mlr3 book

Michel Lang Patrick Schratz Martin Binder Florian Pfisterer
Jakob Richter Nicholas G. Reich Bernd Bischl

2020-02-24

Contents

Quickstart

As a 30-second introductory example, we will train a decision tree model on the first 120 rows of iris data set and make predictions on the final 30, measuring the accuracy of the trained model.

```
library("mlr3")
task = tsk("iris")
learner = lrn("classif.rpart")

# train a model of this learner for a subset of the task
learner$train(task, row_ids = 1:120)
# this is what the decision tree looks like
learner$model
```

```
## n= 120
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
## 1) root 120 70 setosa (0.41667 0.41667 0.16667)
##   2) Petal.Length< 2.45 50  0 setosa (1.00000 0.00000 0.00000) *
##   3) Petal.Length>=2.45 70 20 versicolor (0.00000 0.71429 0.28571)
##     6) Petal.Length< 4.95 49  1 versicolor (0.00000 0.97959 0.02041) *
##     7) Petal.Length>=4.95 21  2 virginica (0.00000 0.09524 0.90476) *
```

```
predictions = learner$predict(task, row_ids = 121:150)
predictions
```

```
## <PredictionClassif> for 30 observations:
##     row_id     truth   response
##        121 virginica  virginica
##        122 virginica versicolor
##        123 virginica  virginica
## ---
##        148 virginica  virginica
##        149 virginica  virginica
##        150 virginica  virginica
```

```
# accuracy of our model on the test set of the final 30 rows
predictions$score(msr("classif.acc"))
```

```
## classif.acc
##      0.8333
```

More examples can be found in the mlr3gallery, a collection of use cases and examples.

1 Introduction and Overview

The mlr3 (Lang et al. 2019) package and ecosystem provide a generic, object-oriented, and extensible framework for classification, regression, survival analysis, and other machine learning tasks for the R language (R Core Team 2019). We do not implement any learners ourselves, but provide a unified interface to many existing learners in R. This unified interface provides functionality to extend and combine existing learners, intelligently select and tune the most appropriate technique for a task, and perform large-scale comparisons that enable meta-learning. Examples of this advanced functionality include hyperparameter tuning, feature selection, and ensemble construction. Parallelization of many operations is natively supported.

Target Audience

mlr3 provides a domain-specific language for machine learning in R. We target both **practitioners** who want to quickly apply machine learning algorithms and **researchers** who want to implement, benchmark, and compare their new methods in a structured environment. The package is a complete rewrite of an earlier version of mlr that leverages many years of experience to provide a state-of-the-art system that is easy to use and extend. It is intended for users who have basic knowledge of machine learning and R and who are interested in complex projects that use advanced functionality as well as one-liners to quickly prototype specific tasks.

Why a Rewrite?

mlr (Bischl et al. 2016) was first released to CRAN in 2013, with the core design and architecture dating back much further. Over time, the addition of many features has led to a considerably more complex design that made it harder to build, maintain, and extend than we had hoped for. With hindsight, we saw that some of the design and architecture choices in mlr made it difficult to support new features, in particular with respect to pipelines. Furthermore, the R ecosystem as well as helpful packages such as data.table have undergone major changes in the meantime. It would have been nearly impossible to integrate all of these changes into the original design of mlr. Instead, we decided to start working on a reimplementation in 2018, which resulted in the first release of mlr3 on CRAN in July 2019. The new design and the integration of further and newly developed R packages (R6, future, data.table) makes mlr3 much easier to use, maintain, and more efficient compared to mlr.

Design Principles

We follow these general design principles in the mlr3 package and ecosystem.

- Backend over frontend. Most packages of the mlr3 ecosystem focus on processing and transforming data, applying machine learning algorithms, and computing results. We do not provide graphical user interfaces (GUIs); visualizations of data and results are provided in extra packages.
- Embrace R6 for a clean, object-oriented design, object state-changes, and reference semantics.
- Embrace data.table for fast and convenient data frame computations.
- Unify container and result classes as much as possible and provide result data in `data.table`s. This considerably simplifies the API and allows easy selection and “split-apply-combine” (ag-

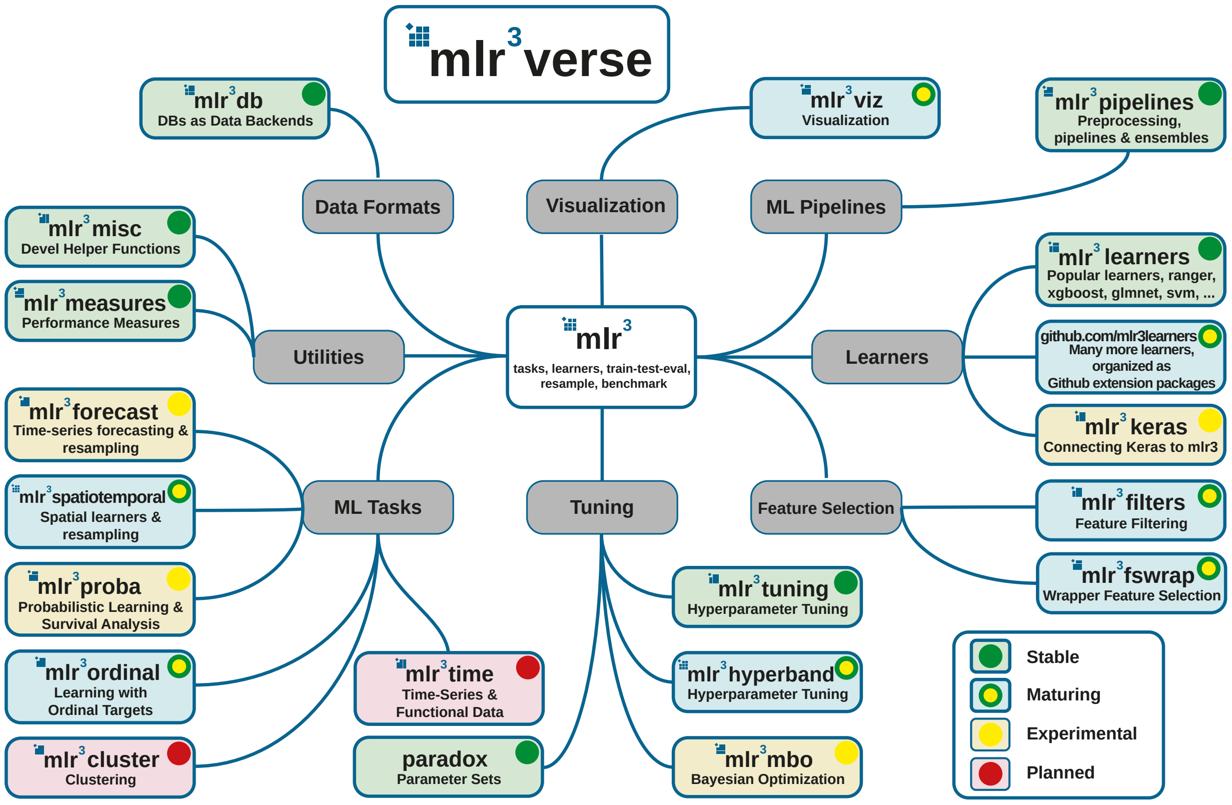
gregation) operations. We combine `data.table` and `R6` to place references to non-atomic and compound objects in tables and make heavy use of list columns.
- Defensive programming and type safety. All user input is checked with `checkmate` (Lang 2017). Return types are documented, and mechanisms popular in base R which "simplify" the result unpredictably (e.g., `sapply()` or the `drop` argument in `[.data.frame`) are avoided.
- Be light on dependencies. One of the main maintenance burdens for mlr was to keep up with changing learner interfaces and behavior of the many packages it depended on. We require far fewer packages in mlr3 to make installation and maintenance easier.

mlr3 requires the following packages:

- backports: Ensures backward compatibility with older R releases. Developed by members of the mlr3 team.
- checkmate: Fast argument checks. Developed by members of the mlr3 team.
- mlr3misc: Miscellaneous functions used in multiple mlr3 extension packages. Developed by the mlr3 team.
- mlr3measures: Performance measures for classification and regression. Developed by members of the mlr3 team.
- paradox: Descriptions of parameters and parameter sets. Developed by the mlr3 team.
- R6: Reference class objects.
- data.table: Extension of R's `data.frame`.
- digest: Hash digests.
- uuid: Unique string identifiers.
- lgr: Logging facility.
- mlbench: A collection of machine learning data sets.

None of these packages adds any extra recursive dependencies to mlr3.

mlr3 provides additional functionality through extra packages:

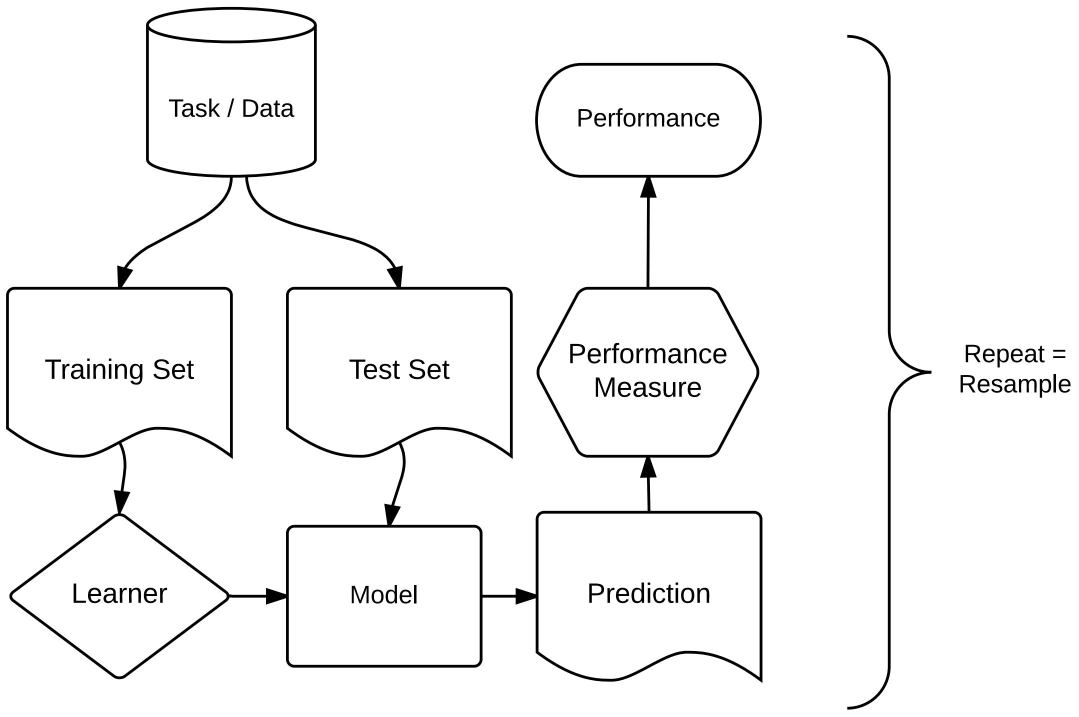

mlr3verse
mlr3db DBs as Data Backends
mlr3viz Visualization
mlr3pipelines Preprocessing, pipelines & ensembles
Data Formats
Visualization
ML Pipelines
mlr3misc Devel Helper Functions
mlr3measures Performance Measures
Utilities
mlr3 tasks, learners, train-test-eval, resample, benchmark
Learners
mlr3learners Popular learners, ranger, xgboost, glmnet, svm, ...
github.com/mlr3learners Many more learners, organized as Github extension packages
mlr3keras Connecting Keras to mlr3
mlr3forecast Time-series forecasting & resampling
mlr3spatiotemporal Spatial learners & resampling
ML Tasks
Tuning
Feature Selection
mlr3filters Feature Filtering
mlr3fswrap Wrapper Feature Selection
mlr3proba Probabilistic Learning & Survival Analysis
mlr3tuning Hyperparameter Tuning
mlr3ordinal Learning with Ordinal Targets
mlr3time Time-Series & Functional Data
mlr3hyperband Hyperparameter Tuning
mlr3cluster Clustering
paradox Parameter Sets
mlr3mbo Bayesian Optimization
Stable
Maturing
Experimental
Planned

A complete list with links to the respective repository can be found on the wiki page on extension packages.

- For parallelization, mlr3 utilizes the future and future.apply packages.
- To enable progress bars, use progressr.
- To capture output, warnings, and exceptions, evaluate and callr can be used.

2 Basics

This chapter will teach you the essential building blocks, R6 classes, and operations of mlr3 for machine learning. A typical machine learning workflow looks like this:



The data, which mlr3 encapsulates in tasks, is split into non-overlapping training and test sets. We are interested in models that generalize to new data rather than just memorizing the training data, and separate test data allows to objectively evaluate models with respect to that. The training data is given to a machine learning algorithm, which we call a learner in mlr3. The learner uses the training data to build a model of the relationship of the input features to the output target values. This model is then used to produce predictions on the test data, which are compared to the ground truth values to assess the quality of the model. mlr3 offers a number of different measures to quantify how well a model performs based on the difference between predicted and actual values. Usually this measure is a numeric score.

The process of splitting up data into training and test sets, building a model, and evaluating it may be repeated several times, resampling different training and test sets from the original data each time. Multiple resampling iterations allow us to get a better, more generalizable performance estimate for a particular type of model as it is tested under different conditions and less likely to get lucky or unlucky because of a particular way the data was resampled.

In many cases, this simple workflow is not sufficient to deal with real-world data, which may require normalization, imputation of missing values, or feature selection. We will cover more complex workflows that allow to do this and even more later in the book.

This chapter covers the following subtopics:

Tasks

Tasks encapsulate the data with meta-information, such as the name of the prediction target column. We cover how to:

- access predefined tasks,
- specify a task type,
- create a task,
- work with a task's API,
- assign roles to rows and columns of a task,
- implement task mutators, and
- retrieve the data that is stored in a task.

Learners

Learners encapsulate machine learning algorithms to train models and make predictions for a task. They are provided by R and other packages. We cover how to:

- access the set of classification and regression learners that come with mlr3 and retrieve a specific learner,
- access the set of hyperparameter values of a learner and modify them.

How to modify and extend learners is covered in a supplemental advanced technical section.

Train and predict

The section on the train and predict methods illustrates how to use tasks and learners to train a model and make predictions on a new data set. In particular, we cover how to:

- set up tasks and learners properly,
- set up train and test splits for a task,
- train the learner on the training set to produce a model,
- generate predictions on the test set, and
- assess the performance of the model by comparing predicted and actual values.

Resampling

A resampling is a method to create training and test splits. We cover how to

- access and select resampling strategies,
- instantiate the split into training and test sets by applying the resampling, and
- execute the resampling to obtain results.

Additional information on resampling can be found in the section about nested resampling and in the chapter on model optimization.

Benchmarking

Benchmarking is used to compare the performance of different models, for example models trained with different learners, on different tasks, or with different resampling methods. We cover how to

- create a benchmarking design,
- execute a design and aggregate results, and
- convert benchmarking objects to resample objects.

Binary classification

Binary classification is a special case of classification where the target variable to predict has only two possible values. In this case, additional considerations apply; in particular:

- ROC curves and the threshold where to predict one class versus the other, and
- threshold tuning (WIP).

Before we get into the details of how to use mlr3 for machine learning, we give a brief introduction to R6 as it is a relatively new part of R. mlr3 heavily relies on R6 and all basic building blocks it provides are R6 classes:

- tasks,
- learners,
- measures, and
- resamplings.

2.1 Quick R6 Intro for Beginners

R6 is one of R's more recent dialects for object-oriented programming (OO). It addresses shortcomings of earlier OO implementations in R, such as S3, which we used in mlr. If you have done any object-oriented programming before, R6 should feel familiar. We focus on the parts of R6 that you need to know to use mlr3 here.

- Objects are created by calling the constructor of an `R6Class()` object, specifically the `$new()` method. For example, `foo = Foo$new(bar = 1)` creates a new object of class `Foo`, setting the `bar` argument of the constructor to `1`.
- Classes have mutable state, which is encapsulated in their fields, which can be accessed through the dollar operator. We can access the `bar` value in the `Foo` class through `foo$bar` and set its value by assigning the field, e.g. `foo$bar = 2`.
- In addition to fields, objects expose methods that may allow to inspect the object's state, retrieve information, or perform an action that may change the internal state of the object. For example, the `$train` method of a learner changes the internal state of the learner by building and storing a trained model, which can then be used to make predictions given data.
- Objects can have public and private fields and methods. In mlr3, you can only access the public variables and methods. Private fields and methods are only relevant to change or extend mlr3.
- R6 variables are references to objects rather then the actual objects, which are stored in an environment. For example `foo2 = foo` does not create a copy of `foo` in `foo2`, but another reference to the same actual object. Setting `foo$bar = 3` will also change `foo2$bar` to `3` and vice versa.
- To copy an object, use the `$clone()` method and the `deep = TRUE` argument for nested objects, for example `foo2 = foo$clone(deep = TRUE)`.

For further information see wiki for short descriptions and links to the respective repositories.

For more details on R6, have a look at the R6 vignettes.

2.2 Tasks

Tasks are objects that contain the data and additional meta-data for a machine learning problem. The meta-data is for example the name of the target variable (the prediction) for supervised machine learning problems, or the type of the dataset (e.g. a *spatial* or *survival*). This information is used for specific operations that can be performed on a task.

2.2.1 Task Types

To create a task from a `data.frame()` or `data.table()` object, the task type needs to be specified:

Classification Task: The target is a label (stored as `character()`or`factor()`) with only few distinct values. → `mlr3::TaskClassif`

Regression Task: The target is a numeric quantity (stored as `integer()` or `double()`). → `mlr3::TaskRegr`

Survival Task: The target is the (right-censored) time to an event. → `mlr3proba::TaskSurv` in add-on package mlr3proba

Ordinal Regression Task: The target is ordinal. → `TaskOrdinal` in add-on package mlr3ordinal

Cluster Task: An unsupervised task type; there is no target and the aim is to identify similar groups within the feature space. → Not yet implemented

Spatial Task: Observations in the task have spatio-temporal information (e.g. coordinates). → Not yet implemented, but started in add-on package mlr3spatiotemporal

2.2.2 Task Creation

As an example, we will create a regression task using the `mtcars` data set from the package `datasets` and predict the target `"mpg"` (miles per gallon). We only consider the first two features in the dataset for brevity.

First, we load and prepare the data.

```
data("mtcars", package = "datasets")
data = mtcars[, 1:3]
str(data)
```

```
## 'data.frame':    32 obs. of  3 variables:
##  $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
##  $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
##  $ disp: num  160 160 108 258 360 ...
```

Next, we create the task using the constructor for a regression task object (`TaskRegr$new`) and give the following information:

1. `id`: An arbitrary identifier for the task, used in plots and summaries.
2. `backend`: This parameter allows fine-grained control over how data is accessed. Here, we simply provide the dataset which is automatically converted to a `DataBackendDataTable`. Alternatively, we could also construct a `DataBackend` manually.
3. `target`: The name of the target column for the regression problem.

```
library("mlr3")

task_mtcars = TaskRegr$new(id = "cars", backend = data, target = "mpg")
print(task_mtcars)
```

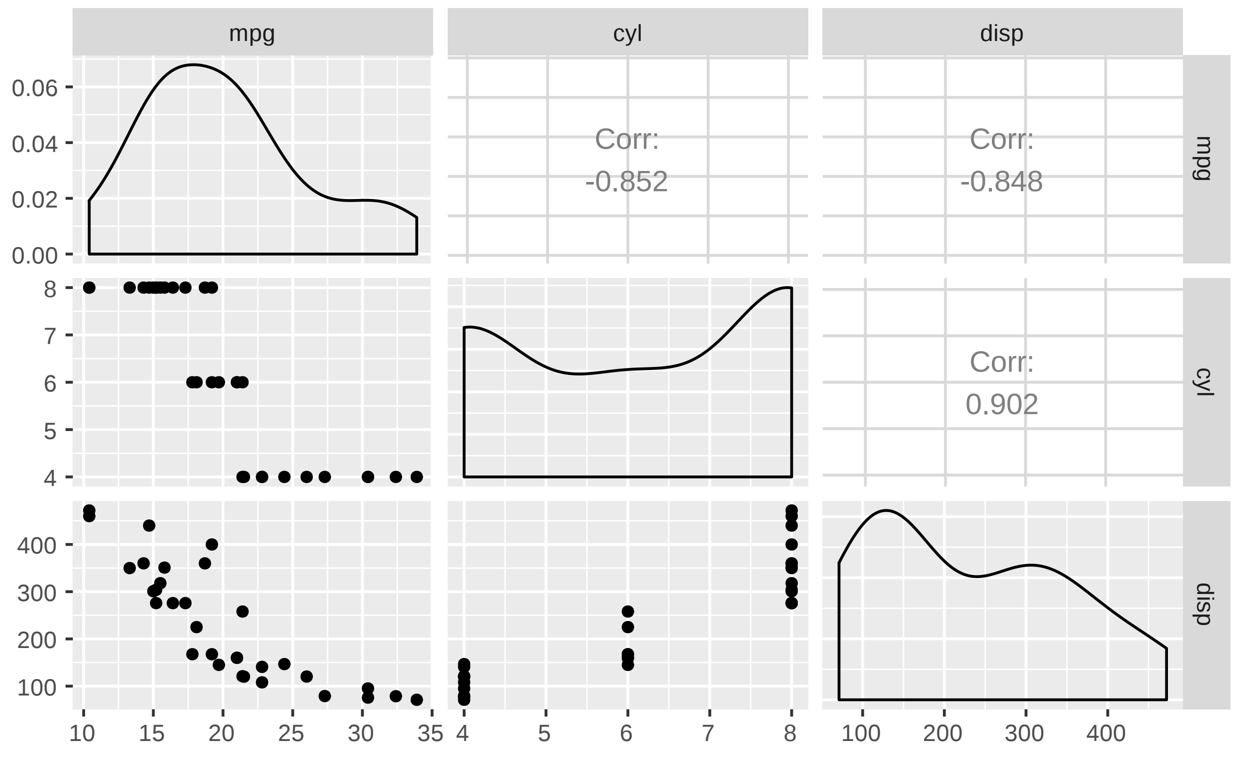
```
## <TaskRegr:cars> (32 x 3)
## * Target: mpg
## * Properties: -
## * Features (2):
##   - dbl (2): cyl, disp
```

The `print()` method gives a short summary of the task: It has 32 observations and 3 columns, of which 2 are features.

We can also plot the task using the mlr3viz package, which gives a graphical summary of its properties:

```
library("mlr3viz")
autoplot(task_mtcars, type = "pairs")
```

```
## Registered S3 method overwritten by 'GGally':
##   method from
##   +.gg   ggplot2
```



2.2.3 Predefined tasks

mlr3 ships with a few predefined machine learning tasks. All tasks are stored in an R6 `Dictionary` (a key-value store) named `mlr_tasks`. Printing it gives the keys (the names of the datasets):

```
mlr_tasks
```

```
## <DictionaryTask> with 9 stored values
## Keys: boston_housing, german_credit, iris, mtcars, pima, sonar, spam,
##   wine, zoo
```

We can get a more informative summary of the example tasks by converting the dictionary to a `data.table()` object:

```
library("data.table")
as.data.table(mlr_tasks)
```

```
##                key task_type nrow ncol lgl int dbl chr fct ord pxc
## 1: boston_housing      regr  506   19   0   3  13   0   2   0   0
## 2:  german_credit   classif 1000   21   0   0   7   0  12   1   0
## 3:           iris   classif  150    5   0   0   4   0   0   0   0
## 4:         mtcars      regr   32   11   0   0  10   0   0   0   0
## 5:           pima   classif  768    9   0   0   8   0   0   0   0
## 6:          sonar   classif  208   61   0   0  60   0   0   0   0
## 7:           spam   classif 4601   58   0   0  57   0   0   0   0
## 8:           wine   classif  178   14   0   2  11   0   0   0   0
## 9:            zoo   classif  101   17  15   1   0   0   0   0   0
```

In the above display, the columns "lgl" (logical), "int" (integer), "dbl" (double), "chr" (character), "fct" (factor) and "ord" (ordinal) display the number of features (or columns) in the dataset with the corresponding datatype.

To get a task from the dictionary, one can use the `$get()` method from the `mlr_tasks` class and assign the return value to a new object. For example, to use the iris data set for classification:

```
task_iris = mlr_tasks$get("iris")
print(task_iris)
```

```
## <TaskClassif:iris> (150 x 5)
## * Target: Species
## * Properties: multiclass
## * Features (4):
##   - dbl (4): Petal.Length, Petal.Width, Sepal.Length, Sepal.Width
```

Alternatively, you can also use the convenience function `tsk()`, which also constructs a task from the dictionary.

```
tsk("iris")
```

```
## <TaskClassif:iris> (150 x 5)
## * Target: Species
## * Properties: multiclass
## * Features (4):
##   - dbl (4): Petal.Length, Petal.Width, Sepal.Length, Sepal.Width
```

2.2.4 Task API

All task properties and characteristics can be queried using the task's public fields and methods (see `Task`). Methods are also used to change the behavior of the task.

2.2.4.1 Retrieving Data

The data stored in a task can be retrieved directly from fields, for example:

```
task_iris$nrow
```

```
## [1] 150
```

```
task_iris$ncol
```

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

More information can be obtained through methods of the object, for example:

```
task_iris$data()
```

```
##        Species Petal.Length Petal.Width Sepal.Length Sepal.Width
##   1:    setosa          1.4         0.2          5.1         3.5
##   2:    setosa          1.4         0.2          4.9         3.0
##   3:    setosa          1.3         0.2          4.7         3.2
##   4:    setosa          1.5         0.2          4.6         3.1
##   5:    setosa          1.4         0.2          5.0         3.6
##  ---
## 146: virginica          5.2         2.3          6.7         3.0
## 147: virginica          5.0         1.9          6.3         2.5
## 148: virginica          5.2         2.0          6.5         3.0
## 149: virginica          5.4         2.3          6.2         3.4
## 150: virginica          5.1         1.8          5.9         3.0
```

In mlr3, each row (observation) has a unique identifier, stored as an `integer()`. These can be passed as arguments to the `$data()` method to select specific rows:

```
head(task_iris$row_ids)
```

```
## [1] 1 2 3 4 5 6
```

```
# retrieve data for rows with ids 1, 51, and 101
task_iris$data(rows = c(1, 51, 101))
```

```
##       Species Petal.Length Petal.Width Sepal.Length Sepal.Width
## 1:     setosa          1.4         0.2          5.1         3.5
## 2: versicolor          4.7         1.4          7.0         3.2
## 3:  virginica          6.0         2.5          6.3         3.3
```

Similarly, target and feature columns also have unique identifiers, i.e. names. These names are stored in the public slots `$feature_names` and `$target_names`. Here "target" refers to the variable we want to predict and "feature" to the predictors for the task.

```
task_iris$feature_names
```

```
## [1] "Petal.Length" "Petal.Width"  "Sepal.Length" "Sepal.Width"
```

```
task_iris$target_names
```

```
## [1] "Species"
```

The `row_ids` and column names can be combined when selecting a subset of the data:

```
# retrieve data for rows 1, 51, and 101 and only select column "Species"
task_iris$data(rows = c(1, 51, 101), cols = "Species")
```

```
##       Species
## 1:     setosa
## 2: versicolor
## 3:  virginica
```

To extract the complete data from the task, one can simply convert it to a `data.table`:

```
summary(as.data.table(task_iris))
```

```
##        Species    Petal.Length    Petal.Width    Sepal.Length    Sepal.Width
##  setosa    :50   Min.   :1.00   Min.   :0.1   Min.   :4.30   Min.   :2.00
##  versicolor:50   1st Qu.:1.60   1st Qu.:0.3   1st Qu.:5.10   1st Qu.:2.80
##  virginica :50   Median :4.35   Median :1.3   Median :5.80   Median :3.00
##                  Mean   :3.76   Mean   :1.2   Mean   :5.84   Mean   :3.06
##                  3rd Qu.:5.10   3rd Qu.:1.8   3rd Qu.:6.40   3rd Qu.:3.30
##                  Max.   :6.90   Max.   :2.5   Max.   :7.90   Max.   :4.40
```

2.2.4.2 Roles (Rows and Columns)

It is possible to assign roles to rows and columns. These roles affect the behavior of the task for different operations. Furthermore, these roles provide additional meta-data for it.

For example, the previously-constructed *mtcars* task has the following column roles:

```
print(task_mtcars$col_roles)
```

```
## $feature
## [1] "cyl"  "disp"
##
## $target
## [1] "mpg"
##
## $name
## character(0)
##
## $order
## character(0)
##
## $stratum
## character(0)
##
## $group
## character(0)
##
## $weight
## character(0)
```

To add the row names of `mtcars` as an additional feature, we first add them to the data table and then recreate the task.

```
# with `keep.rownames`, data.table stores the row names in an extra column "rn"
data = as.data.table(mtcars[, 1:3], keep.rownames = TRUE)
task = TaskRegr$new(id = "cars", backend = data, target = "mpg")

# there is a new feature called "rn"
task$feature_names
```

```
## [1] "cyl"  "disp" "rn"
```

The row names are now a feature whose values are stored in the column `"rn"`. We include this column here for educational purposes only. Generally speaking, there is no point in having a feature that uniquely identifies each row. Furthermore, the character data type will cause problems with many types of machine learning algorithms. The identifier may be useful to label points in plots and identify outliers however. To use the new column for only this purpose, we will change the role of the `"rn"` column and remove it from the set of active features. This is done by simply modifying the field `$col_roles`, which is a named list of vectors of column names. Each vector in this list corresponds to a column role, and the column names contained in that vector are designated as having that role. Supported column roles can be found in the manual of Task.

```
# supported column roles, see ?Task
names(task$col_roles)
```

```
## [1] "feature" "target"  "name"    "order"   "stratum" "group"   "weight"
```

```
# assign column "rn" the role "name"
task$col_roles$name = "rn"

# remove "rn" from role "feature"
task$col_roles$feature = setdiff(task$col_roles$feature, "rn")

# "rn" not listed as feature anymore
task$feature_names
```

```
## [1] "cyl"  "disp"
```

```
# "rn" also does not appear anymore when we access the data
task$data(rows = 1:2)
```

```
##    mpg cyl disp
## 1:  21   6  160
## 2:  21   6  160
```

```
task$head(2)
```

```
##    mpg cyl disp
## 1:  21   6  160
## 2:  21   6  160
```

Changing the role does not change the underlying data. Changing the role only changes the view on it. The data is not copied in the code above. The view is changed in-place though, i.e. the task object itself is modified.

Just like columns, it is also possible to assign different roles to rows.

Rows can have two different roles:

1. Role `use`: Rows that are generally available for model fitting (although they may also be used

as test set in resampling). This role is the default role.

2. Role `validation`: Rows that are not used for training. Rows that have missing values in the target column during task creation are automatically set to the validation role.

There are several reasons to hold some observations back or treat them differently:

1. It is often good practice to validate the final model on an external validation set to identify possible overfitting.
2. Some observations may be unlabeled, e.g. in competitions like Kaggle.

These observations cannot be used for training a model, but can be used to get predictions.

2.2.4.3 Task Mutators

As shown above, modifying `$col_roles` or `$row_roles` changes the view on the data. The additional convenience method `$filter()` subsets the current view based on row ids and `$select()` subsets the view based on feature names.

```
task = tsk("iris")
task$select(c("Sepal.Width", "Sepal.Length")) # keep only these features
task$filter(1:3) # keep only these rows
task$head()
```

```
##    Species Sepal.Length Sepal.Width
## 1:  setosa          5.1         3.5
## 2:  setosa          4.9         3.0
## 3:  setosa          4.7         3.2
```

While the methods discussed above allow to subset the data, the methods `$rbind()` and `$cbind()` allow to add extra rows and columns to a task. Again, the original data is not changed. The additional rows or columns are only added to the view of the data.

```
task$cbind(data.table(foo = letters[1:3])) # add column foo
task$head()
```

```
##    Species Sepal.Length Sepal.Width foo
## 1:  setosa          5.1         3.5   a
## 2:  setosa          4.9         3.0   b
## 3:  setosa          4.7         3.2   c
```

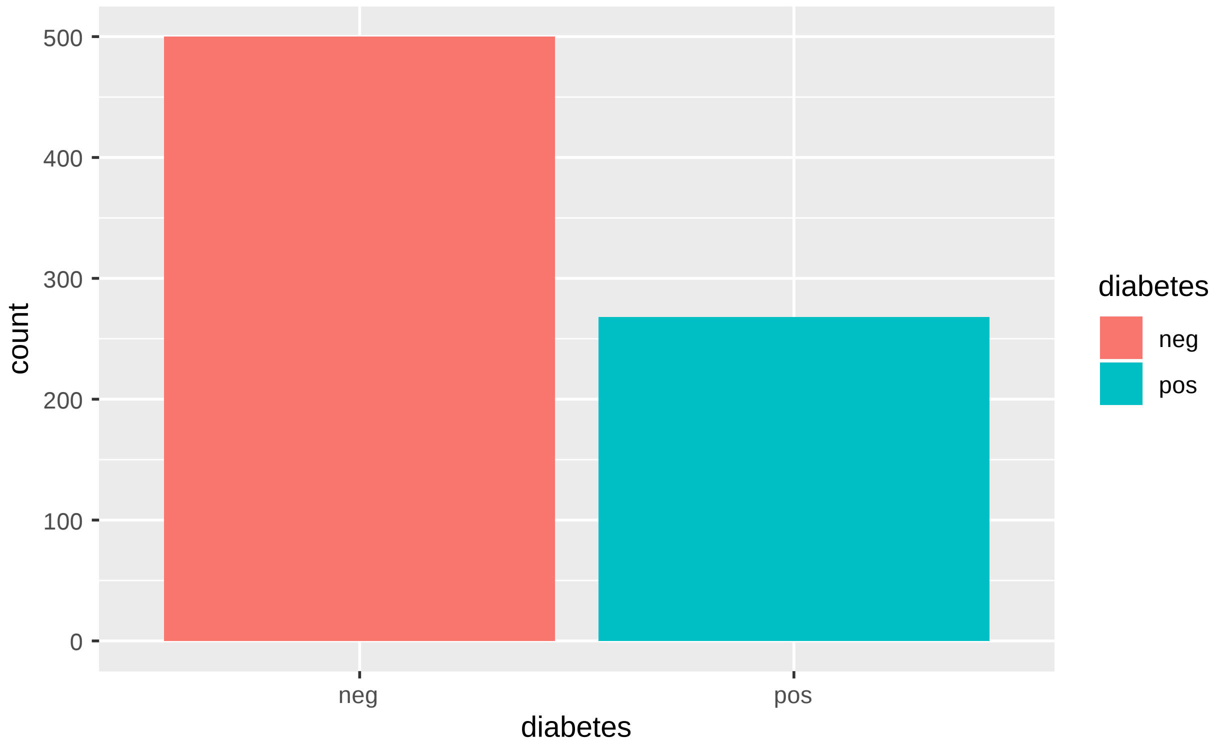
2.2.5 Plotting Tasks

The mlr3viz package provides plotting facilities for many classes implemented in mlr3. The available plot types depend on the inherited class, but all plots are returned as ggplot2 objects which can be easily customized.

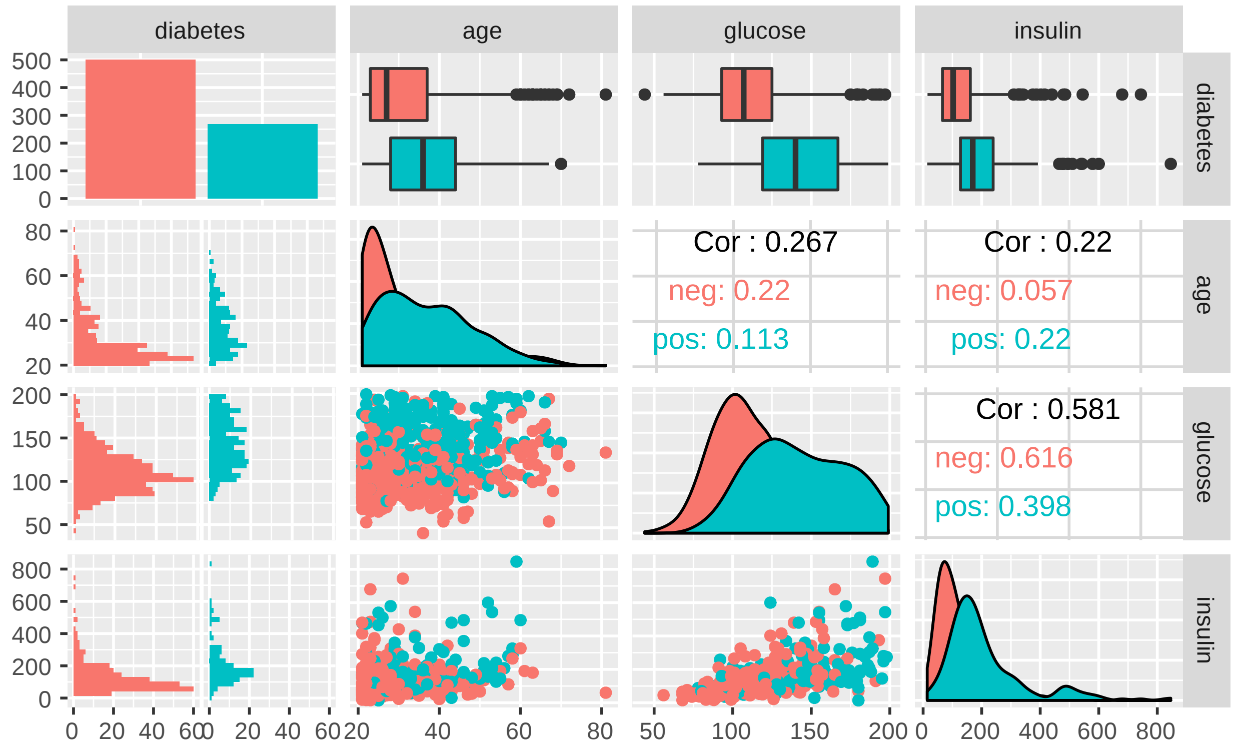
For classification tasks (inheriting from `TaskClassif`), see the documentation of `mlr3viz::autoplot.TaskClass` for the implemented plot types. Here are some examples to get an impression:

```
library("mlr3viz")

# get the pima indians task
task = tsk("pima")
```

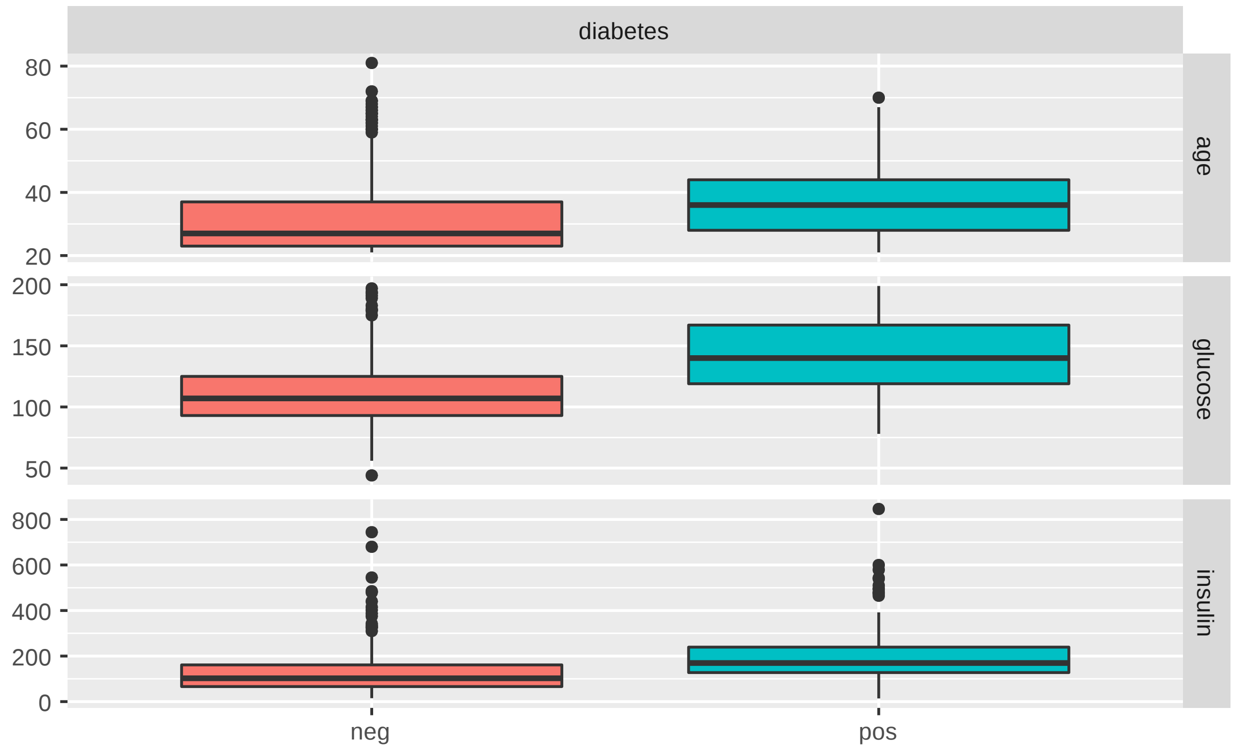
```r
# subset task to only use the 3 first features
task$select(head(task$feature_names, 3))

# default plot: class frequencies
autoplot(task)
```



```r
# pairs plot (requires package GGally)
autoplot(task, type = "pairs")
```

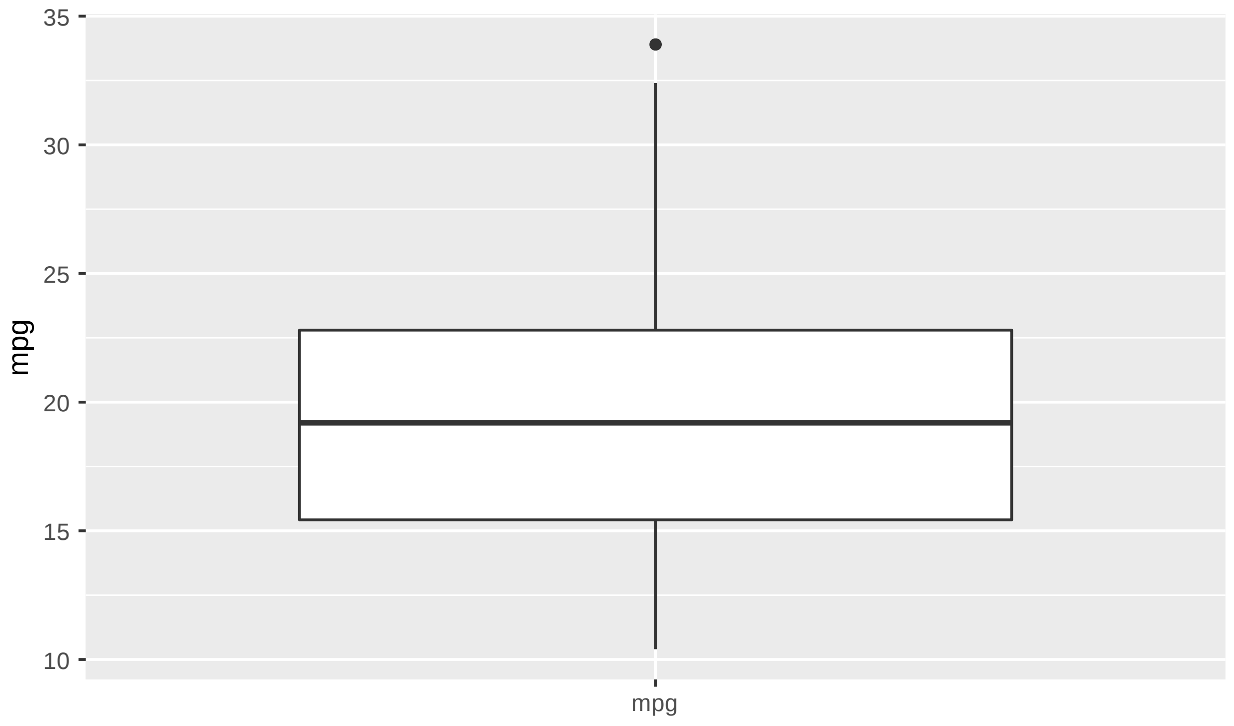


```r
# duo plot (requires package GGally)
autoplot(task, type = "duo")
```

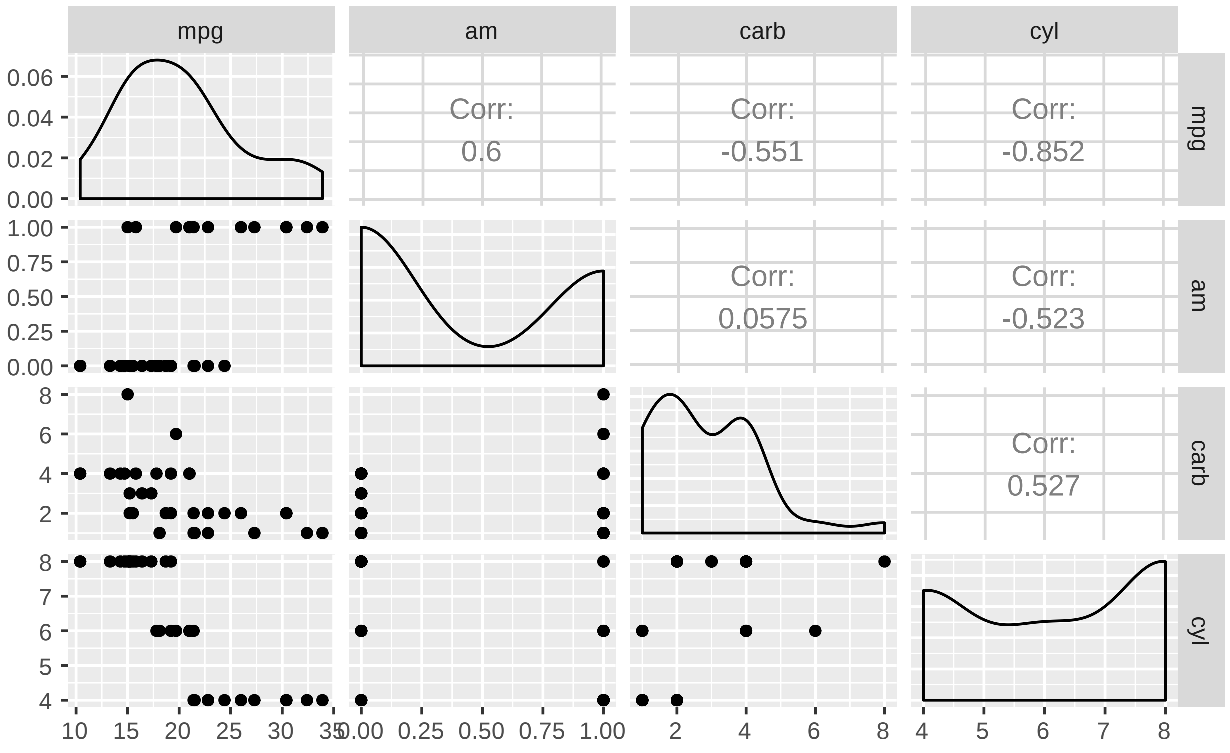


Of course, you can do the same for regression tasks (inheriting from `TaskRegr`) as documented in `mlr3viz::autoplot.TaskRegr`:

```r
library("mlr3viz")

# get the mtcars task
task = tsk("mtcars")

# subset task to only use the 3 first features
task$select(head(task$feature_names, 3))

# default plot: boxplot of target variable
autoplot(task)
```



```r
# pairs plot (requires package GGally)
autoplot(task, type = "pairs")
```



2.3 Learners

Objects of class `mlr3::Learner` provide a unified interface to many popular machine learning algorithms in R. They consist of methods to train and predict a model for a `mlr3::Task` and provide meta-information about the learners, such as the hyperparameters you can set.

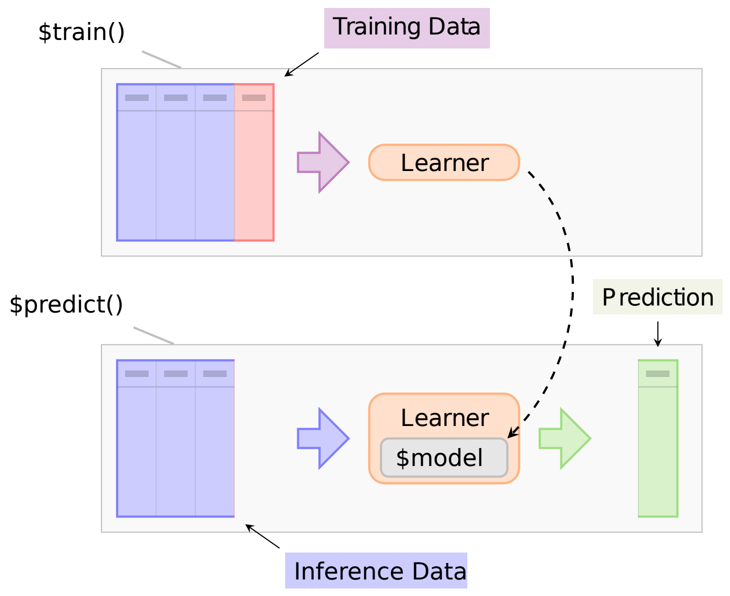
The package ships with a minimal set of classification and regression learners to avoid lots of dependencies:

- `mlr_learners_classif.featureless`: Simple baseline classification learner, constantly predicts the label that is most frequent in the training set.
- `mlr_learners_classif.rpart`: Single classification tree from rpart.
- `mlr_learners_regr.featureless`: Simple baseline regression learner, constantly predicts with the mean.
- `mlr_learners_regr.rpart`: Single regression tree from rpart.

Some of the most popular learners are connected via the mlr3learners package:

- (penalized) linear and logistic regression
- k-Nearest Neighbors regression and classification
- Linear and Quadratic Discriminant Analysis
- Naive Bayes
- Support-Vector machines
- Gradient Boosting
- Random Regression Forests and Random Classification Forests
- Kriging

More learners are collected on GitHub in the mlr3learners organization. Their state is also listed on the wiki of the mlr3learners repository. Below a graphical illustration of the role of a learner:



The base class of each learner is `Learner`, specialized for regression as `LearnerRegr` and for classification as `LearnerClassif`. In contrast to the `Task`, the creation of a custom Learner is usually not required and a more advanced topic. Hence, we refer the reader to Section 6.1 and proceed with an overview of the interface of already implemented learners.

2.3.1 Predefined Learners

Similar to `mlr_tasks`, the `Dictionary` `mlr_learners` can be queried for available learners:

```r
library("mlr3learners")
mlr_learners
```

```
## <DictionaryLearner> with 21 stored values
## Keys: classif.debug, classif.featureless, classif.glmnet, classif.kknn,
##   classif.lda, classif.log_reg, classif.naive_bayes, classif.qda,
##   classif.ranger, classif.rpart, classif.svm, classif.xgboost,
##   regr.featureless, regr.glmnet, regr.kknn, regr.km, regr.lm,
##   regr.ranger, regr.rpart, regr.svm, regr.xgboost
```

Each learner has the following information:

- `feature_types`: the type of features the learner can deal with.
- `packages`: the packages required to train a model with this learner and make predictions.
- `properties`: additional properties and capabilities. For example, a learner has the property "missings" if it is able to handle missing feature values, and "importance" if it computes and allows to extract data on the relative importance of the features. A complete list of these is available in the mlr3 reference on regression learners and classification learners.
- `predict_types`: possible prediction types. For example, a classification learner can predict labels ("response") or probabilities ("prob"). For a complete list of possible predict types see the mlr3 reference.

For a tabular overview of integrated learners, see Section 9.1.

You can get a specific learner using its `id`, listed under `key` in the dictionary:

```r
learner = mlr_learners$get("classif.rpart")
print(learner)
```

```
## <LearnerClassifRpart:classif.rpart>
## * Model: -
## * Parameters: xval=0
## * Packages: rpart
## * Predict Type: response
## * Feature types: logical, integer, numeric, factor, ordered
## * Properties: importance, missings, multiclass, selected_features,
##   twoclass, weights
```

The field `param_set` stores a description of the hyperparameters the learner has, their ranges, defaults, and current values:

```r
learner$param_set
```

```
## ParamSet:
##                 id    class lower upper levels     default value
## 1:        minsplit ParamInt     1   Inf                 20
## 2:       minbucket ParamInt     1   Inf        <NoDefault>
## 3:              cp ParamDbl     0     1               0.01
## 4:      maxcompete ParamInt     0   Inf                  4
## 5:    maxsurrogate ParamInt     0   Inf                  5
## 6:        maxdepth ParamInt     1    30                 30
## 7:    usesurrogate ParamInt     0     2                  2
## 8: surrogatestyle ParamInt     0     1                  0
## 9:            xval ParamInt     0   Inf                 10     0
```

The set of current hyperparameter values is stored in the `values` field of the `param_set` field. You can change the current hyperparameter values by assigning a named list to this field:

```r
learner$param_set$values = list(cp = 0.01, xval = 0)
learner
```

```
## <LearnerClassifRpart:classif.rpart>
## * Model: -
## * Parameters: cp=0.01, xval=0
## * Packages: rpart
## * Predict Type: response
## * Feature types: logical, integer, numeric, factor, ordered
## * Properties: importance, missings, multiclass, selected_features,
##   twoclass, weights
```

Note that this operation just overwrites all previously set parameters. If you just want to add or update hyperparameters, you can use `mlr3misc::insert_named()`:

```r
learner$param_set$values = mlr3misc::insert_named(
  learner$param_set$values,
  list(cp = 0.02, minsplit = 2)
)
learner
```

```
## <LearnerClassifRpart:classif.rpart>
## * Model: -
## * Parameters: cp=0.02, xval=0, minsplit=2
## * Packages: rpart
## * Predict Type: response
## * Feature types: logical, integer, numeric, factor, ordered
## * Properties: importance, missings, multiclass, selected_features,
##   twoclass, weights
```

This updates `cp` to `0.02`, sets `minsplit` to `2` and keeps the previously set parameter `xval`.

Again, there is an alternative to writing down the lengthy `mlr_learners$get()` part: `lrn()`. This function additionally allows to construct learners with specific hyperparameters or settings of a different identifier in one go:

```
lrn("classif.rpart", id = "rp", cp = 0.001)
```

```
## <LearnerClassifRpart:rp>
## * Model: -
## * Parameters: xval=0, cp=0.001
## * Packages: rpart
## * Predict Type: response
## * Feature types: logical, integer, numeric, factor, ordered
## * Properties: importance, missings, multiclass, selected_features,
##   twoclass, weights
```

If you pass hyperparameters here, they are added to the parameters in a `insert_named()`-fashion.

2.4 Train and Predict

In this section, we explain how tasks and learners can be used to train a model and predict to a new dataset. The concept is demonstrated on a supervised classification using the iris dataset and the **rpart** learner, which builds a singe classification tree.

Training a learner means fitting a model to a given data set. Subsequently, we want to predict the label for new observations. These predictions are compared to the ground truth values in order to assess the predictive performance of the model.

2.4.1 Creating Task and Learner Objects

The first step is to generate the following mlr3 objects from the task dictionary and the learner dictionary, respectively:

1. The classification task:

```
task = tsk("sonar")
```

2. A learner for the classification tree:

```
learner = lrn("classif.rpart")
```

2.4.2 Setting up the train/test splits of the data

It is common to train on a majority of the data. Here we use 80% of all available observations and predict on the remaining 20%. For this purpose, we create two index vectors:

```
train_set = sample(task$nrow, 0.8 * task$nrow)
test_set = setdiff(seq_len(task$nrow), train_set)
```

In Section 2.5 we will learn how mlr3 can automatically create training and test sets based on different resampling strategies.

2.4.3 Training the learner

The field `$model` stores the model that is produced in the training step. Before the `$train()` method is called on a learner object, this field is `NULL`:

```
learner$model
```

```
## NULL
```

Next, the classification tree is trained using the train set of the iris task by calling the `$train()` method of the `Learner`:

```
learner$train(task, row_ids = train_set)
```

This operation modifies the learner in-place. We can now access the stored model via the field `$model`:

```
print(learner$model)
```

```
## n= 166
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
##  1) root 166 78 M (0.5301 0.4699)
##    2) V11>=0.1983 92 20 M (0.7826 0.2174)
##      4) V16< 0.6665 69  8 M (0.8841 0.1159) *
##      5) V16>=0.6665 23 11 R (0.4783 0.5217)
##       10) V30< 0.481 13  2 M (0.8462 0.1538) *
##       11) V30>=0.481 10  0 R (0.0000 1.0000) *
##    3) V11< 0.1983 74 16 R (0.2162 0.7838)
##      6) V47>=0.1711 10  2 M (0.8000 0.2000) *
##      7) V47< 0.1711 64  8 R (0.1250 0.8750) *
```

2.4.4 Predicting

After the model has been trained, we use the remaining part of the data for prediction. Remember that we initially split the data in `train_set` and `test_set`.

```
prediction = learner$predict(task, row_ids = test_set)
print(prediction)
```

```
## <PredictionClassif> for 42 observations:
##     row_id truth response
##          4     R        M
##          5     R        M
##          8     R        M
## ---
##        199     M        M
##        204     M        M
##        205     M        M
```

The `$predict()` method of the `Learner` returns a `Prediction` object. More precisely, a `LearnerClassif` returns a `PredictionClassif` object.

A prediction objects holds the row ids of the test data, the respective true label of the target column and the respective predictions. The simplest way to extract this information is by converting the `Prediction` object to a `data.table()`:

```
head(as.data.table(prediction))
```

```
##    row_id truth response
## 1:      4     R        M
## 2:      5     R        M
## 3:      8     R        M
## 4:     12     R        R
## 5:     17     R        M
## 6:     20     R        M
```

For classification, you can also extract the confusion matrix:

```
prediction$confusion
```

```
##         truth
## response  M  R
##        M 19 10
##        R  4  9
```

2.4.5 Changing the Predict Type

Classification learners default to predicting the class label. However, many classifiers additionally also tell you how sure they are about the predicted label by providing posterior probabilities. To switch to predicting these probabilities, the `predict_type` field of a `LearnerClassif` must be changed from `"response"` to `"prob"` before training:

```
learner$predict_type = "prob"

# re-fit the model
learner$train(task, row_ids = train_set)

# rebuild prediction object
prediction = learner$predict(task, row_ids = test_set)
```

The prediction object now contains probabilities for all class labels:

```
# data.table conversion
head(as.data.table(prediction))
```

```
##    row_id truth response prob.M prob.R
## 1:      4     R        M 0.8000 0.2000
## 2:      5     R        M 0.8841 0.1159
## 3:      8     R        M 0.8841 0.1159
## 4:     12     R        R 0.1250 0.8750
## 5:     17     R        M 0.8841 0.1159
## 6:     20     R        M 0.8462 0.1538
```

```
# directly access the predicted labels:
head(prediction$response)
```

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

```
# directly access the matrix of probabilities:
head(prediction$prob)
```

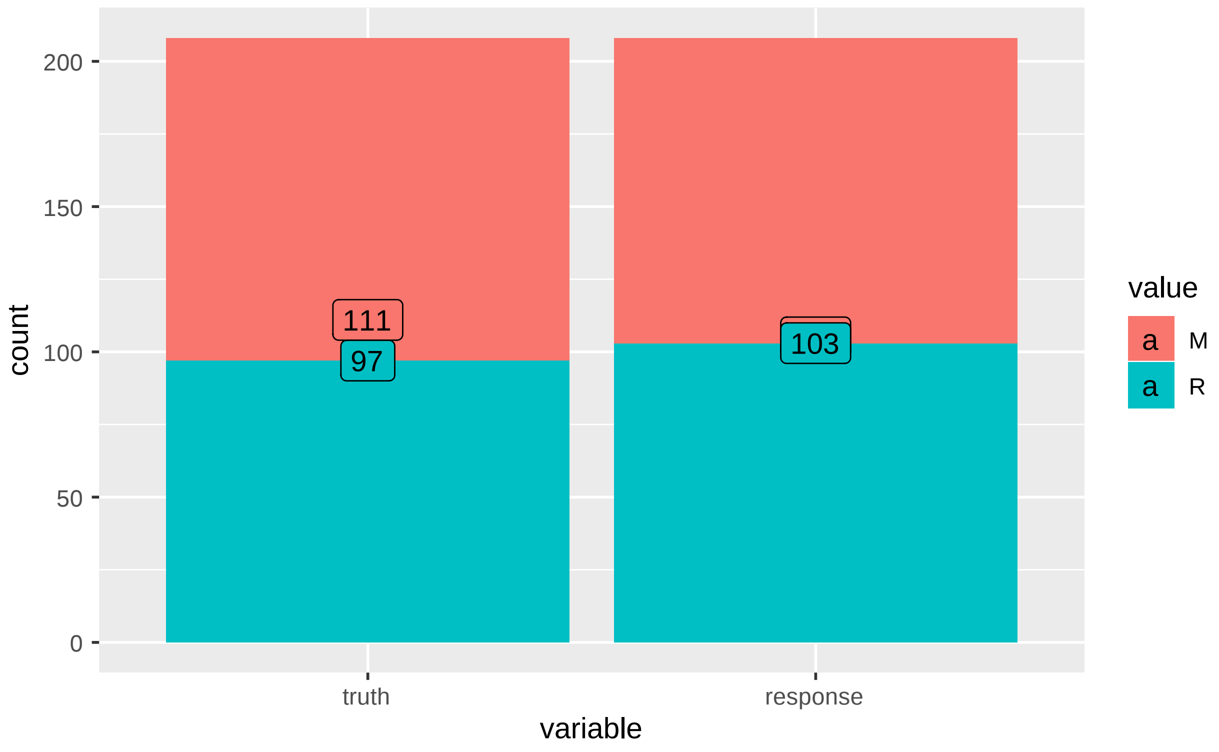
```
##           M      R
## [1,] 0.8000 0.2000
## [2,] 0.8841 0.1159
## [3,] 0.8841 0.1159
## [4,] 0.1250 0.8750
## [5,] 0.8841 0.1159
## [6,] 0.8462 0.1538
```

Analogously to predicting probabilities, many `regression learners` support the extraction of standard error estimates by setting the predict type to `"se"`.

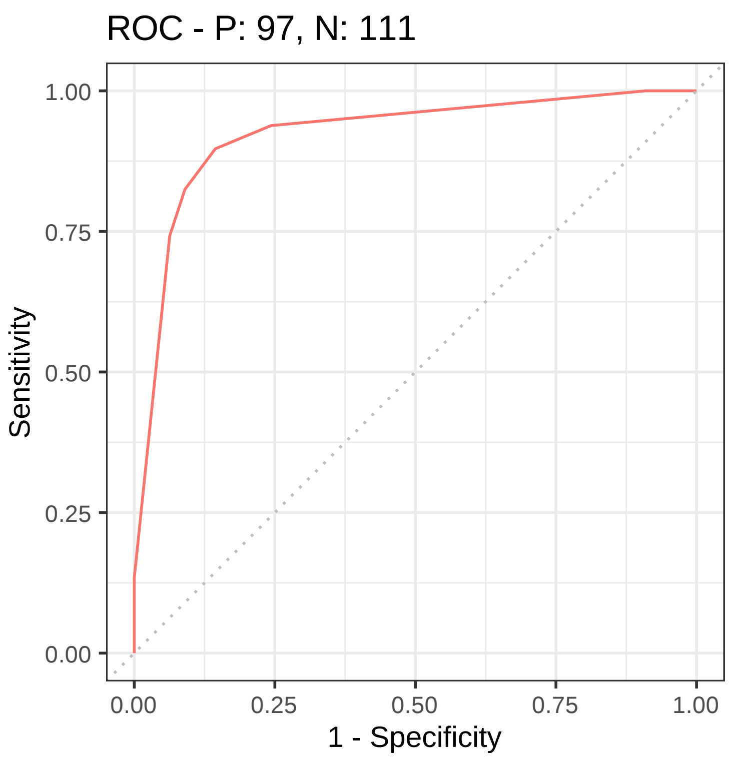
2.4.6 Plotting Predictions

Analogously to plotting tasks, mlr3viz provides a `autoplot()` method for `Prediction` objects. All available types are listed on the manual page of `autoplot.PredictionClassif()` or `autoplot.PredictionClassif()`, respectively.

```
library("mlr3viz")

task = tsk("sonar")
learner = lrn("classif.rpart", predict_type = "prob")
learner$train(task)
prediction = learner$predict(task)
autoplot(prediction)
```

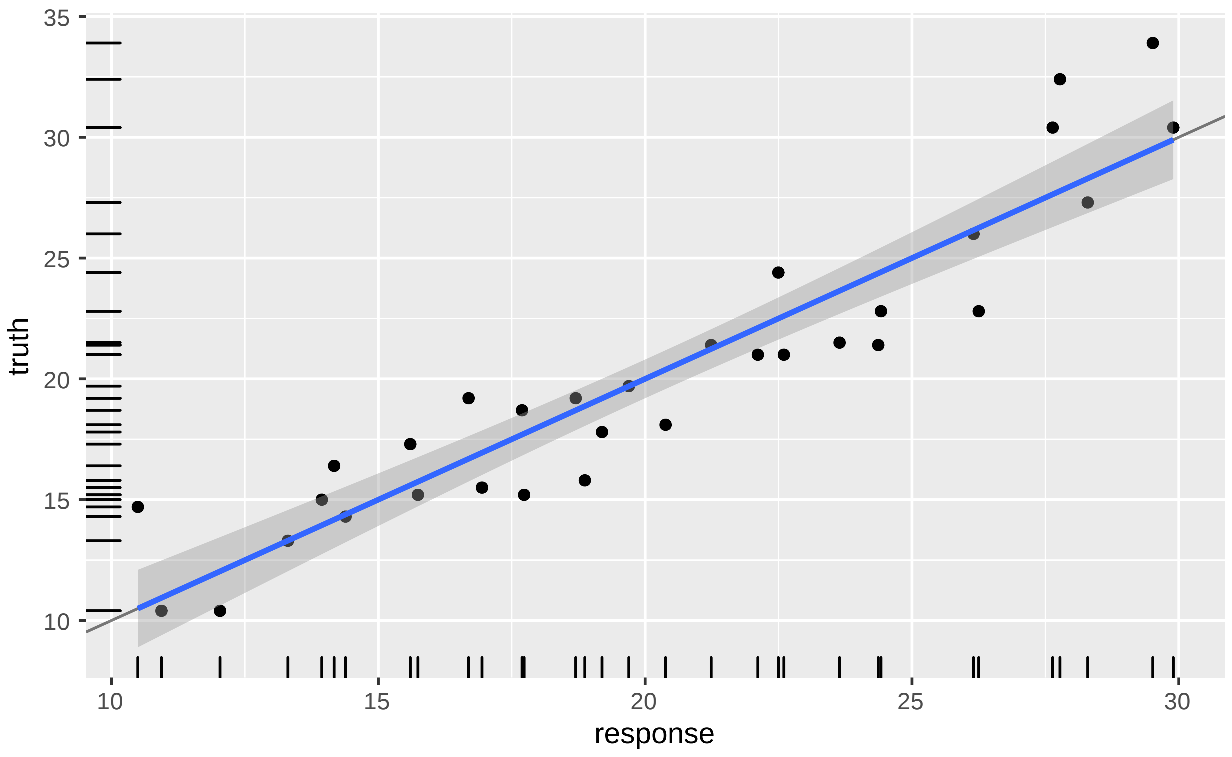


```r
autoplot(prediction, type = "roc")
```



```r
library("mlr3viz")
library("mlr3learners")
```

```r
local({ # we do this locally to not overwrite the objects from previous chunks
  task = tsk("mtcars")
  learner = lrn("regr.lm")
  learner$train(task)
  prediction = learner$predict(task)
  autoplot(prediction)
})
```



2.4.7 Performance assessment

The last step of modeling is usually the performance assessment. To assess the quality of the predictions, the predicted labels are compared with the true labels. How this comparison is calculated is defined by a measure, which is given by a `Measure` object. Note that if the prediction was made on a dataset without the target column, i.e. without true labels, then no performance can be calculated.

Predefined available measures are stored in `mlr_measures` (with convenience getter `msr()`):

```r
mlr_measures
```

```
## <DictionaryMeasure> with 51 stored values
## Keys: classif.acc, classif.auc, classif.bacc, classif.ce,
##   classif.costs, classif.dor, classif.fbeta, classif.fdr, classif.fn,
##   classif.fnr, classif.fomr, classif.fp, classif.fpr, classif.logloss,
##   classif.mcc, classif.npv, classif.ppv, classif.precision,
##   classif.recall, classif.sensitivity, classif.specificity, classif.tn,
##   classif.tnr, classif.tp, classif.tpr, debug, oob_error, regr.bias,
##   regr.ktau, regr.mae, regr.mape, regr.maxae, regr.medae, regr.medse,
##   regr.mse, regr.msle, regr.pbias, regr.rae, regr.rmse, regr.rmsle,
##   regr.rrse, regr.rse, regr.rsq, regr.sae, regr.smape, regr.srho,
##   regr.sse, selected_features, time_both, time_predict, time_train
```

We choose **accuracy** (`classif.acc`) as a specific performance measure and call the method `$score()` of the `Prediction` object to quantify the predictive performance.

```
measure = msr("classif.acc")
prediction$score(measure)
```

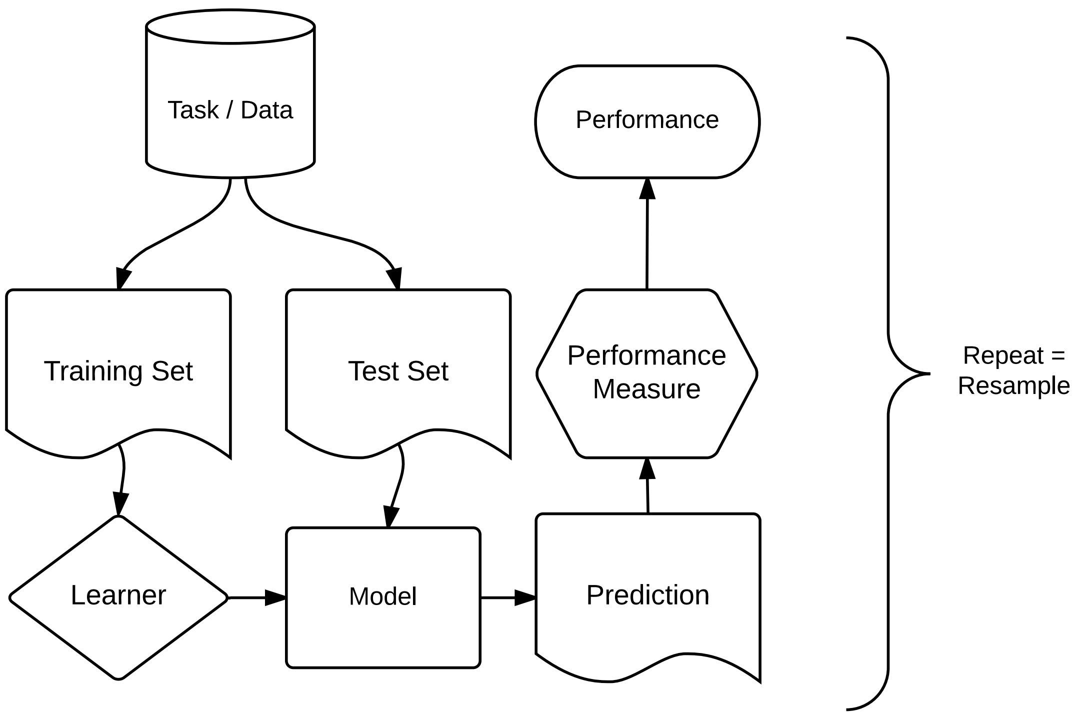
```
## classif.acc
##       0.875
```

Note that, if no measure is specified, classification defaults to classification error (`classif.ce`) and regression defaults to the mean squared error (`regr.mse`).

2.5 Resampling

Resampling strategies are usually used to assess the performance of a learning algorithm. `mlr3` entails 6 predefined resampling strategies: Cross-validation, Leave-one-out cross validation, Repeated cross-validation, Out-of-bag bootstrap and other variants (e.g. b632), Monte-Carlo cross-validation and Holdout. The following sections provide guidance on how to set and select a resampling strategy and how to subsequently instantiate the resampling process.

Below you can find a graphical illustration of the resampling process:



2.5.1 Settings

In this example we use the **iris** task and a simple classification tree from the rpart package.

```
task = tsk("iris")
learner = lrn("classif.rpart")
```

When performing resampling with a dataset, we first need to define which approach should be used. mlr3 resampling strategies and their parameters can be queried by looking at the data.table output of the `mlr_resamplings` dictionary:

```
as.data.table(mlr_resamplings)
```

```
##            key        params iters
## 1:   bootstrap repeats,ratio    30
## 2:      custom                   0
## 3:          cv         folds    10
## 4:     holdout         ratio     1
## 5: repeated_cv repeats,folds   100
## 6: subsampling repeats,ratio    30
```

Additional resampling methods for special use cases will be available via extension packages, such as mlr3spatiotemporal for spatial data (still in development).

The model fit conducted in the train/predict/score chapter is equivalent to a "holdout resampling", so let's consider this one first. Again, we can retrieve elements from the dictionary `mlr_resamplings` via `$get()` or with the convenience function`rsmp()`:

```
resampling = rsmp("holdout")
print(resampling)
```

```
## <ResamplingHoldout> with 1 iterations
## * Instantiated: FALSE
## * Parameters: ratio=0.6667
```

Note that the `$is_instantiated` field is set to `FALSE`. This means we did not actually apply the strategy on a dataset yet. Applying the strategy on a dataset is done in the next section Instantiation.

By default we get a .66/.33 split of the data. There are two ways in which the ratio can be changed:

1. Overwriting the slot in `$param_set$values` using a named list:

```
resampling$param_set$values = list(ratio = 0.8)
```

2. Specifying the resampling parameters directly during construction:

```
rsmp("holdout", ratio = 0.8)
```

```
## <ResamplingHoldout> with 1 iterations
## * Instantiated: FALSE
## * Parameters: ratio=0.8
```

2.5.2 Instantiation

So far we just set the stage and selected the resampling strategy.

To actually perform the splitting and obtain indices for the training and the test split the resampling needs a `Task`. By calling the method `instantiate()`, we split the indices of the data into indices for training and test sets. These resulting indices are stored in the `Resampling` object:

```
resampling = rsmp("cv", folds = 3L)
resampling$instantiate(task)
resampling$iters
```

```
## [1] 3
```

```
str(resampling$train_set(1))
```

```
##  int [1:100] 2 5 7 10 13 14 16 17 18 19 ...
```

```
str(resampling$test_set(1))
```

```
##  int [1:50] 1 3 8 9 12 21 22 24 31 32 ...
```

2.5.3 Execution

With a `Task`, a `Learner` and a `Resampling` object we can call `resample()`, which fits the learner to the task at hand according to the given resampling strategy. This in turn creates a `ResampleResult` object.

Before we go into more detail, let’s change the resampling to a “3-fold cross-validation” to better illustrate what operations are possible with a `ResampleResult`. Additionally, when actually fitting the models, we tell `resample()` to keep the fitted models by setting the `store_models` option to `TRUE`:

```
task = tsk("pima")
learner = lrn("classif.rpart", maxdepth = 3, predict_type = "prob")
resampling = rsmp("cv", folds = 3L)

rr = resample(task, learner, resampling, store_models = TRUE)
print(rr)
```

```
## <ResampleResult> of 3 iterations
## * Task: pima
## * Learner: classif.rpart
## * Warnings: 0 in 0 iterations
## * Errors: 0 in 0 iterations
```

The following operations are supported with `ResampleResult` objects:

Calculate the average performance across all resampling iterations:

```
rr$aggregate(msr("classif.ce"))
```

```
## classif.ce
##     0.2839
```

Extract the performance for the individual resampling iterations:

```
rr$score(msr("classif.ce"))
```

```
##               task task_id                 learner    learner_id     resampling
## 1: <TaskClassif>    pima <LearnerClassifRpart> classif.rpart <ResamplingCV>
## 2: <TaskClassif>    pima <LearnerClassifRpart> classif.rpart <ResamplingCV>
## 3: <TaskClassif>    pima <LearnerClassifRpart> classif.rpart <ResamplingCV>
##    resampling_id iteration prediction classif.ce
## 1:            cv         1     <list>     0.2773
## 2:            cv         2     <list>     0.3164
## 3:            cv         3     <list>     0.2578
```

Check for warnings or errors:

```
rr$warnings
```

```
## Empty data.table (0 rows and 2 cols): iteration,msg
```

```
rr$errors
```

```
## Empty data.table (0 rows and 2 cols): iteration,msg
```

Extract and inspect the resampling splits:

```
rr$resampling
```

```
## <ResamplingCV> with 3 iterations
## * Instantiated: TRUE
## * Parameters: folds=3
```

```
rr$resampling$iters
```

```
## [1] 3
```

```
str(rr$resampling$test_set(1))
```

```
##  int [1:256] 2 4 6 7 11 14 21 22 23 25 ...
```

```
str(rr$resampling$train_set(1))
```

```
##  int [1:512] 1 8 9 13 17 18 26 30 31 34 ...
```

Retrieve the learner of a specific iteration and inspect it:

```
lrn = rr$learners[[1]]
lrn$model
```

```
## n= 512
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
##  1) root 512 186 neg (0.63672 0.36328)
##    2) glucose< 111.5 213  30 neg (0.85915 0.14085)
##      4) pregnant< 6.5 174  14 neg (0.91954 0.08046) *
##      5) pregnant>=6.5 39  16 neg (0.58974 0.41026)
##       10) pedigree< 0.6175 29   8 neg (0.72414 0.27586) *
##       11) pedigree>=0.6175 10   2 pos (0.20000 0.80000) *
##    3) glucose>=111.5 299 143 pos (0.47826 0.52174)
##      6) mass< 29.95 91  24 neg (0.73626 0.26374) *
##      7) mass>=29.95 208  76 pos (0.36538 0.63462) *
```

Extract the predictions:

```r
rr$prediction() # all predictions merged into a single Prediction
```

```
## <PredictionClassif> for 768 observations:
##     row_id truth response prob.pos prob.neg
##          2   neg      neg  0.08046   0.9195
##          4   neg      neg  0.08046   0.9195
##          6   neg      neg  0.26374   0.7363
## ---
##        764   neg      neg  0.18018   0.8198
##        767   pos      neg  0.18018   0.8198
##        768   neg      neg  0.18018   0.8198
```

```r
rr$predictions()[[1]] # prediction of first resampling iteration
```

```
## <PredictionClassif> for 256 observations:
##     row_id truth response prob.pos prob.neg
##          2   neg      neg  0.08046   0.9195
##          4   neg      neg  0.08046   0.9195
##          6   neg      neg  0.26374   0.7363
## ---
##        762   pos      pos  0.63462   0.3654
##        763   neg      neg  0.27586   0.7241
##        765   neg      pos  0.63462   0.3654
```

Note that if you want to compare multiple Learners in a fair manner, it is important to ensure that each learner operates on the same resampling instance. This can be achieved by manually instantiating the instance before fitting model(s) on it.

Hint: If your aim is to compare different `Task`, `Learner` or `Resampling`, you are better off using the `benchmark()` function which is covered in the next section on benchmarking. It is a wrapper around `resample()`, simplifying the handling of large comparison grids.

If you discover this only after you've run multiple `resample()` calls, don't worry. You can combine multiple `ResampleResult` objects into a `BenchmarkResult` (also explained in the section benchmarking).

2.5.4 Custom resampling

Sometimes it is necessary to perform resampling with custom splits. If you want to do that because you are coming from a specific modeling field, first take a look at the mlr3 extension packages, to check wheter your resampling method has been implemented already. If this is not the case, feel welcome to extend an existing package or create your own extension package.

A manual resampling instance can be created using the `"custom"` template.

```r
resampling = rsmp("custom")
resampling$instantiate(task,
  train = list(c(1:10, 51:60, 101:110)),
  test = list(c(11:20, 61:70, 111:120))
)
resampling$iters
```

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

```
resampling$train_set(1)
```

```
##  [1]   1   2   3   4   5   6   7   8   9  10  51  52  53  54  55  56  57  58  59
## [20]  60 101 102 103 104 105 106 107 108 109 110
```

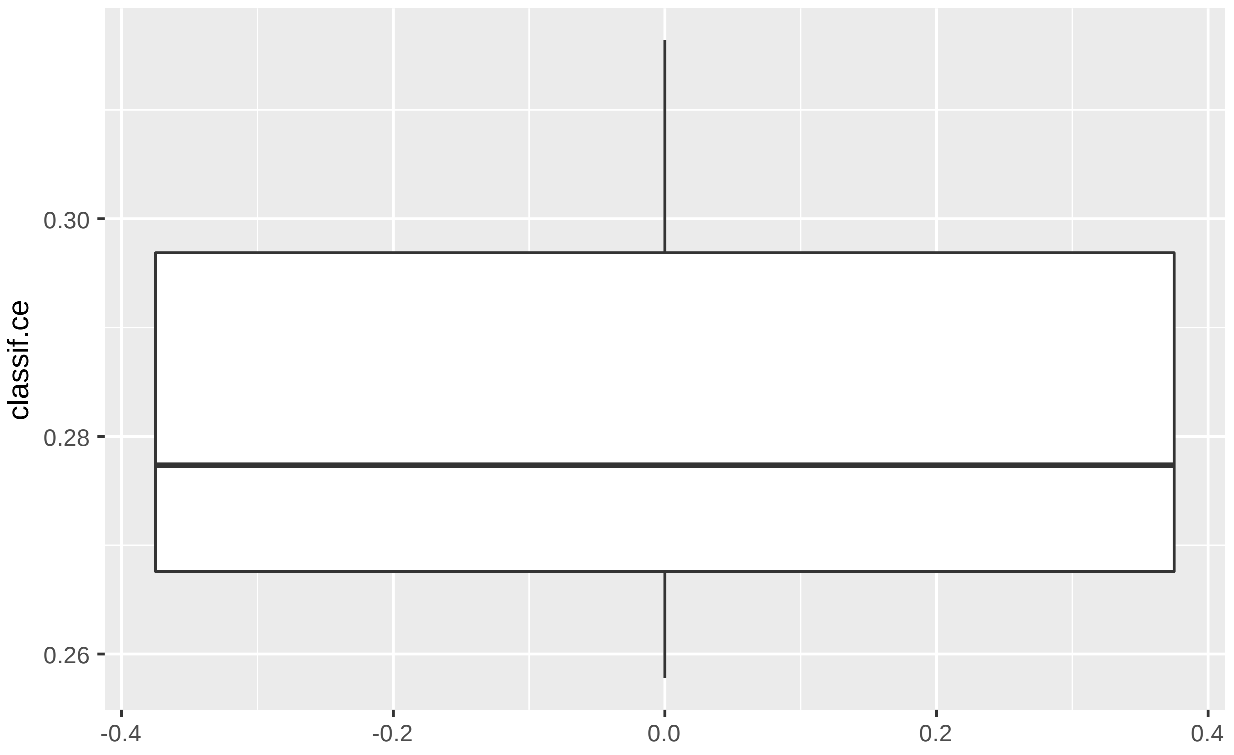
```
resampling$test_set(1)
```

```
##  [1]  11  12  13  14  15  16  17  18  19  20  61  62  63  64  65  66  67  68  69
## [20]  70 111 112 113 114 115 116 117 118 119 120
```

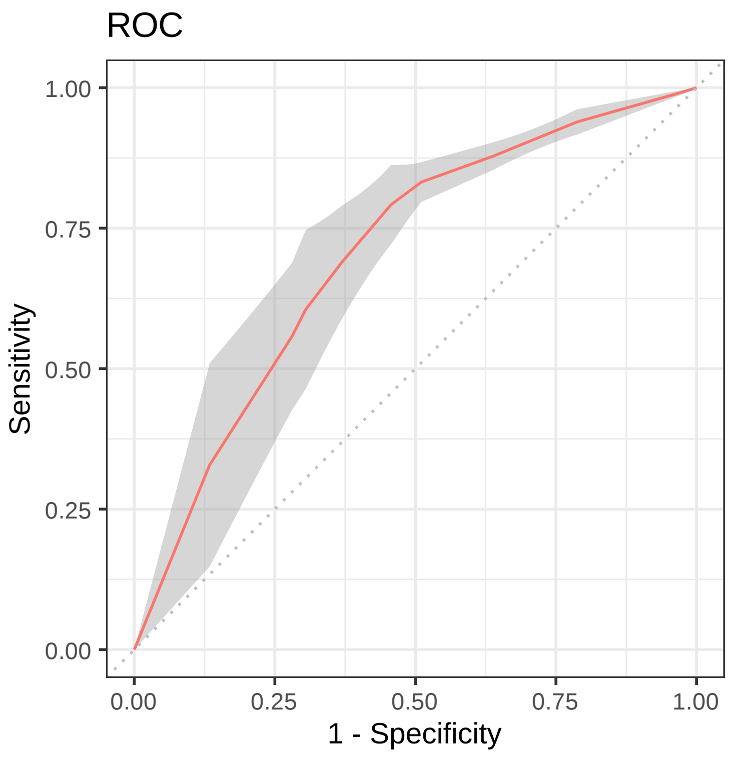
2.5.5 Plotting Resample Results

Again, mlr3viz provides a `autoplot()` method.

```
library("mlr3viz")

autoplot(rr)
```



```
autoplot(rr, type = "roc")
```



All available plot types are listed on the manual page of `autoplot.ResampleResult()`.

2.6 Benchmarking

Comparing the performance of different learners on multiple tasks and/or different resampling schemes is a common task. This operation is usually referred to as "benchmarking" in the field of machine-learning. The mlr3 package offers the `benchmark()` function for convenience.

2.6.1 Design Creation

In mlr3 we require you to supply a "design" of your benchmark experiment. By "design" we essentially mean the matrix of settings you want to execute. A "design" consists of unique combinations of `Task`, `Learner` and `Resampling` triplets.

Here, we call `benchmark()` to perform a single holdout split on a single task and two learners. We use the `benchmark_grid()` function to create an exhaustive design and instantiate the resampling properly, so that all learners are executed on the same train/test split for each task:

```
library("data.table")
design = benchmark_grid(
  tasks = tsk("iris"),
  learners = list(lrn("classif.rpart"), lrn("classif.featureless")),
  resamplings = rsmp("holdout")
)
print(design)
```

```
##              task                       learner          resampling
## 1: <TaskClassif>       <LearnerClassifRpart> <ResamplingHoldout>
## 2: <TaskClassif> <LearnerClassifFeatureless> <ResamplingHoldout>
```

```r
bmr = benchmark(design)
```

Instead of using benchmark_grid() you could also create the design manually as a data.table and use the full flexibility of the benchmark() function. The design does not have to be exhaustive, e.g. it can also contain a different learner for each task. However, you should note that benchmark_grid() makes sure to instantiate the resamplings for each task. If you create the design manually, even if the same task is used multiple times, the train/test splits will be different for each row of the design if you do not **manually instantiate** the resampling before creating the design.

Let's construct a more complex design to show the full capabilities of the benchmark() function.

```r
# get some example tasks
tasks = lapply(c("german_credit", "sonar"), tsk)

# get some learners and for all learners ...
# * predict probabilities
# * predict also on the training set
library("mlr3learners")
learners = c("classif.featureless", "classif.rpart", "classif.ranger", "classif.kknn")
learners = lapply(learners, lrn,
  predict_type = "prob", predict_sets = c("train", "test"))

# compare via 3-fold cross validation
resamplings = rsmp("cv", folds = 3)

# create a BenchmarkDesign object
design = benchmark_grid(tasks, learners, resamplings)
print(design)
```

```
##             task                    learner    resampling
## 1: <TaskClassif> <LearnerClassifFeatureless> <ResamplingCV>
## 2: <TaskClassif>       <LearnerClassifRpart> <ResamplingCV>
## 3: <TaskClassif>      <LearnerClassifRanger> <ResamplingCV>
## 4: <TaskClassif>        <LearnerClassifKKNN> <ResamplingCV>
## 5: <TaskClassif> <LearnerClassifFeatureless> <ResamplingCV>
## 6: <TaskClassif>       <LearnerClassifRpart> <ResamplingCV>
## 7: <TaskClassif>      <LearnerClassifRanger> <ResamplingCV>
## 8: <TaskClassif>        <LearnerClassifKKNN> <ResamplingCV>
```

2.6.2 Execution and Aggregation of Results

After the benchmark design is ready, we can directly call benchmark():

```r
# execute the benchmark
bmr = benchmark(design)
```

Note that we did not instantiate the resampling instance manually. benchmark_grid() took care of it for us: Each resampling strategy is instantiated once for each task during the construction of the exhaustive grid.

After the benchmark, we can calculate and aggregate the performance with $aggregate():

```
# measures:
# * area under the curve (auc) on training
# * area under the curve (auc) on test
measures = list(
  msr("classif.auc", id = "auc_train", predict_sets = "train"),
  msr("classif.auc", id = "auc_test")
)
bmr$aggregate(measures)
```

```
##    nr  resample_result       task_id          learner_id resampling_id iters
## 1:  1 <ResampleResult> german_credit classif.featureless            cv     3
## 2:  2 <ResampleResult> german_credit       classif.rpart            cv     3
## 3:  3 <ResampleResult> german_credit      classif.ranger            cv     3
## 4:  4 <ResampleResult> german_credit        classif.kknn            cv     3
## 5:  5 <ResampleResult>         sonar classif.featureless            cv     3
## 6:  6 <ResampleResult>         sonar       classif.rpart            cv     3
## 7:  7 <ResampleResult>         sonar      classif.ranger            cv     3
## 8:  8 <ResampleResult>         sonar        classif.kknn            cv     3
##    auc_train auc_test
## 1:    0.5000   0.5000
## 2:    0.8031   0.7192
## 3:    0.9984   0.7903
## 4:    0.9881   0.6960
## 5:    0.5000   0.5000
## 6:    0.9264   0.7534
## 7:    1.0000   0.9109
## 8:    0.9985   0.9053
```

Subsequently, we can aggregate the results further. For example, we might be interested which learner performed best over all tasks simultaneously. Simply aggregating the performances with the mean is usually not statistically sound. Instead, we calculate the rank statistic for each learner grouped by task. Then the calculated ranks grouped by learner are aggregated with data.table. Since the AUC needs to be maximized, we multiply with -1 so that the best learner gets a rank of 1.

```
tab = bmr$aggregate(measures)
print(tab)
```

```
##    nr  resample_result       task_id          learner_id resampling_id iters
## 1:  1 <ResampleResult> german_credit classif.featureless            cv     3
## 2:  2 <ResampleResult> german_credit       classif.rpart            cv     3
## 3:  3 <ResampleResult> german_credit      classif.ranger            cv     3
## 4:  4 <ResampleResult> german_credit        classif.kknn            cv     3
## 5:  5 <ResampleResult>         sonar classif.featureless            cv     3
## 6:  6 <ResampleResult>         sonar       classif.rpart            cv     3
## 7:  7 <ResampleResult>         sonar      classif.ranger            cv     3
## 8:  8 <ResampleResult>         sonar        classif.kknn            cv     3
##    auc_train auc_test
## 1:    0.5000   0.5000
## 2:    0.8031   0.7192
## 3:    0.9984   0.7903
## 4:    0.9881   0.6960
## 5:    0.5000   0.5000
## 6:    0.9264   0.7534
## 7:    1.0000   0.9109
## 8:    0.9985   0.9053
```

```
# group by levels of task_id, return columns:
# - learner_id
# - rank of col '-auc_train' (per level of learner_id)
# - rank of col '-auc_test' (per level of learner_id)
ranks = tab[, .(learner_id, rank_train = rank(-auc_train), rank_test = rank(-auc_test)), by = task_id]
print(ranks)
```

```
##          task_id          learner_id rank_train rank_test
## 1: german_credit classif.featureless          4         4
## 2: german_credit       classif.rpart          3         2
## 3: german_credit      classif.ranger          1         1
## 4: german_credit        classif.kknn          2         3
## 5:         sonar classif.featureless          4         4
## 6:         sonar       classif.rpart          3         3
## 7:         sonar      classif.ranger          1         1
## 8:         sonar        classif.kknn          2         2
```

```
# group by levels of learner_id, return columns:
# - mean rank of col 'rank_train' (per level of learner_id)
# - mean rank of col 'rank_test' (per level of learner_id)
ranks = ranks[, .(mrank_train = mean(rank_train), mrank_test = mean(rank_test)), by = learner_id]

# print the final table, ordered by mean rank of AUC test
ranks[order(mrank_test)]
```

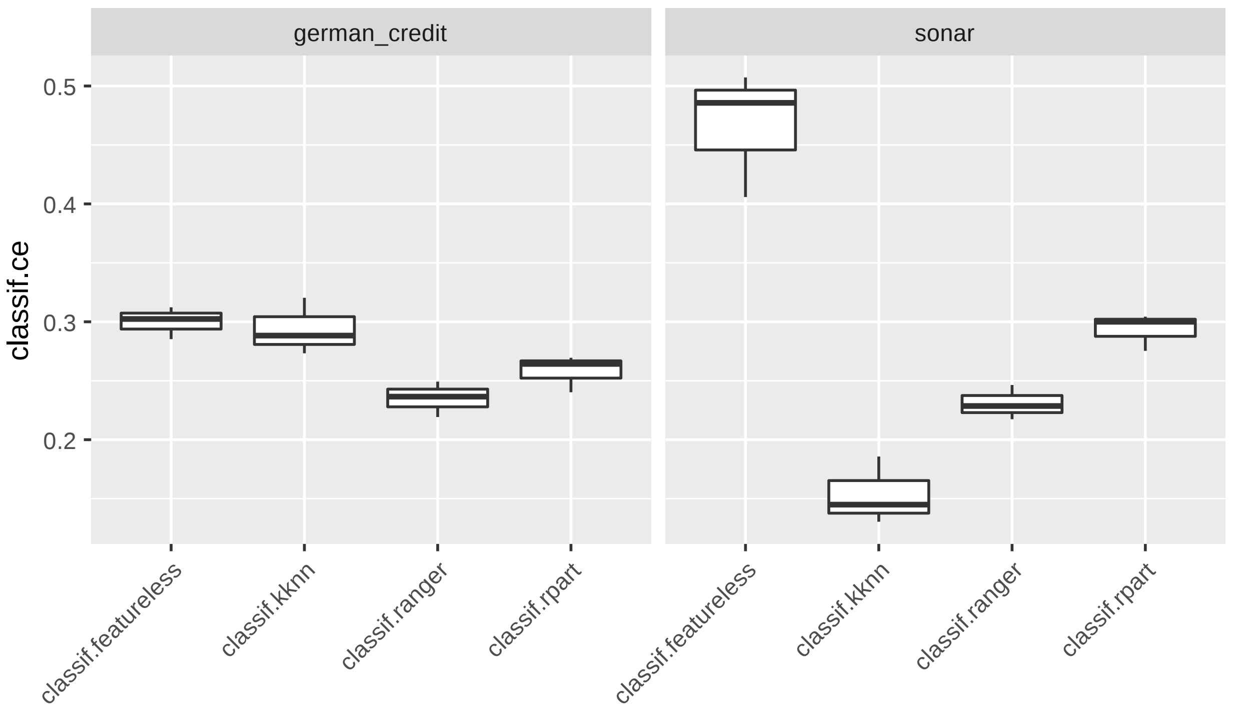
```
##             learner_id mrank_train mrank_test
## 1:      classif.ranger           1        1.0
## 2:       classif.rpart           3        2.5
## 3:        classif.kknn           2        2.5
## 4: classif.featureless           4        4.0
```

Unsurprisingly, the featureless learner is outperformed on both training and test set.

2.6.3 Plotting Benchmark Results

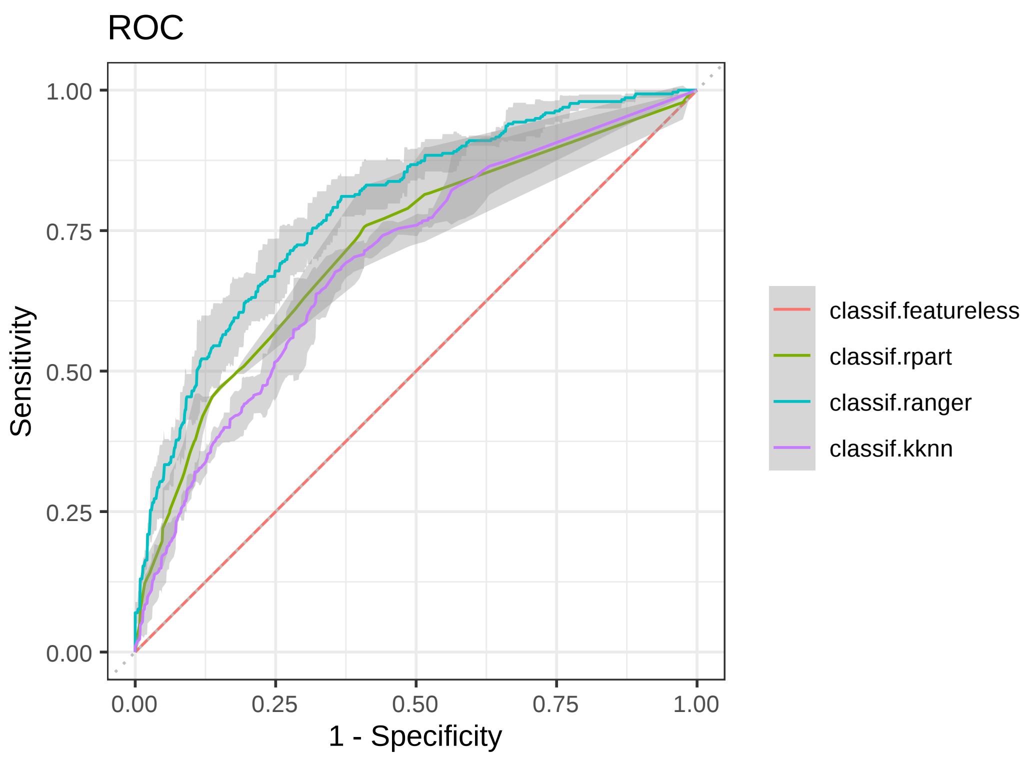
Analogously to plotting tasks, predictions or resample results, mlr3viz also provides a autoplot() method for benchmark results.

```
library("mlr3viz")
library("ggplot2")

autoplot(bmr) + theme(axis.text.x = element_text(angle = 45, hjust = 1))
```



We can also plot ROC curves. To do so, we first need to filter the `BenchmarkResult` to only contain a single `Task`:

```
autoplot(bmr$clone()$filter(task_id = "german_credit"), type = "roc")
```



All available types are listed on the manual page of `autoplot.BenchmarkResult()`.

2.6.4 Extracting ResampleResults

A `BenchmarkResult` object is essentially a collection of multiple `ResampleResult` objects. As these are stored in a column of the aggregated `data.table()`, we can easily extract them:

```
tab = bmr$aggregate(measures)
rr = tab[task_id == "sonar" & learner_id == "classif.ranger"]$resample_result[[1]]
print(rr)
```

```
## <ResampleResult> of 3 iterations
## * Task: sonar
## * Learner: classif.ranger
## * Warnings: 0 in 0 iterations
## * Errors: 0 in 0 iterations
```

We can now investigate this resampling and even single resampling iterations using one of the approaches shown in the previous section:

```
measure = msr("classif.auc")
rr$aggregate(measure)
```

```
## classif.auc
##      0.9109
```

```
# get the iteration with worst AUC
perf = rr$score(measure)
i = which.min(perf$classif.auc)

# get the corresponding learner and train set
print(rr$learners[[i]])
```

```
## <LearnerClassifRanger:classif.ranger>
## * Model: -
## * Parameters: list()
## * Packages: ranger
## * Predict Type: prob
## * Feature types: logical, integer, numeric, character, factor, ordered
## * Properties: importance, multiclass, oob_error, twoclass, weights
```

```
head(rr$resampling$train_set(i))
```

```
## [1]  1  3  4  8  9 19
```

2.6.5 Converting and Merging ResampleResults

It is also possible to cast a single `ResampleResult` to a `BenchmarkResult` using the converter `as_benchmark_result()`.

```
task = tsk("iris")
resampling = rsmp("holdout")$instantiate(task)

rr1 = resample(task, lrn("classif.rpart"), resampling)
```

```
rr2 = resample(task, lrn("classif.featureless"), resampling)

# Cast both ResampleResults to BenchmarkResults
bmr1 = as_benchmark_result(rr1)
bmr2 = as_benchmark_result(rr2)

# Merge 2nd BMR into the first BMR
bmr1$combine(bmr2)

bmr1
```

```
## <BenchmarkResult> of 2 rows with 2 resampling runs
##  nr task_id          learner_id resampling_id iters warnings errors
##   1    iris        classif.rpart       holdout     1        0      0
##   2    iris classif.featureless       holdout     1        0      0
```

2.7 Binary classification

Classification problems with a target variable containing only two classes are called "binary". For such binary target variables, you can specify the *positive class* within the classification task object during task creation. If not explicitly set during construction, the positive class defaults to the first level of the target variable.

```
# during construction
data("Sonar", package = "mlbench")
task = TaskClassif$new(id = "Sonar", Sonar, target = "Class", positive = "R")

# switch positive class to level 'M'
task$positive = "M"
```

2.7.1 ROC Curve and Thresholds

ROC Analysis, which stands for "receiver operating characteristics", is a subfield of machine learning which studies the evaluation of binary prediction systems. We saw earlier that one can retrieve the confusion matrix of a Prediction by accessing the `$confusion` field:

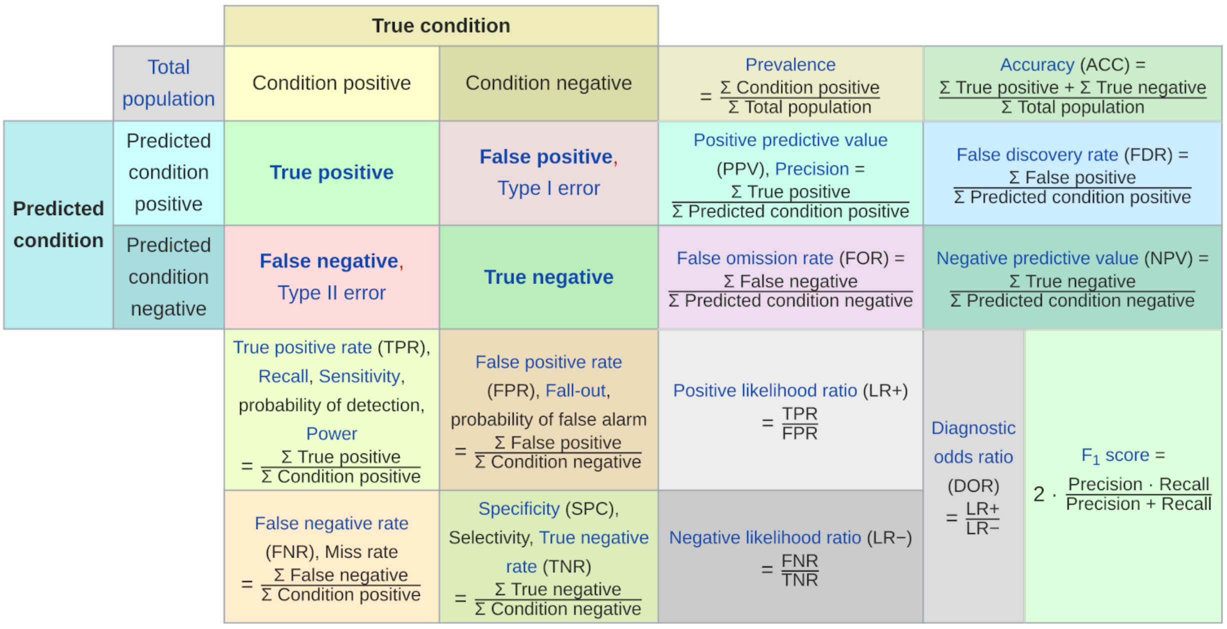
```
learner = lrn("classif.rpart", predict_type = "prob")
pred = learner$train(task)$predict(task)
C = pred$confusion
print(C)
```

```
##         truth
## response  M  R
##        M 95 10
##        R 16 87
```

The confusion matrix contains the counts of correct and incorrect class assignments, grouped by class labels. The columns illustrate the true (observed) labels and the rows display the predicted labels. The positive is always the first row or column in the confusion matrix. Thus, the element in C_{11} is the number of times our model predicted the positive class and was right about it. Analogously, the element in C_{22} is the number of times our model predicted the negative class and was also right about it. The elements on the diagonal are called True Positives (TP) and True Negatives (TN). The element C_{12} is the number of times we falsely predicted a positive label, and is called False Positives (FP). The element C_{21} is called False Negatives (FN).

We can now normalize in rows and columns of the confusion matrix to derive several informative metrics:

- **True Positive Rate (TPR)**: How many of the true positives did we predict as positive?
- **True Negative Rate (TNR)**: How many of the true negatives did we predict as negative?
- **Positive Predictive Value PPV**: If we predict positive how likely is it a true positive?
- **Negative Predictive Value NPV**: If we predict negative how likely is it a true negative?



		True condition				
	Total population	Condition positive	Condition negative	Prevalence $= \frac{\Sigma \text{ Condition positive}}{\Sigma \text{ Total population}}$	Accuracy (ACC) = $\frac{\Sigma \text{ True positive} + \Sigma \text{ True negative}}{\Sigma \text{ Total population}}$	
Predicted condition	Predicted condition positive	**True positive**	**False positive**, Type I error	Positive predictive value (PPV), Precision = $\frac{\Sigma \text{ True positive}}{\Sigma \text{ Predicted condition positive}}$	False discovery rate (FDR) = $\frac{\Sigma \text{ False positive}}{\Sigma \text{ Predicted condition positive}}$	
	Predicted condition negative	**False negative**, Type II error	**True negative**	False omission rate (FOR) = $\frac{\Sigma \text{ False negative}}{\Sigma \text{ Predicted condition negative}}$	Negative predictive value (NPV) = $\frac{\Sigma \text{ True negative}}{\Sigma \text{ Predicted condition negative}}$	
		True positive rate (TPR), Recall, Sensitivity, probability of detection, Power $= \frac{\Sigma \text{ True positive}}{\Sigma \text{ Condition positive}}$	False positive rate (FPR), Fall-out, probability of false alarm $= \frac{\Sigma \text{ False positive}}{\Sigma \text{ Condition negative}}$	Positive likelihood ratio (LR+) $= \frac{\text{TPR}}{\text{FPR}}$	Diagnostic odds ratio (DOR) $= \frac{\text{LR+}}{\text{LR−}}$	F_1 score = $2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$
		False negative rate (FNR), Miss rate $= \frac{\Sigma \text{ False negative}}{\Sigma \text{ Condition positive}}$	Specificity (SPC), Selectivity, True negative rate (TNR) $= \frac{\Sigma \text{ True negative}}{\Sigma \text{ Condition negative}}$	Negative likelihood ratio (LR−) $= \frac{\text{FNR}}{\text{TNR}}$		

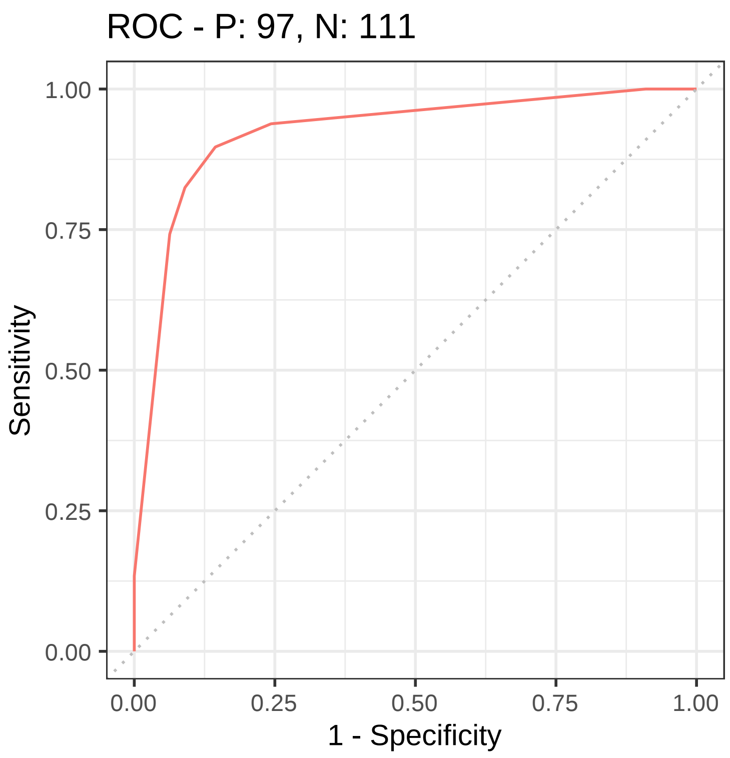
Source: Wikipedia

It is difficult to achieve a high TPR and low FPR in conjunction, so one uses them for constructing the ROC Curve. We characterize a classifier by its TPR and FPR values and plot them in a coordinate system. The best classifier lies on the top-left corner. The worst classifier lies at the diagonal. Classifiers lying on the diagonal produce random labels (with different proportions). If each positive x will be randomly classified with 25% as "positive", we get a TPR of 0.25. If we assign each negative x randomly to "positive" we get a FPR of 0.25. In practice, we should never obtain a classifier below the diagonal, as inverting the predicted labels will result in a reflection at the diagonal.

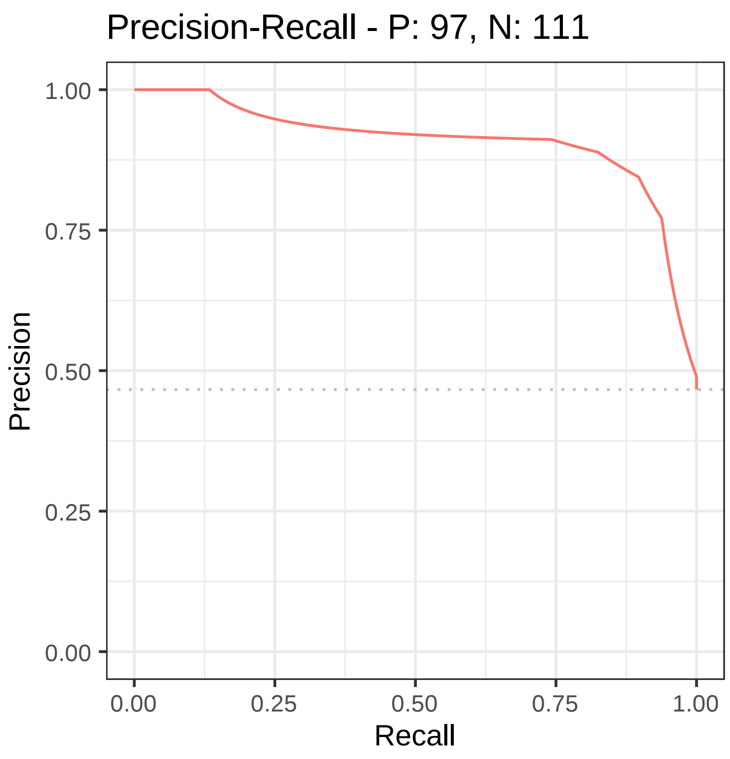
A scoring classifier is a model which produces scores or probabilities, instead of discrete labels. To obtain probabilities from a learner in mlr3, you have to set `predict_type = "prob"` for a `ref("LearnerClassif")`. Whether a classifier can predict probabilities is given in its `$predict_types` field. Thresholding flexibly converts measured probabilities to labels. Predict 1 (positive class) if $\hat{f}(x) > \tau$ else predict 0. Normally, one could use $\tau = 0.5$ to convert probabilities to labels, but for imbalanced or cost-sensitive situations another threshold could be more suitable. After thresholding, any metric defined on labels can be used.

For `mlr3` prediction objects, the ROC curve can easily be created with mlr3viz which relies on the precrec to calculate and plot ROC curves:

```r
library("mlr3viz")

# TPR vs FPR / Sensitivity vs (1 - Specificity)
ggplot2::autoplot(pred, type = "roc")
```



```r
# Precision vs Recall
ggplot2::autoplot(pred, type = "prc")
```



2.7.2 Threshold Tuning

3 Model Optimization

Model Tuning

Machine learning algorithms have default values set for their hyperparameters. Irrespective, these hyperparameters need to be changed by the user to achieve optimal performance on the given dataset. A manual selection of hyperparameter values is not recommended as this approach rarely leads to an optimal performance. To substantiate the validity of the selected hyperparameters (= tuning), data-driven optimization is recommended. In order to tune a machine learning algorithm, one has to specify (1) the search space, (2) the optimization algorithm (aka tuning method) and (3) an evaluation method, i.e., a resampling strategy and (4) a performance measure.

In summary, the sub-chapter on tuning illustrates how to:

- undertake empirically sound hyperparameter selection
- select the optimizing algorithm
- trigger the tuning
- automate tuning

This sub-chapter also requires the package mlr3tuning, an extension package which supports hyperparameter tuning.

Feature Selection

The second part of this chapter explains feature selection also known as variable selection. Feature selection is the process of finding a subset of relevant features of the available data. The purpose can be manifold:

- Enhance the interpretability of the model,
- speed up model fitting or
- improve the learner performance by reducing noise in the data.

In this book we focus mainly on the last aspect. Different approaches exist to identify the relevant features. In this sub-chapter on feature selection, three approaches are emphasized:

- Filter algorithms select features independently of the learner according to a score.
- Variable importance filters select features that are important according to a learner.
- Wrapper methods iteratively select features to optimize a performance measure.

Note, that filters do not require a learner. *Variable importance filters* require a learner that can calculate feature importance values once it is trained. The obtained importance values can be used to subset the data, which then can be used to train any learner. *Wrapper methods* can be used with any learner but need to train the learner multiple times.

Nested Resampling

In order to get a good estimate of generalization performance and avoid data leakage, both an outer (performance) and an inner (tuning/feature selection) resampling process are necessary. Following features are discussed in this chapter:

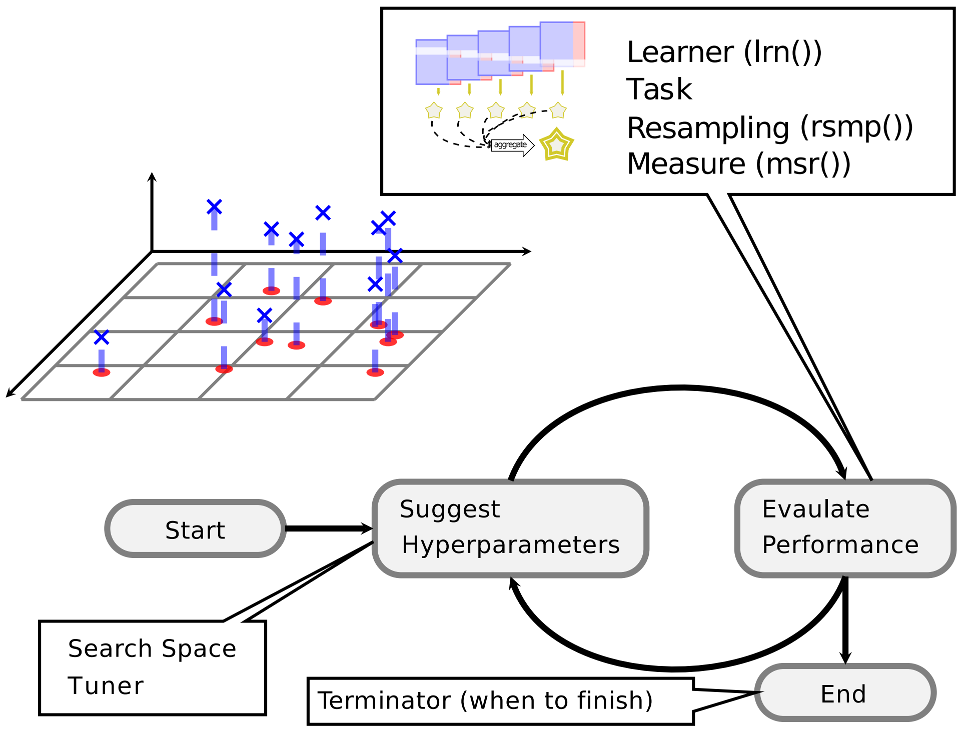
- Inner and outer resampling strategies in nested resampling
- The execution of nested resampling
- The evaluation of executed resampling iterations

This sub-section will provide instructions on how to implement nested resampling, accounting for both inner and outer resampling in mlr3.

3.1 Hyperparameter Tuning

Hyperparameters are second-order parameters of machine learning models that, while often not explicitly optimized during the model estimation process, can have important impacts on the outcome and predictive performance of a model. Typically, hyperparameters are fixed before training a model. However, because the output of a model can be sensitive to the specification of hyperparameters, it is often recommended to make an informed decision about which hyperparameter settings may yield better model performance. In many cases, hyperparameter settings may be chosen *a priori*, but it can be advantageous to try different settings before fitting your model on the training data. This process is often called 'tuning' your model.

Hyperparameter tuning is supported via the extension package mlr3tuning. Below you can find an illustration of the process:



At the heart of mlr3tuning are the R6 classes:

- `TuningInstance`: This class describes the tuning problem and stores results.
- `Tuner`: This class is the base class for implementations of tuning algorithms.

3.1.1 The `TuningInstance` Class

The following sub-section examines the optimization of a simple classification tree on the Pima Indian Diabetes data set.

```
task = tsk("pima")
print(task)
```

```
## <TaskClassif:pima> (768 x 9)
## * Target: diabetes
## * Properties: twoclass
## * Features (8):
##   - dbl (8): age, glucose, insulin, mass, pedigree, pregnant, pressure,
##     triceps
```

We use the classification tree from rpart and choose a subset of the hyperparameters we want to tune. This is often referred to as the "tuning space".

```
learner = lrn("classif.rpart")
learner$param_set
```

```
## ParamSet:
##                 id    class lower upper levels     default value
## 1:        minsplit ParamInt     1   Inf                 20
## 2:       minbucket ParamInt     1   Inf        <NoDefault>
## 3:              cp ParamDbl     0     1               0.01
## 4:      maxcompete ParamInt     0   Inf                  4
## 5:    maxsurrogate ParamInt     0   Inf                  5
## 6:        maxdepth ParamInt     1    30                 30
## 7:    usesurrogate ParamInt     0     2                  2
## 8: surrogatestyle ParamInt     0     1                  0
## 9:            xval ParamInt     0   Inf                 10     0
```

Here, we opt to tune two parameters:

- The complexity `cp`
- The termination criterion `minsplit`

The tuning space has to be bound, therefore one has to set lower and upper bounds:

```
library("paradox")
tune_ps = ParamSet$new(list(
  ParamDbl$new("cp", lower = 0.001, upper = 0.1),
  ParamInt$new("minsplit", lower = 1, upper = 10)
))
tune_ps
```

```
## ParamSet:
##          id    class lower upper levels     default value
## 1:       cp ParamDbl 0.001   0.1        <NoDefault>
## 2: minsplit ParamInt 1.000  10.0        <NoDefault>
```

Next, we need to specify how to evaluate the performance. For this, we need to choose a resampling strategy and a performance measure.

```
hout = rsmp("holdout")
measure = msr("classif.ce")
```

Finally, one has to select the budget available, to solve this tuning instance. This is done by selecting one of the available `Terminators`:

- Terminate after a given time (`TerminatorClockTime`)
- Terminate after a given amount of iterations (`TerminatorEvals`)
- Terminate after a specific performance is reached (`TerminatorPerfReached`)
- Terminate when tuning does not improve (`TerminatorStagnation`)
- A combination of the above in an *ALL* or *ANY* fashion (`TerminatorCombo`)

For this short introduction, we specify a budget of 20 evaluations and then put everything together into a `TuningInstance`:

```
library("mlr3tuning")

evals20 = term("evals", n_evals = 20)

instance = TuningInstance$new(
  task = task,
  learner = learner,
  resampling = hout,
  measures = measure,
  param_set = tune_ps,
  terminator = evals20
)
print(instance)
```

```
## <TuningInstance>
## * State:  Not tuned
## * Task: <TaskClassif:pima>
## * Learner: <LearnerClassifRpart:classif.rpart>
## * Measures: classif.ce
## * Resampling: <ResamplingHoldout>
## * Terminator: <TerminatorEvals>
## * bm_args: list()
## * n_evals: 0
## ParamSet:
##          id    class lower upper levels     default value
## 1:       cp ParamDbl 0.001   0.1        <NoDefault>
## 2: minsplit ParamInt 1.000  10.0        <NoDefault>
```

To start the tuning, we still need to select how the optimization should take place. In other words, we need to choose the **optimization algorithm** via the `Tuner` class.

3.1.2 The `Tuner` Class

The following algorithms are currently implemented in mlr3tuning:

- Grid Search (`TunerGridSearch`)
- Random Search (`TunerRandomSearch`) (Bergstra and Bengio 2012)
- Generalized Simulated Annealing (`TunerGenSA`)

In this example, we will use a simple grid search with a grid resolution of 10:

```
tuner = tnr("grid_search", resolution = 5)
```

Since we have only numeric parameters, TunerGridSearch will create a grid of equally-sized steps between the respective upper and lower bounds. As we have two hyperparameters with a resolution of 5, the two-dimensional grid consists of $5^2 = 25$ configurations. Each configuration serves as hyperparameter setting for the classification tree and triggers a 3-fold cross validation on the task. All configurations will be examined by the tuner (in a random order), until either all configurations are evaluated or the Terminator signals that the budget is exhausted.

3.1.3 Triggering the Tuning

To start the tuning, we simply pass the TuningInstance to the `$tune()` method of the initialized Tuner. The tuner proceeds as follow:

1. The Tuner proposes at least one hyperparameter configuration (the Tuner and may propose multiple points to improve parallelization, which can be controlled via the setting `batch_size`).
2. For each configuration, a Learner is fitted on Task using the provided Resampling. The results are combined with other results from previous iterations to a single BenchmarkResult.
3. The Terminator is queried if the budget is exhausted. If the budget is not exhausted, restart with 1) until it is.
4. Determine the configuration with the best observed performance.
5. Return a named list with the hyperparameter settings (`"values"`) and the corresponding measured performance (`"performance"`).

```
result = tuner$tune(instance)
print(result)
```

```
## NULL
```

One can investigate all resamplings which were undertaken, using the `$archive()` method of the TuningInstance. Here, we just extract the performance values and the hyperparameters:

```
instance$archive(unnest = "params")[, c("cp", "minsplit", "classif.ce")]
```

```
##          cp minsplit classif.ce
##  1: 0.05050        1     0.2656
##  2: 0.05050        5     0.2656
##  3: 0.10000        8     0.2656
##  4: 0.07525       10     0.2656
##  5: 0.02575        8     0.2500
##  6: 0.10000        1     0.2656
##  7: 0.05050       10     0.2656
##  8: 0.00100       10     0.3320
##  9: 0.05050        3     0.2656
## 10: 0.02575       10     0.2500
## 11: 0.00100        3     0.3438
## 12: 0.02575        1     0.2500
## 13: 0.00100        8     0.3359
## 14: 0.02575        5     0.2500
## 15: 0.07525        1     0.2656
## 16: 0.07525        3     0.2656
## 17: 0.02575        3     0.2500
## 18: 0.07525        5     0.2656
## 19: 0.10000       10     0.2656
## 20: 0.00100        5     0.3477
```

In sum, the grid search evaluated 20/25 different configurations of the grid in a random order before the `Terminator` stopped the tuning.

Now the optimized hyperparameters can take the previously created `Learner`, set the returned hyperparameters and train it on the full dataset.

```r
learner$param_set$values = instance$result$params
learner$train(task)
```

The trained model can now be used to make a prediction on external data. Note that predicting on observations present in the `task`, should be avoided. The model has seen these observations already during tuning and therefore results would be statistically biased. Hence, the resulting performance measure would be over-optimistic. Instead, to get statistically unbiased performance estimates for the current task, nested resampling is required.

3.1.4 Automating the Tuning

The `AutoTuner` wraps a learner and augments it with an automatic tuning for a given set of hyperparameters. Because the `AutoTuner` itself inherits from the `Learner` base class, it can be used like any other learner. Analogously to the previous subsection, a new classification tree learner is created. This classification tree learner automatically tunes the parameters `cp` and `minsplit` using an inner resampling (holdout). We create a terminator which allows 10 evaluations, and use a simple random search as tuning algorithm:

```r
library("paradox")
library("mlr3tuning")

learner = lrn("classif.rpart")
resampling = rsmp("holdout")
measures = msr("classif.ce")
tune_ps = ParamSet$new(list(
  ParamDbl$new("cp", lower = 0.001, upper = 0.1),
  ParamInt$new("minsplit", lower = 1, upper = 10)
))
terminator = term("evals", n_evals = 10)
tuner = tnr("random_search")

at = AutoTuner$new(
  learner = learner,
  resampling = resampling,
  measures = measures,
  tune_ps = tune_ps,
  terminator = terminator,
  tuner = tuner
)
at
```

```
## <AutoTuner:classif.rpart.tuned>
## * Model: -
## * Parameters: xval=0
## * Packages: rpart
## * Predict Type: response
## * Feature types: logical, integer, numeric, factor, ordered
## * Properties: importance, missings, multiclass, selected_features,
##   twoclass, weights
```

We can now use the learner like any other learner, calling the `$train()` and `$predict()` method. This time however, we pass it to `benchmark()` to compare the tuner to a classification tree without tuning. This way, the `AutoTuner` will do its resampling for tuning on the training set of the respective split of the outer resampling. The learner then undertakes predictions using the test set of the outer resampling. This yields unbiased performance measures, as the observations in the test set have not been used during tuning or fitting of the respective learner. This is called nested resampling.

To compare the tuned learner with the learner using its default, we can use `benchmark()`:

```
grid = benchmark_grid(
  task = tsk("pima"),
  learner = list(at, lrn("classif.rpart")),
  resampling = rsmp("cv", folds = 3)
)
bmr = benchmark(grid)
bmr$aggregate(measures)
```

```
##    nr  resample_result task_id          learner_id resampling_id iters
## 1:  1 <ResampleResult>    pima classif.rpart.tuned            cv     3
## 2:  2 <ResampleResult>    pima       classif.rpart            cv     3
##    classif.ce
## 1:     0.2552
## 2:     0.2487
```

Note that we do not expect any differences compared to the non-tuned approach for multiple reasons:

- the task is too easy
- the task is rather small, and thus prone to overfitting
- the tuning budget (10 evaluations) is small
- rpart does not benefit that much from tuning

3.2 Feature Selection / Filtering

Often, data sets include a large number of features. The technique of extracting a subset of relevant features is called “feature selection”.

The objective of feature selection is to fit the sparse dependent of a model on a subset of available data features in the most suitable manner. Feature selection can enhance the interpretability of the model, speed up the learning process and improve the learner performance. Different approaches exist to identify the relevant features. Two different approaches are emphasized in the literature:

one is called Filtering and the other approach is often referred to as feature subset selection or wrapper methods.

What are the differences (Chandrashekar and Sahin 2014)?

- **Filtering**: An external algorithm computes a rank of the variables (e.g. based on the correlation to the response). Then, features are subsetted by a certain criteria, e.g. an absolute number or a percentage of the number of variables. The selected features will then be used to fit a model (with optional hyperparameters selected by tuning). This calculation is usually cheaper than "feature subset selection" in terms of computation time.
- **Wrapper Methods**: Here, no ranking of features is done. Features are selected by a (random) subset of the data. Then, we fit a model and subsequently assess the performance. This is done for a lot of feature combinations in a cross-validation (CV) setting and the best combination is reported. This method is very computationally intensive as a lot of models are fitted. Also, strictly speaking all these models would need to be tuned before the performance is estimated. This would require an additional nested level in a CV setting. After undertaken all of these steps, the selected subset of features is again fitted (with optional hyperparameters selected by tuning).

There is also a third approach which can be attributed to the "filter" family: The embedded feature-selection methods of some `Learner`. Read more about how to use these in section embedded feature-selection methods.

Ensemble filters built upon the idea of stacking single filter methods. These are not yet implemented.

All functionality that is related to feature selection is implemented via the extension package mlr3filters.

3.2.1 Filters

Filter methods assign an importance value to each feature. Based on these values the features can be ranked. Thereafter, we are able to select a feature subset. There is a list of all implemented filter methods in the Appendix.

3.2.2 Calculating filter values

Currently, only classification and regression tasks are supported.

The first step it to create a new R object using the class of the desired filter method. Each object of class `Filter` has a `.$calculate()` method which calculates the filter values and ranks them in a descending order.

```r
library("mlr3filters")
filter = FilterJMIM$new()

task = tsk("iris")
filter$calculate(task)

as.data.table(filter)
```

```
##         feature  score
## 1: Sepal.Length 1.0401
## 2:  Petal.Width 0.9894
## 3: Petal.Length 0.9881
## 4:  Sepal.Width 0.8314
```

Some filters support changing specific hyperparameters. This is done similar to setting hyperparameters of a Learner using .$param_set$values:

```
filter_cor = FilterCorrelation$new()
filter_cor$param_set
```

```
## ParamSet:
##        id    class lower upper
## 1:    use ParamFct    NA    NA
## 2: method ParamFct    NA    NA
##                                                                  levels
## 1: everything,all.obs,complete.obs,na.or.complete,pairwise.complete.obs
## 2:                                             pearson,kendall,spearman
##       default value
## 1: everything
## 2:    pearson
```

```
# change parameter 'method'
filter_cor$param_set$values = list(method = "spearman")
filter_cor$param_set
```

```
## ParamSet:
##        id    class lower upper
## 1:    use ParamFct    NA    NA
## 2: method ParamFct    NA    NA
##                                                                  levels
## 1: everything,all.obs,complete.obs,na.or.complete,pairwise.complete.obs
## 2:                                             pearson,kendall,spearman
##       default    value
## 1: everything
## 2:    pearson spearman
```

Rather than taking the “long” R6 way to create a filter, there is also a built-in shorthand notation for filter creation:

```
filter = flt("cmim")
filter
```

```
## <FilterCMIM:cmim>
## Task Types: classif, regr
## Task Properties: -
## Packages: praznik
## Feature types: integer, numeric, factor, ordered
```

3.2.3 Variable Importance Filters

All Learner with the property “importance” come with integrated feature selection methods.

You can find a list of all learners with this property in the Appendix.

For some learners the desired filter method needs to be set during learner creation. For example, learner `classif.ranger` (in the package mlr3learners) comes with multiple integrated methods. See the help page of `ranger::ranger`. To use method "impurity", you need to set the filter method during construction.

```
library("mlr3learners")
lrn = lrn("classif.ranger", importance = "impurity")
```

Now you can use the `mlr3filters::FilterImportance` class for algorithm-embedded methods to filter a `Task`.

```
library("mlr3learners")

task = tsk("iris")
filter = flt("importance", learner = lrn)
filter$calculate(task)
head(as.data.table(filter), 3)
```

```
##          feature score
## 1:  Petal.Width 45.99
## 2: Petal.Length 40.19
## 3: Sepal.Length 10.53
```

3.2.4 Ensemble Methods

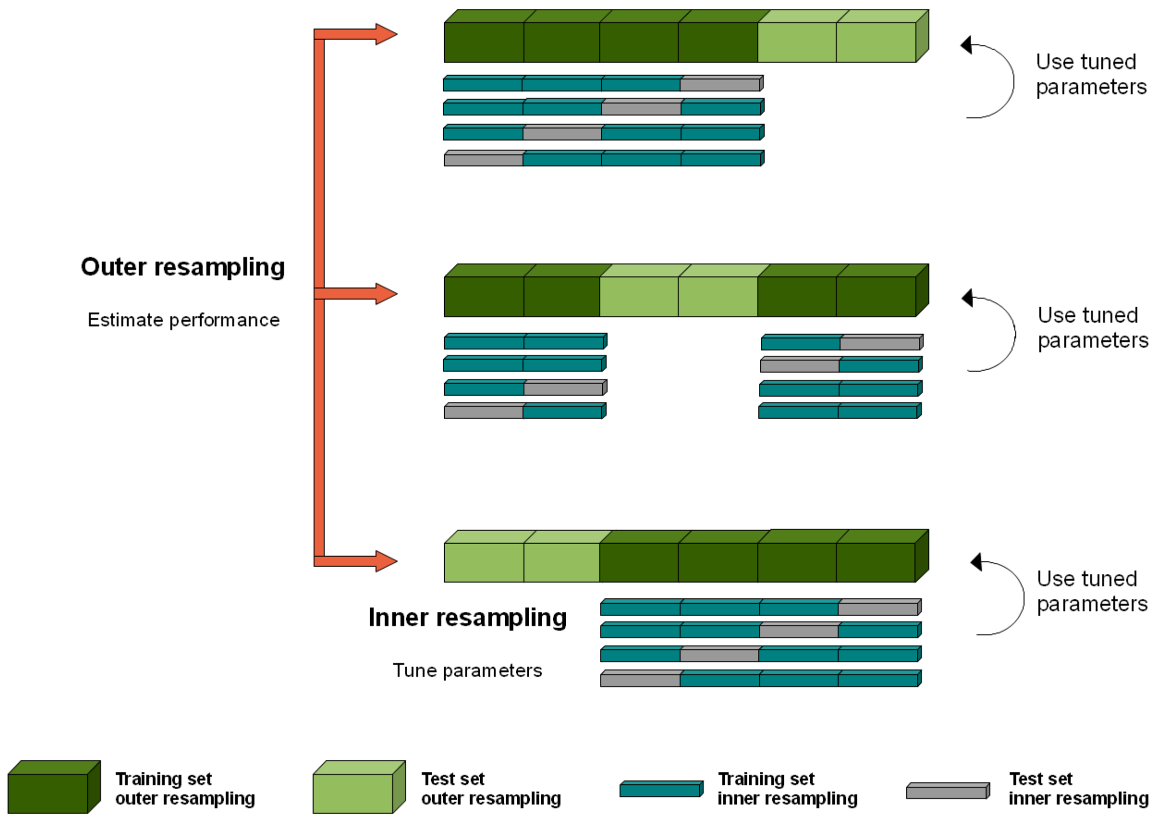
Work in progress.

3.2.5 Wrapper Methods

Work in progress - via package mlr3fswrap

3.3 Nested Resampling

In order to obtain unbiased performance estimates for learners, all parts of the model building (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 hyperparameter 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. Subsequently, we can evaluate the performance of the learner on the outer test sets.

In mlr3, you can run nested resampling for free without programming any loops by using the `mlr3tuning::AutoTuner` class. This works as follows:

1. Generate a wrapped Learner via class `mlr3tuning::AutoTuner` or `mlr3filters::AutoSelect` (not yet implemented).
2. Specify all required settings - see section "Automating the Tuning" for help.
3. Call function `resample()` or `benchmark()` with the created `Learner`.

You can freely combine different inner and outer resampling strategies.

A common setup is prediction and performance evaluation on a fixed outer test set. This can be achieved by passing the `Resampling` strategy (`rsmp("holdout")`) as the outer resampling instance to either `resample()` or `benchmark()`.

The inner resampling strategy could be a cross-validation one (`rsmp("cv")`) as the sizes of the outer training sets might differ. Per default, the inner resample description is instantiated once for every outer training set.

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

3.3.1 Execution

To optimize hyperparameters or conduct feature selection in a nested resampling you need to create learners using either:

- the `AutoTuner` class, or
- the `mlr3filters::AutoSelect` class (not yet implemented)

We use the example from section "Automating the Tuning" and pipe the resulting learner into a `resample()` call.

```
library("mlr3tuning")
task = tsk("iris")
learner = lrn("classif.rpart")
resampling = rsmp("holdout")
measures = msr("classif.ce")
param_set = paradox::ParamSet$new(
  params = list(paradox::ParamDbl$new("cp", lower = 0.001, upper = 0.1)))
terminator = term("evals", n_evals = 5)
tuner = tnr("grid_search", resolution = 10)

at = AutoTuner$new(learner, resampling, measures = measures,
  param_set, terminator, tuner = tuner)
```

Now construct the `resample()` call:

```
resampling_outer = rsmp("cv", folds = 3)
rr = resample(task = task, learner = at, resampling = resampling_outer)
```

3.3.2 Evaluation

With the created `ResampleResult` we can now inspect the executed resampling iterations more closely. See the section on Resampling for more detailed information about `ResampleResult` objects.

For example, we can query the aggregated performance result:

```
rr$aggregate()
```

```
## classif.ce
##    0.07333
```

Check for any errors in the folds during execution (if there is not output, warnings or errors recorded, this is an empty `data.table()`:

```
rr$errors
```

```
## Empty data.table (0 rows and 2 cols): iteration,msg
```

Or take a look at the confusion matrix of the joined predictions:

```r
rr$prediction()$confusion
```

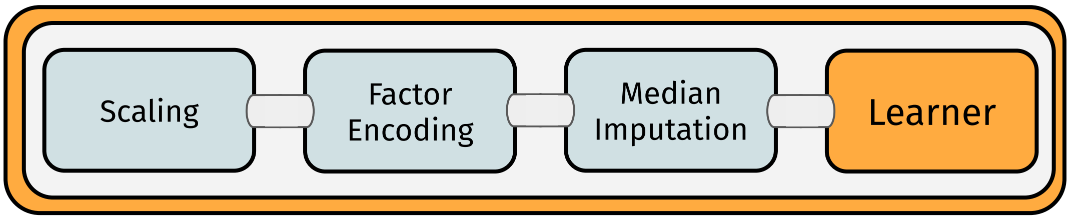
```
##             truth
## response     setosa versicolor virginica
##   setosa         50          0         0
##   versicolor      0         45         6
##   virginica       0          5        44
```

4 Pipelines

mlr3pipelines is a dataflow programming toolkit. This chapter focuses on the applicant's side of the package. A more in-depth and technically oriented vignette can be found in the mlr3pipeline vignette.

Machine learning workflows can be written as directed "Graphs"/"Pipelines" that represent data flows between preprocessing, model fitting, and ensemble learning units in an expressive and intuitive language. We will most often use the term "Graph" in this manual but it can interchangeably be used with "pipeline" or "workflow".

Below you can examine an example for such a graph:



Single computational steps can be represented as so-called PipeOps, which can then be connected with directed edges in a Graph. The scope of mlr3pipelines is still growing. Currently supported features are:

- Data manipulation and preprocessing operations, e.g. PCA, feature filtering, imputation
- Task subsampling for speed and outcome class imbalance handling
- mlr3 Learner operations for prediction and stacking
- Ensemble methods and aggregation of predictions

Additionally, we implement several meta operators that can be used to construct powerful pipelines:

- Simultaneous path branching (data going both ways)
- Alternative path branching (data going one specific way, controlled by hyperparameters)

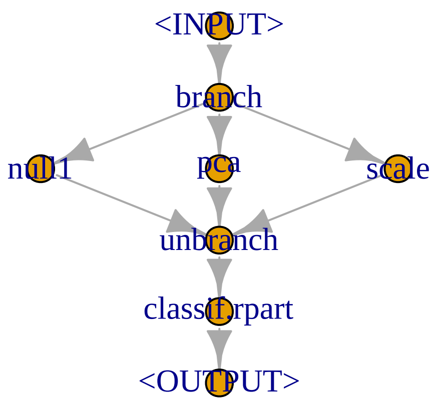
An extensive introduction to creating custom **PipeOps** (PO's) can be found in the technical introduction.

Using methods from mlr3tuning, it is even possible to simultaneously optimize parameters of multiple processing units.

A predecessor to this package is the mlrCPO package, which works with mlr 2.x. Other packages that provide, to varying degree, some preprocessing functionality or machine learning domain specific language, are:

- the caret package and the related recipes project
- the dplyr package

An example for a Pipeline that can be constructed using mlr3pipelines is depicted below:



4.1 The Building Blocks: PipeOps

The building blocks of mlr3pipelines are **PipeOp**-objects (PO). They can be constructed directly using `PipeOp<NAME>$new()`, but the recommended way is to retrieve them from the `mlr_pipeops` dictionary:

```r
library("mlr3pipelines")
as.data.table(mlr_pipeops)
```

```
##                  key      packages input.num output.num input.type.train
##  1:           boxcox bestNormalize         1          1             Task
##  2:           branch                       1         NA                *
##  3:            chunk                       1         NA             Task
##  4:   classbalancing                       1          1      TaskClassif
##  5:       classifavg         stats        NA          1             NULL
##  6:     classweights                       1          1      TaskClassif
##  7:         colapply                       1          1             Task
##  8: collapsefactors                       1          1             Task
##  9:             copy                       1         NA                *
## 10:           encode         stats         1          1             Task
## 11:     encodeimpact                       1          1             Task
## 12:       encodelmer   lme4,nloptr         1          1             Task
## 13:     featureunion                      NA          1             Task
## 14:           filter                       1          1             Task
## 15:       fixfactors                       1          1             Task
## 16:          histbin      graphics         1          1             Task
## 17:             ica       fastICA         1          1             Task
## 18:      imputehist      graphics         1          1             Task
## 19:      imputemean                       1          1             Task
## 20:    imputemedian         stats         1          1             Task
## 21:    imputenewlvl                       1          1             Task
## 22:    imputesample                       1          1             Task
## 23:       kernelpca       kernlab         1          1             Task
## 24:         learner                       1          1      TaskClassif
## 25:      learner_cv                       1          1      TaskClassif
## 26:         missind                       1          1             Task
## 27:     modelmatrix         stats         1          1             Task
## 28:          mutate                       1          1             Task
## 29:             nop                       1          1                *
## 30:             pca                       1          1             Task
## 31:     quantilebin         stats         1          1             Task
## 32:         regravg                      NA          1             NULL
## 33: removeconstants                       1          1             Task
## 34:           scale                       1          1             Task
## 35:     scalemaxabs                       1          1             Task
## 36:      scalerange                       1          1             Task
## 37:          select                       1          1             Task
## 38:           smote   smotefamily         1          1             Task
## 39:     spatialsign                       1          1             Task
## 40:       subsample                       1          1             Task
## 41:        unbranch                      NA          1                *
## 42:      yeojohnson bestNormalize         1          1             Task
##                  key      packages input.num output.num input.type.train
##     input.type.predict output.type.train output.type.predict
##  1:               Task              Task                Task
##  2:                  *                 *                   *
##  3:               Task              Task                Task
##  4:        TaskClassif       TaskClassif         TaskClassif
##  5:  PredictionClassif              NULL   PredictionClassif
##  6:        TaskClassif       TaskClassif         TaskClassif
##  7:               Task              Task                Task
##  8:               Task              Task                Task
##  9:                  *                 *                   *
## 10:               Task              Task                Task
## 11:               Task              Task                Task
## 12:               Task              Task                Task
## 13:               Task              Task                Task
## 14:               Task              Task                Task
## 15:               Task              Task                Task
## 16:               Task              Task                Task
## 17:               Task              Task                Task
## 18:               Task              Task                Task
## 19:               Task              Task                Task
## 20:               Task              Task                Task
## 21:               Task              Task                Task
## 22:               Task              Task                Task
## 23:                Task               Task                Task
## 24:         TaskClassif               NULL    PredictionClassif
## 25:         TaskClassif        TaskClassif          TaskClassif
## 26:                Task               Task                Task
## 27:                Task               Task                Task
## 28:                Task               Task                Task
## 29:                   *                  *                   *
## 30:                Task               Task                Task
## 31:                Task               Task                Task
## 32:      PredictionRegr               NULL      PredictionRegr
## 33:                Task               Task                Task
## 34:                Task               Task                Task
## 35:                Task               Task                Task
## 36:                Task               Task                Task
## 37:                Task               Task                Task
## 38:                Task               Task                Task
## 39:                Task               Task                Task
## 40:                Task               Task                Task
## 41:                   *                  *                   *
## 42:                Task               Task                Task
##     input.type.predict output.type.train output.type.predict
```

Single POs can be created using `mlr_pipeops$get(<name>)`:

```
pca = mlr_pipeops$get("pca")
```

or using **syntactic sugar**

```
pca = po("pca")
```

Some POs require additional arguments for construction:

```
learner = mlr_pipeops$get("learner")

# Error in as_learner(learner) : argument "learner" is missing, with no default argument "learner" is
```

```
learner = mlr_pipeops$get("learner", mlr_learners$get("classif.rpart"))
```

or in short `po("learner", lrn("classif.rpart"))`.

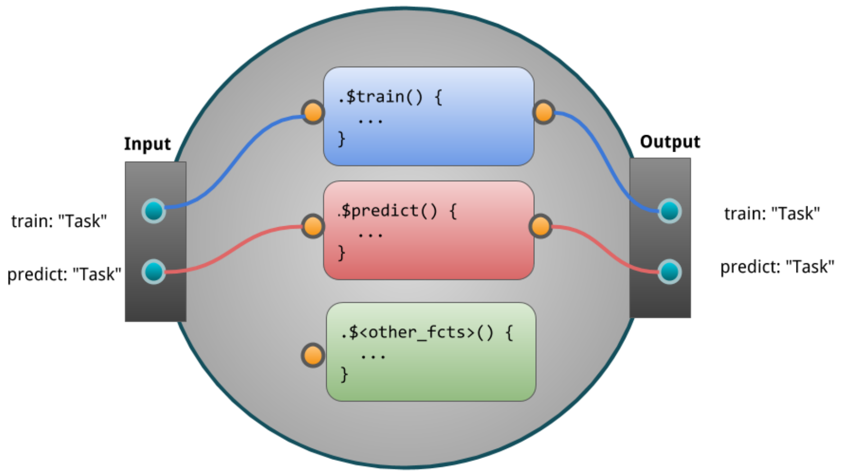
Hyperparameters of POs can be set through the `param_vals` argument. Here we set the fraction of features for a filter:

```
filter = mlr_pipeops$get("filter",
  filter = mlr3filters::FilterVariance$new(),
  param_vals = list(filter.frac = 0.5))
```

or in short notation:

```
po("filter", mlr3filters::FilterVariance$new(), filter.frac = 0.5)
```

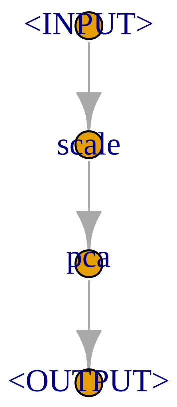
The figure below shows an exemplary `PipeOp`. It takes an input, transforms it during `.$train` and `.$predict` and returns data:



4.2 The Pipeline Operator: %>>%

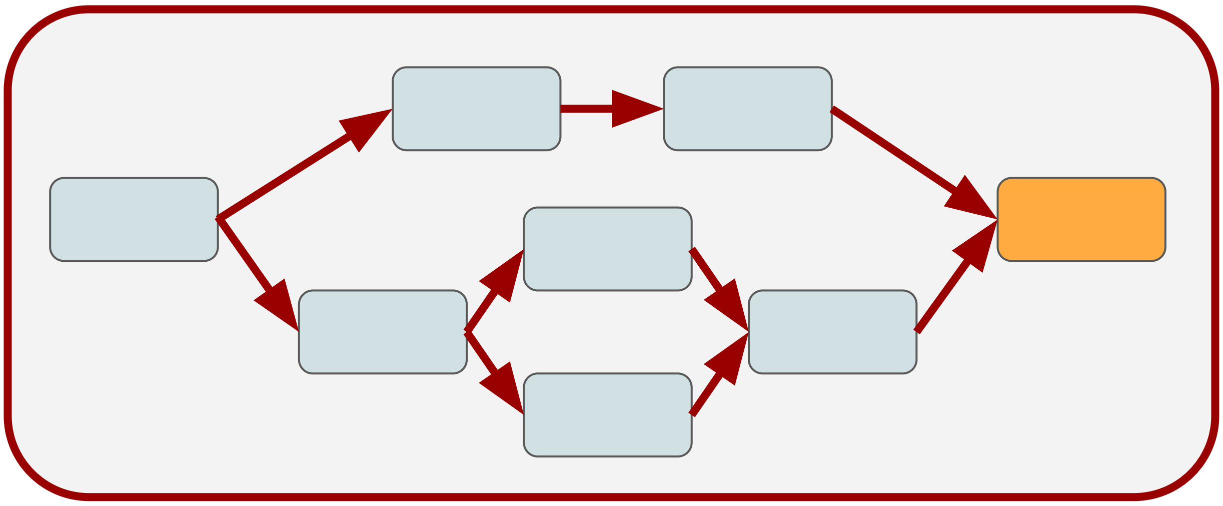
It is possible to create intricate `Graphs` with edges going all over the place (as long as no loops are introduced). Irrespective, there is usually a clear direction of flow between "layers" in the `Graph`. It is therefore convenient to build up a `Graph` from layers. This can be done using the `%>>%` ("double-arrow") operator. It takes either a `PipeOp` or a `Graph` on each of its sides and connects all of the outputs of its left-hand side to one of the inputs each of its right-hand side. The number of inputs therefore must match the number of outputs.

```r
library("magrittr")

gr = mlr_pipeops$get("scale") %>>% mlr_pipeops$get("pca")
gr$plot(html = FALSE)
```



4.3 Nodes, Edges and Graphs

POs are combined into Graphs. The manual way (= hard way) to construct a Graph is to create an empty graph first. Then one fills the empty graph with POs, and connects edges between the POs. Conceptually, this may look like this:



POs are identified by their `$id`. Note that the operations all modify the object in-place and return the object itself. Therefore, multiple modifications can be chained.

For this example we use the `pca` PO defined above and a new PO named "mutate". The latter creates a new feature from existing variables. Additionally, we use the filter PO again.

```
mutate = mlr_pipeops$get("mutate")

filter = mlr_pipeops$get("filter",
  filter = mlr3filters::FilterVariance$new(),
  param_vals = list(filter.frac = 0.5))
```

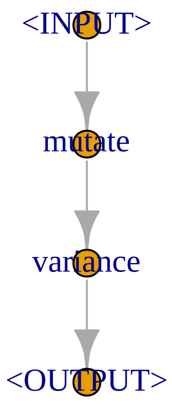
```
graph = Graph$new()$
  add_pipeop(mutate)$
  add_pipeop(filter)$
  add_edge("mutate", "variance")  # add connection mutate -> filter
```

The much quicker way is to use the `%>>%` operator to chain POs or `Graph` s. The same result as above can be achieved by doing the following:

```
graph = mutate %>>% filter
```

Now the `Graph` can be inspected using its `$plot()` function:

```
graph$plot()
```



Chaining multiple POs of the same kind

If multiple POs of the same kind should be chained, it is necessary to change the `id` to avoid name clashes. This can be done by either accessing the `$id` slot or during construction:

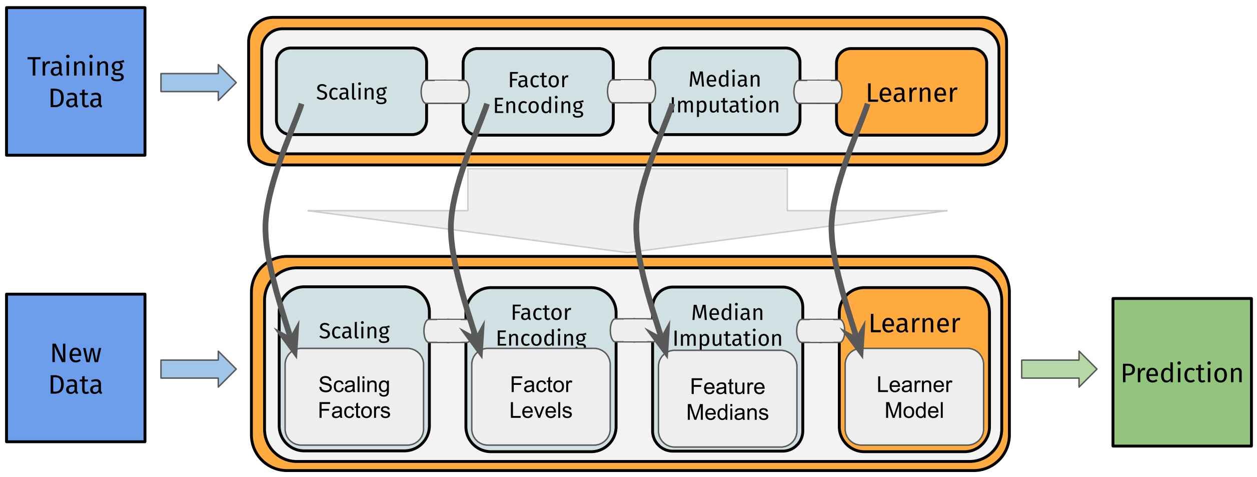
```
graph$add_pipeop(mlr_pipeops$get("pca"))
```

```
graph$add_pipeop(mlr_pipeops$get("pca", id = "pca2"))
```

4.4 Modeling

The main purpose of a `Graph` is to build combined preprocessing and model fitting pipelines that can be used as `mlr3` `Learner`.

Conceptually, the process may be summarized as follows:



In the following we chain two preprocessing tasks:

- mutate (creation of a new feature)
- filter (filtering the dataset)

Subsequently one can chain a PO learner to train and predict on the modified dataset.

```
mutate = mlr_pipeops$get("mutate")
filter = mlr_pipeops$get("filter",
  filter = mlr3filters::FilterVariance$new(),
  param_vals = list(filter.frac = 0.5))

graph = mutate %>>%
  filter %>>%
  mlr_pipeops$get("learner",
    learner = mlr_learners$get("classif.rpart"))
```

Until here we defined the main pipeline stored in `Graph`. Now we can train and predict the pipeline:

```
task = mlr_tasks$get("iris")
graph$train(task)
```

```
## $classif.rpart.output
## NULL
```

```
graph$predict(task)
```

```
## $classif.rpart.output
## <PredictionClassif> for 150 observations:
##     row_id     truth  response
##          1    setosa    setosa
##          2    setosa    setosa
##          3    setosa    setosa
## ---
##        148 virginica virginica
##        149 virginica virginica
##        150 virginica virginica
```

Rather than calling `$train()` and `$predict()` manually, we can put the pipeline `Graph` into a `GraphLearner` object. A `GraphLearner` encapsulates the whole pipeline (including the preprocessing steps) and can be put into `resample()` or `benchmark()` . If you are familiar with the old *mlr* package, this is the equivalent of all the `make*Wrapper()` functions. The pipeline being encapsulated (here `Graph`) must always produce a `Prediction` with its `$predict()` call, so it will probably contain at least one `PipeOpLearner` .

```
glrn = GraphLearner$new(graph)
```

This learner can be used for model fitting, resampling, benchmarking, and tuning:

```
cv3 = rsmp("cv", folds = 3)
resample(task, glrn, cv3)
```

```
## <ResampleResult> of 3 iterations
## * Task: iris
## * Learner: mutate.variance.classif.rpart
## * Warnings: 0 in 0 iterations
## * Errors: 0 in 0 iterations
```

4.4.1 Setting Hyperparameters

Individual POs offer hyperparameters because they contain `$param_set` slots that can be read and written from `$param_set$values` (via the paradox package). The parameters get passed down to the `Graph`, and finally to the `GraphLearner` . This makes it not only possible to easily change the behavior of a `Graph` / `GraphLearner` and try different settings manually, but also to perform tuning using the mlr3tuning package.

```
glrn$param_set$values$variance.filter.frac = 0.25
cv3 = rsmp("cv", folds = 3)
resample(task, glrn, cv3)
```

```
## <ResampleResult> of 3 iterations
## * Task: iris
## * Learner: mutate.variance.classif.rpart
## * Warnings: 0 in 0 iterations
## * Errors: 0 in 0 iterations
```

4.4.2 Tuning

If you are unfamiliar with tuning in mlr3, we recommend to take a look at the section about tuning first. Here we define a `ParamSet` for the “rpart” learner and the “variance” filter which should be optimized during the tuning process.

```
library("paradox")
ps = ParamSet$new(list(
  ParamDbl$new("classif.rpart.cp", lower = 0, upper = 0.05),
  ParamDbl$new("variance.filter.frac", lower = 0.25, upper = 1)
))
```

After having defined the `PerformanceEvaluator`, a random search with 10 iterations is created. For the inner resampling, we are simply using holdout (single split into train/test) to keep the runtimes reasonable.

```
library("mlr3tuning")
instance = TuningInstance$new(
  task = task,
  learner = glrn,
  resampling = rsmp("holdout"),
  measures = msr("classif.ce"),
  param_set = ps,
  terminator = term("evals", n_evals = 20)
)
```

```
tuner = TunerRandomSearch$new()
tuner$tune(instance)
```

The tuning result can be found in the `result` slot.

```
instance$result
```

4.5 Non-Linear Graphs

The Graphs seen so far all have a linear structure. Some POs may have multiple input or output channels. These channels make it possible to create non-linear Graphs with alternative paths taken by the data.

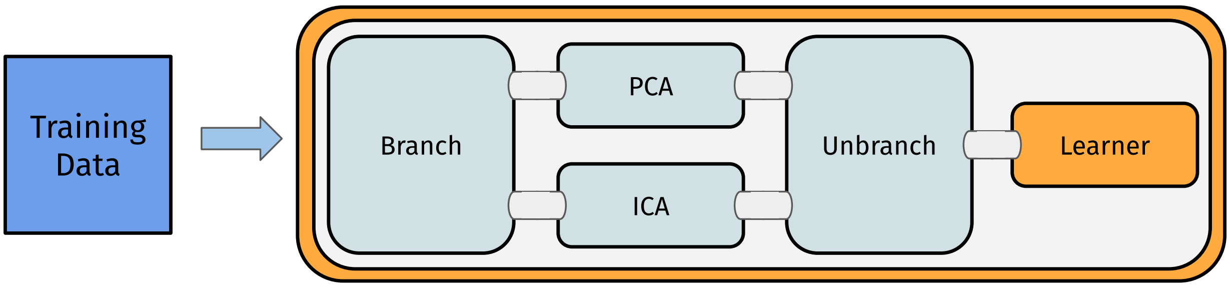
Possible types are:

- Branching: Splitting of a node into several paths, e.g. useful when comparing multiple feature-selection methods (pca, filters). Only one path will be executed.
- Copying: Splitting of a node into several paths, all paths will be executed (sequentially). Parallel execution is not yet supported.
- Stacking: Single graphs are stacked onto each other, i.e. the output of one `Graph` is the input for another. In machine learning this means that the prediction of one `Graph` is used as input for another `Graph`

4.5.1 Branching & Copying

The `PipeOpBranch` and `PipeOpUnbranch` POs make it possible to specify multiple alternative paths. Only one path is actually executed, the others are ignored. The active path is determined by a hyperparameter. This concept makes it possible to tune alternative preprocessing paths (or learner models).

Below a conceptual visualization of branching:



`PipeOp(Un)Branch` is initialized either with the number of branches, or with a `character`-vector indicating the names of the branches. If names are given, the "branch-choosing" hyperparameter becomes more readable. In the following, we set three options:

1. Doing nothing ("nop")
2. Applying a PCA
3. Scaling the data

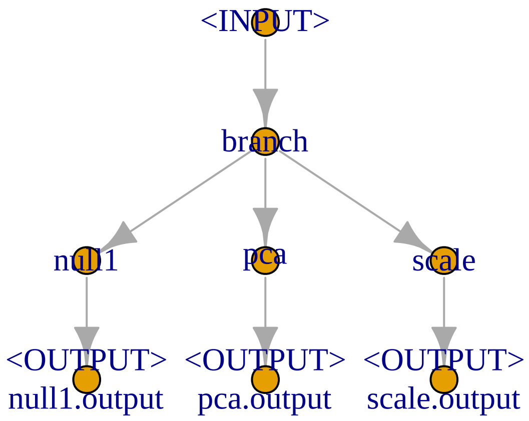
It is important to "unbranch" again after "branching", so that the outputs are merged into one result objects.

In the following we first create the branched graph and then show what happens if the "unbranching" is not applied:

```
graph = mlr_pipeops$get("branch", c("nop", "pca", "scale")) %>>%
  gunion(list(
      mlr_pipeops$get("nop", id = "null1"),
      mlr_pipeops$get("pca"),
      mlr_pipeops$get("scale")
  ))
```

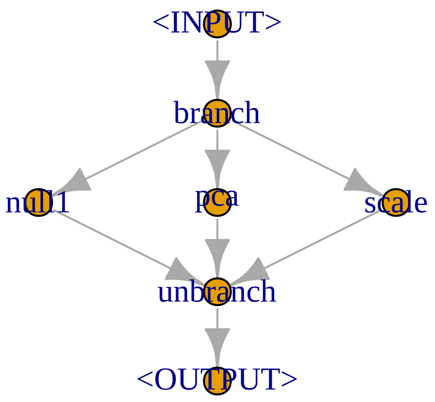
Without "unbranching" one creates the following graph:

```
graph$plot(html = FALSE)
```



Now when “unbranching”, we obtain the following results:

```r
(graph %>>% mlr_pipeops$get("unbranch", c("nop", "pca", "scale")))$plot(html = FALSE)
```



The same can be achieved using a shorter notation:

```r
# List of pipeops
opts = list(po("nop", "no_op"), po("pca"), po("scale"))
# List of po ids
opt_ids = mlr3misc::map_chr(opts, `[[`, "id")
po("branch", options = opt_ids) %>>%
  gunion(opts) %>>%
  po("unbranch", options = opt_ids)
```

```
## Graph with 5 PipeOps:
##         ID         State        sccssors       prdcssors
##     branch <<UNTRAINED>> no_op,pca,scale
##      no_op <<UNTRAINED>>        unbranch          branch
##        pca <<UNTRAINED>>        unbranch          branch
##      scale <<UNTRAINED>>        unbranch          branch
##   unbranch <<UNTRAINED>>                 no_op,pca,scale
```

4.5.2 Model Ensembles

We can leverage the different operations presented to connect POs. This allows us to form powerful graphs.

Before we go into details, we split the task into train and test indices.

```r
task = mlr_tasks$get("iris")
train.idx = sample(seq_len(task$nrow), 120)
test.idx = setdiff(seq_len(task$nrow), train.idx)
```

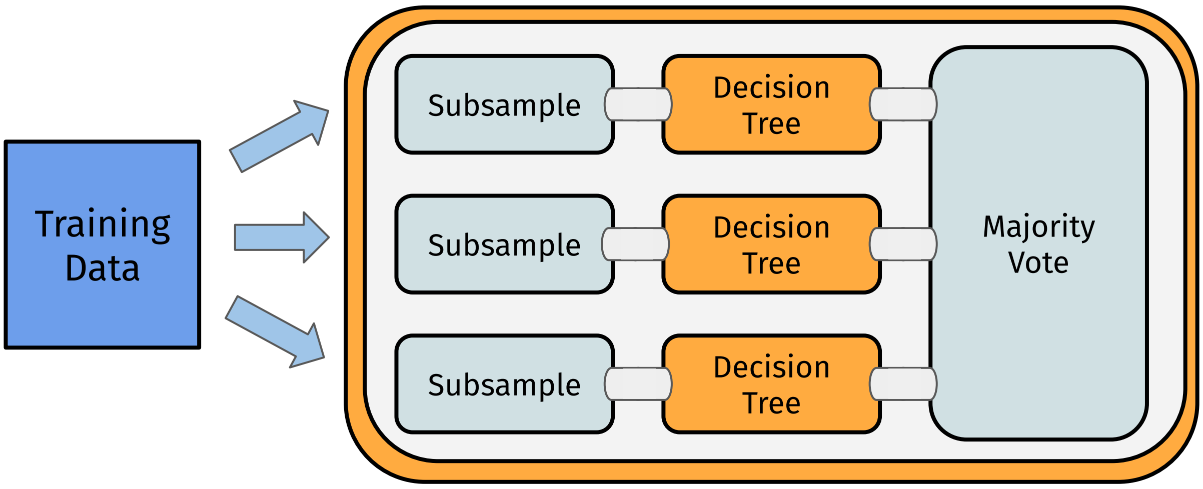
4.5.2.1 Bagging

We first examine Bagging introduced by (Breiman 1996). The basic idea is to create multiple predictors and then aggregate those to a single, more powerful predictor.

> "… multiple versions are formed by making bootstrap replicates of the learning set and using these as new learning sets" (Breiman 1996)

Bagging then aggregates a set of predictors by averaging (regression) or majority vote (classification). The idea behind bagging is, that a set of weak, but different predictors can be combined in order to arrive at a single, better predictor.

We can achieve this by downsampling our data before training a learner, repeating this e.g. 10 times and then performing a majority vote on the predictions. Graphically, it may be summarized as follows:



First, we create a simple pipeline, that uses `PipeOpSubsample` before a `PipeOpLearner` is trained:

```
single_pred = PipeOpSubsample$new(param_vals = list(frac = 0.7)) %>>%
  PipeOpLearner$new(mlr_learners$get("classif.rpart"))
```

We can now copy this operation 10 times using `greplicate` . The `greplicate` command allows us to parallelize many copies of an operation.

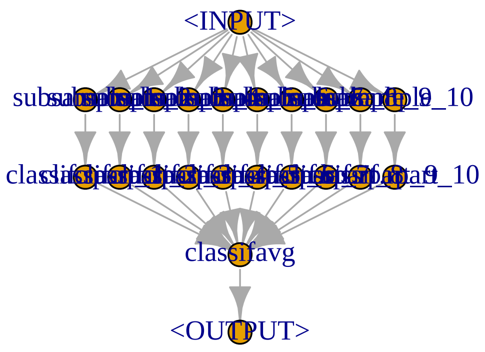
```
pred_set = greplicate(single_pred, 10L)
```

Afterwards we need to aggregate the 10 pipelines to form a single model:

```
bagging = pred_set %>>%
  PipeOpClassifAvg$new(innum = 10L)
```

Now we can plot again to see what happens:

```
bagging$plot(html = FALSE)
```


<INPUT>
subsample_1 ... subsample_10
classif.rpart_1 ... classif.rpart_10
classifavg
<OUTPUT>

This pipeline can again be used in conjunction with GraphLearner in order for Bagging to be used like a Learner:

```
baglrn = GraphLearner$new(bagging)
baglrn$train(task, train.idx)
baglrn$predict(task, test.idx)
```

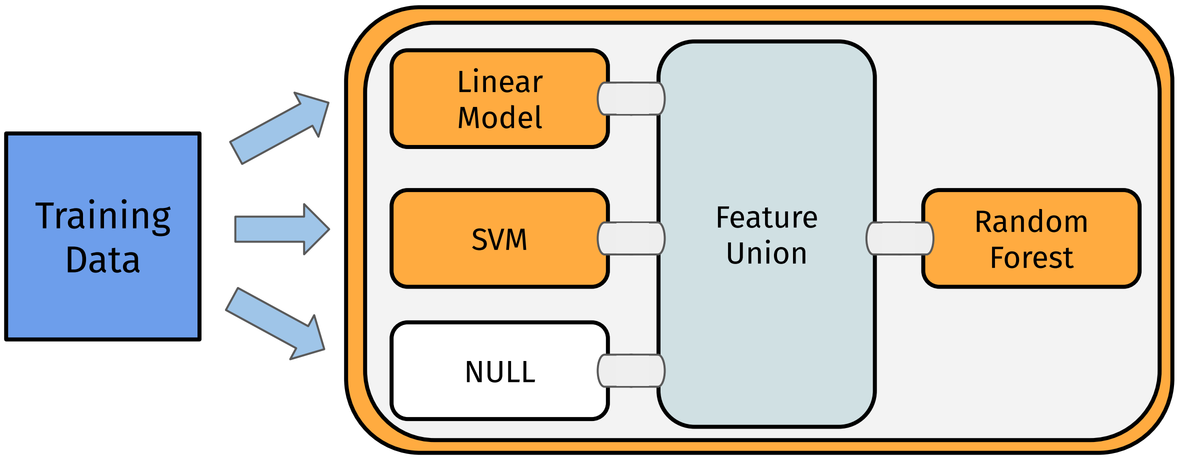
```
## <PredictionClassif> for 30 observations:
##     row_id     truth  response
##          3    setosa    setosa
##          6    setosa    setosa
##          9    setosa    setosa
## ---
##        136 virginica virginica
##        140 virginica virginica
##        144 virginica virginica
```

In conjunction with different `Backends`, this can be a very powerful tool. In cases when the data does not fully fit in memory, one can obtain a fraction of the data for each learner from a DataBackend and then aggregate predictions over all learners.

4.5.2.2 Stacking

Stacking (Wolpert 1992) is another technique that can improve model performance. The basic idea behind stacking is the use of predictions from one model as features for a subsequent model to possibly improve performance.

Below an conceptual illustration of stacking:



As an example we can train a decision tree and use the predictions from this model in conjunction with the original features in order to train an additional model on top.

To limit overfitting, we additionally do not predict on the original predictions of the learner. Instead, we predict on out-of-bag predictions. To do all this, we can use PipeOpLearnerCV .

PipeOpLearnerCV performs nested cross-validation on the training data, fitting a model in each fold. Each of the models is then used to predict on the out-of-fold data. As a result, we obtain predictions for every data point in our input data.

We first create a "level 0" learner, which is used to extract a lower level prediction. Additionally, we `clone()` the learner object to obtain a copy of the learner. Subsequently, one sets a custom id for the `PipeOp` .

```
lrn = mlr_learners$get("classif.rpart")
lrn_0 = PipeOpLearnerCV$new(lrn$clone())
lrn_0$id = "rpart_cv"
```

We use `PipeOpNOP` in combination with `gunion`, in order to send the unchanged Task to the next level. There it is combined with the predictions from our decision tree learner.

```
level_0 = gunion(list(lrn_0, PipeOpNOP$new()))
```

Afterwards, we want to concatenate the predictions from `PipeOpLearnerCV` and the original Task using `PipeOpFeatureUnion` :

```
combined = level_0 %>>% PipeOpFeatureUnion$new(2)
```

Now we can train another learner on top of the combined features:

```
stack = combined %>>% PipeOpLearner$new(lrn$clone())
stack$plot(html = FALSE)
```

```
stacklrn = GraphLearner$new(stack)
stacklrn$train(task, train.idx)
stacklrn$predict(task, test.idx)
```

In this vignette, we showed a very simple use-case for stacking. In many real-world applications, stacking is done for multiple levels and on multiple representations of the dataset. On a lower level, different preprocessing methods can be defined in conjunction with several learners. On a higher level, we can then combine those predictions in order to form a very powerful model.

4.5.2.3 Multilevel Stacking

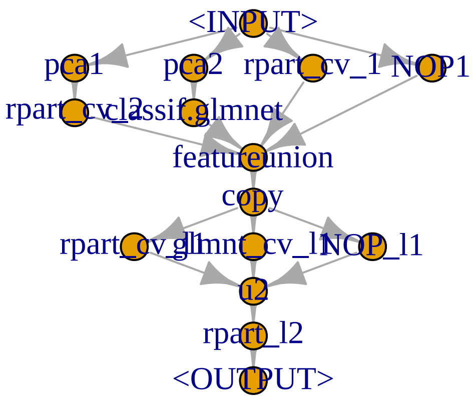
In order to showcase the power of mlr3pipelines, we will show a more complicated stacking example.

In this case, we train a `glmnet` and 2 different `rpart` models (some transform its inputs using `PipeOpPCA`) on our task in the "level 0" and concatenate them with the original features (via `gunion`). The result is then passed on to "level 1", where we copy the concatenated features 3 times and put this task into an `rpart` and a `glmnet` model. Additionally, we keep a version of the "level 0" output (via `PipeOpNOP`) and pass this on to "level 2". In "level 2" we simply concatenate all "level 1" outputs and train a final decision tree.

In the following examples, use `<lrn>$param_set$values$<param_name> = <param_value>` to set hyperparameters for the different learner.

```
library("magrittr")
library("mlr3learners") # for classif.glmnet

rprt = lrn("classif.rpart", predict_type = "prob")
glmn = lrn("classif.glmnet", predict_type = "prob")

#  Create Learner CV Operators
lrn_0 = PipeOpLearnerCV$new(rprt, id = "rpart_cv_1")
lrn_0$param_set$values$maxdepth = 5L
lrn_1 = PipeOpPCA$new(id = "pca1") %>>% PipeOpLearnerCV$new(rprt, id = "rpart_cv_2")
```

```
lrn_1$param_set$values$rpart_cv_2.maxdepth = 1L
lrn_2 = PipeOpPCA$new(id = "pca2") %>>% PipeOpLearnerCV$new(glmn)

# Union them with a PipeOpNULL to keep original features
level_0 = gunion(list(lrn_0, lrn_1,lrn_2, PipeOpNOP$new(id = "NOP1")))

# Cbind the output 3 times, train 2 learners but also keep level
# 0 predictions
level_1 = level_0 %>>%
  PipeOpFeatureUnion$new(4) %>>%
  PipeOpCopy$new(3) %>>%
  gunion(list(
    PipeOpLearnerCV$new(rprt, id = "rpart_cv_l1"),
    PipeOpLearnerCV$new(glmn, id = "glmnt_cv_l1"),
    PipeOpNOP$new(id = "NOP_l1")
  ))

# Cbind predictions, train a final learner
level_2 = level_1 %>>%
  PipeOpFeatureUnion$new(3, id = "u2") %>>%
  PipeOpLearner$new(rprt,
    id = "rpart_l2")

# Plot the resulting graph
level_2$plot(html = FALSE)
```



```
task = tsk("iris")
lrn = GraphLearner$new(level_2)
```

And we can again call `.$train` and `.$predict`:

```r
lrn$
  train(task, train.idx)$
  predict(task, test.idx)$
  score()
```

```
## classif.ce
##    0.06667
```

4.6 Special Operators

This section introduces some special operators, that might be useful in numerous further applications.

4.6.1 Imputation: `PipeOpImpute`

An often occurring setting is the imputation of missing data. Imputation methods range from relatively simple imputation using either *mean*, *median* or histograms to way more involved methods including using machine learning algorithms in order to predict missing values.

The following `PipeOp`, `PipeOpImpute`:

- imputes numeric values from a histogram
- adds a new level for factors
- adds a column marking whether a value for a given feature was missing or not

```r
pom = PipeOpMissInd$new()
pon = PipeOpImputeHist$new(id = "imputer_num", param_vals = list(affect_columns = is.numeric))
pof = PipeOpImputeNewlvl$new(id = "imputer_fct", param_vals = list(affect_columns = is.factor))
imputer = pom %>>% pon %>>% pof
```

A learner can thus be equipped with automatic imputation of missing values by adding an imputation Pipeop.

```r
polrn = PipeOpLearner$new(mlr_learners$get("classif.rpart"))
lrn = GraphLearner$new(graph = imputer %>>% polrn)
```

4.6.2 Feature Engineering: `PipeOpMutate`

New features can be added or computed from a task using `PipeOpMutate` . The operator evaluates one or multiple expressions provided in an `alist`. In this example, we compute some new features on top of the `iris` task. Then we add them to the data as illustrated below:

```r
pom = PipeOpMutate$new()

# Define a set of mutations
mutations = list(
  Sepal.Sum = ~ Sepal.Length + Sepal.Width,
  Petal.Sum = ~ Petal.Length + Petal.Width,
  Sepal.Petal.Ratio = ~ (Sepal.Length / Petal.Length)
)
pom$param_set$values$mutation = mutations
```

If outside data is required, we can make use of the env parameter. Moreover, we provide an environment, where expressions are evaluated (env defaults to .GlobalEnv).

4.6.3 Training on data subsets: PipeOpChunk

In cases, where data is too big to fit into the machine's memory, an often-used technique is to split the data into several parts. Subsequently, the parts are trained on each part of the data.

After undertaking these steps, we aggregate the models. In this example, we split our data into 4 parts using PipeOpChunk . Additionally, we create 4 PipeOpLearner POS, which are then trained on each split of the data.

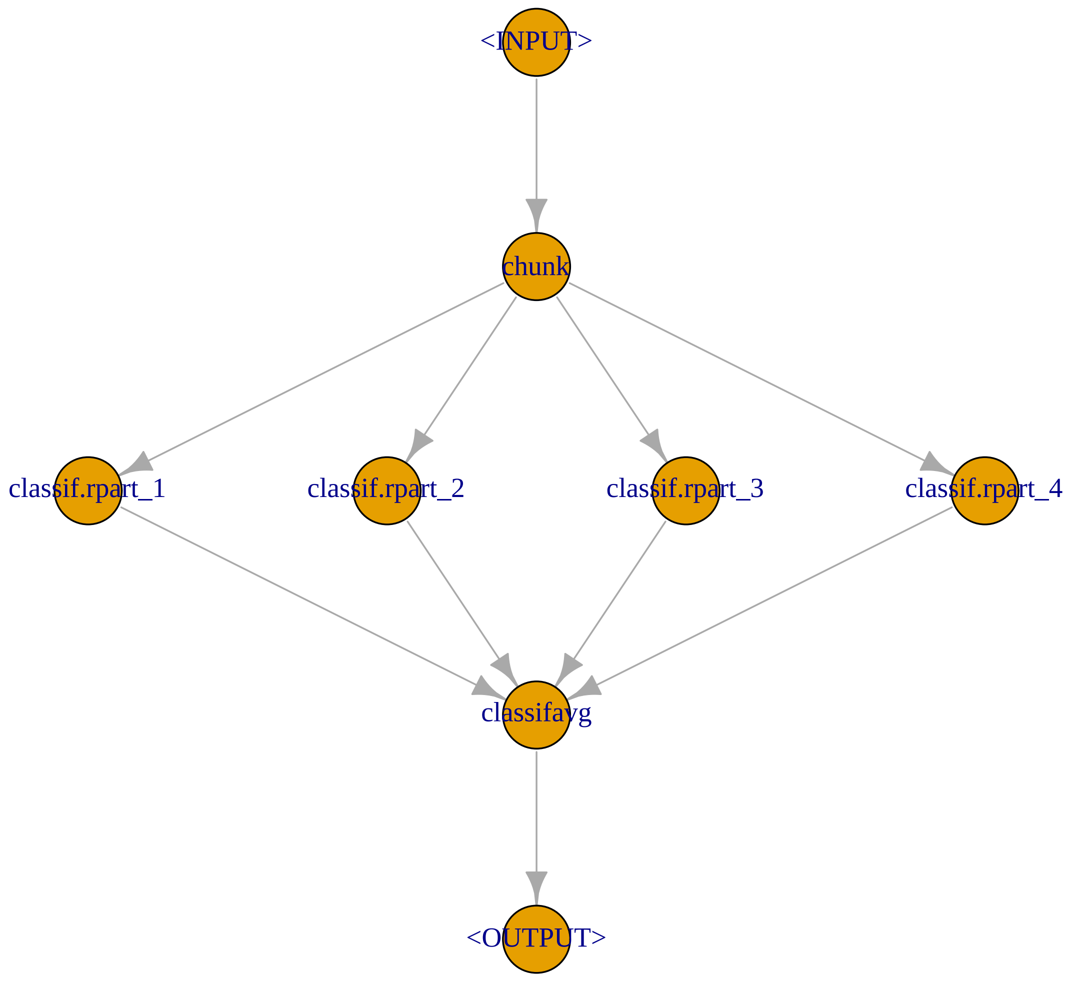
```
chks = PipeOpChunk$new(4)
lrns = greplicate(PipeOpLearner$new(mlr_learners$get("classif.rpart")), 4)
```

Afterwards we can use PipeOpClassifAvg to aggregate the predictions from the 4 different models into a new one.

```
mjv = PipeOpClassifAvg$new(4)
```

We can now connect the different operators and visualize the full graph:

```
pipeline = chks %>>% lrns %>>% mjv
pipeline$plot(html = FALSE)
```



```r
task = mlr_tasks$get("iris")
train.idx = sample(seq_len(task$nrow), 120)
test.idx = setdiff(seq_len(task$nrow), train.idx)

pipelrn = GraphLearner$new(pipeline)
pipelrn$train(task, train.idx)$
  predict(task, train.idx)$
  score()
```

```
## classif.ce
##    0.08333
```

4.6.4 Feature Selection: `PipeOpFilter` and `PipeOpSelect`

The package mlr3filters contains many different `mlr3filters::Filter`s that can be used to select features for subsequent learners. This is often required when the data has a large amount of features.

A `PipeOp` for filters is `PipeOpFilter`:

```
PipeOpFilter$new(mlr3filters::FilterInformationGain$new())
```

```
## PipeOp: <information_gain> (not trained)
## values: <list()>
## Input channels <name [train type, predict type]>:
##   input [Task,Task]
## Output channels <name [train type, predict type]>:
##   output [Task,Task]
```

How many features to keep can be set using `filter_nfeat`, `filter_frac` and `filter_cutoff`.

Filters can be selected / de-selected by name using `PipeOpSelect`.

4.7 In-depth look into mlr3pipelines

This vignette is an in-depth introduction to mlr3pipelines, the dataflow programming toolkit for machine learning in `R` using mlr3. It will go through basic concepts and then give a few examples that both show the simplicity as well as the power and versatility of using mlr3pipelines.

4.7.1 What's the Point

Machine learning toolkits often try to abstract away the processes happening inside machine learning algorithms. This makes it easy for the user to switch out one algorithm for another without having to worry about what is happening inside it, what kind of data it is able to operate on etc. The benefit of using `mlr3`, for example, is that one can create a `Learner`, a `Task`, a `Resampling` etc. and use them for typical machine learning operations. It is trivial to exchange individual components and therefore use, for example, a different `Learner` in the same experiment for comparison.

```
task = TaskClassif$new("iris", as_data_backend(iris), "Species")
lrn = mlr_learners$get("classif.rpart")
rsmp = mlr_resamplings$get("holdout")
resample(task, lrn, rsmp)
```

```
## <ResampleResult> of 1 iterations
## * Task: iris
## * Learner: classif.rpart
## * Warnings: 0 in 0 iterations
## * Errors: 0 in 0 iterations
```

However, this modularity breaks down as soon as the learning algorithm encompasses more than just model fitting, like data preprocessing, ensembles or other meta models. mlr3pipelines takes modularity one step further than `mlr3`: it makes it possible to build individual steps within a `Learner` out of building blocks called **`PipeOps`**.

4.7.2 `PipeOp`: Pipeline Operators

The most basic unit of functionality within mlr3pipelines is the **`PipeOp`**, short for "pipeline operator", which represents a trans-formative operation on input (for example a training dataset) leading to output. It can therefore be seen as a generalized notion of a function, with a certain twist: `PipeOps` behave differently during a "training phase" and a "prediction phase". The training phase will typically generate a certain model of the data that is saved as internal state. The prediction phase will then operate on the input data depending on the trained model.

An example of this behavior is the *principal component analysis* operation ("`PipeOpPCA`"): During training, it will transform incoming data by rotating it in a way that leads to uncorrelated features ordered by their contribution to total variance. It will *also* save the rotation matrix to be used during for new data. This makes it possible to perform "prediction" with single rows of new data, where a row's scores on each of the principal components (the components of the training data!) is computed.

```
po = mlr_pipeops$get("pca")
po$train(list(task))[[1]]$data()
```

```
##        Species    PC1      PC2      PC3       PC4
##   1:    setosa -2.684  0.31940 -0.02791 -0.002262
##   2:    setosa -2.714 -0.17700 -0.21046 -0.099027
##   3:    setosa -2.889 -0.14495  0.01790 -0.019968
##   4:    setosa -2.745 -0.31830  0.03156  0.075576
##   5:    setosa -2.729  0.32675  0.09008  0.061259
##  ---
## 146: virginica  1.944  0.18753  0.17783 -0.426196
## 147: virginica  1.527 -0.37532 -0.12190 -0.254367
## 148: virginica  1.764  0.07886  0.13048 -0.137001
## 149: virginica  1.901  0.11663  0.72325 -0.044595
## 150: virginica  1.390 -0.28266  0.36291  0.155039
```

```
single_line_task = task$clone()$filter(1)
po$predict(list(single_line_task))[[1]]$data()
```

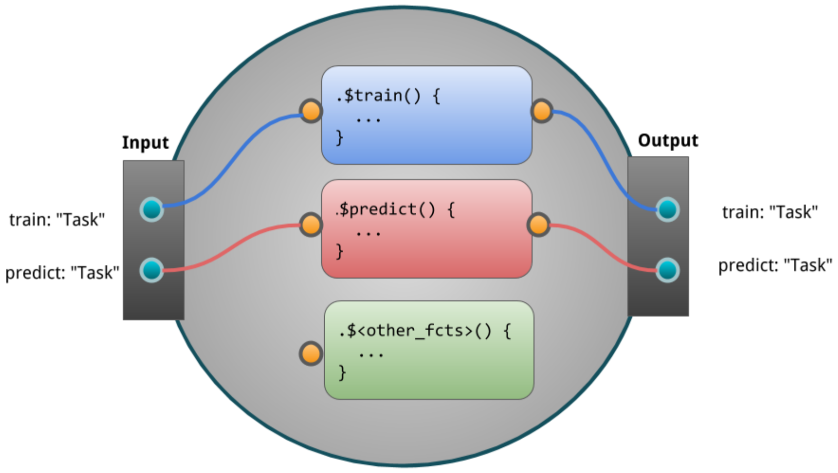
```
##    Species    PC1    PC2      PC3       PC4
## 1:  setosa -2.684 0.3194 -0.02791 -0.002262
```

```
po$state
```

```
## Standard deviations (1, .., p=4):
## [1] 2.0563 0.4926 0.2797 0.1544
##
## Rotation (n x k) = (4 x 4):
##                   PC1      PC2      PC3     PC4
## Petal.Length  0.85667 -0.17337  0.07624  0.4798
## Petal.Width   0.35829 -0.07548  0.54583 -0.7537
## Sepal.Length  0.36139  0.65659 -0.58203 -0.3155
## Sepal.Width  -0.08452  0.73016  0.59791  0.3197
```

This shows the most important primitives incorporated in a `PipeOp`: * **`$train()`**, taking a list of input arguments, turning them into a list of outputs, meanwhile saving a state in **`$state`** * **`$predict()`**, taking a list of input arguments, turning them into a list of outputs, making use of the saved `$state` * **`$state`**, the "model" trained with `$train()` and utilized during `$predict()`.

Schematically we can represent the `PipeOp` like so:



4.7.2.1 Why the `$state`

It is important to take a moment and notice the importance of a `$state` variable and the `$train()` / `$predict()` dichotomy in a `PipeOp`. There are many preprocessing methods, for example scaling of parameters or imputation, that could in theory just be applied to training data and prediction / validation data separately, or they could be applied to a task before resampling is performed. This would, however, be fallacious:

- The preprocessing of each instance of prediction data should not depend on the remaining prediction dataset. A prediction on a single instance of new data should give the same result as prediction performed on a whole dataset.
- If preprocessing is performed on a task *before* resampling is done, information about the test set can leak into the training set. Resampling should evaluate the generalization performance of the *entire* machine learning method, therefore the behavior of this entire method must only depend only on the content of the *training* split during resampling.

4.7.2.2 Where to Get `PipeOps`

Each `PipeOp` is an instance of an "`R6`" class, many of which are provided by the mlr3pipelines package itself. They can be constructed explicitly ("`PipeOpPCA$new()`") or retrieved from the `mlr_pipelines` collection: `mlr_pipeops$get("pca")`. The entire list of available `PipeOps`, and some meta-information, can be retrieved using `as.data.table()`:

```r
as.data.table(mlr_pipeops)[, c("key", "input.num", "output.num")]
```

```
##                  key input.num output.num
##  1:           boxcox         1          1
##  2:          branch         1         NA
##  3:           chunk         1         NA
##  4:  classbalancing         1          1
##  5:      classifavg        NA          1
##  6:     classweights         1          1
##  7:        colapply         1          1
##  8: collapsefactors         1          1
##  9:            copy         1         NA
## 10:          encode         1          1
## 11:    encodeimpact         1          1
## 12:      encodelmer         1          1
## 13:    featureunion        NA          1
## 14:          filter         1          1
## 15:      fixfactors         1          1
## 16:         histbin         1          1
## 17:             ica         1          1
## 18:      imputehist         1          1
## 19:      imputemean         1          1
## 20:    imputemedian         1          1
## 21:    imputenewlvl         1          1
## 22:    imputesample         1          1
## 23:       kernelpca         1          1
## 24:         learner         1          1
## 25:      learner_cv         1          1
## 26:         missind         1          1
## 27:     modelmatrix         1          1
## 28:          mutate         1          1
## 29:             nop         1          1
## 30:             pca         1          1
## 31:     quantilebin         1          1
## 32:         regravg        NA          1
## 33: removeconstants         1          1
## 34:           scale         1          1
## 35:     scalemaxabs         1          1
## 36:      scalerange         1          1
## 37:          select         1          1
## 38:           smote         1          1
## 39:     spatialsign         1          1
## 40:       subsample         1          1
## 41:        unbranch        NA          1
## 42:      yeojohnson         1          1
##                 key input.num output.num
```

When retrieving `PipeOps` from the `mlr_pipeops` dictionary, it is also possible to give additional constructor arguments, such as an id or parameter values.

```
mlr_pipeops$get("pca", param_vals = list(rank. = 3))
```

```
## PipeOp: <pca> (not trained)
## values: <rank.=3>
## Input channels <name [train type, predict type]>:
```

```
##   input [Task,Task]
## Output channels <name [train type, predict type]>:
##   output [Task,Task]
```

4.7.3 PipeOp Channels

4.7.3.1 Input Channels

Just like functions, `PipeOps` can take multiple inputs. These multiple inputs are always given as elements in the input list. For example, there is a `PipeOpFeatureUnion` that combines multiple tasks with different features and "`cbind()`s" them together, creating one combined task. When two halves of the `iris` task are given, for example, it recreates the original task:

```
iris_first_half = task$clone()$select(c("Petal.Length", "Petal.Width"))
iris_second_half = task$clone()$select(c("Sepal.Length", "Sepal.Width"))

pofu = mlr_pipeops$get("featureunion", innum = 2)

pofu$train(list(iris_first_half, iris_second_half))[[1]]$data()
```

```
##        Species Petal.Length Petal.Width Sepal.Length Sepal.Width
##   1:    setosa          1.4         0.2          5.1         3.5
##   2:    setosa          1.4         0.2          4.9         3.0
##   3:    setosa          1.3         0.2          4.7         3.2
##   4:    setosa          1.5         0.2          4.6         3.1
##   5:    setosa          1.4         0.2          5.0         3.6
##  ---
## 146: virginica          5.2         2.3          6.7         3.0
## 147: virginica          5.0         1.9          6.3         2.5
## 148: virginica          5.2         2.0          6.5         3.0
## 149: virginica          5.4         2.3          6.2         3.4
## 150: virginica          5.1         1.8          5.9         3.0
```

Because `PipeOpFeatureUnion` effectively takes two input arguments here, we can say it has two **input channels**. An input channel also carries information about the *type* of input that is acceptable. The input channels of the `pofu` object constructed above, for example, each accept a `Task` during training and prediction. This information can be queried from the `$input` slot:

```
pofu$input
```

```
##      name train predict
## 1: input1  Task    Task
## 2: input2  Task    Task
```

Other `PipeOps` may have channels that take different types during different phases. The `backuplearner` `PipeOp`, for example, takes a `NULL` and a `Task` during training, and a `Prediction` and a `Task` during prediction:

```
## TODO this is an important case to handle here, do not delete unless there is a better example.
## mlr_pipeops$get("backuplearner")$input
```

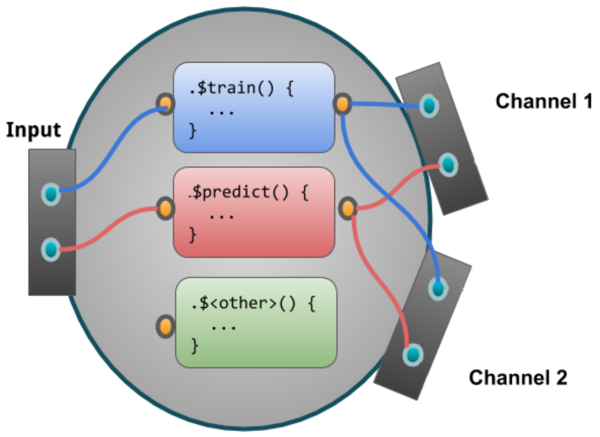
4.7.3.2 Output Channels

Unlike the typical notion of a function, `PipeOp`s can also have multiple **output channels**. `$train()` and `$predict()` always return a list, so certain `PipeOp`s may return lists with more than one element. Similar to input channels, the information about the number and type of outputs given by a `PipeOp` is available in the `$output` slot. The `chunk` PipeOp, for example, chunks a given `Task` into subsets and consequently returns multiple `Task` objects, both during training and prediction. The number of output channels must be given during construction through the `outnum` argument.

```
mlr_pipeops$get("chunk", outnum = 3)$output
```

```
##        name train predict
## 1: output1  Task    Task
## 2: output2  Task    Task
## 3: output3  Task    Task
```

Note that the number of output channels during training and prediction is the same. A schema of a `PipeOp` with two output channels:



4.7.3.3 Channel Configuration

Most `PipeOp`s have only one input channel (so they take a list with a single element), but there are a few with more than one; In many cases, the number of input or output channels is determined during construction, e.g. through the `innum` / `outnum` arguments. The `input.num` and `output.num` columns of the `mlr_pipeops`-table above show the default number of channels, and `NA` if the number depends on a construction argument.

The default printer of a `PipeOp` gives information about channel names and types:

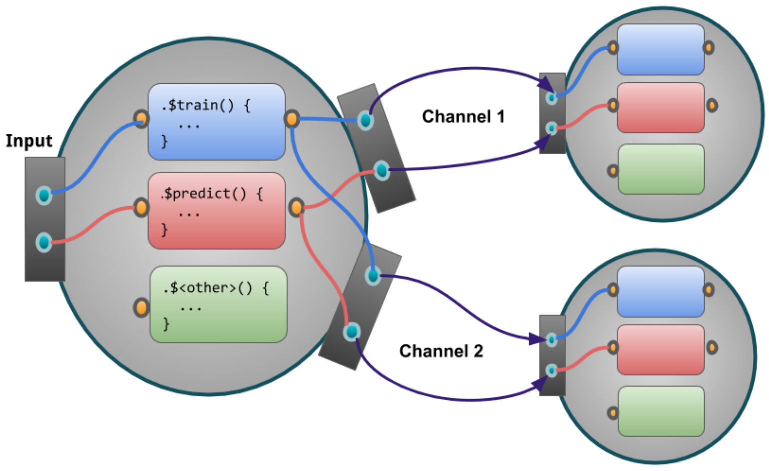
```
## mlr_pipeops$get("backuplearner")
```

4.7.4 Graph: Networks of PipeOps

4.7.4.1 Basics

What is the advantage of this tedious way of declaring input and output channels and handling in/output through lists? Because each PipeOp has a known number of input and output channels that always produce or accept data of a known type, it is possible to network them together in **Graph**s. A Graph is a collection of PipeOps with "edges" that mandate that data should be flowing along them. Edges always pass between PipeOp *channels*, so it is not only possible to explicitly prescribe which position of an input or output list an edge refers to, it makes it possible to make different components of a PipeOp's output flow to multiple different other PipeOps, as well as to have a PipeOp gather its input from multiple other PipeOps.

A schema of a simple graph of PipeOps:



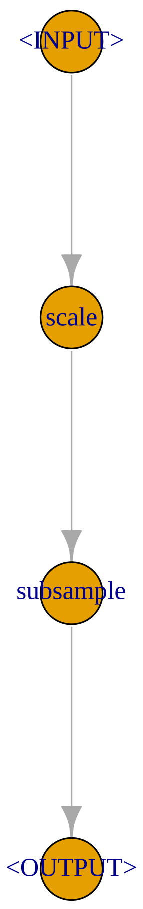
A Graph is empty when first created, and PipeOps can be added using the **$add_pipeop()** method. The **$add_edge()** method is used to create connections between them. While the printer of a Graph gives some information about its layout, the most intuitive way of visualizing it is using the $plot() function.

```r
gr = Graph$new()
gr$add_pipeop(mlr_pipeops$get("scale"))
gr$add_pipeop(mlr_pipeops$get("subsample", param_vals = list(frac = 0.1)))
gr$add_edge("scale", "subsample")
```

```r
print(gr)
```

```
## Graph with 2 PipeOps:
##          ID         State  sccssors prdcssors
##       scale <<UNTRAINED>> subsample
##   subsample <<UNTRAINED>>                scale
```

```r
gr$plot(html = FALSE)
```



A `Graph` itself has a **`$train()`** and a **`$predict()`** method that accept some data and propagate this data through the network of `PipeOp`s. The return value corresponds to the output of the `PipeOp` output channels that are not connected to other `PipeOp`s.

```
gr$train(task)[[1]]$data()
```

```
##       Species Petal.Length Petal.Width Sepal.Length Sepal.Width
##   1:     setosa     -1.33575     -1.3111     -0.89767     1.01560
##   2:     setosa     -1.33575     -1.4422     -1.25996    -0.13154
##   3:     setosa     -1.39240     -1.0487     -0.53538     1.93331
##   4:     setosa     -1.33575     -1.4422     -1.13920     1.24503
##   5: versicolor      0.64692      0.3945      1.27607     0.09789
##   6: versicolor     -0.25945     -0.2615     -1.13920    -1.50811
##   7: versicolor     -0.03286     -0.2615     -0.41462    -1.50811
##   8: versicolor      0.42033      0.3945     -0.53538    -0.13154
##  9: versicolor      0.19373      0.1321     -0.29386    -0.13154
## 10:  virginica      0.76021      0.9192     -0.05233    -0.81982
## 11:  virginica      0.87351      0.9192      0.67225    -0.81982
## 12:  virginica      0.87351      1.4440      0.67225     0.32732
## 13:  virginica      1.15675      0.5256      1.63836    -0.13154
## 14:  virginica      0.76021      0.3945      0.55149    -0.59040
## 15:  virginica      1.32669      1.4440      2.24217    -0.13154
```

```r
gr$predict(single_line_task)[[1]]$data()
```

```
##    Species Petal.Length Petal.Width Sepal.Length Sepal.Width
## 1:  setosa       -1.336      -1.311      -0.8977       1.016
```

The collection of `PipeOps` inside a `Graph` can be accessed through the **`$pipeops`** slot. The set of edges in the Graph can be inspected through the **`$edges`** slot. It is possible to modify individual `PipeOps` and edges in a Graph through these slots, but this is not recommended because no error checking is performed and it may put the `Graph` in an unsupported state.

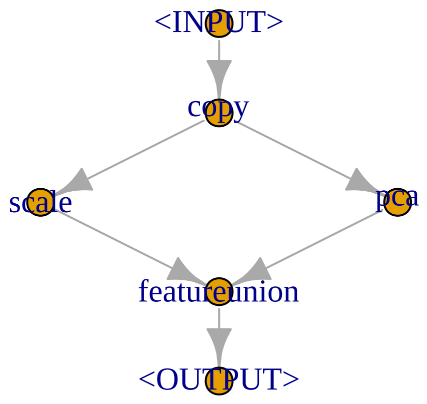
4.7.4.2 Networks

The example above showed a linear preprocessing pipeline, but it is in fact possible to build true "graphs" of operations, as long as no loops are introduced[1]. `PipeOps` with multiple output channels can feed their data to multiple different subsequent `PipeOps`, and `PipeOps` with multiple input channels can take results from different `PipeOps`. When a `PipeOp` has more than one input / output channel, then the `Graph`'s `$add_edge()` method needs an additional argument that indicates which channel to connect to. This argument can be given in the form of an integer, or as the name of the channel.

The following constructs a `Graph` that copies the input and gives one copy each to a "scale" and a "pca" `PipeOp`. The resulting columns of each operation are put next to each other by "featureunion".

```r
gr = Graph$new()$
  add_pipeop(mlr_pipeops$get("copy", outnum = 2))$
  add_pipeop(mlr_pipeops$get("scale"))$
  add_pipeop(mlr_pipeops$get("pca"))$
  add_pipeop(mlr_pipeops$get("featureunion", innum = 2))

gr$
  add_edge("copy", "scale", src_channel = 1)$        # designating channel by index
  add_edge("copy", "pca", src_channel = "output2")$  # designating channel by name
  add_edge("scale", "featureunion", dst_channel = 1)$
  add_edge("pca", "featureunion", dst_channel = 2)

gr$plot(html = FALSE)
```

[1]It is tempting to denote this as a "directed acyclic graph", but this would not be entirely correct because edges run between channels of `PipeOps`, not `PipeOps` themselves.



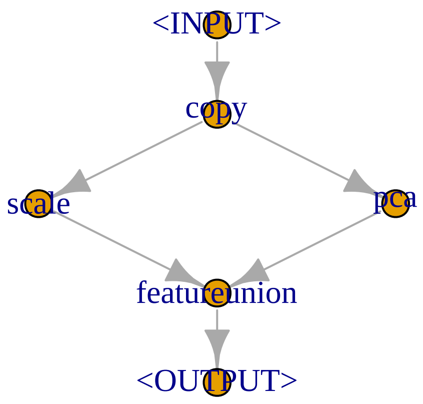
```r
gr$train(iris_first_half)[[1]]$data()
```

```
##        Species Petal.Length Petal.Width    PC1       PC2
##   1:    setosa      -1.3358     -1.3111 -2.561 -0.006922
##   2:    setosa      -1.3358     -1.3111 -2.561 -0.006922
##   3:    setosa      -1.3924     -1.3111 -2.653  0.031850
##   4:    setosa      -1.2791     -1.3111 -2.469 -0.045694
##   5:    setosa      -1.3358     -1.3111 -2.561 -0.006922
##  ---
## 146: virginica       0.8169      1.4440  1.756  0.455479
## 147: virginica       0.7036      0.9192  1.417  0.164312
## 148: virginica       0.8169      1.0504  1.640  0.178946
## 149: virginica       0.9302      1.4440  1.940  0.377936
## 150: virginica       0.7602      0.7880  1.470  0.033362
```

4.7.4.3 Syntactic Sugar

Although it is possible to create intricate `Graphs` with edges going all over the place (as long as no loops are introduced), there is usually a clear direction of flow between "layers" in the `Graph`. It is therefore convenient to build up a `Graph` from layers, which can be done using the **`%>>%`** ("double-arrow") operator. It takes either a `PipeOp` or a `Graph` on each of its sides and connects all of the outputs of its left-hand side to one of the inputs each of its right-hand side–the number of inputs therefore must match the number of outputs. Together with the **`gunion()`** operation, which takes `PipeOps` or `Graphs` and arranges them next to each other akin to a (disjoint) graph union, the above network can more easily be constructed as follows:

```r
gr = mlr_pipeops$get("copy", outnum = 2) %>>%
  gunion(list(mlr_pipeops$get("scale"), mlr_pipeops$get("pca"))) %>>%
  mlr_pipeops$get("featureunion", innum = 2)

gr$plot(html = FALSE)
```



4.7.4.4 `PipeOp` IDs and ID Name Clashes

`PipeOps` within a graph are addressed by their **`$id`**-slot. It is therefore necessary for all `PipeOps` within a `Graph` to have a unique `$id`. The `$id` can be set during or after construction, but it should not directly be changed after a `PipeOp` was inserted in a `Graph`. At that point, the **`$set_names()`**-method can be used to change `PipeOp` ids.

```
po1 = mlr_pipeops$get("scale")
po2 = mlr_pipeops$get("scale")
po1 %>>% po2  ## name clash
```

```
## Error in gunion(list(g1, g2)): Assertion on 'ids of pipe operators' failed: Must have unique na
```

```
po2$id = "scale2"
gr = po1 %>>% po2
gr
```

```
## Graph with 2 PipeOps:
##      ID         State sccssors prdcssors
##   scale <<UNTRAINED>>   scale2
##  scale2 <<UNTRAINED>>              scale
```

```
## Alternative ways of getting new ids:
mlr_pipeops$get("scale", id = "scale2")
```

```
## PipeOp: <scale2> (not trained)
## values: <list()>
## Input channels <name [train type, predict type]>:
##   input [Task,Task]
## Output channels <name [train type, predict type]>:
##   output [Task,Task]
```

```
PipeOpScale$new(id = "scale2")
```

```
## PipeOp: <scale2> (not trained)
## values: <list()>
## Input channels <name [train type, predict type]>:
##   input [Task,Task]
## Output channels <name [train type, predict type]>:
##   output [Task,Task]
```

```
## sometimes names of PipeOps within a Graph need to be changed
gr2 = mlr_pipeops$get("scale") %>>% mlr_pipeops$get("pca")
gr %>>% gr2
```

```
## Error in gunion(list(g1, g2)): Assertion on 'ids of pipe operators' failed: Must have unique na
```

```
gr2$set_names("scale", "scale3")
gr %>>% gr2
```

```
## Graph with 4 PipeOps:
##      ID         State sccssors prdcssors
##   scale <<UNTRAINED>>   scale2
##  scale2 <<UNTRAINED>>   scale3     scale
##  scale3 <<UNTRAINED>>      pca    scale2
##     pca <<UNTRAINED>>             scale3
```

4.7.5 Learners in Graphs, Graphs in Learners

The true power of mlr3pipelines derives from the fact that it can be integrated seamlessly with `mlr3`. Two components are mainly responsible for this:

- **`PipeOpLearner`**, a `PipeOp` that encapsulates a `mlr3` `Learner` and creates a `PredictionData` object in its `$predict()` phase
- **`GraphLearner`**, a `mlr3` `Learner` that can be used in place of any other `mlr3` `Learner`, but which does prediction using a `Graph` given to it

Note that these are dual to each other: One takes a `Learner` and produces a `PipeOp` (and by extension a `Graph`); the other takes a `Graph` and produces a `Learner`.

4.7.5.1 `PipeOpLearner`

The `PipeOpLearner` is constructed using a `mlr3` `Learner` and will use it to create `PredictionData` in the `$predict()` phase. The output during `$train()` is `NULL`. It can be used after a preprocessing pipeline, and it is even possible to perform operations on the `PredictionData`, for example by averaging multiple predictions or by using the "`PipeOpBackupLearner`" operator to impute predictions that a given model failed to create.

The following is a very simple `Graph` that performs training and prediction on data after performing principal component analysis.

```
gr = mlr_pipeops$get("pca") %>>% mlr_pipeops$get("learner",
  mlr_learners$get("classif.rpart"))
```

```
gr$train(task)
```

```
## $classif.rpart.output
## NULL
```

```
gr$predict(task)
```

```
## $classif.rpart.output
## <PredictionClassif> for 150 observations:
##     row_id     truth  response
##          1    setosa    setosa
##          2    setosa    setosa
##          3    setosa    setosa
## ---
##        148 virginica virginica
##        149 virginica virginica
##        150 virginica virginica
```

4.7.5.2 `GraphLearner`

Although a `Graph` has `$train()` and `$predict()` functions, it can not be used directly in places where `mlr3 Learners` can be used like resampling or benchmarks. For this, it needs to be wrapped in a `GraphLearner` object, which is a thin wrapper that enables this functionality. The resulting `Learner` is extremely versatile, because every part of it can be modified, replaced, parameterized and optimized over. Resampling the graph above can be done the same way that resampling of the `Learner` was performed in the introductory example.

```
lrngrph = GraphLearner$new(gr)
resample(task, lrngrph, rsmp)
```

```
## <ResampleResult> of 1 iterations
## * Task: iris
## * Learner: pca.classif.rpart
## * Warnings: 0 in 0 iterations
## * Errors: 0 in 0 iterations
```

4.7.6 Hyperparameters

mlr3pipelines relies on the paradox package to provide parameters that can modify each `PipeOp`'s behavior. `paradox` parameters provide information about the parameters that can be changed, as well as their types and ranges. They provide a unified interface for benchmarks and parameter optimization ("tuning"). For a deep dive into `paradox`, see the mlr3book.

The `ParamSet`, representing the space of possible parameter configurations of a `PipeOp`, can be inspected by accessing the `$param_set` slot of a `PipeOp` or a `Graph`.

```
op_pca = mlr_pipeops$get("pca")
op_pca$param_set
```

```
## ParamSet: pca
##                 id    class lower upper      levels     default value
## 1:          center ParamLgl    NA    NA  TRUE,FALSE        TRUE
## 2:          scale. ParamLgl    NA    NA  TRUE,FALSE       FALSE
## 3:           rank. ParamInt     1   Inf
## 4: affect_columns ParamUty    NA    NA              <NoDefault>
```

To set or retrieve a parameter, the **`$param_set$values`** slot can be accessed. Alternatively, the `param_vals` value can be given during construction.

```
op_pca$param_set$values$center = FALSE
op_pca$param_set$values
```

```
## $center
## [1] FALSE
```

```
op_pca = mlr_pipeops$get("pca", param_vals = list(center = TRUE))
op_pca$param_set$values
```

```
## $center
## [1] TRUE
```

Each `PipeOp` can bring its own individual parameters which are collected together in the `Graph`'s `$param_set`. A `PipeOp`'s parameter names are prefixed with its `$id` to prevent parameter name clashes.

```
gr = op_pca %>>% mlr_pipeops$get("scale")
gr$param_set
```

```
## ParamSet:
##                       id    class lower upper      levels     default value
## 1:            pca.center ParamLgl    NA    NA  TRUE,FALSE        TRUE  TRUE
## 2:            pca.scale. ParamLgl    NA    NA  TRUE,FALSE       FALSE
## 3:             pca.rank. ParamInt     1   Inf
## 4:    pca.affect_columns ParamUty    NA    NA              <NoDefault>
## 5:          scale.center ParamLgl    NA    NA  TRUE,FALSE        TRUE
## 6:           scale.scale ParamLgl    NA    NA  TRUE,FALSE        TRUE
## 7: scale.affect_columns ParamUty    NA    NA              <NoDefault>
```

```
gr$param_set$values
```

```
## $pca.center
## [1] TRUE
```

Both `PipeOpLearner` and `GraphLearner` preserve parameters of the objects they encapsulate.

```
op_rpart = mlr_pipeops$get("learner", mlr_learners$get("classif.rpart"))
op_rpart$param_set
```

```
## ParamSet: classif.rpart
##                 id    class lower upper levels     default value
## 1:        minsplit ParamInt     1   Inf                 20
## 2:       minbucket ParamInt     1   Inf        <NoDefault>
## 3:             cp ParamDbl     0     1              0.01
## 4:     maxcompete ParamInt     0   Inf                 4
## 5:   maxsurrogate ParamInt     0   Inf                 5
## 6:       maxdepth ParamInt     1    30                30
## 7:   usesurrogate ParamInt     0     2                 2
## 8: surrogatestyle ParamInt     0     1                 0
## 9:           xval ParamInt     0   Inf                10     0
```

```
glrn = GraphLearner$new(gr %>>% op_rpart)
glrn$param_set
```

```
## ParamSet:
##                              id    class lower upper      levels     default
##  1:                  pca.center ParamLgl    NA    NA  TRUE,FALSE        TRUE
##  2:                  pca.scale. ParamLgl    NA    NA  TRUE,FALSE       FALSE
##  3:                   pca.rank. ParamInt     1   Inf
##  4:          pca.affect_columns ParamUty    NA    NA             <NoDefault>
##  5:                scale.center ParamLgl    NA    NA  TRUE,FALSE        TRUE
##  6:                 scale.scale ParamLgl    NA    NA  TRUE,FALSE        TRUE
##  7:        scale.affect_columns ParamUty    NA    NA             <NoDefault>
##  8:       classif.rpart.minsplit ParamInt     1   Inf                     20
##  9:      classif.rpart.minbucket ParamInt     1   Inf             <NoDefault>
## 10:             classif.rpart.cp ParamDbl     0     1                   0.01
## 11:     classif.rpart.maxcompete ParamInt     0   Inf                      4
## 12:   classif.rpart.maxsurrogate ParamInt     0   Inf                      5
## 13:       classif.rpart.maxdepth ParamInt     1    30                     30
## 14:   classif.rpart.usesurrogate ParamInt     0     2                      2
## 15: classif.rpart.surrogatestyle ParamInt     0     1                      0
## 16:           classif.rpart.xval ParamInt     0   Inf                     10
##     value
##  1:  TRUE
##  2:
##  3:
##  4:
##  5:
##  6:
##  7:
##  8:
##  9:
## 10:
## 11:
## 12:
## 13:
## 14:
## 15:
## 16:     0
```

5 Technical

This chapter provides an overview of technical details of the mlr3 framework.

Parallelization

At first, some details about Parallelization and the usage of the future are given. Parallelization refers to the process of running multiple jobs simultaneously. This process is employed to minimize the necessary computing power. Algorithms consist of both sequential (non-parallelizable) and parallelizable parts. Therefore, parallelization does not always alter performance in a positive substantial manner. Summed up, this sub-chapter illustrates how and when to use parallelization in mlr3.

Database Backends

The section Database Backends describes how to work with database backends that mlr3 supports. Database backends can be helpful for large data processing which does not fit in memory or is stored natively in a database (e.g. SQLite). Specifically when working with large data sets, or when undertaking numerous tasks simultaneously, it can be advantageous to interface out-of-memory data. The section provides an illustration of how to implement Database Backends using of NYC flight data.

Parameters

In the section Parameters instructions are given on how to:

- define parameter sets for learners
- undertake parameter sampling
- apply parameter transformations

For illustrative purposes, this sub-chapter uses the paradox package, the successor of ParamHelpers.

Logging and Verbosity

The sub-chapter on Logging and Verbosity shows how to change the most important settings related to logging. In mlr3 we use the lgr package.

Transition Guide

Lastly, we provide a Transition Guide for users of the old mlr who want to switch to mlr3.

5.1 Parallelization

Parallelization refers to the process of running multiple jobs in parallel, simultaneously. This process allows for significant savings in computing power.

mlr3 uses the future backends for parallelization. Make sure you have installed the required packages future and future.apply:

mlr3 is capable of parallelizing a variety of different scenarios. One of the most used cases is to parallelize the `Resampling` iterations. See Section Resampling for a detailed introduction to resampling.

In the following section, we will use the *spam* task and a simple classification tree (`"classif.rpart"`) to showcase parallelization. We use the future package to parallelize the resampling by selecting a backend via the function `future::plan()`. We use the `future::multiprocess` backend here which uses forks (c.f. `parallel::mcparallel()`) on UNIX based systems and a `socket cluster` on Windows or if running in RStudio:

```
future::plan("multiprocess")

task = tsk("spam")
learner = lrn("classif.rpart")
resampling = rsmp("subsampling")

time = Sys.time()
resample(task, learner, resampling)
Sys.time() - time
```

By default all CPUs of your machine are used unless you specify argument `workers` in `future::plan()`.

On most systems you should see a decrease in the reported elapsed time. On some systems (e.g. Windows), the overhead for parallelization is quite large though. Therefore, it is advised to only enable parallelization for resamplings where each iteration runs at least 10 seconds.

Choosing the parallelization level

If you are transitioning from mlr, you might be used to selecting different parallelization levels, e.g. for resampling, benchmarking or tuning. In mlr3 this is no longer required. All kind of events are rolled out on the same level. Therefore, there is no need to decide whether you want to parallelize the tuning OR the resampling.

In mlr3 this is no longer required. All kind of events are rolled out on the same level - there is no need to decide whether you want to parallelize the tuning OR the resampling.

Just lean back and let the machine do the work :-)

5.2 Error Handling

To demonstrate how to properly deal with misbehaving learners, mlr3 ships with the learner `classif.debug`:

```
task = tsk("iris")
learner = lrn("classif.debug")
print(learner)
```

```
## <LearnerClassifDebug:classif.debug>
## * Model: -
## * Parameters: list()
## * Packages: -
## * Predict Type: response
## * Feature types: logical, integer, numeric, character, factor, ordered
## * Properties: missings, multiclass, twoclass
```

This learner comes with special hyperparameters that let us control

1. what conditions should be signaled (message, warning, error, segfault) with what probability
2. during which stage the conditions should be signaled (train or predict)
3. the ratio of predictions being `NA` (`predict_missing`)

```
learner$param_set
```

```
## ParamSet:
##                    id    class lower upper      levels     default value
##  1:     message_train ParamDbl     0     1                         0
##  2:   message_predict ParamDbl     0     1                         0
##  3:     warning_train ParamDbl     0     1                         0
##  4:   warning_predict ParamDbl     0     1                         0
##  5:       error_train ParamDbl     0     1                         0
##  6:     error_predict ParamDbl     0     1                         0
##  7:    segfault_train ParamDbl     0     1                         0
##  8:  segfault_predict ParamDbl     0     1                         0
##  9:   predict_missing ParamDbl     0     1                         0
## 10:        save_tasks ParamLgl    NA    NA  TRUE,FALSE       FALSE
## 11:                 x ParamDbl     0     1             <NoDefault>
```

With the learner's default settings, the learner will do nothing special: The learner learns a random label and creates constant predictions.

```
task = tsk("iris")
learner$train(task)$predict(task)$confusion
```

```
##             truth
## response     setosa versicolor virginica
##   setosa          0          0         0
##   versicolor      0          0         0
##   virginica      50         50        50
```

We now set a hyperparameter to let the debug learner signal an error during the train step. By default,mlr3 does not catch conditions such as warnings or errors raised by third-party code like learners:

```
learner$param_set$values = list(error_train = 1)
learner$train(tsk("iris"))
```

```
## Error in learner$.__enclos_env__$private$.train(task): Error from classif.debug->train()
```

If this would be a regular learner, we could now start debugging with `traceback()` (or create a MRE to file a bug report).

However, machine learning algorithms raising errors is not uncommon as algorithms typically cannot process all possible data. Thus, we need a mechanism to

1. capture all signaled conditions such as messages, warnings and errors so that we can analyze them post-hoc, and
2. a statistically sound way to proceed the calculation and be able to aggregate over partial results.

These two mechanisms are explained in the following subsections.

5.2.1 Encapsulation

With encapsulation, exceptions do not stop the program flow and all output is logged to the learner (instead of printed to the console). Each `Learner` has a field `encapsulate` to control how the train or predict steps are executed. One way to encapsulate the execution is provided by the package evaluate (see `encapsulate()` for more details):

```
task = tsk("iris")
learner = lrn("classif.debug")
learner$param_set$values = list(warning_train = 1, error_train = 1)
learner$encapsulate = c(train = "evaluate", predict = "evaluate")

learner$train(task)
```

After training the learner, one can access the recorded log via the fields `log`, `warnings` and `errors`:

```
learner$log
```

```
##    stage   class                                msg
## 1: train warning Warning from classif.debug->train()
## 2: train   error   Error from classif.debug->train()
```

```
learner$warnings
```

```
## [1] "Warning from classif.debug->train()"
```

```
learner$errors
```

```
## [1] "Error from classif.debug->train()"
```

Another method for encapsulation is implemented in the callr package. callr spawns a new R process to execute the respective step, and thus even guards the current session from segfaults. On the downside, starting new processes comes with a computational overhead.

```
learner$encapsulate = c(train = "callr", predict = "callr")
learner$param_set$values = list(segfault_train = 1)
learner$train(task = task)
learner$errors
```

```
## [1] "callr process exited with status -11"
```

Without a model, it is not possible to get predictions though:

```
learner$predict(task)
```

```
## Error: Cannot predict, Learner 'classif.debug' has not been trained yet
```

To handle the missing predictions in a graceful way during `resample()` or `benchmark()`, fallback learners are introduced next.

5.2.2 Fallback learners

Fallback learners have the purpose to allow scoring results in cases where a `Learner` is misbehaving in some sense. Some typical examples include:

- The learner fails to fit a model during training, e.g., if some convergence criterion is not met or the learner ran out of memory.
- The learner fails to predict for some or all observations. A typical case is e.g. new factor levels in the test data.

We first handle the most common case that a learner completely breaks while fitting a model or while predicting on new data. If the learner fails in either of these two steps, we rely on a second learner to generate predictions: the fallback learner.

In the next example, in addition to the debug learner, we attach a simple featureless learner to the debug learner. So whenever the debug learner fails (which is every time with the given parametrization) and encapsulation in enabled, `mlr3` falls back to the predictions of the featureless learner internally:

```r
task = tsk("iris")
learner = lrn("classif.debug")
learner$param_set$values = list(error_train = 1)
learner$encapsulate = c(train = "evaluate")
learner$fallback = lrn("classif.featureless")
learner$train(task)
learner
```

```
## <LearnerClassifDebug:classif.debug>
## * Model: -
## * Parameters: error_train=1
## * Packages: -
## * Predict Type: response
## * Feature types: logical, integer, numeric, character, factor, ordered
## * Properties: missings, multiclass, twoclass
## * Errors: Error from classif.debug->train()
```

Note that the log contains the captured error (which is also included in the print output), and although we don't have a model, we can still get predictions:

```r
learner$model
```

```
## NULL
```

```r
prediction = learner$predict(task)
prediction$score()
```

```
## classif.ce
##     0.6667
```

While the fallback learner is of limited use for this stepwise train-predict procedure, it is invaluable for larger benchmark studies where only few resampling iterations are failing. Here, we need to replace the missing scores with a number in order to aggregate over all resampling iterations. And imputing a number which is equivalent to guessing labels often seems to be the right amount of penalization.

In the following snippet we compare the previously created debug learner with a simple classification tree. We re-parametrize the debug learner to fail in roughly 30% of the resampling iterations during the training step:

```r
learner$param_set$values = list(error_train = 0.3)
```

```r
bmr = benchmark(benchmark_grid(tsk("iris"), list(learner, lrn("classif.rpart")), rsmp("cv")))
aggr = bmr$aggregate(conditions = TRUE)
aggr
```

```
##    nr  resample_result task_id    learner_id resampling_id iters warnings
## 1:  1 <ResampleResult>    iris classif.debug            cv    10        0
## 2:  2 <ResampleResult>    iris classif.rpart            cv    10        0
##    errors classif.ce
## 1:      5    0.68667
## 2:      0    0.06667
```

To further investigate the errors, we can extract the ResampleResult:

```r
rr = aggr[learner_id == "classif.debug"]$resample_result[[1L]]
rr$errors
```

```
##    iteration                              msg
## 1:         4 Error from classif.debug->train()
## 2:         6 Error from classif.debug->train()
## 3:         7 Error from classif.debug->train()
## 4:         9 Error from classif.debug->train()
## 5:        10 Error from classif.debug->train()
```

A similar yet different problem emerges when a learner predicts only a subset of the observations in the test set (and predicts `NA` for others). Handling such predictions in a statistically sound way is not straight-forward and a common source for over-optimism when reporting results. Imagine that our goal is to benchmark two algorithms using a 10-fold cross validation on some binary classification task:

- Algorithm A is a ordinary logistic regression.
- Algorithm B is also a ordinary logistic regression, but with a twist: If the logistic regression is rather certain about the predicted label ($>$ 90% probability), it returns the label and a missing value otherwise.

When comparing the performance of these two algorithms, it is obviously not fair to average over all predictions of algorithm A while only average over the "easy-to-predict" observations for algorithm B. By doing so, algorithm B would easily outperform algorithm A, but you have not factored in that you can not generate predictions for many observations. On the other hand, it is also not feasible to exclude all observations from the test set of a benchmark study where at least one algorithm failed to predict a label. Instead, we proceed by imputing all missing predictions with something naive, e.g., by predicting the majority class with a featureless learner. And as the majority class may depend on the resampling split (or we opt for some other arbitrary baseline learner), it is best to just train a second learner on the same resampling split.

Long story short, if a fallback learner is involved, missing predictions of the base learner will be automatically replaced with predictions from the fallback learner. This is illustrated in the following example:

```r
task = tsk("iris")
learner = lrn("classif.debug")

# this hyperparameter sets the ratio of missing predictions
learner$param_set$values = list(predict_missing = 0.5)
```

```
# without fallback
p = learner$train(task)$predict(task)
table(p$response, useNA = "always")
```

```
##
##     setosa versicolor  virginica       <NA>
##         75          0          0         75
```

```
# with fallback
learner$fallback = lrn("classif.featureless")
p = learner$train(task)$predict(task)
table(p$response, useNA = "always")
```

```
##
##     setosa versicolor  virginica       <NA>
##          0         75         75          0
```

Summed up, by combining encapsulation and fallback learners, it is possible to benchmark even quite unreliable or instable learning algorithms in a convenient way.

5.3 Database Backends

In mlr3, `Task`s store their data in an abstract data format, the `DataBackend`. The default backend uses data.table via the `DataBackendDataTable` as an in-memory data base.

For larger data, or when working with many tasks in parallel, it can be advantageous to interface an out-of-memory data. We use the excellent R package dbplyr which extends dplyr to work on many popular data bases like MariaDB, PostgreSQL or SQLite.

5.3.1 Use Case: NYC Flights

To generate a halfway realistic scenario, we use the NYC flights data set from package nycflights13:

```
# load data
requireNamespace("DBI")
```

```
## Loading required namespace: DBI
```

```
requireNamespace("RSQLite")
```

```
## Loading required namespace: RSQLite
```

```
requireNamespace("nycflights13")
```

```
## Loading required namespace: nycflights13
```

```
data("flights", package = "nycflights13")
str(flights)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    336776 obs. of  19 variables:
##  $ year          : int  2013 2013 2013 2013 2013 2013 2013 2013 2013 2013 ...
##  $ month         : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ day           : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ dep_time      : int  517 533 542 544 554 554 555 557 557 558 ...
##  $ sched_dep_time: int  515 529 540 545 600 558 600 600 600 600 ...
##  $ dep_delay     : num  2 4 2 -1 -6 -4 -5 -3 -3 -2 ...
##  $ arr_time      : int  830 850 923 1004 812 740 913 709 838 753 ...
##  $ sched_arr_time: int  819 830 850 1022 837 728 854 723 846 745 ...
##  $ arr_delay     : num  11 20 33 -18 -25 12 19 -14 -8 8 ...
##  $ carrier       : chr  "UA" "UA" "AA" "B6" ...
##  $ flight        : int  1545 1714 1141 725 461 1696 507 5708 79 301 ...
##  $ tailnum       : chr  "N14228" "N24211" "N619AA" "N804JB" ...
##  $ origin        : chr  "EWR" "LGA" "JFK" "JFK" ...
##  $ dest          : chr  "IAH" "IAH" "MIA" "BQN" ...
##  $ air_time      : num  227 227 160 183 116 150 158 53 140 138 ...
##  $ distance      : num  1400 1416 1089 1576 762 ...
##  $ hour          : num  5 5 5 5 6 5 6 6 6 6 ...
##  $ minute        : num  15 29 40 45 0 58 0 0 0 0 ...
##  $ time_hour    : POSIXct, format: "2013-01-01 05:00:00" "2013-01-01 05:00:00" ...
```

```r
# add column of unique row ids
flights$row_id = 1:nrow(flights)

# create sqlite database in temporary file
path = tempfile("flights", fileext = ".sqlite")
con = DBI::dbConnect(RSQLite::SQLite(), path)
tbl = DBI::dbWriteTable(con, "flights", as.data.frame(flights))
DBI::dbDisconnect(con)

# remove in-memory data
rm(flights)
```

5.3.2 Preprocessing with `dplyr`

With the SQLite database in `path`, we now re-establish a connection and switch to dplyr/dbplyr for some essential preprocessing.

```r
# establish connection
con = DBI::dbConnect(RSQLite::SQLite(), path)

# select the "flights" table, enter dplyr
library("dplyr")
```

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```r
library("dbplyr")
```

```
##
```

```
## Attaching package: 'dbplyr'
## The following objects are masked from 'package:dplyr':
##
##     ident, sql
```

```
tbl = tbl(con, "flights")
```

First, we select a subset of columns to work on:

```
keep = c("row_id", "year", "month", "day", "hour", "minute", "dep_time",
  "arr_time", "carrier", "flight", "air_time", "distance", "arr_delay")
tbl = select(tbl, keep)
```

Additionally, we remove those observations where the arrival delay (`arr_delay`) has a missing value:

```
tbl = filter(tbl, !is.na(arr_delay))
```

To keep runtime reasonable for this toy example, we filter the data to only use every second row:

```
tbl = filter(tbl, row_id %% 2 == 0)
```

The factor levels of the feature `carrier` are merged so that infrequent carriers are replaced by level "other":

```
tbl = mutate(tbl, carrier = case_when(
    carrier %in% c("OO", "HA", "YV", "F9", "AS", "FL", "VX", "WN") ~ "other",
    TRUE ~ carrier)
)
```

5.3.3 DataBackendDplyr

The processed table is now used to create a mlr3db::DataBackendDplyr from mlr3db:

```
library("mlr3db")
b = as_data_backend(tbl, primary_key = "row_id")
```

We can now use the interface of DataBackend to query some basic information of the data:

```
b$nrow
```

```
## [1] 163707
```

```
b$ncol
```

```
## [1] 13
```

```
b$head()
```

```
##    row_id year month day hour minute dep_time arr_time carrier flight air_time
## 1:      2 2013     1   1    5     29      533      850      UA   1714      227
## 2:      4 2013     1   1    5     45      544     1004      B6    725      183
## 3:      6 2013     1   1    5     58      554      740      UA   1696      150
## 4:      8 2013     1   1    6      0      557      709      EV   5708       53
## 5:     10 2013     1   1    6      0      558      753      AA    301      138
## 6:     12 2013     1   1    6      0      558      853      B6     71      158
##    distance arr_delay
## 1:     1416        20
## 2:     1576       -18
## 3:      719        12
## 4:      229       -14
## 5:      733         8
## 6:     1005        -3
```

Note that the DataBackendDplyr does not know about any rows or columns we have filtered out with dplyr before, it just operates on the view we provided.

5.3.4 Model fitting

We create the following mlr3 objects:

- A regression task, based on the previously created mlr3db::DataBackendDplyr.
- A regression learner (regr.rpart).
- A resampling strategy: 3 times repeated subsampling using 2% of the observations for training ("subsampling")
- Measures "mse", "time_predict" and "time_predict"

```
task = TaskRegr$new("flights_sqlite", b, target = "arr_delay")
learner = lrn("regr.rpart")
measures = mlr_measures$mget(c("regr.mse", "time_train", "time_predict"))
resampling = rsmp("subsampling")
resampling$param_set$values = list(repeats = 3, ratio = 0.02)
```

We pass all these objects to resample() to perform a simple resampling with three iterations. In each iteration, only the required subset of the data is queried from the SQLite data base and passed to rpart::rpart():

```
rr = resample(task, learner, resampling)
print(rr)
```

```
## <ResampleResult> of 3 iterations
## * Task: flights_sqlite
## * Learner: regr.rpart
## * Warnings: 0 in 0 iterations
## * Errors: 0 in 0 iterations
```

```
rr$aggregate(measures)
```

```
##     regr.mse   time_train time_predict
##     1242.253        0.175        1.562
```

5.3.5 Cleanup

Finally, we remove the `tbl` object and close the connection.

```
rm(tbl)
DBI::dbDisconnect(con)
```

5.4 Parameters (using paradox)

The paradox package offers a language for the description of *parameter spaces*, as well as tools for useful operations on these parameter spaces. A parameter space is often useful when describing:

- A set of sensible input values for an R function
- The set of possible values that slots of a configuration object can take
- The search space of an optimization process

The tools provided by `paradox` therefore relate to:

- **Parameter checking**: Verifying that a set of parameters satisfies the conditions of a parameter space
- **Parameter sampling**: Generating parameter values that lie in the parameter space for systematic exploration of program behavior depending on these parameters

paradox is, by nature, an auxiliary package that derives its usefulness from other packages that make use of it. It is heavily utilized in other mlr-org packages such as mlr3, mlr3pipelines, and mlr3tuning.

5.4.1 Reference Based Objects

paradox is the spiritual successor to the ParamHelpers package and was written from scratch using the R6 class system. The most important consequence of this is that all objects created in `paradox` are "reference-based", unlike most other objects in R. When a change is made to a `ParamSet` object, for example by adding a parameter using the `$add()` function, all variables that point to this `ParamSet` will contain the changed object. To create an independent copy of a `ParamSet`, the `$clone()` method needs to be used:

```
library("paradox")

ps = ParamSet$new()
ps2 = ps
ps3 = ps$clone(deep = TRUE)
print(ps) # the same for ps2 and ps3
```

```
## ParamSet:
## Empty.
```

```
ps$add(ParamLgl$new("a"))
```

```
print(ps)  # ps was changed
```

```
## ParamSet:
##    id    class lower upper      levels     default value
## 1:  a ParamLgl    NA    NA  TRUE,FALSE <NoDefault>
```

```
print(ps2) # contains the same reference as ps
```

```
## ParamSet:
##    id    class lower upper      levels     default value
## 1:  a ParamLgl    NA    NA  TRUE,FALSE <NoDefault>
```

```r
print(ps3) # is a "clone" of the old (empty) ps
```

```
## ParamSet:
## Empty.
```

5.4.2 Defining a Parameter Space

5.4.2.1 Single Parameters

The basic building block for describing parameter spaces is the `Param` class. It represents a single parameter, which usually can take a single atomic value. Consider, for example, trying to configure the `rpart` package's `rpart.control` object. It has various components (`minsplit`, `cp`, …) that all take a single value, and that would all be represented by a different instance of a `Param` object.

The `Param` class has various sub-classes that represent different value types:

- `ParamInt`: Integer numbers
- `ParamDbl`: Real numbers
- `ParamFct`: String values from a set of possible values, similar to R `factors`
- `ParamLgl`: Truth values (`TRUE` / `FALSE`), as `logicals` in R
- `ParamUty`: Parameter that can take any value

A particular instance of a parameter is created by calling the attached `$new()` function:

```r
library("paradox")
parA = ParamLgl$new(id = "A")
parB = ParamInt$new(id = "B", lower = 0, upper = 10, tags = c("tag1", "tag2"))
parC = ParamDbl$new(id = "C", lower = 0, upper = 4, special_vals = list(NULL))
parD = ParamFct$new(id = "D", levels = c("x", "y", "z"), default = "y")
parE = ParamUty$new(id = "E", custom_check = function(x) checkmate::checkFunction(x))
```

Every parameter must have:

- **id** - A name for the parameter within the parameter set
- **default** - A default value
- **special_vals** - A list of values that are accepted even if they do not conform to the type
- **tags** - Tags that can be used to organize parameters

The numeric (`Int` and `Dbl`) parameters furthermore allow for specification of a **lower** and **upper** bound. Meanwhile, the `Fct` parameter must be given a vector of **levels** that define the possible states its parameter can take. The `Uty` parameter can also have a **`custom_check`** function that must return `TRUE` when a value is acceptable and may return a `character(1)` error description otherwise. The example above defines `parE` as a parameter that only accepts functions.

All values which are given to the constructor are then accessible from the object for inspection using `$`. Although all these values can be changed for a parameter after construction, this can be a bad idea and should be avoided when possible.

Instead, a new parameter should be constructed. Besides the possible values that can be given to a constructor, there are also the `$class`, `$nlevels`, `$is_bounded`, `$has_default`, `$storage_type`, `$is_number` and `$is_categ` slots that give information about a parameter.

A list of all slots can be found in `?Param`.

```
parB$lower
```

```
## [1] 0
```

```
parA$levels
```

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

```
parE$class
```

```
## [1] "ParamUty"
```

It is also possible to get all information of a `Param` as `data.table` by calling `as.data.table`.

```
as.data.table(parA)
```

```
##    id    class lower upper     levels nlevels is_bounded special_vals    default storage_type tag
## 1:  A ParamLgl    NA    NA TRUE,FALSE       2       TRUE       <list> <NoDefault>      logical
```

Type / Range Checking

A `Param` object offers the possibility to check whether a value satisfies its condition, i.e. is of the right type, and also falls within the range of allowed values, using the `$test()`, `$check()`, and `$assert()` functions. `test()` should be used within conditional checks and returns `TRUE` or `FALSE`, while `check()` returns an error description when a value does not conform to the parameter (and thus plays well with the `checkmate::assert()` function). `assert()` will throw an error whenever a value does not fit.

```
parA$test(FALSE)
```

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

```
parA$test("FALSE")
```

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

```
parA$check("FALSE")
```

```
## [1] "Must be of type 'logical flag', not 'character'"
```

Instead of testing single parameters, it is often more convenient to check a whole set of parameters using a `ParamSet`.

5.4.2.2 Parameter Sets

The ordered collection of parameters is handled in a `ParamSet`[1]. It is initialized using the `$new()` function and optionally takes a list of `Params` as argument. Parameters can also be added to the constructed `ParamSet` using the `$add()` function. It is even possible to add whole `ParamSets` to other `ParamSets`.

```
ps = ParamSet$new(list(parA, parB))
ps$add(parC)
```

[1]Although the name is suggestive of a “Set”-valued `Param`, this is unrelated to the other objects that follow the `ParamXxx` naming scheme.

```
ps$add(ParamSet$new(list(parD, parE)))
print(ps)
```

```
## ParamSet:
##    id    class lower upper      levels     default value
## 1:  A ParamLgl    NA    NA  TRUE,FALSE <NoDefault>
## 2:  B ParamInt     0    10             <NoDefault>
## 3:  C ParamDbl     0     4             <NoDefault>
## 4:  D ParamFct    NA    NA       x,y,z           y
## 5:  E ParamUty    NA    NA             <NoDefault>
```

The individual parameters can be accessed through the `$params` slot. It is also possible to get information about all parameters in a vectorized fashion using mostly the same slots as for individual `Params` (i.e. `$class`, `$levels` etc.), see `?ParamSet` for details.

It is possible to reduce `ParamSets` using the **`$subset`** method. Be aware that it modifies a ParamSet in-place, so a "clone" must be created first if the original `ParamSet` should not be modified.

```
psSmall = ps$clone()
psSmall$subset(c("A", "B", "C"))
print(psSmall)
```

```
## ParamSet:
##    id    class lower upper      levels     default value
## 1:  A ParamLgl    NA    NA  TRUE,FALSE <NoDefault>
## 2:  B ParamInt     0    10             <NoDefault>
## 3:  C ParamDbl     0     4             <NoDefault>
```

Just as for `Params`, and much more useful, it is possible to get the `ParamSet` as a `data.table` using `as.data.table()`. This makes it easy to subset parameters on certain conditions and aggregate information about them, using the variety of methods provided by `data.table`.

```
as.data.table(ps)
```

```
##    id    class lower upper     levels nlevels is_bounded special_vals     default storage_type
## 1:  A ParamLgl    NA    NA TRUE,FALSE       2       TRUE       <list> <NoDefault>      logical
## 2:  B ParamInt     0    10                 11       TRUE       <list> <NoDefault>      integer tag1,tag2
## 3:  C ParamDbl     0     4                Inf       TRUE       <list> <NoDefault>      numeric
## 4:  D ParamFct    NA    NA      x,y,z       3       TRUE       <list>           y    character
## 5:  E ParamUty    NA    NA                Inf      FALSE       <list> <NoDefault>         list
```

Type / Range Checking

Similar to individual `Params`, the `ParamSet` provides `$test()`, `$check()` and `$assert()` functions that allow for type and range checking of parameters. Their argument must be a named list with values that are checked against the respective parameters. It is possible to check only a subset of parameters.

```
ps$check(list(A = TRUE, B = 0, E = identity))
```

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

```
ps$check(list(A = 1))
```

```
## [1] "A: Must be of type 'logical flag', not 'double'"
```

```
ps$check(list(Z = 1))
```

```
## [1] "Parameter 'Z' not available. Did you mean 'A' / 'B' / 'C'?"
```

Values in a ParamSet

Although a `ParamSet` fundamentally represents a value space, it also has a slot `$values` that can contain a point within that space. This is useful because many things that define a parameter space need similar operations (like parameter checking) that can be simplified. The `$values` slot contains a named list that is always checked against parameter constraints. When trying to set parameter values, e.g. for `mlr3` `Learners`, it is the `$values` slot of its `$param_set` that needs to be used.

```
ps$values = list(A = TRUE, B = 0)
ps$values$B = 1
print(ps$values)
```

```
## $A
## [1] TRUE
##
## $B
## [1] 1
```

The parameter constraints are automatically checked:

```
ps$values$B = 100
```

```
## Error in (function (xs) : Assertion on 'xs' failed: B: Element 1 is not <= 10.
```

Dependencies

It is often the case that certain parameters are irrelevant or should not be given depending on values of other parameters. An example would be a parameter that switches a certain algorithm feature (for example regularization) on or off, combined with another parameter that controls the behavior of that feature (e.g. a regularization parameter). The second parameter would be said to *depend* on the first parameter having the value `TRUE`.

A dependency can be added using the `$add_dep` method, which takes both the ids of the "depender" and "dependee" parameters as well as a `Condition` object. The `Condition` object represents the check to be performed on the "dependee". Currently it can be created using `CondEqual$new()` and `CondAnyOf$new()`. Multiple dependencies can be added, and parameters that depend on others can again be depended on, as long as no cyclic dependencies are introduced.

The consequence of dependencies are twofold: For one, the `$check()`, `$test()` and `$assert()` tests will not accept the presence of a parameter if its dependency is not met. Furthermore, when sampling or creating grid designs from a `ParamSet`, the dependencies will be respected (see Parameter Sampling, in particular Hierarchical Sampler).

The following example makes parameter `D` depend on parameter `A` being `FALSE`, and parameter `B` depend on parameter `D` being one of `"x"` or `"y"`. This introduces an implicit dependency of `B` on `A` being `FALSE` as well, because `D` does not take any value if `A` is `TRUE`.

```
ps$add_dep("D", "A", CondEqual$new(FALSE))
ps$add_dep("B", "D", CondAnyOf$new(c("x", "y")))
```

```
ps$check(list(A = FALSE, D = "x", B = 1))          # OK: all dependencies met
```

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

```
ps$check(list(A = FALSE, D = "z", B = 1))          # B's dependency is not met
```

```
## [1] "Condition for 'B' not ok: D anyof x, y; instead: D=z"
```

```
ps$check(list(A = FALSE, B = 1))                   # B's dependency is not met
```

```
## [1] "Condition for 'B' not ok: D anyof x, y; instead: D=<not-there>"
```

```
ps$check(list(A = FALSE, D = "z"))                 # OK: B is absent
```

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

```
ps$check(list(A = TRUE))                           # OK: neither B nor D present
```

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

```
ps$check(list(A = TRUE, D = "x", B = 1))           # D's dependency is not met
```

```
## [1] "Condition for 'D' not ok: A equal FALSE; instead: A=TRUE"
```

```
ps$check(list(A = TRUE, B = 1))                    # B's dependency is not met
```

```
## [1] "Condition for 'B' not ok: D anyof x, y; instead: D=<not-there>"
```

Internally, the dependencies are represented as a `data.table`, which can be accessed listed in the `$deps` slot. This `data.table` can even be mutated, to e.g. remove dependencies. There are no sanity checks done when the `$deps` slot is changed this way. Therefore it is advised to be cautious.

```
ps$deps
```

```
##    id on         cond
## 1:  D  A <CondEqual>
## 2:  B  D <CondAnyOf>
```

5.4.2.3 Vector Parameters

Unlike in the old `ParamHelpers` package, there are no more vectorial parameters in `paradox`. Instead, it is now possible to create multiple copies of a single parameter using the `$rep` function. This creates a `ParamSet` consisting of multiple copies of the parameter, which can then (optionally) be added to another `ParamSet`.

```
ps2d = ParamDbl$new("x", lower = 0, upper = 1)$rep(2)
print(ps2d)
```

```
## ParamSet:
##          id    class lower upper levels     default value
## 1: x_rep_1 ParamDbl     0     1         <NoDefault>
## 2: x_rep_2 ParamDbl     0     1         <NoDefault>
```

```
ps$add(ps2d)
print(ps)
```

```
## ParamSet:
##         id    class lower upper      levels     default parents value
## 1:       A ParamLgl    NA    NA  TRUE,FALSE <NoDefault>          TRUE
## 2:       B ParamInt     0    10             <NoDefault>       D     1
## 3:       C ParamDbl     0     4             <NoDefault>
## 4:       D ParamFct    NA    NA       x,y,z           y       A
## 5:       E ParamUty    NA    NA             <NoDefault>
## 6: x_rep_1 ParamDbl     0     1             <NoDefault>
## 7: x_rep_2 ParamDbl     0     1             <NoDefault>
```

It is also possible to use a `ParamUty` to accept vectorial parameters, which also works for parameters of variable length. A `ParamSet` containing a `ParamUty` can be used for parameter checking, but not for sampling. To sample values for a method that needs a vectorial parameter, it is advised to use a parameter transformation function that creates a vector from atomic values.

Assembling a vector from repeated parameters is aided by the parameter's `$tags`: Parameters that were generated by the `$rep()` command automatically get tagged as belonging to a group of repeated parameters.

```
ps$tags
```

```
## $A
## character(0)
##
## $B
## [1] "tag1" "tag2"
##
## $C
## character(0)
##
## $D
## character(0)
##
## $E
## character(0)
##
## $x_rep_1
## [1] "x_rep"
##
## $x_rep_2
## [1] "x_rep"
```

5.4.3 Parameter Sampling

It is often useful to have a list of possible parameter values that can be systematically iterated through, for example to find parameter values for which an algorithm performs particularly well (tuning). `paradox` offers a variety of functions that allow creating evenly-spaced parameter values in a "grid" design as well as random sampling. In the latter case, it is possible to influence the sampling distribution in more or less fine detail.

A point to always keep in mind while sampling is that only numerical and factorial parameters that are bounded can be sampled from, i.e. not `ParamUty`. Furthermore, for most samplers `ParamInt` and `ParamDbl` must have finite lower and upper bounds.

5.4.3.1 Parameter Designs

Functions that sample the parameter space fundamentally return an object of the `Design` class. These objects contain the sampled data as a `data.table` under the `$data` slot, and also offer conversion to a list of parameter-values using the **`$transpose()`** function.

5.4.3.2 Grid Design

The `generate_design_grid()` function is used to create grid designs that contain all combinations of parameter values: All possible values for `ParamLgl` and `ParamFct`, and values with a given resolution for `ParamInt` and `ParamDbl`. The resolution can be given for all numeric parameters, or for specific named parameters through the `param_resolutions` parameter.

```
design = generate_design_grid(psSmall, 2)
print(design)
```

```
## <Design> with 8 rows:
##         A  B C
## 1:  TRUE  0 0
## 2:  TRUE  0 4
## 3:  TRUE 10 0
## 4:  TRUE 10 4
## 5: FALSE  0 0
## 6: FALSE  0 4
## 7: FALSE 10 0
## 8: FALSE 10 4
```

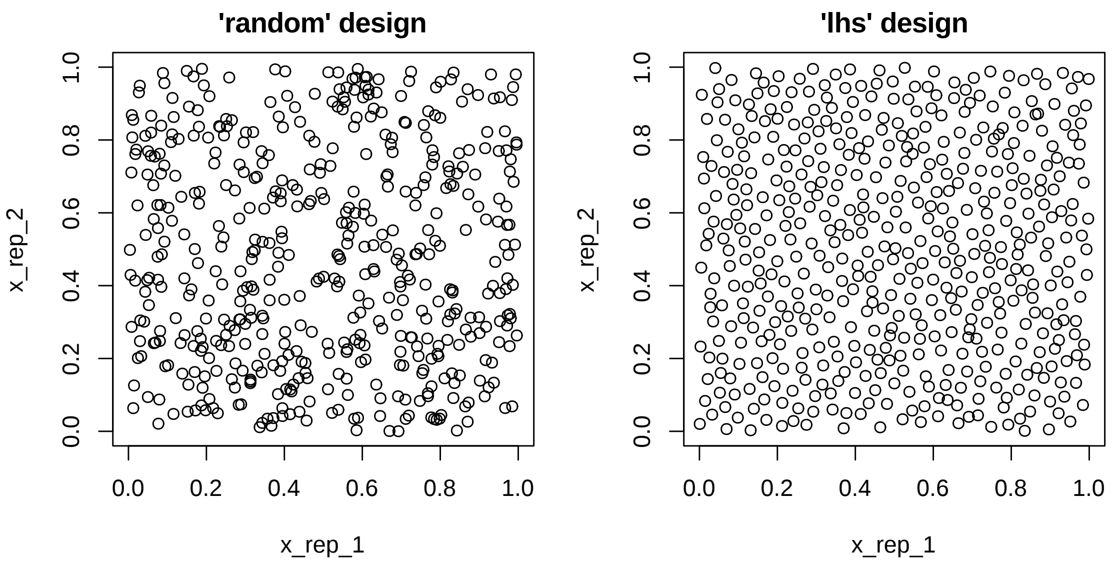
```
generate_design_grid(psSmall, param_resolutions = c(B = 1, C = 2))
```

```
## <Design> with 4 rows:
##    B C     A
## 1: 0 0  TRUE
## 2: 0 0 FALSE
## 3: 0 4  TRUE
## 4: 0 4 FALSE
```

5.4.3.3 Random Sampling

`paradox` offers different methods for random sampling, which vary in the degree to which they can be configured. The easiest way to get a uniformly random sample of parameters is `generate_design_random`. It is also possible to create "latin hypercube" sampled parameter values using **`generate_design_lhs`**, which utilizes the lhs package. LHS-sampling creates low-discrepancy sampled values that cover the parameter space more evenly than purely random values.

```
pvrand = generate_design_random(ps2d, 500)
pvlhs = generate_design_lhs(ps2d, 500)
```



5.4.3.4 Generalized Sampling: The `Sampler` Class

It may sometimes be desirable to configure parameter sampling in more detail. `paradox` uses the `Sampler` abstract base class for sampling, which has many different sub-classes that can be parameterized and combined to control the sampling process. It is even possible to create further sub-classes of the `Sampler` class (or of any of *its* subclasses) for even more possibilities.

Every `Sampler` object has a `sample()` function, which takes one argument, the number of instances to sample, and returns a `Design` object.

1D-Samplers

There is a variety of samplers that sample values for a single parameter. These are `Sampler1DUnif` (uniform sampling), `Sampler1DCateg` (sampling for categorical parameters), `Sampler1DNormal` (normally distributed sampling, truncated at parameter bounds), and `Sampler1DRfun` (arbitrary 1D sampling, given a random-function). These are initialized with a single `Param`, and can then be used to sample values.

```
sampA = Sampler1DCateg$new(parA)
sampA$sample(5)
```

```
## <Design> with 5 rows:
##        A
## 1: FALSE
## 2:  TRUE
## 3:  TRUE
## 4:  TRUE
## 5: FALSE
```

Hierarchical Sampler

The `SamplerHierarchical` sampler is an auxiliary sampler that combines many 1D-Samplers to get a combined distribution. Its name "hierarchical" implies that it is able to respect parameter dependencies. This suggests that parameters only get sampled when their dependencies are met.

The following example shows how this works: The `Int` parameter `B` depends on the `Lgl` parameter `A` being `TRUE`. `A` is sampled to be `TRUE` in about half the cases, in which case `B` takes a value between 0 and 10. In the cases where `A` is `FALSE`, `B` is set to `NA`.

```
psSmall$add_dep("B", "A", CondEqual$new(TRUE))
sampH = SamplerHierarchical$new(psSmall,
  list(Sampler1DCateg$new(parA),
    Sampler1DUnif$new(parB),
    Sampler1DUnif$new(parC))
)
sampled = sampH$sample(1000)
table(sampled$data[, c("A", "B")], useNA = "ifany")
```

```
##        B
## A         0   1   2   3   4   5   6   7   8   9  10 <NA>
##   FALSE   0   0   0   0   0   0   0   0   0   0   0  526
##   TRUE   44  54  41  51  45  22  54  42  33  39  49    0
```

Joint Sampler

Another way of combining samplers is the `SamplerJointIndep`. `SamplerJointIndep` also makes it possible to combine `Samplers` that are not 1D. However, `SamplerJointIndep` currently can not handle `ParamSets` with dependencies.

```
sampJ = SamplerJointIndep$new(
  list(Sampler1DUnif$new(ParamDbl$new("x", 0, 1)),
    Sampler1DUnif$new(ParamDbl$new("y", 0, 1)))
)
sampJ$sample(5)
```

```
## <Design> with 5 rows:
##         x      y
## 1: 0.8550 0.8847
## 2: 0.9319 0.9976
## 3: 0.3609 0.9467
## 4: 0.3545 0.5254
## 5: 0.2843 0.6502
```

SamplerUnif

The `Sampler` used in `generate_design_random` is the `SamplerUnif` sampler, which corresponds to a `HierarchicalSampler` of `Sampler1DUnif` for all parameters.

5.4.4 Parameter Transformation

While the different `Samplers` allow for a wide specification of parameter distributions, there are cases where the simplest way of getting a desired distribution is to sample parameters from a simple distribution (such as the uniform distribution) and then transform them. This can be done by assigning a function to the `$trafo` slot of a `ParamSet`. The `$trafo` function is called with two parameters:

- The list of parameter values to be transformed as `x`
- The `ParamSet` itself as `param_set`

The `$trafo` function must return a list of transformed parameter values.

The transformation is performed when calling the `$transpose` function of the `Design` object returned by a `Sampler` with the `trafo` ParamSet to `TRUE` (the default). The following, for example, creates a parameter that is exponentially distributed:

```
psexp = ParamSet$new(list(ParamDbl$new("par", 0, 1)))
psexp$trafo = function(x, param_set) {
  x$par = -log(x$par)
  x
}
design = generate_design_random(psexp, 2)
print(design)
```

```
## <Design> with 2 rows:
##       par
## 1: 0.7151
## 2: 0.3961
```

```
design$transpose()  # trafo is TRUE
```

```
## [[1]]
## [[1]]$par
## [1] 0.3353
##
##
## [[2]]
## [[2]]$par
## [1] 0.926
```

Compare this to `$transpose()` without transformation:

```
design$transpose(trafo = FALSE)
```

```
## [[1]]
## [[1]]$par
## [1] 0.7151
##
##
## [[2]]
## [[2]]$par
## [1] 0.3961
```

5.4.4.1 Transformation between Types

Usually the design created with one `ParamSet` is then used to configure other objects that themselves have a `ParamSet` which defines the values they take. The `ParamSets` which can be used for random sampling, however, are restricted in some ways: They must have finite bounds, and they may not contain “untyped” (`ParamUty`) parameters. `$trafo` provides the glue for these situations. There is relatively little constraint on the trafo function’s return value, so it is possible to return values that have different bounds or even types than the original `ParamSet`. It is even possible to remove some parameters and add new ones.

Suppose, for example, that a certain method requires a *function* as a parameter. Let’s say a function that summarizes its data in a certain way. The user can pass functions like `median()` or `mean()`, but could also pass quantiles or something completely different. This method would probably use the following `ParamSet`:

```
methodPS = ParamSet$new(
  list(
    ParamUty$new("fun",
      custom_check = function(x) checkmate::checkFunction(x, nargs = 1))
  )
)
print(methodPS)
```

```
## ParamSet:
##     id    class lower upper levels     default value
## 1: fun ParamUty    NA    NA        <NoDefault>
```

If one wanted to sample this method, using one of four functions, a way to do this would be:

```
samplingPS = ParamSet$new(
  list(
    ParamFct$new("fun", c("mean", "median", "min", "max"))
  )
)

samplingPS$trafo = function(x, param_set) {
  # x$fun is a `character(1)`,
  # in particular one of 'mean', 'median', 'min', 'max'.
  # We want to turn it into a function!
  x$fun = get(x$fun, mode = "function")
  x
}
```

```
design = generate_design_random(samplingPS, 2)
print(design)
```

```
## <Design> with 2 rows:
##    fun
## 1: min
## 2: max
```

Note that the `Design` only contains the column “`fun`” as a `character` column. To get a single value as a *function*, the `$transpose` function is used.

```
xvals = design$transpose()
print(xvals[[1]])
```

```
## $fun
## function (..., na.rm = FALSE)  .Primitive("min")
```

We can now check that it fits the requirements set by `methodPS`, and that `fun` it is in fact a function:

```
methodPS$check(xvals[[1]])
```

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

```
xvals[[1]]$fun(1:10)
```

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

Imagine now that a different kind of parametrization of the function is desired: The user wants to give a function that selects a certain quantile, where the quantile is set by a parameter. In that case the `$transpose` function could generate a function in a different way. For interpretability, the parameter is called "`quantile`" before transformation, and the "`fun`" parameter is generated on the fly.

```
samplingPS2 = ParamSet$new(
  list(
    ParamDbl$new("quantile", 0, 1)
  )
)

samplingPS2$trafo = function(x, param_set) {
  # x$quantile is a `numeric(1)` between 0 and 1.
  # We want to turn it into a function!
  list(fun = function(input) quantile(input, x$quantile))
}
```

```
design = generate_design_random(samplingPS2, 2)
print(design)
```

```
## <Design> with 2 rows:
##    quantile
## 1:   0.9958
## 2:   0.3016
```

The `Design` now contains the column "`quantile`" that will be used by the `$transpose` function to create the `fun` parameter. We also check that it fits the requirement set by `methodPS`, and that it is a function.

```
xvals = design$transpose()
print(xvals[[1]])
```

```
## $fun
## function(input) quantile(input, x$quantile)
## <environment: 0x40449f0>
```

```
methodPS$check(xvals[[1]])
```

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

```
xvals[[1]]$fun(1:10)
```

```
## 99.58%
##  9.962
```

5.5 Logging

We use the lgr package for logging and progress output.

5.5.1 Changing mlr3 logging levels

To change the setting for mlr3 for the current session, you need to retrieve the logger (which is a R6 object) from lgr, and then change the threshold of the like this:

```r
requireNamespace("lgr")

logger = lgr::get_logger("mlr3")
logger$set_threshold("<level>")
```

The default log level is `"info"`. All available levels can be listed as follows:

```r
getOption("lgr.log_levels")
```

```
## fatal error  warn  info debug trace
##   100   200   300   400   500   600
```

To increase verbosity, set the log level to a higher value, e.g. to `"debug"` with:

```r
lgr::get_logger("mlr3")$set_threshold("debug")
```

To reduce the verbosity, reduce the log level to warn:

```r
lgr::get_logger("mlr3")$set_threshold("warn")
```

lgr comes with a global option called `"lgr.default_threshold"` which can be set via `options()` to make your choice permanent across sessions.

Also note that extension packages such as mlr3tuning define their own logger which inherits from the mlr3 logger. To disable the output from mlr3, but keep the output from mlr3tuning, first reduce the verbosity for the parent logger mlr3, then restore verbosity for the child logger mlr3tuning:

```r
lgr::get_logger("mlr3")$set_threshold("warn")
lgr::get_logger("mlr3tuning")$set_threshold("info")
```

5.5.2 Redirecting output

Redirecting output is already extensively covered in the documentation and vignette of lgr. Here is just a short example which adds an additional appender to log events into a temporary file in JSON format:

```r
tf = tempfile("mlr3log_", fileext = ".json")

# get the logger as R6 object
logger = lgr::get_logger("mlr")

# add Json appender
logger$add_appender(lgr::AppenderJson$new(tf), name = "json")

# signal a warning
logger$warn("this is a warning from mlr3")
```

```
## WARN  [20:21:49.465] this is a warning from mlr3
```

```r
# print the contents of the file
cat(readLines(tf))
```

```
## {"level":300,"timestamp":"2020-02-24 20:21:49","logger":"mlr","caller":"eval","msg":"this i
```

```r
# remove the appender again
logger$remove_appender("json")
```

6 Extending

This chapter gives instructions on how to extend mlr3 and its extension packages with custom objects.

The approach is always the same:

1. determine the base class you want to inherit from,
2. extend the class with your custom functionality,
3. test your implementation
4. (optionally) add new object to the respective `Dictionary`.

The chapter Create a new learner illustrates the steps needed to create a custom learner in mlr3.

6.1 Adding new Learners

Here, we show how to create a custom `LearnerClassif` step-by-step.

Preferably, you checkout our mlr3learnertemplate for new learners. Alternatively, here is a template snippet for a new classification learner:

```r
LearnerClassifYourLearner = R6::R6Class("LearnerClassifYourLearner",
  inherit = LearnerClassif,
  public = list(
    initialize = function(id = "classif.yourlearner") {
      super$initialize(
        id = id,
        param_set = paradox::ParamSet$new(),
        predict_types = ,
        feature_types = ,
        properties = ,
        packages = ,
      )
    }
  ),

  private = list(
    .train = function(task) {

    },

    .predict = function(task) {

    }
  )
)
```

In the first line of the template, we create a new R6 class with class "LearnerClassifYourLearner". The next line determines the parent class: As we want to create a classification learner, we obviously want to inherit from LearnerClassif.

A learner consists of three parts:

1. Meta information about the learners
2. A private .train() function which takes a (filtered) TaskClassif and returns a model
3. A private.predict() function which operates on the model in self$model (stored during $train()) and a (differently subsetted) TaskClassif to return a named list of predictions.

6.1.1 Meta-information

In the constructor function initialize() the constructor of the super class LearnerClassif is called with meta information about the learner we want to construct. This includes:

- id: The id of the new learner.
- packages: Set of required packages to run the learner.
- param_set: A set of hyperparameters and their description, provided as paradox::ParamSet. It is perfectly fine to add no parameters here for a first draft. For each hyperparameter you want to add, you have to select the appropriate class:
  - paradox::ParamLgl for scalar logical hyperparameters.
  - paradox::ParamInt for scalar integer hyperparameters.
  - paradox::ParamDbl for scalar numeric hyperparameters.
  - paradox::ParamFct for scalar factor hyperparameters (this includes characters).
  - paradox::ParamUty for everything else.
- predict_types: Set of predict types the learner is capable of. These differ depending on the type of the learner.
  - LearnerClassif
    * response: Only predicts a class label for each observation in the test set.
    * prob: Also predicts the posterior probability for each class for each observation in the test set.
  - LearnerRegr
    * response: Only predicts a numeric response for each observation in the test set.
    * se: Also predicts the standard error for each value of response for each observation in the test set.
- feature_types: Set of feature types the learner can handle. See mlr_reflections$task_feature_types for feature types supported by mlr3.
- properties: Set of properties of the learner. Possible properties include:
  - "twoclass": The learner works on binary classification problems.
  - "multiclass": The learner works on multi-class classification problems.
  - "missings": The learner can natively handle missing values.
  - "weights": The learner can work on tasks which have observation weights / case weights.
  - "parallel": The learner can be parallelized, e.g. via threading.
  - "importance": The learner supports extracting importance values for features. If this property is set, you must also implement a public method importance() to retrieve the importance values from the model.
  - "selected_features": The learner supports extracting the features which where used. If this property is set, you must also implement a public method selected_features() to retrieve the set of used features from the model.

For a simplified rpart::rpart(), the initialization could look like this:

```
initialize = function(id = "classif.rpart") {
    ps = paradox::ParamSet$new(list(
      paradox::ParamDbl$new(id = "cp", default = 0.01, lower = 0, upper = 1, tags = "train"),
      paradox::ParamInt$new(id = "xval", default = 10L, lower = 0L, tags = "train")
    ))
    ps$values = list(xval = 0L)

    super$initialize(
        id = id,
        packages = "rpart",
        feature_types = c("logical", "integer", "numeric", "factor"),
        predict_types = c("response", "prob"),
        param_set = ps,
        properties = c("twoclass", "multiclass", "weights", "missings")
    )
}
```

We only have specified a small subset of the available hyperparameters:

- The complexity `"cp"` is numeric, its feasible range is `[0,1]`, it defaults to `0.01` and the parameter is used during `"train"`.
- The complexity `"xval"` is integer, its lower bound `0`, its default is `0` and the parameter is also used during `"train"`. Note that we have changed the default here from `10` to `0` to save some computation time. This is **not** done by setting a different `default` in `ParamInt$new()`, but instead by setting the value explicitly.

6.1.2 Train function

We continue the to adept the template for a rpart::rpart() learner, and now tackle private method `.train()`. The train function takes a Task as input and must return an arbitrary model. First, we write something down that works completely without `mlr3`:

```
data = iris
model = rpart::rpart(Species ~ ., data = iris, xval = 0)
```

In the next step, we replace the data frame `data` with a Task:

```
task = tsk("iris")
model = rpart::rpart(Species ~ ., data = task$data(), xval = 0)
```

The target variable `"Species"` is still hard-coded and specific to the task. This is unnecessary, as the information about the target variable is stored in the task:

```
task$target_names
```

```
## [1] "Species"
```

```
task$formula()
```

```
## Species ~ .
## NULL
```

We can adapt our code accordingly:

```
rpart::rpart(task$formula(), data = task$data(), xval = 0)
```

```
## n= 150
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
## 1) root 150 100 setosa (0.33333 0.33333 0.33333)
##   2) Petal.Length< 2.45 50   0 setosa (1.00000 0.00000 0.00000) *
##   3) Petal.Length>=2.45 100  50 versicolor (0.00000 0.50000 0.50000)
##     6) Petal.Width< 1.75 54   5 versicolor (0.00000 0.90741 0.09259) *
##     7) Petal.Width>=1.75 46   1 virginica (0.00000 0.02174 0.97826) *
```

The last thing missing is the handling of hyperparameters. Instead of the hard-coded `xval`, we query the hyperparameter settings from the `Learner` itself.

To illustrate this, we quickly construct the tree learner from the `mlr3` package, and use the method `get_value()` from the `ParamSet` to retrieve all set hyperparameters with tag `"train"`.

```
self = lrn("classif.rpart")
self$param_set$get_values(tags = "train")
```

```
## $xval
## [1] 0
```

To pass all hyperparameters down to the model fitting function, we recommend to use either `do.call` or the function `mlr3misc::invoke()`.

```
pars = self$param_set$get_values(tags = "train")
mlr3misc::invoke(rpart::rpart, task$formula(),
    data = task$data(), .args = pars)
```

```
## n= 150
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
## 1) root 150 100 setosa (0.33333 0.33333 0.33333)
##   2) Petal.Length< 2.45 50   0 setosa (1.00000 0.00000 0.00000) *
##   3) Petal.Length>=2.45 100  50 versicolor (0.00000 0.50000 0.50000)
##     6) Petal.Width< 1.75 54   5 versicolor (0.00000 0.90741 0.09259) *
##     7) Petal.Width>=1.75 46   1 virginica (0.00000 0.02174 0.97826) *
```

In the final learner, `self` will of course reference the learner itself. In the last step, we wrap everything in a function.

```
.train = function(task) {
  pars = self$param_set$get_values(tags = "train")
  mlr3misc::invoke(rpart::rpart, task$formula(),
    data = task$data(), .args = pars)
}
```

6.1.3 Predict function

The private predict function `.predict()` also operates on a Task as well as on the model stored during `train()` in `self$model`. The return value is a Prediction object. We proceed analogously to the section on the train function. We start with a version without any `mlr3` objects and continue to replace objects until we have reached the desired interface:

```
# inputs:
task = tsk("iris")
self = list(model = rpart::rpart(task$formula(), data = task$data()))

data = iris
response = predict(self$model, newdata = data, type = "class")
prob = predict(self$model, newdata = data, type = "prob")
```

The rpart::predict.rpart() function predicts class labels if argument `type` is set to to `"class"`, and class probabilities if set to `"prob"`.

Next, we transition from `data` to a `task` again and construct a proper PredictionClassif object to return. Additionally, as we do not want to run the prediction twice, we differentiate what type of prediction is requested by querying the set predict type of the learner. The complete internal predict function looks like this:

```
.predict = function(task) {
  self$predict_type = "response"
  response = prob = NULL

  if (self$predict_type == "response") {
    response = predict(self$model, newdata = task$data(), type = "class")
  } else {
    prob = predict(self$model, newdata = task$data(), type = "prob")
  }

  PredictionClassif$new(task, response = response, prob = prob)
}
```

Note that if the learner would need to handle hyperparameters during the predict step, we would proceed analogously to the `train()` step and use `self$params("predict")` in combination with mlr3misc::invoke().

Also note that you cannot rely on the column order of the data returned by `task$data()`, i.e. the order of columns may be different from the order of the columns during `$train()`. You have to make sure that your learner accesses columns by name, not by position (like some algorithms with a matrix interface do). You may have to restore the order manually here, see "classif.svm" for an example.

6.1.4 Final learner

```
LearnerClassifYourRpart = R6::R6Class("LearnerClassifYourRpart",
  inherit = LearnerClassif,
  public = list(
    initialize = function(id = "classif.rpart") {
      ps = paradox::ParamSet$new(list(
        paradox::ParamDbl$new(id = "cp", default = 0.01, lower = 0, upper = 1, tags = "train"),
```

```r
        paradox::ParamInt$new(id = "xval", default = 0L, lower = 0L, tags = "train")
      ))
      ps$values = list(xval = 0L)

      super$initialize(
        id = id,
        packages = "rpart",
        feature_types = c("logical", "integer", "numeric", "factor"),
        predict_types = c("response", "prob"),
        param_set = ps,
        properties = c("twoclass", "multiclass", "weights", "missings")
      )
    }
  ),

  private = list(
    .train = function(task) {
      pars = self$param_set$get_values(tag = "train")
      mlr3misc::invoke(rpart::rpart, task$formula(), data = task$data(), .args = pars)
    },

    .predict = function(task) {
      self$predict_type = "response"
      response = prob = NULL

      if (self$predict_type == "response") {
        response = predict(self$model, newdata = task$data(), type = "class")
      } else {
        prob = predict(self$model, newdata = task$data(), type = "prob")
      }
      PredictionClassif$new(task, response = response, prob = prob)
    }
  )
)

lrn = LearnerClassifYourRpart$new()
print(lrn)
```

```
## <LearnerClassifYourRpart:classif.rpart>
## * Model: -
## * Parameters: xval=0
## * Packages: rpart
## * Predict Type: response
## * Feature types: logical, integer, numeric, factor
## * Properties: missings, multiclass, twoclass, weights
```

To run some basic tests:

```r
task = tsk("iris")
lrn$train(task)
p = lrn$predict(task)
p$confusion
```

```
##             truth
## response     setosa versicolor virginica
##    setosa         50          0          0
##    versicolor      0         49          5
##    virginica       0          1         45
```

To run a bunch of automatic tests, you may source some auxiliary scripts from the unit tests of `mlr3`:

```r
helper = list.files(system.file("testthat", package = "mlr3"), pattern = "^helper.*\\.[rR]", full.name
ok = lapply(helper, source)
stopifnot(run_autotest(lrn))
```

6.2 Adding new PipeOps

This section showcases how the `mlr3pipelines` package can be extended to include custom `PipeOp`s. To run the following examples, we will need a `Task`; we are using the well-known "Iris" task:

```r
library("mlr3")
task = mlr_tasks$get("iris")
task$data()
```

```
##        Species Petal.Length Petal.Width Sepal.Length Sepal.Width
##   1:    setosa          1.4         0.2          5.1         3.5
##   2:    setosa          1.4         0.2          4.9         3.0
##   3:    setosa          1.3         0.2          4.7         3.2
##   4:    setosa          1.5         0.2          4.6         3.1
##   5:    setosa          1.4         0.2          5.0         3.6
##  ---
## 146: virginica          5.2         2.3          6.7         3.0
## 147: virginica          5.0         1.9          6.3         2.5
## 148: virginica          5.2         2.0          6.5         3.0
## 149: virginica          5.4         2.3          6.2         3.4
## 150: virginica          5.1         1.8          5.9         3.0
```

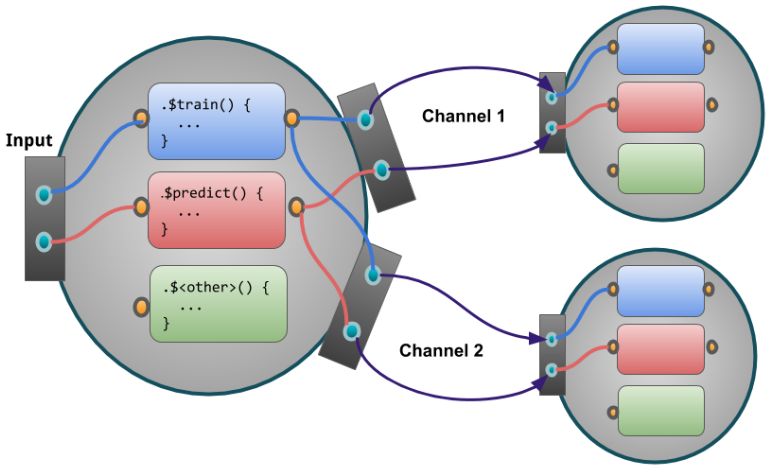
`mlr3pipelines` is fundamentally built around R6. When planning to create custom `PipeOp` objects, it can only help to familiarize yourself with it.

In principle, all a `PipeOp` must do is inherit from the `PipeOp` R6 class and implement the `train()` and `predict()` functions. There are, however, several auxiliary subclasses that can make the creation of *certain* operations much easier.

6.2.1 General Case Example: `PipeOpCopy`

A very simple yet useful `PipeOp` is `PipeOpCopy`, which takes a single input and creates a variable number of output channels, all of which receive a copy of the input data. It is a simple example that showcases the important steps in defining a custom `PipeOp`. We will show a simplified version here, **`PipeOpCopyTwo`**, that creates exactly two copies of its input data.

The following figure visualizes how our `PipeOp` is situated in the `Pipeline` and the significant in- and outputs.



6.2.1.1 First Steps: Inheriting from `PipeOp`

The first part of creating a custom `PipeOp` is inheriting from `PipeOp`. We make a mental note that we need to implement a `train()` and a `predict()` function, and that we probably want to have an `initialize()` as well:

```
PipeOpCopyTwo = R6::R6Class("PipeOpCopyTwo",
  inherit = mlr3pipelines::PipeOp,
  public = list(
    initialize = function(id = "copy.two") {
      ....
    },

    train = function(inputs) {
      ....
    },

    predict = function(inputs) {
      ....
    }
  )
)
```

6.2.1.2 Channel Definitions

We need to tell the `PipeOp` the layout of its channels: How many there are, what their names are going to be, and what types are acceptable. This is done on initialization of the `PipeOp` (using a `super$initialize` call) by giving the `input` and `output` `data.table` objects. These must have three columns: a `"name"` column giving the names of input and output channels, and a `"train"` and `"predict"` column naming the class of objects we expect during training and prediction as input / output. A special value for these classes is `"*"`, which indicates that any class will be accepted; our simple copy operator accepts any kind of input, so this will be useful. We have only one input, but two output channels.

By convention, we name a single channel `"input"` or `"output"`, and a group of channels [`"input1"`, `"input2"`, …], unless there is a reason to give specific different names. Therefore, our `input` `data.table` will have a single row `<"input", "*", "*">`, and our `output` table will have two rows, `<"output1", "*", "*">` and `<"output2", "*", "*">`.

All of this is given to the `PipeOp` creator. Our `initialize()` will thus look as follows:

```
    initialize = function(id = "copy.two") {
      input = data.table::data.table(name = "input", train = "*", predict = "*")
      # the following will create two rows and automatically fill the `train`
      # and `predict` cols with "*"
      output = data.table::data.table(
        name = c("output1", "output2"),
        train = "*", predict = "*"
      )
      super$initialize(id,
        input = input,
        output = output
      )
    }
```

6.2.1.3 Train and Predict

Both `train()` and `predict()` will receive a `list` as input and must give a `list` in return. According to our `input` and `output` definitions, we will always get a list with a single element as input, and will need to return a list with two elements. Because all we want to do is create two copies, we will just create the copies using `c(inputs, inputs)`.

Two things to consider:

- The `train()` function must always modify the `self$state` variable to something that is not `NULL` or `NO_OP`. This is because the `$state` slot is used as a signal that `PipeOp` has been trained on data, even if the state itself is not important to the `PipeOp` (as in our case). Therefore, our `train()` will set `self$state = list()`.
- It is not necessary to "clone" our input or make deep copies, because we don't modify the data. However, if we were changing a reference-passed object, for example by changing data in a `Task`, we would have to make a deep copy first. This is because a `PipeOp` may never modify its input object by reference.

Our `train()` and `predict()` functions are now:

```
train = function(inputs) {
  self$state = list()
  c(inputs, inputs)
}
```

```
predict = function(inputs) {
  c(inputs, inputs)
}
```

6.2.1.4 Putting it Together

The whole definition thus becomes

```
PipeOpCopyTwo = R6::R6Class("PipeOpCopyTwo",
  inherit = mlr3pipelines::PipeOp,
  public = list(
    initialize = function(id = "copy.two") {
      super$initialize(id,
        input = data.table::data.table(name = "input", train = "*", predict = "*"),
        output = data.table::data.table(name = c("output1", "output2"),
                              train = "*", predict = "*")
      )
    },

    train = function(inputs) {
      self$state = list()
      c(inputs, inputs)
    },

    predict = function(inputs) {
      c(inputs, inputs)
    }
  )
)
```

We can create an instance of our `PipeOp`, put it in a graph, and see what happens when we train it on something:

```
library("mlr3pipelines")
poct = PipeOpCopyTwo$new()
gr = Graph$new()
gr$add_pipeop(poct)

print(gr)
```

```
## Graph with 1 PipeOps:
##        ID         State sccssors prdcssors
##  copy.two <<UNTRAINED>>
```

```
result = gr$train(task)

str(result)
```

```
## List of 2
##  $ copy.two.output1:Classes 'TaskClassif', 'TaskSupervised', 'Task', 'R6' <TaskClassif:iris>
##  $ copy.two.output2:Classes 'TaskClassif', 'TaskSupervised', 'Task', 'R6' <TaskClassif:iris>
```

6.2.2 Special Case: Preprocessing

Many PipeOps perform an operation on exactly one `Task`, and return exactly one `Task`. They may even not care about the "Target" / "Outcome" variable of that task, and only do some modification of some input data. However, it is usually important to them that the `Task` on which they perform prediction has the same data columns as the `Task` on which they train. For these cases, the auxiliary base class `PipeOpTaskPreproc` exists. It inherits from `PipeOp` itself, and other PipeOps should use it if they fall in the kind of use-case named above.

When inheriting from `PipeOpTaskPreproc`, one must either implement the `train_task` and predict_task functions, or the train_dt, predict_dt functions, depending on whether wants to operate on a Task object or on data.tables. In the second case, one can optionally also overload the select_cols function, which chooses which of the incoming Task's features are given to the train_dt / predict_dt functions.

The following will show two examples: PipeOpDropNA, which removes a Task's rows with missing values during training (and implements train_task and predict_task), and PipeOpScale, which scales a Task's numeric columns (and implements train_dt, predict_dt, and select_cols).

6.2.2.1 Example: PipeOpDropNA

Dropping rows with missing values may be important when training a model that can not handle them.

Because mlr3 Tasks only contain a view to the underlying data, it is not necessary to modify data to remove rows with missing values. Instead, the rows can be removed using the Task's $filter method, which modifies the Task in-place. This is done in the train_task function. We take care that we also set the $state slot to signal that the PipeOp was trained.

The predict_task function does not need to do anything; removing missing values during prediction is not as useful, since learners that cannot handle them will just ignore the respective rows. Furthermore, mlr3 expects a Learner to always return just as many predictions as it was given input rows, so a PipeOp that removes Task rows during training can not be used inside a GraphLearner.

When we inherit from PipeOpTaskPreproc, it sets the input and output data.tables for us to only accept a single Task. The only thing we do during initialize() is therefore to set an id (which can optionally be changed by the user).

The complete PipeOpDropNA can therefore be written as follows. Note that it inherits from PipeOpTaskPreproc, unlike the PipeOpCopyTwo example from above:

```r
PipeOpDropNA = R6::R6Class("PipeOpDropNA",
  inherit = mlr3pipelines::PipeOpTaskPreproc,
  public = list(
    initialize = function(id = "drop.na") {
      super$initialize(id)
    },

    train_task = function(task) {
      self$state = list()
      featuredata = task$data(cols = task$feature_names)
      exclude = apply(is.na(featuredata), 1, any)
      task$filter(task$row_ids[!exclude])
    },

    predict_task = function(task) {
      # nothing to be done
      task
    }
  )
)
```

To test this PipeOp, we create a small task with missing values:

```r
smalliris = iris[(1:5) * 30, ]
smalliris[1, 1] = NA
smalliris[2, 2] = NA
sitask = TaskClassif$new("smalliris", as_data_backend(smalliris), "Species")
print(sitask$data())
```

```
##       Species Petal.Length Petal.Width Sepal.Length Sepal.Width
## 1:     setosa          1.6         0.2           NA         3.2
## 2: versicolor          3.9         1.4          5.2          NA
## 3: versicolor          4.0         1.3          5.5         2.5
## 4:  virginica          5.0         1.5          6.0         2.2
## 5:  virginica          5.1         1.8          5.9         3.0
```

We test this by feeding it to a new `Graph` that uses `PipeOpDropNA`.

```r
gr = Graph$new()
gr$add_pipeop(PipeOpDropNA$new())

filtered_task = gr$train(sitask)[[1]]
print(filtered_task$data())
```

```
##       Species Petal.Length Petal.Width Sepal.Length Sepal.Width
## 1: versicolor          4.0         1.3          5.5         2.5
## 2:  virginica          5.0         1.5          6.0         2.2
## 3:  virginica          5.1         1.8          5.9         3.0
```

6.2.2.2 Example: `PipeOpScaleAlways`

An often-applied preprocessing step is to simply **center** and/or **scale** the data to mean 0 and standard deviation 1. This fits the `PipeOpTaskPreproc` pattern quite well. Because it always replaces all columns that it operates on, and does not require any information about the task's target, it only needs to overload the `train_dt` and `predict_dt` functions. This saves some boilerplate-code from getting the correct feature columns out of the task, and replacing them after modification.

Because scaling only makes sense on numeric features, we want to instruct `PipeOpTaskPreproc` to give us only these numeric columns. We do this by overloading the `select_cols` function: It is called by the class to determine which columns to give to `train_dt` and `predict_dt`. Its input is the `Task` that is being transformed, and it should return a `character` vector of all features to work with. When it is not overloaded, it uses all columns; instead, we will set it to only give us numeric columns. Because the `levels()` of the data table given to `train_dt` and `predict_dt` may be different from the levels `task`'s levels, these functions must also take a `levels` argument that is a named list of column names indicating their levels. When working with numeric data, this argument can be ignored, but it should be used instead of `levels(dt[[column]])` for factorial or character columns.

This is the first `PipeOp` where we will be using the `$state` slot for something useful: We save the centering offset and scaling coefficient and use it in `$predict()`!

For simplicity, we are not using hyperparameters and will always scale and center all data. Compare this `PipeOpScaleAlways` operator to the one defined inside the `mlr3pipelines` package, `PipeOpScale`, defined in `PipeOpScale.R`.

```
PipeOpScaleAlways = R6::R6Class("PipeOpScaleAlways",
  inherit = mlr3pipelines::PipeOpTaskPreproc,
  public = list(
    initialize = function(id = "scale.always") {
      super$initialize(id = id)
    },

    select_cols = function(task) {
      task$feature_types[type == "numeric", id]
    },

    train_dt = function(dt, levels, target) {
      sc = scale(as.matrix(dt))
      self$state = list(
        center = attr(sc, "scaled:center"),
        scale = attr(sc, "scaled:scale")
      )
      sc
    },

    predict_dt = function(dt, levels) {
      t((t(dt) - self$state$center) / self$state$scale)
    }
  )
)
```

(Note for the observant: If you check `PipeOpScale.R` from the `mlr3pipelines` package, you will notice that is uses "`get("type")`" and "`get("id")`" instead of "`type`" and "`id`", because the static code checker on CRAN would otherwise complain about references to undefined variables. This is a "problem" with `data.table` and not exclusive to `mlr3pipelines`.)

We can, again, create a new `Graph` that uses this `PipeOp` to test it. Compare the resulting data to the original "iris" `Task` data printed at the beginning:

```
gr = Graph$new()
gr$add_pipeop(PipeOpScaleAlways$new())

result = gr$train(task)

result[[1]]$data()
```

```
##        Species Petal.Length Petal.Width Sepal.Length Sepal.Width
##   1:    setosa      -1.3358     -1.3111     -0.89767     1.01560
##   2:    setosa      -1.3358     -1.3111     -1.13920    -0.13154
##   3:    setosa      -1.3924     -1.3111     -1.38073     0.32732
##   4:    setosa      -1.2791     -1.3111     -1.50149     0.09789
##   5:    setosa      -1.3358     -1.3111     -1.01844     1.24503
##  ---
## 146: virginica       0.8169      1.4440      1.03454    -0.13154
## 147: virginica       0.7036      0.9192      0.55149    -1.27868
## 148: virginica       0.8169      1.0504      0.79301    -0.13154
## 149: virginica       0.9302      1.4440      0.43072     0.78617
## 150: virginica       0.7602      0.7880      0.06843    -0.13154
```

6.2.3 Special Case: Preprocessing with Simple Train

It is possible to make even further simplifications for many PipeOps that perform mostly the same operation during training and prediction. The point of Task preprocessing is often to modify the training data in mostly the same way as prediction data (but in a way that *may* depend on training data).

Consider constant feature removal, for example: The goal is to remove features that have no variance, or only a single factor level. However, what features get removed must be decided during *training*, and may only depend on training data. Furthermore, the actual process of removing features is the same during training and prediction.

A simplification to make is therefore to have a function `get_state(task)` which sets the `$state` slot during training, and a `transform(task)` function, which gets called both during training *and* prediction. This is done in the `PipeOpTaskPreprocSimple` class. Just like `PipeOpTaskPreproc`, one can inherit from this and overload these functions to get a `PipeOp` that performs preprocessing with very little boilerplate code.

Just like `PipeOpTaskPreproc`, `PipeOpTaskPreprocSimple` offers the possibility to instead overload the `get_state_dt(dt, levels)` and `transform_dt(dt, levels)` functions (and optionally, again, the `select_cols(task)` function) to operate on `data.table` feature data instead of the whole `Task`.

Even some methods that do not use `PipeOpTaskPreprocSimple` *could* work in a similar way: The `PipeOpScaleAlways` example from above will be shown to also work with this paradigm.

6.2.3.1 Example: `PipeOpDropConst`

A typical example of a preprocessing operation that does almost the same operation during training and prediction is an operation that drops features depending on a criterion that is evaluated during training. One simple example of this is dropping constant features. Because the `mlr3 Task` class offers a flexible view on underlying data, it is most efficient to drop columns from the task directly using its `$select()` function, so the `get_state_dt(dt, levels)` / `transform_dt(dt, levels)` functions will *not* get used; instead we overload the `get_state(task)` and `transform(task)` functions.

The `get_state()` function's result is saved to the `$state` slot, so we want to return something that is useful for dropping features. We choose to save the names of all the columns that have nonzero variance. For brevity, we use `length(unique(column)) > 1` to check whether more than one distinct value is present; a more sophisticated version could have a tolerance parameter for numeric values that are very close to each other.

The `transform()` function is evaluated both during training *and* prediction, and can rely on the `$state` slot being present. All it does here is call the `Task$select` function with the columns we chose to keep.

The full `PipeOp` could be written as follows:

```r
PipeOpDropConst = R6::R6Class("PipeOpDropConst",
  inherit = mlr3pipelines::PipeOpTaskPreprocSimple,
  public = list(
    initialize = function(id = "drop.const") {
      super$initialize(id = id)
    },
    get_state = function(task) {
      data = task$data(cols = task$feature_names)
      nonconst = sapply(data, function(column) length(unique(column)) > 1)
      list(cnames = colnames(data)[nonconst])
    },

    transform = function(task) {
      task$select(self$state$cnames)
    }
  )
)
```

This can be tested using the first five rows of the "Iris" `Task`, for which one feature (`"Petal.Width"`) is constant:

```
irishead = task$clone()$filter(1:5)
irishead$data()
```

```
##    Species Petal.Length Petal.Width Sepal.Length Sepal.Width
## 1:  setosa          1.4         0.2          5.1         3.5
## 2:  setosa          1.4         0.2          4.9         3.0
## 3:  setosa          1.3         0.2          4.7         3.2
## 4:  setosa          1.5         0.2          4.6         3.1
## 5:  setosa          1.4         0.2          5.0         3.6
```

```
gr = Graph$new()$add_pipeop(PipeOpDropConst$new())
dropped_task = gr$train(irishead)[[1]]

dropped_task$data()
```

```
##    Species Petal.Length Sepal.Length Sepal.Width
## 1:  setosa          1.4          5.1         3.5
## 2:  setosa          1.4          4.9         3.0
## 3:  setosa          1.3          4.7         3.2
## 4:  setosa          1.5          4.6         3.1
## 5:  setosa          1.4          5.0         3.6
```

We can also see that the `$state` was correctly set. Calling `$predict()` with this graph, even with different data (the whole Iris `Task`!) will still drop the `"Petal.Width"` column, as it should.

```
gr$pipeops$drop.const$state
```

```
## $cnames
## [1] "Petal.Length" "Sepal.Length" "Sepal.Width"
##
## $affected_cols
## [1] "Petal.Length" "Petal.Width"  "Sepal.Length" "Sepal.Width"
##
## $intasklayout
##              id    type
## 1: Petal.Length numeric
## 2:  Petal.Width numeric
## 3: Sepal.Length numeric
## 4:  Sepal.Width numeric
```

```
##
## $outtasklayout
##              id    type
## 1: Petal.Length numeric
## 2: Sepal.Length numeric
## 3:  Sepal.Width numeric
```

```
dropped_predict = gr$predict(task)[[1]]

dropped_predict$data()
```

```
##        Species Petal.Length Sepal.Length Sepal.Width
##   1:    setosa          1.4          5.1         3.5
##   2:    setosa          1.4          4.9         3.0
##   3:    setosa          1.3          4.7         3.2
##   4:    setosa          1.5          4.6         3.1
##   5:    setosa          1.4          5.0         3.6
##  ---
## 146: virginica          5.2          6.7         3.0
## 147: virginica          5.0          6.3         2.5
## 148: virginica          5.2          6.5         3.0
## 149: virginica          5.4          6.2         3.4
## 150: virginica          5.1          5.9         3.0
```

6.2.3.2 Example: `PipeOpScaleAlwaysSimple`

This example will show how a `PipeOpTaskPreprocSimple` can be used when only working on feature data in form of a `data.table`. Instead of calling the `scale()` function, the `center` and `scale` values are calculated directly and saved to the `$state` slot. The `transform_dt` function will then perform the same operation during both training and prediction: subtract the `center` and divide by the `scale` value. As in the `PipeOpScaleAlways` example above, we use `select_cols()` so that we only work on numeric columns.

```
PipeOpScaleAlwaysSimple = R6::R6Class("PipeOpScaleAlwaysSimple",
  inherit = mlr3pipelines::PipeOpTaskPreprocSimple,
  public = list(
    initialize = function(id = "scale.always.simple") {
      super$initialize(id = id)
    },

    select_cols = function(task) {
      task$feature_types[type == "numeric", id]
    },

    get_state_dt = function(dt, levels, target) {
      list(
        center = sapply(dt, mean),
        scale = sapply(dt, sd)
      )
    },

    transform_dt = function(dt, levels) {
      t((t(dt) - self$state$center) / self$state$scale)
    }
  )
)
```

We can compare this PipeOp to the one above to show that it behaves the same.

```
gr = Graph$new()$add_pipeop(PipeOpScaleAlways$new())
result_posa = gr$train(task)[[1]]

gr = Graph$new()$add_pipeop(PipeOpScaleAlwaysSimple$new())
result_posa_simple = gr$train(task)[[1]]
```

```
result_posa$data()
```

```
##        Species Petal.Length Petal.Width Sepal.Length Sepal.Width
##   1:    setosa      -1.3358     -1.3111     -0.89767     1.01560
##   2:    setosa      -1.3358     -1.3111     -1.13920    -0.13154
##   3:    setosa      -1.3924     -1.3111     -1.38073     0.32732
##   4:    setosa      -1.2791     -1.3111     -1.50149     0.09789
##   5:    setosa      -1.3358     -1.3111     -1.01844     1.24503
##  ---
## 146: virginica       0.8169      1.4440      1.03454    -0.13154
## 147: virginica       0.7036      0.9192      0.55149    -1.27868
## 148: virginica       0.8169      1.0504      0.79301    -0.13154
## 149: virginica       0.9302      1.4440      0.43072     0.78617
## 150: virginica       0.7602      0.7880      0.06843    -0.13154
```

```
result_posa_simple$data()
```

```
##        Species Petal.Length Petal.Width Sepal.Length Sepal.Width
##   1:    setosa      -1.3358     -1.3111     -0.89767     1.01560
##   2:    setosa      -1.3358     -1.3111     -1.13920    -0.13154
##   3:    setosa      -1.3924     -1.3111     -1.38073     0.32732
##   4:    setosa      -1.2791     -1.3111     -1.50149     0.09789
##   5:    setosa      -1.3358     -1.3111     -1.01844     1.24503
##  ---
## 146: virginica       0.8169      1.4440      1.03454    -0.13154
## 147: virginica       0.7036      0.9192      0.55149    -1.27868
## 148: virginica       0.8169      1.0504      0.79301    -0.13154
## 149: virginica       0.9302      1.4440      0.43072     0.78617
## 150: virginica       0.7602      0.7880      0.06843    -0.13154
```

6.2.4 Hyperparameters

mlr3pipelines uses the paradox package to define parameter spaces for PipeOps. Parameters for PipeOps can modify their behavior in certain ways, e.g. switch centering or scaling off in the PipeOpScale operator. The unified interface makes it possible to have parameters for whole Graphs that modify the individual PipeOp's behavior. The Graphs, when encapsulated in GraphLearners, can even be tuned using the tuning functionality in mlr3tuning.

Hyperparameters are declared during initialization, when calling the PipeOp's $initialize() function, by giving a param_set argument. The param_set must be a ParamSet from the paradox package; see the mlr3book for more information on how to define parameter spaces. After construction, the ParamSet can be accessed through the $param_set slot. While it is *possible* to modify this ParamSet, using e.g. the $add() and $add_dep() functions, *after* adding it to the PipeOp, it is strongly advised against.

Hyperparameters can be set and queried through the $values slot. When setting hyperparameters, they are automatically checked to satisfy all conditions set by the $param_set, so it is not necessary to type check them. Be aware that it is always possible to *remove* hyperparameter values.

When a PipeOp is initialized, it usually does not have any parameter values—$values takes the value list(). It is possible to set initial parameter values in the $initialize() constructor; this must be done *after* the super$initialize() call where the corresponding ParamSet must be supplied. This is because setting $values checks against the current $param_set, which would fail if the $param_set was not set yet.

When using an underlying library function (the scale function in PipeOpScale, say), then there is usually a “default” behaviour of that function when a parameter is not given. It is good practice to use this default behaviour whenever a parameter is not set (or when it was removed). This can easily be done when using the mlr3misc library’s mlr3misc::invoke() function, which has functionality similar to do.call().

6.2.4.1 Hyperparameter Example: PipeOpScale

How to use hyperparameters can best be shown through the example of PipeOpScale, which is very similar to the example above, PipeOpScaleAlways. The difference is made by the presence of hyperparameters. PipeOpScale constructs a ParamSet in its $initialize function and passes this on to the super$initialize function:

```r
PipeOpScale$public_methods$initialize
```

```
## function (id = "scale", param_vals = list())
## {
##     ps = ParamSet$new(params = list(ParamLgl$new("center", default = TRUE,
##         tags = c("train", "scale")), ParamLgl$new("scale", default = TRUE,
##         tags = c("train", "scale"))))
##     super$initialize(id = id, param_set = ps, param_vals = param_vals)
## }
## <bytecode: 0x5a232a0>
## <environment: namespace:mlr3pipelines>
```

The user has access to this and can set and get parameters. Types are automatically checked:

```r
pss = PipeOpScale$new()
print(pss$param_set)
```

```
## ParamSet: scale
##                id    class lower upper      levels     default value
## 1:         center ParamLgl    NA    NA  TRUE,FALSE        TRUE
## 2:          scale ParamLgl    NA    NA  TRUE,FALSE        TRUE
## 3: affect_columns ParamUty    NA    NA             <NoDefault>
```

```r
pss$param_set$values$center = FALSE
print(pss$param_set$values)
```

```
## $center
## [1] FALSE
```

```
pss$param_set$values$scale = "TRUE"  # bad input is checked!
```

```
## Error in (function (xs) : Assertion on 'xs' failed: scale: Must be of type 'logical flag', not '
```

How `PipeOpScale` handles its parameters can be seen in its `$train` method: It gets the relevant parameters from its `$values` slot and uses them in the `mlr3misc::invoke()` call. This has the advantage over calling `scale()` directly that if a parameter is not given, its default value from the `scale()` function will be used.

```
PipeOpScale$public_methods$train
```

```
## function (dt, levels, target)
## {
##   sc = invoke(scale, as.matrix(dt), .args = self$param_set$get_values(tags = "scale"))
##     self$state = list(center = attr(sc, "scaled:center") %??%
##         0, scale = attr(sc, "scaled:scale") %??% 1)
##     constfeat = self$state$scale == 0
##     self$state$scale[constfeat] = 1
##     sc[, constfeat] = 0
##     sc
## }
## <bytecode: 0x5a28080>
## <environment: namespace:mlr3pipelines>
```

Another change that is necessary compared to `PipeOpScaleAlways` is that the attributes `"scaled:scale"` and `"scaled:center"` are not always present, depending on parameters, and possibly need to be set to default values 1 or 0, respectively.

It is now even possible (if a bit pointless) to call `PipeOpScale` with both `scale` and `center` set to `FALSE`, which returns the original dataset, unchanged.

```
pss$param_set$values$scale = FALSE
pss$param_set$values$center = FALSE

gr = Graph$new()
gr$add_pipeop(pss)

result = gr$train(task)

result[[1]]$data()
```

```
##        Species Petal.Length Petal.Width Sepal.Length Sepal.Width
##   1:    setosa          1.4         0.2          5.1         3.5
##   2:    setosa          1.4         0.2          4.9         3.0
##   3:    setosa          1.3         0.2          4.7         3.2
##   4:    setosa          1.5         0.2          4.6         3.1
##   5:    setosa          1.4         0.2          5.0         3.6
##  ---
## 146: virginica          5.2         2.3          6.7         3.0
## 147: virginica          5.0         1.9          6.3         2.5
## 148: virginica          5.2         2.0          6.5         3.0
## 149: virginica          5.4         2.3          6.2         3.4
## 150: virginica            5.1          1.8          5.9          3.0
```

7 Special Tasks

This chapter explores the different functions of `mlr3` when dealing with specific data sets that require further statistical modification to undertake sensible analysis. Following topics are discussed:

Survival Analysis

This sub-chapter explains how to conduct sound survival analysis in mlr3. Survival analysis is used to monitor the period of time until a specific event takes places. This specific event could be e.g. death, transmission of a disease, marriage or divorce. Two considerations are important when conducting survival analysis:

- Whether the event occurred within the frame of the given data
- How much time it took until the event occurred

In summary, this sub-chapter explains how to account for these considerations and conduct survival analysis using the `mlr3proba` extension package.

Spatial Analysis

Spatial analysis data observations entail reference information about spatial characteristics. One of the largest shortcomings of spatial data analysis is the inevitable auto-correlation in spatial data. Auto-correlation is especially severe in data with marginal spatial variation. The sub-chapter on spatial analysis provides instructions on how to handle the problems associated with spatial data accordingly.

Ordinal Analysis

This is work in progress. See mlr3ordinal for the current state.

Functional Analysis

Functional analysis contains data that consists of curves varying over a continuum e.g. time, frequency or wavelength. This type of analysis is frequently used when examining measurements over various time points. Steps on how to accommodate functional data structures in mlr3 are explained in the functional analysis sub-chapter.

Multilabel Classification

Multilabel classification deals with objects that can belong to more than one category at the same time. Numerous target labels are attributed to a single observation. Working with multilabel data requires one to use modified algorithms, to accommodate data specific characteristics. Two approaches to multilabel classification are prominently used:

- The problem transformation method
- The algorithm adaption method

Instructions on how to deal with multilabel classification in mlr3 can be found in this sub-chapter.

Cost Sensitive Classification

This sub-chapter deals with the implementation of cost-sensitive classification. Regular classification aims to minimize the misclassification rate and thus all types of misclassification errors are deemed equally severe. Cost-sensitive classification is a setting where the costs caused by different kinds of errors are not assumed to be equal. The objective is to minimize the expected costs.

Analytical data for a big credit institution is used as a use case to illustrate the different features. Firstly, the sub-chapter provides guidance on how to implement a first model. Subsequently, the sub-chapter contains instructions on how to modify cost sensitivity measures, thresholding and threshold tuning.

7.1 Survival Analysis

Survival analysis examines data on whether a specific event of interest takes place and how long it takes till this event occurs. One cannot use ordinary regression analysis when dealing with survival analysis data sets. Firstly, survival data contains solely positive values and therefore needs to be transformed to avoid biases. Secondly, ordinary regression analysis cannot deal with censored observations accordingly. Censored observations are observations in which the event of interest has not occurred, yet. Survival analysis allows the user to handle censored data with limited time frames that sometimes do not entail the event of interest. Note that survival analysis accounts for both censored and uncensored observations while adjusting respective model parameters.

The package mlr3proba extends mlr3 with the following objects for survival analysis:

- `TaskSurv` to define (right-censored) survival tasks
- `LearnerSurv` as base class for survival learners
- `PredictionSurv` as specialized class for `Prediction` objects
- `MeasureSurv` as specialized class for performance measures

In this example we demonstrate the basic functionality of the package on the `rats` data from the survival package. This task ships as pre-defined `TaskSurv` with mlr3proba.

```r
library("mlr3proba")
task = tsk("rats")
print(task)
```

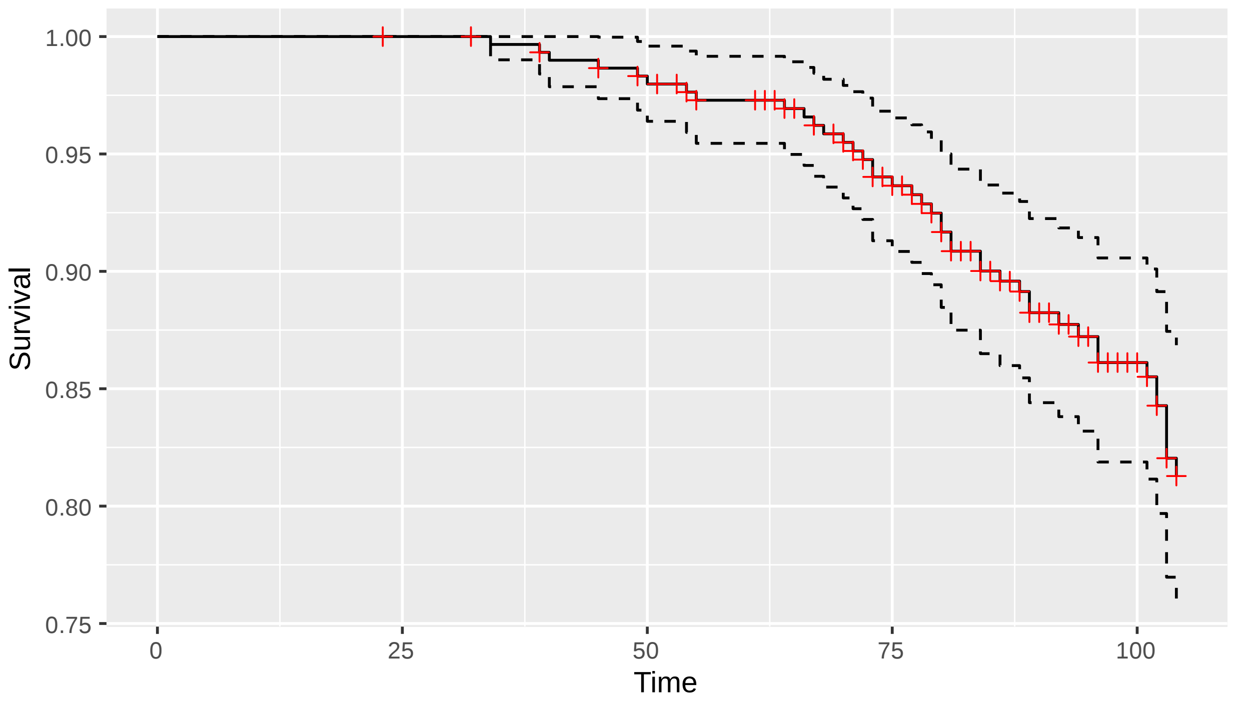
```
## <TaskSurv:rats> (300 x 5)
## * Target: time, status
## * Properties: -
## * Features (3):
##   - int (2): litter, rx
##   - fct (1): sex
```

```r
# the target column is a survival object:
head(task$truth())
```

```
## [1] 101+  49  104+  91+ 104+ 102+
```

```r
# kaplan-meier plot
library("mlr3viz")
autoplot(task)
```

```
## Registered S3 method overwritten by 'GGally':
##   method from
##   +.gg   ggplot2
```



Now, we conduct a small benchmark study on the `rats` task using some of the integrated survival learners:

```
# some integrated learners
learners = lapply(c("surv.coxph", "surv.kaplan", "surv.ranger"), lrn)
print(learners)
```

```
## [[1]]
## <LearnerSurvCoxPH:surv.coxph>
## * Model: -
## * Parameters: list()
## * Packages: survival, distr6
## * Predict Type: distr
## * Feature types: logical, integer, numeric, factor
## * Properties: importance
##
## [[2]]
## <LearnerSurvKaplan:surv.kaplan>
## * Model: -
## * Parameters: list()
## * Packages: survival, distr6
## * Predict Type: crank
## * Feature types: logical, integer, numeric, character, factor, ordered
## * Properties: missings
##
## [[3]]
## <LearnerSurvRanger:surv.ranger>
## * Model: -
## * Parameters: list()
## * Packages: ranger, distr6
## * Predict Type: distr
## * Feature types: logical, integer, numeric, character, factor, ordered
## * Properties: importance, oob_error, weights
```

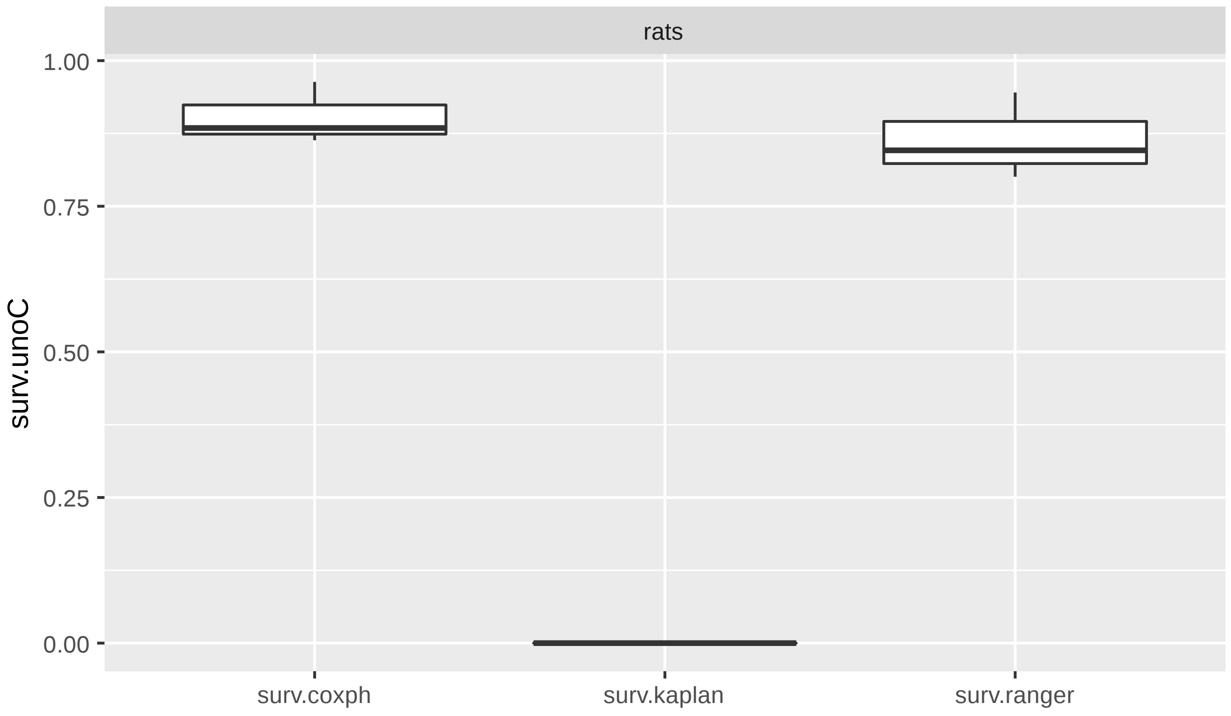
```r
# Uno's C-Index for survival
measure = msr("surv.unoC")
print(measure)
```

```
## <MeasureSurvUnoC:surv.unoC>
## * Packages: survAUC
## * Range: [0, 1]
## * Minimize: FALSE
## * Properties: na_score, requires_task, requires_train_set
## * Predict type: crank
```

```r
set.seed(1)
bmr = benchmark(benchmark_grid(task, learners, rsmp("cv", folds = 3)))
bmr$aggregate(measure)
```

```
##    nr  resample_result task_id  learner_id resampling_id iters surv.unoC
## 1:  1 <ResampleResult>    rats  surv.coxph            cv     3    0.9037
## 2:  2 <ResampleResult>    rats surv.kaplan            cv     3    0.0000
## 3:  3 <ResampleResult>    rats surv.ranger            cv     3    0.8640
```

```r
autoplot(bmr, measure = measure)
```



7.2 Spatial Analysis

Spatial data observations entail reference information about spatial characteristics. This information is frequently stored as coordinates named 'x' and 'y'. Treating spatial data using non-spatial data methods could lead to over-optimistic treatment. This is due to the underlying auto-correlation in spatial data.

See mlr3spatiotemporal for the current state of the implementation.

7.3 Ordinal Analysis

This is work in progress. See mlr3ordinal for the current state of the implementation.

7.4 Functional Analysis

Functional data is data containing an ordering on the dimensions. This implies that functional data consists of curves varying over a continuum, such as time, frequency, or wavelength.

7.4.1 How to model functional data?

There are two ways to model functional data:

- Modification of the learner, so that the learner is suitable for the functional data
- Modification of the task, so that the task matches the standard- or classification-learner

The development has not started yet, we are looking for contributers. Open an issue in mlr3fda if you are interested!

7.5 Multilabel Classification

Multilabel classification deals with objects that can belong to more than one category at the same time.

The development has not started yet, we are looking for contributers. Open an issue in mlr3multilabel if you are interested!

7.6 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. Cost sensitive classification does not assume that the costs caused by different kinds of errors are equal. The objective of cost sensitive classification is to minimize the expected costs.

Imagine you are an analyst for a big credit institution. Let's also assume that a correct decision of the bank would result in 35% of the profit at the end of a specific period. A correct decision means that the bank predicts that a customer will pay their bills (hence would obtain a loan), and the customer indeed has good credit. On the other hand, a wrong decision means that the bank predicts that the customer's credit is in good standing, but the opposite is true. This would result in a loss of 100% of the given loan.

	Good Customer (truth)	Bad Customer (truth)
Good Customer (predicted)	+ 0.35	- 1.0
Bad Customer (predicted)	0	0

Expressed as costs (instead of profit), we can write down the cost-matrix as follows:

```
costs = matrix(c(-0.35, 0, 1, 0), nrow = 2)
dimnames(costs) = list(response = c("good", "bad"), truth = c("good", "bad"))
print(costs)
```

```
##         truth
## response  good bad
##     good -0.35   1
##     bad   0.00   0
```

An exemplary data set for such a problem is the German Credit task:

```
library("mlr3")
task = tsk("german_credit")
table(task$truth())
```

```
##
## good  bad
##  700  300
```

The data has 70% customers who are able to pay back their credit, and 30% bad customers who default on the debt. A manager, who doesn't have any model, could decide to give either everybody a credit or to give nobody a credit. The resulting costs for the German credit data are:

```
# nobody:
(700 * costs[2, 1] + 300 * costs[2, 2]) / 1000
```

```
## [1] 0
```

```
# everybody
(700 * costs[1, 1] + 300 * costs[1, 2]) / 1000
```

```
## [1] 0.055
```

If the average loan is $20,000, the credit institute would lose more than one million dollar if it would grant everybody a credit:

```
# average profit * average loan * number of customers
0.055 * 20000 * 1000
```

```
## [1] 1100000
```

Our goal is to find a model which minimizes the costs (and thereby maximizes the expected profit).

7.6.1 A First Model

For our first model, we choose an ordinary logistic regression (implemented in the add-on package mlr3learners). We first create a classification task, then resample the model using a 10-fold cross validation and extract the resulting confusion matrix:

```
library("mlr3learners")
learner = lrn("classif.log_reg")
rr = resample(task, learner, rsmp("cv"))

confusion = rr$prediction()$confusion
print(confusion)
```

```
##         truth
## response good bad
##     good  602 156
##     bad    98 144
```

To calculate the average costs like above, we can simply multiply the elements of the confusion matrix with the elements of the previously introduced cost matrix, and sum the values of the resulting matrix:

```
avg_costs = sum(confusion * costs) / 1000
print(avg_costs)
```

```
## [1] -0.0547
```

With an average loan of $20,000, the logistic regression yields the following costs:

```
avg_costs * 20000 * 1000
```

```
## [1] -1094000
```

Instead of losing over $1,000,000, the credit institute now can expect a profit of more than $1,000,000.

7.6.2 Cost-sensitive Measure

Our natural next step would be to further improve the modeling step in order to maximize the profit. For this purpose we first create a cost-sensitive classification measure which calculates the costs based on our cost matrix. This allows us to conveniently quantify and compare modeling decisions. Fortunately, there already is a predefined measure `Measure` for this purpose: `MeasureClassifCosts`:

```
cost_measure = msr("classif.costs", costs = costs)
print(cost_measure)
```

```
## <MeasureClassifCosts:classif.costs>
## * Packages: -
## * Range: [-Inf, Inf]
## * Minimize: TRUE
## * Properties: requires_task
## * Predict type: response
```

If we now call resample() or benchmark(), the cost-sensitive measures will be evaluated. We compare the logistic regression to a simple featureless learner and to a random forest from package ranger :

```
learners = list(
  lrn("classif.log_reg"),
  lrn("classif.featureless"),
  lrn("classif.ranger")
)
cv3 = rsmp("cv", folds = 3)
bmr = benchmark(benchmark_grid(task, learners, cv3))
bmr$aggregate(cost_measure)
```

```
##    nr  resample_result       task_id          learner_id resampling_id iters
## 1:  1 <ResampleResult> german_credit     classif.log_reg            cv     3
## 2:  2 <ResampleResult> german_credit classif.featureless            cv     3
## 3:  3 <ResampleResult> german_credit      classif.ranger            cv     3
##    classif.costs
## 1:      -0.05444
## 2:       0.05503
## 3:      -0.04840
```

As expected, the featureless learner is performing comparably bad. The logistic regression and the random forest work equally well.

7.6.3 Thresholding

Although we now correctly evaluate the models in a cost-sensitive fashion, the models themselves are unaware of the classification costs. They assume the same costs for both wrong classification decisions (false positives and false negatives). Some learners natively support cost-sensitive classification (e.g., XXX). However, we will concentrate on a more generic approach which works for all models which can predict probabilities for class labels: thresholding.

Most learners can calculate the probability p for the positive class. If p exceeds the threshold 0.5, they predict the positive class, and the negative class otherwise.

For our binary classification case of the credit data, the we primarily want to minimize the errors where the model predicts "good", but truth is "bad" (i.e., the number of false positives) as this is the more expensive error. If we now increase the threshold to values > 0.5, we reduce the number of false negatives. Note that we increase the number of false positives simultaneously, or, in other words, we are trading false positives for false negatives.

```
# fit models with probability prediction
learner = lrn("classif.log_reg", predict_type = "prob")
rr = resample(task, learner, rsmp("cv"))
p = rr$prediction()
print(p)
```

```
## <PredictionClassif> for 1000 observations:
##     row_id truth response prob.good prob.bad
##         48  good     good    0.8709  0.12913
##         51  good     good    0.6125  0.38749
##         59  good      bad    0.4930  0.50702
## ---
##         984   bad     good    0.5522  0.44779
##         988  good     good    0.9363  0.06366
##         994  good      bad    0.3525  0.64752
```

```
# helper function to try different threshold values interactively
with_threshold = function(p, th) {
  p$set_threshold(th)
  list(confusion = p$confusion, costs = p$score(measures = cost_measure, task = task))
}

with_threshold(p, 0.5)
```

```
## $confusion
##         truth
## response good bad
##     good  609 153
##     bad    91 147
##
## $costs
## classif.costs
##      -0.06015
```

```
with_threshold(p, 0.75)
```

```
## $confusion
##         truth
## response good bad
##     good  471  73
##     bad   229 227
##
## $costs
## classif.costs
##      -0.09185
```

```
with_threshold(p, 1.0)
```

```
## $confusion
##         truth
## response good bad
##     good    2   0
##     bad   698 300
##
## $costs
## classif.costs
##        -7e-04
```

```
# TODO: include plot of threshold vs performance
```

Instead of manually trying different threshold values, one uses use `optimize()` to find a good threshold value w.r.t. our performance measure:

```
# simple wrapper function which takes a threshold and returns the resulting model performance
# this wrapper is passed to optimize() to find its minimum for thresholds in [0.5, 1]
```

```
f = function(th) {
  with_threshold(p, th)$costs
}
best = optimize(f, c(0.5, 1))
print(best)
```

```
## $minimum
## [1] 0.8014
##
## $objective
## classif.costs
##      -0.09475
```

```
# optimized confusion matrix:
with_threshold(p, best$minimum)$confusion
```

```
##         truth
## response good bad
##     good  425  54
##     bad   275 246
```

Note that the function optimize() is intended for unimodal functions and therefore may converge to a local optimum here. See below for better alternatives to find good threshold values.

7.6.4 Threshold Tuning

More following soon!

Upcoming sections will entail:

- threshold tuning as pipeline operator
- joint hyperparameter optimization

8 Model Interpretation

8.1 IML

8.2 Dalex

9 Appendix

9.1 Integrated Learners

```
## Loading required namespace: mlr3learners

## Loading required namespace: mlr3proba

## Warning: Package 'e1071' required but not installed for Learner
## 'classif.naive_bayes'

## Warning: Package 'e1071' required but not installed for Learner 'classif.svm'

## Warning: Package 'DiceKriging' required but not installed for Learner 'regr.km'

## Warning: Package 'e1071' required but not installed for Learner 'regr.svm'

## Warning: Package 'flexsurv' required but not installed for Learner
## 'surv.flexible'

## Warning: Package 'gbm' required but not installed for Learner 'surv.gbm'

## Warning: Package 'obliqueRSF' required but not installed for Learner
## 'surv.obliqueRSF'

## Warning: Package 'penalized' required but not installed for Learner
## 'surv.penalized'

## Warning: Package 'randomForestSRC' required but not installed for Learner
## 'surv.randomForestSRC'

## Warning: Package 'survivalsvm' required but not installed for Learner 'surv.svm'
```

Id	Feature Types	Required packages
classif.debug	lgl, int, dbl, chr, fct, ord	
classif.featureless	lgl, int, dbl, chr, fct, ord	
classif.glmnet	lgl, int, dbl	[glmnet](https://cran.r-project.org/package=glmnet)
classif.kknn	lgl, int, dbl, fct, ord	[kknn](https://cran.r-project.org/package=kknn)
classif.lda	lgl, int, dbl, fct, ord	[MASS](https://cran.r-project.org/package=MASS)
classif.log_reg	lgl, int, dbl, chr, fct, ord	[stats](https://cran.r-project.org/package=stats)
classif.naive_bayes	lgl, int, dbl, fct	[e1071](https://cran.r-project.org/package=e1071)
classif.qda	lgl, int, dbl, fct, ord	[MASS](https://cran.r-project.org/package=MASS)
classif.ranger	lgl, int, dbl, chr, fct, ord	[ranger](https://cran.r-project.org/package=ranger)
classif.rpart	lgl, int, dbl, fct, ord	[rpart](https://cran.r-project.org/package=rpart)
classif.svm	lgl, int, dbl	[e1071](https://cran.r-project.org/package=e1071)
classif.xgboost	lgl, int, dbl	[xgboost](https://cran.r-project.org/package=xgboost)
regr.featureless	lgl, int, dbl, chr, fct, ord	[stats](https://cran.r-project.org/package=stats)
regr.glmnet	lgl, int, dbl	[glmnet](https://cran.r-project.org/package=glmnet)
regr.kknn	lgl, int, dbl, fct, ord	[kknn](https://cran.r-project.org/package=kknn)
regr.km	lgl, int, dbl	[DiceKriging](https://cran.r-project.org/package=DiceKr
regr.lm	lgl, int, dbl, fct	[stats](https://cran.r-project.org/package=stats)
regr.ranger	lgl, int, dbl, chr, fct, ord	[ranger](https://cran.r-project.org/package=ranger)
regr.rpart	lgl, int, dbl, fct, ord	[rpart](https://cran.r-project.org/package=rpart)
regr.svm	lgl, int, dbl	[e1071](https://cran.r-project.org/package=e1071)
regr.xgboost	lgl, int, dbl	[xgboost](https://cran.r-project.org/package=xgboost)
surv.blackboost	int, dbl, fct	[distr6](https://cran.r-project.org/package=distr6), [mbo
surv.coxph	lgl, int, dbl, fct	[distr6](https://cran.r-project.org/package=distr6), [survi
surv.cvglmnet	int, dbl, fct	[glmnet](https://cran.r-project.org/package=glmnet), [su
surv.flexible	lgl, int, fct, dbl	[distr6](https://cran.r-project.org/package=distr6), [flexs
surv.gamboost	int, dbl, fct, lgl	[distr6](https://cran.r-project.org/package=distr6), [mbo
surv.gbm	int, dbl, fct, ord	[gbm](https://cran.r-project.org/package=gbm)
surv.glmboost	int, dbl, fct, lgl	[distr6](https://cran.r-project.org/package=distr6), [mbo
surv.glmnet	int, dbl, fct	[glmnet](https://cran.r-project.org/package=glmnet), [su
surv.kaplan	lgl, int, dbl, chr, fct, ord	[distr6](https://cran.r-project.org/package=distr6), [survi
surv.mboost	int, dbl, fct, lgl	[distr6](https://cran.r-project.org/package=distr6), [mbo
surv.nelson	lgl, int, dbl, chr, fct, ord	[distr6](https://cran.r-project.org/package=distr6), [survi
surv.obliqueRSF	int, dbl, fct, ord	[distr6](https://cran.r-project.org/package=distr6), [obli
surv.parametric	lgl, int, dbl, fct	[distr6](https://cran.r-project.org/package=distr6), [set6]
surv.penalized	int, dbl, fct, ord	[distr6](https://cran.r-project.org/package=distr6), [pena
surv.randomForestSRC	lgl, int, dbl, fct, ord	[distr6](https://cran.r-project.org/package=distr6), [rand
surv.ranger	lgl, int, dbl, chr, fct, ord	[distr6](https://cran.r-project.org/package=distr6), [rang
surv.rpart	lgl, int, dbl, chr, fct, ord	[distr6](https://cran.r-project.org/package=distr6), [rpar
surv.svm	int, dbl	[survivalsvm](https://cran.r-project.org/package=surviva

9.2 Integrated Performance Measures

Also see the overview on the website of mlr3measures.

Id	Task Type	Required packages	Task Pr
classif.acc	classif	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	
classif.auc	classif	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	twoclass
classif.bacc	classif	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	
classif.ce	classif	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	
classif.costs	classif		
classif.dor	classif	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	twoclass
classif.fbeta	classif	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	twoclass
classif.fdr	classif	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	twoclass
classif.fn	classif	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	twoclass
classif.fnr	classif	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	twoclass
classif.fomr	classif	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	twoclass
classif.fp	classif	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	twoclass
classif.fpr	classif	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	twoclass
classif.logloss	classif	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	
classif.mcc	classif	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	twoclass
classif.npv	classif	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	twoclass
classif.ppv	classif	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	twoclass
classif.precision	classif	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	twoclass
classif.recall	classif	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	twoclass
classif.sensitivity	classif	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	twoclass
classif.specificity	classif	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	twoclass
classif.tn	classif	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	twoclass
classif.tnr	classif	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	twoclass
classif.tp	classif	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	twoclass
classif.tpr	classif	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	twoclass
debug	NA		
oob_error	NA		
regr.bias	regr	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	
regr.ktau	regr	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	
regr.mae	regr	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	
regr.mape	regr	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	
regr.maxae	regr	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	
regr.medae	regr	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	
regr.medse	regr	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	
regr.mse	regr	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	
regr.msle	regr	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	
regr.pbias	regr	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	
regr.rae	regr	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	
regr.rmse	regr	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	
regr.rmsle	regr	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	
regr.rrse	regr	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	
regr.rse	regr	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	
regr.rsq	regr	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	
regr.sae	regr	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	
regr.smape	regr	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	
regr.srho	regr	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	
regr.sse	regr	[mlr3measures](https://cran.r-project.org/package=mlr3measures)	
selected_features	NA		
time_both	NA		
time_predict	NA		
time_train	NA		

9.3 Integrated Filter Methods

9.3.1 Standalone filter methods

Name	Task	task_properties	param_set	Features
carscore	Regr	character(0)	<environment>	numeric
correlation	Regr	character(0)	<environment>	Integer, Numeric
cmim	Classif & Regr	character(0)	<environment>	Integer, Numeric, Factor, Orc
information_gain	Classif & Regr	character(0)	<environment>	Integer, Numeric, Factor, Orc
mrmr	Classif & Regr	character(0)	<environment>	c(”numeric”, ”factor”, ”integ
variance	Classif & Regr	character(0)	<environment>	Integer, Numeric
anova	Classif	character(0)	<environment>	Integer, Numeric
auc	Classif	twoclass	<environment>	Integer, Numeric
disr	Classif	character(0)	<environment>	Integer, Numeric, Factor, Orc
importance	Classif	character(0)	<environment>	c(”logical”, ”integer”, ”nume
jmi	Classif	character(0)	<environment>	Integer, Numeric, Factor, Orc
jmim	Classif	character(0)	<environment>	Integer, Numeric, Factor, Orc
kruskal_test	Classif	character(0)	<environment>	Integer, Numeric
mim	Classif	character(0)	<environment>	Integer, Numeric, Factor, Orc
njmim	Classif	character(0)	<environment>	Integer, Numeric, Factor, Orc
performance	Classif	character(0)	<environment>	c(”logical”, ”integer”, ”nume

9.3.2 Algorithms With Embedded Filter Methods

```
## [1] "classif.featureless" "classif.rpart"       "regr.featureless"
## [4] "regr.rpart"
```

9.4 Integrated PipeOps

key	packages	input.num	output.num	input.train	input.predict	output.
boxcox	bestNormalize	1	1	Task	Task	Task
branch		1	–	Any	Any	Any
chunk		1	–	Task	Task	Task
classbalancing		1	1	TaskClassif	TaskClassif	TaskCla
classifavg	stats	–	1	–	PredictionClassif	–
classweights		1	1	TaskClassif	TaskClassif	TaskCla
colapply		1	1	Task	Task	Task
collapsefactors		1	1	Task	Task	Task
copy		1	–	Any	Any	Any
crankcompose	distr6	1	1	–	PredictionSurv	–
distrcompose	distr6	2	1	–	PredictionSurv	–
encode	stats	1	1	Task	Task	Task
encodeimpact		1	1	Task	Task	Task
encodelmer	lme4 | nloptr	1	1	Task	Task	Task
featureunion		–	1	Task	Task	Task
filter		1	1	Task	Task	Task
fixfactors		1	1	Task	Task	Task
histbin	graphics	1	1	Task	Task	Task
ica	fastICA	1	1	Task	Task	Task
imputehist	graphics	1	1	Task	Task	Task
imputemean		1	1	Task	Task	Task
imputemedian	stats	1	1	Task	Task	Task
imputenewlvl		1	1	Task	Task	Task
imputesample		1	1	Task	Task	Task
kernelpca	kernlab	1	1	Task	Task	Task
learner		1	1	TaskClassif	TaskClassif	–
learner_cv		1	1	TaskClassif	TaskClassif	TaskCla
missind		1	1	Task	Task	Task
modelmatrix	stats	1	1	Task	Task	Task
mutate		1	1	Task	Task	Task
nop		1	1	Any	Any	Any
pca		1	1	Task	Task	Task
quantilebin	stats	1	1	Task	Task	Task
regravg		–	1	–	PredictionRegr	–
removeconstants		1	1	Task	Task	Task
scale		1	1	Task	Task	Task
scalemaxabs		1	1	Task	Task	Task
scalerange		1	1	Task	Task	Task
select		1	1	Task	Task	Task
smote	smotefamily	1	1	Task	Task	Task
spatialsign		1	1	Task	Task	Task
subsample		1	1	Task	Task	Task
unbranch		–	1	Any	Any	Any
yeojohnson	bestNormalize	1	1	Task	Task	Task

9.5 Framework Comparison

Below, we collected some examples, where **mlr3pipelines** is compared to different other software packages, such as **mlr**, **recipes** and **sklearn**.

Before diving deeper, we give a short introduction to PipeOps.

9.5.1 An introduction to "PipeOp"s

In this example, we create a linear Pipeline. After scaling all input features, we rotate our data using principal component analysis. After this transformation, we use a simple Decision Tree learner for classification.

As exemplary data, we will use the "`iris`" classification task. This object contains the famous iris dataset and some meta-information, such as the target variable.

```
library("mlr3")
task = mlr_tasks$get("iris")
```

We quickly split our data into a train and a test set:

```
test.idx = sample(seq_len(task$nrow), 30)
train.idx = setdiff(seq_len(task$nrow), test.idx)
# Set task to only use train indexes
task$row_roles$use = train.idx
```

A Pipeline (or `Graph`) contains multiple pipeline operators ("`PipeOp`"s), where each `PipeOp` transforms the data when it flows through it. For this use case, we require 3 transformations:

- A `PipeOp` that scales the data
- A `PipeOp` that performs PCA
- A `PipeOp` that contains the **Decision Tree** learner

A list of available `PipeOps` can be obtained from

```
library("mlr3pipelines")
mlr_pipeops$keys()
```

```
##  [1] "boxcox"          "branch"          "chunk"           "classbalancing"
##  [5] "classifavg"      "classweights"    "colapply"        "collapsefactors"
##  [9] "copy"            "encode"          "encodeimpact"    "encodelmer"
## [13] "featureunion"    "filter"          "fixfactors"      "histbin"
## [17] "ica"             "imputehist"      "imputemean"      "imputemedian"
## [21] "imputenewlvl"    "imputesample"    "kernelpca"       "learner"
## [25] "learner_cv"      "missind"         "modelmatrix"     "mutate"
## [29] "nop"             "pca"             "quantilebin"     "regravg"
## [33] "removeconstants" "scale"           "scalemaxabs"     "scalerange"
## [37] "select"          "smote"           "spatialsign"     "subsample"
## [41] "unbranch"        "yeojohnson"
```

First we define the required `PipeOps`:

```
op1 = PipeOpScale$new()
op2 = PipeOpPCA$new()
op3 = PipeOpLearner$new(learner = mlr_learners$get("classif.rpart"))
```

9.5.1.1 A quick glance into a PipeOp

In order to get a better understanding of what the respective PipeOps do, we quickly look at one of them in detail:

The most important slots in a PipeOp are:

- `$train()`: A function used to train the PipeOp.
- `$predict()`: A function used to predict with the PipeOp.

The `$train()` and `$predict()` functions define the core functionality of our PipeOp. In many cases, in order to not leak information from the training set into the test set it is imperative to treat train and test data separately. For this we require a `$train()` function that learns the appropriate transformations from the training set and a `$predict()` function that applies the transformation on future data.

In the case of `PipeOpPCA` this means the following:

- `$train()` learns a rotation matrix from its input and saves this matrix to an additional slot, `$state`. It returns the rotated input data stored in a new `Task`.
- `$predict()` uses the rotation matrix stored in `$state` in order to rotate future, unseen data. It returns those in a new `Task`.

9.5.1.2 Constructing the Pipeline

We can now connect the `PipeOps` constructed earlier to a **Pipeline**. We can do this using the `%>>%` operator.

```
linear_pipeline = op1 %>>% op2 %>>% op3
```

The result of this operation is a "`Graph`". A `Graph` connects the input and output of each `PipeOp` to the following `PipeOp`. This allows us to specify linear processing pipelines. In this case, we connect the output of the **scaling** PipeOp to the input of the **PCA** PipeOp and the output of the **PCA** PipeOp to the input of **PipeOpLearner**.

We can now train the `Graph` using the `iris` `Task`.

```
linear_pipeline$train(task)
```

```
## $classif.rpart.output
## NULL
```

When we now train the graph, the data flows through the graph as follows:

- The Task flows into the `PipeOpScale`. The `PipeOp` scales each column in the data contained in the Task and returns a new Task that contains the scaled data to its output.
- The scaled Task flows into the `PipeOpPCA`. PCA transforms the data and returns a (possibly smaller) Task, that contains the transformed data.
- This transformed data then flows into the learner, in our case **classif.rpart**. It is then used to train the learner, and as a result saves a model that can be used to predict new data.

In order to predict on new data, we need to save the relevant transformations our data went through while training. As a result, each `PipeOp` saves a state, where information required to appropriately transform future data is stored. In our case, this is **mean** and **standard deviation** of each column for `PipeOpScale`, the PCA rotation matrix for `PipeOpPCA` and the learned model for `PipeOpLearner`.

```
# predict on test.idx
task$row_roles$use = test.idx
linear_pipeline$predict(task)
```

```
## $classif.rpart.output
## <PredictionClassif> for 30 observations:
##     row_id     truth  response
##          4    setosa    setosa
##         13    setosa    setosa
##         14    setosa    setosa
## ---
##        136 virginica virginica
##        141 virginica virginica
##        148 virginica virginica
```

9.5.2 mlr3pipelines vs. mlr

In order to showcase the benefits of mlr3pipelines over mlr's `Wrapper` mechanism, we compare the case of imputing missing values before filtering the top 2 features and then applying a learner.

While mlr wrappers are generally less verbose and require a little less code, this heavily inhibits flexibility. As an example, wrappers can generally not process data in parallel.

9.5.2.1 mlr

```
library("mlr")
# We first create a learner
lrn = makeLearner("classif.rpart")
# Wrap this learner in a FilterWrapper
lrn.wrp = makeFilterWrapper(lrn, fw.abs = 2L)
# And wrap the resulting wrapped learner into an ImputeWrapper.
lrn.wrp = makeImputeWrapper(lrn.wrp)

# Afterwards, we can train the resulting learner on a task
train(lrn, iris.task)
```

9.5.2.2 mlr3pipelines

```
library("mlr3")
library("mlr3pipelines")
library("mlr3filters")

impute = PipeOpImpute$new()
filter = PipeOpFilter$new(filter = FilterVariance$new(), param_vals = list(filter.nfeat = 2L))
rpart = PipeOpLearner$new(mlr_learners$get("classif.rpart"))

# Assemble the Pipeline
```

```
pipeline = impute %>>% filter %>>% rpart
# And convert to a 'GraphLearner'
learner = GraphLearner$new(id = "Pipeline", pipeline)
```

The fact that **mlr's** wrappers have to be applied inside-out, i.e. in the reverse order is often confusing. This is way more straight-forward in `mlr3pipelines`, where we simply chain the different methods using `%>>%`. Additionally, `mlr3pipelines` offers way greater possibilities with respect to the kinds of Pipelines that can be constructed. In `mlr3pipelines`, we allow for the construction of parallel and conditional pipelines. This was previously not possible.

9.5.3 mlr3pipelines vs. sklearn.pipeline.Pipeline

In order to broaden the horizon, we compare to **Python** sklearn's `Pipeline` methods. `sklearn.pipeline.Pipeline` sequentially applies a list of transforms before fitting a final estimator. Intermediate steps of the pipeline are `transforms`, i.e. steps that can learn from the data, but also transform the data while it flows through it. The purpose of the pipeline is to assemble several steps that can be cross-validated together while setting different parameters. For this, it enables setting parameters of the various steps.

It is thus conceptually very similar to mlr3pipelines. Similarly to mlr3pipelines, we can tune over a full `Pipeline` using various tuning methods. `Pipeline` mainly supports linear pipelines. This means, that it can execute parallel steps, such as for example **Bagging**, but it does not support conditional execution, i.e. `PipeOpBranch`. At the same time, the different `transforms` in the pipeline can be cached, which makes tuning over the configuration space of a `Pipeline` more efficient, as executing some steps multiple times can be avoided.

We compare functionality available in both mlr3pipelines and `sklearn.pipeline.Pipeline` to give a comparison.

The following example obtained from the sklearn documentation showcases a **Pipeline** that first Selects a feature and performs PCA on the original data, concatenates the resulting datasets and applies a Support Vector Machine.

9.5.3.1 sklearn

```
from sklearn.pipeline import Pipeline, FeatureUnion
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
from sklearn.datasets import load_iris
from sklearn.decomposition import PCA
from sklearn.feature_selection import SelectKBest

iris = load_iris()

X, y = iris.data, iris.target

# This dataset is way too high-dimensional. Better do PCA:
pca = PCA(n_components=2)

# Maybe some original features where good, too?
selection = SelectKBest(k=1)
```

```
# Build estimator from PCA and Univariate selection:
combined_features = FeatureUnion([("pca", pca), ("univ_select", selection)])

# Use combined features to transform dataset:
X_features = combined_features.fit(X, y).transform(X)

svm = SVC(kernel="linear")

# Do grid search over k, n_components and C:
pipeline = Pipeline([("features", combined_features), ("svm", svm)])

param_grid = dict(features__pca__n_components=[1, 2, 3],
                  features__univ_select__k=[1, 2],
                  svm__C=[0.1, 1, 10])

grid_search = GridSearchCV(pipeline, param_grid=param_grid, cv=5, verbose=10)
grid_search.fit(X, y)
```

9.5.3.2 mlr3pipelines

```
library("mlr3verse")
iris = mlr_tasks$get("iris")

# Build the steps
copy = PipeOpCopy$new(2)
pca = PipeOpPCA$new()
selection = PipeOpFilter$new(filter = FilterVariance$new())
union = PipeOpFeatureUnion$new(2)
svm = PipeOpLearner$new(mlr_learners$get("classif.svm", param_vals = list(kernel = "linear")))

# Assemble the Pipeline
pipeline = copy %>>% gunion(list(pca, selection)) %>>% union %>>% svm
learner = GraphLearner$new(id = "Pipeline", pipeline)

# For tuning, we define the resampling and the Parameter Space
resampling = mlr3::mlr_resamplings$get("cv", param_vals = list(folds = 5L))

param_set = paradox::ParamSet$new(params = list(
  paradox::ParamDbl$new("classif.svm.cost", lower = 0.1, upper = 1),
  paradox::ParamInt$new("pca.rank.",  lower = 1, upper = 3),
  paradox::ParamInt$new("variance.filter.nfeat",  lower = 1, upper = 2)
))

pe = PerformanceEvaluator$new(iris, learner, resampling, param_set)
terminator = TerminatorEvaluations$new(60)
tuner = TunerGridSearch$new(pe, terminator, resolution = 10)$tune()

# Set the learner to the optimal values and train
learner$param_set$values = tuner$tune_result()$values
```

In summary, we can achieve similar results with a comparable number of lines, while at the same time offering greater flexibility with respect to which kinds of pipelines we want to optimize over. At the same time, experiments using `mlr3` can now be arbitrarily parallelized using `futures`.

9.5.4 mlr3pipelines vs recipes

recipes is a new package, that covers some of the same applications steps as mlr3pipelines. Both packages feature the possibility to connect different pre- and post-processing methods using a pipe-operator. As the recipes package tightly integrates with the tidymodels ecosystem, much of the functionality integrated there can be used in `recipes`. We compare recipes to mlr3pipelines using an example from the recipes vignette.

The aim of the analysis is to predict whether customers pay back their loans given some information on the customers. In order to do this, we build a model that does the following:

1. It first imputes missing values using k-nearest neighbors
2. All factor variables are converted to numerics using dummy encoding
3. The data is first centered then scaled.

In order to validate the algorithm, data is first split into a train and test set using `initial_split`, `training`, `testing`. The recipe trained on the train data (see steps above) is then applied to the test data.

9.5.4.1 recipes

```r
library("tidymodels")
library("rsample")
data("credit_data", package = "modeldata")

set.seed(55)
train_test_split = initial_split(credit_data)
credit_train = training(train_test_split)
credit_test = testing(train_test_split)

rec = recipe(Status ~ ., data = credit_train) %>%
  step_knnimpute(all_predictors()) %>%
  step_dummy(all_predictors(), -all_numeric()) %>%
  step_center(all_numeric()) %>%
  step_scale(all_numeric())

trained_rec = prep(rec, training = credit_train)

# Apply to train and test set
train_data <- bake(trained_rec, new_data = credit_train)
test_data  <- bake(trained_rec, new_data = credit_test)
```

Afterwards, the transformed data can be used during train and predict:

```r
# Train
rf = rand_forest(mtry = 12, trees = 200, mode = "classification") %>%
  set_engine("ranger", importance = 'impurity') %>%
  fit(Status ~ ., data = train_data)

# Predict
prds = predict(rf, test_data)
```

9.5.4.2 mlr3pipelines

The same analysis can be performed in mlr3pipelines. Note, that for now we do not impute via `knn` but instead via sampling.

```r
library("data.table")
library("mlr3")
library("mlr3learners")
library("mlr3pipelines")
data("credit_data", package = "modeldata")
set.seed(55)

# Create the task
tsk = TaskClassif$new(id = "credit_task", target = "Status",
  backend = as_data_backend(data.table(credit_data)))

# Build up the Pipeline:
g = PipeOpImputeSample$new(id = "impute") %>>%
  PipeOpEncode$new(param_vals = list(method = "one-hot")) %>>%
  PipeOpScale$new() %>>%
  PipeOpLearner$new(mlr_learners$get("classif.ranger",
    param_vals = list(num.trees = 200, mtry = 12))

# We can visualize what happens to the data using the `plot` function:
g$plot()

# And we can use `mlr3's` full functionality be wrapping the Graph into a GraphLearner.
glrn = GraphLearner$new(g)
resample(tsk, glrn, mlr_resamplings$get("holdout"))
```

References

Bergstra, James, and Yoshua Bengio. 2012. "Random Search for Hyper-Parameter Optimization." *J. Mach. Learn. Res.* 13. JMLR.org: 281–305.

Bischl, Bernd, Michel Lang, Lars Kotthoff, Julia Schiffner, Jakob Richter, Erich Studerus, Giuseppe Casalicchio, and Zachary M. Jones. 2016. "mlr: Machine Learning in R." *Journal of Machine Learning Research* 17 (170): 1–5. http://jmlr.org/papers/v17/15-066.html.

Breiman, Leo. 1996. "Bagging Predictors." *Machine Learning* 24 (2). Springer: 123–40.

Chandrashekar, Girish, and Ferat Sahin. 2014. "A Survey on Feature Selection Methods." *Computers and Electrical Engineering* 40 (1): 16–28. https://doi.org/https://doi.org/10.1016/j.compeleceng.2013.11.024.

Lang, Michel. 2017. "checkmate: Fast Argument Checks for Defensive R Programming." *The R Journal* 9 (1): 437–45. https://doi.org/10.32614/RJ-2017-028.

Lang, Michel, Martin Binder, Jakob Richter, Patrick Schratz, Florian Pfisterer, Stefan Coors, Quay Au, Giuseppe Casalicchio, Lars Kotthoff, and Bernd Bischl. 2019. "mlr3: A Modern Object-Oriented Machine Learning Framework in R." *Journal of Open Source Software*, December. https://doi.org/10.21105/joss.01903.

R Core Team. 2019. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. https://www.R-project.org/.

Wolpert, David H. 1992. "Stacked Generalization." *Neural Networks* 5 (2): 241–59. https://doi.org/https://doi.org/10.1016/S0893-6080(05)80023-1.