"))
task.w = extractFDAFeatures(tsk1, feat.methods = feat.methods)

## Use the Daubechie wavelet with filter length 4.
feat.methods = list("NIR" = extractFDAWavelets(filter = "d4"))
task.wd4 = extractFDAFeatures(tsk1, feat.methods = feat.methods)
```

#### 4.2.5.2 Fourier transformation

Now, we use the fourier feature transformation. Either the amplitude or the phase of the complex fourier coefficients can be used for analysis. This can be specified in the additional *fft.coeff* argument:

```
## Specify the feature extraction method and generate new task.
## We use the fourier features and the amplitude for NIR, as well as
    the phase for UVVIS
feat.methods = list("NIR" = extractFDAFourier(trafo.coeff =
    "amplitude"),
```

```
                    "UVVIS" = extractFDAFourier(trafo.coeff =
                        "phase"))
task.fourier = extractFDAFeatures(tsk1, feat.methods = feat.methods)
task.fourier
```

#### 4.2.5.3 Wrappers

Additionally we can wrap the preprocessing around a standard learner such as **classif.rpart**. For additional information, please consider the **Wrappers** section.

```
## Use a FDAFeatExtractWrapper
feat.methods = list("UVVIS" = extractFDAMultiResFeatures(), "NIR" =
    extractFDAFourier())
wrapped.lrn = makeExtractFDAFeatsWrapper("classif.rpart",
    feat.methods = feat.methods)
wrapped.lrn
#> Learner classif.rpart.extracted from package rpart
#> Type: classif
#> Name: ; Short name:
#> Class: extractFDAFeatsWrapper
#> Properties:
    twoclass,multiclass,missings,numerics,factors,ordered,prob,weights,featimp,functionals,singl
#> Predict-Type: response
#> Hyperparameters: xval=0
```

## 4.3 Handling of Spatial Data

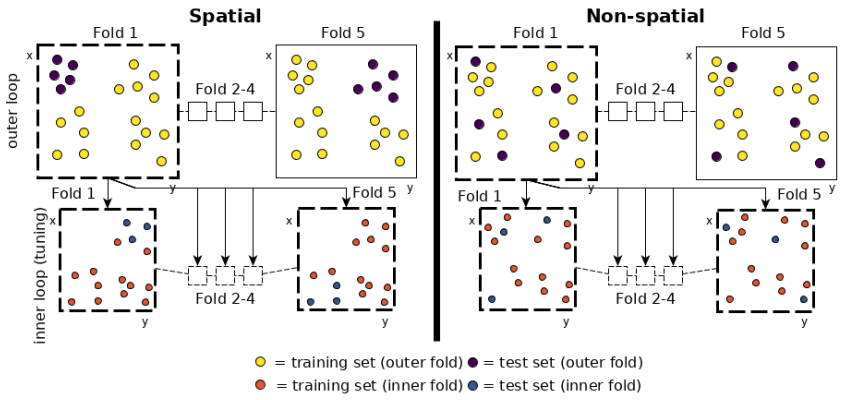
### 4.3.1 Introduction

Spatial data is different from non-spatial data by having a spatial reference information attached to each observation. This information is usually stored as coordinates, often named `x` and `y`. Coordinates are either stored in UTM (Universal Transverse Mercator) or latitude/longitude format.

Treating spatial data sets like non-spatial ones leads to overoptimistic results in predictive accuracy of models (Brenning 2005). This is due to the underlying spatial autocorrelation in the data. Spatial autocorrelation does occur in all spatial data sets. Magnitude varies depending on the characteristics of the data set. The closer observations are located to each other, the more similar they are.

If common validation procedures like cross-validation are applied to such data sets, they assume independence of the observation upfront to provide unbiased estimates. However, this assumption is violated in the spatial case due to spatial

autocorrelation. Subsequently, non-spatial cross-validation will fail to provide accurate performance estimates.




By doing a random sampling of the data set (i.e., non-spatial sampling), training and test set data are often located directly next to each other (in geographical space). Hence, the test set will contain observations which are somewhat similar (due to spatial autocorrelation) to observations in the training set. This leads to the effect that the model, which was trained on the training set, performs quite well on the test data because it already knows it to some degree.

To reduce this bias on the resulting predictive accuracy estimate, Brenning 2005 suggested using spatial partitioning in favor of random partitioning (see Figure 1). Here, spatial clusters are equal to the number of folds chosen. These spatially disjoint subsets of the data introduce a spatial distance between training and test set. This reduces the influence of spatial autocorrelation and subsequently also the overoptimistic predictive accuracy estimates. The example in Figure 1 shows a five-fold nested cross-validation setting and exhibits the difference between spatial and non-spatial partitioning. The nested approach is used when hyperparameter tuning is performed.

### 4.3.2 How to use spatial partitioning in mlr

Spatial partitioning can be used when performing cross-validation. In any resample call you can choose `SpCV` or `SpRepCV` to use it. While `SpCV` will perform a spatial cross-validation with only one repetition, `SpRepCV` gives you the option to choose any number of repetitions. As a rule of thumb, usually 100 repetitions are used with the aim to reduce variance introduced by partitioning.

There are some prerequisites for this:

When specifying the task, you need to explicitly state that the task is `spatial` one by setting `spatial = TRUE`. Next, there need to be two variables named `x` and `y` in your data set that store the coordinate information. If this applies, the

coordinates will be used for spatial partitioning if `SpCV` or `SpRepCV` are selected as resampling strategies.

Also, these variables **will be removed** for any train() or predict() call. This means they will not be used as predictors within the specified learner but only to set up the sampling for the cross-validation task.

Coordinates must be named `x` and `y` to be used for spatial partitioning. If named differently and `spatial = TRUE` was set during task creation, an error will occur.

### 4.3.3 Examples

The `bc.task.spatial` data set serves as an example data set for spatial modeling tasks in `mlr`. The task attribute `spatial` is set to `TRUE` to indicate that the data set also stores coordinate information.

In this example, the "Random Forest" algorithm (package ranger) is used to model a binomial response variable.

For performance assessment, a repeated spatial cross-validation with 5 folds and 10 repetitions is chosen.

#### 4.3.3.1 Spatial Cross-Validation

```
data("bc.task.spatial")
bc.task.spatial
#> Supervised task: basque
#> Type: classif
#> Target: diplo01
#> Observations: 944
#> Features:
#>    numerics     factors     ordered functionals
#>           9           2           0           0
#> Missings: FALSE
#> Has weights: FALSE
#> Has blocking: FALSE
#> Is spatial: TRUE
#> Classes: 2
#>   0   1
#> 720 224
#> Positive class: 1

learner.glm = makeLearner("classif.ranger", predict.type = "prob")

resampling = makeResampleDesc("SpRepCV", fold = 5, reps = 10)
```

```
set.seed(123)
out = resample(learner = learner.glm, task = bc.task.spatial,
  resampling = resampling, measures = list(auc))
#> Resampling: repeated spatial cross-validation
#> Measures:             auc
#> [Resample] iter 1:    0.8564829
#> [Resample] iter 2:    0.8903922
#> [Resample] iter 3:    0.8018739
#> [Resample] iter 4:    0.8170660
#> [Resample] iter 5:    0.8626639
#> [Resample] iter 6:    0.8888889
#> [Resample] iter 7:    0.9200000
#> [Resample] iter 8:    0.8844538
#> [Resample] iter 9:    0.7644215
#> [Resample] iter 10:   0.7704265
#> [Resample] iter 11:   0.8883220
#> [Resample] iter 12:   0.8430926
#> [Resample] iter 13:   0.7603609
#> [Resample] iter 14:   0.7996886
#> [Resample] iter 15:   0.9112150
#> [Resample] iter 16:   0.8575808
#> [Resample] iter 17:   0.8758170
#> [Resample] iter 18:   0.9722531
#> [Resample] iter 19:   0.7683578
#> [Resample] iter 20:   0.7707363
#> [Resample] iter 21:   0.8282783
#> [Resample] iter 22:   0.8040078
#> [Resample] iter 23:   0.8559476
#> [Resample] iter 24:   0.8466709
#> [Resample] iter 25:   0.8556985
#> [Resample] iter 26:   0.9694784
#> [Resample] iter 27:   0.8534002
#> [Resample] iter 28:   0.8696655
#> [Resample] iter 29:   0.7530708
#> [Resample] iter 30:   0.7707363
#> [Resample] iter 31:   0.7859410
#> [Resample] iter 32:   0.8797678
#> [Resample] iter 33:   0.7657106
#> [Resample] iter 34:   0.8925759
#> [Resample] iter 35:   0.9153037
#> [Resample] iter 36:   0.8602616
#> [Resample] iter 37:   0.8804180
#> [Resample] iter 38:   0.7662884
#> [Resample] iter 39:   0.7716682
#> [Resample] iter 40:   0.9719888
#> [Resample] iter 41:   0.7636481
```

```
#> [Resample] iter 42:   0.7815193
#> [Resample] iter 43:   0.9147196
#> [Resample] iter 44:   0.8892013
#> [Resample] iter 45:   0.8720978
#> [Resample] iter 46:   0.8436748
#> [Resample] iter 47:   0.7144397
#> [Resample] iter 48:   0.7679477
#> [Resample] iter 49:   0.8233804
#> [Resample] iter 50:   0.7238220
#>
#> Aggregated Result: auc.test.mean=0.8379085
#>

mean(out$measures.test$auc)
#> [1] 0.8379085
```

We can check for the introduced spatial autocorrelation bias here by performing the same modeling task using a non-spatial partitioning setting. To force non-spatial cross-validation here, variables `x` and `y` need to be removed from the data set because they are not used for partitioning when using `RepCV` and also should not be used as predictors.

Additionally, the `spatial` task attribute needs to be set to `FALSE`. Finally, `RepCV` is chosen instead of `SpRepCV`.

#### 4.3.3.2 Non-Spatial Cross-Validation

```
data("bc.task.spatial")
bc.task.spatial
#> Supervised task: basque
#> Type: classif
#> Target: diplo01
#> Observations: 944
#> Features:
#>    numerics     factors     ordered functionals
#>           9           2           0           0
#> Missings: FALSE
#> Has weights: FALSE
#> Has blocking: FALSE
#> Is spatial: TRUE
#> Classes: 2
#>   0   1
#> 720 224
#> Positive class: 1

bc.task.spatial$task.desc$spatial = FALSE
```

```
bc.task.spatial$env$data$x = NULL
bc.task.spatial$env$data$y = NULL

learner.glm = makeLearner("classif.ranger", predict.type = "prob")

resampling = makeResampleDesc("RepCV", fold = 5, reps = 10)

set.seed(123)
out = resample(learner = learner.glm, task = bc.task.spatial,
  resampling = resampling, measures = list(auc))
#> Resampling: repeated cross-validation
#> Measures:             auc
#> [Resample] iter 1:    0.8967930
#> [Resample] iter 2:    0.9045999
#> [Resample] iter 3:    0.9294306
#> [Resample] iter 4:    0.8802067
#> [Resample] iter 5:    0.9297806
#> [Resample] iter 6:    0.9167055
#> [Resample] iter 7:    0.9145773
#> [Resample] iter 8:    0.9190111
#> [Resample] iter 9:    0.9230211
#> [Resample] iter 10:   0.8920038
#> [Resample] iter 11:   0.9174018
#> [Resample] iter 12:   0.9216362
#> [Resample] iter 13:   0.8649985
#> [Resample] iter 14:   0.9400488
#> [Resample] iter 15:   0.9335277
#> [Resample] iter 16:   0.8901363
#> [Resample] iter 17:   0.8874161
#> [Resample] iter 18:   0.9116547
#> [Resample] iter 19:   0.9056988
#> [Resample] iter 20:   0.9353667
#> [Resample] iter 21:   0.9007716
#> [Resample] iter 22:   0.9178832
#> [Resample] iter 23:   0.9260784
#> [Resample] iter 24:   0.8679113
#> [Resample] iter 25:   0.9532938
#> [Resample] iter 26:   0.8807246
#> [Resample] iter 27:   0.9007646
#> [Resample] iter 28:   0.9112245
#> [Resample] iter 29:   0.9021704
#> [Resample] iter 30:   0.9442953
#> [Resample] iter 31:   0.9243403
#> [Resample] iter 32:   0.9407455
#> [Resample] iter 33:   0.8723823
#> [Resample] iter 34:   0.9146044
```

```
#> [Resample] iter 35:   0.8980796
#> [Resample] iter 36:   0.9012854
#> [Resample] iter 37:   0.9103448
#> [Resample] iter 38:   0.9683020
#> [Resample] iter 39:   0.9169096
#> [Resample] iter 40:   0.8891844
#> [Resample] iter 41:   0.8992375
#> [Resample] iter 42:   0.8888889
#> [Resample] iter 43:   0.9168440
#> [Resample] iter 44:   0.9027697
#> [Resample] iter 45:   0.9038690
#> [Resample] iter 46:   0.9262346
#> [Resample] iter 47:   0.8470183
#> [Resample] iter 48:   0.8931188
#> [Resample] iter 49:   0.9615858
#> [Resample] iter 50:   0.9296015
#>
#> Aggregated Result: auc.test.mean=0.9104896
#>

mean(out$measures.test$auc)
#> [1] 0.9104896
```

The introduced bias (caused by spatial autocorrelation) in performance in this example is around 0.07 AUROC.

### 4.3.4 Notes

- Usually, coordinates are not used as predictors in spatial modeling. If you insist on using them and also want to perform spatial cross-validation, add them to the data set with a different name than `x` or `y`, e.g. `x1` and `y1`. That way they will be treated as normal predictors.
- Some models are more affected by spatial autocorrelation than others. In general, it can be said that the more flexible a model is, the more it will profit from underlying spatial autocorrelation. Simpler models (e.g., GLM) will show less overoptimistic performance estimates.
- The concept of spatial cross-validation was originally implemented in package sperrorest. This package comes with even more partitioning options and the ability to visualize the spatial grouping of folds. We plan to integrate more functions from sperrorest into `mlr` so stay tuned!
- For more detailed information, see Brenning 2005 and Brenning2012.

## 4.4 Integrating Another Learner

In order to integrate a learning algorithm into mlr some interface code has to be written. Three functions are mandatory for each learner.

- First, define a new learner class with a name, description, capabilities, parameters, and a few other things. (An object of this class can then be generated by makeLearner.)
- Second, you need to provide a function that calls the learner function and builds the model given data (which makes it possible to invoke training by calling mlr's train function).
- Finally, a prediction function that returns predicted values given new data is required (which enables invoking prediction by calling mlr's predict function).

Technically, integrating a learning method means introducing a new S3 class and implementing the corresponding methods for the generic functions makeRLerner, trainLearner, and predictLearner. Therefore we start with a quick overview of the involved classes and constructor functions.

### 4.4.1 Classes, constructors, and naming schemes

As you already know makeLearner generates an object of class Learner.

```
class(makeLearner(cl = "classif.lda"))
#> [1] "classif.lda"     "RLearnerClassif" "RLearner"
    "Learner"

class(makeLearner(cl = "regr.lm"))
#> [1] "regr.lm"      "RLearnerRegr" "RLearner"     "Learner"

class(makeLearner(cl = "surv.coxph"))
#> [1] "surv.coxph"   "RLearnerSurv" "RLearner"     "Learner"

class(makeLearner(cl = "cluster.kmeans"))
#> [1] "cluster.kmeans"  "RLearnerCluster" "RLearner"
    "Learner"

class(makeLearner(cl = "multilabel.rFerns"))
#> [1] "multilabel.rFerns"  "RLearnerMultilabel" "RLearner"
#> [4] "Learner"
```

The first element of each class attribute vector is the name of the learner class passed to the `cl` argument of makeLearner. Obviously, this adheres to the naming conventions

- `"classif.<R_method_name>"` for classification,
- `"multilabel.<R_method_name>"` for multilabel classification,
- `"regr.<R_method_name>"` for regression,
- `"surv.<R_method_name>"` for survival analysis, and
- `"cluster.<R_method_name>"` for clustering.

Additionally, there exist intermediate classes that reflect the type of learning problem, i.e., all classification learners inherit from RLearnerClassif, all regression learners from RLearnerRegr and so on. Their superclasses are RLearner and finally Learner. For all these (sub)classes there exist constructor functions makeRLearner, makeRLearnerClassif, makeRLearneRegr etc. that are called internally by makeLearner.

A short side remark: As you might have noticed there does not exist a special learner class for cost-sensitive classification (costsens) with example-specific costs. This type of learning task is currently exclusively handled through wrappers like makeCostSensWeightedPairsWrapper.

In the following we show how to integrate learners for the five types of learning tasks mentioned above. Defining a completely new type of learner that has special properties and does not fit into one of the existing schemes is of course possible, but much more advanced and not covered here.

We use a classification example to explain some general principles (so even if you are interested in integrating a learner for another type of learning task you might want to read the following section). Examples for other types of learning tasks are shown later on.

### 4.4.2 Classification

We show how the Linear Discriminant Analysis from package MASS has been integrated into the classification learner `classif.lda` in mlr as an example.

#### 4.4.2.1 Definition of the learner

The minimal information required to define a learner is the mlr name of the learner, its package, the parameter set, and the set of properties of your learner. In addition, you may provide a human-readable name, a short name and a note with information relevant to users of the learner.

First, name your learner. According to the naming conventions above the name starts with `classif.` and we choose `classif.lda`.

Second, we need to define the parameters of the learner. These are any options that can be set when running it to change how it learns, how input is interpreted, how and what output is generated, and so on. mlr provides a number of functions to define parameters, a complete list can be found in the documentation of LearnerParam of the ParamHelpers package.

In our example, we have discrete and numeric parameters, so we use makeDiscreteLearnerParam and makeNumericLearnerParam to incorporate the complete description of the parameters. We include all possible values for discrete parameters and lower and upper bounds for numeric parameters. Strictly speaking it is not necessary to provide bounds for all parameters and if this information is not available they can be estimated, but providing accurate and specific information here makes it possible to tune the learner much better (see the section on tuning).

Next, we add information on the properties of the learner (see also the section on learners). Which types of features are supported (numerics, factors)? Are case weights supported? Are class weights supported? Can the method deal with missing values in the features and deal with NA's in a meaningful way (not `na.omit`)? Are one-class, two-class, multi-class problems supported? Can the learner predict posterior probabilities?

If the learner supports class weights the name of the relevant learner parameter can be specified via argument `class.weights.param`.

Below is the complete code for the definition of the LDA learner. It has one discrete parameter, `method`, and two continuous ones, `nu` and `tol`. It supports classification problems with two or more classes and can deal with numeric and factor explanatory variables. It can predict posterior probabilities.

```
makeRLearner.classif.lda = function() {
  makeRLearnerClassif(
    cl = "classif.lda",
    package = "MASS",
    par.set = makeParamSet(
      makeDiscreteLearnerParam(id = "method", default = "moment",
          values = c("moment", "mle", "mve", "t")),
      makeNumericLearnerParam(id = "nu", lower = 2, requires =
          quote(method == "t")),
      makeNumericLearnerParam(id = "tol", default = 1e-4, lower = 0),
      makeDiscreteLearnerParam(id = "predict.method", values =
          c("plug-in", "predictive", "debiased"),
        default = "plug-in", when = "predict"),
      makeLogicalLearnerParam(id = "CV", default = FALSE, tunable =
          FALSE)
    ),
    properties = c("twoclass", "multiclass", "numerics", "factors",
        "prob"),
    name = "Linear Discriminant Analysis",
    short.name = "lda",
    note = "Learner param 'predict.method' maps to 'method' in
        predict.lda."
  )
}
```

#### 4.4.2.2 Creating the training function of the learner

Once the learner has been defined, we need to tell mlr how to call it to train a model. The name of the function has to start with `trainLearner.`, followed by the mlr name of the learner as defined above (`classif.lda` here). The prototype of the function looks as follows.

```
function(.learner, .task, .subset, .weights = NULL, ...) { }
```

This function must fit a model on the data of the task `.task` with regard to the subset defined in the integer vector `.subset` and the parameters passed in the `...` arguments. Usually, the data should be extracted from the task using getTaskData. This will take care of any subsetting as well. It must return the fitted model. mlr assumes no special data type for the return value – it will be passed to the predict function we are going to define below, so any special code the learner may need can be encapsulated there.

For our example, the definition of the function looks like this. In addition to the data of the task, we also need the formula that describes what to predict. We use the function getTaskFormula to extract this from the task.

```
trainLearner.classif.lda = function(.learner, .task, .subset,
    .weights = NULL, ...) {
  f = getTaskFormula(.task)
  MASS::lda(f, data = getTaskData(.task, .subset), ...)
}
```

#### 4.4.2.3 Creating the prediction method

Finally, the prediction function needs to be defined. The name of this function starts with `predictLearner.`, followed again by the mlr name of the learner. The prototype of the function is as follows.

```
function(.learner, .model, .newdata, ...) { }
```

It must predict for the new observations in the `data.frame` `.newdata` with the wrapped model `.model`, which is returned from the training function. The actual model the learner built is stored in the `$learner.model` member and can be accessed simply through `.model$learner.model`.

For classification, you have to return a factor of predicted classes if `.learner$predict.type` is `"response"`, or a matrix of predicted probabilities if `.learner$predict.type` is `"prob"` and this type of prediction is supported by the learner. In the latter case the matrix must have the same number of columns as there are classes in the task and the columns have to be named by the class names.

The definition for LDA looks like this. It is pretty much just a straight pass-through of the arguments to the predict function and some extraction of prediction data depending on the type of prediction requested.

```
predictLearner.classif.lda = function(.learner, .model, .newdata,
    predict.method = "plug-in", ...) {
  p = predict(.model$learner.model, newdata = .newdata, method =
      predict.method, ...)
  if (.learner$predict.type == "response")
    return(p$class) else return(p$posterior)
}
```

### 4.4.3 Regression

The main difference for regression is that the type of predictions are different (numeric instead of labels or probabilities) and that not all of the properties are relevant. In particular, whether one-, two-, or multi-class problems and posterior probabilities are supported is not applicable.

Apart from this, everything explained above applies. Below is the definition for the earth learner from the earth package.

```
makeRLearner.regr.earth = function() {
  makeRLearnerRegr(
    cl = "regr.earth",
    package = "earth",
    par.set = makeParamSet(
      makeLogicalLearnerParam(id = "keepxy", default = FALSE,
          tunable = FALSE),
      makeNumericLearnerParam(id = "trace", default = 0, upper = 10,
          tunable = FALSE),
      makeIntegerLearnerParam(id = "degree", default = 1L, lower =
          1L),
      makeNumericLearnerParam(id = "penalty"),
      makeIntegerLearnerParam(id = "nk", lower = 0L),
      makeNumericLearnerParam(id = "thres", default = 0.001),
      makeIntegerLearnerParam(id = "minspan", default = 0L),
      makeIntegerLearnerParam(id = "endspan", default = 0L),
      makeNumericLearnerParam(id = "newvar.penalty", default = 0),
      makeIntegerLearnerParam(id = "fast.k", default = 20L, lower =
          0L),
      makeNumericLearnerParam(id = "fast.beta", default = 1),
      makeDiscreteLearnerParam(id = "pmethod", default = "backward",
        values = c("backward", "none", "exhaustive", "forward",
            "seqrep", "cv")),
      makeIntegerLearnerParam(id = "nprune")
    ),
    properties = c("numerics", "factors"),
    name = "Multivariate Adaptive Regression Splines",
    short.name = "earth",
```

```
    note = ""
  )
}
```

```
trainLearner.regr.earth = function(.learner, .task, .subset,
    .weights = NULL, ...) {
  f = getTaskFormula(.task)
  earth::earth(f, data = getTaskData(.task, .subset), ...)
}
```

```
predictLearner.regr.earth = function(.learner, .model, .newdata,
    ...) {
  predict(.model$learner.model, newdata = .newdata)[, 1L]
}
```

Again most of the data is passed straight through to/from the train/predict functions of the learner.

### 4.4.4 Survival analysis

For survival analysis, you have to return so-called linear predictors in order to compute the default measure for this task type, the cindex (for `.learner$predict.type == "response"`). For `.learner$predict.type == "prob"`, there is no substantially meaningful measure (yet). You may either ignore this case or return something like predicted survival curves (cf. example below).

There are three properties that are specific to survival learners: "rcens", "lcens" and "icens", defining the type(s) of censoring a learner can handle – right, left and/or interval censored.

Let's have a look at how the Cox Proportional Hazard Model from package survival has been integrated into the survival learner `surv.coxph` in mlr as an example:

```
makeRLearner.surv.coxph = function() {
  makeRLearnerSurv(
    cl = "surv.coxph",
    package = "survival",
    par.set = makeParamSet(
      makeDiscreteLearnerParam(id = "ties", default = "efron",
          values = c("efron", "breslow", "exact")),
      makeLogicalLearnerParam(id = "singular.ok", default = TRUE),
      makeNumericLearnerParam(id = "eps", default = 1e-09, lower =
          0),
      makeNumericLearnerParam(id = "toler.chol", default =
          .Machine$double.eps^0.75, lower = 0),
```

```
      makeIntegerLearnerParam(id = "iter.max", default = 20L, lower
          = 1L),
      makeNumericLearnerParam(id = "toler.inf", default =
          sqrt(.Machine$double.eps^0.75), lower = 0),
      makeIntegerLearnerParam(id = "outer.max", default = 10L, lower
          = 1L),
      makeLogicalLearnerParam(id = "model", default = FALSE, tunable
          = FALSE),
      makeLogicalLearnerParam(id = "x", default = FALSE, tunable =
          FALSE),
      makeLogicalLearnerParam(id = "y", default = TRUE, tunable =
          FALSE)
    ),
    properties = c("missings", "numerics", "factors", "weights",
        "prob", "rcens"),
    name = "Cox Proportional Hazard Model",
    short.name = "coxph",
    note = ""
  )
}
```

```
trainLearner.surv.coxph = function(.learner, .task, .subset,
    .weights = NULL, ...) {
  f = getTaskFormula(.task)
  data = getTaskData(.task, subset = .subset)
  if (is.null(.weights)) {
    survival::coxph(formula = f, data = data, ...)
  } else {
    survival::coxph(formula = f, data = data, weights = .weights,
        ...)
  }
}
```

```
predictLearner.surv.coxph = function(.learner, .model, .newdata,
    ...) {
  predict(.model$learner.model, newdata = .newdata, type = "lp", ...)
}
```

### 4.4.5 Clustering

For clustering, you have to return a numeric vector with the IDs of the clusters that the respective datum has been assigned to. The numbering should start at 1.

Below is the definition for the FarthestFirst learner from the RWeka package. Weka starts the IDs of the clusters at 0, so we add 1 to the predicted clusters.

RWeka has a different way of setting learner parameters; we use the special `Weka_control` function to do this.

```
makeRLearner.cluster.FarthestFirst = function() {
  makeRLearnerCluster(
    cl = "cluster.FarthestFirst",
    package = "RWeka",
    par.set = makeParamSet(
      makeIntegerLearnerParam(id = "N", default = 2L, lower = 1L),
      makeIntegerLearnerParam(id = "S", default = 1L, lower = 1L),
      makeLogicalLearnerParam(id = "output-debug-info", default =
          FALSE, tunable = FALSE)
    ),
    properties = c("numerics"),
    name = "FarthestFirst Clustering Algorithm",
    short.name = "farthestfirst"
  )
}
```

```
trainLearner.cluster.FarthestFirst = function(.learner, .task,
    .subset, .weights = NULL, ...) {
  ctrl = RWeka::Weka_control(...)
  RWeka::FarthestFirst(getTaskData(.task, .subset), control = ctrl)
}
```

```
predictLearner.cluster.FarthestFirst = function(.learner, .model,
    .newdata, ...) {
  as.integer(predict(.model$learner.model, .newdata, ...)) + 1L
}
```

### 4.4.6 Multilabel classification

As stated in the multilabel section, multilabel classification methods can be divided into problem transformation methods and algorithm adaptation methods.

At this moment the only problem transformation method implemented in mlr is the binary relevance method. Integrating more of these methods requires good knowledge of the architecture of the mlr package.

The integration of an algorithm adaptation multilabel classification learner is easier and works very similar to the normal multiclass-classification. In contrast to the multiclass case, not all of the learner properties are relevant. In particular, whether one-, two-, or multi-class problems are supported is not applicable. Furthermore the prediction function output must be a matrix with each prediction of a label in one column and the names of the labels as column names. If `.learner$predict.type` is `"response"` the predictions must be logical. If

`.learner$predict.type` is `"prob"` and this type of prediction is supported by the learner, the matrix must consist of predicted probabilities.

Below is the definition of the rFerns learner from the rFerns package, which does not support probability predictions.

```
makeRLearner.multilabel.rFerns = function() {
  makeRLearnerMultilabel(
    cl = "multilabel.rFerns",
    package = "rFerns",
    par.set = makeParamSet(
      makeIntegerLearnerParam(id = "depth", default = 5L),
      makeIntegerLearnerParam(id = "ferns", default = 1000L)
    ),
    properties = c("numerics", "factors", "ordered"),
    name = "Random ferns",
    short.name = "rFerns",
    note = ""
  )
}
```

```
trainLearner.multilabel.rFerns = function(.learner, .task, .subset,
    .weights = NULL, ...) {
  d = getTaskData(.task, .subset, target.extra = TRUE)
  rFerns::rFerns(x = d$data, y = as.matrix(d$target), ...)
}
```

```
predictLearner.multilabel.rFerns = function(.learner, .model,
    .newdata, ...) {
  as.matrix(predict(.model$learner.model, .newdata, ...))
}
```

### 4.4.7 Creating a new method for extracting feature importance values

Some learners, for example decision trees and random forests, can calculate feature importance values, which can be extracted from a fitted model using function getFeatureImportance.

If your newly integrated learner supports this you need to

- add `"featimp"` to the learner properties and
- implement a new S3 method for function getFeatureImportanceLearner (which later is called internally by getFeatureImportance)

in order to make this work.

This method takes the Learner `.learner`, the WrappedModel `.model` and potential further arguments and calculates or extracts the feature importance. It must return a named vector of importance values.

Below are two simple examples. In case of `"classif.rpart"` the feature importance values can be easily extracted from the fitted model.

```
getFeatureImportanceLearner.classif.rpart = function(.learner,
    .model, ...) {
  mod = getLearnerModel(.model, more.unwrap = TRUE)
  mod$variable.importance
}
```

For the random forest from package randomForestSRC function vimp is called.

```
getFeatureImportanceLearner.classif.randomForestSRC =
    function(.learner, .model, ...) {
  mod = getLearnerModel(.model, more.unwrap = TRUE)
  randomForestSRC::vimp(mod, ...)$importance[, "all"]
}
```

### 4.4.8 Creating a new method for extracting out-of-bag predictions

Many ensemble learners generate out-of-bag predictions (OOB predictions) automatically. mlr provides the function getOOBPreds to access these predictions in the mlr framework.

If your newly integrated learner is able to calculate OOB predictions and you want to be able to access them in mlr via getOOBPreds you need to

- add `"oobpreds"` to the learner properties and
- implement a new S3 method for function getOOBPredsLearner (which later is called internally by getOOBPreds).

This method takes the Learner `.learner` and the WrappedModel `.model` and extracts the OOB predictions. It must return the predictions in the same format as the predictLearner function.

```
getOOBPredsLearner.classif.randomForest = function(.learner, .model)
    {
  if (.learner$predict.type == "response") {
    m = getLearnerModel(.model, more.unwrap = TRUE)
    unname(m$predicted)
  } else {
    getLearnerModel(.model, more.unwrap = TRUE)$votes
  }
}
```

### 4.4.9 Registering your learner

If your interface code to a new learning algorithm exists only locally, i.e., it is not (yet) merged into mlr or does not live in an extra package with a proper namespace you might want to register the new S3 methods to make sure that these are found by, e.g., listLearners. You can do this as follows:

```
registerS3method("makeRLearner", "<awesome_new_learner_class>",
    makeRLearner.<awesome_new_learner_class>)
registerS3method("trainLearner", "<awesome_new_learner_class>",
    trainLearner.<awesome_new_learner_class>)
registerS3method("predictLearner", "<awesome_new_learner_class>",
    predictLearner.<awesome_new_learner_class>)
```

If you have written more methods, for example in order to extract feature importance values or out-of-bag predictions these also need to be registered in the same manner, for example:

```
registerS3method("getFeatureImportanceLearner",
    "<awesome_new_learner_class>",
  getFeatureImportanceLearner.<awesome_new_learner_class>)
```

For the new learner to work with parallelization, you may have to export the new methods explicitly:

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

### 4.4.10 Further information for developers

If you haven't written a learner interface for private use only, but intend to send a pull request to have it included in the mlr package there are a few things to take care of, most importantly unit testing!

For general information about contributing to the package, unit testing, version control setup and the like please also read the coding guidelines in the mlr Wiki.

- The R file containing the interface code should adhere to the naming convention `RLearner_<type>_<learner_name>.R`, e.g., `RLearner_classif_lda.R`, see for example https://github.com/mlr-org/mlr/blob/master/R/RLearner_classif_lda.R and contain the necessary roxygen `@export` tags to register the S3 methods in the NAMESPACE.
- The learner interfaces should work out of the box without requiring any parameters to be set, e.g., `train("classif.lda", iris.task)` should run. Sometimes, this makes it necessary to change or set some additional defaults as explained above and – very important – informing the user about this in the `note`.

- The parameter set of the learner should be as complete as possible.
- Every learner interface must be unit tested.

#### 4.4.10.1 Unit testing

The tests make sure that we get the same results when the learner is invoked through the mlr interface and when using the original functions. If you are not familiar or want to learn more about unit testing and package testthat have a look at the Testing chapter in Hadley Wickham's R packages.

In mlr all unit tests are in the following directory: https://github.com/mlr-org/mlr/tree/master/tests/testthat. For each learner interface there is an individual file whose name follows the scheme `test_<type>_<learner_name>.R`, for example https://github.com/mlr-org/mlr/blob/master/tests/testthat/test_classif_lda.R.

Below is a snippet from the tests of the lda interface https://github.com/mlr-org/mlr/blob/master/tests/testthat/test_classif_lda.R.

```
test_that("classif_lda", {
  requirePackagesOrSkip("MASS", default.method = "load")

  set.seed(getOption("mlr.debug.seed"))
  m = MASS::lda(formula = multiclass.formula, data =
      multiclass.train)
  set.seed(getOption("mlr.debug.seed"))
  p = predict(m, newdata = multiclass.test)

  testSimple("classif.lda", multiclass.df, multiclass.target,
      multiclass.train.inds, p$class)
  testProb("classif.lda", multiclass.df, multiclass.target,
      multiclass.train.inds, p$posterior)
})
```

The tests make use of numerous helper objects and helper functions. All of these are defined in the `helper_` files in https://github.com/mlr-org/mlr/blob/master/tests/testthat/.

In the above code the first line just loads package MASS or skips the test if the package is not available. The objects `multiclass.formula`, `multiclass.train`, `multiclass.test` etc. are defined in https://github.com/mlr-org/mlr/blob/master/tests/testthat/helper_objects.R. We tried to choose fairly self-explanatory names: For example `multiclass` indicates a multi-class classification problem, `multiclass.train` contains data for training, `multiclass.formula` a formula object etc.

The test fits an lda model on the training set and makes predictions on the test set using the original functions lda and predict.lda. The helper func-

tions `testSimple` and `testProb` perform training and prediction on the same data using the mlr interface – `testSimple` for `predict.type = "response` and `testProbs` for `predict.type = "prob"` – and check if the predicted class labels and probabilities coincide with the outcomes `p$class` and `p$posterior`.

In order to get reproducible results seeding is required for many learners. The `"mlr.debug.seed"` works as follows: When invoking the tests the option `"mlr.debug.seed"` is set (see https://github.com/mlr-org/mlr/blob/master/tests/testthat/helper_zzz.R), and `set.seed(getOption("mlr.debug.seed"))` is used to specify the seed. Internally, mlr's train and predict.WrappedModel functions check if the `"mlr.debug.seed"` option is set and if yes, also specify the seed.

Note that the option `"mlr.debug.seed"` is only set for testing, so no seeding happens in normal usage of mlr.

Let's look at a second example. Many learners have parameters that are commonly changed or tuned and it is important to make sure that these are passed through correctly. Below is a snippet from https://github.com/mlr-org/mlr/blob/master/tests/testthat/test_regr_randomForest.R.

```
test_that("regr_randomForest", {
  requirePackagesOrSkip("randomForest", default.method = "load")

  parset.list = list(
    list(),
    list(ntree = 5, mtry = 2),
    list(ntree = 5, mtry = 4),
    list(proximity = TRUE, oob.prox = TRUE),
    list(nPerm = 3)
  )

  old.predicts.list = list()

  for (i in 1:length(parset.list)) {
    parset = parset.list[[i]]
    pars = list(formula = regr.formula, data = regr.train)
    pars = c(pars, parset)
    set.seed(getOption("mlr.debug.seed"))
    m = do.call(randomForest::randomForest, pars)
    set.seed(getOption("mlr.debug.seed"))
    p = predict(m, newdata = regr.test, type = "response")
    old.predicts.list[[i]] = p
  }

  testSimpleParsets("regr.randomForest", regr.df, regr.target,
    regr.train.inds, old.predicts.list, parset.list)
})
```

All tested parameter configurations are collected in the `parset.list`. In order to make sure that the default parameter configuration is tested the first element of the `parset.list` is an empty list. Then we simply loop over all parameter settings and store the resulting predictions in `old.predicts.list`. Again the helper function `testSimpleParsets` does the same using the mlr interface and compares the outcomes.

Additional to tests for individual learners we also have general tests that loop through all integrated learners and make for example sure that learners have the correct properties (e.g. a learner with property `"factors"` can cope with factor features, a learner with property `"weights"` takes observation weights into account properly etc.). For example https://github.com/mlr-org/mlr/blob/master/tests/testthat/test_learners_all_classif.R runs through all classification learners. Similar tests exist for all types of learning methods like regression, cluster and survival analysis as well as multilabel classification.

In order to run all tests for, e.g., classification learners on your machine you can invoke the tests from within **R** by

```
devtools::test("mlr", filter = "classif")
```

or from the command line using Michel's rt tool

```
rtest --filter=classif
```

## 4.5 Integrating Another Measure

In some cases, you might want to evaluate a Prediction or ResamplePrediction with a Measure which is not yet implemented in mlr. This could be either a performance measure which is not listed in the Appendix or a measure that uses a misclassification cost matrix.

### 4.5.1 Performance measures and aggregation schemes

Performance measures in mlr are objects of class Measure. For example the mse (mean squared error) looks as follows.

```
str(mse)
#> List of 10
#>  $ id        : chr "mse"
#>  $ minimize  : logi TRUE
#>  $ properties: chr [1:3] "regr" "req.pred" "req.truth"
#>  $ fun       :function (task, model, pred, feats, extra.args)
#>  $ extra.args: list()
#>  $ best      : num 0
#>  $ worst     : num Inf
```

```
#>  $ name      : chr "Mean of squared errors"
#>  $ note      : chr "Defined as: mean((response - truth)^2)"
#>  $ aggr      :List of 4
#>   ..$ id        : chr "test.mean"
#>   ..$ name      : chr "Test mean"
#>   ..$ fun       :function (task, perf.test, perf.train, measure,
    group, pred)
#>   ..$ properties: chr "req.test"
#>   ..- attr(*, "class")= chr "Aggregation"
#>  - attr(*, "class")= chr "Measure"

mse$fun
#> function (task, model, pred, feats, extra.args)
#> {
#>     measureMSE(pred$data$truth, pred$data$response)
#> }
#> <bytecode: 0xef0ba28>
#> <environment: namespace:mlr>

measureMSE
#> function (truth, response)
#> {
#>     mean((response - truth)^2)
#> }
#> <bytecode: 0xf236388>
#> <environment: namespace:mlr>
```

See the Measure documentation page for a detailed description of the object slots.

At the core is slot `$fun` which contains the function that calculates the performance value. The actual work is done by function measureMSE. Similar functions, generally adhering to the naming scheme `measure` followed by the capitalized measure ID, exist for most performance measures. See the measures help page for a complete list.

Just as Task and Learner objects each Measure has an identifier `$id` which is for example used to annotate results and plots. For plots there is also the option to use the longer measure `$name` instead. See the tutorial page on Visualization for more information.

Moreover, a Measure includes a number of `$properties` that indicate for which types of learning problems it is suitable and what information is required to calculate it. Obviously, most measures need the Prediction object (`"req.pred"`) and, for supervised problems, the true values of the target variable(s) (`"req.truth"`). You can use functions getMeasureProperties and hasMeasureProperties to determine the properties of a Measure. Moreover, listMeasureProperties shows all measure properties currently available in mlr.

```
listMeasureProperties()
#>  [1] "classif"       "classif.multi" "multilabel"    "regr"
#>  [5] "surv"          "cluster"       "costsens"      "req.pred"
#>  [9] "req.truth"     "req.task"      "req.feats"     "req.model"
#> [13] "req.prob"
```

Additional to its properties, each Measure knows its extreme values `$best` and `$worst` and if it wants to be minimized or maximized (`$minimize`) during tuning or feature selection.

For resampling slot `$aggr` specifies how the overall performance across all resampling iterations is calculated. Typically, this is just a matter of aggregating the performance values obtained on the test sets `perf.test` or the training sets `perf.train` by a simple function. The by far most common scheme is test.mean, i.e., the unweighted mean of the performances on the test sets.

```
str(test.mean)
#> List of 4
#>  $ id         : chr "test.mean"
#>  $ name       : chr "Test mean"
#>  $ fun        :function (task, perf.test, perf.train, measure,
    group, pred)
#>  $ properties: chr "req.test"
#>  - attr(*, "class")= chr "Aggregation"

test.mean$fun
#> function (task, perf.test, perf.train, measure, group, pred)
#> mean(perf.test)
#> <bytecode: 0x8927908>
#> <environment: namespace:mlr>
```

All aggregation schemes are objects of class Aggregation with the function in slot `$fun` doing the actual work. The `$properties` member indicates if predictions (or performance values) on the training or test data sets are required to calculate the aggregation.

You can change the aggregation scheme of a Measure via function setAggregation. See the tutorial page on resampling for some examples and the aggregations help page for all available aggregation schemes.

You can construct your own Measure and Aggregation objects via functions makeMeasure, makeCostMeasure, makeCustomResampledMeasure and makeAggregation. Some examples are shown in the following.

### 4.5.2 Constructing a performance measure

Function makeMeasure provides a simple way to construct your own performance measure.

Below this is exemplified by re-implementing the mean misclassification error (mmce). We first write a function that computes the measure on the basis of the true and predicted class labels. Note that this function must have certain formal arguments listed in the documentation of makeMeasure. Then the Measure object is created and we work with it as usual with the performance function.

See the **R** documentation of makeMeasure for more details on the various parameters.

```
### Define a function that calculates the misclassification rate
my.mmce.fun = function(task, model, pred, feats, extra.args) {
  tb = table(getPredictionResponse(pred), getPredictionTruth(pred))
  1 - sum(diag(tb)) / sum(tb)
}

### Generate the Measure object
my.mmce = makeMeasure(
  id = "my.mmce", name = "My Mean Misclassification Error",
  properties = c("classif", "classif.multi", "req.pred",
      "req.truth"),
  minimize = TRUE, best = 0, worst = 1,
  fun = my.mmce.fun
)

### Train a learner and make predictions
mod = train("classif.lda", iris.task)
pred = predict(mod, task = iris.task)

### Calculate the performance using the new measure
performance(pred, measures = my.mmce)
#> my.mmce
#>    0.02

### Apparently the result coincides with the mlr implementation
performance(pred, measures = mmce)
#> mmce
#> 0.02
```

### 4.5.3 Constructing a measure for ordinary misclassification costs

For in depth explanations and details see the tutorial page on cost-sensitive classification.

To create a measure that involves ordinary, i.e., class-dependent misclassification costs you can use function makeCostMeasure. You first need to define the cost matrix. The rows indicate true and the columns predicted classes and the rows

and columns have to be named by the class labels. The cost matrix can then be wrapped in a Measure object and predictions can be evaluated as usual with the performance function.

See the **R** documentation of function makeCostMeasure for details on the various parameters.

```
### Create the cost matrix
costs = matrix(c(0, 2, 2, 3, 0, 2, 1, 1, 0), ncol = 3)
rownames(costs) = colnames(costs) = getTaskClassLevels(iris.task)

### Encapsulate the cost matrix in a Measure object
my.costs = makeCostMeasure(
  id = "my.costs", name = "My Costs",
  costs = costs,
  minimize = TRUE, best = 0, worst = 3
)

### Train a learner and make a prediction
mod = train("classif.lda", iris.task)
pred = predict(mod, newdata = iris)

### Calculate the average costs
performance(pred, measures = my.costs)
#>   my.costs
#> 0.02666667
```

### 4.5.4 Creating an aggregation scheme

It is possible to create your own aggregation scheme using function makeAggregation. You need to specify an identifier `id`, the `properties`, and write a function that does the actual aggregation. Optionally, you can `name` your aggregation scheme.

Possible settings for `properties` are `"req.test"` and `"req.train"` if predictions on either the training or test sets are required, and the vector `c("req.train", "req.test")` if both are needed.

The aggregation function must have a certain signature detailed in the documentation of makeAggregation. Usually, you will only need the performance values on the test sets `perf.test` or the training sets `perf.train`. In rare cases, e.g., the Prediction object `pred` or information stored in the Task object might be required to obtain the aggregated performance. For an example have a look at the definition of function test.join.

#### 4.5.4.1 Example: Evaluating the range of measures

Let's say you are interested in the range of the performance values obtained on individual test sets.

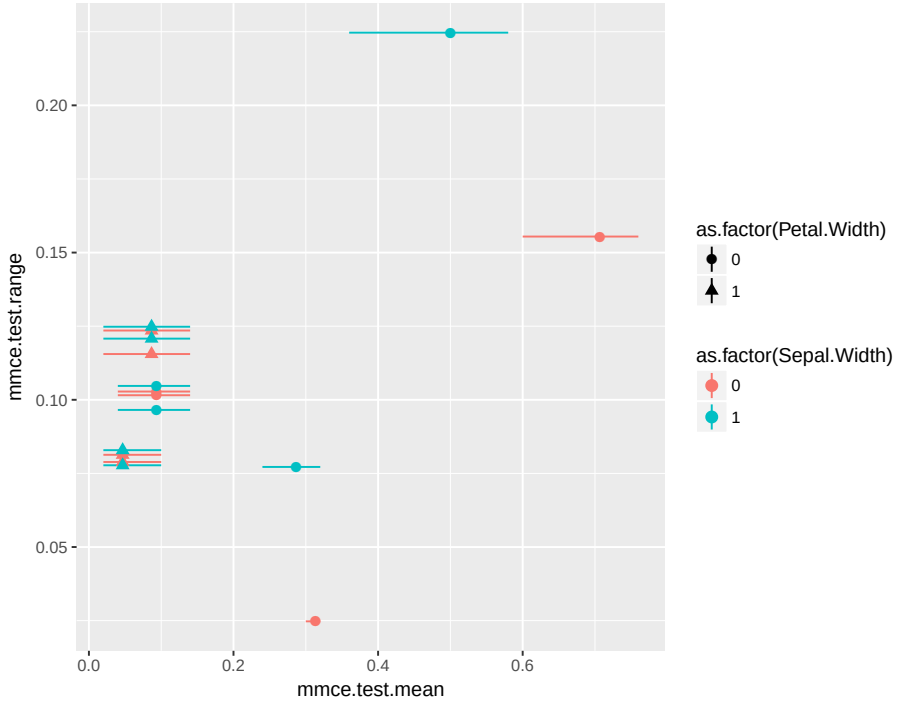
```
my.range.aggr = makeAggregation(id = "test.range", name = "Test
    Range",
  properties = "req.test",
  fun = function (task, perf.test, perf.train, measure, group, pred)
    diff(range(perf.test))
)
```

`perf.train` and `perf.test` are both numerical vectors containing the performances on the train and test data sets. In most cases (unless you are using bootstrap as resampling strategy or have set `predict = "both"` in makeResampleDesc) the `perf.train` vector is empty.

Now we can run a feature selection based on the first measure in the provided list and see how the other measures turn out.

```
### mmce with default aggregation scheme test.mean
ms1 = mmce

### mmce with new aggregation scheme my.range.aggr
ms2 = setAggregation(ms1, my.range.aggr)

### Minimum and maximum of the mmce over test sets
ms1min = setAggregation(ms1, test.min)
ms1max = setAggregation(ms1, test.max)

### Feature selection
rdesc = makeResampleDesc("CV", iters = 3)
res = selectFeatures("classif.rpart", iris.task, rdesc, measures =
    list(ms1, ms2, ms1min, ms1max),
  control = makeFeatSelControlExhaustive(), show.info = FALSE)

### Optimization path, i.e., performances for the 16 possible
    feature subsets
perf.data = as.data.frame(res$opt.path)
head(perf.data[1:8])
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width mmce.test.mean
#> 1            0           0            0           0     0.70666667
#> 2            1           0            0           0     0.31333333
#> 3            0           1            0           0     0.50000000
#> 4            0           0            1           0     0.09333333
#> 5            0           0            0           1     0.04666667
#> 6            1           1            0           0     0.28666667
#>   mmce.test.range mmce.test.min mmce.test.max
#> 1            0.16          0.60          0.76
```

```
#> 2           0.02          0.30          0.32
#> 3           0.22          0.36          0.58
#> 4           0.10          0.04          0.14
#> 5           0.08          0.02          0.10
#> 6           0.08          0.24          0.32

pd = position_jitter(width = 0.005, height = 0)
p = ggplot(aes(x = mmce.test.range, y = mmce.test.mean, ymax =
    mmce.test.max, ymin = mmce.test.min,
  color = as.factor(Sepal.Width), pch = as.factor(Petal.Width)),
      data = perf.data) +
  geom_pointrange(position = pd) +
  coord_flip()
print(p)
```




The plot shows the range versus the mean misclassification error. The value on the y-axis thus corresponds to the length of the error bars. (Note that the points and error bars are jittered in y-direction.)

## 4.6 Creating an Imputation Method

Function makeImputeMethod permits to create your own imputation method. For this purpose you need to specify a *learn* function that extracts the necessary information and an *impute* function that does the actual imputation. The *learn* and *impute* functions both have at least the following formal arguments:

- `data` is a data.frame with missing values in some features.
- `col` indicates the feature to be imputed.
- `target` indicates the target variable(s) in a supervised learning task.

### 4.6.1 Example: Imputation using the mean

Let's have a look at function imputeMean.

```
imputeMean = function() {
  makeImputeMethod(learn = function(data, target, col)
      mean(data[[col]], na.rm = TRUE),
    impute = simpleImpute)
}
```

imputeMean calls the unexported mlr function `simpleImpute` which is defined as follows.

```
simpleImpute = function(data, target, col, const) {
  if (is.na(const))
    stopf("Error imputing column '%s'. Maybe all input data was
        missing?", col)
  x = data[[col]]
  if (is.logical(x) && !is.logical(const)) {
    x = as.factor(x)
  }
  if (is.factor(x) && const %nin% levels(x)) {
    levels(x) = c(levels(x), as.character(const))
  }
  replace(x, is.na(x), const)
}
```

The *learn* function calculates the mean of the non-missing observations in column `col`. The mean is passed via argument `const` to the *impute* function that replaces all missing values in feature `col`.

### 4.6.2 Writing your own imputation method

Now let's write a new imputation method: A frequently used simple technique for longitudinal data is *last observation carried forward* (LOCF). Missing values are replaced by the most recent observed value.

In the **R** code below the *learn* function determines the last observed value previous to each `NA` (`values`) as well as the corresponding number of consecutive `NA`'s (`times`). The *impute* function generates a vector by replicating the entries in `values` according to `times` and replaces the `NA`'s in feature `col`.

```
imputeLOCF = function() {
  makeImputeMethod(
    learn = function(data, target, col) {
      x = data[[col]]
      ind = is.na(x)
      dind = diff(ind)
      lastValue = which(dind == 1)  # position of the last observed
          value previous to NA
      lastNA = which(dind == -1)    # position of the last of
          potentially several consecutive NA's
      values = x[lastValue]         # last observed value previous
          to NA
      times = lastNA - lastValue    # number of consecutive NA's
      return(list(values = values, times = times))
    },
    impute = function(data, target, col, values, times) {
      x = data[[col]]
      replace(x, is.na(x), rep(values, times))
    }
  )
}
```

Note that this function is just for demonstration and is lacking some checks for real-world usage (for example 'What should happen if the first value in `x` is already missing?'). Below it is used to impute the missing values in features `Ozone` and `Solar.R` in the airquality data set.

```
data(airquality)
imp = impute(airquality, cols = list(Ozone = imputeLOCF(), Solar.R =
    imputeLOCF()),
  dummy.cols = c("Ozone", "Solar.R"))
head(imp$data, 10)
#>    Ozone Solar.R Wind Temp Month Day Ozone.dummy Solar.R.dummy
#> 1     41     190  7.4   67     5   1       FALSE         FALSE
#> 2     36     118  8.0   72     5   2       FALSE         FALSE
#> 3     12     149 12.6   74     5   3       FALSE         FALSE
#> 4     18     313 11.5   62     5   4       FALSE         FALSE
#> 5     18     313 14.3   56     5   5        TRUE          TRUE
#> 6     28     313 14.9   66     5   6       FALSE          TRUE
#> 7     23     299  8.6   65     5   7       FALSE         FALSE
#> 8     19      99 13.8   59     5   8       FALSE         FALSE
#> 9      8      19 20.1   61     5   9       FALSE         FALSE
```

```
#> 10     8     194  8.6   69     5  10        TRUE         FALSE
```

## 4.7 Integrating Another Filter Method

A lot of feature filter methods are already integrated in mlr and a complete list is given in the Appendix or can be obtained using listFilterMethods. You can easily add another filter, be it a brand new one or a method which is already implemented in another package, via function makeFilter.

### 4.7.1 Filter objects

In mlr all filter methods are objects of class Filter and are registered in an environment called `.FilterRegister` (where listFilterMethods looks them up to compile the list of available methods). To get to know their structure let's have a closer look at the `"rank.correlation"` filter which interfaces function correls in package Rfast.

```
filters = as.list(mlr:::.FilterRegister)
filters$rank.correlation
#> Filter: 'rank.correlation'
#> Packages: 'Rfast'
#> Supported tasks: regr
#> Supported features: numerics

str(filters$rank.correlation)
#> List of 6
#>  $ name              : chr "rank.correlation"
#>  $ desc              : chr "Spearman's correlation between
    feature and target"
#>  $ pkg               : chr "Rfast"
#>  $ supported.tasks   : chr "regr"
#>  $ supported.features: chr "numerics"
#>  $ fun               :function (task, nselect, ...)
#>  - attr(*, "class")= chr "Filter"

filters$rank.correlation$fun
#> function (task, nselect, ...)
#> {
#>     d = getTaskData(task, target.extra = TRUE)
#>     y = Rfast::correls(d$target, d$data, type = "spearman")
#>     for (i in which(is.na(y[, "correlation"]))) {
#>         y[i, "correlation"] = cor(d$target, d$data[, i], use =
    "complete.obs",
#>             method = "spearman")
```

```
#>     }
#>     setNames(abs(y[, "correlation"]), getTaskFeatureNames(task))
#> }
#> <bytecode: 0xdc268f0>
#> <environment: namespace:mlr>
```

The core element is `$fun` which calculates the feature importance. For the `"rank.correlation"` filter it just extracts the data and formula from the `task` and passes them on to the correls function.

Additionally, each Filter object has a `$name`, which should be short and is for example used to annotate graphics (cp. plotFilterValues), and a slightly more detailed description in slot `$desc`. If the filter method is implemented by another package its name is given in the `$pkg` member. Moreover, the supported task types and feature types are listed.

### 4.7.2 Writing a new filter method

You can integrate your own filter method using makeFilter. This function generates a Filter object and also registers it in the `.FilterRegister` environment.

The arguments of makeFilter correspond to the slot names of the Filter object above. Currently, feature filtering is only supported for supervised learning tasks and possible values for `supported.tasks` are `"regr"`, `"classif"` and `"surv"`. `supported.features` can be `"numerics"`, `"factors"` and `"ordered"`.

`fun` must be a function with at least the following formal arguments:

- `task` is a mlr learning Task.
- `nselect` corresponds to the argument of generateFilterValuesData of the same name and specifies the number of features for which to calculate importance scores. Some filter methods have the option to stop after a certain number of top-ranked features have been found in order to save time and ressources when the number of features is high. The majority of filter methods integrated in mlr doesn't support this and thus `nselect` is ignored in most cases. An exception is the minimum redundancy maximum relevance filter from package mRMRe.
- `...` for additional arguments.

`fun` must return a named vector of feature importance values. By convention the most important features receive the highest scores.

If you are making use of the `nselect` option `fun` can either return a vector of `nselect` scores or a vector as long as the total numbers of features in the task filled with `NA`s for all features whose scores weren't calculated.

When writing `fun` many of the getter functions for Tasks come in handy, particularly getTaskData, getTaskFormula and getTaskFeatureNames. It's worth

having a closer look at getTaskData which provides many options for formatting the data and recoding the target variable.

As a short demonstration we write a totally meaningless filter that determines the importance of features according to alphabetical order, i.e., giving highest scores to features with names that come first (`decreasing = TRUE`) or last (`decreasing = FALSE`) in the alphabet.

```
makeFilter(
  name = "nonsense.filter",
  desc = "Calculates scores according to alphabetical order of
      features",
  pkg = "",
  supported.tasks = c("classif", "regr", "surv"),
  supported.features = c("numerics", "factors", "ordered"),
  fun = function(task, nselect, decreasing = TRUE, ...) {
    feats = getTaskFeatureNames(task)
    imp = order(feats, decreasing = decreasing)
    names(imp) = feats
    imp
  }
)
#> Filter: 'nonsense.filter'
#> Packages: ''
#> Supported tasks: classif,regr,surv
#> Supported features: numerics,factors,ordered
```

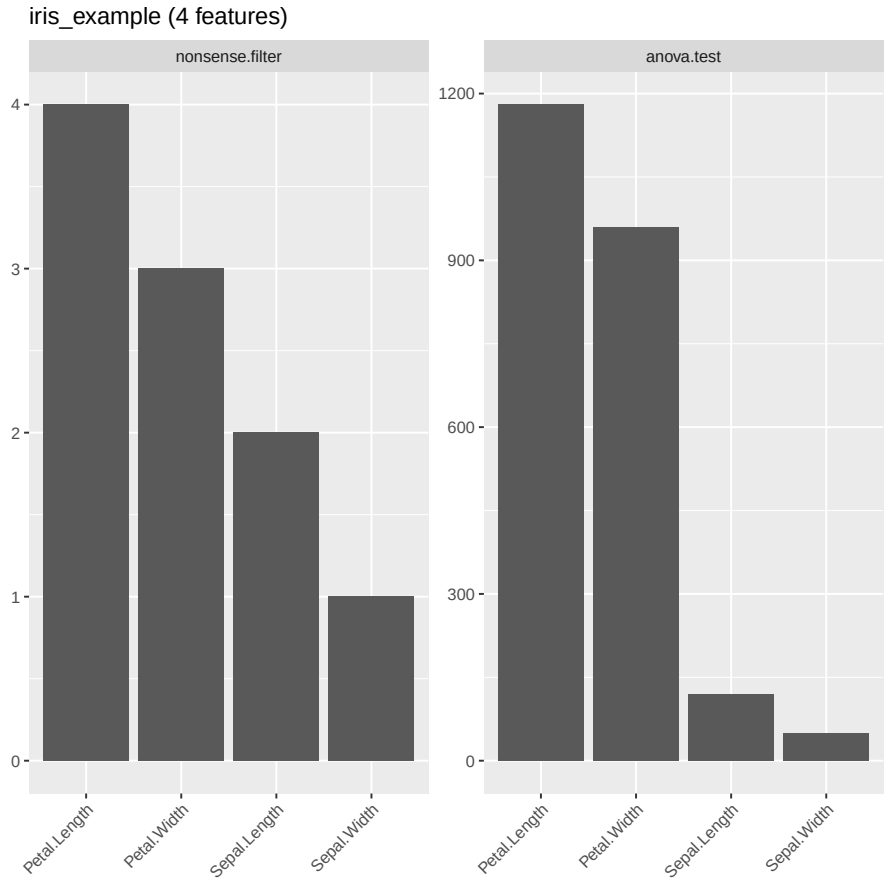
The `nonsense.filter` is now registered in mlr and shown by listFilterMethods.

```
listFilterMethods()$id
#>  [1] anova.test                 auc
#>  [3] carscore                   cforest.importance
#>  [5] chi.squared                gain.ratio
#>  [7] information.gain           kruskal.test
#>  [9] linear.correlation         mrmr
#> [11] nonsense.filter            oneR
#> [13] permutation.importance     randomForest.importance
#> [15] randomForestSRC.rfsrc      randomForestSRC.var.select
#> [17] rank.correlation           relief
#> [19] symmetrical.uncertainty    univariate.model.score
#> [21] variance
#> 24 Levels: anova.test auc carscore cforest.importance ... variance
```

You can use it like any other filter method already integrated in mlr (i.e., via the `method` argument of generateFilterValuesData or the `fw.method` argument of makeFilterWrapper; see also the page on feature selection).

```
d = generateFilterValuesData(iris.task, method =
    c("nonsense.filter", "anova.test"))
```

```
d
#> FilterValues:
#> Task: iris_example
#>           name    type nonsense.filter anova.test
#> 1 Sepal.Length numeric               2  119.26450
#> 2  Sepal.Width numeric               1   49.16004
#> 3 Petal.Length numeric               4 1180.16118
#> 4  Petal.Width numeric               3  960.00715

plotFilterValues(d)
```




```
iris.task.filtered = filterFeatures(iris.task, method =
    "nonsense.filter", abs = 2)
iris.task.filtered
#> Supervised task: iris_example
#> Type: classif
```

```
#> Target: Species
#> Observations: 150
#> Features:
#>    numerics     factors     ordered functionals
#>           2           0           0           0
#> Missings: FALSE
#> Has weights: FALSE
#> Has blocking: FALSE
#> Is spatial: FALSE
#> Classes: 3
#>     setosa versicolor  virginica
#>         50         50         50
#> Positive class: NA

getTaskFeatureNames(iris.task.filtered)
#> [1] "Petal.Length" "Petal.Width"
```

You might also want to have a look at the source code of the filter methods already integrated in mlr for some more complex and meaningful examples.