mlr3 book

Marc Becker Martin Binder Bernd Bischl Lars Kotthoff
Michel Lang Florian Pfisterer Nicholas G. Reich Jakob Richter
Patrick Schratz Raphael Sonabend Damir Pulatov

2021-11-24

Contents

Quickstart

If you haven't installed mlr3 already, do so now:

```
install.packages("mlr3")
```

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_ids     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.

We highly recommend to keep some of our cheatsheets handy while learning mlr3.

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 and feature selection. Parallelization of many operations is natively supported.

Target Audience

We expect that users of mlr3 have at least basic knowledge of machine learning and R. The later chapters of this book describe advanced functionality that requires more advanced knowledge of both. mlr3 is suitable for complex projects that use advanced functionality as well as one-liners to quickly prototype specific tasks.

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.

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 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 (especially R6, future, and data.table) makes mlr3 much easier to use, maintain, and more efficient compared to its predecessor 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.tables`. This considerably simplifies the API and allows easy selection and "split-apply-combine" (aggregation) 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.

Package Ecosystem

mlr3 builds upon the following packages not developed by core members of 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.

All these packages are well curated and mature; we expect no problems with dependencies. Additionally, we suggest the following packages for extra functionality:

- For parallelization: future / future.apply.
- For progress bars: progressr.
- For capturing output, warnings, and exceptions: evaluate or callr.

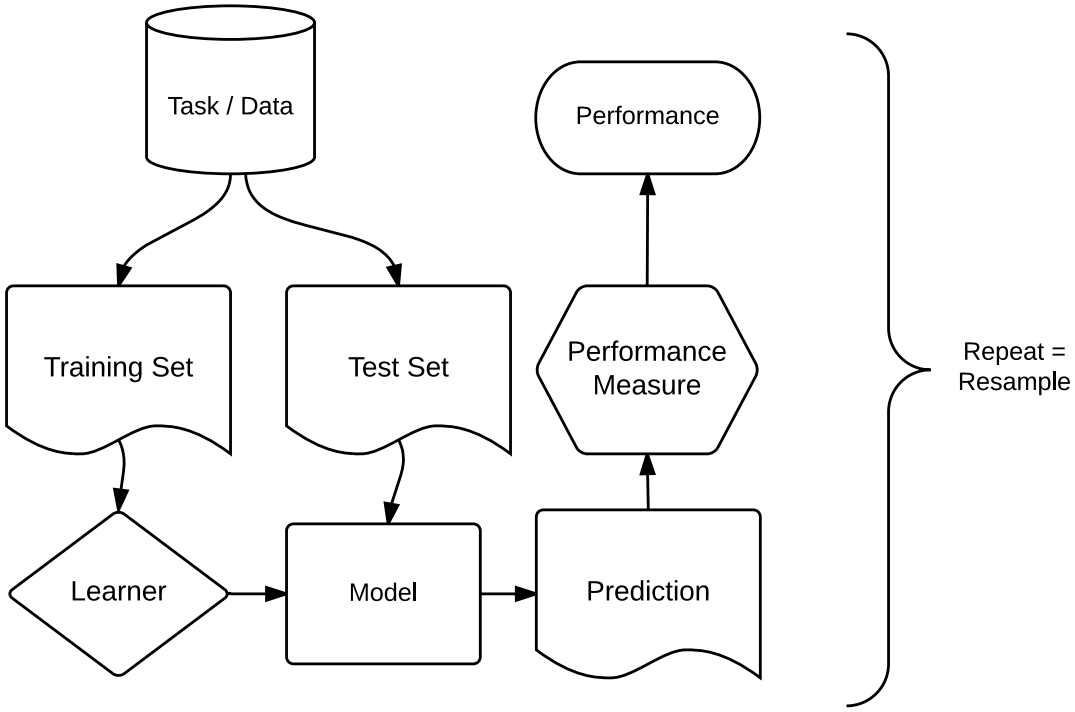
The mlr3 package itself provides the base functionality that the rest of ecosystem rely on and some of the fundamental building blocks for machine learning. The following packages extend mlr3 with capabilities for preprocessing, pipelining, visualizations, additional learners, additional task types, and more:

To view the mlr3verse image for an overview of the mlr3 package ecosystem, follow this link: `https://raw.githubusercontent.com/mlr-org/mlr3/master/man/figures/mlr3verse.svg`.

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

2 Basics

This chapter will teach you the essential building blocks of mlr3, as well as its R6 classes and operations used 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. As we are interested in models that extrapolate to new data rather than just memorizing the training data, the separate test data allows to objectively evaluate models with respect to generalization. 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 can 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 and less biased generalizable performance estimate for a particular type of model. As data are usually partitioned randomly into training and test sets, a single split can for example produce trainng and test sets that are

very different, hence creating to the misleading impression that the particular type of model does not perform well.

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 topics:

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. These are provided other packages. We cover how to:

- access the set of classification and regression learners that come with mlr3 and retrieve a specific learner (more types of learners are covered later in the book),
- access the set of hyperparameter values of a learner and modify them.

How to extend learners and implement your own 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:

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

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 `R6::R6Class()` object, specifically the initialization method `$new()`. For example, `foo = Foo$new(bar = 1)` creates a new object of class `Foo`, setting the `bar` argument of the constructor to the value `1`. Most objects in mlr3 are created through special functions (e.g. `lrn("regr.rpart")`) that encapsulate this and are also referred to as *sugar functions*.
- Objects have mutable state that is encapsulated in their fields, which can be accessed through the dollar operator. We can access the `bar` value in the `foo` variable from above through `foo$bar` and set its value by assigning the field, e.g. `foo$bar = 2`.
- In addition to fields, objects expose methods that allow to inspect the object's state, retrieve information, or perform an action that changes the internal state of the object. For example, the `$train` method of a learner changes the internal state of the learner by building and storing a trained model, which can then be used to make predictions, given data.
- Objects can have public and private fields and methods. The public fields and methods define the API to interact with the object. Private methods are only relevant for you if you want to extend mlr3, e.g. with new learners.
- R6 objects are internally environments, and as such have reference semantics. For example, `foo2 = foo` does not create a copy of `foo` in `foo2`, but another reference to the same actual object. Setting `foo$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 more details on R6, have a look at the excellent R6 vignettes, especially the introduction.

2.2 Tasks

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

2.2.1 Task Types

To create a task from a `data.frame()`, `data.table()` or `Matrix()`, you first need to select the right task type:

- **Classification Task**: The target is a label (stored as `character()`or`factor()`) with only relatively few distinct values → `TaskClassif`.
- **Regression Task**: The target is a numeric quantity (stored as `integer()` or `double()`) → `TaskRegr`.

- **Survival Task**: The target is the (right-censored) time to an event. More censoring types are currently in development → `mlr3proba::TaskSurv` in add-on package mlr3proba.
- **Density Task**: An unsupervised task to estimate the density → `mlr3proba::TaskDens` in add-on package mlr3proba.
- **Cluster Task**: An unsupervised task type; there is no target and the aim is to identify similar groups within the feature space → `mlr3cluster::TaskClust` in add-on package mlr3cluster.
- **Spatial Task**: Observations in the task have spatio-temporal information (e.g. coordinates) → `mlr3spatiotempcv::TaskRegrST` or `mlr3spatiotempcv::TaskClassifST` in add-on package mlr3spatiotempcv.
- **Ordinal Regression Task**: The target is ordinal → `TaskOrdinal` in add-on package mlr3ordinal (still in development).

2.2.2 Task Creation

As an example, we will create a regression task using the `mtcars` data set from the package `datasets`. It contains characteristics for different types of cars, along with their fuel consumption. We predict the numeric target variable `"mpg"` (miles per gallon). We only consider the first two features in the dataset for brevity.

First, we load and prepare the data, outputting it as a string to get a better idea of what it looks like.

```
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 a regression task, i.e. we construct a new instance of the R6 class `TaskRegr`. Usually, this is done by calling the constructor `TaskRegr$new()`. Instead, we are calling the converter `as_task_regr()` to convert our `data.frame()` stored in the variable `data` to a task and provide the following information:

1. `x`: Object to convert. Works for `data.frame()`/`data.table()`/`tibble()` abstract data backends implemented in the class `DataBackendDataTable`. The latter allows to connect to out-of-memory storage systems like SQL servers via the extension package mlr3db.
2. `target`: The name of the prediction target column for the regression problem.
3. `id` (optional): An arbitrary identifier for the task, used in plots and summaries. If not provided, the deparsed name of `x` will be used.

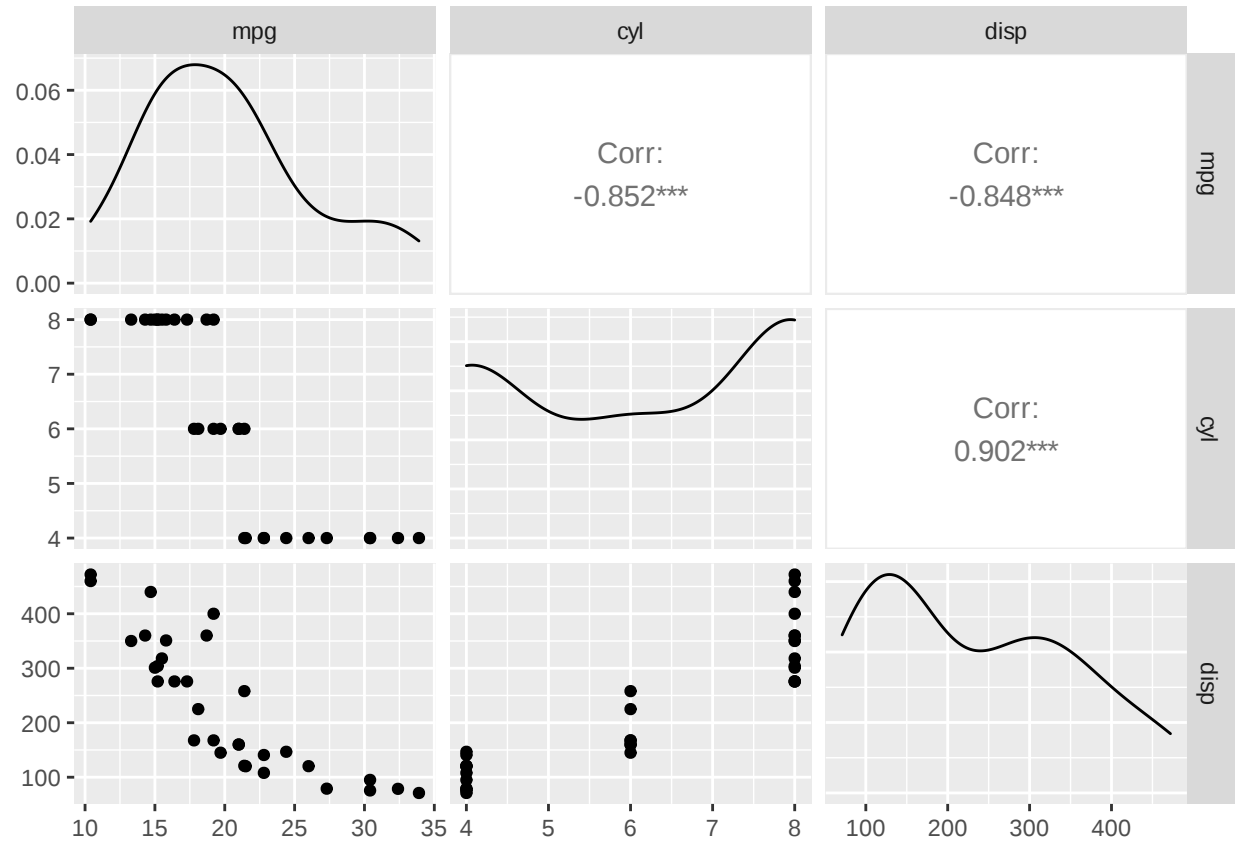
```
library("mlr3")

task_mtcars = as_task_regr(data, target = "mpg", id = "cars")
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")
```



Note that instead of loading all the extension packages individually, it is often more convenient to load the mlr3verse package instead. `mlr3verse` imports most `mlr3` packages and re-exports functions which are used for common machine learning and data science tasks.

2.2.3 Predefined tasks

mlr3 includes 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 11 stored values
## Keys: boston_housing, breast_cancer, german_credit, iris, mtcars,
##   penguins, 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:

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

```
##                key task_type nrow ncol properties lgl int dbl chr fct ord pxc
##  1: boston_housing      regr  506   19             0   3  13   0   2   0   0
##  2:  breast_cancer   classif  683   10   twoclass  0   0   0   0   0   9   0
##  3:  german_credit   classif 1000   21   twoclass  0   3   0   0  14   3   0
##  4:           iris   classif  150    5 multiclass  0   0   4   0   0   0   0
##  5:         mtcars      regr   32   11             0   0  10   0   0   0   0
##  6:       penguins   classif  344    8 multiclass  0   3   2   0   2   0   0
##  7:           pima   classif  768    9   twoclass  0   0   8   0   0   0   0
##  8:          sonar   classif  208   61   twoclass  0   0  60   0   0   0   0
##  9:           spam   classif 4601   58   twoclass  0   0  57   0   0   0   0
## 10:           wine   classif  178   14 multiclass  0   2  11   0   0   0   0
## 11:            zoo   classif  101   17 multiclass 15   1   0   0   0   0   0
```

Above, the columns `"lgl"` (`logical`), `"int"` (`integer`), `"dbl"` (`double`), `"chr"` (`character`), `"fct"` (`factor`), `"ord"` (`ordered factor`) and `"pxc"` (`POSIXct` time) show the number of features in the task of the respective type.

To get a task from the dictionary, use the `$get()` method from the `mlr_tasks` class and assign the return value to a new variable As getting a task from a dictionary is a very common problem, `mlr3` provides the shortcut function `tsk()`. Here, we retrieve the `palmer penguins task`, which is provided by the package palmerpenguins:

```
task_penguins = tsk("penguins")
print(task_penguins)
```

```
## <TaskClassif:penguins> (344 x 8)
## * Target: species
## * Properties: multiclass
## * Features (7):
##   - int (3): body_mass, flipper_length, year
##   - dbl (2): bill_depth, bill_length
##   - fct (2): island, sex
```

Note that loading extension packages can add to dictionaries such as `mlr_tasks`. For example, mlr3data adds some more example and toy tasks for regression and classification, and mlr3proba adds survival and density estimation tasks. Both packages are loaded automatically when the mlr3verse package is loaded:

```
library("mlr3verse")
as.data.table(mlr_tasks)[, 1:4]
```

```
##                key task_type  nrow ncol
##  1:           actg      surv  1151   13
##  2:   bike_sharing      regr 17379   14
##  3: boston_housing      regr   506   19
##  4:  breast_cancer   classif   683   10
##  5:       faithful      dens   272    1
##  6:           gbcs      surv   686   10
##  7:  german_credit   classif  1000   21
##  8:          grace      surv  1000    8
##  9:           ilpd   classif   583   11
## 10:           iris   classif   150    5
## 11:     kc_housing      regr 21613   20
## 12:           lung      surv   228   10
## 13:      moneyball      regr  1232   15
## 14:         mtcars      regr    32   11
## 15:      optdigits   classif  5620   65
## 16:       penguins   classif   344    8
## 17:           pima   classif   768    9
## 18:         precip      dens    70    1
## 19:           rats      surv   300    5
## 20:          sonar   classif   208   61
## 21:           spam   classif  4601   58
## 22:        titanic   classif  1309   11
## 23:   unemployment      surv  3343    6
## 24:      usarrests     clust    50    4
## 25:           whas      surv   481   11
## 26:           wine   classif   178   14
## 27:            zoo   classif   101   17
##                key task_type  nrow ncol
```

To get more information about a particular task, the corresponding man page can be found under `mlr_tasks_[id]`, e.g. `mlr_tasks_german_credit`:

```
help("mlr_tasks_german_credit")
```

2.2.4 Task API

All properties and characteristics of tasks can be queried using the task's public fields and methods (see `Task`). Methods can also be used to change the stored data and 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 for `task_mtcars` that we defined above we can get the number of rows and columns:

```
task_mtcars
```

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

```
task_mtcars$nrow
```

```
## [1] 32
```

```
task_mtcars$ncol
```

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

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

```
task_mtcars$data()
```

```
##      mpg cyl  disp
##  1: 21.0   6 160.0
##  2: 21.0   6 160.0
##  3: 22.8   4 108.0
##  4: 21.4   6 258.0
##  5: 18.7   8 360.0
##  6: 18.1   6 225.0
##  7: 14.3   8 360.0
##  8: 24.4   4 146.7
##  9: 22.8   4 140.8
## 10: 19.2   6 167.6
## 11: 17.8   6 167.6
## 12: 16.4   8 275.8
## 13: 17.3   8 275.8
## 14: 15.2   8 275.8
## 15: 10.4   8 472.0
## 16: 10.4   8 460.0
## 17: 14.7   8 440.0
## 18: 32.4   4  78.7
## 19: 30.4   4  75.7
## 20: 33.9   4  71.1
## 21: 21.5   4 120.1
## 22: 15.5   8 318.0
## 23: 15.2   8 304.0
## 24: 13.3   8 350.0
## 25: 19.2   8 400.0
```

```
## 26: 27.3   4  79.0
## 27: 26.0   4 120.3
## 28: 30.4   4  95.1
## 29: 15.8   8 351.0
## 30: 19.7   6 145.0
## 31: 15.0   8 301.0
## 32: 21.4   4 121.0
##      mpg cyl  disp
```

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_mtcars$row_ids)
```

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

```
# retrieve data for rows with IDs 1, 5, and 10
task_mtcars$data(rows = c(1, 5, 10))
```

```
##     mpg cyl  disp
## 1: 21.0   6 160.0
## 2: 18.7   8 360.0
## 3: 19.2   6 167.6
```

Note that although the row IDs are typically just the sequence from `1` to `nrow(data)`, they are only guaranteed to be unique natural numbers. Keep that in mind, especially if you work with data stored in a real data base management system (see backends).

Similarly to row IDs, target and feature columns also have unique identifiers, i.e. names (stored as `character()`). Their names can be accessed via 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. The target will usually be only a single name.

```
task_mtcars$feature_names
```

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

```
task_mtcars$target_names
```

```
## [1] "mpg"
```

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

```
# retrieve data for rows 1, 5, and 10 and only select column "mpg"
task_mtcars$data(rows = c(1, 5, 10), cols = "mpg")
```

```
##     mpg
## 1: 21.0
## 2: 18.7
## 3: 19.2
```

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

```
# show summary of entire data
summary(as.data.table(task_mtcars))
```

```
##       mpg             cyl             disp
##  Min.   :10.4   Min.   :4.00   Min.   : 71.1
##  1st Qu.:15.4   1st Qu.:4.00   1st Qu.:120.8
##  Median :19.2   Median :6.00   Median :196.3
##  Mean   :20.1   Mean   :6.19   Mean   :230.7
##  3rd Qu.:22.8   3rd Qu.:8.00   3rd Qu.:326.0
##  Max.   :33.9   Max.   :8.00   Max.   :472.0
```

2.2.4.2 Binary classification

Classification problems with a target variable with only two classes are called binary classification tasks. They are special in the sense that one of these classes is denoted *positive* and the other one *negative*. 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 = as_task_classif(Sonar, target = "Class", positive = "R")

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

2.2.4.3 Roles (Rows and Columns)

We have seen that during task creation, target and feature roles are assigned to columns. Target refers to the variable we want to predict and features are the predictors (also called co-variates) for the target. It is possible to assign different roles to rows and columns. These roles affect the behavior of the task for different operations. For other possible roles and their meaning, see the documentation of `Task`.

For example, the previously-constructed `task_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)
```

Columns can have no role (they are ignored) or have multiple roles. To add the row names of `task_mtcars` as an additional feature, we first add them to the underlying data as regular column and then recreate the task with the new column.

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

# there is a new feature called "rn"
task_mtcars$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, for example to identify outliers. To achieve this, we will change the role of the `rn` column by removing it from the list of features and assign the new role `"name"`. There are two ways to do this:

1. Use the `Task` method `$set_col_roles()` (recommended).
2. Simply modify the field `$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 have that role.

Supported column roles can be found in the manual of `Task`, or just by printing the names of the field `$col_roles`:

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

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

```
# assign column "rn" the role "name", remove from other roles
task_mtcars$set_col_roles("rn", roles = "name")

# note that "rn" not listed as feature anymore
task_mtcars$feature_names
```

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

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

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

Changing the role does not change the underlying data, it just updates 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.

2.2.4.4 Task Mutators

As shown above, modifying `$col_roles` or `$row_roles` (either via `set_col_roles()`/`set_row_roles()` or directly by modifying the named list) 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_penguins = tsk("penguins")
task_penguins$select(c("body_mass", "flipper_length")) # keep only these features
task_penguins$filter(1:3) # keep only these rows
task_penguins$head()
```

```
##    species body_mass flipper_length
## 1:  Adelie      3750            181
## 2:  Adelie      3800            186
## 3:  Adelie      3250            195
```

While the methods above allow us 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_penguins$cbind(data.frame(letters = letters[1:3])) # add column letters
task_penguins$head()
```

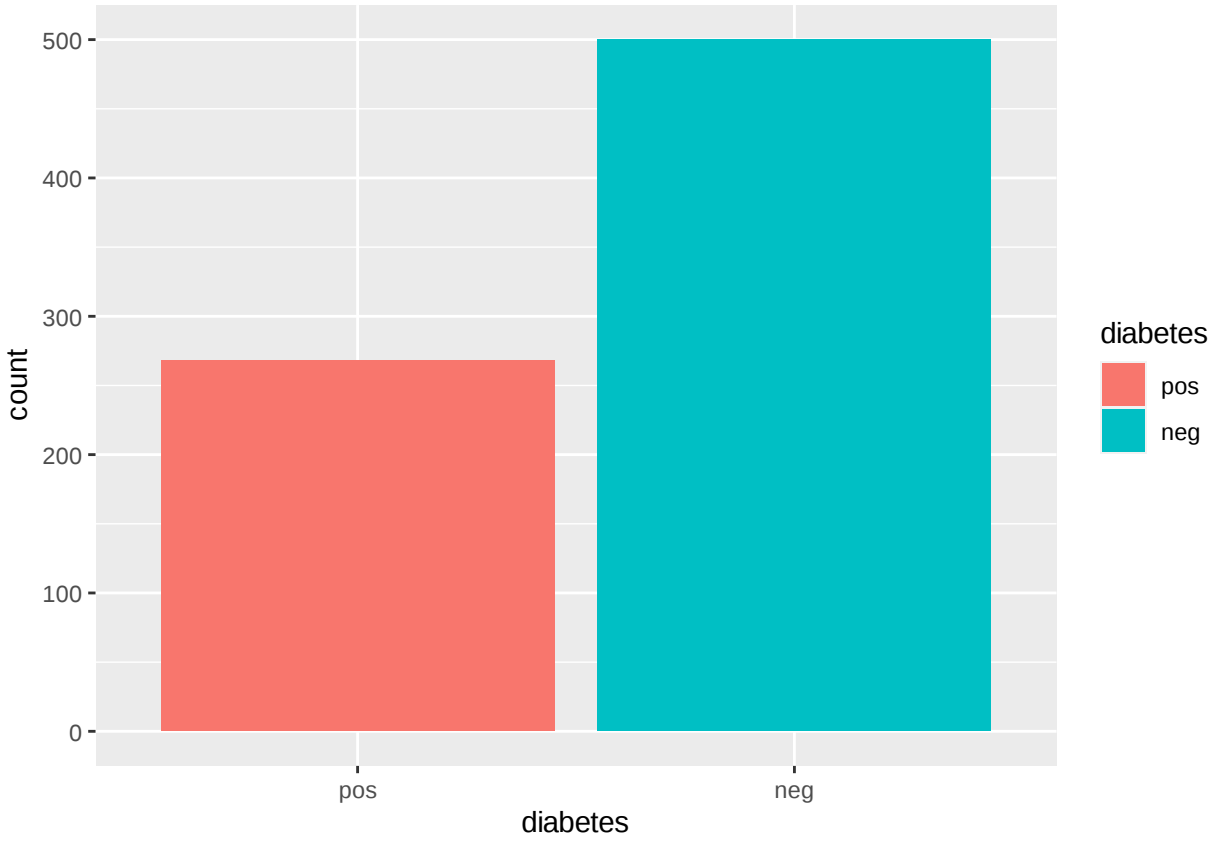
```
##    species body_mass flipper_length letters
## 1:  Adelie      3750            181       a
## 2:  Adelie      3800            186       b
## 3:  Adelie      3250            195       c
```

2.2.5 Plotting Tasks

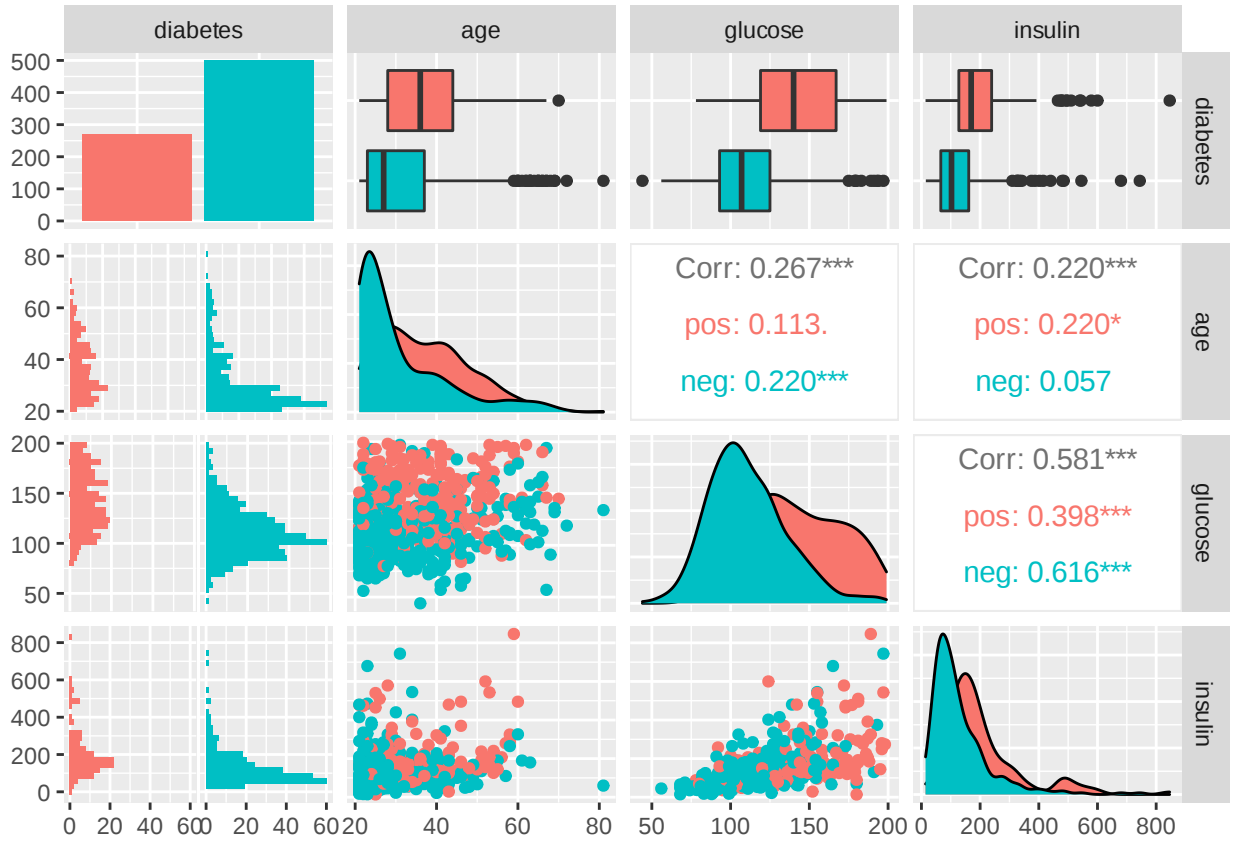
The mlr3viz package provides plotting facilities for many classes implemented in mlr3. The available plot types depend on the class, but all plots are returned as ggplot2 objects which can be easily customized.

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:

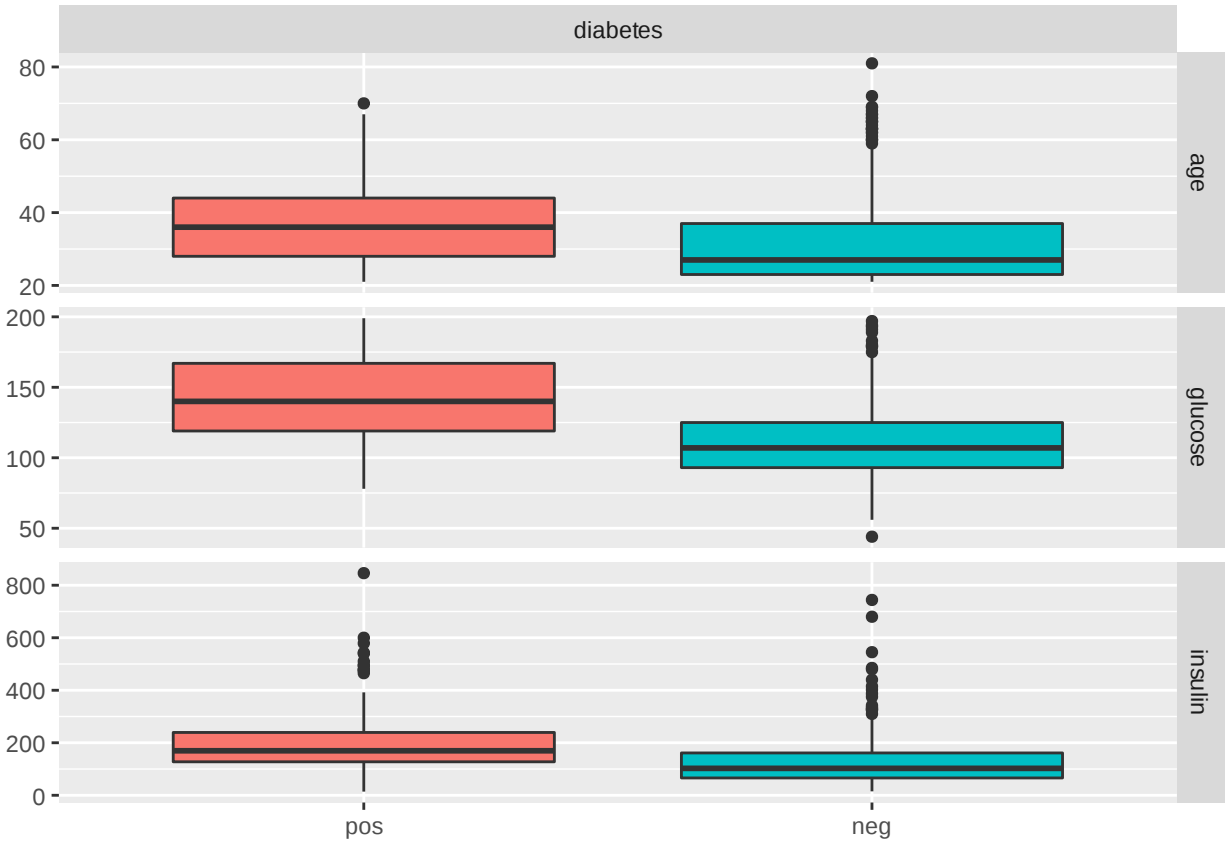
```
library("mlr3viz")

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

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

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




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

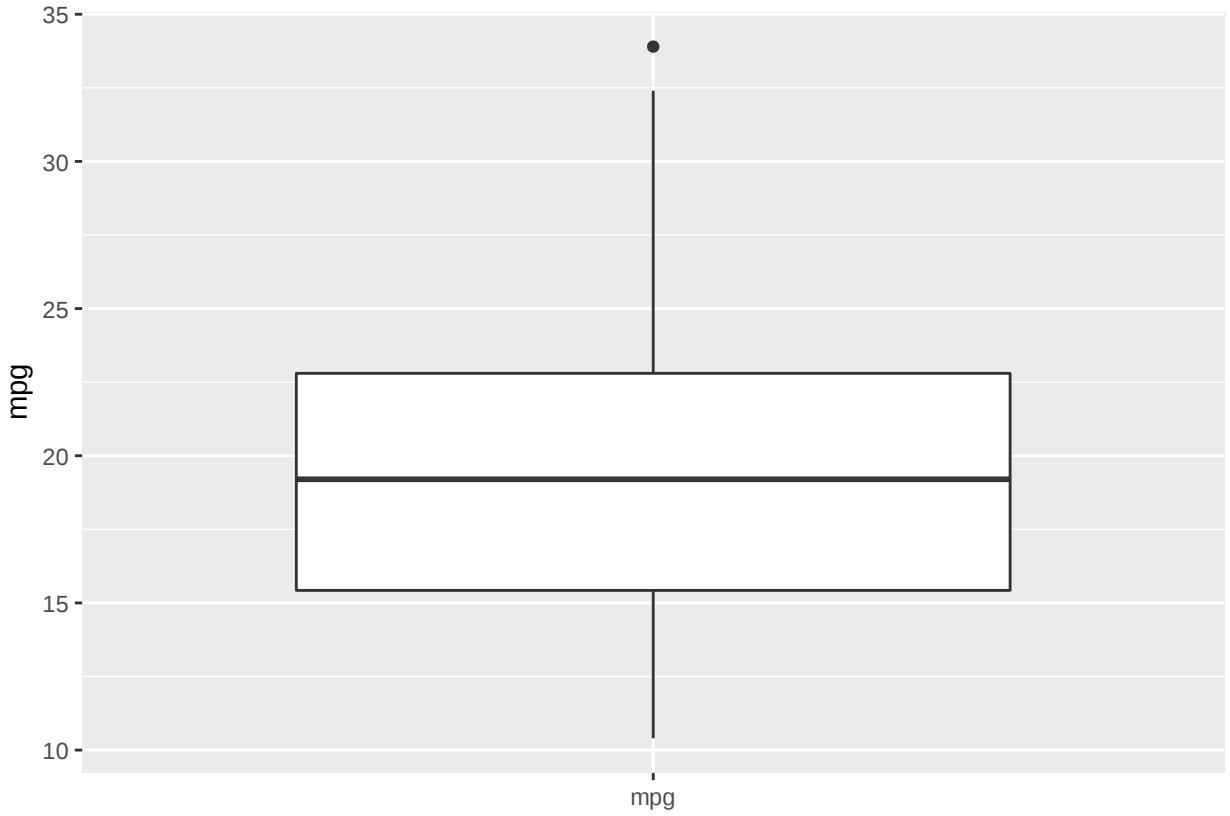



```
# duo plot (requires package GGally)
autoplot(task, type = "duo")
```

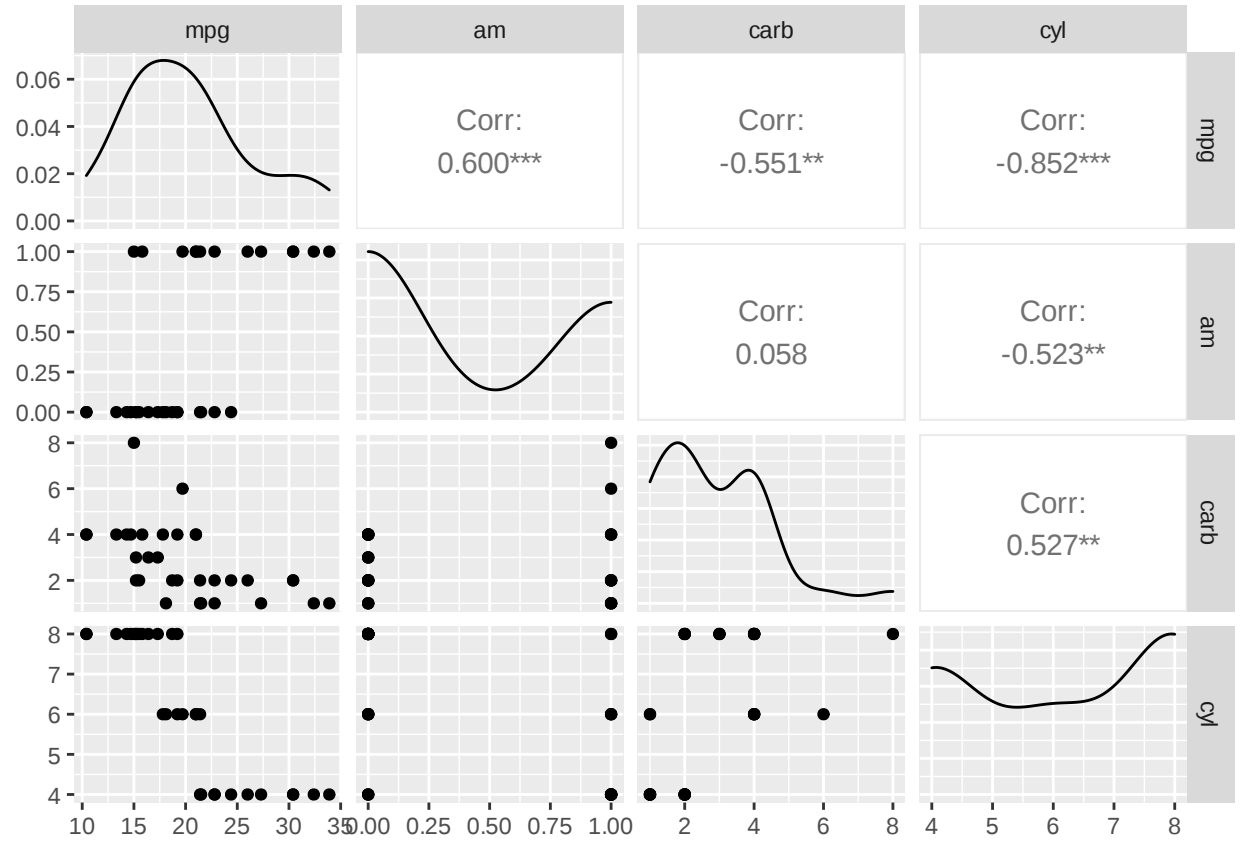


Of course, you can do the same for regression tasks (inheriting from `TaskRegr`) as documented in `mlr3viz::autoplot.TaskRegr`:

```
library("mlr3viz")

# get the complete 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)
```




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

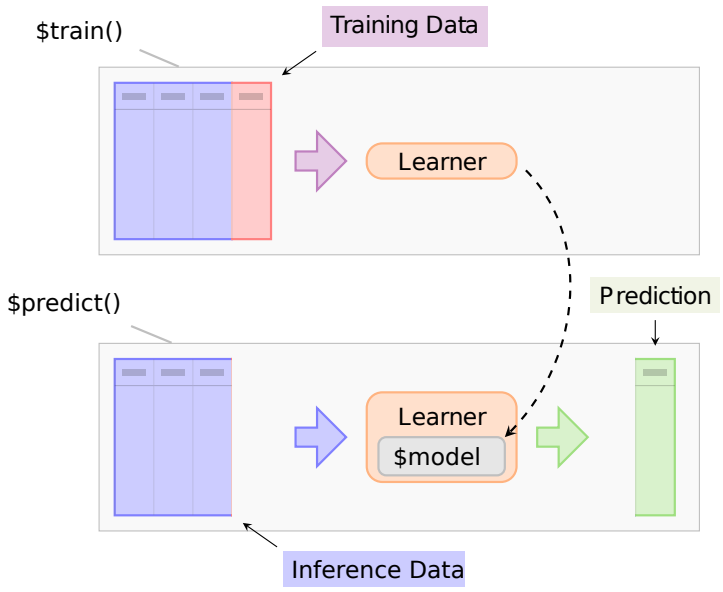


2.3 Learners

Objects of class `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 `Task` and provide meta-information about the learners, such as the hyperparameters (which control the behavior of the learner) you can set.

The base class of each learner is `Learner`, specialized for regression as `LearnerRegr` and for classification as `LearnerClassif`. Other types of learners, provided by extension packages, also inherit from the `Learner` base class, e.g. `mlr3proba::LearnerSurv` or `mlr3cluster::LearnerClust`. In contrast to `Task`s, the creation of a custom Learner is usually not required and a more advanced topic. Hence, we refer the reader to Section 7.1 and proceed with an overview of the interface of already implemented learners.

All Learners work in a two-stage procedure:



- **training step**: The training data (features and target) is passed to the Learner's `$train()` function which trains and stores a model, i.e. the relationship of the target and features.
- **predict step**: The new data, usually a different slice of the original data than used for training, is passed to the `$predict()` method of the Learner. The model trained in the first step is used to predict the missing target, e.g. labels for classification problems or the numerical value for regression problems.

2.3.1 Predefined Learners

The `mlr3` package ships with the following set of classification and regression learners. We deliberately keep this small to avoid unnecessary dependencies:

- `mlr_learners_classif.featureless`: Simple baseline classification learner (inheriting from `LearnerClassif`). The default is to predict the label that is most frequent in the training set every time.
- `mlr_learners_regr.featureless`: Simple baseline regression learner (inheriting from `LearnerRegr`). The default is to predict the mean of the target in training set every time.
- `mlr_learners_classif.rpart`: Single classification tree from package rpart.
- `mlr_learners_regr.rpart`: Single regression tree from package rpart.

This set of baseline learners is usually insufficient for a real data analysis. Thus, we have cherry-picked implementations of the most popular machine learning method and collected them in the mlr3learners package:

- Linear and logistic regression
- Penalized Generalized Linear Models
- k-Nearest Neighbors regression and classification
- Kriging
- Linear and Quadratic Discriminant Analysis
- Naive Bayes
- Support-Vector machines
- Gradient Boosting
- Random Forests for regression, classification and survival

More machine learning methods and alternative implementations are collected in the mlr3extralearners repository.

A full list of implemented learners across all packages is given in this interactive list and also via `mlr3extralearners::list_mlr3learners()`.

```
head(mlr3extralearners::list_mlr3learners()) # show first six learners
```

```
##          name   class                 id      mlr3_package
## 1: AdaBoostM1 classif classif.AdaBoostM1 mlr3extralearners
## 2:       bart classif       classif.bart mlr3extralearners
## 3:        C50 classif        classif.C50 mlr3extralearners
## 4:   catboost classif   classif.catboost mlr3extralearners
## 5:    cforest classif    classif.cforest mlr3extralearners
## 6:      ctree classif      classif.ctree mlr3extralearners
##                           required_packages
## 1:                 mlr3extralearners,RWeka
## 2:                mlr3extralearners,dbarts
## 3:                   mlr3extralearners,C50
## 4:               mlr3extralearners,catboost
## 5: mlr3extralearners,partykit,sandwich,coin
## 6: mlr3extralearners,partykit,sandwich,coin
##                                       properties
## 1:                             multiclass,twoclass
## 2:                                twoclass,weights
## 3:            missings,multiclass,twoclass,weights
## 4: importance,missings,multiclass,twoclass,weights
## 5:           multiclass,oob_error,twoclass,weights
## 6:                     multiclass,twoclass,weights
##                     feature_types predict_types
## 1:         numeric,factor,ordered response,prob
## 2: integer,numeric,factor,ordered response,prob
## 3:         numeric,factor,ordered response,prob
## 4:         numeric,factor,ordered response,prob
## 5: integer,numeric,factor,ordered response,prob
## 6: integer,numeric,factor,ordered response,prob
```

The full list of learners uses a large number of extra packages, which sometimes break. We check the status of each learner's integration automatically, the latest build status of all learners is shown here.

To get one of the predefined learners, you need to access the `mlr_learners` `Dictionary` which, similar to `mlr_tasks`, is automatically populated with more learners by extension packages.

```
# load most mlr3 packages to populate the dictionary
library("mlr3verse")
mlr_learners
```

```
## <DictionaryLearner> with 136 stored values
## Keys: classif.AdaBoostM1, classif.bart, classif.C50, classif.catboost,
##   classif.cforest, classif.ctree, classif.cv_glmnet, classif.debug,
##   classif.earth, classif.extratrees, classif.featureless, classif.fnn,
##   classif.gam, classif.gamboost, classif.gausspr, classif.gbm,
##   classif.glmboost, classif.glmnet, classif.IBk, classif.J48,
##   classif.JRip, classif.kknn, classif.ksvm, classif.lda,
##   classif.liblinear, classif.lightgbm, classif.LMT, classif.log_reg,
##   classif.lssvm, classif.mob, classif.multinom, classif.naive_bayes,
##   classif.nnet, classif.OneR, classif.PART, classif.qda,
##   classif.randomForest, classif.ranger, classif.rfsrc, classif.rpart,
##   classif.svm, classif.xgboost, clust.agnes, clust.ap, clust.cmeans,
##   clust.cobweb, clust.dbscan, clust.diana, clust.em, clust.fanny,
##   clust.featureless, clust.ff, clust.hclust, clust.kkmeans,
##   clust.kmeans, clust.MBatchKMeans, clust.meanshift, clust.pam,
##   clust.SimpleKMeans, clust.xmeans, dens.hist, dens.kde, dens.kde_kd,
##   dens.kde_ks, dens.locfit, dens.logspline, dens.mixed, dens.nonpar,
##   dens.pen, dens.plug, dens.spline, regr.bart, regr.catboost,
##   regr.cforest, regr.ctree, regr.cubist, regr.cv_glmnet, regr.debug,
##   regr.earth, regr.extratrees, regr.featureless, regr.fnn, regr.gam,
##   regr.gamboost, regr.gausspr, regr.gbm, regr.glm, regr.glmboost,
##   regr.glmnet, regr.IBk, regr.kknn, regr.km, regr.ksvm, regr.liblinear,
##   regr.lightgbm, regr.lm, regr.M5Rules, regr.mars, regr.mob,
##   regr.randomForest, regr.ranger, regr.rfsrc, regr.rpart, regr.rvm,
##   regr.svm, regr.xgboost, surv.akritas, surv.blackboost, surv.cforest,
##   surv.coxboost, surv.coxph, surv.coxtime, surv.ctree,
##   surv.cv_coxboost, surv.cv_glmnet, surv.deephit, surv.deepsurv,
##   surv.dnnsurv, surv.flexible, surv.gamboost, surv.gbm, surv.glmboost,
##   surv.glmnet, surv.kaplan, surv.loghaz, surv.mboost, surv.nelson,
##   surv.obliqueRSF, surv.parametric, surv.pchazard, surv.penalized,
##   surv.ranger, surv.rfsrc, surv.rpart, surv.svm, surv.xgboost
```

To obtain an object from the dictionary you can also use the shortcut function `lrn()` or the generic `mlr_learners$get()` method, e.g. `lrn("classif.rpart")`.

2.3.2 Learner API

Each learner provides the following meta-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.
- `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.

You can retrieve a specific learner using its ID:

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

```
## <LearnerClassifRpart:classif.rpart>
## * Model: -
## * Parameters: xval=0
## * Packages: mlr3, rpart
## * Predict Type: response
## * Feature types: logical, integer, numeric, factor, ordered
## * Properties: importance, missings, multiclass, selected_features,
##   twoclass, weights
```

Each learner has hyperparameters that control its behavior, for example the minimum number of samples in the leaf of a decision tree, or whether to provide verbose output durning training. Setting hyperparameters to values appropriate for a given machine learning task is crucial. The field `param_set` stores a description of the hyperparameters the learner has, their ranges, defaults, and current values:

```
learner$param_set
```

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

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

```
## <LearnerClassifRpart:classif.rpart>
## * Model: -
## * Parameters: cp=0.01, xval=0
## * Packages: mlr3, rpart
## * Predict Type: response
## * Feature types: logical, integer, numeric, factor, ordered
## * Properties: importance, missings, multiclass, selected_features,
##   twoclass, weights
```

Note that this operation overwrites all previously set parameters. You can also get the current set of hyperparameter values, modify it, and write it back to the learner:

```
pv = learner$param_set$values
pv$cp = 0.02
learner$param_set$values = pv
```

This sets `cp` to `0.02` but keeps any other values that were set previously.

Note that the `lrn()` function also accepts additional arguments to update hyperparameters or set fields of the learner in one go:

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

```
## [1] "rp"
```

```
learner$param_set$values
```

```
## $xval
## [1] 0
##
## $cp
## [1] 0.001
```

More on this is discussed in the section on Hyperparameter Tuning.

2.3.2.1 Thresholding

Models trained on binary classification tasks that predict the probability for the positive class usually use a simple rule to determine the predicted class label: if the probability is more than 50%, predict the positive label, otherwise predict the negative label. In some cases you may want to adjust this threshold, for example if the classes are very unbalanced (i.e. one is much more prevalent than the other).

In the example below, we change the threshold to 0.2, which improves the True Positive Rate (TPR). Note that while the new threshold classifies more observations from the positive class correctly, the True Negative Rate (TNR) decreases. Depending on the application, this may or may not be desired.

```
data("Sonar", package = "mlbench")
task = as_task_classif(Sonar, target = "Class", positive = "M")
learner = lrn("classif.rpart", predict_type = "prob")
pred = learner$train(task)$predict(task)

measures = msrs(c("classif.tpr", "classif.tnr")) # use msrs() to get a list of multiple measures
pred$confusion
```

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

```
pred$score(measures)
```

```
## classif.tpr classif.tnr
##      0.8559      0.8969
```

```
pred$set_threshold(0.2)
pred$confusion
```

```
##          truth
## response   M   R
##        M 104  25
##        R   7  72
```

```
pred$score(measures)
```

```
## classif.tpr classif.tnr
##      0.9369      0.7423
```

Thresholds can be tuned automatically with respect to a performance measure with the mlr3pipelines package, i.e. using `PipeOpTuneThreshold`.

2.4 Train, Predict, Assess Performance

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

Training a learner means fitting a model to a given data set – essentially an optimization problem that determines the best parameters (not hyperparameters!) of the model given the data. We then predict the label for observations that the model has not seen during training. 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

First of all, we load the mlr3verse package, which will load all other packages we need here.

```
library("mlr3verse")
```

Now, we retrieve the task and the learner from `mlr_tasks` (with shortcut `tsk()`) and `mlr_learners` (with shortcut `lrn()`), respectively:

```
task = tsk("penguins")
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, to give the learner a better chance of fitting a good model. Here we use 80% of all available observations to train 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 3.2 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
```

Now we fit the classification tree using the training set of the task by calling the `$train()` method of `learner`:

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

This operation modifies the learner in-place by adding the fitted model to the existing object. We can now access the stored model via the field `$model`:

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

```
## n= 275
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
## 1) root 275 156 Adelie (0.432727 0.192727 0.374545)
##   2) flipper_length< 207 169  52 Adelie (0.692308 0.301775 0.005917)
##     4) bill_length< 43.35 118   4 Adelie (0.966102 0.033898 0.000000) *
##     5) bill_length>=43.35 51   4 Chinstrap (0.058824 0.921569 0.019608) *
##   3) flipper_length>=207 106   4 Gentoo (0.018868 0.018868 0.962264) *
```

Inspecting the output, we see that the learner has identified features in the task that are predictive of the class (the type of penguin) and uses them to partition observations in the tree. There are additional details on how the data is partitioned across branches of the tree; the textual representation of the model depends on the type of learner. For more information on this particular type of model, see `rpart::print.rpart()`.

2.4.4 Predicting

After the model has been fitted to the training data, we use the test set 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 69 observations:
##     row_ids     truth  response
##           3    Adelie    Adelie
##           4    Adelie    Adelie
##          14    Adelie    Adelie
## ---
##         339 Chinstrap Chinstrap
##         340 Chinstrap    Gentoo
##         343 Chinstrap    Gentoo
```

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)) # show first six predictions
```

```
##    row_ids  truth response
## 1:       3 Adelie   Adelie
## 2:       4 Adelie   Adelie
## 3:      14 Adelie   Adelie
## 4:      21 Adelie   Adelie
## 5:      22 Adelie   Adelie
## 6:      30 Adelie   Adelie
```

For classification, you can also extract the confusion matrix:

```
prediction$confusion
```

```
##            truth
## response    Adelie Chinstrap Gentoo
##   Adelie        32         1      0
##   Chinstrap      1        11      0
##   Gentoo         0         3     21
```

The confusion matrix shows, for each class, how many observations were predicted to be in that class and how many were actually in it (more information on Wikipedia). The entries along the diagonal denote the correctly classified observations. In this case, we can see that our classifier is really quite good and correctly predicting almost all observations.

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 for the classes. To predict these probabilities, the `predict_type` field of a `LearnerClassif` must be changed from `"response"` (the default) 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 in addition to the predicted label (the one with the highest probability):

```
# data.table conversion
head(as.data.table(prediction)) # show first six
```

```
##    row_ids  truth response prob.Adelie prob.Chinstrap prob.Gentoo
## 1:       3 Adelie   Adelie      0.9661         0.0339           0
## 2:       4 Adelie   Adelie      0.9661         0.0339           0
## 3:      14 Adelie   Adelie      0.9661         0.0339           0
## 4:      21 Adelie   Adelie      0.9661         0.0339           0
## 5:      22 Adelie   Adelie      0.9661         0.0339           0
## 6:      30 Adelie   Adelie      0.9661         0.0339           0
```

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

```
## [1] Adelie Adelie Adelie Adelie Adelie Adelie
## Levels: Adelie Chinstrap Gentoo
```

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

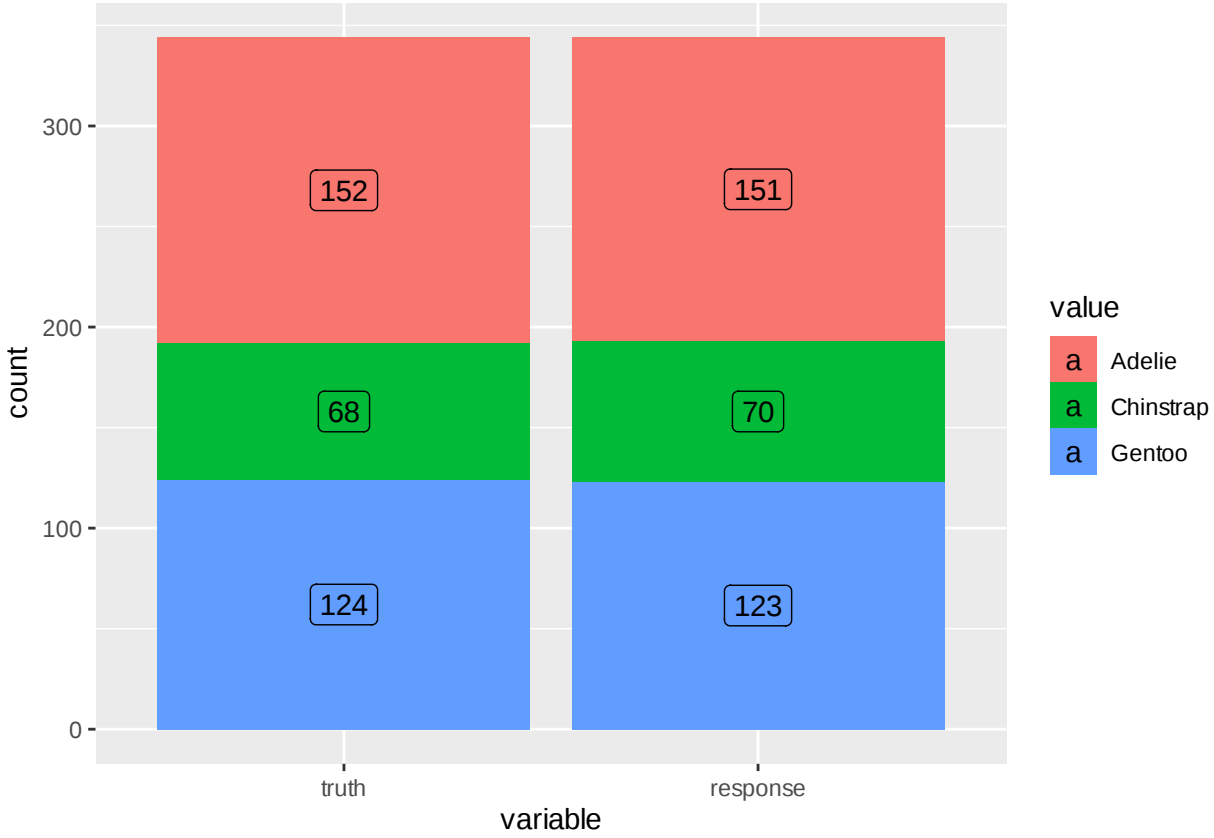
```
##      Adelie Chinstrap Gentoo
## [1,] 0.9661    0.0339      0
## [2,] 0.9661    0.0339      0
## [3,] 0.9661    0.0339      0
## [4,] 0.9661    0.0339      0
## [5,] 0.9661    0.0339      0
## [6,] 0.9661    0.0339      0
```

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

2.4.6 Plotting Predictions

Similarly to plotting tasks, mlr3viz provides an `autoplot()` method for `Prediction` objects. All available types are listed in the manual pages for `autoplot.PredictionClassif()`, `autoplot.PredictionRegr()` and the other prediction types (defined by extension packages).

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



2.4.7 Performance assessment

The last step of modeling is usually assessing the performance of the trained model. We have already had a look at this with the confusion matrix, but it is often convenient to quantify the performance of a model with a single number. The exact nature of this comparison 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 known true labels, then no performance can be calculated.

Available measures are stored in `mlr_measures` (with convenience getter function `msr()`):

```
mlr_measures
```

```
## <DictionaryMeasure> with 87 stored values
## Keys: aic, bic, classif.acc, classif.auc, classif.bacc, classif.bbrier,
##   classif.ce, classif.costs, classif.dor, classif.fbeta, classif.fdr,
##   classif.fn, classif.fnr, classif.fomr, classif.fp, classif.fpr,
##   classif.logloss, classif.mbrier, classif.mcc, classif.npv,
##   classif.ppv, classif.prauc, classif.precision, classif.recall,
##   classif.sensitivity, classif.specificity, classif.tn, classif.tnr,
##   classif.tp, classif.tpr, clust.ch, clust.db, clust.dunn,
##   clust.silhouette, clust.wss, debug, dens.logloss, 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, sim.jaccard, sim.phi,
##   surv.brier, surv.calib_alpha, surv.calib_beta, surv.chambless_auc,
##   surv.cindex, surv.dcalib, surv.graf, surv.hung_auc, surv.intlogloss,
##   surv.logloss, surv.mae, surv.mse, surv.nagelk_r2, surv.oquigley_r2,
##   surv.rmse, surv.schmid, surv.song_auc, surv.song_tnr, surv.song_tpr,
##   surv.uno_auc, surv.uno_tnr, surv.uno_tpr, surv.xu_r2, time_both,
##   time_predict, time_train
```

We choose accuracy (`classif.acc`) as our specific performance measure here and call the method `$score()` of the `prediction` object to quantify the predictive performance of our model.

```
measure = msr("classif.acc")
print(measure)
```

```
## <MeasureClassifSimple:classif.acc>
## * Packages: mlr3, mlr3measures
## * Range: [0, 1]
## * Minimize: FALSE
## * Average: macro
## * Parameters: list()
## * Properties: -
## * Predict type: response
```

```
prediction$score(measure)
```

```
## classif.acc
##      0.9651
```

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

3 Performance Evaluation and Comparison

Now that we are familiar with the basics of how to create tasks and learners, how to fit models, and do some basic performance evaluation, let's have a look at some of the details, and in particular how mlr3 makes it easy to perform many common machine learning steps.

We will cover the following topics:

Binary classification and ROC curves

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 of where to predict one class versus the other.

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. This is usually done to get an overview of how different methods perform across different tasks. We cover how to

- create a benchmarking design,
- execute a design and aggregate results, and
- convert benchmarking objects to other types of objects that can be used for different purposes.

3.1 ROC Curve and Thresholds

As we have seen before, binary classification is special because of the presence of a positive and negative class and a threshold probability to ditinguish between the two. ROC Analysis, which stands for receiver operating characteristics, applies specifically to this case and allows to get a better picture of the trade offs when choosing between the two classes. We saw earlier that one can retrieve the confusion matrix of a `Prediction` by accessing the `$confusion` field:

```
data("Sonar", package = "mlbench")
task = as_task_classif(Sonar, target = "Class", positive = "M")

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 upper left quandrant denotes the number of times our model predicted the positive class and was correct about it. Similarly, the lower right quadrant denotes the number of times our model predicted the negative class and was also correct about it. Together, the elements on the diagonal are called True Positives (TP) and True Negatives (TN). The upper right quadrant denotes the number of times we falsely predicted a positive label, and is called False Positives (FP). The lower left quadrant is called False Negatives (FN).

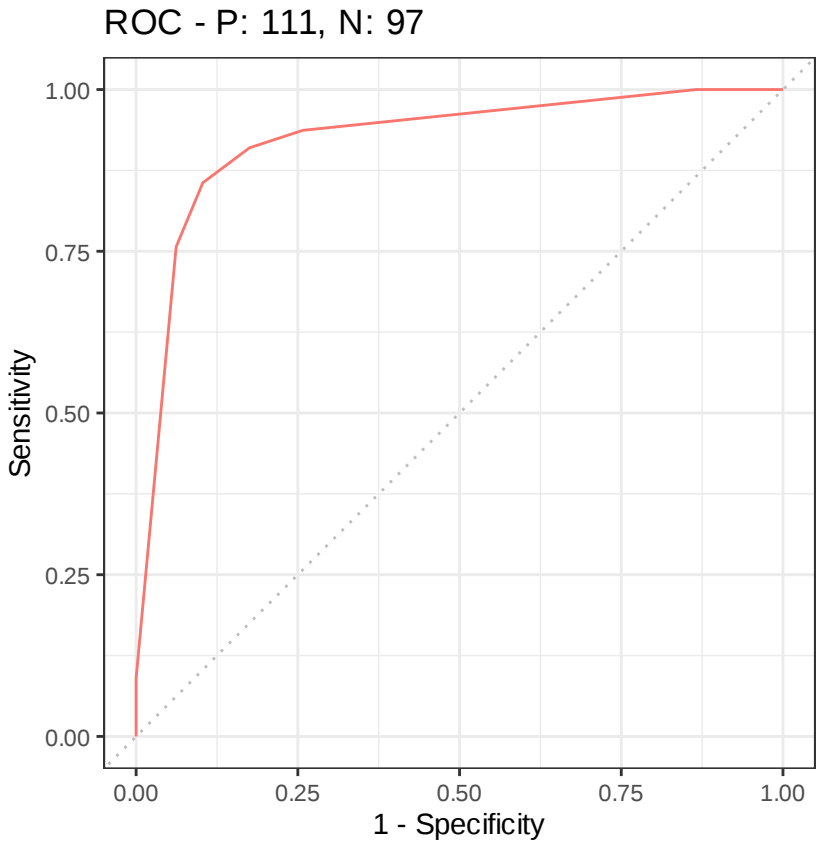
We can derive the following performance metrics from the confusion matrix:

- **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?

It is difficult to achieve a high TPR and low FPR at the same time. We can characterize the behavior of a binary classifier for different thresholds by plotting the TPR and FPR values – this is the ROC curve. The best classifier lies on the top-left corner – the TPR is 1 and the FPR is 0. The worst classifier lies at the diagonal. Classifiers on the diagonal produce labels essentially randomly (possibly with different proportions). For example, 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 – ROC curves for different labels are symmetric with respect to the diagonal, so a curve below the diagonal would indicate that the positive and negative class labels have been switched by the classifier.

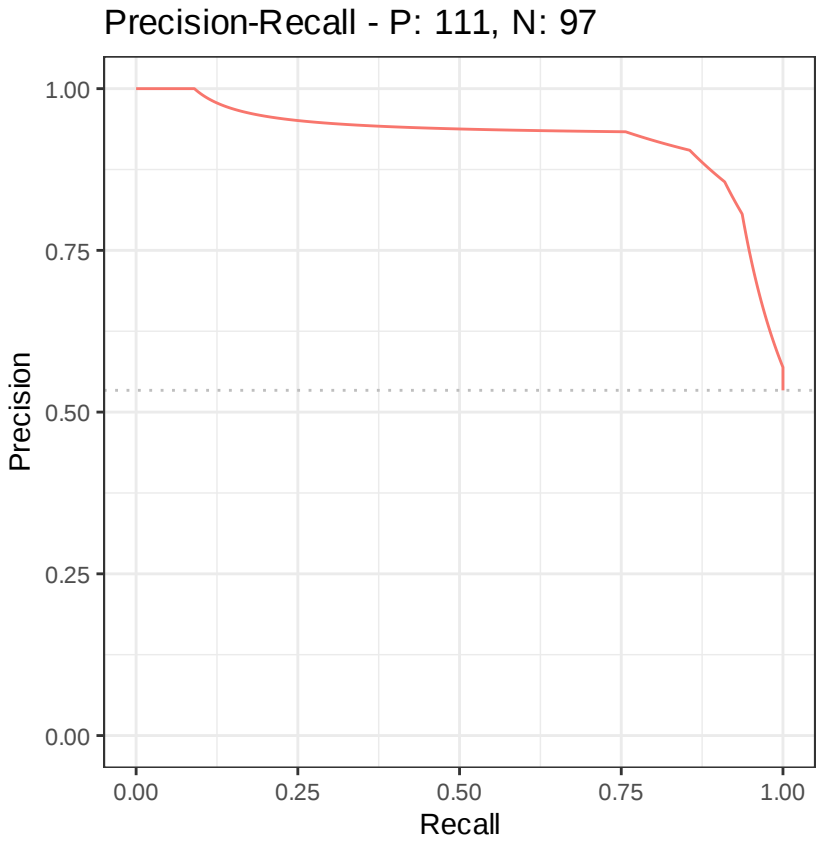
For `mlr3` prediction objects, the ROC curve can easily be created with mlr3viz which relies on the precrec to calculate and plot ROC curves:

```
library("mlr3viz")

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



We can also plot the precision-recall curve (PPV vs. TPR). The main difference between ROC curves and precision-recall curves (PRC) is that the number of true-negative results is not used for making a PRC. PRCs are preferred over ROC curves for imbalanced populations.

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



3.2 Resampling

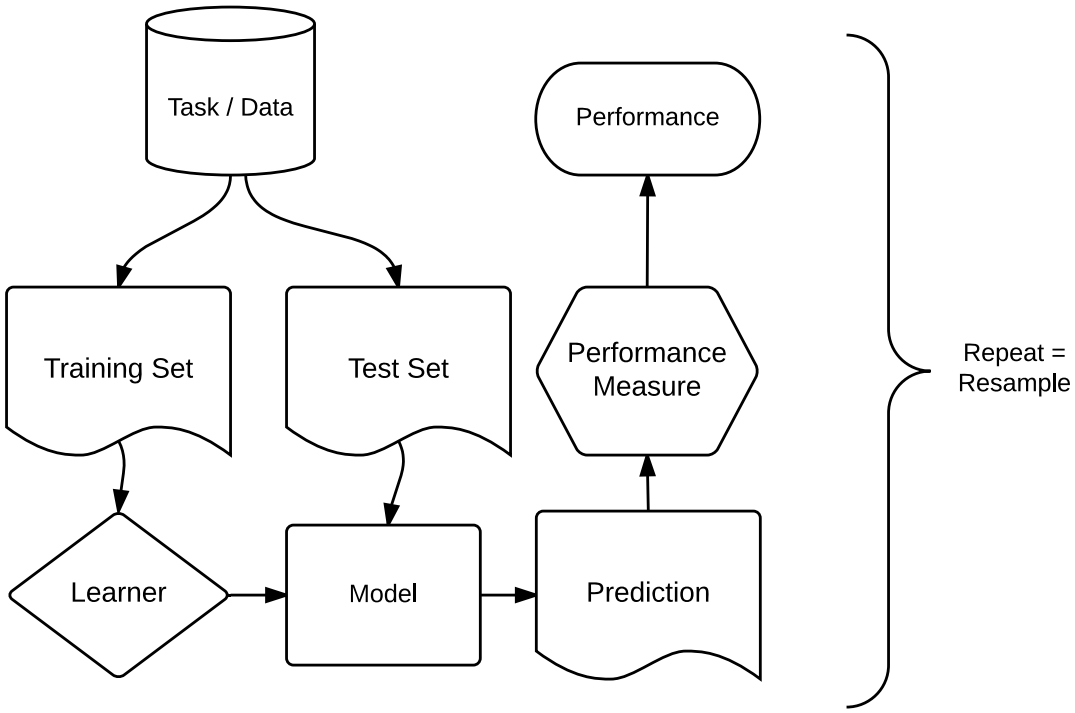
When evaluating the performance of a model, we are interested in its generalization performance – how well will it perform on new data that has not been seen during training? We can estimate the generalization performance by evaluating a model on a test set, as we have done above, that was created to contain only observations that are not contained in the training set. There are many different strategies for partitioning a data set into training and test; in `mlr3` we call these strategies "resampling". `mlr3` includes the following predefined resampling strategies:

- `cross validation` (`"cv"`),
- `leave-one-out cross validation` (`"loo"`),
- `repeated cross validation` (`"repeated_cv"`),
- `bootstrapping` (`"bootstrap"`),
- `subsampling` (`"subsampling"`),
- `holdout` (`"holdout"`),
- `in-sample resampling` (`"insample"`), and
- `custom resampling` (`"custom"`).

In particular, it is often desirable to repeatedly split the entire data in different ways to ensure that a "lucky" or "unlucky" split does not bias the generalization performance estimate. Without resampling strategies like the ones we provide here, this is a tedious and error-prone process.

The following sections provide guidance on how to select a resampling strategy and how to use it.

Here is a graphical illustration of the resampling process in general:



3.2.1 Settings

We will use the `penguins` task and a simple classification tree from the rpart package as an example here.

```
library("mlr3verse")

task = tsk("penguins")
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; this also lists the parameters that can be changed to affect the behavior of each strategy:

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

```
##            key        params iters
## 1:   bootstrap ratio,repeats    30
## 2:      custom                   NA
## 3:   custom_cv                   NA
## 4:          cv         folds    10
## 5:     holdout         ratio     1
```

```
## 6:    insample                   1
## 7:         loo                  NA
## 8: repeated_cv folds,repeats   100
## 9: subsampling ratio,repeats    30
```

Additional resampling methods for special use cases are available via extension packages, such as mlr3spatiotemporal for spatial data.

What we showed in the train/predict/score part is the equivalent of holdout resampling, done manually, so let's consider this one first. We can retrieve elements from the dictionary `mlr_resamplings` via `$get()` or 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 to a dataset yet.

By default we get a .66/.33 split of the data into training and test. There are two ways in which this 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
```

3.2.2 Instantiation

So far we have only chosen a resampling strategy; we now need to instantiate it with data.

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` objects.

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

```
##  int [1:275] 2 3 4 5 7 8 9 11 12 15 ...
```

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

```
##  int [1:69] 1 6 10 13 14 18 42 43 51 52 ...
```

Note that if you want to compare multiple Learners in a fair manner, using the same instantiated resampling for each learner is mandatory, such that each learner gets exactly the same training data and the performance of the trained model is evaluated in exactly the same test set. A way to greatly simplify the comparison of multiple learners is discussed in the section on benchmarking.

3.2.3 Execution

With a Task, a Learner, and a Resampling object we can call resample(), which fits the learner on the training set and evaluates it on the test set. This may happen multiple times, depending on the given resampling strategy. The result of running the resample() function is a ResampleResult object. We can tell resample() to keep the fitted models (for example for later inspection) by setting the store_models option to TRUEand then starting the computation:

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

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

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

Here we use a three-fold cross-validation resampling, which trains and evaluates on three different training and test sets. The returned ResampleResult, stored as rr, provides various getters to access and aggregate the stored information. Here are a few examples:

- Calculate the average performance across all resampling iterations, in terms of classification error:

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

  ```
  ## classif.ce
  ##     0.06097
  ```

- Extract the performance for the individual resampling iterations:

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

```
##                 task  task_id                    learner    learner_id
## 1: <TaskClassif[49]> penguins <LearnerClassifRpart[37]> classif.rpart
## 2: <TaskClassif[49]> penguins <LearnerClassifRpart[37]> classif.rpart
## 3: <TaskClassif[49]> penguins <LearnerClassifRpart[37]> classif.rpart
##            resampling resampling_id iteration              prediction
## 1: <ResamplingCV[19]>            cv         1 <PredictionClassif[20]>
## 2: <ResamplingCV[19]>            cv         2 <PredictionClassif[20]>
## 3: <ResamplingCV[19]>            cv         3 <PredictionClassif[20]>
##    classif.ce
## 1:    0.08696
## 2:    0.06087
## 3:    0.03509
```

This is useful to check if one (or more) of the iterations are very different from the average.

- 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; this allows to see in detail which observations were used for what purpose when:

```
rr$resampling
```

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

```
rr$resampling$iters
```

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

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

```
##  int [1:115] 3 5 9 11 12 15 16 18 28 33 ...
```

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

```
##  int [1:229] 1 6 13 20 22 23 24 26 27 29 ...
```

- Retrieve the model trained in a specific iteration and inspect it, for example to investigate why the performance in this iteration was very different from the average:

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

```
## n= 229
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
##  1) root 229 126 Adelie (0.44978 0.18341 0.36681)
##    2) flipper_length< 207.5 141  40 Adelie (0.71631 0.28369 0.00000)
##      4) bill_length< 42.4 95   0 Adelie (1.00000 0.00000 0.00000) *
##      5) bill_length>=42.4 46   6 Chinstrap (0.13043 0.86957 0.00000)
##       10) body_mass>=4175 8   3 Adelie (0.62500 0.37500 0.00000) *
##       11) body_mass< 4175 38   1 Chinstrap (0.02632 0.97368 0.00000) *
##    3) flipper_length>=207.5 88   4 Gentoo (0.02273 0.02273 0.95455) *
```

- Extract the individual predictions:

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

```
## <PredictionClassif> for 344 observations:
##     row_ids     truth  response prob.Adelie prob.Chinstrap prob.Gentoo
##           3    Adelie    Adelie     1.00000         0.0000     0.00000
##           5    Adelie    Adelie     1.00000         0.0000     0.00000
##           9    Adelie    Adelie     1.00000         0.0000     0.00000
## ---
##         336 Chinstrap Chinstrap     0.04878         0.9268     0.02439
##         337 Chinstrap Chinstrap     0.25000         0.5000     0.25000
##         339 Chinstrap Chinstrap     0.04878         0.9268     0.02439
```

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

```
## <PredictionClassif> for 115 observations:
##     row_ids     truth  response prob.Adelie prob.Chinstrap prob.Gentoo
##           3    Adelie    Adelie     1.00000        0.00000      0.0000
##           5    Adelie    Adelie     1.00000        0.00000      0.0000
##           9    Adelie    Adelie     1.00000        0.00000      0.0000
## ---
##         342 Chinstrap Chinstrap     0.02632        0.97368      0.0000
##         343 Chinstrap    Gentoo     0.02273        0.02273      0.9545
##         344 Chinstrap Chinstrap     0.02632        0.97368      0.0000
```

- Filter the result to only keep specified resampling iterations:

```
rr$filter(c(1, 3))
print(rr)
```

```
## <ResampleResult> of 2 iterations
## * Task: penguins
## * Learner: classif.rpart
## * Warnings: 0 in 0 iterations
## * Errors: 0 in 0 iterations
```

3.2.4 Custom resampling

Sometimes it is necessary to perform resampling with custom splits, e.g. to reproduce results reported in a study. A manual resampling instance can be created using the `"custom"` template.

```
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
```

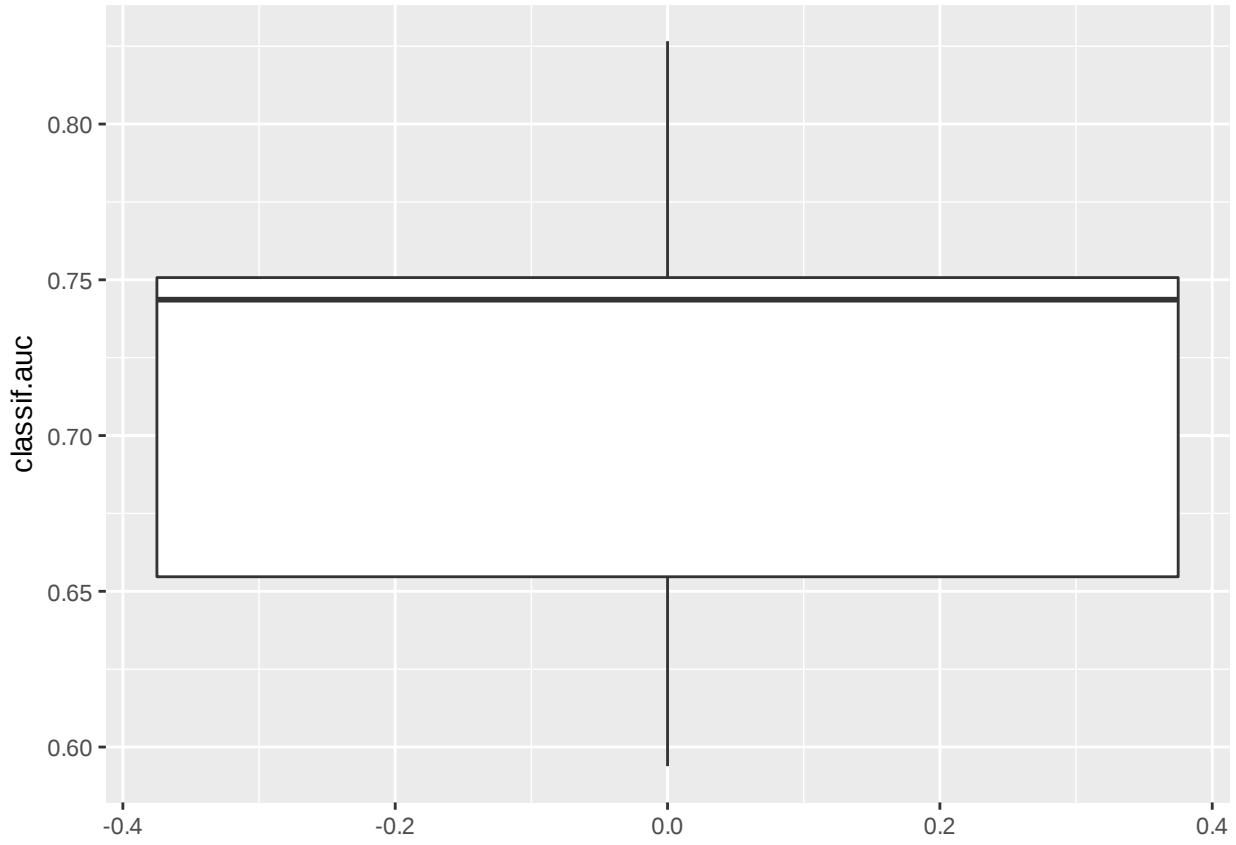
3.2.5 Resampling with (predefined) groups

In some cases, it is desirable to keep observations together, i.e. to not separate them into training and test set. This can be defined through the column role `"group"` during Task creation, i.e. a special column in the data specifies the groups (see also the help page on this column role). In `mlr` this was called "blocking". See also the mlr3gallery post on this topic for a practical example.

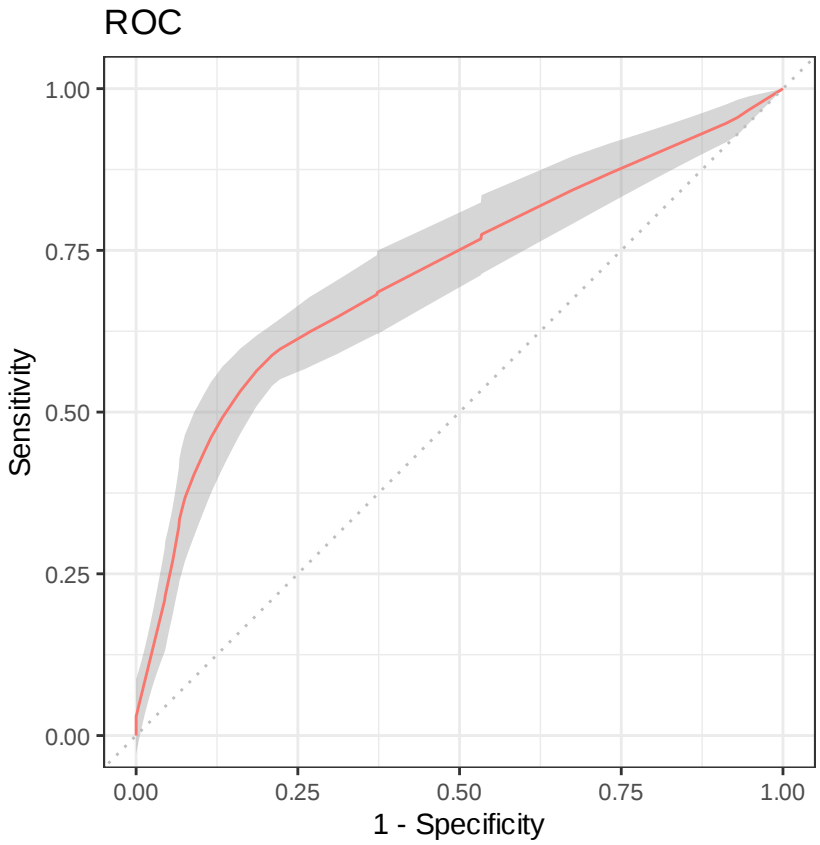
3.2.6 Plotting Resample Results

mlr3viz provides a `autoplot()` method for resampling results. As an example, we create a binary classification task with two features, perform a resampling with a 10-fold cross-validation and visualize the results:

```
task = tsk("pima")
task$select(c("glucose", "mass"))
learner = lrn("classif.rpart", predict_type = "prob")
rr = resample(task, learner, rsmp("cv"), store_models = TRUE)

# boxplot of AUC values across the 10 folds
autoplot(rr, measure = msr("classif.auc"))
```

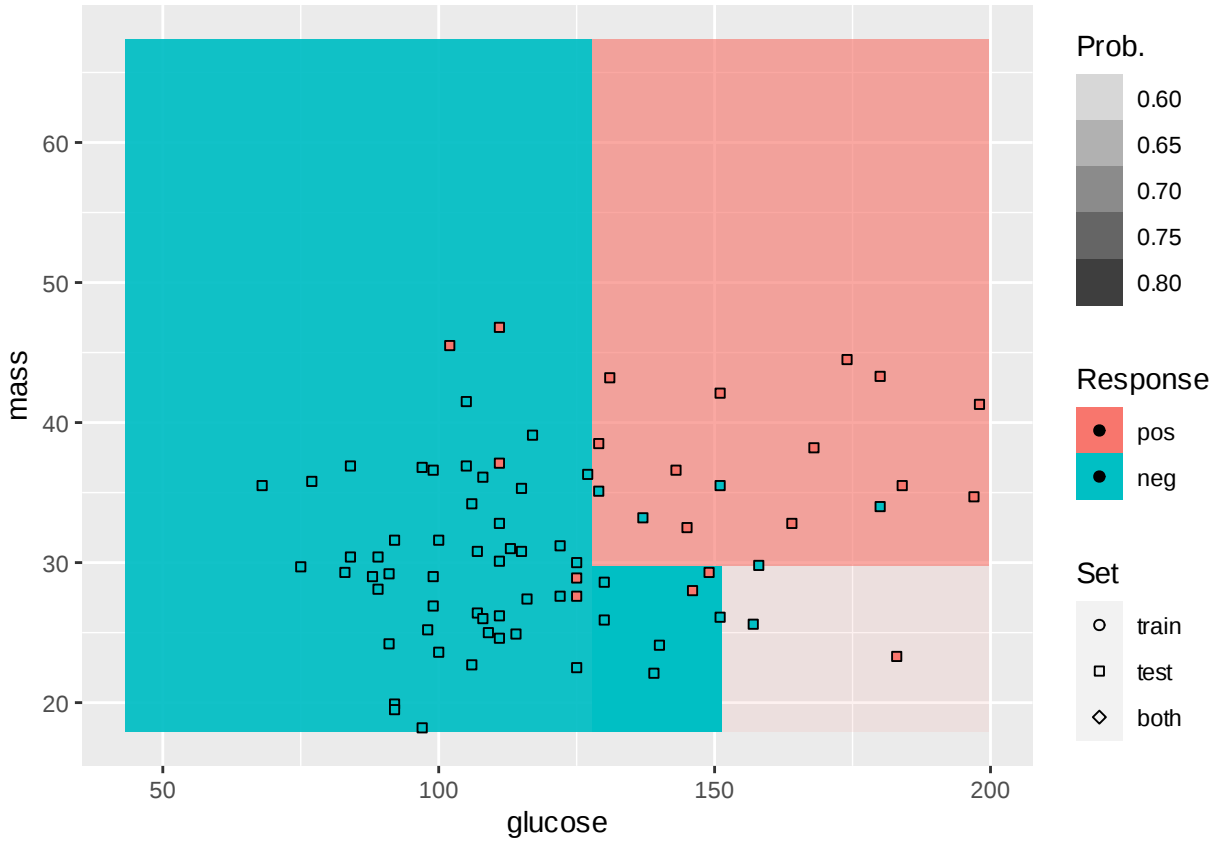



```
# ROC curve, averaged over 10 folds
autoplot(rr, type = "roc")
```



We can also plot the predictions of individual models:

```
# learner predictions for the first fold
rr$filter(1)
autoplot(rr, type = "prediction")
```



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

3.3 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()` convenience function that takes care of most of the work of repeatedly training and evaluating models under the same conditions.

3.3.1 Design Creation

Benchmark experiments in `mlr3` are specified through a design. Such a design is essentially a table of scenarios to be evaluated; in particular unique combinations of `Task`, `Learner` and `Resampling` triplets.

We use the `benchmark_grid()` function to create an exhaustive design (that evaluates each learner on each task with each resampling) and instantiate the resampling properly, so that all learners are executed on the same train/test split for each tasks. We set the learners to predict probabilities and also tell them to predict for the observations of the training set (by setting `predict_sets` to `c("train", "test")`). Additionally, we use `tsks()`, `lrns()`, and `rsmps()` to retrieve lists of `Task`, `Learner` and `Resampling` in the same fashion as `tsk()`, `lrn()` and `rsmp()`.

```
library("mlr3verse")

design = benchmark_grid(
  tasks = tsks(c("spam", "german_credit", "sonar")),
  learners = lrns(c("classif.ranger", "classif.rpart", "classif.featureless"),
    predict_type = "prob", predict_sets = c("train", "test")),
  resamplings = rsmps("cv", folds = 3)
)
print(design)
```

```
##                 task                          learner         resampling
## 1: <TaskClassif[49]>      <LearnerClassifRanger[37]> <ResamplingCV[19]>
## 2: <TaskClassif[49]>       <LearnerClassifRpart[37]> <ResamplingCV[19]>
## 3: <TaskClassif[49]> <LearnerClassifFeatureless[37]> <ResamplingCV[19]>
## 4: <TaskClassif[49]>      <LearnerClassifRanger[37]> <ResamplingCV[19]>
## 5: <TaskClassif[49]>       <LearnerClassifRpart[37]> <ResamplingCV[19]>
## 6: <TaskClassif[49]> <LearnerClassifFeatureless[37]> <ResamplingCV[19]>
## 7: <TaskClassif[49]>      <LearnerClassifRanger[37]> <ResamplingCV[19]>
## 8: <TaskClassif[49]>       <LearnerClassifRpart[37]> <ResamplingCV[19]>
## 9: <TaskClassif[49]> <LearnerClassifFeatureless[37]> <ResamplingCV[19]>
```

The created `design` can be passed to `benchmark()` to start the computation. It is also possible to create a custom design manually, for example to exclude certain task-learner combinations. However, if you create a custom task with `data.table()`, 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. See the help page on `benchmark_grid()` for an example.

3.3.2 Execution and Aggregation of Results

After the benchmark design is ready, we can call `benchmark()` on it:

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

Note that we did not have to instantiate the resampling 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.

Once the benchmarking is done (and, depending on the size of your design, this can take quite some time), we can aggregate the performance with `$aggregate()`. We create two measures to calculate the area under the curve (AUC) for the training and the test set:

```
measures = list(
  msr("classif.auc", predict_sets = "train", id = "auc_train"),
  msr("classif.auc", id = "auc_test")
)

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

```
##    nr      resample_result       task_id          learner_id resampling_id
## 1:  1 <ResampleResult[22]>          spam      classif.ranger            cv
## 2:  2 <ResampleResult[22]>          spam       classif.rpart            cv
## 3:  3 <ResampleResult[22]>          spam classif.featureless            cv
## 4:  4 <ResampleResult[22]> german_credit      classif.ranger            cv
## 5:  5 <ResampleResult[22]> german_credit       classif.rpart            cv
## 6:  6 <ResampleResult[22]> german_credit classif.featureless            cv
## 7:  7 <ResampleResult[22]>         sonar      classif.ranger            cv
## 8:  8 <ResampleResult[22]>         sonar       classif.rpart            cv
## 9:  9 <ResampleResult[22]>         sonar classif.featureless            cv
##    iters auc_train auc_test
## 1:     3    0.9995   0.9846
## 2:     3    0.9100   0.9090
## 3:     3    0.5000   0.5000
## 4:     3    0.9987   0.8022
## 5:     3    0.7912   0.6953
## 6:     3    0.5000   0.5000
## 7:     3    1.0000   0.9192
## 8:     3    0.9229   0.7687
## 9:     3    0.5000   0.5000
```

We can aggregate the results even further. For example, we might be interested to know which learner performed best across all tasks. 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 the data.table package. As larger AUC scores are better, we multiply the values by -1 such that the best learner has a rank of 1.

```
library("data.table")
# 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:          spam      classif.ranger          1         1
## 2:          spam       classif.rpart          2         2
## 3:          spam classif.featureless          3         3
## 4: german_credit      classif.ranger          1         1
## 5: german_credit       classif.rpart          2         2
## 6: german_credit classif.featureless          3         3
## 7:         sonar      classif.ranger          1         1
## 8:         sonar       classif.rpart          2         2
## 9:         sonar classif.featureless          3         3
```

```
# 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)]
```

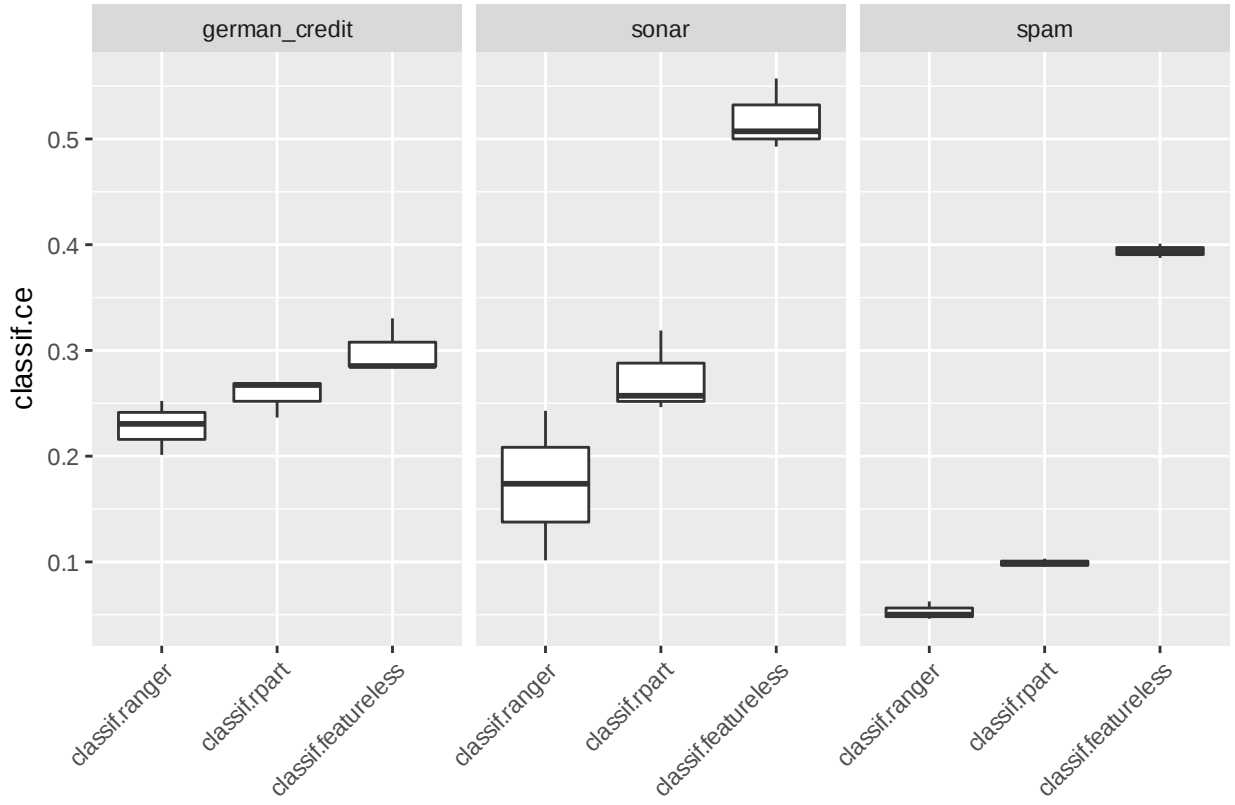
```
##             learner_id mrank_train mrank_test
## 1:      classif.ranger           1          1
## 2:       classif.rpart           2          2
## 3: classif.featureless           3          3
```

Unsurprisingly, the featureless learner is worse overall. The winner is the classification forest, which outperforms a single classification tree.

3.3.3 Plotting Benchmark Results

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

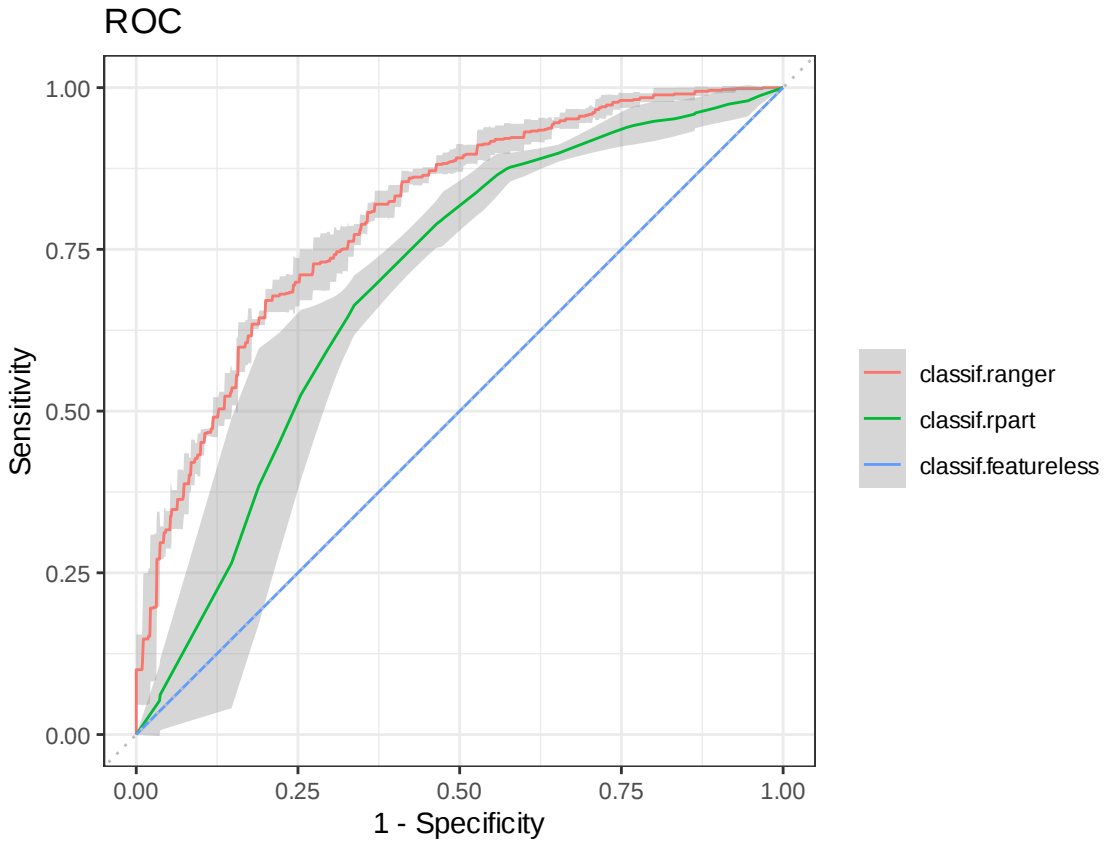
```
autoplot(bmr) + ggplot2::theme(axis.text.x = ggplot2::element_text(angle = 45, hjust = 1))
```



Such a plot gives a nice overview of the overall performance and how learners compare on different tasks in an intuitive way.

We can also plot ROC (receiver operating characteristics) curves. We filter the `BenchmarkResult` to only contain a single `Task`, then we simply plot the result:

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



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

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

```
## <ResampleResult> of 3 iterations
## * Task: german_credit
```

```
## * 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
##      0.8022
```

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

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

```
## <LearnerClassifRanger:classif.ranger>
## * Model: -
## * Parameters: num.threads=1
## * Packages: mlr3, mlr3learners, ranger
## * Predict Type: prob
## * Feature types: logical, integer, numeric, character, factor, ordered
## * Properties: hotstart_backward, importance, multiclass, oob_error,
##   twoclass, weights
```

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

```
## [1]  3  4  8  9 10 12
```

3.3.5 Converting and Merging

A `ResampleResult` can be converted to a `BenchmarkResult` with the function `as_benchmark_result()`. We can also merge two `BenchmarkResults` into a larger result object, for example for two related benchmarks that were done on different machines.

```
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
```

4 Model Optimization

In machine learning, when you are dissatisfied with the performance of a model, you might ask yourself how to best improve the model: * Can it be improved by tweaking the hyperparameters of the learner, i.e. the configuration options that affect its behavior? * Or, should you just use a completely different learner for this particular task? This chapter might help answer this question.

Model Tuning

Machine learning algorithms have default values set for their hyperparameters. In many cases, these hyperparameters need to be changed by the user to achieve optimal performance on the given dataset. While you can certainly search for hyperparameter settings that improve performance manually, we do not recommend this approach as it is tedious and rarely leads to the best performance. Fortunately, the `mlr3` ecosystem provides packages and tools for automated tuning. In order to tune a machine learning algorithm, you have to specify (1) the search space, (2) the optimization algorithm (i.e. tuning method), (3) an evaluation method (i.e. a resampling strategy), and (4) a performance measure.

In the tuning part, we will have a look at:

- empirically sound hyperparameter tuning,
- selecting the optimizing algorithm,
- defining search spaces concisely,
- triggering the tuning, and
- automating tuning.

We will use the mlr3tuning package, which supports common tuning operations.

Feature Selection

Tuning the hyperparameters is only one way of improving the performance of your model. The second part of this chapter explains feature selection, also known as variable or descriptor selection. Feature selection is the process of finding the feature subset that is most relevant with respect to the prediction or for which the learner fits a model with the highest performance. Apart from improving model performance, there are additional reasons to perform feature selection:

- enhance the interpretability of the model,
- speed up model fitting, or
- eliminate the need to collect lots of expensive features.

Here, we mostly focus on feature selection as a means for improving model performance.

There are different approaches to identify the relevant features. In the feature selection part, we describe three methods:

- Filter algorithms select features independently of the learner by scoring the different features.
- Variable importance filters select features that are important according the model induced by a learner.

- Wrapper methods iteratively select features to optimize a performance measure, each time fitting and evaluating a model with a different subset of features.

Note that filters operate independently of learners. Variable importance filters rely on the learner to extract information on feature importance from a trained model, for example by inspecting a learned decision tree and returning the features that are used in the first few levels. The obtained importance values can be used to subset the data, which can then be used to train a learner. Wrapper methods can be used with any learner but need to train the learner potentially many times, making this the most expensive method.

Nested Resampling

For hyperparameter tuning, a normal resampling (e.g. a cross-validation) is no longer sufficient to ensure an unbiased evaluation. Consider the following thought experiment to gain intuition for why this is the case. Suppose a learner has a hyperparameter that has no real effect on the fitted model, but only introduces random noise into the predictions. Evaluating different values for this hyperparameter, one will show the best performance (purely randomly). This is the hyperparameter value that will be chosen as the best, although the hyperparameter has no real effect. To discover this, another separate validation set is required – it will reveal that the "optimized" setting really does not perform better than anything else.

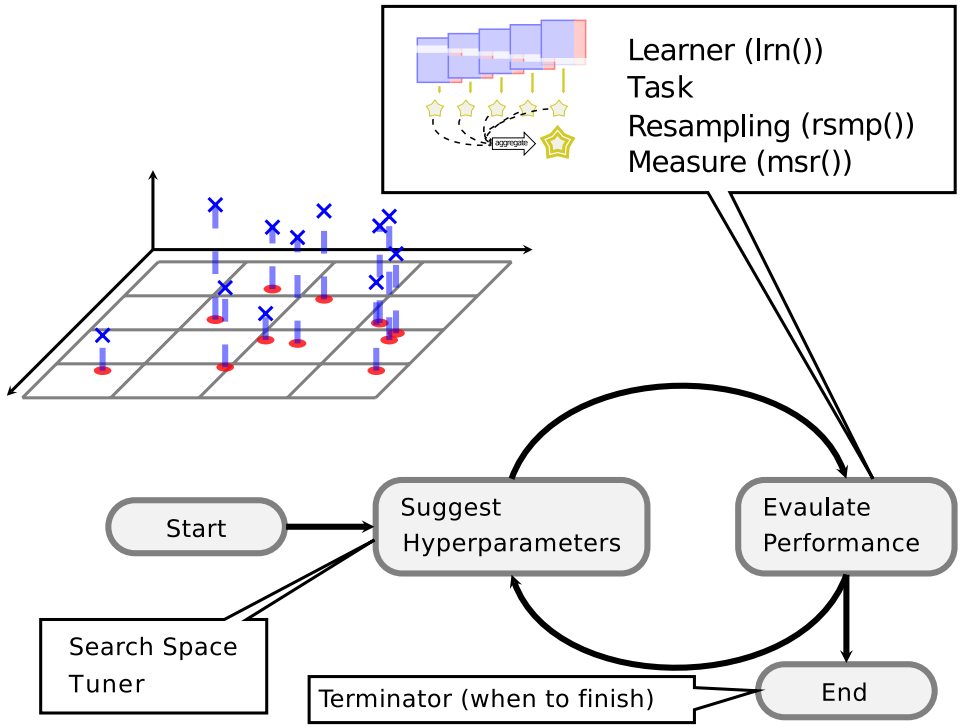
We need a nested resampling to ensure unbiased estimates of the generalization error during hyperparameter optimization. We discuss the following aspects in this part:

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

4.1 Hyperparameter Tuning

Hyperparameters are the parameters of the learners that control how a model is fit to the data. They are sometimes called second-level or second-order parameters of machine learning – the parameters of the *models* are the first-order parameters and "fit" to the data durning model training. The hyerparameters of a learner can have a major impact on the performance of a learned model, but are often only optimized in an ad-hoc manner or not at all. This process is often called model 'tuning'.

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



At the heart of mlr3tuning are the R6 classes

- `TuningInstanceSingleCrit`, `TuningInstanceMultiCrit` to describe the tuning problem and store the results, and
- `Tuner` as the base class for implementations of tuning algorithms.

4.1.1 The `TuningInstance*` Classes

We will examine the optimization of a simple classification tree on the `Pima Indian Diabetes` data set as an introductory example here.

```
library("mlr3verse")
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 rpart classification tree and choose a subset of the hyperparameters we want to tune. This is often referred to as the "tuning space". First, let's look at all the hyperparameters that are available. Information on what they do can be found in `the documentation of the learner`.

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

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

Here, we opt to tune two hyperparameters:

- The complexity hyperparameter `cp` that controls when the learner considers introducing another branch.
- The `minsplit` hyperparameter that controls how many observations must be present in a leaf for another split to be attempted.

The tuning space needs to be bounded with lower and upper bounds for the values of the hyperparameters:

```
search_space = ps(
  cp = p_dbl(lower = 0.001, upper = 0.1),
  minsplit = p_int(lower = 1, upper = 10)
)
search_space
```

```
## <ParamSet>
##          id    class lower upper nlevels        default value
## 1:       cp ParamDbl 0.001   0.1     Inf <NoDefault[3]>
## 2: minsplit ParamInt 1.000  10.0      10 <NoDefault[3]>
```

The bounds are usually set based on experience.

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

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

Finally, we have to specify the budget available for tuning. This is a crucial step, as exhaustively evaluating all possible hyperparameter configurations is usually not feasible. `mlr3` allows to specify complex termination criteria by selecting one of the available `Terminators`:

- Terminate after a given time (TerminatorClockTime).
- Terminate after a given number of iterations (TerminatorEvals).
- Terminate after a specific performance has been reached (TerminatorPerfReached).
- Terminate when tuning does find a better configuration for a given number of iterations (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 TuningInstanceSingleCrit:

```
library("mlr3tuning")
```

```
## Loading required package: paradox
```

```
evals20 = trm("evals", n_evals = 20)

instance = TuningInstanceSingleCrit$new(
  task = task,
  learner = learner,
  resampling = hout,
  measure = measure,
  search_space = search_space,
  terminator = evals20
)
instance
```

```
## <TuningInstanceSingleCrit>
## * State:  Not optimized
## * Objective: <ObjectiveTuning:classif.rpart_on_pima>
## * Search Space:
## <ParamSet>
##          id    class lower upper nlevels        default value
## 1:       cp ParamDbl 0.001   0.1     Inf <NoDefault[3]>
## 2: minsplit ParamInt 1.000  10.0      10 <NoDefault[3]>
## * Terminator: <TerminatorEvals>
## * Terminated: FALSE
## * Archive:
## <ArchiveTuning>
## Null data.table (0 rows and 0 cols)
```

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.

4.1.2 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)
- Non-Linear Optimization (TunerNLoptr)

If you're interested in learning more about these approaches, the Wikipedia page on hyperparameter optimization is a good place to start.

In this example, we will use a simple grid search with a grid resolution of 5.

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

As we have only numeric parameters, TunerGridSearch will create an equidistant grid between the respective upper and lower bounds. Our two-dimensional grid of resolution 5 consists of $5^2 = 25$ configurations. Each configuration is a distinct setting of hyperparameter values for the previously defined Learner which is then fitted to the task and evaluated using the provided Resampling. 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, i.e. here the tuner will stop after evaluating 20 of the 25 total configurations.

4.1.3 Triggering the Tuning

To start the tuning, we simply pass the TuningInstanceSingleCrit to the `$optimize()` method of the initialized Tuner. The tuner proceeds as follows:

1. The Tuner proposes at least one hyperparameter configuration to evaluate (the Tuner may propose multiple points to be able to evaluate them in parallel, which can be controlled via the setting `batch_size`).
2. For each configuration, the given Learner is fitted on the Task and evaluated using the provided Resampling. All evaluations are stored in the archive of the TuningInstanceSingleCrit.
3. The Terminator is queried if the budget is exhausted. If the budget is not exhausted, go back to 1), else terminate.
4. Determine the configurations with the best observed performance from the archive.
5. Store the best configurations as result in the tuning instance object. The best hyperparameter settings (`$result_learner_param_vals`) and the corresponding measured performance (`$result_y`) can be retrieved from the tuning instance.

```
tuner$optimize(instance)
```

```
## INFO  [16:04:11.758] [bbotk] Starting to optimize 2 parameter(s) with '<OptimizerGridSearc
## INFO  [16:04:11.800] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:11.975] [bbotk] Result of batch 1:
## INFO  [16:04:11.977] [bbotk]     cp minsplit classif.ce runtime_learners
## INFO  [16:04:11.977] [bbotk]  0.001        5     0.2734            0.022
## INFO  [16:04:11.977] [bbotk]                                      uhash
## INFO  [16:04:11.977] [bbotk]  f941926a-ce9e-483b-9dce-583531452e93
## INFO  [16:04:11.978] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:12.064] [bbotk] Result of batch 2:
## INFO  [16:04:12.066] [bbotk]      cp minsplit classif.ce runtime_learners
## INFO  [16:04:12.066] [bbotk]  0.0505        3     0.2266            0.013
```

```
## INFO  [16:04:12.066] [bbotk]                                  uhash
## INFO  [16:04:12.066] [bbotk]  a1e2401b-c5e0-4ee3-a9b6-74ed04363b14
## INFO  [16:04:12.068] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:12.165] [bbotk] Result of batch 3:
## INFO  [16:04:12.167] [bbotk]     cp minsplit classif.ce runtime_learners
## INFO  [16:04:12.167] [bbotk]  0.001        1     0.2773            0.014
## INFO  [16:04:12.167] [bbotk]                                  uhash
## INFO  [16:04:12.167] [bbotk]  771763d8-b099-439f-b1e7-aec5000451f2
## INFO  [16:04:12.169] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:12.255] [bbotk] Result of batch 4:
## INFO  [16:04:12.257] [bbotk]       cp minsplit classif.ce runtime_learners
## INFO  [16:04:12.257] [bbotk]  0.02575        3     0.2305            0.013
## INFO  [16:04:12.257] [bbotk]                                  uhash
## INFO  [16:04:12.257] [bbotk]  3c9a197a-abff-4ee3-a4c1-95ca3955b708
## INFO  [16:04:12.259] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:12.345] [bbotk] Result of batch 5:
## INFO  [16:04:12.348] [bbotk]     cp minsplit classif.ce runtime_learners
## INFO  [16:04:12.348] [bbotk]  0.001        3     0.2734            0.014
## INFO  [16:04:12.348] [bbotk]                                  uhash
## INFO  [16:04:12.348] [bbotk]  d4444d88-608b-443e-9c8d-7c78d8e67021
## INFO  [16:04:12.349] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:12.436] [bbotk] Result of batch 6:
## INFO  [16:04:12.439] [bbotk]      cp minsplit classif.ce runtime_learners
## INFO  [16:04:12.439] [bbotk]  0.0505        5     0.2266            0.013
## INFO  [16:04:12.439] [bbotk]                                  uhash
## INFO  [16:04:12.439] [bbotk]  6af4bf40-ee4a-4487-af74-7d591156a76d
## INFO  [16:04:12.440] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:12.541] [bbotk] Result of batch 7:
## INFO  [16:04:12.544] [bbotk]   cp minsplit classif.ce runtime_learners
## INFO  [16:04:12.544] [bbotk]  0.1        8     0.2656            0.012 f1f648a8-b684-4b0f-
## INFO  [16:04:12.545] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:12.645] [bbotk] Result of batch 8:
## INFO  [16:04:12.647] [bbotk]       cp minsplit classif.ce runtime_learners
## INFO  [16:04:12.647] [bbotk]  0.02575        5     0.2305            0.014
## INFO  [16:04:12.647] [bbotk]                                  uhash
## INFO  [16:04:12.647] [bbotk]  58bea9f2-1d85-4dae-a822-8846a71c226a
## INFO  [16:04:12.649] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:12.744] [bbotk] Result of batch 9:
## INFO  [16:04:12.746] [bbotk]      cp minsplit classif.ce runtime_learners
## INFO  [16:04:12.746] [bbotk]  0.0505       10     0.2266            0.015
## INFO  [16:04:12.746] [bbotk]                                  uhash
## INFO  [16:04:12.746] [bbotk]  01fa1fcb-d4b9-4270-bd21-78a3a4a9dd85
## INFO  [16:04:12.748] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:12.842] [bbotk] Result of batch 10:
## INFO  [16:04:12.844] [bbotk]       cp minsplit classif.ce runtime_learners
## INFO  [16:04:12.844] [bbotk]  0.02575        8     0.2305            0.013
## INFO  [16:04:12.844] [bbotk]                                  uhash
## INFO  [16:04:12.844] [bbotk]  4e38076d-99bb-4146-b99c-20039054372c
## INFO  [16:04:12.846] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:12.930] [bbotk] Result of batch 11:
```

```
## INFO  [16:04:12.932] [bbotk]       cp minsplit classif.ce runtime_learners
## INFO  [16:04:12.932] [bbotk]  0.07525       10     0.2266            0.013
## INFO  [16:04:12.932] [bbotk]                                    uhash
## INFO  [16:04:12.932] [bbotk]  245a9e6c-8325-47f3-9853-2250ec7c1a84
## INFO  [16:04:12.933] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:13.021] [bbotk] Result of batch 12:
## INFO  [16:04:13.023] [bbotk]   cp minsplit classif.ce runtime_learners
## INFO  [16:04:13.023] [bbotk]  0.1        3     0.2656            0.015 64c4f2d9-eb99-4d3d-
## INFO  [16:04:13.025] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:13.121] [bbotk] Result of batch 13:
## INFO  [16:04:13.123] [bbotk]     cp minsplit classif.ce runtime_learners
## INFO  [16:04:13.123] [bbotk]  0.001       10     0.2891            0.021
## INFO  [16:04:13.123] [bbotk]                                    uhash
## INFO  [16:04:13.123] [bbotk]  e765b2da-c951-4e63-9b5f-679d7839e7e7
## INFO  [16:04:13.124] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:13.216] [bbotk] Result of batch 14:
## INFO  [16:04:13.218] [bbotk]       cp minsplit classif.ce runtime_learners
## INFO  [16:04:13.218] [bbotk]  0.02575        1     0.2305            0.014
## INFO  [16:04:13.218] [bbotk]                                    uhash
## INFO  [16:04:13.218] [bbotk]  62812c0d-6864-44be-adb8-0a8e88d09f68
## INFO  [16:04:13.220] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:13.306] [bbotk] Result of batch 15:
## INFO  [16:04:13.308] [bbotk]       cp minsplit classif.ce runtime_learners
## INFO  [16:04:13.308] [bbotk]  0.02575       10     0.2305            0.014
## INFO  [16:04:13.308] [bbotk]                                    uhash
## INFO  [16:04:13.308] [bbotk]  f2e0a760-5a01-488c-ad56-a0f0548d1ec4
## INFO  [16:04:13.309] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:13.400] [bbotk] Result of batch 16:
## INFO  [16:04:13.402] [bbotk]   cp minsplit classif.ce runtime_learners
## INFO  [16:04:13.402] [bbotk]  0.1       10     0.2656            0.013 cc440ffd-3388-4d38-
## INFO  [16:04:13.406] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:13.505] [bbotk] Result of batch 17:
## INFO  [16:04:13.507] [bbotk]       cp minsplit classif.ce runtime_learners
## INFO  [16:04:13.507] [bbotk]  0.07525        3     0.2266            0.013
## INFO  [16:04:13.507] [bbotk]                                    uhash
## INFO  [16:04:13.507] [bbotk]  0f54a269-7b09-41f7-8ad8-2680a223a6bb
## INFO  [16:04:13.509] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:13.599] [bbotk] Result of batch 18:
## INFO  [16:04:13.601] [bbotk]       cp minsplit classif.ce runtime_learners
## INFO  [16:04:13.601] [bbotk]  0.07525        1     0.2266            0.013
## INFO  [16:04:13.601] [bbotk]                                    uhash
## INFO  [16:04:13.601] [bbotk]  e97798d4-67aa-4db6-a1c6-84dae638ba49
## INFO  [16:04:13.602] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:13.697] [bbotk] Result of batch 19:
## INFO  [16:04:13.699] [bbotk]   cp minsplit classif.ce runtime_learners
## INFO  [16:04:13.699] [bbotk]  0.1        1     0.2656             0.02 5ee61abd-d52e-4af0-
## INFO  [16:04:13.700] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:13.785] [bbotk] Result of batch 20:
## INFO  [16:04:13.787] [bbotk]      cp minsplit classif.ce runtime_learners
## INFO  [16:04:13.787] [bbotk]  0.0505        8     0.2266            0.013
```

```
## INFO  [16:04:13.787] [bbotk]                                  uhash
## INFO  [16:04:13.787] [bbotk]  d595a803-aea3-43ea-b587-9a4fb3adb2af
## INFO  [16:04:13.793] [bbotk] Finished optimizing after 20 evaluation(s)
## INFO  [16:04:13.794] [bbotk] Result:
## INFO  [16:04:13.795] [bbotk]      cp minsplit learner_param_vals  x_domain classif.ce
## INFO  [16:04:13.795] [bbotk]  0.0505        3          <list[3]> <list[2]>     0.2266
```

```
##        cp minsplit learner_param_vals  x_domain classif.ce
## 1: 0.0505        3          <list[3]> <list[2]>     0.2266
```

```
instance$result_learner_param_vals
```

```
## $xval
## [1] 0
##
## $cp
## [1] 0.0505
##
## $minsplit
## [1] 3
```

```
instance$result_y
```

```
## classif.ce
##     0.2266
```

You can investigate all of the evaluations that were performed; they are stored in the archive of the TuningInstanceSingleCrit and can be accessed by using `as.data.table()`:

```
as.data.table(instance$archive)
```

```
##          cp minsplit classif.ce x_domain_cp x_domain_minsplit runtime_learners
##  1: 0.00100        5     0.2734     0.00100                 5            0.022
##  2: 0.05050        3     0.2266     0.05050                 3            0.013
##  3: 0.00100        1     0.2773     0.00100                 1            0.014
##  4: 0.02575        3     0.2305     0.02575                 3            0.013
##  5: 0.00100        3     0.2734     0.00100                 3            0.014
##  6: 0.05050        5     0.2266     0.05050                 5            0.013
##  7: 0.10000        8     0.2656     0.10000                 8            0.012
##  8: 0.02575        5     0.2305     0.02575                 5            0.014
##  9: 0.05050       10     0.2266     0.05050                10            0.015
## 10: 0.02575        8     0.2305     0.02575                 8            0.013
## 11: 0.07525       10     0.2266     0.07525                10            0.013
## 12: 0.10000        3     0.2656     0.10000                 3            0.015
## 13: 0.00100       10     0.2891     0.00100                10            0.021
## 14: 0.02575        1     0.2305     0.02575                 1            0.014
## 15: 0.02575       10     0.2305     0.02575                10            0.014
```

```
## 16: 0.10000       10     0.2656     0.10000                10            0.013
## 17: 0.07525        3     0.2266     0.07525                 3            0.013
## 18: 0.07525        1     0.2266     0.07525                 1            0.013
## 19: 0.10000        1     0.2656     0.10000                 1            0.020
## 20: 0.05050        8     0.2266     0.05050                 8            0.013
##                timestamp batch_nr      resample_result
##  1: 2021-11-24 16:04:11        1 <ResampleResult[22]>
##  2: 2021-11-24 16:04:12        2 <ResampleResult[22]>
##  3: 2021-11-24 16:04:12        3 <ResampleResult[22]>
##  4: 2021-11-24 16:04:12        4 <ResampleResult[22]>
##  5: 2021-11-24 16:04:12        5 <ResampleResult[22]>
##  6: 2021-11-24 16:04:12        6 <ResampleResult[22]>
##  7: 2021-11-24 16:04:12        7 <ResampleResult[22]>
##  8: 2021-11-24 16:04:12        8 <ResampleResult[22]>
##  9: 2021-11-24 16:04:12        9 <ResampleResult[22]>
## 10: 2021-11-24 16:04:12       10 <ResampleResult[22]>
## 11: 2021-11-24 16:04:12       11 <ResampleResult[22]>
## 12: 2021-11-24 16:04:13       12 <ResampleResult[22]>
## 13: 2021-11-24 16:04:13       13 <ResampleResult[22]>
## 14: 2021-11-24 16:04:13       14 <ResampleResult[22]>
## 15: 2021-11-24 16:04:13       15 <ResampleResult[22]>
## 16: 2021-11-24 16:04:13       16 <ResampleResult[22]>
## 17: 2021-11-24 16:04:13       17 <ResampleResult[22]>
## 18: 2021-11-24 16:04:13       18 <ResampleResult[22]>
## 19: 2021-11-24 16:04:13       19 <ResampleResult[22]>
## 20: 2021-11-24 16:04:13       20 <ResampleResult[22]>
```

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

The associated resampling iterations can be accessed in the `BenchmarkResult` of the tuning instance:

```
instance$archive$benchmark_result
```

```
## <BenchmarkResult> of 20 rows with 20 resampling runs
##  nr task_id    learner_id resampling_id iters warnings errors
##   1    pima classif.rpart       holdout     1        0      0
##   2    pima classif.rpart       holdout     1        0      0
##   3    pima classif.rpart       holdout     1        0      0
##   4    pima classif.rpart       holdout     1        0      0
##   5    pima classif.rpart       holdout     1        0      0
##   6    pima classif.rpart       holdout     1        0      0
##   7    pima classif.rpart       holdout     1        0      0
##   8    pima classif.rpart       holdout     1        0      0
##   9    pima classif.rpart       holdout     1        0      0
##  10    pima classif.rpart       holdout     1        0      0
##  11    pima classif.rpart       holdout     1        0      0
##  12    pima classif.rpart       holdout     1        0      0
```

```
##  13    pima classif.rpart       holdout     1        0      0
##  14    pima classif.rpart       holdout     1        0      0
##  15    pima classif.rpart       holdout     1        0      0
##  16    pima classif.rpart       holdout     1        0      0
##  17    pima classif.rpart       holdout     1        0      0
##  18    pima classif.rpart       holdout     1        0      0
##  19    pima classif.rpart       holdout     1        0      0
##  20    pima classif.rpart       holdout     1        0      0
```

The `uhash` column links the resampling iterations to the evaluated configurations stored in `instance$archive$data`. This allows e.g. to score the included `ResampleResult`s on a different performance measure.

```
instance$archive$benchmark_result$score(msr("classif.acc"))
```

```
##                                   uhash nr               task task_id
##  1: f941926a-ce9e-483b-9dce-583531452e93  1 <TaskClassif[49]>    pima
##  2: a1e2401b-c5e0-4ee3-a9b6-74ed04363b14  2 <TaskClassif[49]>    pima
##  3: 771763d8-b099-439f-b1e7-aec5000451f2  3 <TaskClassif[49]>    pima
##  4: 3c9a197a-abff-4ee3-a4c1-95ca3955b708  4 <TaskClassif[49]>    pima
##  5: d4444d88-608b-443e-9c8d-7c78d8e67021  5 <TaskClassif[49]>    pima
##  6: 6af4bf40-ee4a-4487-af74-7d591156a76d  6 <TaskClassif[49]>    pima
##  7: f1f648a8-b684-4b0f-9ab5-1a515876981b  7 <TaskClassif[49]>    pima
##  8: 58bea9f2-1d85-4dae-a822-8846a71c226a  8 <TaskClassif[49]>    pima
##  9: 01fa1fcb-d4b9-4270-bd21-78a3a4a9dd85  9 <TaskClassif[49]>    pima
## 10: 4e38076d-99bb-4146-b99c-20039054372c 10 <TaskClassif[49]>    pima
## 11: 245a9e6c-8325-47f3-9853-2250ec7c1a84 11 <TaskClassif[49]>    pima
## 12: 64c4f2d9-eb99-4d3d-a866-22393c675328 12 <TaskClassif[49]>    pima
## 13: e765b2da-c951-4e63-9b5f-679d7839e7e7 13 <TaskClassif[49]>    pima
## 14: 62812c0d-6864-44be-adb8-0a8e88d09f68 14 <TaskClassif[49]>    pima
## 15: f2e0a760-5a01-488c-ad56-a0f0548d1ec4 15 <TaskClassif[49]>    pima
## 16: cc440ffd-3388-4d38-8b04-2623126b94f5 16 <TaskClassif[49]>    pima
## 17: 0f54a269-7b09-41f7-8ad8-2680a223a6bb 17 <TaskClassif[49]>    pima
## 18: e97798d4-67aa-4db6-a1c6-84dae638ba49 18 <TaskClassif[49]>    pima
## 19: 5ee61abd-d52e-4af0-9b97-2b9834bf9cbb 19 <TaskClassif[49]>    pima
## 20: d595a803-aea3-43ea-b587-9a4fb3adb2af 20 <TaskClassif[49]>    pima
##                        learner    learner_id              resampling
##  1: <LearnerClassifRpart[37]> classif.rpart <ResamplingHoldout[19]>
##  2: <LearnerClassifRpart[37]> classif.rpart <ResamplingHoldout[19]>
##  3: <LearnerClassifRpart[37]> classif.rpart <ResamplingHoldout[19]>
##  4: <LearnerClassifRpart[37]> classif.rpart <ResamplingHoldout[19]>
##  5: <LearnerClassifRpart[37]> classif.rpart <ResamplingHoldout[19]>
##  6: <LearnerClassifRpart[37]> classif.rpart <ResamplingHoldout[19]>
##  7: <LearnerClassifRpart[37]> classif.rpart <ResamplingHoldout[19]>
##  8: <LearnerClassifRpart[37]> classif.rpart <ResamplingHoldout[19]>
##  9: <LearnerClassifRpart[37]> classif.rpart <ResamplingHoldout[19]>
## 10: <LearnerClassifRpart[37]> classif.rpart <ResamplingHoldout[19]>
## 11: <LearnerClassifRpart[37]> classif.rpart <ResamplingHoldout[19]>
## 12: <LearnerClassifRpart[37]> classif.rpart <ResamplingHoldout[19]>
```

```
## 13: <LearnerClassifRpart[37]> classif.rpart <ResamplingHoldout[19]>
## 14: <LearnerClassifRpart[37]> classif.rpart <ResamplingHoldout[19]>
## 15: <LearnerClassifRpart[37]> classif.rpart <ResamplingHoldout[19]>
## 16: <LearnerClassifRpart[37]> classif.rpart <ResamplingHoldout[19]>
## 17: <LearnerClassifRpart[37]> classif.rpart <ResamplingHoldout[19]>
## 18: <LearnerClassifRpart[37]> classif.rpart <ResamplingHoldout[19]>
## 19: <LearnerClassifRpart[37]> classif.rpart <ResamplingHoldout[19]>
## 20: <LearnerClassifRpart[37]> classif.rpart <ResamplingHoldout[19]>
##     resampling_id iteration              prediction classif.acc
##  1:       holdout         1 <PredictionClassif[20]>      0.7266
##  2:       holdout         1 <PredictionClassif[20]>      0.7734
##  3:       holdout         1 <PredictionClassif[20]>      0.7227
##  4:       holdout         1 <PredictionClassif[20]>      0.7695
##  5:       holdout         1 <PredictionClassif[20]>      0.7266
##  6:       holdout         1 <PredictionClassif[20]>      0.7734
##  7:       holdout         1 <PredictionClassif[20]>      0.7344
##  8:       holdout         1 <PredictionClassif[20]>      0.7695
##  9:       holdout         1 <PredictionClassif[20]>      0.7734
## 10:       holdout         1 <PredictionClassif[20]>      0.7695
## 11:       holdout         1 <PredictionClassif[20]>      0.7734
## 12:       holdout         1 <PredictionClassif[20]>      0.7344
## 13:       holdout         1 <PredictionClassif[20]>      0.7109
## 14:       holdout         1 <PredictionClassif[20]>      0.7695
## 15:       holdout         1 <PredictionClassif[20]>      0.7695
## 16:       holdout         1 <PredictionClassif[20]>      0.7344
## 17:       holdout         1 <PredictionClassif[20]>      0.7734
## 18:       holdout         1 <PredictionClassif[20]>      0.7734
## 19:       holdout         1 <PredictionClassif[20]>      0.7344
## 20:       holdout         1 <PredictionClassif[20]>      0.7734
```

Now we can take the optimized hyperparameters, set them for the previously-created `Learner`, and train it on the full dataset.

```
learner$param_set$values = instance$result_learner_param_vals
learner$train(task)
```

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

4.1.4 Automating the Tuning

We can automate this entire process in `mlr3` so that learners are tuned transparently, without the need to extract information on the best hyperparameter settings at the end.. The `AutoTuner` wraps a learner and augments it with an automatic tuning process for a given set of hyperparameters. Because the `AutoTuner` itself inherits from the `Learner` base class, it can be used like any other

learner. In keeping with our example above, we create a classification learner that tunes itself automatically. This classification tree learner 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:

```
learner = lrn("classif.rpart")
search_space = ps(
  cp = p_dbl(lower = 0.001, upper = 0.1),
  minsplit = p_int(lower = 1, upper = 10)
)
terminator = trm("evals", n_evals = 10)
tuner = tnr("random_search")

at = AutoTuner$new(
  learner = learner,
  resampling = rsmp("holdout"),
  measure = msr("classif.ce"),
  search_space = search_space,
  terminator = terminator,
  tuner = tuner
)
at
```

```
## <AutoTuner:classif.rpart.tuned>
## * Model: -
## * Search Space:
## <ParamSet>
##          id    class lower upper nlevels        default value
## 1:       cp ParamDbl 0.001   0.1     Inf <NoDefault[3]>
## 2: minsplit ParamInt 1.000  10.0      10 <NoDefault[3]>
## * Packages: mlr3, 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. The differnce to a normal learner is that `$train()` runs the tuning, which will take longer than a normal training process.

```
at$train(task)
```

```
## INFO  [16:04:14.312] [bbotk] Starting to optimize 2 parameter(s) with '<OptimizerRandomSea
## INFO  [16:04:14.330] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:14.425] [bbotk] Result of batch 1:
## INFO  [16:04:14.428] [bbotk]       cp minsplit classif.ce runtime_learners
## INFO  [16:04:14.428] [bbotk]  0.05837        1     0.2773            0.014
## INFO  [16:04:14.428] [bbotk]                                       uhash
## INFO  [16:04:14.428] [bbotk]  fc9e5869-173c-45b7-8200-177c19f251fa
## INFO  [16:04:14.432] [bbotk] Evaluating 1 configuration(s)
```

```
## INFO  [16:04:14.518] [bbotk] Result of batch 2:
## INFO  [16:04:14.520] [bbotk]       cp minsplit classif.ce runtime_learners
## INFO  [16:04:14.520] [bbotk]  0.02861        2     0.2188            0.013
## INFO  [16:04:14.520] [bbotk]                                   uhash
## INFO  [16:04:14.520] [bbotk]  3ab85af7-e901-4a3d-be8b-6167aaa30b2b
## INFO  [16:04:14.523] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:14.609] [bbotk] Result of batch 3:
## INFO  [16:04:14.611] [bbotk]       cp minsplit classif.ce runtime_learners
## INFO  [16:04:14.611] [bbotk]  0.01703        3     0.2383            0.013
## INFO  [16:04:14.611] [bbotk]                                   uhash
## INFO  [16:04:14.611] [bbotk]  8705dfb5-02aa-4752-ad82-43a3a8c356da
## INFO  [16:04:14.615] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:14.704] [bbotk] Result of batch 4:
## INFO  [16:04:14.706] [bbotk]     cp minsplit classif.ce runtime_learners
## INFO  [16:04:14.706] [bbotk]  0.041        2     0.2773            0.012
## INFO  [16:04:14.706] [bbotk]                                   uhash
## INFO  [16:04:14.706] [bbotk]  71c75843-42c9-4926-b534-61c524008228
## INFO  [16:04:14.711] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:14.810] [bbotk] Result of batch 5:
## INFO  [16:04:14.812] [bbotk]       cp minsplit classif.ce runtime_learners
## INFO  [16:04:14.812] [bbotk]  0.03785        5     0.2773            0.014
## INFO  [16:04:14.812] [bbotk]                                   uhash
## INFO  [16:04:14.812] [bbotk]  0a5567a8-8909-421d-80e5-4a55b3daefc7
## INFO  [16:04:14.816] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:14.914] [bbotk] Result of batch 6:
## INFO  [16:04:14.916] [bbotk]       cp minsplit classif.ce runtime_learners
## INFO  [16:04:14.916] [bbotk]  0.03003        4     0.2188            0.015
## INFO  [16:04:14.916] [bbotk]                                   uhash
## INFO  [16:04:14.916] [bbotk]  d51a3ce4-167b-472e-95bd-eba86932d347
## INFO  [16:04:14.920] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:15.037] [bbotk] Result of batch 7:
## INFO  [16:04:15.039] [bbotk]       cp minsplit classif.ce runtime_learners
## INFO  [16:04:15.039] [bbotk]  0.07742       10     0.2773            0.012
## INFO  [16:04:15.039] [bbotk]                                   uhash
## INFO  [16:04:15.039] [bbotk]  dddc8085-7760-4a36-bddc-07a8d4234be7
## INFO  [16:04:15.042] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:15.136] [bbotk] Result of batch 8:
## INFO  [16:04:15.138] [bbotk]       cp minsplit classif.ce runtime_learners
## INFO  [16:04:15.138] [bbotk]  0.05818       10     0.2773            0.014
## INFO  [16:04:15.138] [bbotk]                                   uhash
## INFO  [16:04:15.138] [bbotk]  44fabc0f-2dbb-4e50-8949-a72f09ab4502
## INFO  [16:04:15.142] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:15.228] [bbotk] Result of batch 9:
## INFO  [16:04:15.230] [bbotk]       cp minsplit classif.ce runtime_learners
## INFO  [16:04:15.230] [bbotk]  0.07665       10     0.2773            0.012
## INFO  [16:04:15.230] [bbotk]                                   uhash
## INFO  [16:04:15.230] [bbotk]  f2972e4d-ab27-4d79-b3ae-1084b8527450
## INFO  [16:04:15.234] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:15.323] [bbotk] Result of batch 10:
## INFO  [16:04:15.325] [bbotk]       cp minsplit classif.ce runtime_learners
```

```
## INFO  [16:04:15.325] [bbotk]  0.08818        2     0.2773            0.012
## INFO  [16:04:15.325] [bbotk]                                     uhash
## INFO  [16:04:15.325] [bbotk]  c2b725b8-b698-46d7-a59d-e44b26484995
## INFO  [16:04:15.333] [bbotk] Finished optimizing after 10 evaluation(s)
## INFO  [16:04:15.334] [bbotk] Result:
## INFO  [16:04:15.336] [bbotk]       cp minsplit learner_param_vals  x_domain classif.ce
## INFO  [16:04:15.336] [bbotk]  0.02861        2          <list[3]> <list[2]>     0.2188
```

We can also pass it to `resample()` and `benchmark()`, just like any other learner. This would result in a nested resampling.

4.2 Tuning Search Spaces

When running an optimization, it is important to inform the tuning algorithm about what hyperparameters are valid. Here the names, types, and valid ranges of each hyperparameter are important. All this information is communicated with objects of the class `ParamSet`, which is defined in paradox. While it is possible to create `ParamSet`-objects using its `$new`-constructor, it is much shorter and readable to use the `ps`-shortcut, which will be presented here. For an in-depth description of paradox and its classes, see the `paradox` chapter.

Note, that `ParamSet` objects exist in two contexts. First, `ParamSet`-objects are used to define the space of valid parameter setting for a learner (and other objects). Second, they are used to define a search space for tuning. We are mainly interested in the latter. For an example we can consider the `minsplit` parameter of the `classif.rpart Learner`. The `ParamSet` associated with the learner has a lower but *no* upper bound. However, for tuning the value, a lower *and* upper bound must be given because tuning search spaces need to be bounded. For `Learner` or `PipeOp` objects, typically "unbounded" `ParamSets` are used. Here, however, we will mainly focus on creating "bounded" `ParamSets` that can be used for tuning. See the in-depth `paradox` chapter for more details on using `ParamSets` to define parameter ranges for use-cases besides tuning.

4.2.1 Creating `ParamSets`

An empty `ParamSet` – not yet very useful – can be constructed using just the `ps` call:

```
library("mlr3verse")

search_space = ps()
print(search_space)
```

```
## <ParamSet>
## Empty.
```

`ps` takes named `Domain` arguments that are turned into parameters. A possible search space for the `"classif.svm"` learner could for example be:

```
search_space = ps(
  cost = p_dbl(lower = 0.1, upper = 10),
  kernel = p_fct(levels = c("polynomial", "radial"))
)
print(search_space)
```

```
## <ParamSet>
##        id    class lower upper nlevels        default value
## 1:   cost ParamDbl   0.1    10     Inf <NoDefault[3]>
## 2: kernel ParamFct    NA    NA       2 <NoDefault[3]>
```

There are five domain constructors that produce a parameters when given to ps:

Constructor	Description	Is bounded?	Underlying Class
p_dbl	Real valued parameter ("double")	When upper and lower are given	ParamDbl
p_int	Integer parameter	When upper and lower are given	ParamInt
p_fct	Discrete valued parameter ("factor")	Always	ParamFct
p_lgl	Logical / Boolean parameter	Always	ParamLgl
p_uty	Untyped parameter	Never	ParamUty

These domain constructors each take some of the following arguments:

- **lower**, **upper**: lower and upper bound of numerical parameters (p_dbl and p_int). These need to be given to get bounded parameter spaces valid for tuning.
- **levels**: Allowed categorical values for p_fct parameters. Required argument for p_fct. See below for more details on this parameter.
- **trafo**: transformation function, see below.
- **depends**: dependencies, see below.
- **tags**: Further information about a parameter, used for example by the hyperband tuner.
- **default**: Value corresponding to default behavior when the parameter is not given. Not used for tuning search spaces.
- **special_vals**: Valid values besides the normally accepted values for a parameter. Not used for tuning search spaces.
- **custom_check**: Function that checks whether a value given to p_uty is valid. Not used for tuning search spaces.

The lower, upper, or levels parameters are always at the first (or second, for upper) position of the respective constructors, so it is preferred to omit them when defining a ParamSet, for improved conciseness:

```
search_space = ps(cost = p_dbl(0.1, 10), kernel = p_fct(c("polynomial", "radial")))
```

4.2.2 Transformations (`trafo`)

We can use the paradox function `generate_design_grid` to look at the values that would be evaluated by grid search. (We are using `rbindlist()` here because the result of `$transpose()` is a list that is harder to read. If we didn't use `$transpose()`, on the other hand, the transformations that we investigate here are not applied.)

```
library("data.table")
rbindlist(generate_design_grid(search_space, 3)$transpose())
```

```
##     cost     kernel
## 1:  0.10 polynomial
## 2:  0.10     radial
## 3:  5.05 polynomial
## 4:  5.05     radial
## 5: 10.00 polynomial
## 6: 10.00     radial
```

We notice that the `cost` parameter is taken on a linear scale. We assume, however, that the difference of cost between `0.1` and `1` should have a similar effect as the difference between `1` and `10`. Therefore it makes more sense to tune it on a *logarithmic scale*. This is done by using a **transformation** (`trafo`). This is a function that is applied to a parameter after it has been sampled by the tuner. We can tune `cost` on a logarithmic scale by sampling on the linear scale `[-1, 1]` and computing `10^x` from that value.

```
search_space = ps(
  cost = p_dbl(-1, 1, trafo = function(x) 10^x),
  kernel = p_fct(c("polynomial", "radial"))
)
rbindlist(generate_design_grid(search_space, 3)$transpose())
```

```
##    cost     kernel
## 1:  0.1 polynomial
## 2:  0.1     radial
## 3:  1.0 polynomial
## 4:  1.0     radial
## 5: 10.0 polynomial
## 6: 10.0     radial
```

It is even possible to attach another transformation to the `ParamSet` as a whole that gets executed after individual parameter's transformations were performed. It is given through the `.extra_trafo` argument and should be a function with parameters `x` and `param_set` that takes a list of parameter values in `x` and returns a modified list. This transformation can access all parameter values of an evaluation and modify them with interactions. It is even possible to add or remove parameters. (The following is a bit of a silly example.)

```
search_space = ps(
  cost = p_dbl(-1, 1, trafo = function(x) 10^x),
  kernel = p_fct(c("polynomial", "radial")),
  .extra_trafo = function(x, param_set) {
    if (x$kernel == "polynomial") {
      x$cost = x$cost * 2
    }
    x
  }
)
rbindlist(generate_design_grid(search_space, 3)$transpose())
```

```
##    cost     kernel
## 1:  0.2 polynomial
## 2:  0.1     radial
## 3:  2.0 polynomial
## 4:  1.0     radial
## 5: 20.0 polynomial
## 6: 10.0     radial
```

The available types of search space parameters are limited: continuous, integer, discrete, and logical scalars. There are many machine learning algorithms, however, that take parameters of other types, for example vectors or functions. These can not be defined in a search space `ParamSet`, and they are often given as `ParamUty` in the `Learner`'s `ParamSet`. When trying to tune over these hyperparameters, it is necessary to perform a Transformation that changes the type of a parameter.

An example is the `class.weights` parameter of the SVM, which takes a named vector of class weights with one entry for each target class. The trafo that would tune `class.weights` for the `tsk("spam")` dataset could be:

```
search_space = ps(
  class.weights = p_dbl(0.1, 0.9, trafo = function(x) c(spam = x, nonspam = 1 - x))
)
generate_design_grid(search_space, 3)$transpose()
```

```
## [[1]]
## [[1]]$class.weights
##    spam nonspam
##     0.1     0.9
##
##
## [[2]]
## [[2]]$class.weights
##    spam nonspam
##     0.5     0.5
##
##
## [[3]]
## [[3]]$class.weights
```

```
##    spam nonspam
##     0.9     0.1
```

(We are omitting `rbindlist()` in this example because it breaks the vector valued return elements.)

4.2.3 Automatic Factor Level Transformation

A common use-case is the necessity to specify a list of values that should all be tried (or sampled from). It may be the case that a hyperparameter accepts function objects as values and a certain list of functions should be tried. Or it may be that a choice of special numeric values should be tried. For this, the `p_fct` constructor's `level` argument may be a value that is not a `character` vector, but something else. If, for example, only the values `0.1`, `3`, and `10` should be tried for the `cost` parameter, even when doing random search, then the following search space would achieve that:

```
search_space = ps(
  cost = p_fct(c(0.1, 3, 10)),
  kernel = p_fct(c("polynomial", "radial"))
)
rbindlist(generate_design_grid(search_space, 3)$transpose())
```

```
##    cost     kernel
## 1:  0.1 polynomial
## 2:  0.1     radial
## 3:  3.0 polynomial
## 4:  3.0     radial
## 5: 10.0 polynomial
## 6: 10.0     radial
```

This is equivalent to the following:

```
search_space = ps(
  cost = p_fct(c("0.1", "3", "10"),
    trafo = function(x) list(`0.1` = 0.1, `3` = 3, `10` = 10)[[x]]),
  kernel = p_fct(c("polynomial", "radial"))
)
rbindlist(generate_design_grid(search_space, 3)$transpose())
```

```
##    cost     kernel
## 1:  0.1 polynomial
## 2:  0.1     radial
## 3:  3.0 polynomial
## 4:  3.0     radial
## 5: 10.0 polynomial
## 6: 10.0     radial
```

This may seem silly, but makes sense when considering that factorial tuning parameters are always `character` values:

```
search_space = ps(
  cost = p_fct(c(0.1, 3, 10)),
  kernel = p_fct(c("polynomial", "radial"))
)
typeof(search_space$params$cost$levels)
```

```
## [1] "character"
```

Be aware that this results in an "unordered" hyperparameter, however. Tuning algorithms that make use of ordering information of parameters, like genetic algorithms or model based optimization, will perform worse when this is done. For these algorithms, it may make more sense to define a `p_dbl` or `p_int` with a more fitting trafo.

The `class.weights` case from above can also be implemented like this, if there are only a few candidates of `class.weights` vectors that should be tried. Note that the `levels` argument of `p_fct` must be named if there is no easy way for `as.character()` to create names:

```
search_space = ps(
  class.weights = p_fct(
    list(
      candidate_a = c(spam = 0.5, nonspam = 0.5),
      candidate_b = c(spam = 0.3, nonspam = 0.7)
    )
  )
)
generate_design_grid(search_space)$transpose()
```

```
## [[1]]
## [[1]]$class.weights
##    spam nonspam
##     0.5     0.5
##
##
## [[2]]
## [[2]]$class.weights
##    spam nonspam
##     0.3     0.7
```

4.2.4 Parameter Dependencies (`depends`)

Some parameters are only relevant when another parameter has a certain value, or one of several values. The SVM, for example, has the `degree` parameter that is only valid when `kernel` is `"polynomial"`. This can be specified using the `depends` argument. It is an expression that must involve other parameters and be of the form `<param> == <scalar>`, `<param> %in% <vector>`, or multiple of these chained by `&&`. To tune the `degree` parameter, one would need to do the following:

```
search_space = ps(
  cost = p_dbl(-1, 1, trafo = function(x) 10^x),
  kernel = p_fct(c("polynomial", "radial")),
  degree = p_int(1, 3, depends = kernel == "polynomial")
)
rbindlist(generate_design_grid(search_space, 3)$transpose(), fill = TRUE)
```

```
##     cost     kernel degree
##  1:  0.1 polynomial      1
##  2:  0.1 polynomial      2
##  3:  0.1 polynomial      3
##  4:  0.1     radial     NA
##  5:  1.0 polynomial      1
##  6:  1.0 polynomial      2
##  7:  1.0 polynomial      3
##  8:  1.0     radial     NA
##  9: 10.0 polynomial      1
## 10: 10.0 polynomial      2
## 11: 10.0 polynomial      3
## 12: 10.0     radial     NA
```

4.2.5 Creating Tuning ParamSets from other ParamSets

Having to define a tuning ParamSet for a Learner that already has parameter set information may seem unnecessarily tedious, and there is indeed a way to create tuning ParamSets from a Learner's ParamSet, making use of as much information as already available.

This is done by setting values of a Learner's ParamSet to so-called TuneTokens, constructed with a to_tune call. This can be done in the same way that other hyperparameters are set to specific values. It can be understood as the hyperparameters being tagged for later tuning. The resulting ParamSet used for tuning can be retrieved using the `$search_space()` method.

```
learner = lrn("classif.svm")
learner$param_set$values$kernel = "polynomial" # for example
learner$param_set$values$degree = to_tune(lower = 1, upper = 3)

print(learner$param_set$search_space())
```

```
## <ParamSet>
##        id    class lower upper nlevels        default value
## 1: degree ParamInt     1     3       3 <NoDefault[3]>
```

```
rbindlist(generate_design_grid(learner$param_set$search_space(), 3)$transpose())
```

```
##    degree
## 1:      1
## 2:      2
## 3:      3
```

It is possible to omit `lower` here, because it can be inferred from the lower bound of the `degree` parameter itself. For other parameters, that are already bounded, it is possible to not give any bounds at all, because their ranges are already bounded. An example is the logical `shrinking` hyperparameter:

```
learner$param_set$values$shrinking = to_tune()

print(learner$param_set$search_space())
```

```
## <ParamSet>
##           id    class lower upper nlevels        default value
## 1:    degree ParamInt     1     3       3 <NoDefault[3]>
## 2: shrinking ParamLgl    NA    NA       2           TRUE
```

```
rbindlist(generate_design_grid(learner$param_set$search_space(), 3)$transpose())
```

```
##    degree shrinking
## 1:      1      TRUE
## 2:      1     FALSE
## 3:      2      TRUE
## 4:      2     FALSE
## 5:      3      TRUE
## 6:      3     FALSE
```

`to_tune` can also be constructed with a `Domain` object, i.e. something constructed with a `p_***` call. This way it is possible to tune continuous parameters with discrete values, or to give trafos or dependencies. One could, for example, tune the `cost` as above on three given special values, and introduce a dependency of `shrinking` on it. Notice that a short form for `to_tune(<levels>)` is a short form of `to_tune(p_fct(<levels>))`. (When introducing the dependency, we need to use the `degree` value from *before* the implicit trafo, which is the name or `as.character()` of the respective value, here `"val2"`!)

```
learner$param_set$values$type = "C-classification" # needs to be set because of a bug in paradox
learner$param_set$values$cost = to_tune(c(val1 = 0.3, val2 = 0.7))
learner$param_set$values$shrinking = to_tune(p_lgl(depends = cost == "val2"))

print(learner$param_set$search_space())
```

```
## <ParamSet>
##           id    class lower upper nlevels        default parents value
## 1:      cost ParamFct    NA    NA       2 <NoDefault[3]>
## 2:    degree ParamInt     1     3       3 <NoDefault[3]>
## 3: shrinking ParamLgl    NA    NA       2 <NoDefault[3]>    cost
## Trafo is set.
```

```
rbindlist(generate_design_grid(learner$param_set$search_space(), 3)$transpose(), fill = TRUE)
```

```
##    degree cost shrinking
## 1:      1  0.3        NA
## 2:      1  0.7      TRUE
## 3:      1  0.7     FALSE
## 4:      2  0.3        NA
## 5:      2  0.7      TRUE
## 6:      2  0.7     FALSE
## 7:      3  0.3        NA
## 8:      3  0.7      TRUE
## 9:      3  0.7     FALSE
```

The `search_space()` picks up dependencies fromt the underlying `ParamSet` automatically. So if the `kernel` is tuned, then `degree` automatically gets the dependency on it, without us having to specify that. (Here we reset `cost` and `shrinking` to `NULL` for the sake of clarity of the generated output.)

```
learner$param_set$values$cost = NULL
learner$param_set$values$shrinking = NULL
learner$param_set$values$kernel = to_tune(c("polynomial", "radial"))

print(learner$param_set$search_space())
```

```
## <ParamSet>
##        id    class lower upper nlevels        default parents value
## 1: degree ParamInt     1     3       3 <NoDefault[3]>  kernel
## 2: kernel ParamFct    NA    NA       2 <NoDefault[3]>
```

```
rbindlist(generate_design_grid(learner$param_set$search_space(), 3)$transpose(), fill = TRUE)
```

```
##        kernel degree
## 1: polynomial      1
## 2: polynomial      2
## 3: polynomial      3
## 4:     radial     NA
```

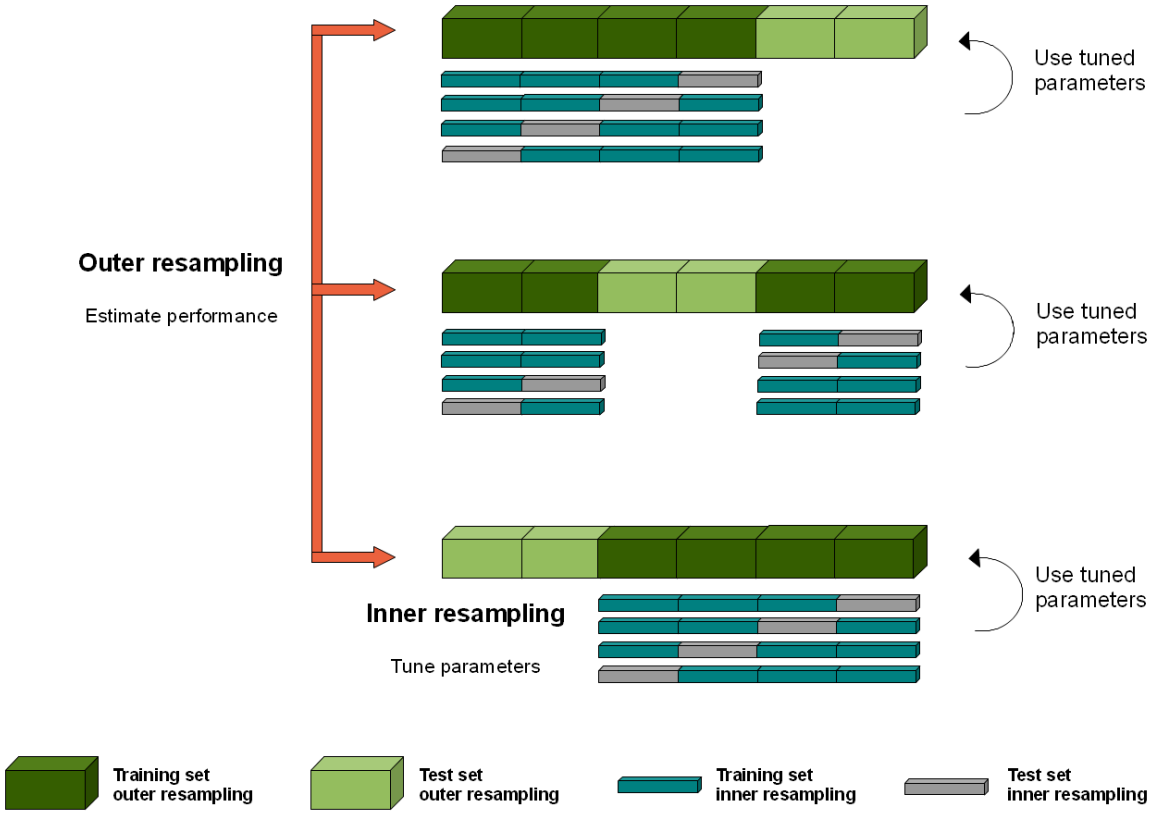
It is even possible to define whole `ParamSet`s that get tuned over for a single parameter. This may be especially useful for vector hyperparameters that should be searched along multiple dimensions. This `ParamSet` must, however, have an `.extra_trafo` that returns a list with a single element, because it corresponds to a single hyperparameter that is being tuned. Suppose the `class.weights` hyperparameter should be tuned along two dimensions:

```
learner$param_set$values$class.weights = to_tune(
  ps(spam = p_dbl(0.1, 0.9), nonspam = p_dbl(0.1, 0.9),
     .extra_trafo = function(x, param_set) list(c(spam = x$spam, nonspam = x$nonspam))
))
head(generate_design_grid(learner$param_set$search_space(), 3)$transpose(), 3)
```

```
## [[1]]
## [[1]]$kernel
## [1] "polynomial"
##
## [[1]]$degree
## [1] 1
##
## [[1]]$class.weights
##    spam nonspam
##     0.1     0.1
##
##
## [[2]]
## [[2]]$kernel
## [1] "polynomial"
##
## [[2]]$degree
## [1] 1
##
## [[2]]$class.weights
##    spam nonspam
##     0.1     0.5
##
##
## [[3]]
## [[3]]$kernel
## [1] "polynomial"
##
## [[3]]$degree
## [1] 1
##
## [[3]]$class.weights
##    spam nonspam
##     0.1     0.9
```

4.3 Nested Resampling

Evaluating a machine learning model often requires an additional layer of resampling when hyperparameters or features have to be selected. Nested resampling separates these model selection steps from the process estimating the performance of the model. If the same data is used for the model selection steps and the evaluation of the model itself, the resulting performance estimate of the model might be severely biased. One reason is that the repeated evaluation of the model on the test data could leak information about its structure into the model, what results in over-optimistic performance estimates. Keep in mind that nested resampling is a statistical procedure to estimate the predictive performance of the model trained on the full dataset. Nested resampling is not a procedure to select optimal hyperparameters. The resampling produces many hyperparameter configurations which should be not used to construct a final model (Simon 2007).



The graphic above illustrates nested resampling for hyperparameter 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. The aggregated performance on the outer test sets is the unbiased performance estimate of the model.

4.3.1 Execution

The previous section examined the optimization of a simple classification tree on the `mlr_tasks_pima`. We continue the example and estimate the predictive performance of the model with nested resampling.

We use a 4-fold cross-validation in the inner resampling loop. The `AutoTuner` executes the hyperparameter tuning and is stopped after 5 evaluations. The hyperparameter configurations are proposed by grid search.

```
library("mlr3verse")

learner = lrn("classif.rpart")
resampling = rsmp("cv", folds = 4)
measure = msr("classif.ce")
```

```
search_space = ps(cp = p_dbl(lower = 0.001, upper = 0.1))
terminator = trm("evals", n_evals = 5)
tuner = tnr("grid_search", resolution = 10)

at = AutoTuner$new(learner, resampling, measure, terminator, tuner, search_space)
```

A 3-fold cross-validation is used in the outer resampling loop. On each of the three outer train sets hyperparameter tuning is done and we receive three optimized hyperparameter configurations. To execute the nested resampling, we pass the `AutoTuner` to the `resample()` function. We have to set `store_models = TRUE` because we need the `AutoTuner` models to investigate the inner tuning.

```
task = tsk("pima")
outer_resampling = rsmp("cv", folds = 3)

rr = resample(task, at, outer_resampling, store_models = TRUE)
```

```
## INFO  [16:04:24.592] [bbotk] Starting to optimize 1 parameter(s) with '<OptimizerGridSearc
## INFO  [16:04:24.633] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:24.831] [bbotk] Result of batch 1:
## INFO  [16:04:24.834] [bbotk]     cp classif.ce runtime_learners
## INFO  [16:04:24.834] [bbotk]  0.078     0.2559            0.054 8a751124-0a01-4d3c-a3ef-e6
## INFO  [16:04:24.836] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:25.036] [bbotk] Result of batch 2:
## INFO  [16:04:25.039] [bbotk]     cp classif.ce runtime_learners
## INFO  [16:04:25.039] [bbotk]  0.023      0.252            0.053 cd2db115-e06e-452e-8f11-b3
## INFO  [16:04:25.041] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:25.235] [bbotk] Result of batch 3:
## INFO  [16:04:25.237] [bbotk]     cp classif.ce runtime_learners
## INFO  [16:04:25.237] [bbotk]  0.034     0.2559            0.055 78d1a27c-24b4-4aa0-97e2-35
## INFO  [16:04:25.239] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:25.447] [bbotk] Result of batch 4:
## INFO  [16:04:25.449] [bbotk]   cp classif.ce runtime_learners
## INFO  [16:04:25.449] [bbotk]  0.1     0.2773           0.066 3d457284-4fc1-4807-8463-7a47
## INFO  [16:04:25.451] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:25.651] [bbotk] Result of batch 5:
## INFO  [16:04:25.654] [bbotk]     cp classif.ce runtime_learners
## INFO  [16:04:25.654] [bbotk]  0.001     0.2559             0.06 a082e907-7e06-468f-936f-5a
## INFO  [16:04:25.661] [bbotk] Finished optimizing after 5 evaluation(s)
## INFO  [16:04:25.661] [bbotk] Result:
## INFO  [16:04:25.663] [bbotk]     cp learner_param_vals  x_domain classif.ce
## INFO  [16:04:25.663] [bbotk]  0.023          <list[2]> <list[1]>      0.252
## INFO  [16:04:25.743] [bbotk] Starting to optimize 1 parameter(s) with '<OptimizerGridSearc
## INFO  [16:04:25.747] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:25.919] [bbotk] Result of batch 1:
## INFO  [16:04:25.922] [bbotk]     cp classif.ce runtime_learners
## INFO  [16:04:25.922] [bbotk]  0.089     0.2695            0.049 03a4d68b-e7c8-4879-9a8c-aa
## INFO  [16:04:25.923] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:26.132] [bbotk] Result of batch 2:
## INFO  [16:04:26.134] [bbotk]     cp classif.ce runtime_learners
```

```
## INFO  [16:04:26.134] [bbotk]  0.001      0.252           0.067 a9c25239-806d-4289-a3de-62
## INFO  [16:04:26.139] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:26.331] [bbotk] Result of batch 3:
## INFO  [16:04:26.333] [bbotk]   cp classif.ce runtime_learners
## INFO  [16:04:26.333] [bbotk]  0.1     0.2695           0.055 0cdb1c66-40c5-44ac-a78e-39fd
## INFO  [16:04:26.335] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:26.518] [bbotk] Result of batch 4:
## INFO  [16:04:26.520] [bbotk]     cp classif.ce runtime_learners
## INFO  [16:04:26.520] [bbotk]  0.023     0.2266           0.052 35a71df4-7e7f-4d0f-8ea7-10
## INFO  [16:04:26.522] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:26.721] [bbotk] Result of batch 5:
## INFO  [16:04:26.723] [bbotk]     cp classif.ce runtime_learners
## INFO  [16:04:26.723] [bbotk]  0.045     0.2637           0.063 e7ad251f-b20f-4810-9fb0-37
## INFO  [16:04:26.728] [bbotk] Finished optimizing after 5 evaluation(s)
## INFO  [16:04:26.729] [bbotk] Result:
## INFO  [16:04:26.730] [bbotk]     cp learner_param_vals  x_domain classif.ce
## INFO  [16:04:26.730] [bbotk]  0.023          <list[2]> <list[1]>     0.2266
## INFO  [16:04:26.799] [bbotk] Starting to optimize 1 parameter(s) with '<OptimizerGridSearc
## INFO  [16:04:26.802] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:27.046] [bbotk] Result of batch 1:
## INFO  [16:04:27.048] [bbotk]     cp classif.ce runtime_learners
## INFO  [16:04:27.048] [bbotk]  0.078     0.2832            0.05 8321d462-1bf0-4f73-9ef9-2b
## INFO  [16:04:27.049] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:27.239] [bbotk] Result of batch 2:
## INFO  [16:04:27.241] [bbotk]     cp classif.ce runtime_learners
## INFO  [16:04:27.241] [bbotk]  0.023      0.248            0.05 9766e0ec-b774-49f2-bce5-eb
## INFO  [16:04:27.243] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:27.445] [bbotk] Result of batch 3:
## INFO  [16:04:27.447] [bbotk]     cp classif.ce runtime_learners
## INFO  [16:04:27.447] [bbotk]  0.067     0.2832           0.061 0c5e30ab-47fe-4b3a-a68b-94
## INFO  [16:04:27.448] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:27.639] [bbotk] Result of batch 4:
## INFO  [16:04:27.641] [bbotk]     cp classif.ce runtime_learners
## INFO  [16:04:27.641] [bbotk]  0.089     0.2832           0.055 83c5b673-4d3b-4c66-93d6-66
## INFO  [16:04:27.642] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:27.820] [bbotk] Result of batch 5:
## INFO  [16:04:27.822] [bbotk]     cp classif.ce runtime_learners
## INFO  [16:04:27.822] [bbotk]  0.056     0.2676           0.051 c1e16efe-182f-4b69-84f3-1d
## INFO  [16:04:27.828] [bbotk] Finished optimizing after 5 evaluation(s)
## INFO  [16:04:27.829] [bbotk] Result:
## INFO  [16:04:27.830] [bbotk]     cp learner_param_vals  x_domain classif.ce
## INFO  [16:04:27.830] [bbotk]  0.023          <list[2]> <list[1]>      0.248
```

You can freely combine different inner and outer resampling strategies. Nested resampling is not restricted to hyperparameter tuning. You can swap the `AutoTuner` for a `AutoFSelector` and estimate the performance of a model which is fitted on an optimized feature subset.

4.3.2 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.

We check the inner tuning results for stable hyperparameters. This means that the selected hyperparameters should not vary too much. We might observe unstable models in this example because the small data set and the low number of resampling iterations might introduces too much randomness. Usually, we aim for the selection of stable hyperparameters for all outer training sets.

```
extract_inner_tuning_results(rr)
```

```
##    iteration    cp classif.ce learner_param_vals  x_domain task_id
## 1:         1 0.023     0.2480          <list[2]> <list[1]>    pima
## 2:         2 0.023     0.2520          <list[2]> <list[1]>    pima
## 3:         3 0.023     0.2266          <list[2]> <list[1]>    pima
##             learner_id resampling_id
## 1: classif.rpart.tuned            cv
## 2: classif.rpart.tuned            cv
## 3: classif.rpart.tuned            cv
```

Next, we want to compare the predictive performances estimated on the outer resampling to the inner resampling. Significantly lower predictive performances on the outer resampling indicate that the models with the optimized hyperparameters overfit the data.

```
rr$score()
```

```
##                 task task_id         learner          learner_id
## 1: <TaskClassif[49]>    pima <AutoTuner[41]> classif.rpart.tuned
## 2: <TaskClassif[49]>    pima <AutoTuner[41]> classif.rpart.tuned
## 3: <TaskClassif[49]>    pima <AutoTuner[41]> classif.rpart.tuned
##            resampling resampling_id iteration              prediction
## 1: <ResamplingCV[19]>            cv         1 <PredictionClassif[20]>
## 2: <ResamplingCV[19]>            cv         2 <PredictionClassif[20]>
## 3: <ResamplingCV[19]>            cv         3 <PredictionClassif[20]>
##    classif.ce
## 1:     0.2617
## 2:     0.3047
## 3:     0.2344
```

The aggregated performance of all outer resampling iterations is essentially the unbiased performance of the model with optimal hyperparameter found by grid search.

```
rr$aggregate()
```

```
## classif.ce
##     0.2669
```

Note that nested resampling is computationally expensive. For this reason we use relatively small number of hyperparameter configurations and a low number of resampling iterations in this example. 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.

4.3.3 Final Model

We can use the `AutoTuner` to tune the hyperparameters of our learner and fit the final model on the full data set.

```
at$train(task)
```

```
## INFO  [16:04:28.172] [bbotk] Starting to optimize 1 parameter(s) with '<OptimizerGridSearc
## INFO  [16:04:28.175] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:28.364] [bbotk] Result of batch 1:
## INFO  [16:04:28.365] [bbotk]     cp classif.ce runtime_learners
## INFO  [16:04:28.365] [bbotk]  0.034     0.2461            0.059 ff9b1e17-f4db-4fc6-9213-a7
## INFO  [16:04:28.367] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:28.540] [bbotk] Result of batch 2:
## INFO  [16:04:28.542] [bbotk]     cp classif.ce runtime_learners
## INFO  [16:04:28.542] [bbotk]  0.089     0.2578            0.048 bb566908-a356-4515-a02f-6e
## INFO  [16:04:28.543] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:28.727] [bbotk] Result of batch 3:
## INFO  [16:04:28.729] [bbotk]     cp classif.ce runtime_learners
## INFO  [16:04:28.729] [bbotk]  0.078     0.2461            0.052 3d1130bf-2cf4-4190-9827-88
## INFO  [16:04:28.730] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:28.917] [bbotk] Result of batch 4:
## INFO  [16:04:28.919] [bbotk]     cp classif.ce runtime_learners
## INFO  [16:04:28.919] [bbotk]  0.023     0.2435            0.059 051bd2b5-9159-4055-94de-d8
## INFO  [16:04:28.920] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:04:29.108] [bbotk] Result of batch 5:
## INFO  [16:04:29.110] [bbotk]     cp classif.ce runtime_learners
## INFO  [16:04:29.110] [bbotk]  0.056     0.2461            0.049 ce3ce128-eb3c-4e24-b23c-2f
## INFO  [16:04:29.119] [bbotk] Finished optimizing after 5 evaluation(s)
## INFO  [16:04:29.120] [bbotk] Result:
## INFO  [16:04:29.121] [bbotk]     cp learner_param_vals  x_domain classif.ce
## INFO  [16:04:29.121] [bbotk]  0.023          <list[2]> <list[1]>     0.2435
```

The trained model can now be used to make predictions on new data. A common mistake is to report the performance estimated on the resampling sets on which the tuning was performed (`at$tuning_result$classif.ce`) as the model's performance. Instead, we report the performance estimated with nested resampling as the performance of the model.

4.4 Tuning with Hyperband

Besides the more traditional tuning methods, the ecosystem around mlr3 offers another procedure for hyperparameter optimization called Hyperband implemented in the mlr3hyperband package.

Hyperband is a budget-oriented procedure, weeding out suboptimal performing configurations early on during a partially sequential training process, increasing tuning efficiency as a consequence. For this, a combination of incremental resource allocation and early stopping is used: As optimization progresses, computational resources are increased for more promising configurations, while less promising ones are terminated early.

To give an introductory analogy, imagine two horse trainers are given eight untrained horses. Both trainers want to win the upcoming race, but they are only given 32 units of food. Given that each horse can be fed up to 8 units food ("maximum budget" per horse), there is not enough food for all the horses. It is critical to identify the most promising horses early, and give them enough food to improve. So, the trainers need to develop a strategy to split up the food in the best possible way. The first trainer is very optimistic and wants to explore the full capabilities of a horse, because he does not want to pass a judgment on a horse's performance unless it has been fully trained. So, he divides his budget by the maximum amount he can give to a horse (lets say eight, so $32/8 = 4$) and randomly picks four horses - his budget simply is not enough to fully train more. Those four horses are then trained to their full capabilities, while the rest is set free. This way, the trainer is confident about choosing the best out of the four trained horses, but he might have overlooked the horse with the highest potential since he only focused on half of them. The other trainer is more creative and develops a different strategy. He thinks, if a horse is not performing well at the beginning, it will also not improve after further training. Based on this assumption, he decides to give one unit of food to each horse and observes how they develop. After the initial food is consumed, he checks their performance and kicks the slowest half out of his training regime. Then, he increases the available food for the remaining, further trains them until the food is consumed again, only to kick out the worst half once more. He repeats this until the one remaining horse gets the rest of the food. This means only one horse is fully trained, but on the flip side, he was able to start training with all eight horses.

On race day, all the horses are put on the starting line. But which trainer will have the winning horse? The one, who tried to train a maximum amount of horses to their fullest? Or the other one, who made assumptions about the training progress of his horses? How the training phases may possibly look like is visualized in figure 4.1.

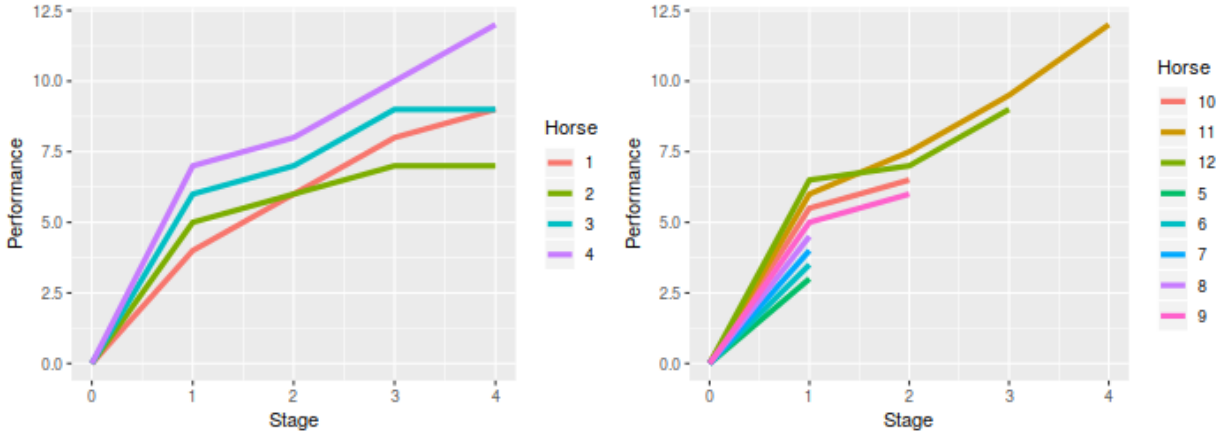


Figure 4.1: Visulization of how the training processes may look like. The left plot corresponds to the non-selective trainer, while the right one to the selective trainer.

Hyperband works very similar in some ways, but also different in others. It is not embodied by one of the trainers in our analogy, but more by the person, who would pay them. Hyperband consists of several brackets, each bracket corresponding to a trainer, and we do not care about

Table 4.2: Hyperband layout for $\eta = 2$ and $R = 8$, consisting of four brackets with n as the amount of active configurations.

stage	budget	n
1	1	8
2	2	4
3	4	2
4	8	1

stage	budget	n
1	2	6
2	4	3
3	8	1

stage	budget	n
1	4	4
2	8	2

stage	budget	n
1	8	4

horses but about hyperparameter configurations of a machine learning algorithm. The budget is not in terms of food, but in terms of a hyperparameter of the learner that scales in some way with the computational effort. An example is the number of epochs we train a neural network, or the number of iterations in boosting. Furthermore, there are not only two brackets (or trainers), but several, each placed at a unique spot between fully explorative of later training stages and extremely selective, equal to higher exploration of early training stages. The level of selection aggressiveness is handled by a user-defined parameter called η. So, $1/\eta$ is the fraction of remaining configurations after a bracket removes his worst performing ones, but η is also the factor by that the budget is increased for the next stage. Because there is a different maximum budget per configuration that makes sense in different scenarios, the user also has to set this as the R parameter. No further parameters are required for Hyperband – the full required budget across all brackets is indirectly given by

$$(\lfloor \log_\eta R \rfloor + 1)^2 * R$$

(Li et al. 2016). To give an idea how a full bracket layout might look like for a specific R and η, a quick overview is given in the following table.

Of course, early termination based on a performance criterion may be disadvantageous if it is done too aggressively in certain scenarios. A learner to jumping radically in its estimated performance during the training phase may get the best configurations canceled too early, simply because they do not improve quickly enough compared to others. In other words, it is often unclear beforehand if having an high amount of configurations n, that gets aggressively discarded early, is better than having a high budget B per configuration. The arising tradeoff, that has to be made, is called the "n versus B/n problem". To create a balance between selection based on early training performance versus exploration of training performances in later training stages, $\lfloor \log_\eta R \rfloor + 1$ brackets are constructed with an associated set of varying sized configurations. Thus, some brackets contain more configurations, with a small initial budget. In these, a lot are discarded after having been trained for only a short amount of time, corresponding to the selective trainer in our horse analogy. Others are constructed with fewer configurations, where discarding only takes place after a significant amount of budget was consumed. The last bracket usually never discards anything, but also starts with only very few configurations – this is equivalent to the trainer explorative of later stages. The former corresponds high n, while the latter high B/n. Even though different brackets are initialized with a different amount of configurations and different initial budget sizes, each bracket is assigned (approximately) the same budget $(\lfloor \log_\eta R \rfloor + 1) * R$.

The configurations at the start of each bracket are initialized by random, often uniform sampling. Note that currently all configurations are trained completely from the beginning, so no online updates of models from stage to stage is happening.

To identify the budget for evaluating Hyperband, the user has to specify explicitly which hyperparameter of the learner influences the budget by extending a single hyperparameter in the `ParamSet`

with an argument (`tags = "budget"`), like in the following snippet:

```
library("mlr3verse")

# Hyperparameter subset of XGBoost
search_space = ps(
  nrounds = p_int(lower = 1, upper = 16, tags = "budget"),
  booster = p_fct(levels = c("gbtree", "gblinear", "dart"))
)
```

Thanks to the broad ecosystem of the mlr3verse a learner does not require a natural budget parameter. A typical case of this would be decision trees. By using subsampling as preprocessing with mlr3pipelines, we can work around a lacking budget parameter.

```
set.seed(123)

# extend "classif.rpart" with "subsampling" as preprocessing step
ll = po("subsample") %>>% lrn("classif.rpart")

# extend hyperparameters of "classif.rpart" with subsampling fraction as budget
search_space = ps(
  classif.rpart.cp = p_dbl(lower = 0.001, upper = 0.1),
  classif.rpart.minsplit = p_int(lower = 1, upper = 10),
  subsample.frac = p_dbl(lower = 0.1, upper = 1, tags = "budget")
)
```

We can now plug the new learner with the extended hyperparameter set into a `TuningInstanceSingleCrit` the same way as usual. Naturally, Hyperband terminates once all of its brackets are evaluated, so a `Terminator` in the tuning instance acts as an upper bound and should be only set to a low value if one is unsure of how long Hyperband will take to finish under the given settings.

```
instance = TuningInstanceSingleCrit$new(
  task = tsk("iris"),
  learner = ll,
  resampling = rsmp("holdout"),
  measure = msr("classif.ce"),
  terminator = trm("none"), # hyperband terminates itself
  search_space = search_space
)
```

Now, we initialize a new instance of the `mlr3hyperband::mlr_tuners_hyperband` class and start tuning with it.

```
library("mlr3hyperband")
```

```
## Loading required package: mlr3tuning
```

```
## Loading required package: paradox
```

```
tuner = tnr("hyperband", eta = 3)

# reduce logging output
lgr::get_logger("bbotk")$set_threshold("warn")

tuner$optimize(instance)
```

```
##    classif.rpart.cp classif.rpart.minsplit subsample.frac learner_param_vals
## 1:          0.07348                      5         0.1111          <list[6]>
##     x_domain classif.ce
## 1: <list[3]>       0.02
```

To receive the results of each sampled configuration, we simply run the following snippet.

```
as.data.table(instance$archive)[, c(
  "subsample.frac",
  "classif.rpart.cp",
  "classif.rpart.minsplit",
  "classif.ce"
), with = FALSE]
```

```
##     subsample.frac classif.rpart.cp classif.rpart.minsplit classif.ce
##  1:         0.1111          0.02533                      3       0.04
##  2:         0.1111          0.07348                      5       0.02
##  3:         0.1111          0.08490                      3       0.02
##  4:         0.1111          0.05026                      6       0.02
##  5:         0.1111          0.03940                      4       0.02
##  6:         0.1111          0.02540                      7       0.42
##  7:         0.1111          0.01200                      4       0.14
##  8:         0.1111          0.03961                      4       0.02
##  9:         0.1111          0.05762                      6       0.02
## 10:         0.3333          0.07348                      5       0.06
## 11:         0.3333          0.08490                      3       0.04
## 12:         0.3333          0.05026                      6       0.06
## 13:         1.0000          0.08490                      3       0.04
## 14:         0.3333          0.08650                      6       0.02
## 15:         0.3333          0.07491                      9       0.06
## 16:         0.3333          0.06716                      6       0.04
## 17:         0.3333          0.06218                      9       0.08
## 18:         0.3333          0.03785                      4       0.06
## 19:         1.0000          0.08650                      6       0.04
## 20:         1.0000          0.02724                     10       0.04
## 21:         1.0000          0.05689                      3       0.04
## 22:         1.0000          0.09141                      4       0.04
##     subsample.frac classif.rpart.cp classif.rpart.minsplit classif.ce
```

You can access the best found configuration through the instance object.

```
instance$result
```

```
##    classif.rpart.cp classif.rpart.minsplit subsample.frac learner_param_vals
## 1:          0.07348                      5         0.1111          <list[6]>
##     x_domain classif.ce
## 1: <list[3]>       0.02
```

```
instance$result_learner_param_vals
```

```
## $subsample.frac
## [1] 0.1111
##
## $subsample.stratify
## [1] FALSE
##
## $subsample.replace
## [1] FALSE
##
## $classif.rpart.xval
## [1] 0
##
## $classif.rpart.cp
## [1] 0.07348
##
## $classif.rpart.minsplit
## [1] 5
```

```
instance$result_y
```

```
## classif.ce
##       0.02
```

If you are familiar with the original paper, you may have wondered how we just used Hyperband with a parameter ranging from `0.1` to `1.0` (Li et al. 2016). The answer is, with the help the internal rescaling of the budget parameter. mlr3hyperband automatically divides the budget parameters boundaries with its lower bound, ending up with a budget range starting again at `1`, like it is the case originally. If we want an overview of what bracket layout Hyperband created and how the rescaling in each bracket worked, we can print a compact table to see this information.

```
unique(as.data.table(instance$archive)[, .(bracket, bracket_stage, budget_scaled, budget_real, n_confi
```

```
##    bracket bracket_stage budget_scaled budget_real n_configs
## 1:       2             0         1.111      0.1111         9
## 2:       2             1         3.333      0.3333         3
## 3:       2             2        10.000      1.0000         1
## 4:       1             0         3.333      0.3333         5
## 5:       1             1        10.000      1.0000         1
## 6:       0             0        10.000      1.0000         3
```

In the traditional way, Hyperband uses uniform sampling to receive a configuration sample at the start of each bracket. But it is also possible to define a custom `Sampler` for each hyperparameter.

```
search_space = ps(
  nrounds = p_int(lower = 1, upper = 16, tags = "budget"),
  eta = p_dbl(lower = 0, upper = 1),
  booster = p_fct(levels = c("gbtree", "gblinear", "dart"))
)

instance = TuningInstanceSingleCrit$new(
  task = tsk("iris"),
  learner = lrn("classif.xgboost"),
  resampling = rsmp("holdout"),
  measure = msr("classif.ce"),
  terminator = trm("none"), # hyperband terminates itself
  search_space = search_space
)

# beta distribution with alpha = 2 and beta = 5
# categorical distribution with custom probabilities
sampler = SamplerJointIndep$new(list(
  Sampler1DRfun$new(search_space$params$eta, function(n) rbeta(n, 2, 5)),
  Sampler1DCateg$new(search_space$params$booster, prob = c(0.2, 0.3, 0.5))
))
```

Then, the defined sampler has to be given as an argument during instance creation. Afterwards, the usual tuning can proceed.

```
tuner = tnr("hyperband", eta = 2, sampler = sampler)
set.seed(123)
tuner$optimize(instance)
```

```
##    nrounds    eta booster learner_param_vals  x_domain classif.ce
## 1:       1 0.2415    dart          <list[5]> <list[3]>       0.04
```

```
instance$result
```

```
##    nrounds    eta booster learner_param_vals  x_domain classif.ce
## 1:       1 0.2415    dart          <list[5]> <list[3]>       0.04
```

Furthermore, we extended the original algorithm, to make it also possible to use mlr3hyperband for multi-objective optimization. To do this, simply specify more measures in the `TuningInstanceMultiCrit` and run the rest as usual.

```
instance = TuningInstanceMultiCrit$new(
  task = tsk("pima"),
  learner = lrn("classif.xgboost"),
  resampling = rsmp("holdout"),
  measures = msrs(c("classif.tpr", "classif.fpr")),
  terminator = trm("none"), # hyperband terminates itself
  search_space = search_space
```

```
)

tuner = tnr("hyperband", eta = 4)
tuner$optimize(instance)
```

```
##     nrounds     eta  booster learner_param_vals  x_domain classif.tpr
##  1:       1 0.34093 gblinear          <list[5]> <list[3]>      0.0000
##  2:       1 0.50700 gblinear          <list[5]> <list[3]>      0.0000
##  3:       1 0.77369 gblinear          <list[5]> <list[3]>      0.0000
##  4:       1 0.55967 gblinear          <list[5]> <list[3]>      0.0000
##  5:       1 0.46843 gblinear          <list[5]> <list[3]>      0.0000
##  6:       1 0.27922 gblinear          <list[5]> <list[3]>      0.0000
##  7:       1 0.79274 gblinear          <list[5]> <list[3]>      0.0000
##  8:       1 0.58557 gblinear          <list[5]> <list[3]>      0.0000
##  9:       1 0.39476 gblinear          <list[5]> <list[3]>      0.0000
## 10:       1 0.58145 gblinear          <list[5]> <list[3]>      0.0000
## 11:      16 0.01919   gbtree          <list[5]> <list[3]>      0.6374
## 12:       4 0.55531     dart          <list[5]> <list[3]>      0.5824
## 13:       4 0.65920   gbtree          <list[5]> <list[3]>      0.6044
## 14:      16 0.13587     dart          <list[5]> <list[3]>      0.6264
##     classif.fpr
##  1:      0.0000
##  2:      0.0000
##  3:      0.0000
##  4:      0.0000
##  5:      0.0000
##  6:      0.0000
##  7:      0.0000
##  8:      0.0000
##  9:      0.0000
## 10:      0.0000
## 11:      0.2364
## 12:      0.1576
## 13:      0.1758
## 14:      0.1818
```

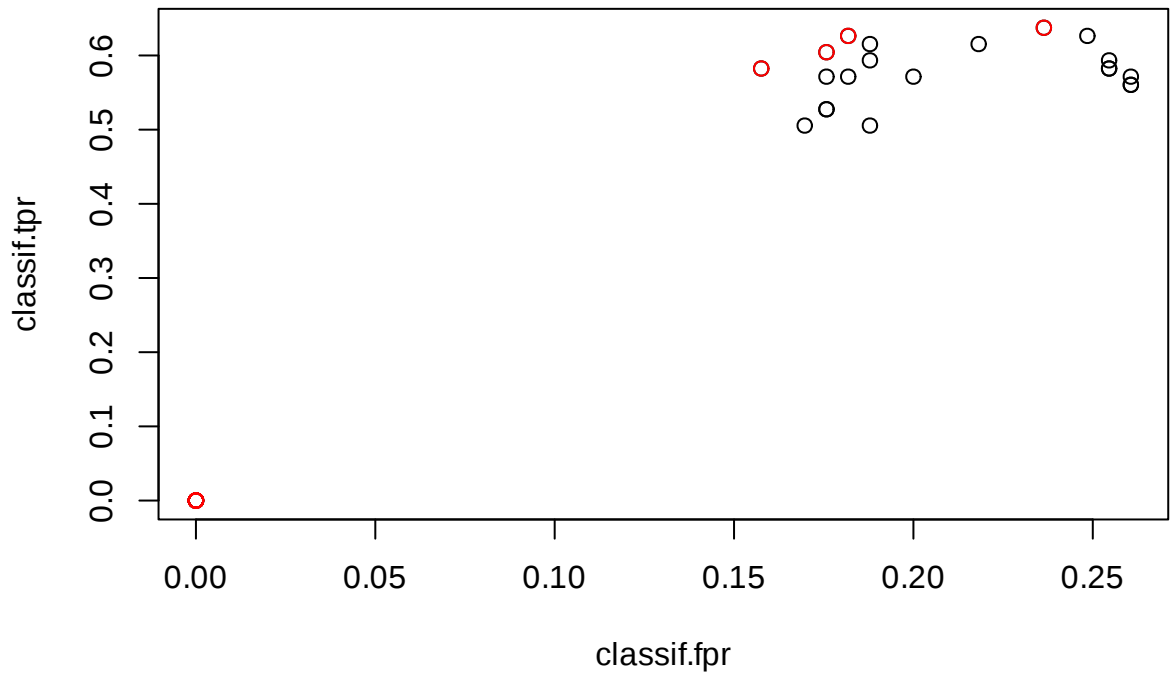
Now the result is not a single best configuration but an estimated Pareto front. All red points are not dominated by another parameter configuration regarding their *fpr* and *tpr* performance measures.

```
instance$result
```

```
##     nrounds     eta  booster learner_param_vals  x_domain classif.tpr
##  1:       1 0.34093 gblinear          <list[5]> <list[3]>      0.0000
##  2:       1 0.50700 gblinear          <list[5]> <list[3]>      0.0000
##  3:       1 0.77369 gblinear          <list[5]> <list[3]>      0.0000
##  4:       1 0.55967 gblinear          <list[5]> <list[3]>      0.0000
##  5:       1 0.46843 gblinear          <list[5]> <list[3]>      0.0000
```

```
##  6:       1 0.27922 gblinear          <list[5]> <list[3]>      0.0000
##  7:       1 0.79274 gblinear          <list[5]> <list[3]>      0.0000
##  8:       1 0.58557 gblinear          <list[5]> <list[3]>      0.0000
##  9:       1 0.39476 gblinear          <list[5]> <list[3]>      0.0000
## 10:       1 0.58145 gblinear          <list[5]> <list[3]>      0.0000
## 11:      16 0.01919   gbtree          <list[5]> <list[3]>      0.6374
## 12:       4 0.55531     dart          <list[5]> <list[3]>      0.5824
## 13:       4 0.65920   gbtree          <list[5]> <list[3]>      0.6044
## 14:      16 0.13587     dart          <list[5]> <list[3]>      0.6264
##     classif.fpr
##  1:      0.0000
##  2:      0.0000
##  3:      0.0000
##  4:      0.0000
##  5:      0.0000
##  6:      0.0000
##  7:      0.0000
##  8:      0.0000
##  9:      0.0000
## 10:      0.0000
## 11:      0.2364
## 12:      0.1576
## 13:      0.1758
## 14:      0.1818
```

```
plot(classif.tpr ~ classif.fpr, instance$archive$data)
points(classif.tpr ~ classif.fpr, instance$result, col = "red")
```



4.5 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 (Guyon and Elisseeff 2003; Chandrashekar and Sahin 2014)?

- **Filtering**: An external algorithm computes a rank of the features (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. All filters are connected via package mlr3filters.
- **Wrapper Methods**: Here, no ranking of features is done. Instead, an optimization algorithm selects a subset of the features, evaluates the set by calculating the resampled predictive performance, and then proposes a new set of features (or terminates). A simple example is the sequential forward selection. This method is usually computationally very 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 final set of selected features is again fitted (with optional hyperparameters selected by tuning). Wrapper methods are implemented in the mlr3fselect package.

- **Embedded Methods**: Many learners internally select a subset of the features which they find helpful for prediction. These subsets can usually be queried, as the following example demonstrates:

```
library("mlr3verse")

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

# ensure that the learner selects features
stopifnot("selected_features" %in% learner$properties)

# fit a simple classification tree
learner = learner$train(task)

# extract all features used in the classification tree:
learner$selected_features()
```

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

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

4.5.1 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.

4.5.2 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. Similar to other instances in `mlr3`, these are registered in a dictionary (`mlr_filters`) with an associated shortcut function `flt()`. Each object of class `Filter` has a `.$calculate()` method which computes the filter values and ranks them in a descending order.

```
filter = flt("jmim")

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

as.data.table(filter)
```

```
##         feature  score
## 1:  Petal.Width 1.0000
## 2: Sepal.Length 0.6667
## 3: Petal.Length 0.3333
## 4:  Sepal.Width 0.0000
```

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

```
filter_cor = flt("correlation")
filter_cor$param_set
```

```
## <ParamSet>
##        id    class lower upper nlevels    default value
## 1:    use ParamFct    NA    NA       5 everything
## 2: method ParamFct    NA    NA       3    pearson
```

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

```
## <ParamSet>
##        id    class lower upper nlevels    default    value
## 1:    use ParamFct    NA    NA       5 everything
## 2: method ParamFct    NA    NA       3    pearson spearman
```

4.5.3 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 comes with multiple integrated methods, c.f. the help page of ranger::ranger(). To use method "impurity", you need to set the filter method during construction.

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

Now you can use the FilterImportance filter class for algorithm-embedded methods:

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

```
##         feature  score
## 1: Petal.Length 45.295
## 2:  Petal.Width 43.253
## 3: Sepal.Length  8.587
```

4.5.4 Wrapper Methods

Wrapper feature selection is supported via the mlr3fselect extension package. At the heart of mlr3fselect are the R6 classes:

- `FSelectInstanceSingleCrit`, `FSelectInstanceMultiCrit`: These two classes describe the feature selection problem and store the results.
- `FSelector`: This class is the base class for implementations of feature selection algorithms.

4.5.5 The `FSelectInstance` Classes

The following sub-section examines the feature selection on the `Pima` data set which is used to predict whether or not a patient has diabetes.

```
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.

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

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

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

Finally, one has to choose the available budget for the feature selection. 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 feature selection 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 `FSelectInstanceSingleCrit`:

```
evals20 = trm("evals", n_evals = 20)

instance = FSelectInstanceSingleCrit$new(
  task = task,
  learner = learner,
  resampling = hout,
  measure = measure,
  terminator = evals20
)
instance
```

```
## <FSelectInstanceSingleCrit>
## * State:  Not optimized
## * Objective: <ObjectiveFSelect:classif.rpart_on_pima>
## * Search Space:
## <ParamSet>
##          id    class lower upper nlevels        default value
## 1:      age ParamLgl    NA    NA       2 <NoDefault[3]>
## 2:  glucose ParamLgl    NA    NA       2 <NoDefault[3]>
## 3:  insulin ParamLgl    NA    NA       2 <NoDefault[3]>
## 4:     mass ParamLgl    NA    NA       2 <NoDefault[3]>
## 5: pedigree ParamLgl    NA    NA       2 <NoDefault[3]>
## 6: pregnant ParamLgl    NA    NA       2 <NoDefault[3]>
## 7: pressure ParamLgl    NA    NA       2 <NoDefault[3]>
## 8:  triceps ParamLgl    NA    NA       2 <NoDefault[3]>
## * Terminator: <TerminatorEvals>
## * Terminated: FALSE
## * Archive:
## <ArchiveFSelect>
## Null data.table (0 rows and 0 cols)
```

To start the feature selection, we still need to select an algorithm which are defined via the `FSelector` class

4.5.6 The `FSelector` Class

The following algorithms are currently implemented in mlr3fselect:

- Random Search (`FSelectorRandomSearch`)
- Exhaustive Search (`FSelectorExhaustiveSearch`)
- Sequential Search (`FSelectorSequential`)
- Recursive Feature Elimination (`FSelectorRFE`)
- Design Points (`FSelectorDesignPoints`)

In this example, we will use a simple random search and retrieve it from the dictionary `mlr_fselectors` with the `fs()` function:

```
fselector = fs("random_search")
```

4.5.7 Triggering the Tuning

To start the feature selection, we simply pass the FSelectInstanceSingleCrit to the `$optimize()` method of the initialized FSelector. The algorithm proceeds as follows

1. The FSelector proposes at least one feature subset and may propose multiple subsets to improve parallelization, which can be controlled via the setting `batch_size`).
2. For each feature subset, the given Learner is fitted on the Task using the provided Resampling. All evaluations are stored in the archive of the FSelectInstanceSingleCrit.
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 feature subset with the best observed performance.
5. Store the best feature subset as the result in the instance object. The best feature subset (`$result_feature_set`) and the corresponding measured performance (`$result_y`) can be accessed from the instance.

```
# reduce logging output
lgr::get_logger("bbotk")$set_threshold("warn")

fselector$optimize(instance)
```

```
##     age glucose insulin mass pedigree pregnant pressure triceps
## 1: TRUE    TRUE    TRUE TRUE     TRUE     TRUE    FALSE    TRUE
##                                         features classif.ce
## 1: age,glucose,insulin,mass,pedigree,pregnant,...     0.2578
```

```
instance$result_feature_set
```

```
## [1] "age"      "glucose"  "insulin"  "mass"     "pedigree" "pregnant" "triceps"
```

```
instance$result_y
```

```
## classif.ce
##     0.2578
```

One can investigate all resamplings which were undertaken, as they are stored in the archive of the FSelectInstanceSingleCrit and can be accessed by using `as.data.table()`:

```
as.data.table(instance$archive)
```

```
##       age glucose insulin  mass pedigree pregnant pressure triceps classif.ce
##  1:  TRUE    TRUE    TRUE FALSE    FALSE    FALSE     TRUE   FALSE     0.2812
##  2:  TRUE    TRUE    TRUE  TRUE     TRUE     TRUE    FALSE    TRUE     0.2578
##  3:  TRUE    TRUE    TRUE FALSE     TRUE    FALSE    FALSE    TRUE     0.3008
##  4:  TRUE   FALSE    TRUE FALSE     TRUE     TRUE     TRUE   FALSE     0.3203
##  5: FALSE   FALSE    TRUE FALSE    FALSE    FALSE    FALSE   FALSE     0.4883
##  6:  TRUE   FALSE   FALSE  TRUE    FALSE    FALSE    FALSE   FALSE     0.3711
```

```
##  7:  TRUE   FALSE    TRUE  TRUE     TRUE     TRUE     TRUE   FALSE     0.2891
##  8:  TRUE   FALSE    TRUE FALSE     TRUE     TRUE     TRUE   FALSE     0.3203
##  9:  TRUE    TRUE    TRUE  TRUE     TRUE     TRUE     TRUE    TRUE     0.2617
## 10:  TRUE   FALSE    TRUE  TRUE     TRUE     TRUE     TRUE    TRUE     0.2891
## 11:  TRUE    TRUE   FALSE FALSE    FALSE    FALSE     TRUE   FALSE     0.2812
## 12:  TRUE   FALSE   FALSE FALSE     TRUE     TRUE     TRUE    TRUE     0.3086
## 13: FALSE   FALSE   FALSE FALSE    FALSE     TRUE    FALSE   FALSE     0.3281
## 14:  TRUE   FALSE    TRUE FALSE    FALSE    FALSE     TRUE    TRUE     0.3125
## 15: FALSE   FALSE   FALSE  TRUE    FALSE    FALSE    FALSE   FALSE     0.4492
## 16:  TRUE   FALSE   FALSE FALSE    FALSE     TRUE    FALSE   FALSE     0.3359
## 17: FALSE   FALSE    TRUE FALSE     TRUE    FALSE    FALSE   FALSE     0.3555
## 18:  TRUE    TRUE    TRUE  TRUE    FALSE    FALSE    FALSE   FALSE     0.2734
## 19: FALSE   FALSE   FALSE  TRUE    FALSE    FALSE    FALSE    TRUE     0.4062
## 20:  TRUE   FALSE    TRUE  TRUE     TRUE     TRUE     TRUE    TRUE     0.2891
##     runtime_learners           timestamp batch_nr      resample_result
##  1:            0.083 2021-11-24 16:04:59        1 <ResampleResult[22]>
##  2:            0.095 2021-11-24 16:04:59        2 <ResampleResult[22]>
##  3:            0.081 2021-11-24 16:04:59        3 <ResampleResult[22]>
##  4:            0.096 2021-11-24 16:05:00        4 <ResampleResult[22]>
##  5:            0.076 2021-11-24 16:05:00        5 <ResampleResult[22]>
##  6:            0.074 2021-11-24 16:05:00        6 <ResampleResult[22]>
##  7:            0.101 2021-11-24 16:05:00        7 <ResampleResult[22]>
##  8:            0.164 2021-11-24 16:05:01        8 <ResampleResult[22]>
##  9:            0.084 2021-11-24 16:05:01        9 <ResampleResult[22]>
## 10:            0.090 2021-11-24 16:05:01       10 <ResampleResult[22]>
## 11:            0.090 2021-11-24 16:05:01       11 <ResampleResult[22]>
## 12:            0.103 2021-11-24 16:05:02       12 <ResampleResult[22]>
## 13:            0.084 2021-11-24 16:05:02       13 <ResampleResult[22]>
## 14:            0.083 2021-11-24 16:05:02       14 <ResampleResult[22]>
## 15:            0.105 2021-11-24 16:05:02       15 <ResampleResult[22]>
## 16:            0.085 2021-11-24 16:05:03       16 <ResampleResult[22]>
## 17:            0.072 2021-11-24 16:05:03       17 <ResampleResult[22]>
## 18:            0.065 2021-11-24 16:05:04       18 <ResampleResult[22]>
## 19:            0.083 2021-11-24 16:05:04       19 <ResampleResult[22]>
## 20:            0.099 2021-11-24 16:05:04       20 <ResampleResult[22]>
```

The associated resampling iterations can be accessed in the `BenchmarkResult`:

```
instance$archive$benchmark_result$data
```

```
## Warning: '.__BenchmarkResult__data' is deprecated.
## Use 'as.data.table(benchmark_result)' instead.
## See help("Deprecated")
```

```
## <ResultData>
##   Public:
##     as_data_table: function (view = NULL, reassemble_learners = TRUE, convert_predictions
##     clone: function (deep = FALSE)
```

```
##     combine: function (rdata)
##     data: list
##     discard: function (backends = FALSE, models = FALSE)
##     initialize: function (data = NULL, store_backends = TRUE)
##     iterations: function (view = NULL)
##     learners: function (view = NULL, states = TRUE, reassemble = TRUE)
##     logs: function (view = NULL, condition)
##     prediction: function (view = NULL, predict_sets = "test")
##     predictions: function (view = NULL, predict_sets = "test")
##     resamplings: function (view = NULL)
##     sweep: function ()
##     task_type: active binding
##     tasks: function (view = NULL)
##     uhashes: function (view = NULL)
##   Private:
##     deep_clone: function (name, value)
##     get_view_index: function (view)
```

The `uhash` column links the resampling iterations to the evaluated feature subsets stored in `instance$archive$data()`. This allows e.g. to score the included `ResampleResult`s on a different measure.

Now the optimized feature subset can be used to subset the task and fit the model on all observations.

```
task$select(instance$result_feature_set)
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 feature selection 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.

4.5.8 Automating the Feature Selection

The `AutoFSelector` wraps a learner and augments it with an automatic feature selection for a given task. Because the `AutoFSelector` 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 starts a feature selection on the given task using an inner resampling (holdout). We create a terminator which allows 10 evaluations, and uses a simple random search as feature selection algorithm:

```
learner = lrn("classif.rpart")
terminator = trm("evals", n_evals = 10)
fselector = fs("random_search")

at = AutoFSelector$new(
  learner = learner,
```

```
  resampling = rsmp("holdout"),
  measure = msr("classif.ce"),
  terminator = terminator,
  fselector = fselector
)
at
```

```
## <AutoFSelector:classif.rpart.fselector>
## * Model: -
## * Parameters: xval=0
## * Packages: mlr3, 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 optimized feature subset to the complete feature set. This way, the `AutoFSelector` will do its resampling for feature selection 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 feature selection or fitting of the respective learner. This is called nested resampling.

To compare the optimized feature subset with the complete feature set, we can use `benchmark()`:

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

bmr = benchmark(grid, store_models = TRUE)
bmr$aggregate(msrs(c("classif.ce", "time_train")))
```

```
##    nr      resample_result task_id              learner_id resampling_id iters
## 1:  1 <ResampleResult[22]>    pima classif.rpart.fselector            cv     3
## 2:  2 <ResampleResult[22]>    pima           classif.rpart            cv     3
##    classif.ce time_train
## 1:     0.2682          0
## 2:     0.2669          0
```

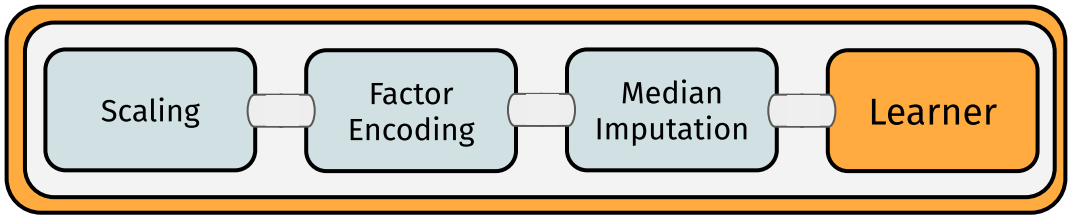
Note that we do not expect any significant differences since we only evaluated a small fraction of the possible feature subsets.

5 Pipelines

mlr3pipelines (Binder et al. 2021) is a dataflow programming toolkit. This chapter focuses on the applicant's side of the package. A more in-depth and technically oriented guide can be found in the In-depth look into mlr3pipelines chapter.

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.

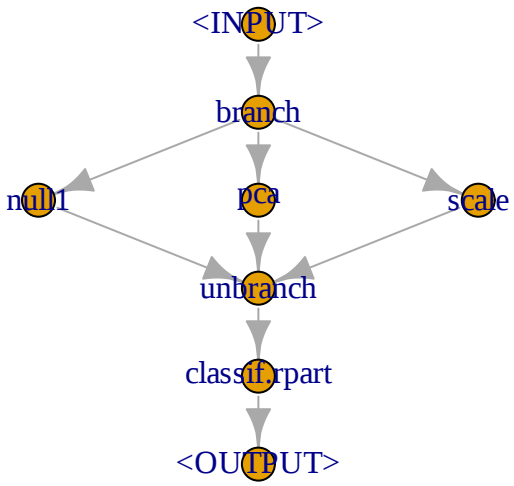
Using methods from mlr3tuning, it is even possible to simultaneously optimize parameters of multiple processing units.

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:



5.1 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:

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

```
##                       key                          packages
##  1:                boxcox      mlr3pipelines,bestNormalize
##  2:                branch                    mlr3pipelines
##  3:                 chunk                    mlr3pipelines
##  4:        classbalancing                    mlr3pipelines
##  5:            classifavg              mlr3pipelines,stats
##  6:          classweights                    mlr3pipelines
##  7:              colapply                    mlr3pipelines
##  8:       collapsefactors                    mlr3pipelines
```

```
##  9:              colroles                    mlr3pipelines
## 10:                  copy                    mlr3pipelines
## 11:          datefeatures                    mlr3pipelines
## 12:                encode              mlr3pipelines,stats
## 13:          encodeimpact                    mlr3pipelines
## 14:            encodelmer        mlr3pipelines,lme4,nloptr
## 15:          featureunion                    mlr3pipelines
## 16:                filter                    mlr3pipelines
## 17:            fixfactors                    mlr3pipelines
## 18:               histbin           mlr3pipelines,graphics
## 19:                   ica            mlr3pipelines,fastICA
## 20:        imputeconstant                    mlr3pipelines
## 21:            imputehist           mlr3pipelines,graphics
## 22:         imputelearner                    mlr3pipelines
## 23:            imputemean                    mlr3pipelines
## 24:          imputemedian              mlr3pipelines,stats
## 25:            imputemode                    mlr3pipelines
## 26:             imputeoor                    mlr3pipelines
## 27:          imputesample                    mlr3pipelines
## 28:             kernelpca            mlr3pipelines,kernlab
## 29:               learner                    mlr3pipelines
## 30:            learner_cv                    mlr3pipelines
## 31:               missind                    mlr3pipelines
## 32:           modelmatrix              mlr3pipelines,stats
## 33:     multiplicityexply                    mlr3pipelines
## 34:     multiplicityimply                    mlr3pipelines
## 35:                mutate                    mlr3pipelines
## 36:                   nmf           mlr3pipelines,MASS,NMF
## 37:                   nop                    mlr3pipelines
## 38:              ovrsplit                    mlr3pipelines
## 39:              ovrunite                    mlr3pipelines
## 40:                   pca                    mlr3pipelines
## 41:                 proxy                    mlr3pipelines
## 42:           quantilebin              mlr3pipelines,stats
## 43:      randomprojection                    mlr3pipelines
## 44:        randomresponse                    mlr3pipelines
## 45:               regravg                    mlr3pipelines
## 46:       removeconstants                    mlr3pipelines
## 47:         renamecolumns                    mlr3pipelines
## 48:             replicate                    mlr3pipelines
## 49:                 scale                    mlr3pipelines
## 50:           scalemaxabs                    mlr3pipelines
## 51:            scalerange                    mlr3pipelines
## 52:                select                    mlr3pipelines
## 53:                 smote        mlr3pipelines,smotefamily
## 54:           spatialsign                    mlr3pipelines
## 55:             subsample                    mlr3pipelines
## 56:          targetinvert                    mlr3pipelines
## 57:          targetmutate                    mlr3pipelines
## 58: targettrafoscalerange                    mlr3pipelines
```

```
## 59:        textvectorizer mlr3pipelines,quanteda,stopwords
## 60:             threshold                    mlr3pipelines
## 61:         tunethreshold              mlr3pipelines,bbotk
## 62:              unbranch                    mlr3pipelines
## 63:                vtreat             mlr3pipelines,vtreat
## 64:            yeojohnson      mlr3pipelines,bestNormalize
##                       key                          packages
##                                  tags
##  1:                   data transform
##  2:                              meta
##  3:                              meta
##  4:   imbalanced data,data transform
##  5:                           ensemble
##  6:   imbalanced data,data transform
##  7:                   data transform
##  8:                   data transform
##  9:                   data transform
## 10:                              meta
## 11:                   data transform
## 12:            encode,data transform
## 13:            encode,data transform
## 14:            encode,data transform
## 15:                           ensemble
## 16: feature selection,data transform
## 17:         robustify,data transform
## 18:                   data transform
## 19:                   data transform
## 20:                         missings
## 21:                         missings
## 22:                         missings
## 23:                         missings
## 24:                         missings
## 25:                         missings
## 26:                         missings
## 27:                         missings
## 28:                   data transform
## 29:                          learner
## 30:  learner,ensemble,data transform
## 31:          missings,data transform
## 32:                   data transform
## 33:                     multiplicity
## 34:                     multiplicity
## 35:                   data transform
## 36:                   data transform
## 37:                              meta
## 38:    target transform,multiplicity
## 39:             multiplicity,ensemble
## 40:                   data transform
## 41:                              meta
## 42:                   data transform
```

```
## 43:                   data transform
## 44:                         abstract
## 45:                         ensemble
## 46:         robustify,data transform
## 47:                   data transform
## 48:                     multiplicity
## 49:                   data transform
## 50:                   data transform
## 51:                   data transform
## 52: feature selection,data transform
## 53:   imbalanced data,data transform
## 54:                   data transform
## 55:                   data transform
## 56:                         abstract
## 57:                target transform
## 58:                target transform
## 59:                  data transform
## 60:                target transform
## 61:                target transform
## 62:                            meta
## 63:   encode,missings,data transform
## 64:                   data transform
##                                tags
##                                          feature_types input.num output.num
##  1:                                    numeric,integer         1          1
##  2:                                                 NA         1         NA
##  3:                                                 NA         1         NA
##  4: logical,integer,numeric,character,factor,ordered,...         1          1
##  5:                                                 NA        NA          1
##  6: logical,integer,numeric,character,factor,ordered,...         1          1
##  7: logical,integer,numeric,character,factor,ordered,...         1          1
##  8:                                      factor,ordered         1          1
##  9: logical,integer,numeric,character,factor,ordered,...         1          1
## 10:                                                 NA         1         NA
## 11:                                             POSIXct         1          1
## 12:                                      factor,ordered         1          1
## 13:                                      factor,ordered         1          1
## 14:                                      factor,ordered         1          1
## 15:                                                 NA        NA          1
## 16: logical,integer,numeric,character,factor,ordered,...         1          1
## 17:                                      factor,ordered         1          1
## 18:                                    numeric,integer         1          1
## 19:                                    numeric,integer         1          1
## 20: logical,integer,numeric,character,factor,ordered,...         1          1
## 21:                                    integer,numeric         1          1
## 22:                          logical,factor,ordered         1          1
## 23:                                    numeric,integer         1          1
## 24:                                    numeric,integer         1          1
## 25:          factor,integer,logical,numeric,ordered         1          1
## 26:       character,factor,integer,numeric,ordered         1          1
```

```
## 27:               factor,integer,logical,numeric,ordered         1          1
## 28:                                     numeric,integer         1          1
## 29:                                                  NA         1          1
## 30: logical,integer,numeric,character,factor,ordered,...         1          1
## 31: logical,integer,numeric,character,factor,ordered,...         1          1
## 32: logical,integer,numeric,character,factor,ordered,...         1          1
## 33:                                                  NA         1         NA
## 34:                                                  NA        NA          1
## 35: logical,integer,numeric,character,factor,ordered,...         1          1
## 36:                                     numeric,integer         1          1
## 37:                                                  NA         1          1
## 38:                                                  NA         1          1
## 39:                                                  NA         1          1
## 40:                                     numeric,integer         1          1
## 41:                                                  NA        NA          1
## 42:                                     numeric,integer         1          1
## 43:                                     numeric,integer         1          1
## 44:                                                  NA         1          1
## 45:                                                  NA        NA          1
## 46: logical,integer,numeric,character,factor,ordered,...         1          1
## 47: logical,integer,numeric,character,factor,ordered,...         1          1
## 48:                                                  NA         1          1
## 49:                                     numeric,integer         1          1
## 50:                                     numeric,integer         1          1
## 51:                                     numeric,integer         1          1
## 52: logical,integer,numeric,character,factor,ordered,...         1          1
## 53: logical,integer,numeric,character,factor,ordered,...         1          1
## 54:                                     numeric,integer         1          1
## 55: logical,integer,numeric,character,factor,ordered,...         1          1
## 56:                                                  NA         2          1
## 57:                                                  NA         1          2
## 58:                                                  NA         1          2
## 59:                                           character         1          1
## 60:                                                  NA         1          1
## 61:                                                  NA         1          1
## 62:                                                  NA        NA          1
## 63: logical,integer,numeric,character,factor,ordered,...         1          1
## 64:                                     numeric,integer         1          1
##                                       feature_types input.num output.num
##     input.type.train  input.type.predict output.type.train output.type.predict
##  1:             Task                Task              Task                Task
##  2:                *                   *                 *                   *
##  3:             Task                Task              Task                Task
##  4:      TaskClassif         TaskClassif       TaskClassif         TaskClassif
##  5:             NULL   PredictionClassif              NULL   PredictionClassif
##  6:      TaskClassif         TaskClassif       TaskClassif         TaskClassif
##  7:             Task                Task              Task                Task
##  8:             Task                Task              Task                Task
##  9:             Task                Task              Task                Task
## 10:                *                   *                 *                   *
```

```
## 11:            Task               Task            Task               Task
## 12:            Task               Task            Task               Task
## 13:            Task               Task            Task               Task
## 14:            Task               Task            Task               Task
## 15:            Task               Task            Task               Task
## 16:            Task               Task            Task               Task
## 17:            Task               Task            Task               Task
## 18:            Task               Task            Task               Task
## 19:            Task               Task            Task               Task
## 20:            Task               Task            Task               Task
## 21:            Task               Task            Task               Task
## 22:            Task               Task            Task               Task
## 23:            Task               Task            Task               Task
## 24:            Task               Task            Task               Task
## 25:            Task               Task            Task               Task
## 26:            Task               Task            Task               Task
## 27:            Task               Task            Task               Task
## 28:            Task               Task            Task               Task
## 29:      TaskClassif        TaskClassif            NULL  PredictionClassif
## 30:      TaskClassif        TaskClassif      TaskClassif        TaskClassif
## 31:            Task               Task            Task               Task
## 32:            Task               Task            Task               Task
## 33:             [*]                [*]               *                  *
## 34:               *                  *             [*]                [*]
## 35:            Task               Task            Task               Task
## 36:            Task               Task            Task               Task
## 37:               *                  *               *                  *
## 38:      TaskClassif        TaskClassif    [TaskClassif]      [TaskClassif]
## 39:           [NULL] [PredictionClassif]            NULL  PredictionClassif
## 40:            Task               Task            Task               Task
## 41:               *                  *               *                  *
## 42:            Task               Task            Task               Task
## 43:            Task               Task            Task               Task
## 44:            NULL         Prediction            NULL         Prediction
## 45:            NULL     PredictionRegr            NULL     PredictionRegr
## 46:            Task               Task            Task               Task
## 47:            Task               Task            Task               Task
## 48:               *                  *             [*]                [*]
## 49:            Task               Task            Task               Task
## 50:            Task               Task            Task               Task
## 51:            Task               Task            Task               Task
## 52:            Task               Task            Task               Task
## 53:            Task               Task            Task               Task
## 54:            Task               Task            Task               Task
## 55:            Task               Task            Task               Task
## 56:       NULL,NULL function,Prediction            NULL         Prediction
## 57:            Task               Task       NULL,Task      function,Task
## 58:        TaskRegr           TaskRegr   NULL,TaskRegr  function,TaskRegr
## 59:            Task               Task            Task               Task
## 60:            NULL  PredictionClassif            NULL  PredictionClassif
```

```
## 61:              Task                Task              NULL          Prediction
## 62:                 *                   *                 *                   *
## 63:              Task                Task              Task                Task
## 64:              Task                Task              Task                Task
##      input.type.train  input.type.predict output.type.train output.type.predict
```

Single POs can be created using `po(<name>)`:

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

or using **syntactic sugar**

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

Some POs require additional arguments for construction:

```
learner = po("learner")

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

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

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

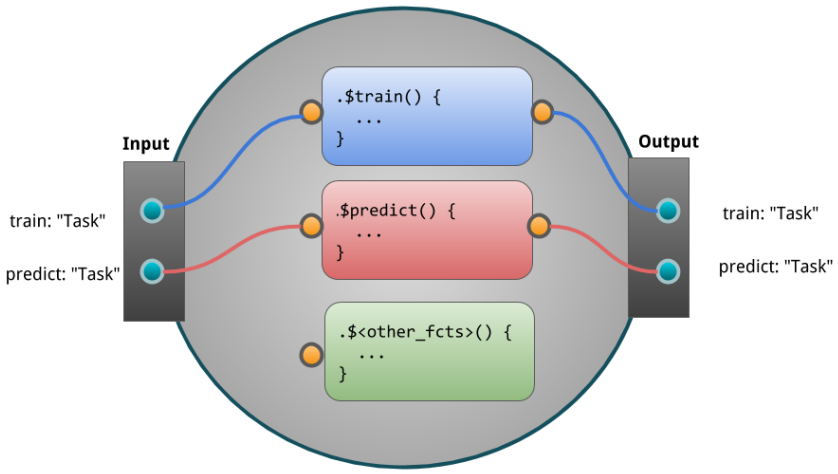
Hyperparameters of POs can be set through the `param_vals` argument. Here we set the fraction of features for a filter:

```
filter = po("filter",
  filter = mlr3filters::flt("variance"),
  param_vals = list(filter.frac = 0.5))
```

or in short notation:

```
po("filter", mlr3filters::flt("variance"), filter.frac = 0.5)
```

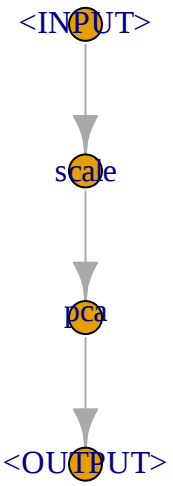
The figure below shows an exemplary `PipeOp`. It takes an input, transforms it during `.$train` and `.$predict` and returns data:



5.2 The Pipeline Operator: %>>%

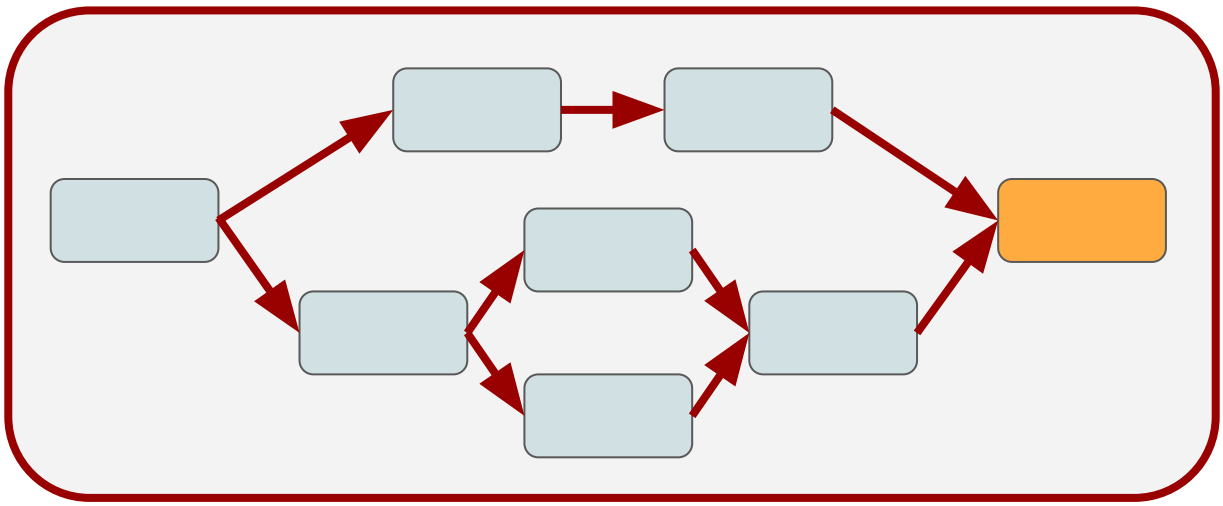
It is possible to create intricate `Graphs` with edges going all over the place (as long as no loops are introduced). Irrespective, there is usually a clear direction of flow between "layers" in the `Graph`. It is therefore convenient to build up a `Graph` from layers. This can be done using the **`%>>%`** ("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.

```
library("magrittr")

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



5.3 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 = po("mutate")

filter = po("filter",
  filter = mlr3filters::flt("variance"),
  param_vals = list(filter.frac = 0.5))
```

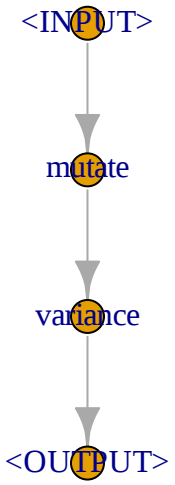
```
graph = Graph$new()$
  add_pipeop(mutate)$
  add_pipeop(filter)$
  add_edge("mutate", "variance") # add connection mutate -> filter
```

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:

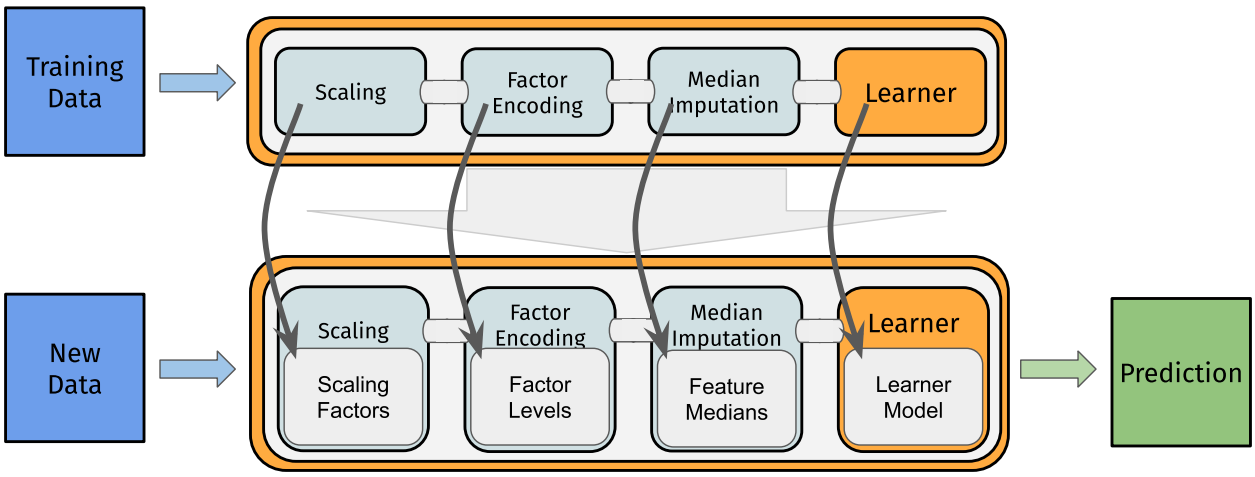
```
graph$add_pipeop(po("pca"))
```

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

5.4 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 = po("mutate")
filter = po("filter",
  filter = mlr3filters::flt("variance"),
  param_vals = list(filter.frac = 0.5))

graph = mutate %>>%
  filter %>>%
  po("learner",
    learner = lrn("classif.rpart"))
```

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

```
task = tsk("iris")
graph$train(task)
```

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

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

```
## $classif.rpart.output
## <PredictionClassif> for 150 observations:
##     row_ids     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 = as_learner(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
```

5.4.1 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
```

5.4.2 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 = ps(
  classif.rpart.cp = p_dbl(lower = 0, upper = 0.05),
  variance.filter.frac = p_dbl(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 = TuningInstanceSingleCrit$new(
  task = task,
  learner = glrn,
  resampling = rsmp("holdout"),
  measure = msr("classif.ce"),
  search_space = ps,
  terminator = trm("evals", n_evals = 20)
)
```

```
tuner = tnr("random_search")
tuner$optimize(instance)
```

The tuning result can be found in the respective `result` slots.

```
instance$result_learner_param_vals
instance$result_y
```

5.5 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.

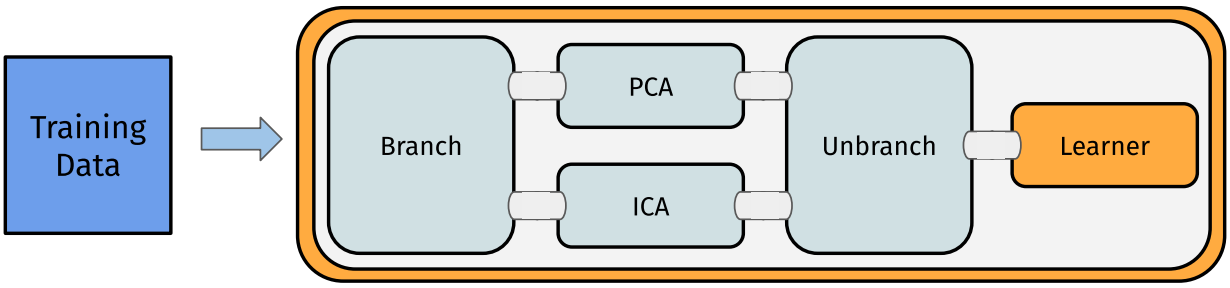
Possible types are:

- 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`

5.5.1 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

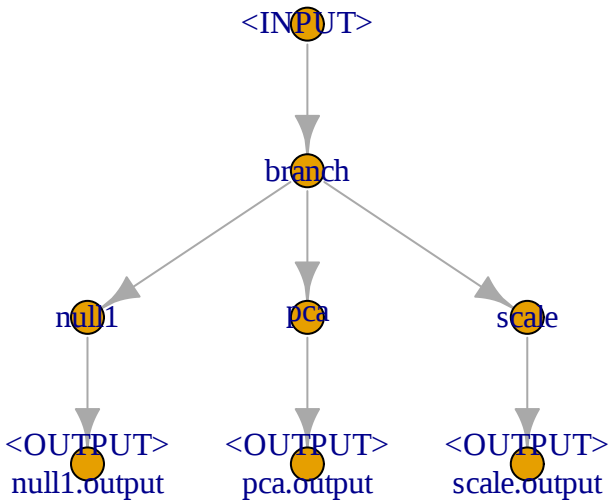
It is important to "unbranch" again after "branching", so that the outputs are merged into one result objects.

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

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

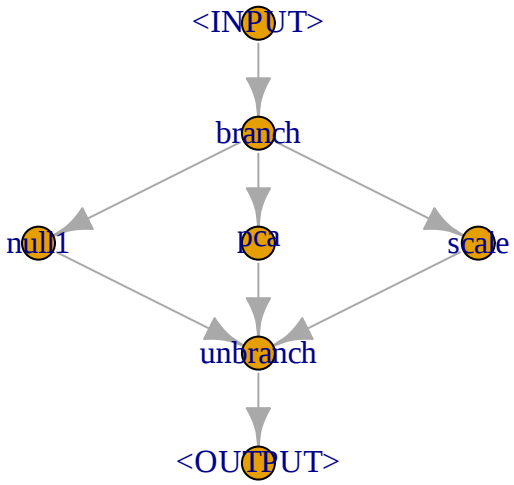
Without "unbranching" one creates the following graph:

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



Now when "unbranching", we obtain the following results:

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



The same can be achieved using a shorter notation:

```
# 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
```

5.5.2 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.

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

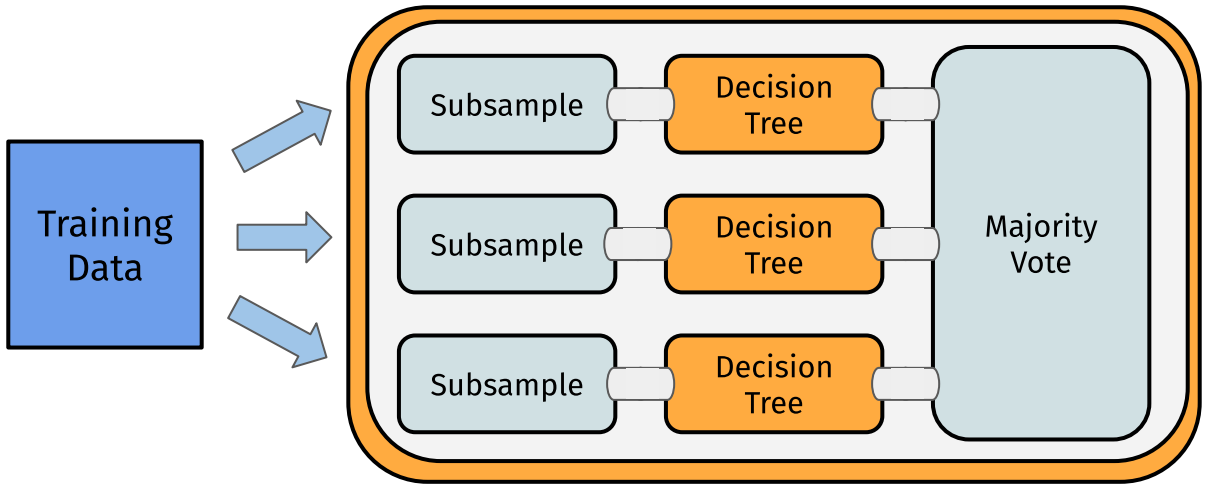
5.5.2.1 Bagging

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 = po("subsample", frac = 0.7) %>>%
  po("learner", lrn("classif.rpart"))
```

We can now copy this operation 10 times using `pipeline_greplicate`. The `pipeline_greplicate` allows us to parallelize many copies of an operation by creating a Graph containing `n` copies of the input Graph. We can also create it using **syntactic sugar** via `ppl()`:

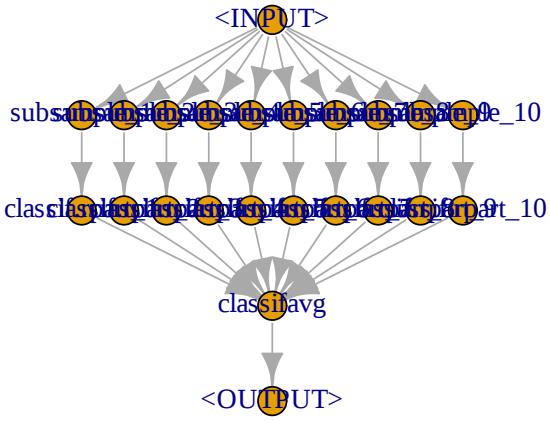
```
pred_set = ppl("greplicate", single_pred, 10L)
```

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

```
bagging = pred_set %>>%
  po("classifavg", innum = 10)
```

Now we can plot again to see what happens:

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



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

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

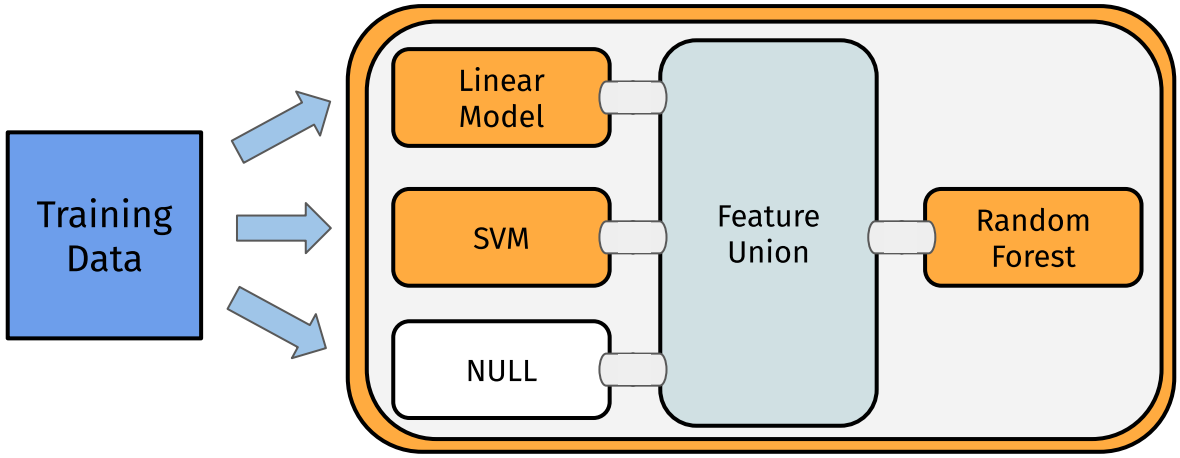
```
## <PredictionClassif> for 30 observations:
##     row_ids     truth  response prob.setosa prob.versicolor prob.virginica
##           1    setosa    setosa           1             0.0            0.0
##           2    setosa    setosa           1             0.0            0.0
##           6    setosa    setosa           1             0.0            0.0
## ---
##         130 virginica virginica           0             0.4            0.6
##         142 virginica virginica           0             0.0            1.0
##         144 virginica virginica           0             0.0            1.0
```

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.

5.5.2.2 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 = lrn("classif.rpart")
lrn_0 = po("learner_cv", 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.

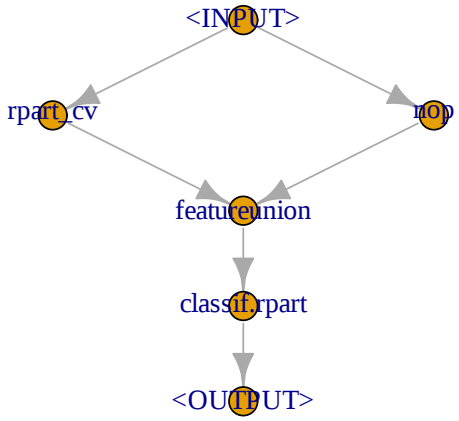
```
level_0 = gunion(list(lrn_0, po("nop")))
```

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

```
combined = level_0 %>>% po("featureunion", 2)
```

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

```
stack = combined %>>% po("learner", lrn$clone())
stack$plot(html = FALSE)
```




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

```
## <PredictionClassif> for 30 observations:
##     row_ids     truth   response
##           1    setosa     setosa
##           2    setosa     setosa
##           6    setosa     setosa
## ---
##         130 virginica versicolor
##         142 virginica  virginica
##         144 virginica  virginica
```

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.

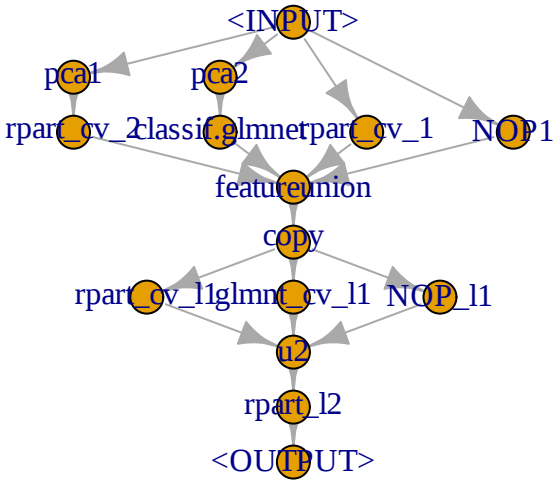
5.5.2.3 Multilevel Stacking

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 = po("learner_cv", rprt, id = "rpart_cv_1")
lrn_0$param_set$values$maxdepth = 5L
lrn_1 = po("pca", id = "pca1") %>>% po("learner_cv", rprt, id = "rpart_cv_2")
lrn_1$param_set$values$rpart_cv_2.maxdepth = 1L
lrn_2 = po("pca", id = "pca2") %>>% po("learner_cv", glmn)

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

# Cbind the output 3 times, train 2 learners but also keep level
# 0 predictions
level_1 = level_0 %>>%
  po("featureunion", 4) %>>%
  po("copy", 3) %>>%
  gunion(list(
    po("learner_cv", rprt, id = "rpart_cv_l1"),
    po("learner_cv", glmn, id = "glmnt_cv_l1"),
    po("nop", id = "NOP_l1")
  ))

# Cbind predictions, train a final learner
level_2 = level_1 %>>%
  po("featureunion", 3, id = "u2") %>>%
  po("learner", rprt, id = "rpart_l2")

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




```
task = tsk("iris")
lrn = as_learner(level_2)
```

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

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

```
## classif.ce
##        0.1
```

5.6 Special Operators

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

5.6.1 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 PipeOps, PipeOpImpute:

- Impute numeric values from a histogram
- Adds a new level for factors
- Add a column marking whether a value for a given feature was missing or not (numeric only)
- We use `po("featureunion")` to cbind the missing indicator features.

```
pom = po("missind")
pon = po("imputehist", id = "imputer_num", affect_columns = is.numeric)
pof = po("imputeoor", id = "imputer_fct", 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.

```
polrn = po("learner", lrn("classif.rpart"))
lrn = as_learner(imputer %>>% polrn)
```

5.6.2 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:

```
pom = po("mutate")

# 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`).

5.6.3 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.

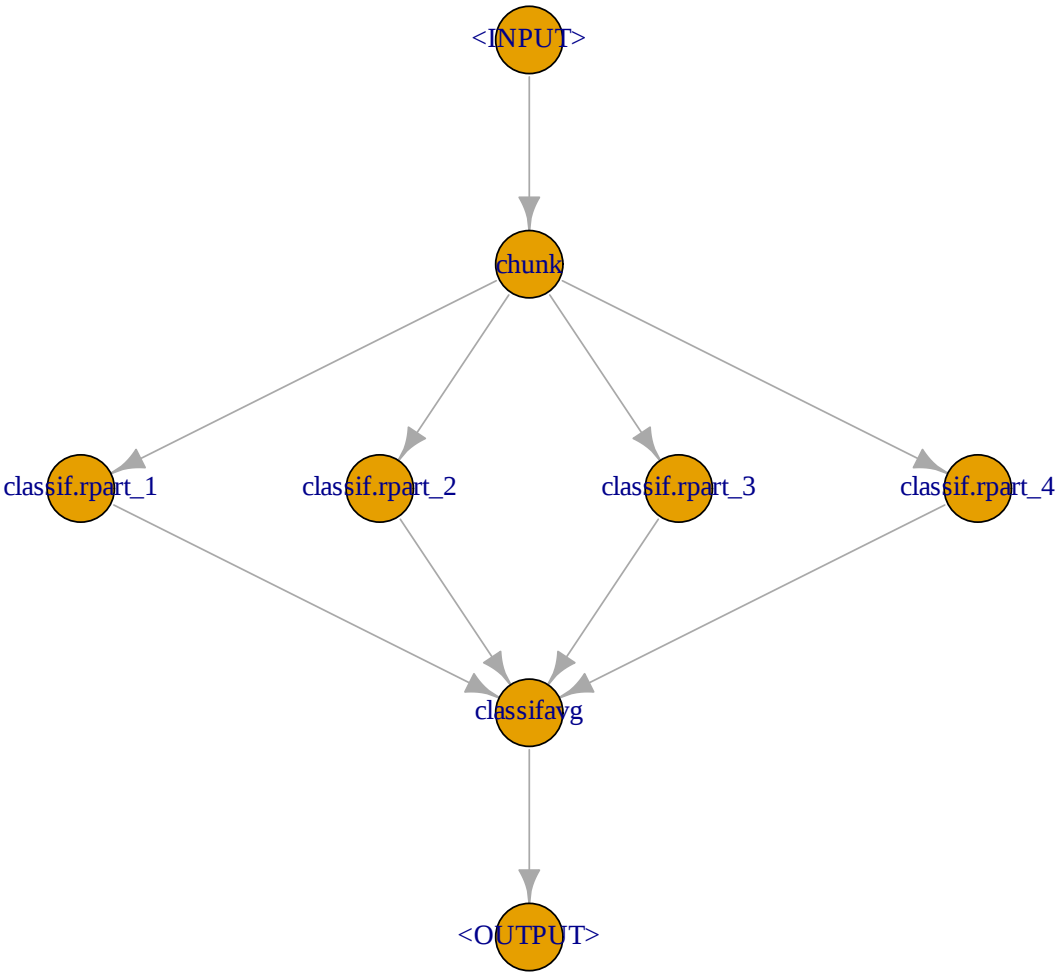
```
chks = po("chunk", 4)
lrns = ppl("greplicate", po("learner", lrn("classif.rpart")), 4)
```

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

```
mjv = po("classifavg", 4)
```

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

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




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

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

```
## classif.ce
##     0.2167
```

5.6.4 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`:

```
po("filter", mlr3filters::flt("information_gain"))
```

```
## 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`.

5.7 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.

5.7.1 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 = as_task_classif(iris, target = "Species")
lrn = lrn("classif.rpart")
rsmp = rsmp("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`.

5.7.2 `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: `PipeOp`s 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 = po("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()
```

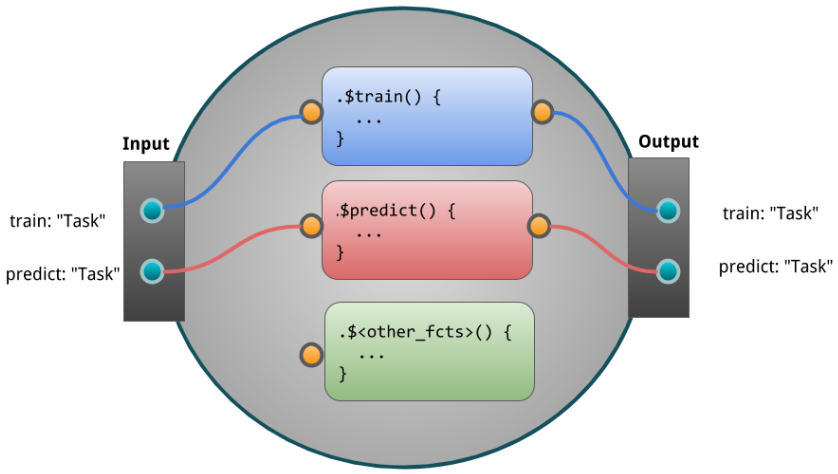
```
##    Species    PC1    PC2      PC3       PC4
## 1:  setosa -2.684 0.3194 -0.02791 -0.002262
```

```
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:



5.7.2.1 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.

5.7.2.2 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_pipeops` dictionary: `po("pca")`. The entire list of available `PipeOps`, and some meta-information, can be retrieved using `as.data.table()`:

```
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:              colroles         1          1
## 10:                  copy         1         NA
## 11:          datefeatures         1          1
## 12:                encode         1          1
## 13:          encodeimpact         1          1
## 14:            encodelmer         1          1
## 15:          featureunion        NA          1
## 16:                filter         1          1
## 17:            fixfactors         1          1
## 18:               histbin         1          1
## 19:                   ica         1          1
## 20:        imputeconstant         1          1
## 21:            imputehist         1          1
## 22:         imputelearner         1          1
## 23:            imputemean         1          1
## 24:          imputemedian         1          1
## 25:            imputemode         1          1
## 26:             imputeoor         1          1
## 27:          imputesample         1          1
## 28:             kernelpca         1          1
## 29:               learner         1          1
## 30:            learner_cv         1          1
## 31:               missind         1          1
## 32:           modelmatrix         1          1
## 33:     multiplicityexply         1         NA
## 34:     multiplicityimply        NA          1
## 35:                mutate         1          1
## 36:                   nmf         1          1
## 37:                   nop         1          1
## 38:              ovrsplit         1          1
## 39:              ovrunite         1          1
```

```
## 40:                   pca         1          1
## 41:                 proxy        NA          1
## 42:           quantilebin         1          1
## 43:      randomprojection         1          1
## 44:        randomresponse         1          1
## 45:               regravg        NA          1
## 46:       removeconstants         1          1
## 47:         renamecolumns         1          1
## 48:             replicate         1          1
## 49:                 scale         1          1
## 50:           scalemaxabs         1          1
## 51:            scalerange         1          1
## 52:                select         1          1
## 53:                 smote         1          1
## 54:           spatialsign         1          1
## 55:            subsample         1          1
## 56:          targetinvert         2          1
## 57:          targetmutate         1          2
## 58: targettrafoscalerange         1          2
## 59:        textvectorizer         1          1
## 60:             threshold         1          1
## 61:         tunethreshold         1          1
## 62:               unbranch        NA          1
## 63:                vtreat         1          1
## 64:            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.

```
po("pca", 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]
```

5.7.3 PipeOp Channels

5.7.3.1 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 = po("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.
## po("backuplearner")$input
```

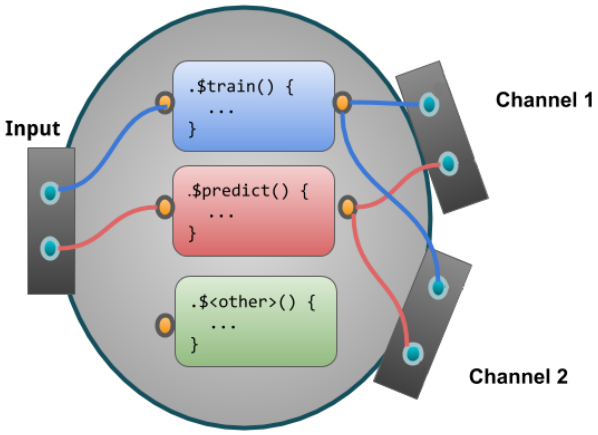
5.7.3.2 Output Channels

Unlike the typical notion of a function, `PipeOps` can also have multiple **output channels**. `$train()` and `$predict()` always return a list, so certain `PipeOps` 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.

```
po("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:



5.7.3.3 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:

```
## po("backuplearner")
```

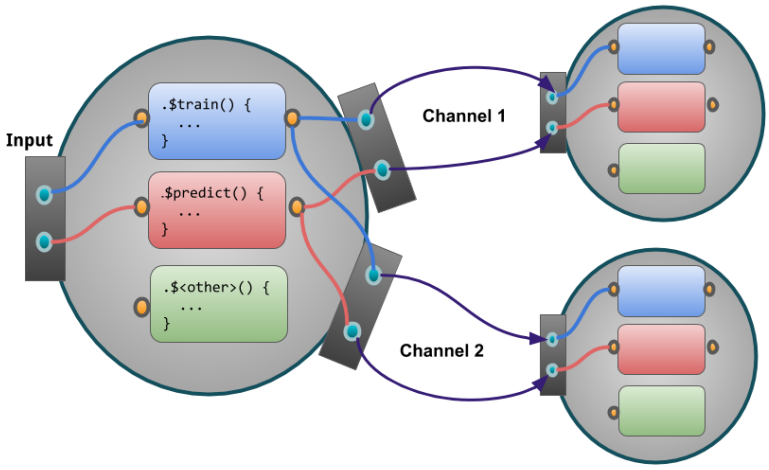
5.7.4 `Graph`: Networks of `PipeOp`s

5.7.4.1 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 `PipeOp`s 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`:



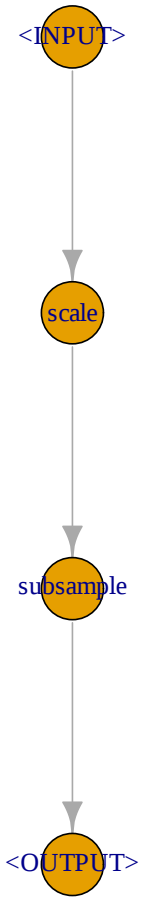
A `Graph` is empty when first created, and `PipeOps` can be added using the **`$add_pipeop()`** method. The **`$add_edge()`** method is used to create connections between them. While the printer of a `Graph` gives some information about its layout, the most intuitive way of visualizing it is using the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

```
gr = Graph$new()
gr$add_pipeop(po("scale"))
gr$add_pipeop(po("subsample", frac = 0.1))
gr$add_edge("scale", "subsample")
```

```
print(gr)
```

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

```
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      -1.2791      -1.311     -1.50149     0.09789
##  2:    setosa      -1.3358      -1.311     -1.01844     1.24503
##  3:    setosa      -1.3358      -1.180     -1.50149     0.78617
##  4:    setosa      -1.2791      -1.311     -1.01844     0.78617
##  5:    setosa      -1.2791      -1.442     -1.13920     0.09789
##  6:    setosa      -1.2791      -1.442     -0.77691     2.39217
```

```
##  7:    setosa      -1.4490      -1.311     -1.01844      0.32732
##  8:    setosa      -1.3924      -1.311     -0.41462      1.01560
##  9:    setosa      -1.2791      -1.311     -0.89767      0.78617
## 10:    setosa      -1.2791      -1.311     -0.65615      1.47446
## 11: virginica       0.7602       1.575     -0.05233     -0.59040
## 12: virginica       0.9868       0.788      0.79301     -0.13154
## 13: virginica       0.6469       0.788      0.55149     -0.81982
## 14: virginica       0.6469       0.788      0.30996     -0.13154
## 15: virginica       1.0434       1.313      0.67225     -0.59040
```

```
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.

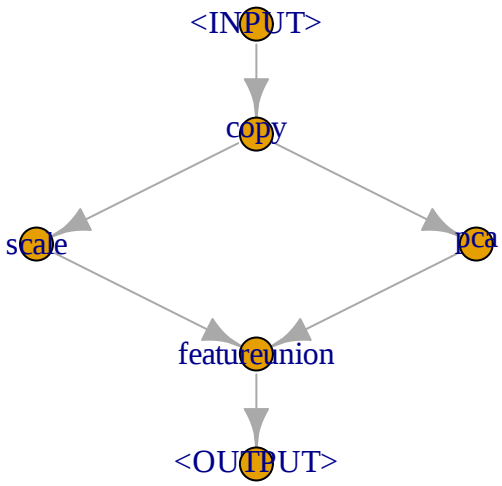
5.7.4.2 Networks

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".

```
gr = Graph$new()$
  add_pipeop(po("copy", outnum = 2))$
  add_pipeop(po("scale"))$
  add_pipeop(po("pca"))$
  add_pipeop(po("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.




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

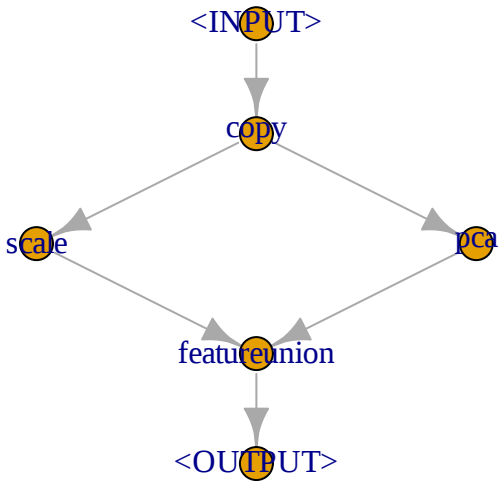
```
##         Species Petal.Length Petal.Width    PC1       PC2
##    1:    setosa      -1.3358     -1.3111 -2.561 -0.006922
##    2:    setosa      -1.3358     -1.3111 -2.561 -0.006922
##    3:    setosa      -1.3924     -1.3111 -2.653  0.031850
##    4:    setosa      -1.2791     -1.3111 -2.469 -0.045694
##    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
```

5.7.4.3 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 `PipeOp`s or `Graph`s and arranges them next to each other akin to a (disjoint) graph union, the above network can more easily be constructed as follows:

```
gr = po("copy", outnum = 2) %>>%
  gunion(list(po("scale"), po("pca"))) %>>%
  po("featureunion", innum = 2)

gr$plot(html = FALSE)
```



5.7.4.4 `PipeOp` IDs and ID Name Clashes

`PipeOp`s within a graph are addressed by their **`$id`**-slot. It is therefore necessary for all `PipeOp`s within a `Graph` to have a unique `$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 = po("scale")
po2 = po("scale")
po1 %>>% po2 ## name clash
```

```
## Error in gunion(list(g1, g2), in_place = c(TRUE, TRUE)): Assertion on 'ids of pipe operato
```

```
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:
po("scale", id = "scale2")
```

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

```
po("scale", id = "scale2")
```

```
## PipeOp: <scale2> (not trained)
## values: <robust=FALSE>
## 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 = po("scale") %>>% po("pca")
gr %>>% gr2
```

```
## Error in gunion(list(g1, g2), in_place = c(TRUE, TRUE)): Assertion on 'ids of pipe operato
```

```
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
```

5.7.5 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`.

5.7.5.1 `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 = po("pca") %>>% po("learner",
  lrn("classif.rpart"))
```

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

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

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

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

5.7.5.2 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 = as_learner(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
```

5.7.6 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 tuning chapter or the in-depth paradox chapter.

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 = po("pca")
op_pca$param_set
```

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

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 = po("pca", 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 %>>% po("scale")
gr$param_set
```

```
## <ParamSetCollection>
##                         id    class lower upper nlevels        default value
## 1:           pca.center ParamLgl    NA    NA       2           TRUE  TRUE
## 2:           pca.scale. ParamLgl    NA    NA       2          FALSE
## 3:            pca.rank. ParamInt     1   Inf     Inf
## 4:   pca.affect_columns ParamUty    NA    NA     Inf  <Selector[1]>
## 5:         scale.center ParamLgl    NA    NA       2           TRUE
## 6:          scale.scale ParamLgl    NA    NA       2           TRUE
## 7:         scale.robust ParamLgl    NA    NA       2 <NoDefault[3]> FALSE
## 8: scale.affect_columns ParamUty    NA    NA     Inf  <Selector[1]>
```

```
gr$param_set$values
```

```
## $pca.center
## [1] TRUE
##
## $scale.robust
## [1] FALSE
```

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

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

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

```
glrn = as_learner(gr %>>% op_rpart)
glrn$param_set
```

```
## <ParamSetCollection>
##                                id    class lower upper nlevels        default
##  1:                   pca.center ParamLgl    NA    NA       2           TRUE
##  2:                   pca.scale. ParamLgl    NA    NA       2          FALSE
##  3:                    pca.rank. ParamInt     1   Inf     Inf
##  4:           pca.affect_columns ParamUty    NA    NA     Inf  <Selector[1]>
##  5:                 scale.center ParamLgl    NA    NA       2           TRUE
##  6:                  scale.scale ParamLgl    NA    NA       2           TRUE
##  7:                 scale.robust ParamLgl    NA    NA       2 <NoDefault[3]>
##  8:         scale.affect_columns ParamUty    NA    NA     Inf  <Selector[1]>
##  9:             classif.rpart.cp ParamDbl     0     1     Inf           0.01
## 10:     classif.rpart.keep_model ParamLgl    NA    NA       2          FALSE
## 11:     classif.rpart.maxcompete ParamInt     0   Inf     Inf              4
## 12:       classif.rpart.maxdepth ParamInt     1    30      30             30
## 13:   classif.rpart.maxsurrogate ParamInt     0   Inf     Inf              5
## 14:      classif.rpart.minbucket ParamInt     1   Inf     Inf <NoDefault[3]>
## 15:       classif.rpart.minsplit ParamInt     1   Inf     Inf             20
## 16: classif.rpart.surrogatestyle ParamInt     0     1       2              0
## 17:   classif.rpart.usesurrogate ParamInt     0     2       3              2
## 18:           classif.rpart.xval ParamInt     0   Inf     Inf             10
##     value
##  1:  TRUE
##  2:
##  3:
##  4:
##  5:
##  6:
##  7: FALSE
##  8:
##  9:
## 10:
## 11:
## 12:
## 13:
## 14:
## 15:
## 16:
## 17:
## 18:     0
```

6 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.

6.1 Parallelization

Parallelization refers to the process of running multiple jobs in parallel, simultaneously. This process allows for significant savings in computing power. We distinguish between implicit parallelism and explicit parallelism.

6.1.1 Implicit Parallelization

We talk about implicit parallelization in this context if we call external code (i.e., code from foreign CRAN packages) which runs in parallel. Many machine learning algorithms can parallelize their model fit using threading, e.g. `ranger` or `xgboost`. Unfortunately, threading conflicts with certain parallel backends used during explicit parallelization, causing the system to be overutilized in the best case and causing hangs or segfaults in the worst case. For this reason, we introduced the convention that implicit parallelization is turned off in the defaults, but can be enabled again via a hyperparameter which is tagged with the label `"threads"`.

```
library("mlr3verse")

learner = lrn("classif.ranger")
learner$param_set$ids(tags = "threads")
```

```
## [1] "num.threads"
```

To enable the parallelization for this learner, we simply can call the helper function `set_threads()`):

```
# set to use 4 CPUs
set_threads(learner, n = 4)
```

```
## <LearnerClassifRanger:classif.ranger>
## * Model: -
## * Parameters: num.threads=4
## * Packages: mlr3, mlr3learners, ranger
## * Predict Type: response
## * Feature types: logical, integer, numeric, character, factor, ordered
## * Properties: hotstart_backward, importance, multiclass, oob_error,
##   twoclass, weights
```

```
# auto-detect cores on the local machine
set_threads(learner)
```

```
## <LearnerClassifRanger:classif.ranger>
## * Model: -
## * Parameters: num.threads=2
## * Packages: mlr3, mlr3learners, ranger
## * Predict Type: response
## * Feature types: logical, integer, numeric, character, factor, ordered
## * Properties: hotstart_backward, importance, multiclass, oob_error,
##   twoclass, weights
```

This also works for filters from mlr3filters and lists of objects, even if some objects do not support threading at all:

```
# retrieve 2 filters
# * variance filter with no support for threading
# * mrmr filter with threading support
filters = flts(c("variance", "mrmr"))

# set threads for all filters which support it
set_threads(filters, n = 4)
```

```
## [[1]]
## <FilterVariance:variance>
## Task Types: classif, regr
## Task Properties: -
## Packages: stats
## Feature types: integer, numeric
##
## [[2]]
## <FilterMRMR:mrmr>
## Task Types: classif, regr
## Task Properties: -
## Packages: praznik
## Feature types: integer, numeric, factor, ordered
```

```
# variance filter is unchanged
filters[[1]]$param_set
```

```
## <ParamSet>
##       id    class lower upper nlevels default value
## 1: na.rm ParamLgl    NA    NA       2    TRUE
```

```
# mrmr now works in parallel with 4 cores
filters[[2]]$param_set
```

```
## <ParamSet>
##         id    class lower upper nlevels default value
## 1: threads ParamInt     0   Inf     Inf       0     4
```

6.1.2 Explicit Parallelization

We talk about explicit parallelization here if mlr3 starts the parallelization itself. The abstraction implemented in future is used to support a broad range of parallel backends. There are two use cases where mlr3 calls future: `resample()` and `benchmark()`. During resampling, all resampling iterations can be executed in parallelization. The same holds for benchmarking, where additionally all combinations in the provided design are also independent. These loops are performed by future using the parallel backend configured with `future::plan()`. Extension packages like mlr3tuning internally call `benchmark()` during tuning and thus work in parallel, too.

In this section, we will use the `spam task` and a simple `classification tree` to showcase the explicit parallelization. In this example, the `future::multisession` parallel backend is selected which should work on all systems.

```
# select the multisession backend
future::plan("multisession")

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, but in practice you cannot expect the runtime to fall linearly as the number of cores increases (Amdahl's law). Depending on the parallel backend, the technical overhead for starting workers, communicating objects, sending back results and shutting down the workers can be quite large. Therefore, it is advised to only enable parallelization for resamplings where each iteration runs at least some seconds.

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 (except for nested resampling, briefly described in the following section). 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.

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

6.1.3 Nested Resampling Parallelization

Nested resampling results in two nested resampling loops. We can choose different parallelization backends for the inner and outer resampling loop, respectively. We just have to pass a list of future backends:

```
# Runs the outer loop in parallel and the inner loop sequentially
future::plan(list("multisession", "sequential"))
# Runs the outer loop sequentially and the inner loop in parallel
future::plan(list("sequential", "multisession"))
```

While nesting real parallelization backends is often unintended and causes unnecessary overhead, it is useful in some distributed computing setups. It can be achieved with future by forcing a fixed number of workers for each loop:

```
# Runs both loops in parallel
future::plan(list(future::tweak("multisession", workers = 2),
  future::tweak("multisession", workers = 4)))
```

This example would run on 8 cores (`= 2 * 4`) on the local machine. The vignette of the future package gives more insight into nested parallelization.

6.2 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: mlr3
## * Predict Type: response
## * Feature types: logical, integer, numeric, character, factor, ordered
## * Properties: hotstart_forward, 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 nlevels        default value
##  1:        error_predict ParamDbl     0     1     Inf              0
##  2:          error_train ParamDbl     0     1     Inf              0
##  3:      message_predict ParamDbl     0     1     Inf              0
##  4:        message_train ParamDbl     0     1     Inf              0
##  5:      predict_missing ParamDbl     0     1     Inf              0
##  6: predict_missing_type ParamFct    NA    NA       2             na
##  7:           save_tasks ParamLgl    NA    NA       2          FALSE
##  8:     segfault_predict ParamDbl     0     1     Inf              0
##  9:       segfault_train ParamDbl     0     1     Inf              0
## 10:          sleep_train ParamUty    NA    NA     Inf <NoDefault[3]>
## 11:        sleep_predict ParamUty    NA    NA     Inf <NoDefault[3]>
## 12:              threads ParamInt     1   Inf     Inf <NoDefault[3]>
## 13:      warning_predict ParamDbl     0     1     Inf              0
## 14:        warning_train ParamDbl     0     1     Inf              0
## 15:                    x ParamDbl     0     1     Inf <NoDefault[3]>
## 16:                 iter ParamInt     1   Inf     Inf              1
```

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         50         50        50
##   versicolor      0          0         0
##   virginica       0          0         0
```

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 .__LearnerClassifDebug__.train(self = self, private = private, : Error from class
```

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.

6.2.1 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.

6.2.2 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:

```
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: mlr3
## * Predict Type: response
## * Feature types: logical, integer, numeric, character, factor, ordered
## * Properties: hotstart_forward, 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:

```
learner$model
```

```
## NULL
```

```
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:

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

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[22]>    iris classif.debug            cv    10        0
## 2:  2 <ResampleResult[22]>    iris classif.rpart            cv    10        0
##    errors classif.ce
## 1:      2    0.74000
## 2:      0    0.06667
```

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

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

```
##    iteration                                msg
## 1:         1 Error from classif.debug->train()
## 2:         5 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:

```
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>
##         75          0         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.

6.3 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.

6.3.1 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)
```

```
## tibble [336,776 x 19] (S3: tbl_df/tbl/data.frame)
##  $ year          : int [1:336776] 2013 2013 2013 2013 2013 2013 2013 2013 2013 2013 ...
##  $ month         : int [1:336776] 1 1 1 1 1 1 1 1 1 1 ...
##  $ day           : int [1:336776] 1 1 1 1 1 1 1 1 1 1 ...
##  $ dep_time      : int [1:336776] 517 533 542 544 554 554 555 557 557 558 ...
##  $ sched_dep_time: int [1:336776] 515 529 540 545 600 558 600 600 600 600 ...
##  $ dep_delay     : num [1:336776] 2 4 2 -1 -6 -4 -5 -3 -3 -2 ...
##  $ arr_time      : int [1:336776] 830 850 923 1004 812 740 913 709 838 753 ...
##  $ sched_arr_time: int [1:336776] 819 830 850 1022 837 728 854 723 846 745 ...
##  $ arr_delay     : num [1:336776] 11 20 33 -18 -25 12 19 -14 -8 8 ...
##  $ carrier       : chr [1:336776] "UA" "UA" "AA" "B6" ...
##  $ flight        : int [1:336776] 1545 1714 1141 725 461 1696 507 5708 79 301 ...
##  $ tailnum       : chr [1:336776] "N14228" "N24211" "N619AA" "N804JB" ...
##  $ origin        : chr [1:336776] "EWR" "LGA" "JFK" "JFK" ...
##  $ dest          : chr [1:336776] "IAH" "IAH" "MIA" "BQN" ...
##  $ air_time      : num [1:336776] 227 227 160 183 116 150 158 53 140 138 ...
##  $ distance      : num [1:336776] 1400 1416 1089 1576 762 ...
##  $ hour          : num [1:336776] 5 5 5 5 6 5 6 6 6 6 ...
##  $ minute        : num [1:336776] 15 29 40 45 0 58 0 0 0 0 ...
##  $ time_hour     : POSIXct[1:336776], format: "2013-01-01 05:00:00" "2013-01-01 05:00:00"
```

```
# 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)
```

6.3.2 Preprocessing with `dplyr`

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

```
# 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
```

```
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))
```

6.3.3 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.

6.3.4 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_train" and "time_predict"

```
task = as_task_regr(b, id = "flights_sqlite", 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
##         1212            0            0
```

6.3.5 Cleanup

Finally, we remove the tbl object and close the connection.

```
rm(tbl)
DBI::dbDisconnect(con)
```

6.4 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.

6.4.1 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 nlevels        default value
## 1:  a ParamLgl    NA    NA       2 <NoDefault[3]>
```

```
print(ps2) # contains the same reference as ps
```

```
## <ParamSet>
##    id    class lower upper nlevels        default value
## 1:  a ParamLgl    NA    NA       2 <NoDefault[3]>
```

```
print(ps3) # is a "clone" of the old (empty) ps
```

```
## <ParamSet>
## Empty.
```

6.4.2 Defining a Parameter Space

6.4.2.1 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 `factor`s
- `ParamLgl`: Truth values (`TRUE` / `FALSE`), as `logical`s in R
- `ParamUty`: Parameter that can take any value

A particular instance of a parameter is created by calling the attached `$new()` function:

```
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 stor
## 1:  A ParamLgl    NA    NA  TRUE,FALSE       2       TRUE    <list[0]> <NoDefault[3]>
```

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.

6.4.2.2 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)
ps$add(ParamSet$new(list(parD, parE)))
print(ps)
```

```
## <ParamSet>
##    id    class lower upper nlevels        default value
## 1:  A ParamLgl    NA    NA       2 <NoDefault[3]>
## 2:  B ParamInt     0    10      11 <NoDefault[3]>
## 3:  C ParamDbl     0     4     Inf <NoDefault[3]>
## 4:  D ParamFct    NA    NA       3              y
## 5:  E ParamUty    NA    NA     Inf <NoDefault[3]>
```

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 nlevels        default value
## 1:  A ParamLgl    NA    NA       2 <NoDefault[3]>
## 2:  B ParamInt     0    10      11 <NoDefault[3]>
## 3:  C ParamDbl     0     4     Inf <NoDefault[3]>
```

[1] Although the name is suggestive of a "Set"-valued Param, this is unrelated to the other objects that follow the ParamXxx naming scheme.

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 stor
## 1:  A ParamLgl    NA    NA TRUE,FALSE       2       TRUE    <list[0]> <NoDefault[3]>
## 2:  B ParamInt     0    10                  11       TRUE    <list[0]> <NoDefault[3]>
## 3:  C ParamDbl     0     4                 Inf       TRUE    <list[1]> <NoDefault[3]>
## 4:  D ParamFct    NA    NA       x,y,z       3       TRUE    <list[0]>               y    c
## 5:  E ParamUty    NA    NA                 Inf      FALSE    <list[0]> <NoDefault[3]>
```

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 self$assert(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] "The parameter 'B' can only be set if the following condition is met 'D   {x, y}'. Ins
```

```
ps$check(list(A = FALSE, B = 1))                   # B's dependency is not met
```

```
## [1] "The parameter 'B' can only be set if the following condition is met 'D   {x, y}'. Ins
```

```
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] "The parameter 'D' can only be set if the following condition is met 'A = FALSE'. Inst
```

```
ps$check(list(A = TRUE, B = 1))                   # B's dependency is not met
```

```
## [1] "The parameter 'B' can only be set if the following condition is met 'D   {x, y}'. Ins
```

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[9]>
## 2:  B  D <CondAnyOf[9]>
```

6.4.2.3 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 nlevels        default value
## 1: x_rep_1 ParamDbl     0     1     Inf <NoDefault[3]>
## 2: x_rep_2 ParamDbl     0     1     Inf <NoDefault[3]>
```

```
ps$add(ps2d)
print(ps)
```

```
## <ParamSet>
##         id    class lower upper nlevels        default parents value
## 1:       A ParamLgl    NA    NA       2 <NoDefault[3]>           TRUE
## 2:       B ParamInt     0    10      11 <NoDefault[3]>       D     1
## 3:       C ParamDbl     0     4     Inf <NoDefault[3]>
## 4:       D ParamFct    NA    NA       3              y       A
## 5:       E ParamUty    NA    NA     Inf <NoDefault[3]>
## 6: x_rep_1 ParamDbl     0     1     Inf <NoDefault[3]>
## 7: x_rep_2 ParamDbl     0     1     Inf <NoDefault[3]>
```

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"
```

6.4.3 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.

6.4.3.1 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.

6.4.3.2 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))
```

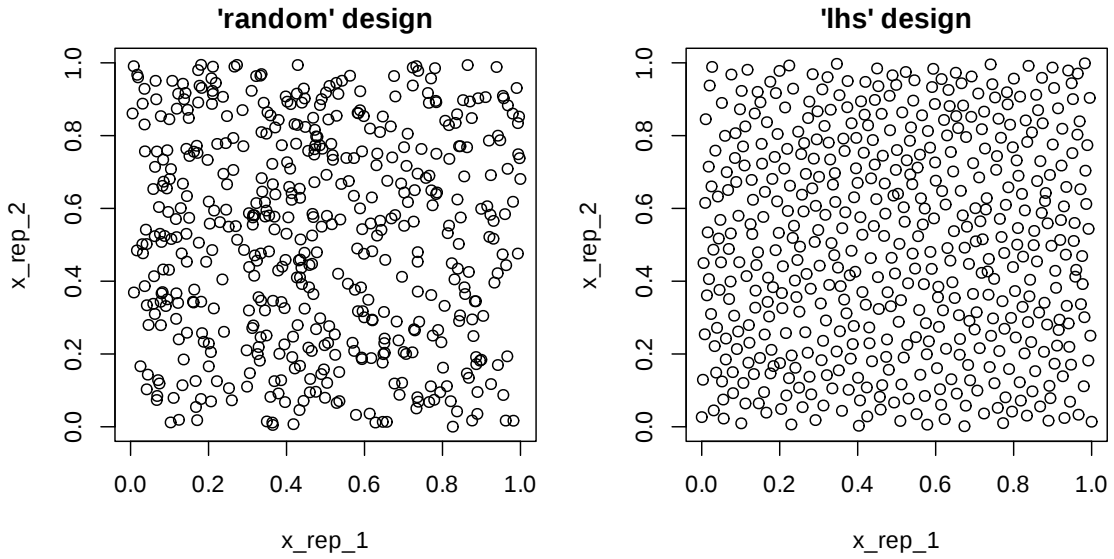
```
## <Design> with 4 rows:
##    B C     A
## 1: 0 0  TRUE
## 2: 0 0 FALSE
## 3: 0 4  TRUE
## 4: 0 4 FALSE
```

6.4.3.3 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)
```



6.4.3.4 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:  TRUE
## 2: FALSE
## 3: FALSE
## 4: FALSE
## 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  521
##   TRUE   52  53  35  43  39  30  48  46  52  37  44    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: 0.4650 0.2377
## 2: 0.1739 0.1938
## 3: 0.6586 0.1461
## 4: 0.9628 0.1311
## 5: 0.4808 0.3562
```

SamplerUnif The Sampler used in generate_design_random is the SamplerUnif sampler, which corresponds to a HierarchicalSampler of Sampler1DUnif for all parameters.

6.4.4 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: 0.2679
## 2: 0.2079
```

```
design$transpose()  # trafo is TRUE
```

```
## [[1]]
## [[1]]$par
## [1] 1.317
##
##
## [[2]]
## [[2]]$par
## [1] 1.571
```

Compare this to $transpose() without transformation:

```
design$transpose(trafo = FALSE)
```

```
## [[1]]
## [[1]]$par
## [1] 0.2679
##
```

```
##
## [[2]]
## [[2]]$par
## [1] 0.2079
```

6.4.4.1 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 `ParamSet`s 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 nlevels        default value
## 1: fun ParamUty    NA    NA     Inf <NoDefault[3]>
```

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: median
## 2: median
```

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 (x, na.rm = FALSE, ...)
## UseMethod("median")
## <bytecode: 0x55cee196d840>
## <environment: namespace:stats>
```

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] "fun: Must have exactly 1 formal arguments, but has 2"
```

```
xvals[[1]]$fun(1:10)
```

```
## [1] 5.5
```

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:   0.8981
## 2:   0.8832
```

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: 0x55cee423b9e0>
```

```
methodPS$check(xvals[[1]])
```

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

```
xvals[[1]]$fun(1:10)
```

```
## 89.80622%
##     9.083
```

6.5 Logging

We use the lgr package for logging and progress output.

6.5.1 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:

```
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:

```
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:

```
lgr::get_logger("mlr3")$set_threshold("debug")
```

To reduce the verbosity, reduce the log level to warn:

```
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 the optimization packages such as mlr3tuning mlr3fselect use the logger of their base package bbotk. To disable the output from mlr3, but keep the output from mlr3tuning, reduce the verbosity for the logger mlr3 and optionally change the logger bbotk to the desired level.

```
lgr::get_logger("mlr3")$set_threshold("warn")
lgr::get_logger("bbotk")$set_threshold("info")
```

6.5.2 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:

```
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")

# print the contents of the file
cat(readLines(tf))

# remove the appender again
logger$remove_appender("json")
```

6.5.3 Immediate Log Feedback

mlr3 uses the future package and encapsulation to make evaluations fast, stable, and reproducible. However, this may lead to logs being delayed, out of order, or, in case of some errors, not present at all. When it is necessary to have immediate access to log messages, for example to investigate problems, one may therefore choose to disable future and encapsulation. This can be done by enabling the debug mode using `options(mlr.debug = TRUE)`; the `$encapsulate` slot of learners should also be set to `"none"` (default) or `"evaluate"`, but not `"callr"`. This should only be done to investigate problems, however, and not for production use, because (1) this disables parallelization, and (2) this leads to different RNG behavior and therefore to results that are not reproducible when the debug mode is not set.

7 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.

7.1 Adding new Learners

Here, we show how to create a custom mlr3learner step-by-step using `mlr3extralearners::create_learner`.

It is strongly recommended that you **first** open a learner request issue to discuss the learner you want to implement if you plan on creating a pull request to the mlr-org. This allows us to discuss the purpose and necessity of the learner before you start to put the real work in!

This section gives insights on how a mlr3learner is constructed and how to troubleshoot issues. See the Learner FAQ subsection for help.

Summary of steps for adding a new learner

1. Check the learner does not already exist here.
2. Fork, clone and load `mlr3extralearners`.
3. Run `mlr3extralearners::create_learner`.
4. Add the learner `param_set`.
5. Manually add `.train` and `.predict` private methods to the learner.
6. If applicable add `importance` and `oob_error` public methods to the learner.
7. If applicable add references to the learner.
8. Check unit tests and paramtests pass (these are automatically created).
9. Run cleaning functions
10. Open a pull request with the new learner template.

(Do not copy/paste the code shown in this section. Use the create_learner to start.)

7.1.1 Setting-up mlr3extralearners

In order to use the `mlr3extralearners::create_learner` function you must have a local copy of the `mlr3extralearners` repository and must specify the correct path to the package. To do so, follow these steps:

1. Fork the repository
2. Clone a local copy of your forked repository.

Then do one of:

- Open a new R session, call `library("mlr3extralearners")` (install if you haven't already), and then run `mlr3extralearners::create_learner` with the `pkg` argument set as the path (the folder location) to the package directory.
- Open a new R session, set your working directory as your newly cloned repository, run `devtools::load_all`, and then run `mlr3extralearners::create_learner`, leaving `pkg = "."`.
- In your newly cloned repository, open the R project, which will automatically set your working directory, run `devtools::load_all`, and then run `mlr3extralearners::create_learner`, leaving `pkg = "."`.

We recommend the last option. It is also important that you are familiar with the three `devtools` commands:

- `devtools::document` - Generates roxygen documentation for your new learner.
- `devtools::load_all` - Loads all functions from `mlr3extralearners` locally, including hidden helper functions.
- `devtools::check` - Checks that the package still passes all tests locally.

7.1.2 Calling create_learner

The learner `classif.rpart` will be used as a running example throughout this section.

```
library("mlr3extralearners")
create_learner(
  pkg = ".",
  classname = "Rpart",
  algorithm = "decision tree",
  type = "classif",
  key = "rpart",
  package = "rpart",
  caller = "rpart",
  feature_types = c("logical", "integer", "numeric", "factor", "ordered"),
  predict_types = c("response", "prob"),
  properties = c("importance", "missings", "multiclass", "selected_features", "twoclass", "weights"),
  references = TRUE,
  gh_name = "RaphaelS1"
)
```

The full documentation for the function arguments is in `mlr3extralearners::create_learner`, in this example we are doing the following:

1. `pkg = "."` - Set the package root to the current directory (assumes `mlr3extralearners` already set as the working directory)
2. `classname = "Rpart"` - Set the R6 class name to LearnerClassifRpart (classif is below)
3. `algorithm = "decision tree"` - Create the title as "Classification Decision Tree Learner", where "Classification" is determined automatically from `type` and "Learner" is added for all learners.
4. `type = "classif"` - Setting the learner as a classification learner, automatically filling the title, class name, id ("classif.rpart") and task type.
5. `key = "rpart"` - Used with `type` to create the unique ID of the learner, `classif.rpart`.
6. `package = "rpart"` - Setting the package from which the learner is implemented, this fills in things like the training function (along with `caller`) and the `man` field.
7. `caller = "rpart"` - This tells the `.train` function, and the description which function is called to run the algorithm, with `package` this automatically fills `rpart::rpart`.
8. `feature_types = c("logical", "integer", "numeric", "factor", "ordered")` - Sets the type of features that can be handled by the learner. See meta information.
9. `predict_types = c("response", "prob"),` - Sets the possible prediction types as response (deterministic) and prob (probabilistic). See meta information.
10. `properties = c("importance", "missings", "multiclass", "selected_features", "twoclass", "weights")` - Sets the properties that are handled by the learner, by including `"importance"` a public method called `importance` will be created that must be manually filled. See meta information.
11. `references = TRUE` - Tells the template to add a "references" tag that must be filled manually.
12. `gh_name = "RaphaelS1"` - Fills the "author" tag with my GitHub handle, this is required as it identifies the maintainer of the learner.

The sections below demonstrate what happens after the function has been run and the files that are created.

7.1.3 learner_package_type_key.R

The first script to complete after running `create_learner` is the file with the form `learner_package_type_key.R`, in our case this will actually be `learner_rpart_classif_rpart.key`. **This name must not be changed** as triggering automated tests rely on a strict naming scheme. For our example, the resulting script looks like this:

```
#' @title Classification Decision Tree Learner
#' @author RaphaelS1
#' @name mlr_learners_classif.rpart
#'
#' @template class_learner
#' @templateVar id classif.rpart
#' @templateVar caller rpart
#'
#' @references
#' <FIXME - DELETE THIS AND LINE ABOVE IF OMITTED>
#'
#' @template seealso_learner
#' @template example
#' @export
LearnerClassifRpart = R6Class("LearnerClassifRpart",
```

```
inherit = LearnerClassif,

public = list(
  #' @description
  #' Creates a new instance of this [R6][R6::R6Class] class.
  initialize = function() {
    # FIXME - MANUALLY ADD PARAM_SET BELOW AND THEN DELETE THIS LINE
    ps = <param_set>

    # FIXME - MANUALLY UPDATE PARAM VALUES BELOW IF APPLICABLE THEN DELETE THIS LINE.
    # OTHERWISE DELETE THIS AND LINE BELOW.
    ps$values = list(<param_vals>)

    super$initialize(
      id = "classif.rpart",
      packages = "rpart",
      feature_types = c("logical", "integer", "numeric", "factor", "ordered"),
      predict_types = c("response", "prob"),
      param_set = ps,
      properties = c("importance", "missings", "multiclass", "selected_features", "twoclass", "weigh
      man = "mlr3extralearners::mlr_learners_classif.rpart"
    )
  },

  # FIXME - ADD IMPORTANCE METHOD HERE AND DELETE THIS LINE.
  # <See LearnerRegrRandomForest for an example>
  #' @description
  #' The importance scores are extracted from the slot <FIXME>.
  #' @return Named `numeric()`.
  importance = function() { }

),

private = list(

  .train = function(task) {
    pars = self$param_set$get_values(tags = "train")

    # set column names to ensure consistency in fit and predict
    self$state$feature_names = task$feature_names

    # FIXME - <Create objects for the train call
    # <At least "data" and "formula" are required>
    formula = task$formula()
    data = task$data()

    # FIXME - <here is space for some custom adjustments before proceeding to the
    # train call. Check other learners for what can be done here>

    # use the mlr3misc::invoke function (it's similar to do.call())
    mlr3misc::invoke(rpart::rpart,
                     formula = formula,
                     data = data,
                     .args = pars)
  },
```

```
    .predict = function(task) {
      # get parameters with tag "predict"
      pars = self$param_set$get_values(tags = "predict")
      # get newdata
      newdata = task$data(cols = task$feature_names)

      pred = mlr3misc::invoke(predict, self$model, newdata = newdata,
                              type = type, .args = pars)

      # FIXME - ADD PREDICTIONS TO LIST BELOW
      list(...)
    }
  )
)

.extralrns_dict$add("classif.rpart", LearnerClassifRpart)
```

Now we have to do the following (from top to bottom):

1. Fill in the references under "references" and delete the tag that starts "FIXME"
2. Replace `<param_set>` with a parameter set
3. Optionally change default values for parameters in `<param_vals>`
4. As we included "importance" in `properties` we have to add a function to the public method `importance`
5. Fill in the private `.train` method, which takes a (filtered) `Task` and returns a model.
6. Fill in the private `.predict` method, which operates on the model in `self$model` (stored during `$train()`) and a (differently subsetted) `Task` to return a named list of predictions.

7.1.4 Meta-information

In the constructor (`initialize()`) the constructor of the super class (e.g. `LearnerClassif`) is called with meta information about the learner which should be constructed. This includes:

- `id`: The ID of the new learner. Usually consists of `<type>.<algorithm>`, for example: `"classif.rpart"`.
- `packages`: The upstream package name of the implemented learner.
- `param_set`: A set of hyperparameters and their descriptions provided as a `paradox::ParamSet`. For each hyperparameter the appropriate class needs to be chosen. When using the `paradox::ps` shortcut, a short constructor of the form `p_***` can be used:
  - `paradox::ParamLgl` / `paradox::p_lgl` for scalar logical hyperparameters.
  - `paradox::ParamInt` / `paradox::p_int` for scalar integer hyperparameters.
  - `paradox::ParamDbl` / `paradox::p_dbl` for scalar numeric hyperparameters.
  - `paradox::ParamFct` / `paradox::p_fct` for scalar factor hyperparameters (this includes characters).
  - `paradox::ParamUty` / `paradox::p_uty` for everything else (e.g. vector paramters or list parameters).
- `predict_types`: Set of predict types the learner is able to handle. These differ depending on the type of the learner. See `mlr_reflections$learner_predict_types` for the full list of feature types supported by `mlr3`.

  - `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 is able to handle. See `mlr_reflections$task_feature_ty` for feature types supported by `mlr3`.
- `properties`: Set of properties of the learner. See `mlr_reflections$learner_properties` for the full list of feature types supported by `mlr3`. 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 supports internal parallelization in some way. Currently not used, this is an experimental property.
  - `"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 were 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.
- `man`: The roxygen identifier of the learner. This is used within the `$help()` method of the super class to open the help page of the learner.

7.1.5 ParamSet

The `param_set` is the set of hyperparameters used in model training and predicting, this is given as a `paradox::ParamSet`. The set consists of a list of hyperparameters, each has a specific class for the hyperparameter type (see above).

For `classif.rpart` the following replace `<param_set>` above:

```
ps = ParamSet$new(list(
  ParamInt$new(id = "minsplit", default = 20L, lower = 1L, tags = "train"),
  ParamInt$new(id = "minbucket", lower = 1L, tags = "train"),
  ParamDbl$new(id = "cp", default = 0.01, lower = 0, upper = 1, tags = "train"),
  ParamInt$new(id = "maxcompete", default = 4L, lower = 0L, tags = "train"),
  ParamInt$new(id = "maxsurrogate", default = 5L, lower = 0L, tags = "train"),
  ParamInt$new(id = "maxdepth", default = 30L, lower = 1L, upper = 30L, tags = "train"),
  ParamInt$new(id = "usesurrogate", default = 2L, lower = 0L, upper = 2L, tags = "train"),
  ParamInt$new(id = "surrogatestyle", default = 0L, lower = 0L, upper = 1L, tags = "train"),
  ParamInt$new(id = "xval", default = 0L, lower = 0L, tags = "train"),
  ParamLgl$new(id = "keep_model", default = FALSE, tags = "train")
))
ps$values = list(xval = 0L)
```

Within mlr3 packages we suggest to stick to the lengthly definition for consistency, however the `<param_set>` can be written shorter, using the `paradox::ps` shortcut:

```
ps = ps(
  minsplit = p_int(lower = 1L, default = 20L, tags = "train"),
  minbucket = p_int(lower = 1L, tags = "train"),
  cp = p_dbl(lower = 0, upper = 1, default = 0.01, tags = "train"),
  maxcompete = p_int(lower = 0L, default = 4L, tags = "train"),
  maxsurrogate = p_int(lower = 0L, default = 5L, tags = "train"),
  maxdepth = p_int(lower = 1L, upper = 30L, default = 30L, tags = "train"),
  usesurrogate = p_int(lower = 0L, upper = 2L, default = 2L, tags = "train"),
  surrogatestyle = p_int(lower = 0L, upper = 1L, default = 0L, tags = "train"),
  xval = p_int(lower = 0L, default = 0L, tags = "train"),
  keep_model = p_lgl(default = FALSE, tags = "train")
)
```

You should read though the learner documentation to find the full list of available parameters. Just looking at some of these in this example:

- `"cp"` is numeric, has a feasible range of `[0,1]` and defaults to `0.01`. The parameter is used during `"train"`.
- `"xval"` is integer has a lower bound of `0`, a default of `0` and the parameter is used during `"train"`.
- `"keep_model"` is logical with a default of `FALSE` and is used during `"train"`.

In some rare cases you may want to change the default parameter values. You can do this by passing a list to `<param_vals>` in the template script above. You can see we have done this for `"classif.rpart"` where the default for `xval` is changed to `0`. Note that the default in the `ParamSet` is recorded as our changed default (0), and not the original (10). It is strongly recommended to only change the defaults if absolutely required, when this is the case add the following to the learner documentation:

```
#' @section Custom mlr3 defaults:
#' - `<parameter>`:
#'   - Actual default: <value>
#'   - Adjusted default: <value>
#'   - Reason for change: <text>
```

7.1.6 Train function

Let's talk about the `.train()` method. The train function takes a `Task` as input and must return a model.

Let's say we want to translate the following call of `rpart::rpart()` into code that can be used inside the `.train()` method.

First, we write something down that works completely without `mlr3`:

```
data = iris
model = rpart::rpart(Species ~ ., data = iris, xval = 0)
```

We need to pass the formula notation `Species ~ .`, the data and the hyperparameters. To get the hyperparameters, we call `self$param_set$get_values()` and query all parameters that are using during `"train"`.

The dataset is extracted from the `Task`.

Last, we call the upstream function `rpart::rpart()` with the data and pass all hyperparameters via argument `.args` using the `mlr3misc::invoke()` function. The latter is simply an optimized version of `do.call()` that we use within the mlr3 ecosystem.

```
.train = function(task) {
  pars = self$param_set$get_values(tags = "train")
  formula = task$formula()
  data = task$data()
  mlr3misc::invoke(rpart::rpart,
    formula = formula,
    data = data,
    .args = pars)
}
```

7.1.7 Predict function

The internal predict method `.predict()` also operates on a `Task` as well as on the fitted model that has been created by the `train()` call previously and has been stored in `self$model`.

The return value is a `Prediction` object. We proceed analogously to what we did in the previous section. 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 list with the return type requested by the user, this is stored in the `$predict_type` slot of a learner class. Note that the `task` is automatically passed to the prediction object, so all you need to do is return the predictions! Make sure the list names are identical to the task predict types.

The final `.predict()` method is below, we could omit the `pars` line as there are no parameters with the `"predict"` tag but we keep it here to be consistent:

```
.predict = function(task) {
  pars = self$param_set$get_values(tags = "predict")
  # get newdata and ensure same ordering in train and predict
  newdata = task$data(cols = self$state$feature_names)
  if (self$predict_type == "response") {
```

```
    response = mlr3misc::invoke(predict,
      self$model,
      newdata = newdata,
      type = "class",
      .args = pars)

    return(list(response = response))
  } else {
    prob = mlr3misc::invoke(predict,
      self$model,
      newdata = newdata,
      type = "prob",
      .args = pars)
    return(list(prob = prob))
  }
}
```

Note that you cannot rely on the column order of the data returned by `task$data()` as the order of columns may be different from the order of the columns during `$.train`. The `newdata` line ensures the ordering is the same by calling the saved order set in `$.train`, don't delete either of these lines!

7.1.8 Control objects/functions of learners

Some learners rely on a "control" object/function such as `glmnet::glmnet.control()`. Accounting for such depends on how the underlying package works:

- If the package forwards the control parameters via `...` and makes it possible to just pass control parameters as additional parameters directly to the train call, there is no need to distinguish both `"train"` and `"control"` parameters. Both can be tagged with "train" in the ParamSet and just be handed over as shown previously.
- If the control parameters need to be passed via a separate argument, the parameters should also be tagged accordingly in the ParamSet. Afterwards they can be queried via their tag and passed separately to `mlr3misc::invoke()`. See example below.

```
control_pars = mlr3misc::(<package>::<function>,
   self$param_set$get_values(tags = "control"))

train_pars = self$param_set$get_values(tags = "train"))

mlr3misc::invoke([...], .args = train_pars, control = control_pars)
```

7.1.9 Testing the learner

Once your learner is created, you are ready to start testing if it works, there are three types of tests: manual, unit and parameter.

7.1.9.1 Train and Predict

For a bare-bone check you can just try to run a simple `train()` call locally.

```
task = tsk("iris") # assuming a Classif learner
lrn = lrn("classif.rpart")
lrn$train(task)
p = lrn$predict(task)
p$confusion
```

If it runs without erroring, that's a very good start!

7.1.9.2 Autotest

To ensure that your learner is able to handle all kinds of different properties and feature types, we have written an "autotest" that checks the learner for different combinations of such.

The "autotest" setup is generated automatically by `create_learner` and will open after running the function, it will have a name with the form `test_package_type_key.R`, in our case this will actually be `test_rpart_classif_rpart.key`. **This name must not be changed** as triggering automated tests rely on a strict naming scheme. In our example this will create the following script, for which no changes are required to pass (assuming the learner was correctly created):

```
install_learners("classif.rpart")

test_that("autotest", {
  learner = LearnerClassifRpart$new()
  expect_learner(learner)
  result = run_autotest(learner)
  expect_true(result, info = result$error)
})
```

For some learners that have required parameters, it is needed to set some values for required parameters after construction so that the learner can be run in the first place.

You can also exclude some specific test arrangements within the "autotest" via the argument `exclude` in the `run_autotest()` function. Currently the `run_autotest()` function lives in inst/testthat of the `mlr_plkg("mlr3")` and still lacks documentation. This should change in the near future.

To finally run the test suite, call `devtools::test()` or hit `CTRL + Shift + T` if you are using RStudio.

7.1.9.3 Checking Parameters

Some learners have a high number of parameters and it is easy to miss out on some during the creation of a new learner. In addition, if the maintainer of the upstream package changes something with respect to the arguments of the algorithm, the learner is in danger to break. Also, new arguments could be added upstream and manually checking for new additions all the time is tedious.

Therefore we have written a "Parameter Check" that runs for every learner asynchronously to the R CMD Check of the package itself. This "Parameter Check" compares the parameters of the mlr3 ParamSet against all arguments available in the upstream function that is called during `$train()` and `$predict()`. Again the file is automatically created and opened by `create_learner`, this will be named like `test_paramtest_package_type_key.R`, so in our example `test_paramtest_rpart_classif_rpart.R`.

The test comes with an `exclude` argument that should be used to *exclude and explain* why certain arguments of the upstream function are not within the ParamSet of the mlr3learner. This will likely be required for all learners as common arguments like `x`, `target` or `data` are handled by the mlr3 interface and are therefore not included within the ParamSet.

However, there might be more parameters that need to be excluded, for example:

- Type dependent parameters, i.e. parameters that only apply for classification or regression learners.
- Parameters that are actually deprecated by the upstream package and which were therefore not included in the mlr3 ParamSet.

All excluded parameters should have a comment justifying their exclusion.

In our example, the final paramtest script looks like:

```
library("mlr3extralearners")
install_learners("classif.rpart")

test_that("classif.rpart train", {
  learner = lrn("classif.rpart")
  fun = rpart::rpart
  exclude = c(
    "formula", # handled internally
    "model", # handled internally
    "data", # handled internally
    "weights", # handled by task
    "subset", # handled by task
    "na.action", # handled internally
    "method", # handled internally
    "x", # handled internally
    "y", # handled internally
    "parms", # handled internally
    "control", # handled internally
    "cost" # handled internally
  )

  ParamTest = run_paramtest(learner, fun, exclude)
  expect_true(ParamTest, info = paste0(
    "Missing parameters:",
    paste0("- '", ParamTest$missing, "'", collapse = "
")))
})

test_that("classif.rpart predict", {
  learner = lrn("classif.rpart")
  fun = rpart:::predict.rpart
  exclude = c(
```

```
    "object", # handled internally
    "newdata", # handled internally
    "type", # handled internally
    "na.action" # handled internally
  )

  ParamTest = run_paramtest(learner, fun, exclude)
  expect_true(ParamTest, info = paste0(
    "Missing parameters:",
    paste0("- '", ParamTest$missing, "'", collapse = "
")))
})
```

7.1.10 Package Cleaning

Once all tests are passing, run the following functions to ensure that the package remains clean and tidy

1. `devtools::document(roclets = c('rd', 'collate', 'namespace'))`
2. If you haven't done this before run: `remotes::install_github('pat-s/styler@mlr-style')`
3. `styler::style_pkg(style = styler::mlr_style)`
4. `usethis::use_tidy_description()`
5. `lintr::lint_package()`

Please fix any errors indicated by `lintr` before creating a pull request. Finally ensure that all `FIXME` are resolved and deleted in the generated files.

You are now ready to add your learner to the mlr3 ecosystem! Simply open a pull request to https://github.com/mlr-org/mlr3extralearners/pulls with the new learner template and complete the checklist in there. Once the pull request is approved and merged, your learner will automatically appear on the package website.

7.1.11 Thanks and Maintenance

Thank you for contributing to the mlr3 ecosystem!

When you created the learner you would have given your GitHub handle, meaning that you are now listed as the learner author and maintainer. This means that if the learner breaks it is your responsibility to fix the learner - you can view the status of your learner here.

7.1.12 Learner FAQ

Question 1

How to deal with Parameters which have no default?

Answer

If the learner does not work without providing a value, set a reasonable default in `param_set$values`, add tag `"required"` to the parameter and document your default properly.

Question 2

Where to add the package of the upstream package in the DESCRIPTION file?

Add it to the "Suggests" section.

Question 3

How to handle arguments from external "control" functions such as `glmnet::glmnet_control()`?

Answer

See "Control objects/functions of learners".

Question 4

How to document if my learner uses a custom default value that differs to the default of the upstream package?

Answer

If you set a custom default for the mlr3learner that does not cope with the one of the upstream package (think twice if this is really needed!), add this information to the help page of the respective learner.

You can use the following skeleton for this:

```
#' @section Custom mlr3 defaults:
#' - `<parameter>`:
#'   - Actual default: <value>
#'   - Adjusted default: <value>
#'   - Reason for change: <text>
```

Question 5

When should the `"required"` tag be used when defining Params and what is its purpose?

Answer

The `"required"` tag should be used when the following conditions are met:

- The upstream function cannot be run without setting this parameter, i.e. it would throw an error.
- The parameter has no default in the upstream function.

In mlr3 we follow the principle that every learner should be constructable without setting custom parameters. Therefore, if a parameter has no default in the upstream function, a custom value is usually set for this parameter in the mlr3learner (remember to document such changes in the help page of the learner).

Even though this practice ensures that no parameter is unset in an mlr3learner and partially removes the usefulness of the `"required"` tag, the tag is still useful in the following scenario:

If a user sets custom parameters after construction of the learner

```
lrn = lrn("<id>")
lrn$param_set$values = list("<param>" = <value>)
```

Here, all parameters besides the ones set in the list would be unset. See `paradox::ParamSet` for more information. If a parameter is tagged as `"required"` in the ParamSet, the call above would error and prompt the user that required parameters are missing.

Question 6

What is this error when I run `devtools::load_all()`

```
> devtools::load_all(".")
Loading mlr3extralearners
Warning message:
.onUnload failed in unloadNamespace() for 'mlr3extralearners', details:
  call: vapply(hooks, function(x) environment(x)$pkgname, NA_character_)
  error: values must be length 1,
 but FUN(X[[1]]) result is length 0
```

Answer

This is not an error but a warning and you can safely ignore it!

7.2 Adding new Measures

In this section we showcase how to implement a custom performance measure.

A good starting point is writing down the loss function independently of mlr3 (we also did this in the mlr3measures package). Here, we illustrate writing measure by implementing the root of the mean squared error for regression problems:

```
root_mse = function(truth, response) {
  mse = mean((truth - response)^2)
  sqrt(mse)
}

root_mse(c(0, 0.5, 1), c(0.5, 0.5, 0.5))
```

```
## [1] 0.4082
```

In the next step, we embed the `root_mse()` function into a new R6 class inheriting from base classes `MeasureRegr`/`Measure`. For classification measures, use `MeasureClassif`. We keep it simple here and only explain the most important parts of the `Measure` class:

```
MeasureRootMSE = R6::R6Class("MeasureRootMSE",
  inherit = mlr3::MeasureRegr,
  public = list(
    initialize = function() {
      super$initialize(
        # custom id for the measure
        id = "root_mse",

        # additional packages required to calculate this measure
        packages = character(),
```

```
        # properties, see below
        properties = character(),

        # required predict type of the learner
        predict_type = "response",

        # feasible range of values
        range = c(0, Inf),

        # minimize during tuning?
        minimize = TRUE
      )
    }
  ),

  private = list(
    # custom scoring function operating on the prediction object
    .score = function(prediction, ...) {
      root_mse = function(truth, response) {
        mse = mean((truth - response)^2)
        sqrt(mse)
      }

      root_mse(prediction$truth, prediction$response)
    }
  )
)
```

This class can be used as template for most performance measures. If something is missing, you might want to consider having a deeper dive into the following arguments:

- `properties`: If you tag you measure with the property `"requires_task"`, the `Task` is automatically passed to your `.score()` function (don't forget to add the argument `task` in the signature). The same is possible with `"requires_learner"` if you need to operate on the `Learner` and `"requires_train_set"` if you want to access the set of training indices in the score function.
- `aggregator`: This function (defaulting to `mean()`) controls how multiple performance scores, i.e. from different resampling iterations, are aggregated into a single numeric value if `average` is set to micro averaging. This is ignored for macro averaging.
- `predict_sets`: Prediction sets (subset of `("train", "test")`) to operate on. Defaults to the "test" set.

Finally, if you want to use your custom measure just like any other measure shipped with mlr3 and access it via the `mlr_measures` dictionary, you can easily add it:

```
mlr3::mlr_measures$add("root_mse", MeasureRootMSE)
```

Typically it is a good idea to put the measure together with the call to `mlr_measures$add()` in a new R file and just source it in your project.

```
## source("measure_root_mse.R")
msr("root_mse")
```

```
## <MeasureRootMSE:root_mse>
## * Packages: mlr3
## * Range: [0, Inf]
## * Minimize: TRUE
## * Average: macro
## * Parameters: list()
## * Properties: -
## * Predict type: response
```

7.3 Adding new PipeOps

This section showcases how the `mlr3pipelines` package can be extended to include custom `PipeOps`. To run the following examples, we will need a `Task`; we are using the well-known "Iris" task:

```
library("mlr3")
task = tsk("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
```

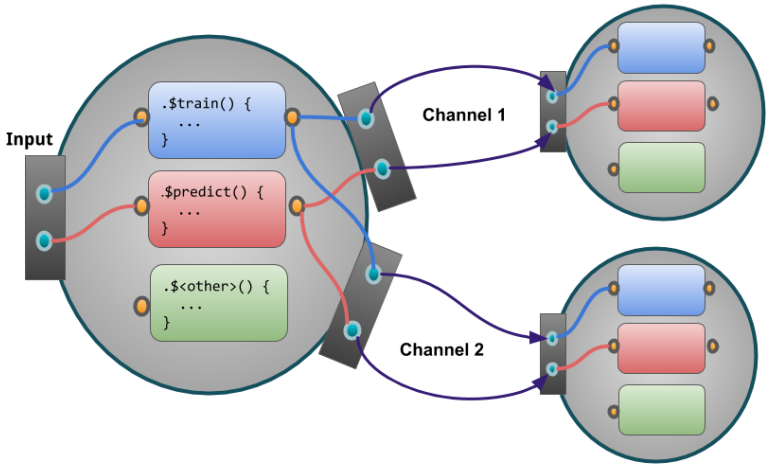
`mlr3pipelines` is fundamentally built around R6. When planning to create custom `PipeOp` objects, it can only help to familiarize yourself with it.

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.

7.3.1 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.



7.3.1.1 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") {
      ....
    },
  ),
  private == list(
    .train = function(inputs) {
      ....
    },

    .predict = function(inputs) {
      ....
    }
  )
)
```

Note, that **private** methods, e.g. `.train` and `.predict` etc are prefixed with a `.`.

7.3.1.2 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
  )
}
```

7.3.1.3 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)
}
```

7.3.1.4 Putting it Together

The whole definition thus becomes

```
PipeOpCopyTwo = R6::R6Class("PipeOpCopyTwo",
  inherit = mlr3pipelines::PipeOp,
  public = list(
    initialize = function(id = "copy.two") {
      super$initialize(id,
        input = data.table::data.table(name = "input", train = "*", predict = "*"),
        output = data.table::data.table(name = c("output1", "output2"),
                                train = "*", predict = "*")
      )
    }
  ),
  private = list(
    .train = function(inputs) {
      self$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:iri
##  $ copy.two.output2:Classes 'TaskClassif', 'TaskSupervised', 'Task', 'R6' <TaskClassif:iri
```

7.3.2 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 private methods .train_task() and .predict_task(), or the methods .train_dt(), .predict_dt(), depending on whether wants to operate on a Task object or on its data as data.tables. In the second case, one can optionally also overload the .select_cols() method, 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()).

7.3.2.1 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 private method .train_task(). We take care that we also set the $state slot to signal that the PipeOp was trained.

The private method .predict_task() 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:

```
PipeOpDropNA = R6::R6Class("PipeOpDropNA",
  inherit = mlr3pipelines::PipeOpTaskPreproc,
  public = list(
    initialize = function(id = "drop.na") {
      super$initialize(id)
    }
  ),

  private = list(
    .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:

```
smalliris = iris[(1:5) * 30, ]
smalliris[1, 1] = NA
smalliris[2, 2] = NA
sitask = as_task_classif(smalliris, target = "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`.

```
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
```

7.3.2.2 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 pass 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 `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`.

```
PipeOpScaleAlways = R6::R6Class("PipeOpScaleAlways",
  inherit = mlr3pipelines::PipeOpTaskPreproc,
  public = list(
    initialize = function(id = "scale.always") {
      super$initialize(id = id)
    }
  ),

  private = list(
    .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
```

7.3.3 Special Case: Preprocessing with Simple Train

It is possible to make even further simplifications for many `PipeOp`s 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 private method `.get_state(task)` which sets the `$state` slot during training, and a private method `.transform(task)`, 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)` methods (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.

7.3.3.1 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) methods.

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() method 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:

```
PipeOpDropConst = R6::R6Class("PipeOpDropConst",
  inherit = mlr3pipelines::PipeOpTaskPreprocSimple,
  public = list(
    initialize = function(id = "drop.const") {
      super$initialize(id = id)
    }
  ),

  private = list(
    .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
##
## $outtaskshell
## Empty data.table (0 rows and 4 cols): Species,Petal.Length,Sepal.Length,Sepal.Width
```

```
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
```

7.3.3.2 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)
    }
  ),

  private = list(
    .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
```

7.3.4 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 tuning chapter or the in-depth `paradox` chapter 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()`.

7.3.4.1 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:

```
PipeOpScale$public_methods$initialize
```

```
## function (id = "scale", param_vals = list())
## .__PipeOpScale__initialize(self = self, private = private, super = super,
##     id = id, param_vals = param_vals)
## <environment: namespace:mlr3pipelines>
```

The user has access to this and can set and get parameters. Types are automatically checked:

```
pss = po("scale")
print(pss$param_set)
```

```
## <ParamSet:scale>
##                id    class lower upper nlevels        default value
## 1:         center ParamLgl    NA    NA       2           TRUE
## 2:          scale ParamLgl    NA    NA       2           TRUE
## 3:         robust ParamLgl    NA    NA       2 <NoDefault[3]> FALSE
## 4: affect_columns ParamUty    NA    NA     Inf  <Selector[1]>
```

```
pss$param_set$values$center = FALSE
print(pss$param_set$values)
```

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

```
pss$param_set$values$scale = "TRUE" # bad input is checked!
```

```
## Error in self$assert(xs): Assertion on 'xs' failed: scale: Must be of type 'logical flag',
```

How `PipeOpScale` handles its parameters can be seen in its `$.train_dt` method: It gets the relevant parameters from its `$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$private_methods$.train_dt
```

```
## function (dt, levels, target)
## .__PipeOpScale__.train_dt(self = self, private = private, super = super,
##     dt = dt, levels = levels, target = target)
## <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
```

7.4 Adding new Tuners

In this section, we show how to implement a custom tuner for mlr3tuning. The main task of a tuner is to iteratively propose new hyperparameter configurations that we want to evaluate for a given task, learner and validation strategy. The second task is to decide which configuration should be returned as a tuning result - usually it is the configuration that led to the best observed performance value. If you want to implement your own tuner, you have to implement an R6-Object that offers an `.optimize` method that implements the iterative proposal and you are free to implement `.assign_result` to differ from the before-mentioned default process of determining the result.

Before you start with the implementation make yourself familiar with the main R6-Objects in bbotk (Black-Box Optimization Toolkit). This package does not only provide basic black box optimization algorithms and but also the objects that represent the optimization problem (`bbotk::OptimInstance`) and the log of all evaluated configurations (`bbotk::Archive`).

There are two ways to implement a new tuner: a) If your new tuner can be applied to any kind of optimization problem it should be implemented as a `bbotk::Optimizer`. Any `bbotk::Optimizer` can be easily transformed to a `mlr3tuning::Tuner`. b) If the new custom tuner is only usable for hyperparameter tuning, for example because it needs to access the task, learner or resampling objects it should be directly implemented in mlr3tuning as a `mlr3tuning::Tuner`.

7.4.1 Adding a new Tuner

This is a summary of steps for adding a new tuner. The fifth step is only required if the new tuner is added via bbotk.

1. Check the tuner does not already exist as a `bbotk::Optimizer` or `mlr3tuning::Tuner` in the GitHub repositories.
2. Use one of the existing optimizers / tuners as a template.
3. Overwrite the `.optimize` private method of the optimizer / tuner.
4. Optionally, overwrite the default `.assign_result` private method.
5. Use the `mlr3tuning::TunerFromOptimizer` class to transform the `bbotk::Optimizer` to a `mlr3tuning::Tuner`.
6. Add unit tests for the tuner and optionally for the optimizer.
7. Open a new pull request for the `mlr3tuning::Tuner` and optionally a second one for the `bbotk::Optimizer`.

7.4.2 Template

If the new custom tuner is implemented via bbotk, use one of the existing optimizer as a template e.g. `bbotk::OptimizerRandomSearch`. There are currently only two tuners that are not based on a `bbotk::Optimizer`: `mlr3hyperband::TunerHyperband` and `mlr3tuning::TunerIrace`. Both are rather complex but you can still use the documentation and class structure as a template. The following steps are identical for optimizers and tuners.

Rewrite the meta information in the documentation and create a new class name. Scientific sources can be added in `R/bibentries.R` which are added under `@source` in the documentation. The example and dictionary sections of the documentation are auto-generated based on the `@templateVar id <tuner_id>`. Change the parameter set of the optimizer / tuner and document them under `@section Parameters`. Do not forget to change `mlr_optimizers$add()` / `mlr_tuners$add()` in the last line which adds the optimizer / tuner to the dictionary.

7.4.3 Optimize method

The `$.optimize()` private method is the main part of the tuner. It takes an instance, proposes new points and calls the `$eval_batch()` method of the instance to evaluate them. Here you can go two ways: Implement the iterative process yourself or call an external optimization function that resides in another package.

7.4.3.1 Writing a custom iteration

Usually, the proposal and evaluation is done in a `repeat`-loop which you have to implement. Please consider the following points:

- You can evaluate one or multiple points per iteration
- You don't have to care about termination, as `$eval_batch()` won't allow more evaluations then allowed by the `bbotk::Terminator`. This implies, that code after the `repeat`-loop will not be executed.
- You don't have to care about keeping track of the evaluations as every evaluation is automatically stored in `inst$archive`.
- If you want to log additional information for each evaluation of the `bbotk::Objective`` in thebbotk::Archiveyou can simply add columns to thedata.tableobject that is passed to$eval_batch()‘.

7.4.3.2 Calling an external optimization function

Optimization functions from external packages usually take an objective function as an argument. In this case, you can pass `inst$objective_function` which internally calls `$eval_batch()`. Check out `OptimizerGenSA` for an example.

7.4.4 Assign result method

The default `$.assign_result()` private method simply obtains the best performing result from the archive. The default method can be overwritten if the new tuner determines the result of the optimization in a different way. The new function must call the `$assign_result()` method of the instance to write the final result to the instance. See `mlr3tuning::TunerIrace` for an implementation of `$.assign_result()`.

7.4.5 Transform optimizer to tuner

This step is only needed if you implement via bbotk. The `mlr3tuning::TunerFromOptimizer` class transforms a `bbotk::Optimizer` to a `mlr3tuning::Tuner`. Just add the `bbotk::Optimizer` to the `optimizer` field. See `mlr3tuning::TunerRandomSearch` for an example.

7.4.6 Add unit tests

The new custom tuner should be thoroughly tested with unit tests. `mlr3tuning::Tuners` can be tested with the `test_tuner()` helper function. If you added the Tuner via a `bbotk::Optimizer`, you should additionally test the `bbotk::Optimizer` with the `test_optimizer()` helper function.

8 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.

Density Estimation

This sub-chapter explains how to conduct (unconditional) density estimation in mlr3. Density estimation is used to estimate the probability density function of a continuous variable. Unconditional density estimation is an unsupervised task so there is no 'value' to predict, instead densities are *estimated*.

This sub-chapter explains how to estimate probability distributions for continuous variables using the `mlr3proba` extension package.

Spatiotemporal Analysis

Spatiotemporal analysis data observations entail reference information about spatial and temporal characteristics. One of the largest issues of spatiotemporal data analysis is the inevitable presence of auto-correlation in the data. Auto-correlation is especially severe in data with marginal spatiotemporal variation. The sub-chapter on Spatiotemporal analysis provides instructions on how to account for spatiotemporal data.

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.

Cluster Analysis

Cluster analysis aims to group data into clusters such that objects that are similar end up in the same cluster.
Fundamentally, clustering and classification are similar. However, clustering is an unsupervised task because observations do not contain true labels while in classification, labels are needed in order to train a model.

This sub-chapter explains how to perform cluster analysis in mlr3 with the help of `mlr3cluster` extension package.

8.1 Survival Analysis

Survival analysis is a sub-field of supervised machine learning in which the aim is to predict the survival distribution of a given individual. Arguably the main feature of survival analysis is that unlike classification and regression, learners are trained on two features:

1. the time until the event takes place
2. the event type: either censoring or death.

At a particular time-point, an individual is either: alive, dead, or censored. Censoring occurs if it is unknown if an individual is alive or dead. For example, say we are interested in patients in hospital and every day it is recorded if they are alive or dead, then after a patient leaves it is unknown if they are alive or dead, hence they are censored. If there was no censoring, then ordinary regression analysis could be used instead. Furthermore, survival data contains solely positive values and therefore needs to be transformed to avoid biases.

Note that survival analysis accounts for both censored and uncensored observations while adjusting respective model parameters.

The package mlr3proba (Sonabend et al. 2021) extends mlr3 with the following objects for survival analysis:

- TaskSurv to define (censored) survival tasks
- LearnerSurv as base class for survival learners
- PredictionSurv as specialized class for Prediction objects
- MeasureSurv as specialized class for performance measures

For a good introduction to survival analysis see *Modelling Survival Data in Medical Research* (Collett 2014).

8.1.1 TaskSurv

Unlike TaskClassif and TaskRegr which have a single 'target' argument, TaskSurv mimics the survival::Surv object and has three to four target arguments (dependent on censoring type). A TaskSurv can be constructed with the function as_task_surv():

```
library("mlr3")
library("mlr3proba")
library("survival")

as_task_surv(survival::bladder2[, -1L], id = "interval_censored",
  time = "start", event = "event", time2 = "stop", type = "interval")
```

```
## <TaskSurv:interval_censored> (178 x 7)
## * Target: start, stop, event
## * Properties: -
## * Features (4):
##   - dbl (2): enum, rx
##   - int (2): number, size
```

```
# type = "right" is default
task = as_task_surv(survival::rats, id = "right_censored",
  time = "time", event = "status", type = "right")

print(task)
```

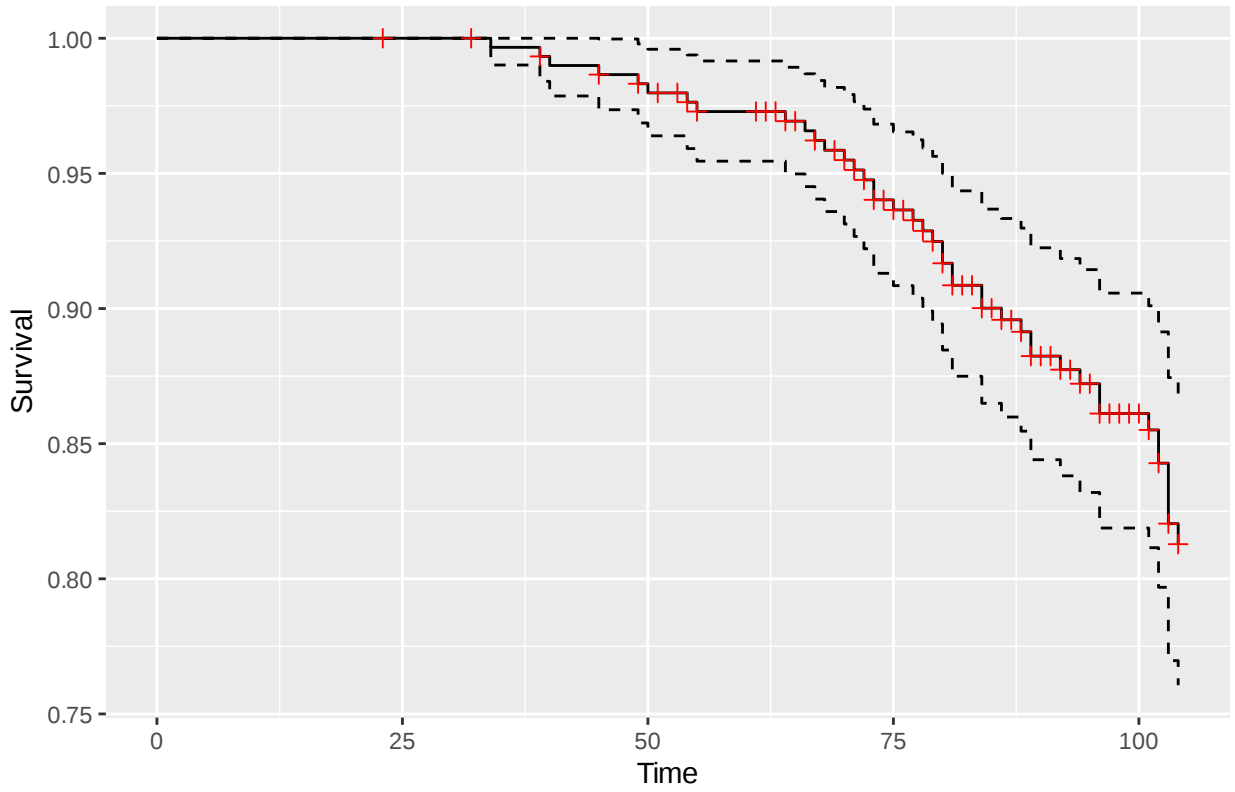
```
## <TaskSurv:right_censored> (300 x 5)
## * Target: time, status
## * Properties: -
## * Features (3):
##   - int (1): litter
##   - dbl (1): rx
##   - chr (1): sex
```

```
# the target column is a survival object:
head(task$truth())
```

```
## [1] 101+  49  104+  91+ 104+ 102+
```

```
# kaplan-meier plot
library("mlr3viz")
autoplot(task)
```

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



8.1.2 Predict Types - crank, lp, and distr

Every `PredictionSurv` object can predict one or more of:

- `lp` - Linear predictor calculated as the fitted coefficients multiplied by the test data.
- `distr` - Predicted survival distribution, either discrete or continuous. Implemented in distr6.
- `crank` - Continuous risk ranking.

`lp` and `crank` can be used with measures of discrimination such as the concordance index. Whilst `lp` is a specific mathematical prediction, `crank` is any continuous ranking that identifies who is more or less likely to experience the event. So far the only implemented learner that only returns a continuous ranking is `surv.svm`. If a `PredictionSurv` returns an `lp` then the `crank` is identical to this. Otherwise `crank` is calculated as the expectation of the predicted survival distribution. Note that for linear proportional hazards models, the ranking (but not necessarily the `crank` score itself) given by `lp` and the expectation of `distr`, is identical.

The example below uses the rats task shipped with mlr3proba.

```
task = tsk("rats")
learn = lrn("surv.coxph")

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

learn$train(task, row_ids = train_set)
prediction = learn$predict(task, row_ids = test_set)

print(prediction)
```

```
## <PredictionSurv> for 60 observations:
##     row_ids time status   crank      lp     distr
##           6  102  FALSE -3.1313 -3.1313 <list[1]>
##           9  104  FALSE -0.2725 -0.2725 <list[1]>
##          12  102  FALSE -3.1197 -3.1197 <list[1]>
## ---
##         277   89   TRUE  0.8912  0.8912 <list[1]>
##         278  104  FALSE  0.2524  0.2524 <list[1]>
##         288  102  FALSE -2.5831 -2.5831 <list[1]>
```

8.1.3 Composition

Finally we take a look at the PipeOps implemented in mlr3proba, which are used for composition of predict types. For example, a predict linear predictor does not have a lot of meaning by itself, but it can be composed into a survival distribution. See mlr3pipelines for full tutorials and details on PipeOps.

```
library("mlr3pipelines")
library("mlr3learners")
# PipeOpDistrCompositor - Train one model with a baseline distribution,
# (Kaplan-Meier or Nelson-Aalen), and another with a predicted linear predictor.
task = tsk("rats")
# remove the factor column for support with glmnet
task$select(c("litter", "rx"))
learner_lp = lrn("surv.glmnet")
learner_distr = lrn("surv.kaplan")
prediction_lp = learner_lp$train(task)$predict(task)
prediction_distr = learner_distr$train(task)$predict(task)
prediction_lp$distr

# Doesn't need training. Base = baseline distribution. ph = Proportional hazards.

pod = po("compose_distr", form = "ph", overwrite = FALSE)
prediction = pod$predict(list(base = prediction_distr, pred = prediction_lp))$output

# Now we have a predicted distr!

prediction$distr
```

```
# This can all be simplified by using the distrcompose pipeline

glm.distr = ppl("distrcompositor", learner = lrn("surv.glmnet"),
  estimator = "kaplan", form = "ph", overwrite = FALSE, graph_learner = TRUE)
glm.distr$train(task)$predict(task)
```

8.1.4 Benchmark Experiment

Finally, we conduct a small benchmark study on the `rats` task using some of the integrated survival learners:

```
library("mlr3learners")

task = tsk("rats")

# some integrated learners
learners = lrns(c("surv.coxph", "surv.kaplan", "surv.ranger"))
print(learners)
```

```
## [[1]]
## <LearnerSurvCoxPH:surv.coxph>
## * Model: -
## * Parameters: list()
## * Packages: mlr3, survival, distr6
## * Predict Type: distr
## * Feature types: logical, integer, numeric, factor
## * Properties: weights
##
## [[2]]
## <LearnerSurvKaplan:surv.kaplan>
## * Model: -
## * Parameters: list()
## * Packages: mlr3, survival, distr6
## * Predict Type: crank
## * Feature types: logical, integer, numeric, character, factor, ordered
## * Properties: missings
##
## [[3]]
## <LearnerSurvRanger:surv.ranger>
## * Model: -
## * Parameters: num.threads=1
## * Packages: mlr3, mlr3learners, ranger
## * Predict Type: distr
## * Feature types: logical, integer, numeric, character, factor, ordered
## * Properties: importance, oob_error, weights
```

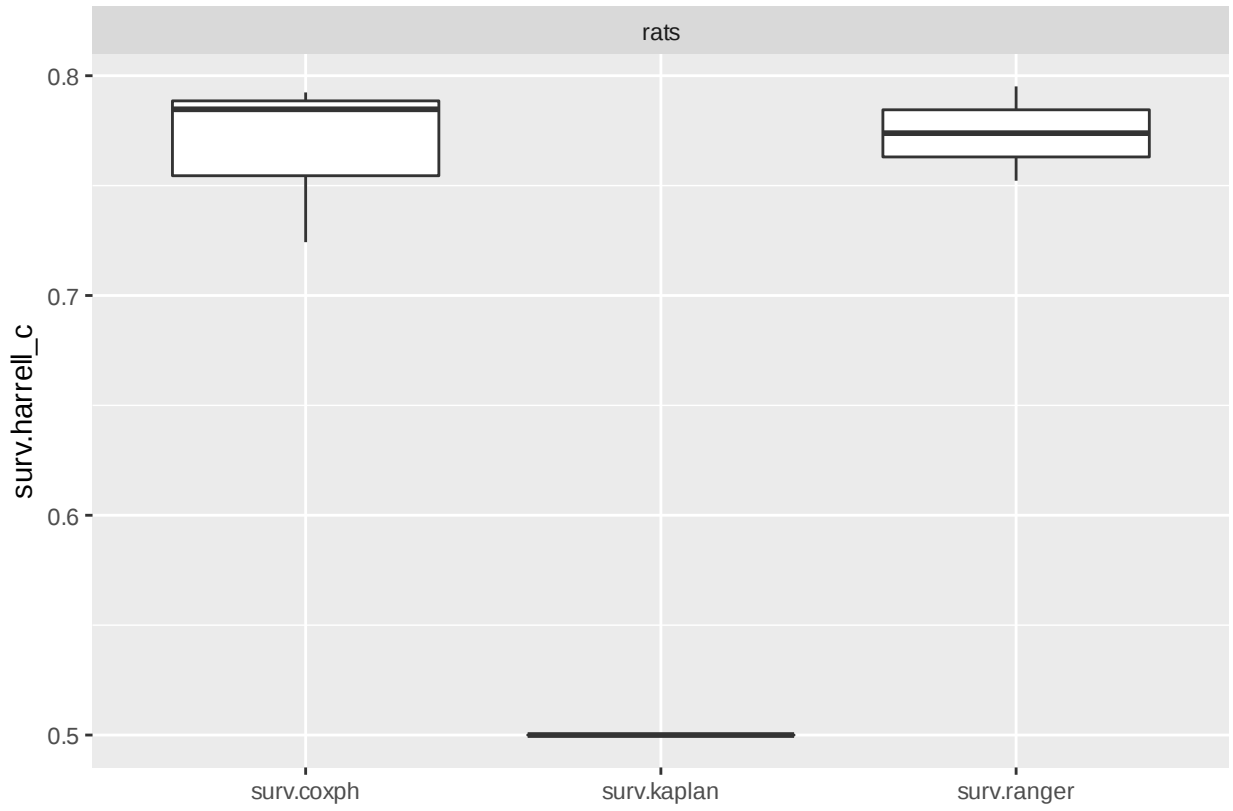
```
# Harrell's C-Index for survival
measure = msr("surv.cindex")
print(measure)
```

```
## <MeasureSurvCindex:surv.harrell_c>
## * Packages: mlr3
## * Range: [0, 1]
## * Minimize: FALSE
## * Average: macro
## * Parameters: list()
## * Properties: -
## * Predict type: crank
## * Return type: Score
```

```
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
## 1:  1 <ResampleResult[22]>    rats  surv.coxph            cv     3
## 2:  2 <ResampleResult[22]>    rats surv.kaplan            cv     3
## 3:  3 <ResampleResult[22]>    rats surv.ranger            cv     3
##    surv.harrell_c
## 1:         0.7671
## 2:         0.5000
## 3:         0.7737
```

```
autoplot(bmr, measure = measure)
```



The experiment indicates that both the Cox PH and the random forest have better discrimination than the Kaplan-Meier baseline estimator, but that the machine learning random forest is not consistently better than the interpretable Cox PH.

8.2 Density Estimation

Density estimation is the learning task to find the unknown distribution from which an i.i.d. data set is generated. We interpret this broadly, with this distribution not necessarily being continuous (so may possess a mass not density). The conditional case, where a distribution is predicted conditional on covariates, is known as 'probabilistic supervised regression', and will be implemented in mlr3proba in the near-future. Unconditional density estimation is viewed as an unsupervised task. For a good overview to density estimation see *Density estimation for statistics and data analysis* (Silverman 1986).

The package mlr3proba extends mlr3 with the following objects for density estimation:

- `TaskDens` to define density tasks
- `LearnerDens` as base class for density estimators
- `PredictionDens` as specialized class for `Prediction` objects
- `MeasureDens` as specialized class for performance measures

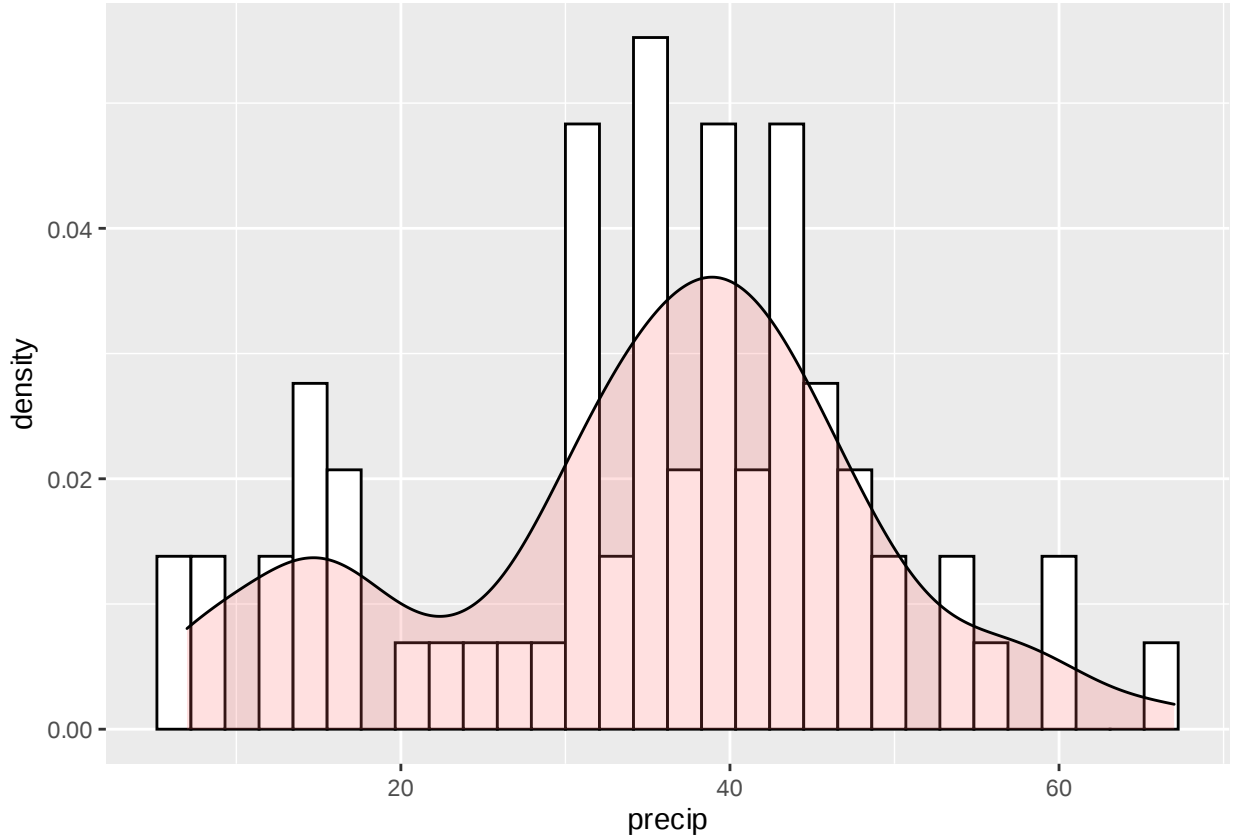
In this example we demonstrate the basic functionality of the package on the `faithful` data from the datasets package. This task ships as pre-defined `TaskDens` with mlr3proba.

```
library("mlr3")
library("mlr3proba")

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

```
## <TaskDens:precip> (70 x 1)
## * Target: -
## * Properties: -
## * Features (1):
##   - dbl (1): precip
```

```
# histogram and density plot
library("mlr3viz")
autoplot(task, type = "overlay")
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Unconditional density estimation is an unsupervised method. Hence, `TaskDens` is an unsupervised task which inherits directly from `Task` unlike `TaskClassif` and `TaskRegr`. However, `TaskDens` still has a `target` argument and a `$truth` field defined by:

- `target` - the name of the variable in the data for which to estimate density
- `$truth` - the values of the `target` column (which is *not* the true density, which is always unknown)

8.2.1 Train and Predict

Density learners have `train` and `predict` methods, though being unsupervised, 'prediction' is actually 'estimation'. In training, a distr6 object is created, see here for full tutorials on how to access the probability density function, `pdf`, cumulative distribution function, `cdf`, and other important fields and methods. The predict method is simply a wrapper around `self$model$pdf` and if available `self$model$cdf`, i.e. evaluates the pdf/cdf at given points. Note that in prediction the points to evaluate the pdf and cdf are determined by the `target` column in the `TaskDens` object used for testing.

```
# create task and learner

task_faithful = TaskDens$new(id = "eruptions", backend = datasets::faithful$eruptions)
learner = lrn("dens.hist")
```

```
# train/test split
train_set = sample(task_faithful$nrow, 0.8 * task_faithful$nrow)
test_set = setdiff(seq_len(task_faithful$nrow), train_set)

# fitting KDE and model inspection
learner$train(task_faithful, row_ids = train_set)
learner$model
```

```
## $distr
## Histogram()
##
## $hist
## $breaks
## [1] 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5
##
## $counts
## [1] 39 33  4  5 26 58 50  2
##
## $density
## [1] 0.35945 0.30415 0.03687 0.04608 0.23963 0.53456 0.46083 0.01843
##
## $mids
## [1] 1.75 2.25 2.75 3.25 3.75 4.25 4.75 5.25
##
## $xname
## [1] "dat"
##
## $equidist
## [1] TRUE
##
## attr(,"class")
## [1] "histogram"
##
## attr(,"class")
## [1] "dens.hist"
```

```
class(learner$model)
```

```
## [1] "dens.hist"
```

```
# make predictions for new data
prediction = learner$predict(task_faithful, row_ids = test_set)
```

Every `PredictionDens` object can estimate:

- `pdf` - probability density function

Some learners can estimate:

- `cdf` - cumulative distribution function

8.2.2 Benchmark Experiment

Finally, we conduct a small benchmark study on the precip task using some of the integrated survival learners:

```
# some integrated learners
learners = lrns(c("dens.hist", "dens.kde"))
print(learners)
```

```
## [[1]]
## <LearnerDensHistogram:dens.hist>
## * Model: -
## * Parameters: list()
## * Packages: mlr3, distr6
## * Predict Type: pdf
## * Feature types: integer, numeric
## * Properties: -
##
## [[2]]
## <LearnerDensKDE:dens.kde>
## * Model: -
## * Parameters: kernel=Epan, bandwidth=silver
## * Packages: mlr3, distr6
## * Predict Type: pdf
## * Feature types: integer, numeric
## * Properties: missings
```

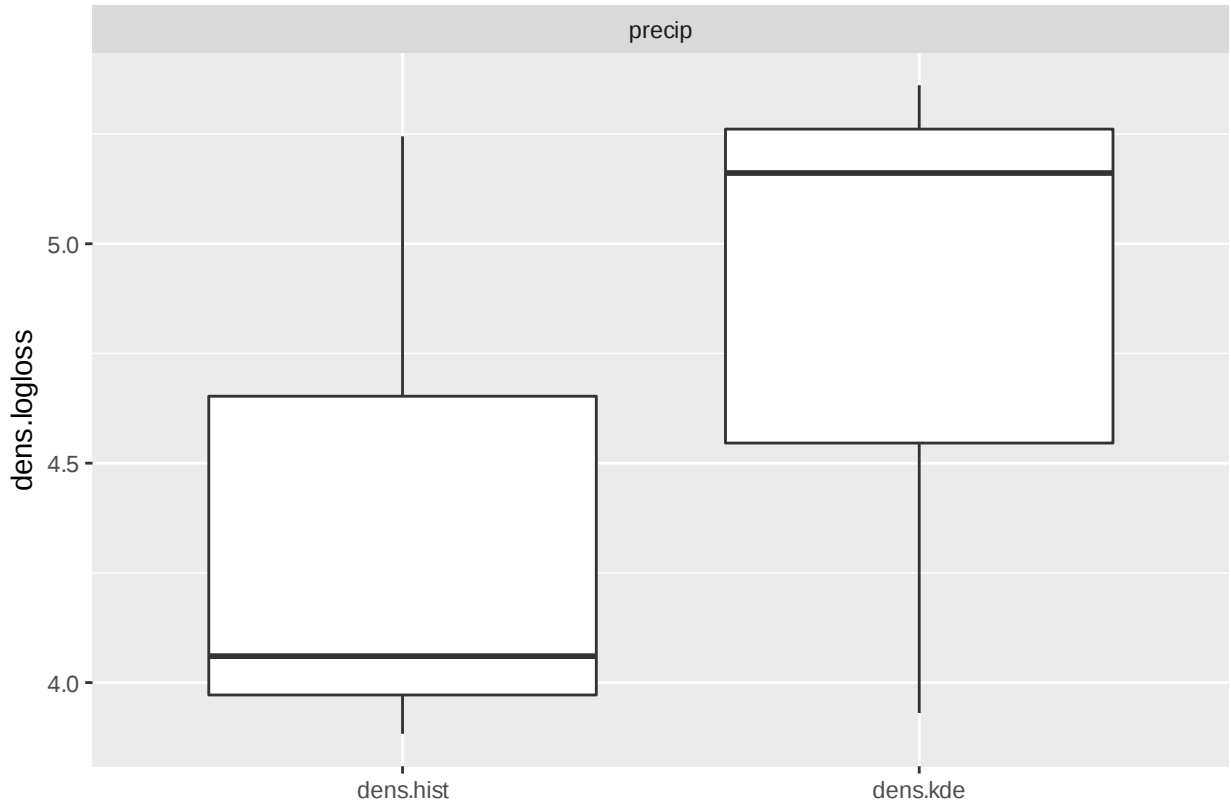
```
# Logloss for probabilistic predictions
measure = msr("dens.logloss")
print(measure)
```

```
## <MeasureDensLogloss:dens.logloss>
## * Packages: mlr3
## * Range: [0, Inf]
## * Minimize: TRUE
## * Average: macro
## * Parameters: list()
## * Properties: -
## * Predict type: pdf
```

```
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 dens.logloss
## 1:  1 <ResampleResult[22]>  precip  dens.hist            cv     3        4.396
## 2:  2 <ResampleResult[22]>  precip   dens.kde            cv     3        4.818
```

```
autoplot(bmr, measure = measure)
```



The results of this experiment show that the sophisticated Penalized Density Estimator does not outperform the baseline Histogram, but that the Kernel Density Estimator has at least consistently `better` (i.e. lower logloss) results.

8.3 Spatiotemporal Analysis

Data observations may entail reference information about spatial or temporal characteristics. Spatial information is stored as coordinates, usually named "x" and "y" or "lat"/"lon". Treating spatiotemporal data using non-spatial data methods can lead to over-optimistic performance estimates. Hence, methods specifically designed to account for the special nature of spatiotemporal data are needed.

In the mlr3 framework, the following packages relate to this field:

- mlr3spatiotemporal (biased-reduced performance estimation)
- mlr3forecasting (time-series support)
- mlr3spatial (spatial prediction support)

The following (sub-)sections introduce the potential pitfalls of spatiotemporal data in machine learning and how to account for it. Note that not all functionality will be covered, and that some of the used packages are still in early lifecycles. If you want to contribute to one of the packages mentioned above, please contact Patrick Schratz.

8.3.1 Creating a spatial Task

To make use of spatial resampling methods, a {mlr3} task that is aware of its spatial characteristic needs to be created. Two child classes exist in {mlr3spatiotempcv} for this purpose:

- `TaskClassifST`
- `TaskRegrST`

To create one of these, one can either pass a `sf` object as the "backend" directly:

```
# create 'sf' object
data_sf = sf::st_as_sf(ecuador, coords = c("x", "y"), crs = 32717)
# create mlr3 task
task = TaskClassifST$new("ecuador_sf",
  backend = data_sf, target = "slides", positive = "TRUE"
)
```

or use a plain `data.frame`. In this case, the constructor of `TaskClassifST` needs a few more arguments:

```
data = mlr3::as_data_backend(ecuador)
task = TaskClassifST$new("ecuador",
  backend = data, target = "slides",
  positive = "TRUE", extra_args = list(coordinate_names = c("x", "y"),
  crs = 32717)
)
```

Now this Task can be used as a normal {mlr3} task in any kind of modeling scenario.

8.3.2 Autocorrelation

Data which includes spatial or temporal information requires special treatment in machine learning (similar to survival, ordinal and other task types listed in the special tasks chapter). In contrast to non-spatial/non-temporal data, observations inherit a natural grouping, either in space or time or in both space and time (Legendre 1993). This grouping causes observations to be autocorrelated, either in space (spatial autocorrelation (SAC)), time (temporal autocorrelation (TAC)) or both space and time (spatiotemporal autocorrelation (STAC)). For simplicity, the acronym STAC is used as a generic term in the following chapter for all the different characteristics introduced above.

What effects does STAC have in statistical/machine learning?

The overarching problem is that STAC violates the assumption that the observations in the train and test datasets are independent (Hastie, Friedman, and Tibshirani 2001). If this assumption

is violated, the reliability of the resulting performance estimates, for example retrieved via cross-validation, is decreased. The magnitude of this decrease is linked to the magnitude of STAC in the dataset, which cannot be determined easily.

One approach to account for the existence of STAC is to use dedicated resampling methods. mlr3spatiotemporal provides access to the most frequently used spatiotemporal resampling methods. The following example showcases how a spatial dataset can be used to retrieve a bias-reduced performance estimate of a learner.

The following examples use the ecuador dataset created by Jannes Muenchow. It contains information on the occurrence of landslides (binary) in the Andes of Southern Ecuador. The landslides were mapped from aerial photos taken in 2000. The dataset is well suited to serve as an example because it it relatively small and of course due to the spatial nature of the observations. Please refer to Muenchow, Brenning, and Richter (2012) for a detailed description of the dataset.

To account for the spatial autocorrelation probably present in the landslide data, we will make use one of the most used spatial partitioning methods, a cluster-based k-means grouping (Brenning 2012), (`"spcv_coords"` in mlr3spatiotemporal). This method performs a clustering in 2D space which contrasts with the commonly used random partitioning for non-spatial data. The grouping has the effect that train and test data are more separated in space as they would be by conducting a random partitioning, thereby reducing the effect of STAC.

By contrast, when using the classical random partitioning approach with spatial data, train and test observations would be located side-by-side across the full study area (a visual example is provided further below). This leads to a high similarity between train and test sets, resulting in "better" but biased performance estimates in every fold of a CV compared to the spatial CV approach. However, these low error rates are mainly caused due to the STAC in the observations and the lack of appropriate partitioning methods and not by the power of the fitted model.

8.3.3 Spatial CV vs. Non-Spatial CV

In the following a spatial and a non-spatial CV will be conducted to showcase the mentioned performance differences.

The performance of a simple classification tree (`"classif.rpart"`) is evaluated on a random partitioning (`"repeated_cv"`) with four folds and two repetitions. The chosen evaluation measure is "classification error" (`"classif.ce"`). The only difference in the spatial setting is that `"repeated_spcv_coords"` is chosen instead of `"repeated_cv"`.

8.3.3.1 Non-Spatial CV

```
library("mlr3")
library("mlr3spatiotempcv")
set.seed(42)

# be less verbose
lgr::get_logger("bbotk")$set_threshold("warn")
lgr::get_logger("mlr3")$set_threshold("warn")

task = tsk("ecuador")
```

```
learner = lrn("classif.rpart", maxdepth = 3, predict_type = "prob")
resampling_nsp = rsmp("repeated_cv", folds = 4, repeats = 2)
rr_nsp = resample(
  task = task, learner = learner,
  resampling = resampling_nsp)

rr_nsp$aggregate(measures = msr("classif.ce"))
```

```
## classif.ce
##     0.3389
```

8.3.3.2 Spatial CV

```
task = tsk("ecuador")

learner = lrn("classif.rpart", maxdepth = 3, predict_type = "prob")
resampling_sp = rsmp("repeated_spcv_coords", folds = 4, repeats = 2)
rr_sp = resample(
  task = task, learner = learner,
  resampling = resampling_sp)

rr_sp$aggregate(measures = msr("classif.ce"))
```

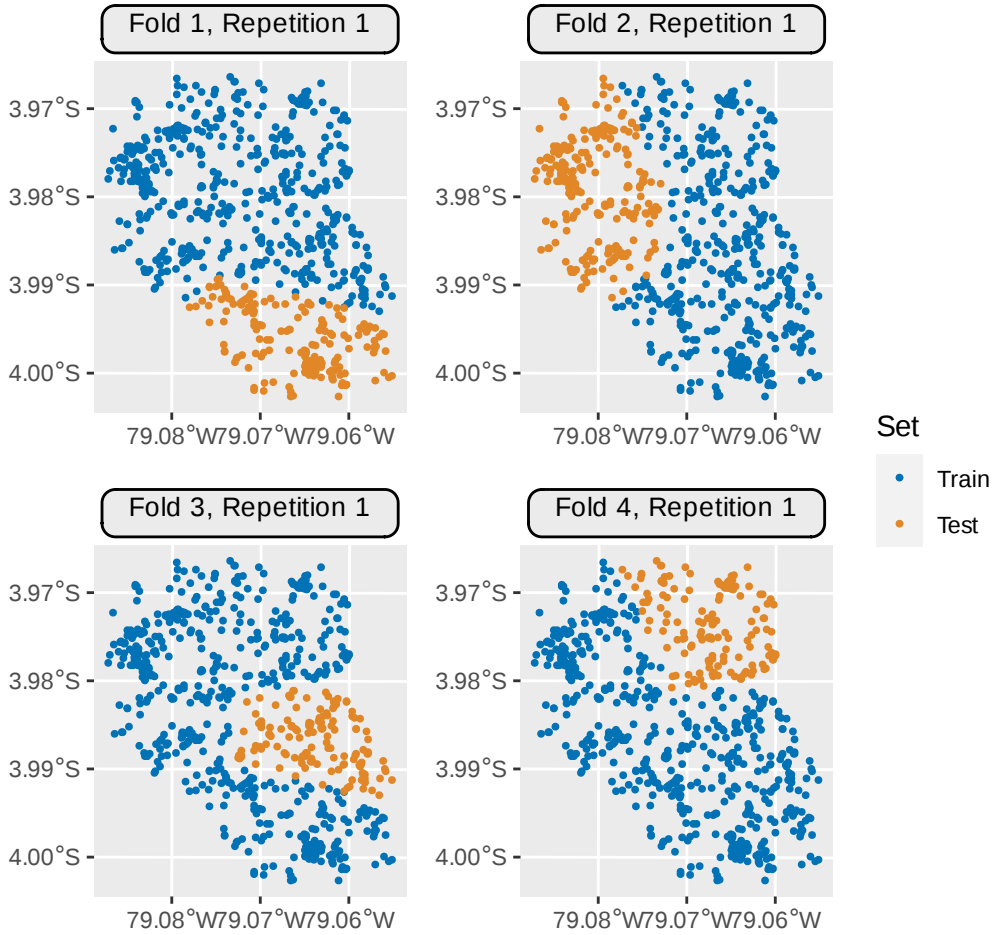
```
## classif.ce
##     0.4125
```

Here, the classification tree learner is around 0.05 percentage points worse when using Spatial Cross-Validation (SpCV) compared to Non-Spatial Cross-Validation (NSpCV). The magnitude of this difference is variable as it depends on the dataset, the magnitude of STAC and the learner itself. For algorithms with a higher tendency of overfitting to the training set, the difference between the two methods will be larger.

8.3.4 Visualization of Spatiotemporal Partitions

Every partitioning method in mlr3spatiotemporal comes with a generic `plot()` method to visualize the created groups. In a 2D space this happens via ggplot2 while for spatiotemporal methods 3D visualizations via plotly are created.

```
autoplot(resampling_sp, task, fold_id = c(1:4), size = 0.7) *
  ggplot2::scale_y_continuous(breaks = seq(-3.97, -4, -0.01)) *
  ggplot2::scale_x_continuous(breaks = seq(-79.06, -79.08, -0.01))
```

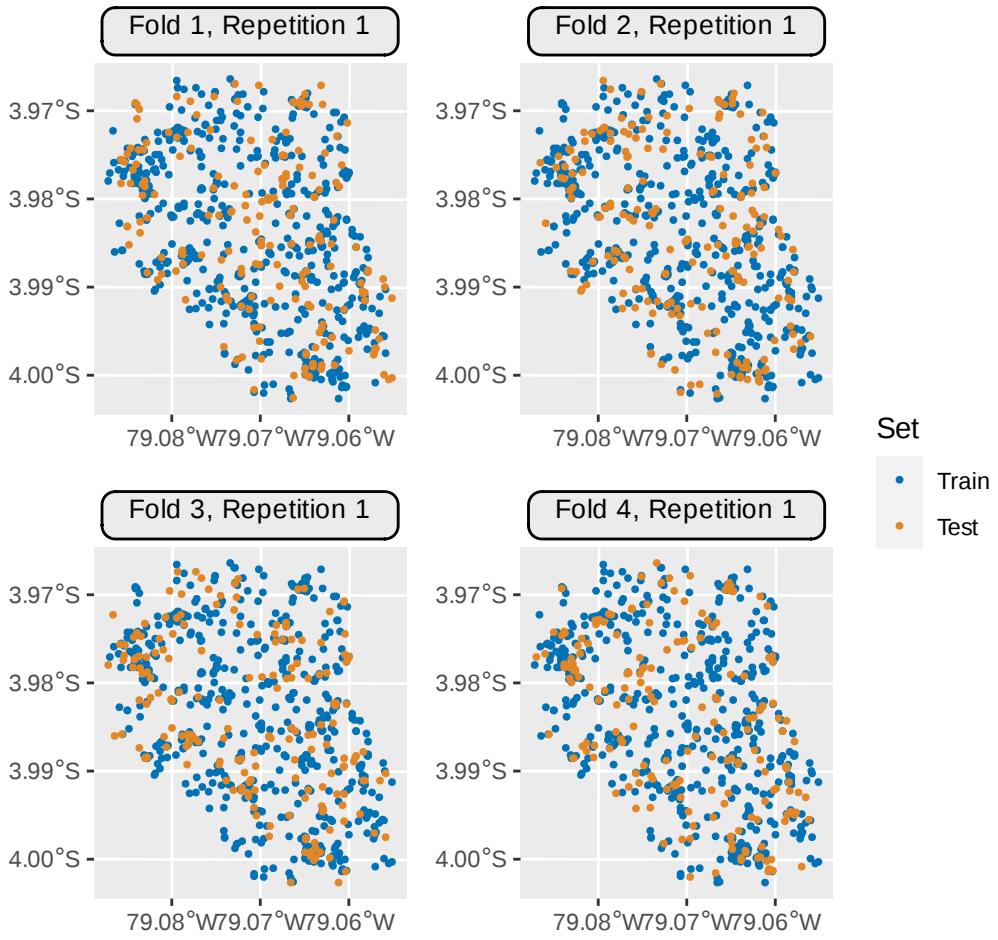


Note that setting the correct CRS for the given data is important which is done during task creation Spatial offsets of up to multiple meters may occur if the wrong CRS is supplied initially.

This example used an already created task via the sugar function `tsk()`. In practice however, one needs to create a spatiotemporal task via `TaskClassifST()`/`TaskRegrST()` and set the `crs` argument.

The spatial grouping of the k-means based approach above contrasts visually ver well compared to the NSpCV (random) partitioning:

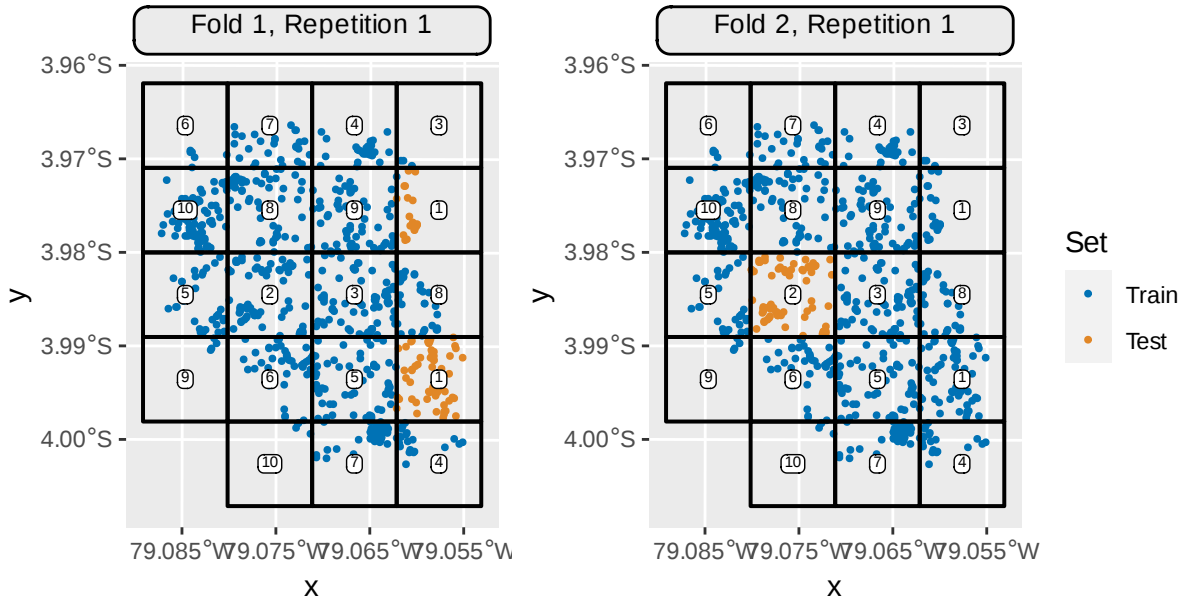
```
autoplot(resampling_nsp, task, fold_id = c(1:4), size = 0.7) *
  ggplot2::scale_y_continuous(breaks = seq(-3.97, -4, -0.01)) *
  ggplot2::scale_x_continuous(breaks = seq(-79.06, -79.08, -0.01))
```



8.3.5 Spatial Block Visualization

The `spcv-block` method makes use of rectangular blocks to divide the study area into equally-sized parts. These blocks can be visualized by their spatial location and fold ID to get a better understanding how these influenced the final partitions.

```
task = tsk("ecuador")
resampling = rsmp("spcv_block", range = 1000L)
resampling$instantiate(task)

## Visualize train/test splits of multiple folds
autoplot(resampling, task, size = 0.7,
  fold_id = c(1, 2), show_blocks = TRUE, show_labels = TRUE) *
  ggplot2::scale_x_continuous(breaks = seq(-79.085, -79.055, 0.01))
```



8.3.6 Choosing a Resampling Method

While the example used the `"spcv_coords"` method, this does not mean that this method is the best or only method suitable for this task. Even though this method is quite popular, it was mainly chosen because of the clear visual grouping differences compared to random partitioning.

In fact, most often multiple spatial partitioning methods can be used for a dataset. It is recommended (required) that users familiarize themselves with each implemented method and decide which method to choose based on the specific characteristics of the dataset. For almost all methods implemented in mlr3spatiotemporal, there is a scientific publication describing the strengths and weaknesses of the respective approach (either linked in the help file of mlr3spatiotemporal or its respective dependency packages).

In the example above, a cross-validation without hyperparameter tuning was shown. If a nested CV is desired, it is recommended to use the same spatial partitioning method for the inner loop (= tuning level). See Schratz et al. (2019) for more details and chapter 11 of Geocomputation with R (Lovelace, Nowosad, and Muenchow 2019)[1].

A list of all implemented methods in mlr3spatiotemporal can be found in the Getting Started vignette of the package.

If you want to learn even more about the field of spatial partitioning, STAC and the problems associated with it, the work of Prof. Hanna Meyer is very much recommended for further reference.

[1]The chapter will soon be rewritten using the **mlr3** and **mlr3spatiotempcv** packages.

8.3.7 Spatial Prediction

Experimental support for spatial prediction with terra, raster, stars and sf objects is available in mlr3spatial.

Until the package is released on CRAN, please see the package vignettes for usage examples.

8.4 Ordinal Analysis

This is work in progress. See mlr3ordinal for the current state of the implementation.

8.5 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.

8.5.1 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 contributors. Open an issue in mlr3fda if you are interested!

8.6 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 contributes. Open an issue in mlr3multioutput if you are interested!

8.7 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).

8.7.1 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  604 152
##     bad    96 148
```

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] -0.0594
```

With an average loan of $20,000, the logistic regression yields the following costs:

```
avg_costs * 20000 * 1000
```

```
## [1] -1188000
```

Instead of losing over $1,000,000, the credit institute now can expect a profit of more than $1,000,000.

8.7.2 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: mlr3
## * Range: [-Inf, Inf]
## * Minimize: TRUE
## * Average: macro
## * Parameters: normalize=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
## 1:  1 <ResampleResult[22]> german_credit     classif.log_reg            cv
## 2:  2 <ResampleResult[22]> german_credit classif.featureless            cv
## 3:  3 <ResampleResult[22]> german_credit      classif.ranger            cv
##    iters classif.costs
## 1:     3      -0.05707
## 2:     3       0.05501
## 3:     3      -0.04215
```

As expected, the featureless learner is performing comparably bad. The logistic regression and the random forest work equally well.

8.7.3 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_ids truth response prob.good prob.bad
##           3  good     good    0.9777  0.02234
##          21  good     good    0.9489  0.05110
##          33  good      bad    0.4467  0.55329
## ---
##         978  good     good    0.7763  0.22366
##         987  good      bad    0.1493  0.85071
##         997  good      bad    0.4411  0.55888
```

```
# 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  604 156
##     bad    96 144
##
## $costs
## classif.costs
##       -0.0554
```

```
with_threshold(p, 0.75)
```

```
## $confusion
##          truth
## response good bad
##     good  472  82
##     bad   228 218
##
## $costs
## classif.costs
##       -0.0832
```

```
with_threshold(p, 1.0)
```

```
## $confusion
##          truth
## response good bad
##     good    0   1
##     bad   700 299
##
## $costs
## classif.costs
##         0.001
```

```
# 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] 0.7926
##
## $objective
## classif.costs
##      -0.08825
```

```
# optimized confusion matrix:
with_threshold(p, best$minimum)$confusion
```

```
##          truth
## response good bad
##     good  435  64
##     bad   265 236
```

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.

8.7.4 Threshold Tuning

Before we start, we have load all required packages:

```
library("mlr3")
library("mlr3pipelines")
library("mlr3tuning")
```

```
## Loading required package: paradox
```

8.7.5 Adjusting thresholds: Two strategies

Currently `mlr3pipelines` offers two main strategies towards adjusting `classification thresholds`. We can either expose the thresholds as a `hyperparameter` of the Learner by using `PipeOpThreshold`. This allows us to tune the `thresholds` via an outside optimizer from `mlr3tuning`.

Alternatively, we can also use `PipeOpTuneThreshold` which automatically tunes the threshold after each learner is fit.

In this blog-post, we'll go through both strategies.

8.7.6 PipeOpThreshold

`PipeOpThreshold` can be put directly after a `Learner`.

A simple example would be:

```
gr = lrn("classif.rpart", predict_type = "prob") %>>% po("threshold")
l = as_learner(gr)
```

Note, that `predict_type` = "prob" is required for `po("threshold")` to have any effect.

The `thresholds` are now exposed as a `hyperparameter` of the `GraphLearner` we created:

```
l$param_set
```

```
## <ParamSetCollection>
##                                id    class lower upper nlevels        default
##  1:             classif.rpart.cp ParamDbl     0     1     Inf           0.01
##  2:     classif.rpart.keep_model ParamLgl    NA    NA       2          FALSE
##  3:     classif.rpart.maxcompete ParamInt     0   Inf     Inf              4
##  4:       classif.rpart.maxdepth ParamInt     1    30      30             30
##  5:   classif.rpart.maxsurrogate ParamInt     0   Inf     Inf              5
##  6:      classif.rpart.minbucket ParamInt     1   Inf     Inf <NoDefault[3]>
##  7:       classif.rpart.minsplit ParamInt     1   Inf     Inf             20
##  8: classif.rpart.surrogatestyle ParamInt     0     1       2              0
##  9:   classif.rpart.usesurrogate ParamInt     0     2       3              2
## 10:           classif.rpart.xval ParamInt     0   Inf     Inf             10
## 11:         threshold.thresholds ParamUty    NA    NA     Inf <NoDefault[3]>
##     value
##  1:
##  2:
##  3:
##  4:
##  5:
##  6:
##  7:
##  8:
##  9:
## 10:     0
## 11:   0.5
```

We can now tune those thresholds from the outside as follows:

Before `tuning`, we have to define which hyperparameters we want to tune over. In this example, we only tune over the `thresholds` parameter of the `threshold` pipeop. you can easily imagine, that we can also jointly tune over additional hyperparameters, i.e. rpart's `cp` parameter.

As the `Task` we aim to optimize for is a binary task, we can simply specify the threshold param:

```
library("paradox")
ps = ps(threshold.thresholds = p_dbl(lower = 0, upper = 1))
```

We now create a `AutoTuner`, which automatically tunes the supplied learner over the `ParamSet` we supplied above.

```
at = AutoTuner$new(
  learner = l,
  resampling = rsmp("cv", folds = 3L),
  measure = msr("classif.ce"),
  search_space = ps,
  terminator = trm("evals", n_evals = 5L),
  tuner = tnr("random_search")
)

at$train(tsk("german_credit"))
```

```
## INFO  [16:06:53.175] [bbotk] Starting to optimize 1 parameter(s) with '<OptimizerRandomSea
## INFO  [16:06:53.222] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:06:53.602] [bbotk] Result of batch 1:
## INFO  [16:06:53.604] [bbotk]  threshold.thresholds classif.ce runtime_learners
## INFO  [16:06:53.604] [bbotk]                0.3405      0.267            0.227
## INFO  [16:06:53.604] [bbotk]                                    uhash
## INFO  [16:06:53.604] [bbotk]  ea5eef08-0a37-421e-bc4c-166a30d07ee8
## INFO  [16:06:53.608] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:06:53.952] [bbotk] Result of batch 2:
## INFO  [16:06:53.954] [bbotk]  threshold.thresholds classif.ce runtime_learners
## INFO  [16:06:53.954] [bbotk]                0.2087      0.298            0.207
## INFO  [16:06:53.954] [bbotk]                                    uhash
## INFO  [16:06:53.954] [bbotk]  8214a76a-9e0c-4e6a-9a0f-1b6400879e20
## INFO  [16:06:53.957] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:06:54.309] [bbotk] Result of batch 3:
## INFO  [16:06:54.310] [bbotk]  threshold.thresholds classif.ce runtime_learners
## INFO  [16:06:54.310] [bbotk]                0.8426       0.35            0.217
## INFO  [16:06:54.310] [bbotk]                                    uhash
## INFO  [16:06:54.310] [bbotk]  fb2dafe1-852a-4f1a-9fac-c04148708ee1
## INFO  [16:06:54.313] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:06:54.651] [bbotk] Result of batch 4:
## INFO  [16:06:54.653] [bbotk]  threshold.thresholds classif.ce runtime_learners
## INFO  [16:06:54.653] [bbotk]                0.8965     0.6162            0.201
## INFO  [16:06:54.653] [bbotk]                                    uhash
## INFO  [16:06:54.653] [bbotk]  53182c7b-7ed6-445b-85eb-01940ef2c79c
## INFO  [16:06:54.656] [bbotk] Evaluating 1 configuration(s)
## INFO  [16:06:54.997] [bbotk] Result of batch 5:
## INFO  [16:06:54.999] [bbotk]  threshold.thresholds classif.ce runtime_learners
## INFO  [16:06:54.999] [bbotk]                0.7662      0.333            0.203
## INFO  [16:06:54.999] [bbotk]                                    uhash
## INFO  [16:06:54.999] [bbotk]  868fa477-b57d-4036-8aaf-b96ce04f21b7
## INFO  [16:06:55.009] [bbotk] Finished optimizing after 5 evaluation(s)
## INFO  [16:06:55.010] [bbotk] Result:
## INFO  [16:06:55.012] [bbotk]  threshold.thresholds learner_param_vals  x_domain classif.ce
## INFO  [16:06:55.012] [bbotk]                0.3405          <list[2]> <list[1]>      0.267
```

Inside the `trafo`, we simply collect all set params into a named vector via `map_dbl` and store it in the `threshold.thresholds` slot expected by the learner.

Again, we create a `AutoTuner`, which automatically tunes the supplied learner over the `ParamSet` we supplied above.

One drawback of this strategy is, that this requires us to fit a new model for each new threshold setting. While setting a threshold and computing performance is relatively cheap, fitting the learner is often more computationally demanding. A better strategy is therefore often to optimize the thresholds separately after each model fit.

8.7.7 PipeOpTunethreshold

`PipeOpTuneThreshold` on the other hand works together with `PipeOpLearnerCV`. It directly optimizes the `cross-validated` predictions made by this `PipeOp`. This is done in order to avoid over-fitting the threshold tuning.

A simple example would be:

```
gr = po("learner_cv", lrn("classif.rpart", predict_type = "prob")) %>>% po("tunethreshold")
l2 = as_learner(gr)
```

Note, that `predict_type` = "prob" is required for `po("tunethreshold")` to work. Additionally, note that this time no `threshold` parameter is exposed, it is automatically tuned internally.

```
l2$param_set
```

```
## <ParamSetCollection>
##                                       id    class lower upper nlevels
##  1:         classif.rpart.resampling.method ParamFct    NA    NA       2
##  2:          classif.rpart.resampling.folds ParamInt     2   Inf     Inf
##  3: classif.rpart.resampling.keep_response ParamLgl    NA    NA       2
##  4:                         classif.rpart.cp ParamDbl     0     1     Inf
##  5:                classif.rpart.keep_model ParamLgl    NA    NA       2
##  6:                classif.rpart.maxcompete ParamInt     0   Inf     Inf
##  7:                  classif.rpart.maxdepth ParamInt     1    30      30
##  8:              classif.rpart.maxsurrogate ParamInt     0   Inf     Inf
##  9:                 classif.rpart.minbucket ParamInt     1   Inf     Inf
## 10:                  classif.rpart.minsplit ParamInt     1   Inf     Inf
## 11:            classif.rpart.surrogatestyle ParamInt     0     1       2
## 12:              classif.rpart.usesurrogate ParamInt     0     2       3
## 13:                      classif.rpart.xval ParamInt     0   Inf     Inf
## 14:            classif.rpart.affect_columns ParamUty    NA    NA     Inf
## 15:                  tunethreshold.measure ParamUty    NA    NA     Inf
## 16:                tunethreshold.optimizer ParamUty    NA    NA     Inf
## 17:                tunethreshold.log_level ParamUty    NA    NA     Inf
##             default      value
##  1: <NoDefault[3]>         cv
##  2: <NoDefault[3]>          3
##  3: <NoDefault[3]>      FALSE
##  4:           0.01
##  5:          FALSE
##  6:              4
##  7:             30
##  8:              5
##  9: <NoDefault[3]>
## 10:             20
## 11:              0
## 12:              2
## 13:             10          0
```

```
## 14:  <Selector[1]>
## 15: <NoDefault[3]> classif.ce
## 16: <NoDefault[3]>      gensa
## 17:  <function[1]>       warn
```

Note that we can set `rsmp("intask")` as a resampling strategy for "learner_cv" in order to evaluate predictions on the "training" data. This is generally not advised, as it might lead to over-fitting on the thresholds but can significantly reduce runtime.

For more information, see the post on Threshold Tuning on the mlr3 gallery.

8.8 Cluster Analysis

Cluster analysis is a type of unsupervised machine learning where the goal is to group data into clusters, where each cluster contains similar observations. The similarity is based on specified metrics that are task and application dependent. Cluster analysis is closely related to classification in a sense that each observation needs to be assigned to a cluster or a class. However, unlike classification problems where each observation is labeled, clustering works on data sets without true labels or class assignments.

The package mlr3cluster extends mlr3 with the following objects for cluster analysis:

- `TaskClust` to define clustering tasks
- `LearnerClust` as base class for clustering learners
- `PredictionClust` as specialized class for `Prediction` objects
- `MeasureClust` as specialized class for performance measures

Since clustering is a type of unsupervised learning, `TaskClust` is slightly different from `TaskRegr` and `TaskClassif` objects. More specifically:

- `truth()` function is missing because observations are not labeled.
- `target` field is empty and will return `character(0)` if accessed anyway.

Additionally, `LearnerClust` provides two extra fields that are absent from supervised learners:

- `assignments` returns cluster assignments for training data. It return `NULL` if accessed before training.
- `save_assignments` is a boolean field that controls whether or not to store training set assignments in a learner.

Finally, `PredictionClust` contains additional two fields:

- `partition` stores cluster partitions.
- `prob` stores cluster probabilities for each observation.

8.8.1 Train and Predict

Clustering learners provide both `train` and `predict` methods. The analysis typically consists of building clusters using all available data. To be consistent with the rest of the library, we refer to this process as training.

Some learners can assign new observations to existing groups with `predict`. However, prediction does not always make sense, as it is the case for hierarchical clustering. In hierarchical clustering, the goal is to build a hierarchy of nested clusters by either splitting large clusters into smaller ones or merging smaller clusters into bigger ones. The final result is a tree or dendrogram which can change if a new data point is added. For consistency with the rest of the ecosystem, mlr3cluster offers `predict` method for hierarchical clusterers but it simply assigns all points to a specified number of clusters by cutting the resulting tree at a corresponding level. Moreover, some learners estimate the probability of each observation belonging to a given cluster. `predict_types` field gives a list of prediction types for each learner.

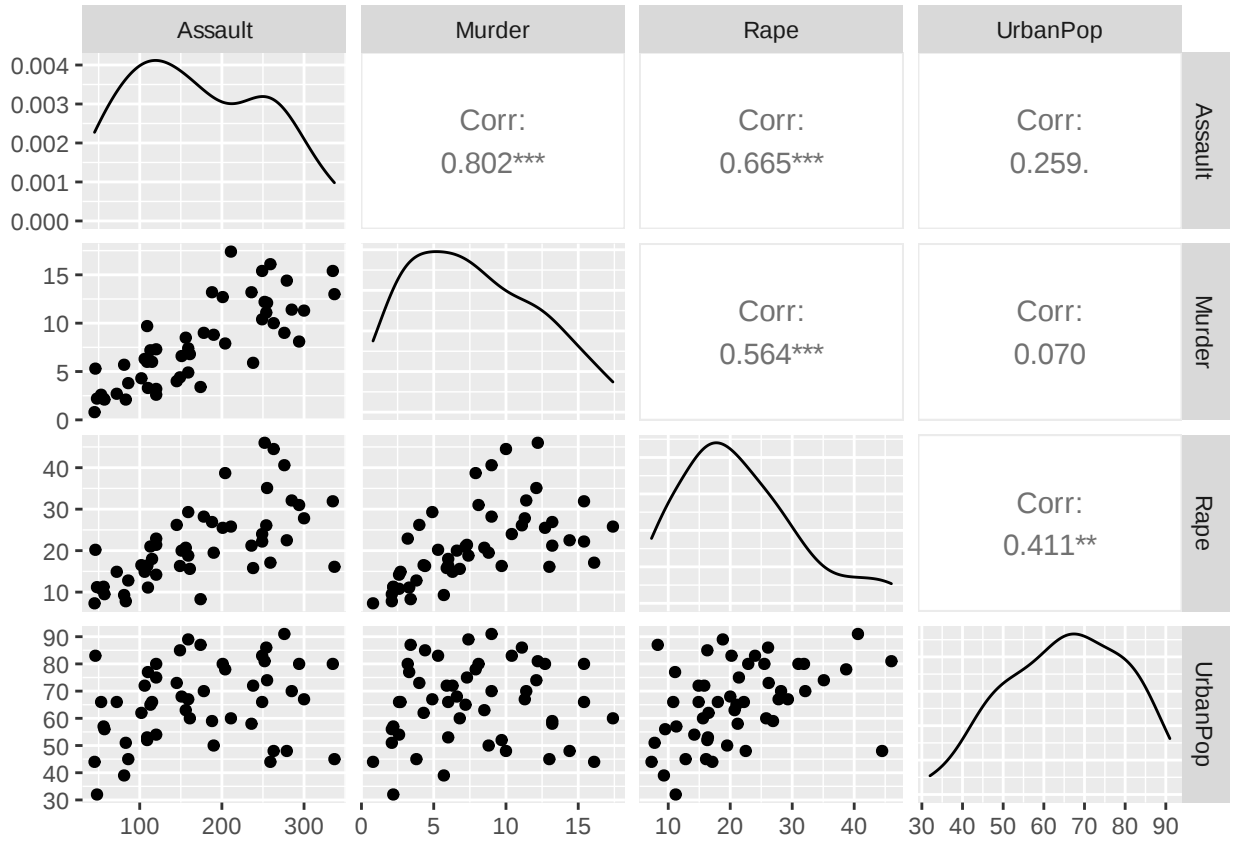
After training, the `model` field stores a learner's model that looks different for each learner depending on the underlying library. `predict` returns a `PredictionClust` object that gives a simplified view of the learned model. If the data given to the `predict` method is the same as the one on which the learner was trained, `predict` simply returns cluster assignments for the "training" observations. On the other hand, if the test set contains new data, `predict` will estimate cluster assignments for that data set. Some learners do not support estimating cluster partitions on new data and will instead return assignments for training data and print a warning message.

In the following example, a `$k$-means learner` is applied on the `US arrest data set`. The class labels are predicted and the contribution of the task features to assignment of the respective class are visualized.

```
library("mlr3")
library("mlr3cluster")
library("mlr3viz")
set.seed(1L)

# create an example task
task = tsk("usarrests")
print(task)
```

```
## <TaskClust:usarrests> (50 x 4)
## * Target: -
## * Properties: -
## * Features (4):
##   - int (2): Assault, UrbanPop
##   - dbl (2): Murder, Rape
```

```
autoplot(task)
```

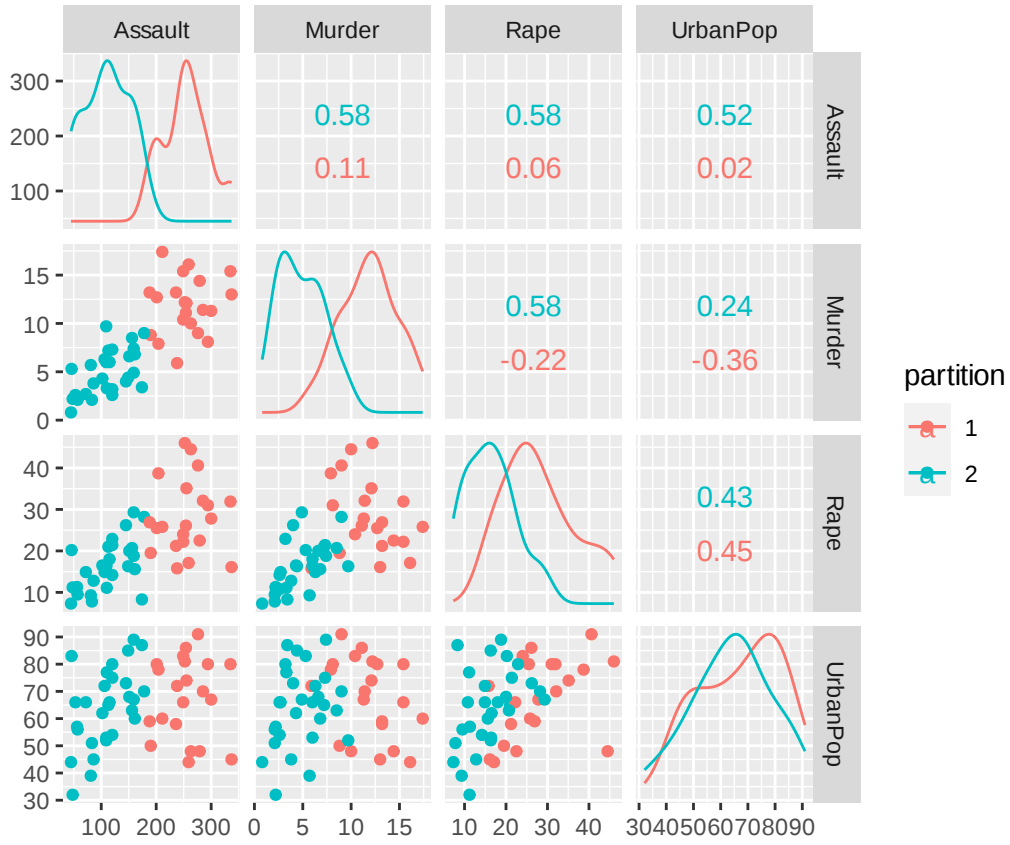



```
# create a k-means learner
learner = lrn("clust.kmeans")

# assigning each observation to one of the two clusters (default in clust.kmeans)
learner$train(task)
learner$model
```

```
## K-means clustering with 2 clusters of sizes 21, 29
##
## Cluster means:
##   Assault Murder  Rape UrbanPop
## 1   255.0 11.857 28.11    67.62
## 2   109.8  4.841 16.25    64.03
##
## Clustering vector:
##  [1] 1 1 1 1 1 1 2 1 1 1 2 2 1 2 2 2 2 1 2 1 2 1 2 1 2 2 2 1 2 2 1 1 1 2 2 2 2 2
## [39] 2 1 2 1 1 2 2 2 2 2 2 2
##
## Within cluster sum of squares by cluster:
## [1] 41637 54762
##  (between_SS / total_SS =  72.9 %)
##
## Available components:
##
## [1] "cluster"      "centers"      "totss"        "withinss"     "tot.withinss"
```

```
## [6] "betweenss"    "size"         "iter"         "ifault"
```

```
# make "predictions" for the same data
prediction = learner$predict(task)
autoplot(prediction, task)
```



8.8.2 Measures

The difference between supervised and unsupervised learning is that there is no ground truth data in unsupervised learning. In a supervised setting, such as classification, we would need to compare our predictions to true labels. Since clustering is an example of unsupervised learning, there are no true labels to which we can compare. However, we can still measure the quality of cluster assignments by quantifying how closely objects within the same cluster are related (cluster cohesion) as well as how distinct different clusters are from each other (cluster separation).

To assess the quality of clustering, there are a few built-in evaluation metrics available. One of them is `within sum of squares (WSS)` which calculates the sum of squared differences between observations and centroids. WSS is useful because it quantifies cluster cohesion. The range of this measure is $[0, \infty)$ where a smaller value means that clusters are more compact.

Another measure is `silhouette quality index` that quantifies how well each point belongs to its assigned cluster versus neighboring cluster. Silhouette values are in $[-1, 1]$ range.

Points with silhouette closer to:

- 1 are well clustered
- 0 lie between two clusters
- -1 likely placed in the wrong cluster

The following is an example of conducting a benchmark experiment with various learners on iris data set without target variable and assessing the quality of each learner with both within sum of squares and silhouette measures.

```
design = benchmark_grid(
  tasks = TaskClust$new("iris", iris[-5]),
  learners = list(
    lrn("clust.kmeans", centers = 3L),
    lrn("clust.pam", k = 2L),
    lrn("clust.cmeans", centers = 3L)),
  resamplings = rsmp("insample"))
print(design)
```

```
##               task                   learner               resampling
## 1: <TaskClust[45]> <LearnerClustKMeans[37]> <ResamplingInsample[19]>
## 2: <TaskClust[45]>    <LearnerClustPAM[37]> <ResamplingInsample[19]>
## 3: <TaskClust[45]> <LearnerClustCMeans[37]> <ResamplingInsample[19]>
```

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

# define measure
measures = list(msr("clust.wss"), msr("clust.silhouette"))
bmr$aggregate(measures)
```

```
##    nr      resample_result task_id   learner_id resampling_id iters clust.wss
## 1:  1 <ResampleResult[22]>    iris clust.kmeans      insample     1     78.85
## 2:  2 <ResampleResult[22]>    iris    clust.pam      insample     1    153.33
## 3:  3 <ResampleResult[22]>    iris clust.cmeans      insample     1     79.03
##    clust.silhouette
## 1:           0.5555
## 2:           0.7158
## 3:           0.5493
```

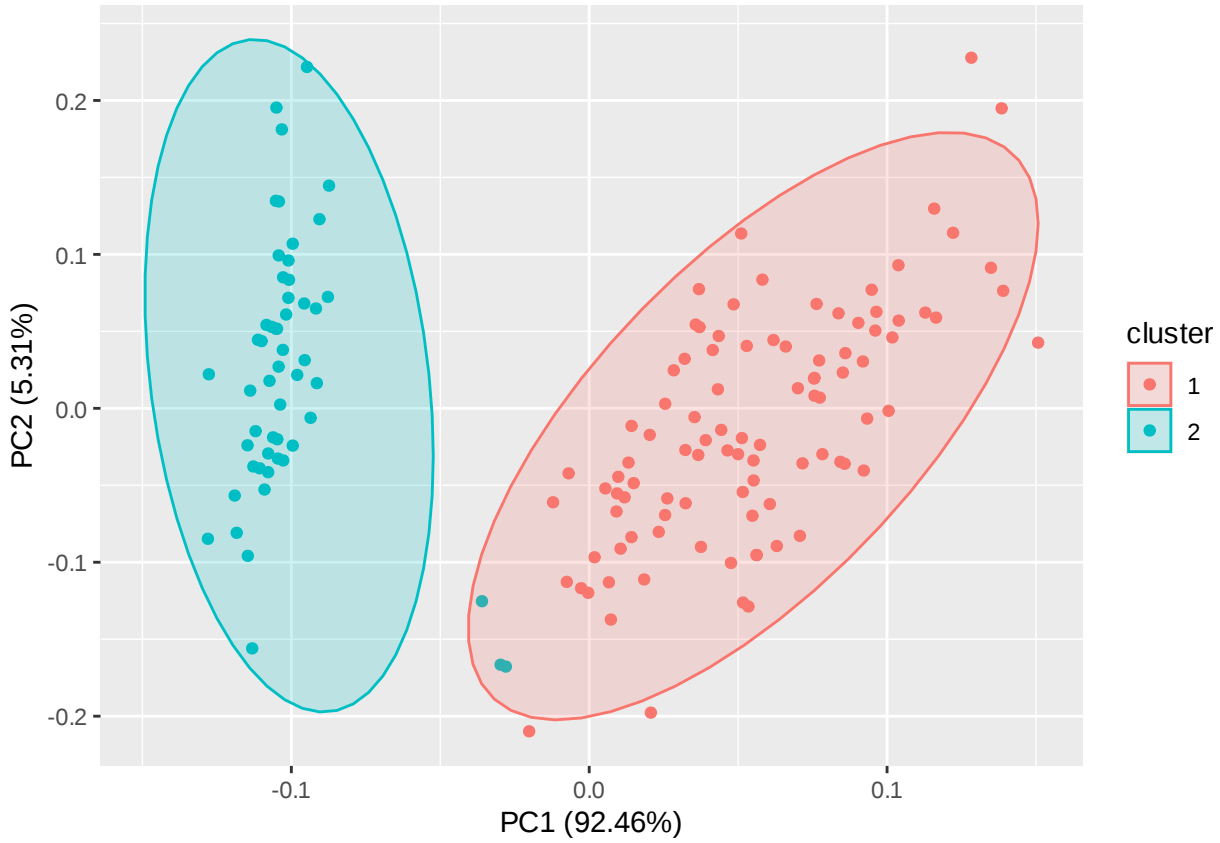
The experiment shows that using k-means algorithm with three centers produces a better within sum of squares score than any other learner considered. However, pam (partitioning around medoids) learner with two clusters performs the best when considering silhouette measure which takes into the account both cluster cohesion and separation.

8.8.3 Visualization

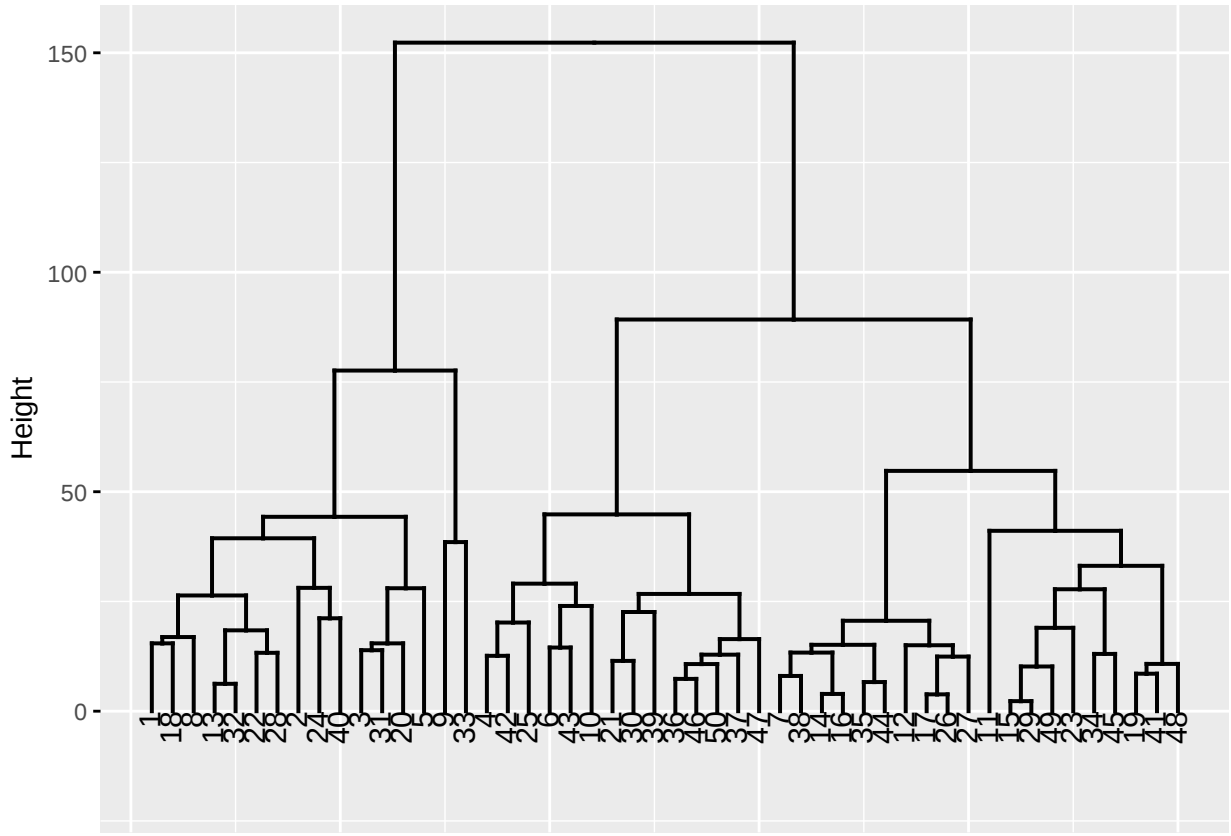
Cluster analysis in mlr3 is integrated with mlr3viz which provides a number of useful plots. Some of those plots are shown below.

```
task = TaskClust$new("iris", iris[-5])
learner = lrn("clust.kmeans")
learner$train(task)
prediction = learner$predict(task)

# performing PCA on task and showing assignments
autoplot(prediction, task, type = "pca")
```

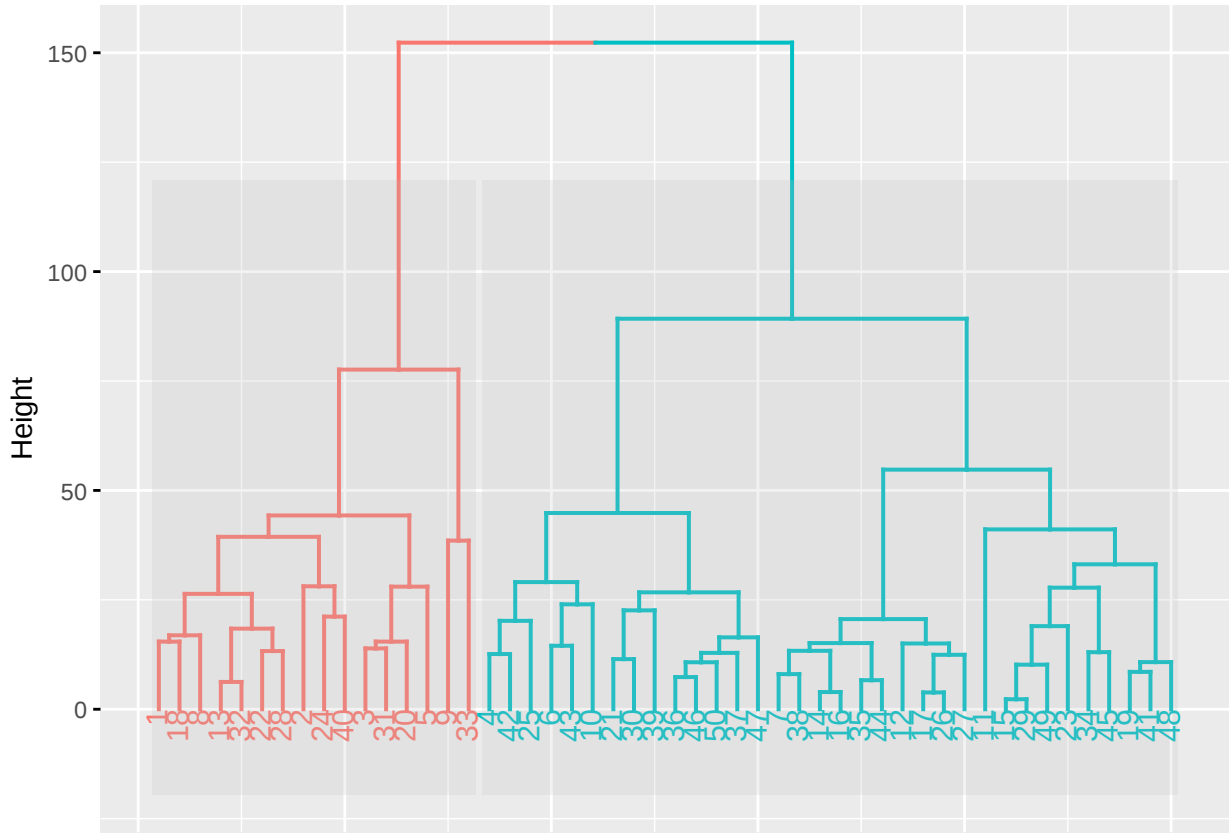

```
# same as above but with probability ellipse that assumes normal distribution
autoplot(prediction, task, type = "pca", frame = TRUE, frame.type = "norm")
```




```
task = tsk("usarrests")
learner = lrn("clust.agnes")
learner$train(task)

# dendrogram for hierarchical clustering
autoplot(learner)
```

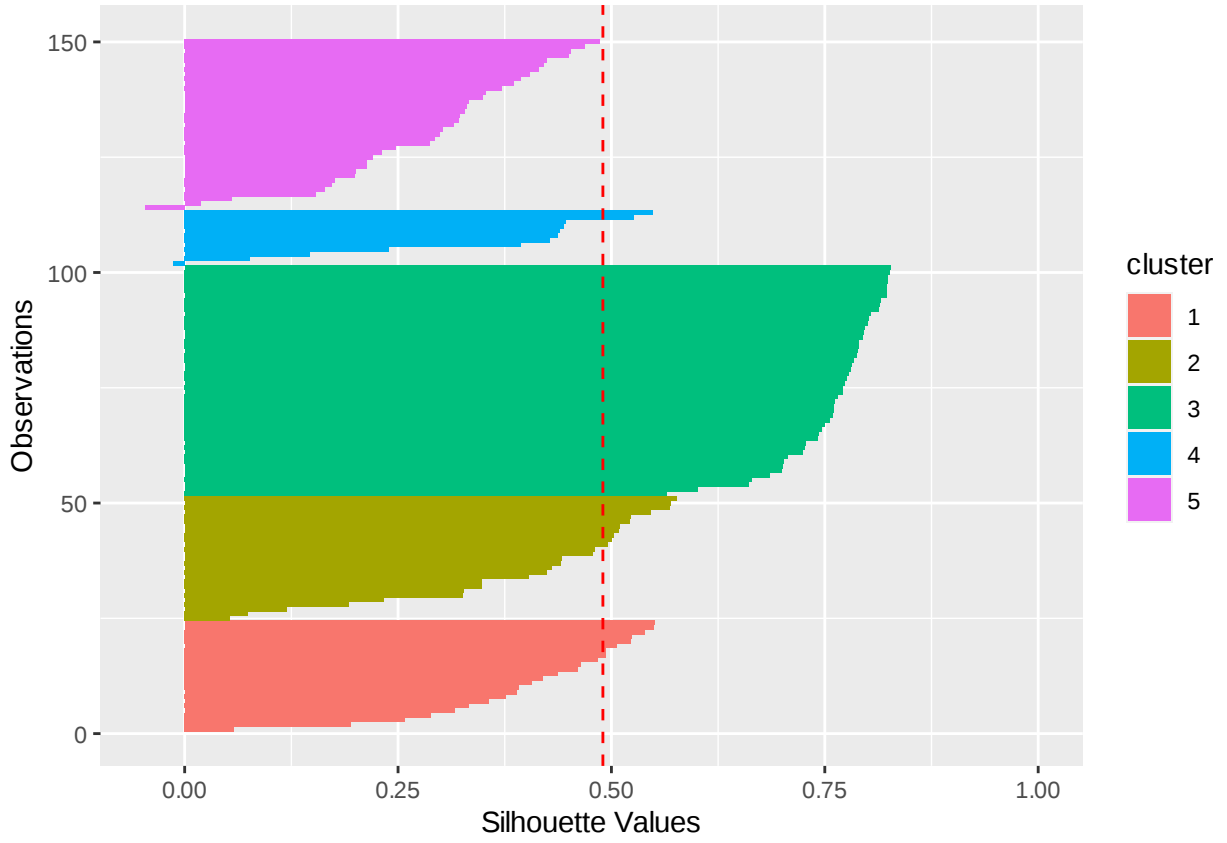



```
# advanced dendrogram options from `factoextra::fviz_dend`
autoplot(learner,
  k = learner$param_set$values$k, rect_fill = TRUE,
  rect = TRUE)
```



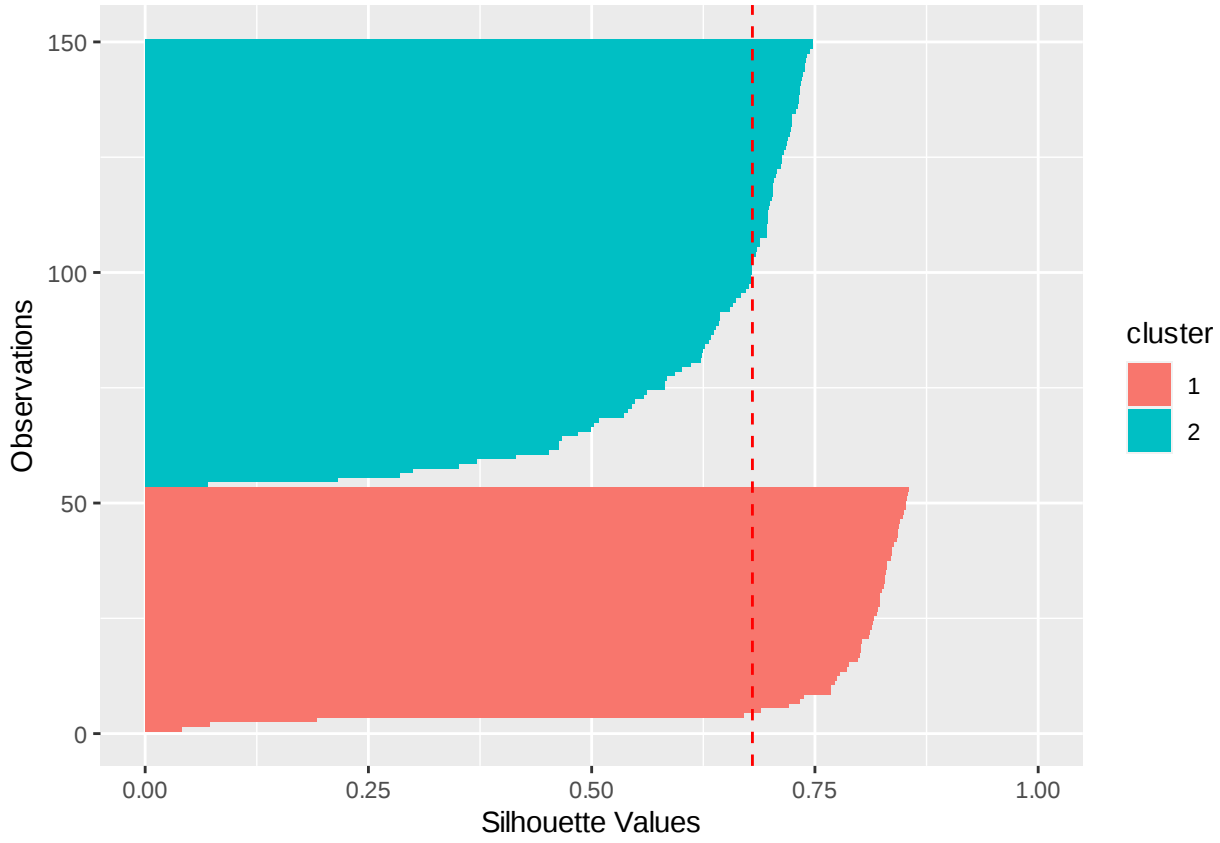
Silhouette plots can help to visually assess the quality of the analysis and help choose a number of clusters for a given data set. The red dotted line shows the mean silhouette value and each bar represents a data point. If most points in each cluster have an index around or higher than mean silhouette, the number of clusters is chosen well.

```
# silhouette plot allows to visually inspect the quality of clustering
task = TaskClust$new("iris", iris[-5])
learner = lrn("clust.kmeans")
learner$param_set$values = list(centers = 5L)
learner$train(task)
prediction = learner$predict(task)
autoplot(prediction, task, type = "sil")
```



The plot shows that all points in cluster 5 and almost all points in clusters 4, 2 and 1 are below average silhouette index. This means that a lot of observations lie either on the border of clusters or are likely assigned to the wrong cluster.

```
learner = lrn("clust.kmeans")
learner$param_set$values = list(centers = 2L)
learner$train(task)
prediction = learner$predict(task)
autoplot(prediction, task, type = "sil")
```



Setting the number of centers to two improves both average silhouette score as well as overall quality of clustering because almost all points in cluster 1 are higher than and a lot of points in cluster 2 are close to mean silhouette. Hence, having two centers might be a better choice for the number of clusters.

9 Model Interpretation

In principle, all generic frameworks for model interpretation are applicable on the models fitted with `mlr3` by just extracting the fitted models from the `Learner` objects.

However, two of the most popular frameworks,

- iml in Subsection 9.1,
- DALEX in Subsection 9.2, and

additionally come with some convenience for `mlr3`.

9.1 IML

Author: Shawn Storm

iml is an R package that interprets the behavior and explains predictions of machine learning models. The functions provided in the iml package are model-agnostic which gives the flexibility to use any machine learning model.

This chapter provides examples of how to use `iml` with `mlr3`. For more information refer to the IML github and the IML book

9.1.1 Penguin Task

To understand what `iml` can offer, we start off with a thorough example. The goal of this example is to figure out the species of penguins given a set of features. The `palmerpenguins::penguins` data set will be used which is an alternative to the `iris` data set. The `penguins` data sets contains 8 variables of 344 penguins:

```
data("penguins", package = "palmerpenguins")
str(penguins)
```

```
## tibble [344 x 8] (S3: tbl_df/tbl/data.frame)
##  $ species          : Factor w/ 3 levels "Adelie","Chinstrap",..: 1 1 1 1 1 1 1 1 1 ...
##  $ island           : Factor w/ 3 levels "Biscoe","Dream",..: 3 3 3 3 3 3 3 3 3 ...
##  $ bill_length_mm   : num [1:344] 39.1 39.5 40.3 NA 36.7 39.3 38.9 39.2 34.1 42 ...
##  $ bill_depth_mm    : num [1:344] 18.7 17.4 18 NA 19.3 20.6 17.8 19.6 18.1 20.2 ...
##  $ flipper_length_mm: int [1:344] 181 186 195 NA 193 190 181 195 193 190 ...
##  $ body_mass_g      : int [1:344] 3750 3800 3250 NA 3450 3650 3625 4675 3475 4250 ...
##  $ sex              : Factor w/ 2 levels "female","male": 2 1 1 NA 1 2 1 2 NA NA ...
##  $ year             : int [1:344] 2007 2007 2007 2007 2007 2007 2007 2007 2007 2007 ...
```

To get started run:

```
library("iml")
library("mlr3")
library("mlr3learners")
set.seed(1)
```

```
penguins = na.omit(penguins)
task_peng = as_task_classif(penguins, target = "species")
```

`penguins = na.omit(penguins)` is to omit the 11 cases with missing values. If not omitted, there will be an error when running the learner from the data points that have N/A for some features.

```
learner = lrn("classif.ranger")
learner$predict_type = "prob"
learner$train(task_peng)
learner$model
```

```
## Ranger result
##
## Call:
##  ranger::ranger(dependent.variable.name = task$target_names, data = task$data(),      prob
##
## Type:                             Probability estimation
## Number of trees:                  500
## Sample size:                      333
## Number of independent variables:  7
## Mtry:                             2
## Target node size:                 10
## Variable importance mode:         none
## Splitrule:                        gini
## OOB prediction error (Brier s.):  0.0179
```

```
x = penguins[which(names(penguins) != "species")]
model = Predictor$new(learner, data = x, y = penguins$species)
```

As explained in Section 2.3, specific learners can be queried with `mlr_learners`. In Section 2.5 it is recommended for some classifiers to use the `predict_type` as `prob` instead of directly predicting a label. This is what is done in this example. `penguins[which(names(penguins) != "species")]` is the data of all the features and `y` will be the penguins`species`. `learner$train(task_peng)` trains the model and `learner$model` stores the model from the training command. `Predictor` holds the machine learning model and the data. All interpretation methods in `iml` need the machine learning model and the data to be wrapped in the `Predictor` object.

Next is the core functionality of `iml`. In this example three separate interpretation methods will be used: FeatureEffects, FeatureImp and Shapley

- `FeatureEffects` computes the effects for all given features on the model prediction. Different methods are implemented: Accumulated Local Effect (ALE) plots, Partial Dependence Plots (PDPs) and Individual Conditional Expectation (ICE) curves.

- `Shapley` computes feature contributions for single predictions with the Shapley value – an approach from cooperative game theory (Shapley Value).
- `FeatureImp` computes the importance of features by calculating the increase in the model's prediction error after permuting the feature (more here).

9.1.2 FeatureEffects

In addition to the commands above the following two need to be ran:

```
num_features = c("bill_length_mm", "bill_depth_mm", "flipper_length_mm", "body_mass_g", "year")
effect = FeatureEffects$new(model)
plot(effect, features = num_features)
```

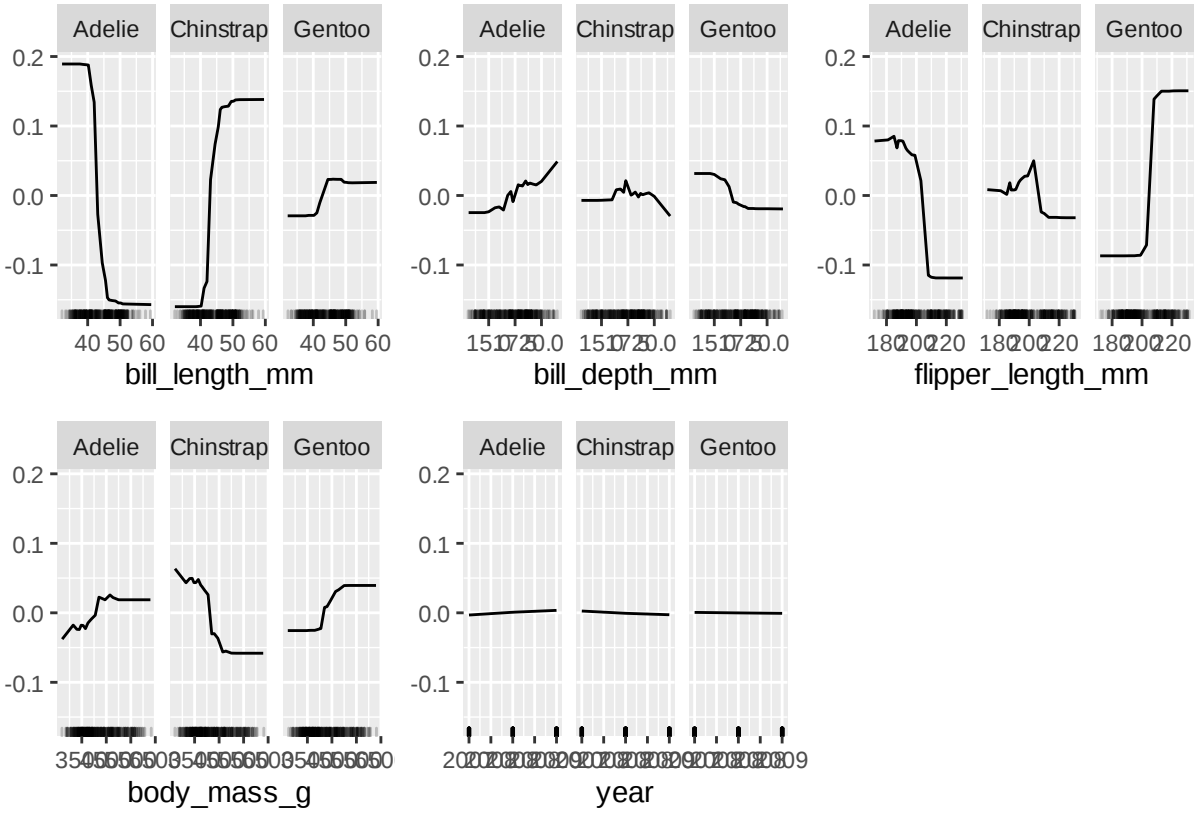


Figure 9.1: Plot of the results from FeatureEffects. FeatureEffects computes and plots feature effects of prediction models

`effect` stores the object from the `FeatureEffect` computation and the results can then be plotted. In this example, all of the features provided by the `penguins` data set were used.

All features except for `year` provide meaningful interpretable information. It should be clear why `year` doesn't provide anything of significance. `bill_length_mm` shows for example that when the bill length is smaller than roughly 40mm, there is a high chance that the penguin is an Adelie.

9.1.3 Shapley

```
x = penguins[which(names(penguins) != "species")]
model = Predictor$new(learner, data = penguins, y = "species")
x.interest = data.frame(penguins[1, ])
shapley = Shapley$new(model, x.interest = x.interest)
plot(shapley)
```

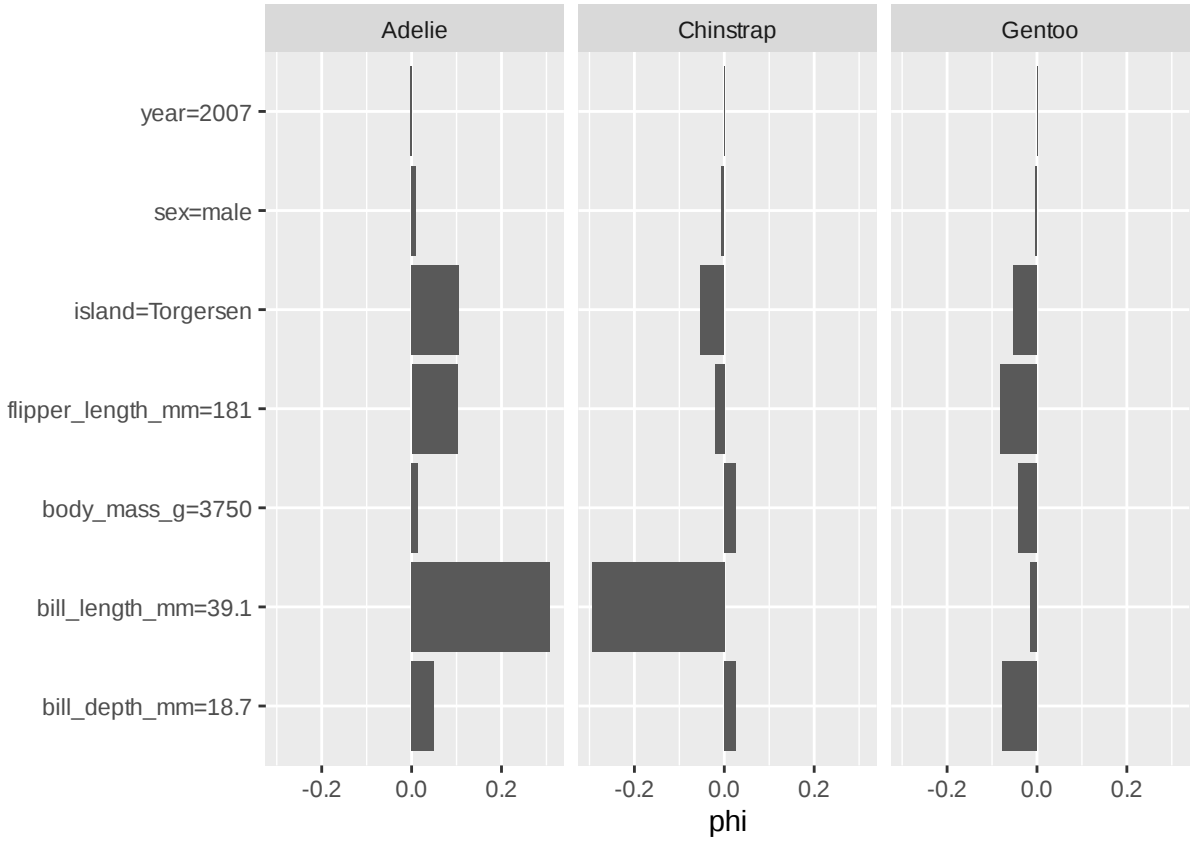


Figure 9.2: Plot of the results from Shapley. ϕ gives the increase or decrease in probability given the values on the vertical axis

The ϕ provides insight into the probability given the values on the vertical axis. For example, a penguin is less likely to be Gentoo if the bill_depth=18.7 is and much more likely to be Adelie than Chinstrap.

9.1.4 FeatureImp

```
effect = FeatureImp$new(model, loss = "ce")
effect$plot(features = num_features)
```

`FeatureImp` shows the level of importance of the features when classifying the penguins. It is clear to see that the `bill_length_mm` is of high importance and one should concentrate on different boundaries of this feature when attempting to classify the three species.

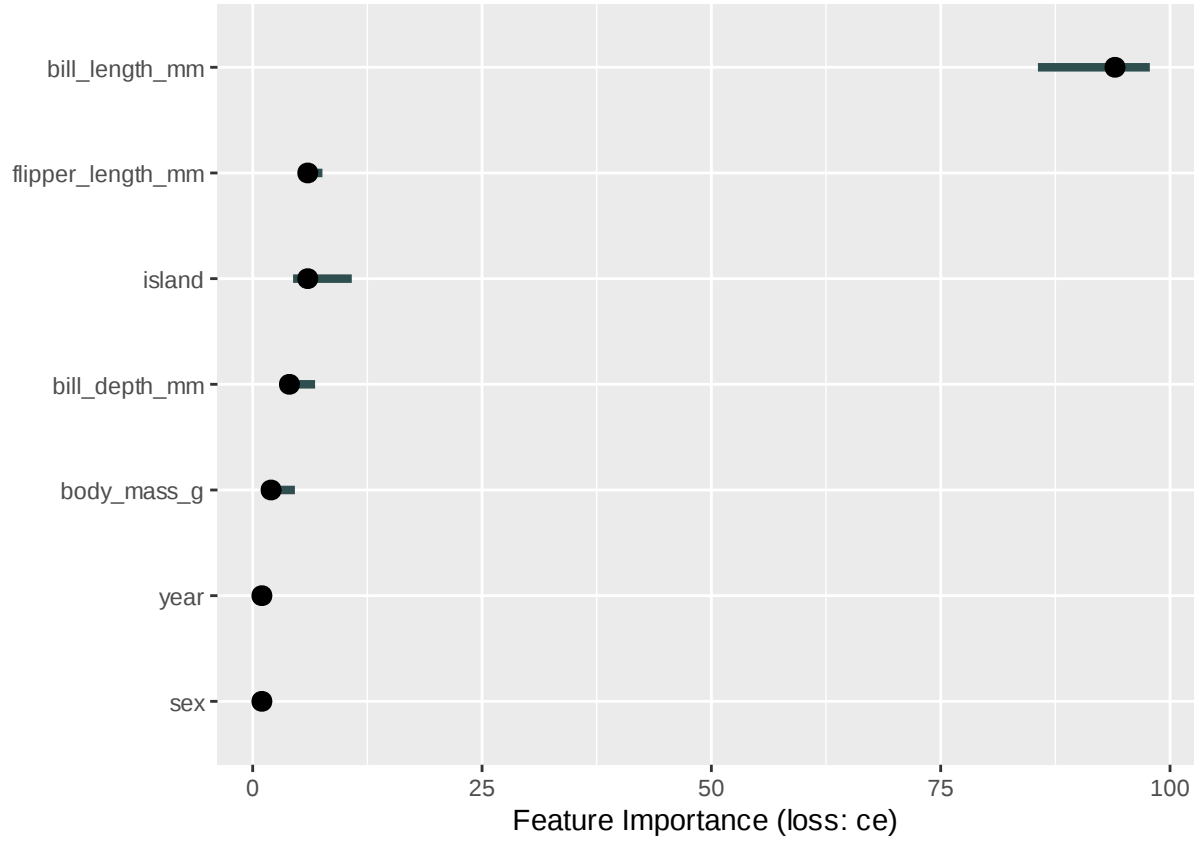


Figure 9.3: Plot of the results from FeatureImp. FeatureImp visualizes the importance of features given the prediction model

9.1.5 Independent Test Data

It is also interesting to see how well the model performs on a test data set. For this section, exactly as was recommended in Section 2.4, 80% of the penguin data set will be used for the training set and 20% for the test set:

```
train_set = sample(task_peng$nrow, 0.8 * task_peng$nrow)
test_set = setdiff(seq_len(task_peng$nrow), train_set)
learner$train(task_peng, row_ids = train_set)
prediction = learner$predict(task_peng, row_ids = test_set)
```

First, we compare the feature importance on training and test set

```
# plot on training
model = Predictor$new(learner, data = penguins[train_set, ], y = "species")
effect = FeatureImp$new(model, loss = "ce")
plot_train = plot(effect, features = num_features)

# plot on test data
model = Predictor$new(learner, data = penguins[test_set, ], y = "species")
effect = FeatureImp$new(model, loss = "ce")
plot_test = plot(effect, features = num_features)

# combine into single plot
library("patchwork")
plot_train + plot_test
```

The results of the train set for `FeatureImp` are very similar, which is expected. We follow a similar approach to compare the feature effects:

```
model = Predictor$new(learner, data = penguins[train_set, ], y = "species")
effect = FeatureEffects$new(model)
plot(effect, features = num_features)
```

```
model = Predictor$new(learner, data = penguins[test_set, ], y = "species")
effect = FeatureEffects$new(model)
plot(effect, features = num_features)
```

As is the case with `FeatureImp`, the test data results show either an over- or underestimate of feature importance / feature effects compared to the results where the entire penguin data set was used. This would be a good opportunity for the reader to attempt to resolve the estimation by playing with the amount of features and the amount of data used for both the test and train data sets of `FeatureImp` and `FeatureEffects`. Be sure to not change the line `train_set = sample(task_peng$nrow, 0.8 * task_peng$nrow)` as it will randomly sample the data again.

9.2 DALEX

Authors: Przemyslaw Biecek, Szymon Maksymiuk

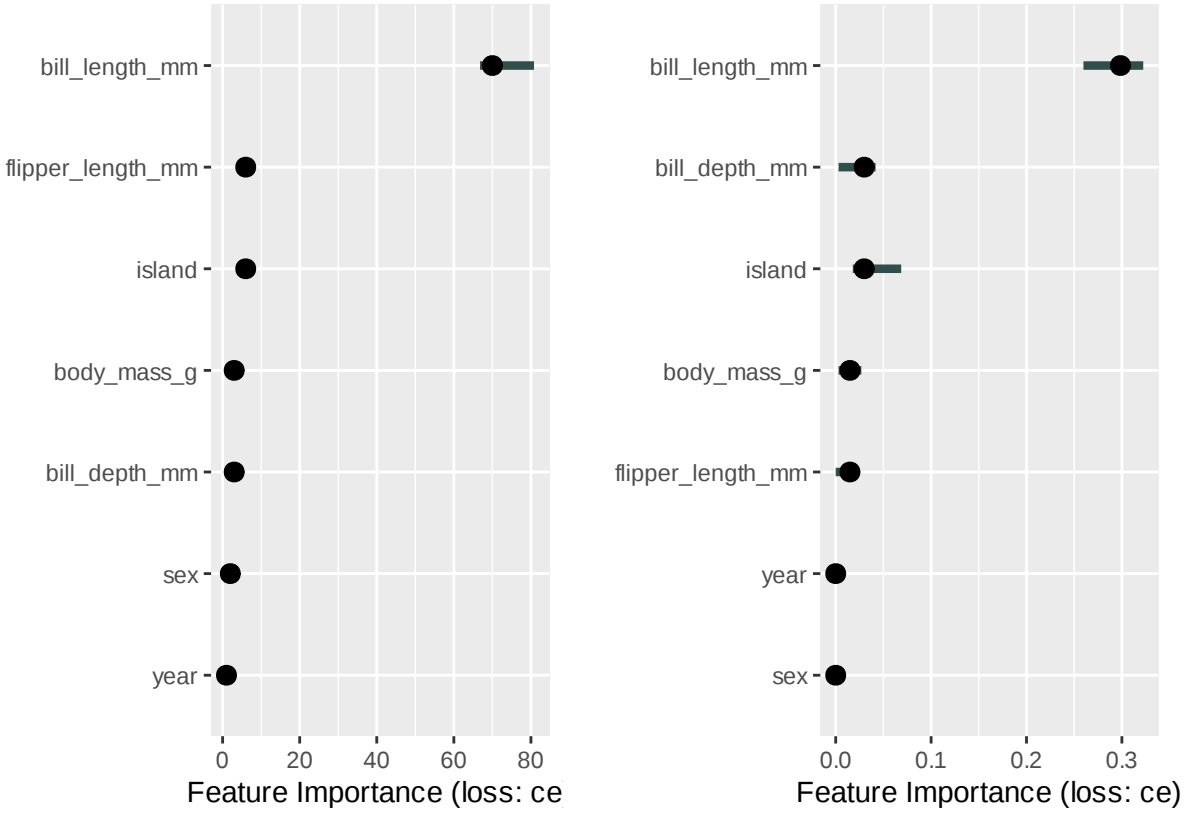


Figure 9.4: FeatImp on train (left) and test (right)

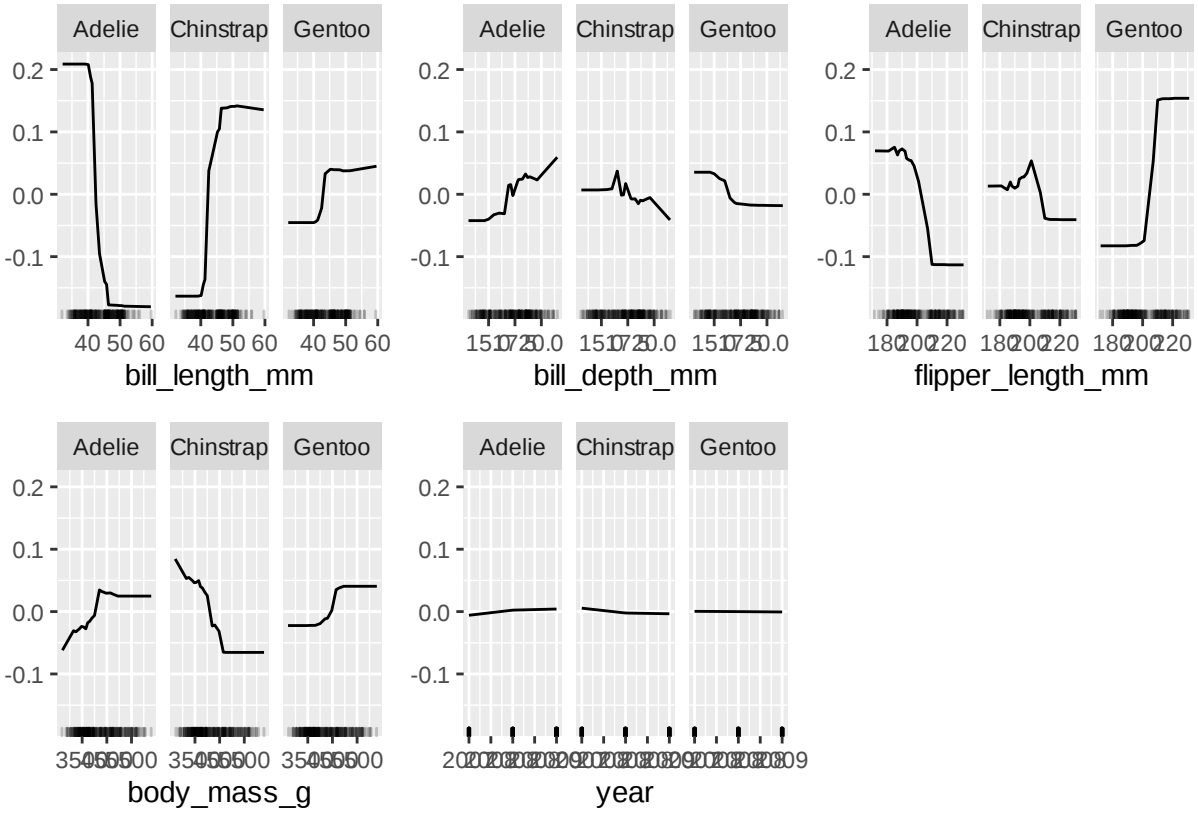


Figure 9.5: FeatEffect train data set

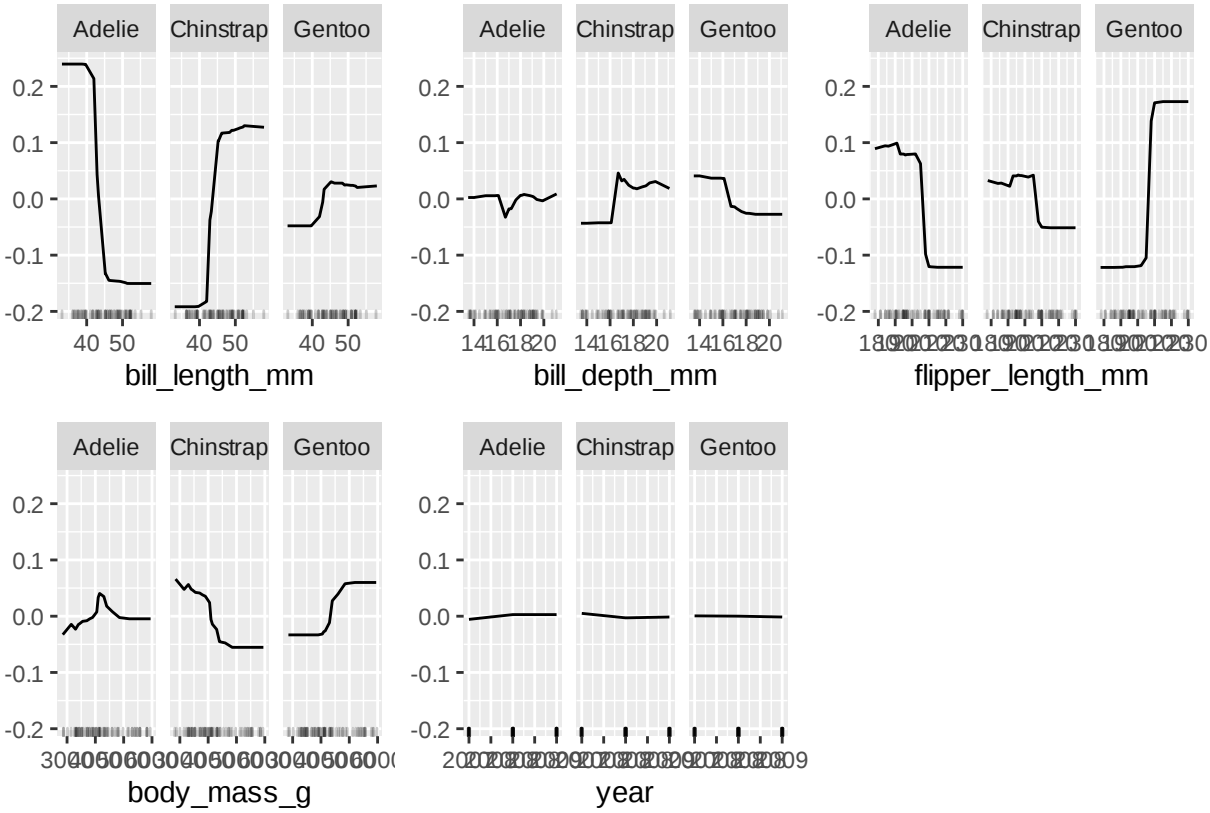


Figure 9.6: FeatEffect test data set

9.2.1 Introduction

The DALEX package X-rays any predictive model and helps to explore, explain and visualize its behaviour. The package implements a collection of methods for Explanatory Model Analysis. It is based on a unified grammar summarised in Figure 9.7.

In the following sections, we will present subsequent methods available in the DALEX package based on a random forest model trained for football players worth prediction on the FIFA 20 data. We will show both methods analyzing the model at the level of a single prediction and the global level - for the whole data set.

The structure of this chapter is the following:

- In Section 9.2.2 we introduce the FIFA 20 dataset and then in section 9.2.3 we train a random regression forest using the ranger package.
- Section 9.2.4 introduces general logic beyond DALEX explainers.
- Section 9.2.5 introduces methods for dataset level model exploration.
- Section 9.2.6 introduces methods for instance-level model exploration.

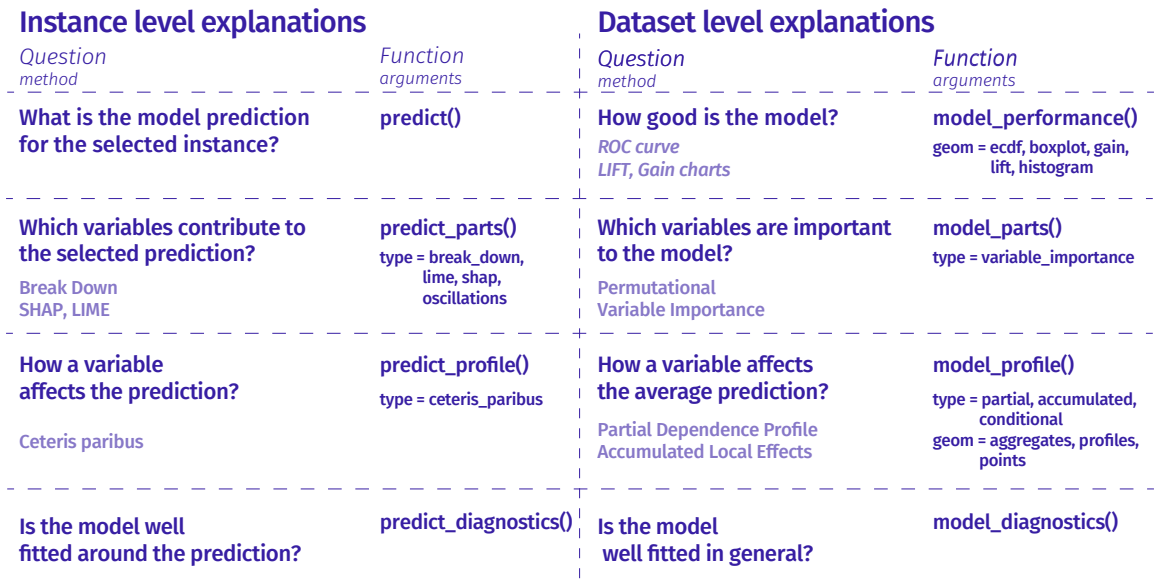


Figure 9.7: Taxonomy of methods for model exploration presented in this chapter. Left part overview methods for instance level exploration while right part is related to dataset level model exploration.

9.2.2 Read data: FIFA

Examples presented in this chapter are based on data retrieved from the FIFA video game. We will use the data scrapped from the sofifa website. The raw data is available at kaggle. After some basic data cleaning, the processed data for the top 5000 football players is available in the DALEX package under the name `fifa`.

```
library("DALEX")
fifa[1:2, c("value_eur", "age", "height_cm", "nationality", "attacking_crossing")]
```

```
##                    value_eur age height_cm nationality attacking_crossing
## L. Messi            95500000  32       170   Argentina                 88
## Cristiano Ronaldo   58500000  34       187    Portugal                 84
```

For every player, we have 42 features available.

```
dim(fifa)
```

```
## [1] 5000   42
```

In the table below we overview these 42 features for three selected players. One of the features, called `value_eur`, is the worth of the footballer in euros. In the next section, we will build a prediction model, which will estimate the worth of the player based on other player characteristics.

	Lionel Messi	Cristiano Ronaldo	Neymar Junior
wage_eur	565000	405000	290000
age	32	34	27
height_cm	170	187	175
weight_kg	72	83	68
nationality	Argentina	Portugal	Brazil
overall	94	93	92
potential	94	93	92
value_eur	95 500 000	58 500 000	105 500 000
attacking_crossing	88	84	87
attacking_finishing	95	94	87
attacking_heading_accuracy	70	89	62
attacking_short_passing	92	83	87
attacking_volleys	88	87	87
skill_dribbling	97	89	96
skill_curve	93	81	88
skill_fk_accuracy	94	76	87
skill_long_passing	92	77	81
skill_ball_control	96	92	95
movement_acceleration	91	89	94
movement_sprint_speed	84	91	89
movement_agility	93	87	96
movement_reactions	95	96	92
movement_balance	95	71	84
power_shot_power	86	95	80
power_jumping	68	95	61
power_stamina	75	85	81
power_strength	68	78	49
power_long_shots	94	93	84
mentality_aggression	48	63	51
mentality_interceptions	40	29	36
mentality_positioning	94	95	87
mentality_vision	94	82	90
mentality_penalties	75	85	90

	Lionel Messi	Cristiano Ronaldo	Neymar Junior
mentality_composure	96	95	94
defending_marking	33	28	27
defending_standing_tackle	37	32	26
defending_sliding_tackle	26	24	29
goalkeeping_diving	6	7	9
goalkeeping_handling	11	11	9
goalkeeping_kicking	15	15	15
goalkeeping_positioning	14	14	15
goalkeeping_reflexes	8	11	11

In order to get a more stable model we remove four variables i.e. `nationality`, `overall`, `potential`, `wage_eur`.

```
fifa[, c("nationality", "overall", "potential", "wage_eur")] = NULL
for (i in 1:ncol(fifa)) fifa[, i] = as.numeric(fifa[, i])
```

9.2.3 Train a model: Ranger

The `DALEX` package works for any model regardless of its internal structure. Examples of how this package works are shown on a random forest model implemented in the ranger package.

We use the `mlr3` package to build a predictive model. First, let's load the required packages.

```
library("mlr3")
library("mlr3learners")
```

Then we can define the regression task - prediction of the `value_eur` variable:

```
fifa_task = as_task_regr(fifa, target = "value_eur")
```

Finally, we train mlr3's `ranger learner` with 250 trees. Note that in this example for brevity we do not split the data into a train/test data. The model is built on the whole data.

```
fifa_ranger = lrn("regr.ranger")
fifa_ranger$param_set$values = list(num.trees = 250)
fifa_ranger$train(fifa_task)
fifa_ranger
```

```
## <LearnerRegrRanger:regr.ranger>
## * Model: ranger
## * Parameters: num.trees=250
## * Packages: mlr3, mlr3learners, ranger
## * Predict Type: response
## * Feature types: logical, integer, numeric, character, factor, ordered
## * Properties: hotstart_backward, importance, oob_error, weights
```

9.2.4 The general workflow

Working with explanations in the DALEX package always consists of three steps schematically shown in the pipe below.

```
model %>%
  explain_mlr3(data = ..., y = ..., label = ...) %>%
  model_parts() %>%
  plot()
```

1. All functions in the DALEX package can work for models with any structure. It is possible because in the first step we create an adapter that allows the downstream functions to access the model in a consistent fashion. In general, such an adapter is created with `DALEX::explain.default()` function, but for models created in the `mlr3` package it is more convenient to use the `DALEXtra::explain_mlr3()`.
2. Explanations are determined by the functions `DALEX::model_parts()`, `DALEX::model_profile()`, `DALEX::predict_parts()` and `DALEX::predict_profile()`. Each of these functions takes the model adapter as its first argument. The other arguments describe how the function works. We will present them in the following section.
3. Explanations can be visualized with the generic function `plot` or summarised with the generic function `print()`. Each explanation is a data frame with an additional class attribute. The `plot` function creates graphs using the ggplot2 package, so they can be easily modified with usual `ggplot2` decorators.

We show this cascade of functions based on the FIFA example.

To get started with the exploration of the model behaviour we need to create an explainer. `DALEX::explain.default` function handles is for all types of predictive models. In the DALEXtra package there generic versions for the most common ML frameworks. Among them the `DALEXtra::explain_mlr3()` function works for `mlr3` models.

This function performs a series of internal checks so the output is a bit verbose. Turn the `verbose = FALSE` argument to make it less wordy.

```
library("DALEX")
library("DALEXtra")

ranger_exp = explain_mlr3(fifa_ranger,
  data     = fifa,
  y        = fifa$value_eur,
  label    = "Ranger RF",
  colorize = FALSE)
```

```
## Preparation of a new explainer is initiated
##   -> model label       :  Ranger RF
##   -> data              :  5000  rows  38  cols
##   -> target variable   :  5000  values
##   -> predict function  :  yhat.LearnerRegr  will be used (  default  )
##   -> predicted values  :  No value for predict function target column. (  default  )
##   -> model_info        :  package mlr3 , ver. 0.13.0 , task regression (  default  )
```

```
##   -> predicted values  :  numerical, min =  447770 , mean =  7472865 , max =  88870933
##   -> residual function :  difference between y and yhat (  default  )
##   -> residuals         :  numerical, min =  -10107467 , mean =  421.9 , max =  18128700
##   A new explainer has been created!
```

9.2.5 Dataset level exploration

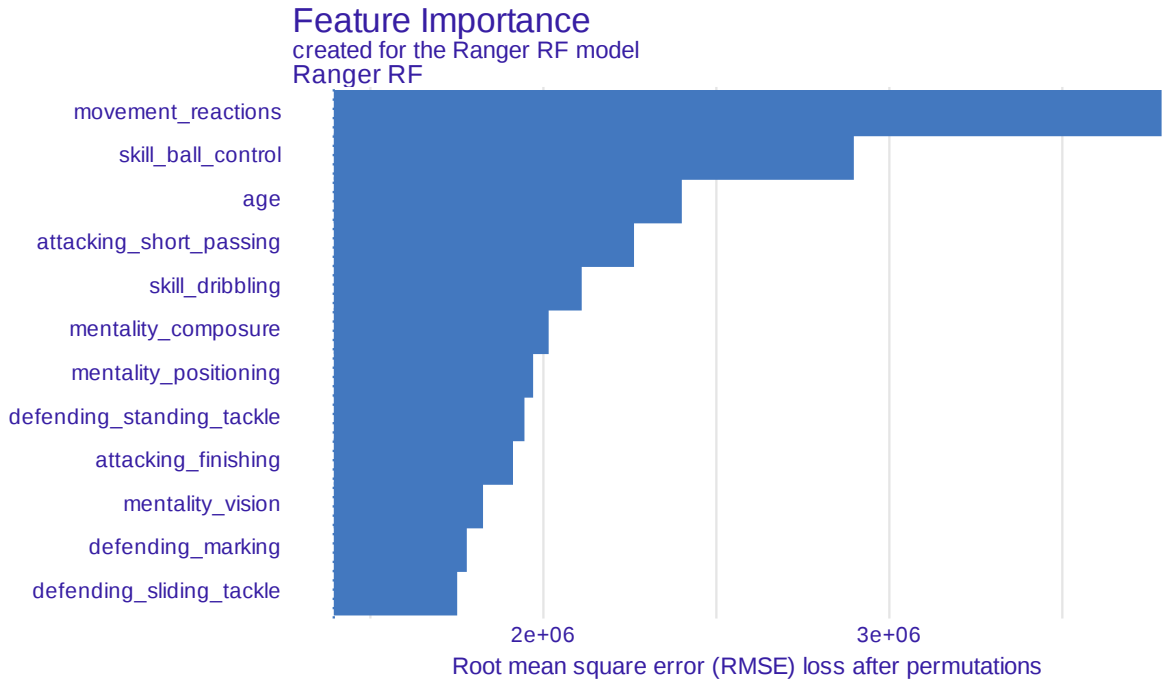
The `DALEX::model_parts()` function calculates the importance of variables using the permutations based importance.

```
fifa_vi = model_parts(ranger_exp)
head(fifa_vi)
```

```
##              variable mean_dropout_loss     label
## 1        _full_model_           1394097 Ranger RF
## 2           value_eur           1394097 Ranger RF
## 3 goalkeeping_kicking           1463250 Ranger RF
## 4    movement_balance           1466884 Ranger RF
## 5           weight_kg           1467757 Ranger RF
## 6           height_cm           1467852 Ranger RF
```

Results can be visualized with generic `plot()`. The chart for all 38 variables would be unreadable, so with the `max_vars` argument, we limit the number of variables on the plot.

```
plot(fifa_vi, max_vars = 12, show_boxplots = FALSE)
```



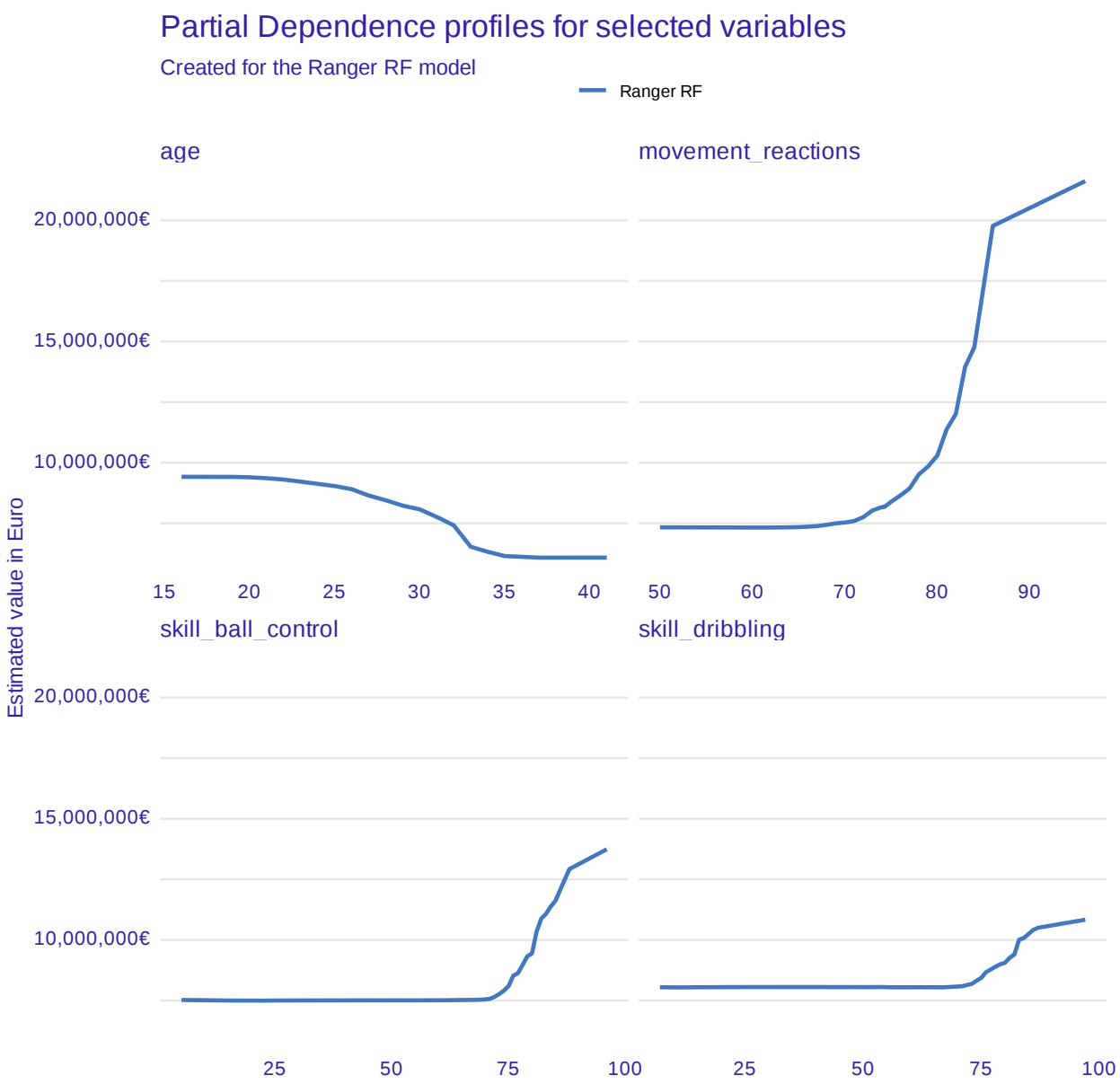
Once we know which variables are most important, we can use Partial Dependence Plots to show how the model, on average, changes with changes in selected variables. In this example, they show the average relation between the particular variables and players' value.

```
selected_variables = c("age", "movement_reactions",
  "skill_ball_control", "skill_dribbling")

fifa_pd = model_profile(ranger_exp,
  variables = selected_variables)$agr_profiles
fifa_pd
```

```
## Top profiles    :
##              _vname_   _label_ _x_  _yhat_ _ids_
## 1 skill_ball_control Ranger RF   5 7524743     0
## 2    skill_dribbling Ranger RF   7 8049872     0
## 3    skill_dribbling Ranger RF  11 8042138     0
## 4    skill_dribbling Ranger RF  12 8043199     0
## 5    skill_dribbling Ranger RF  13 8044775     0
## 6    skill_dribbling Ranger RF  14 8045948     0
```

Again, the result of the explanation can be presented with the generic function `plot()`.

```
library("ggplot2")
plot(fifa_pd) +
  scale_y_continuous("Estimated value in Euro", labels = scales::dollar_format(suffix = "€", prefix =
  ggtitle("Partial Dependence profiles for selected variables")
```



The general trend for most player characteristics is the same. The higher are the skills the higher is the player's worth. With a single exception – variable Age.

9.2.6 Instance level explanation

Time to see how the model behaves for a single observation/player This can be done for any player, but this example we will use the Cristiano Ronaldo.

The function `predict_parts` is an instance-level version of the `model_parts` function introduced in the previous section. For the background behind that method see the Introduction to Break Down.

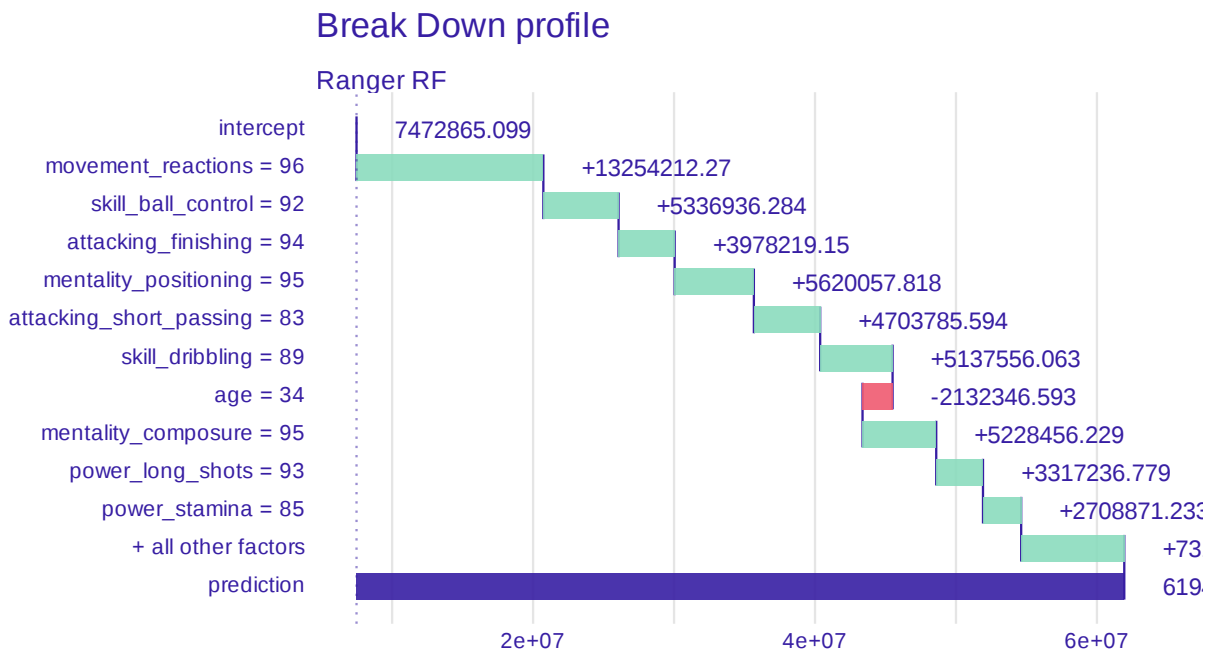
```
ronaldo = fifa["Cristiano Ronaldo", ]
ronaldo_bd_ranger = predict_parts(ranger_exp,
  new_observation = ronaldo)
head(ronaldo_bd_ranger)
```

```
##                                          contribution
## Ranger RF: intercept                          7472865
## Ranger RF: movement_reactions = 96           13254212
## Ranger RF: skill_ball_control = 92            5336936
## Ranger RF: attacking_finishing = 94           3978219
## Ranger RF: mentality_positioning = 95         5620058
## Ranger RF: attacking_short_passing = 83       4703786
```

The generic `plot()` function shows the estimated contribution of variables to the final prediction.

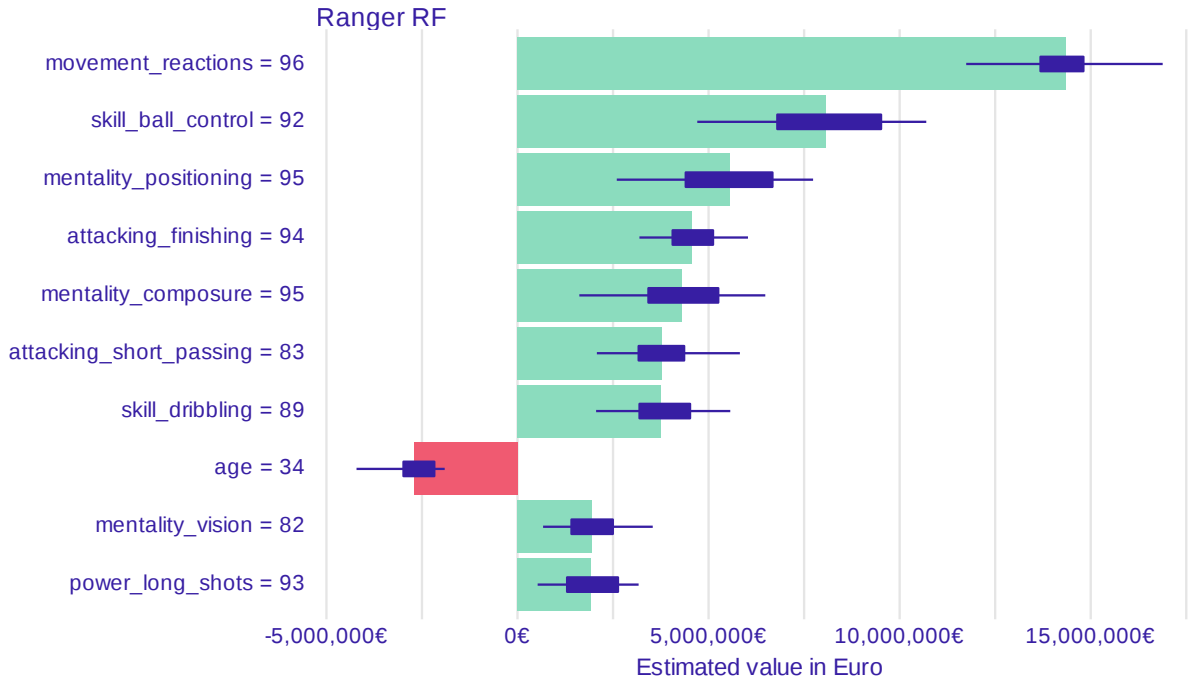
Cristiano is a striker, therefore characteristics that influence his worth are those related to attack, like `attacking_volleys` or `skill_dribbling`. The only variable with negative attribution is `age`.

```
plot(ronaldo_bd_ranger)
```



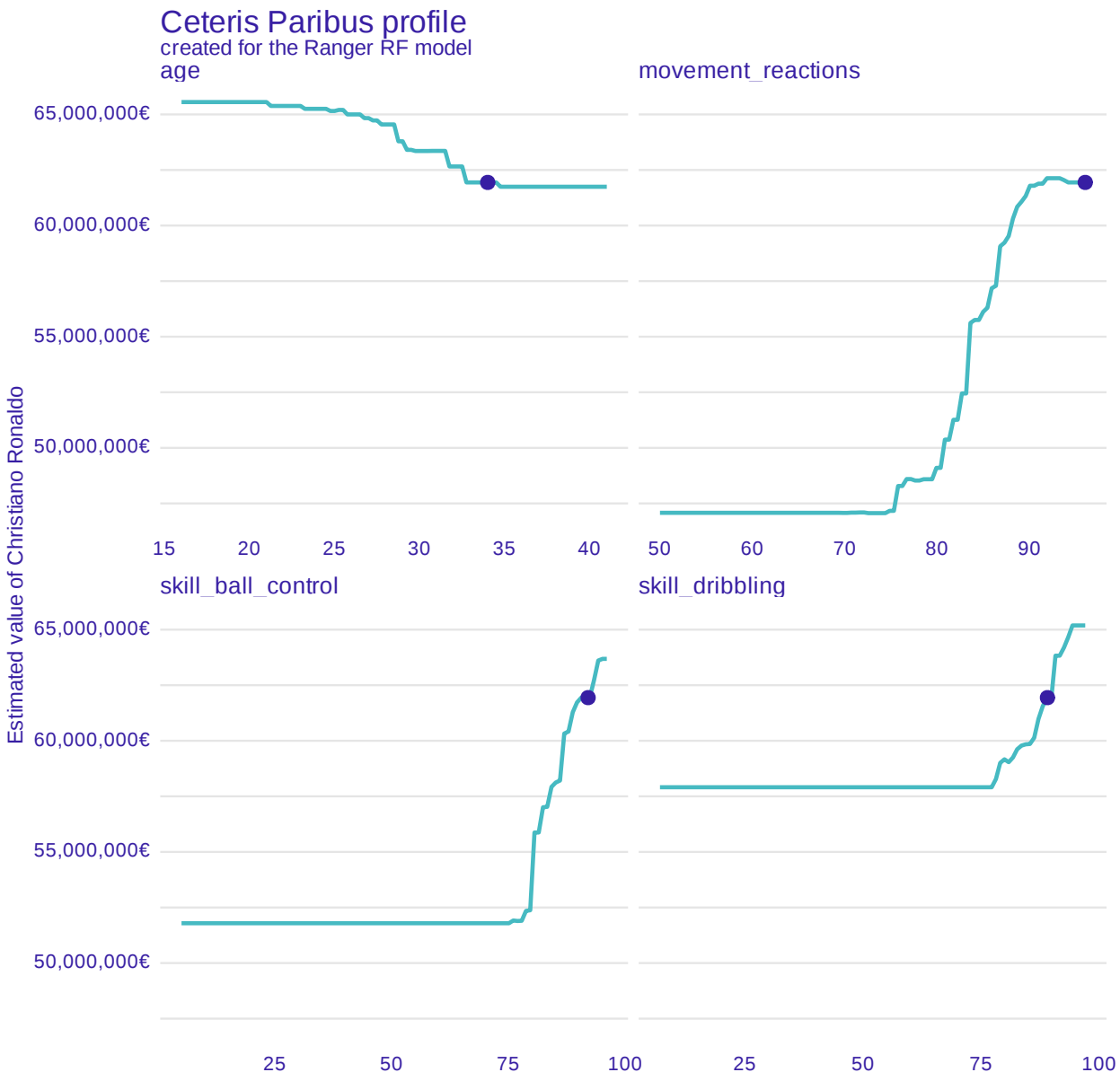
Another way to inspect the local behaviour of the model is to use SHapley Additive exPlanations (SHAP). It locally shows the contribution of variables to a single observation, just like Break Down.

```
ronaldo_shap_ranger = predict_parts(ranger_exp,
  new_observation = ronaldo,
  type = "shap")

plot(ronaldo_shap_ranger) +
  scale_y_continuous("Estimated value in Euro", labels = scales::dollar_format(suffix = "€", prefix =
```



In the previous section, we've introduced a global explanation - Partial Dependence Plots. Ceteris Paribus is the instance level version of that plot. It shows the response of the model for observation when we change only one variable while others stay unchanged. Blue dot stands for the original value.

```
selected_variables = c("age", "movement_reactions",
  "skill_ball_control", "skill_dribbling")

ronaldo_cp_ranger = predict_profile(ranger_exp, ronaldo, variables = selected_variables)

plot(ronaldo_cp_ranger, variables = selected_variables) +
  scale_y_continuous("Estimated value of Christiano Ronaldo", labels = scales::dollar_format(suffix =
```


Ceteris Paribus profile
created for the Ranger RF model
age
movement_reactions
skill_ball_control
skill_dribbling
Estimated value of Christiano Ronaldo
65,000,000€
60,000,000€
55,000,000€
50,000,000€
15
20
25
30
35
40
50
60
70
80
90
25
50
75
100

10 Appendix

10.1 Integrated Learners

Learners are available from one of the following packages:

- mlr3: debug learner and rpart learners.
- mlr3learners: opinionated selection of some default learners.
- mlr3proba: base learners for survival and probabilistic regression.
- mlr3cluster: learners for unsupervised clustering.
- mlr3extralearners: more experimental learners for regression, classification and survival.

Use the interactive search table to look through all our learners.

10.2 Integrated Performance Measures

Also see the overview on the website of mlr3measures.

Id
[‘aic‘](https://mlr3.mlr-org.com/reference/mlr_measures_aic.html)
[‘bic‘](https://mlr3.mlr-org.com/reference/mlr_measures_bic.html)
[‘classif.acc‘](https://mlr3.mlr-org.com/reference/mlr_measures_classif.acc.html)
[‘classif.auc‘](https://mlr3.mlr-org.com/reference/mlr_measures_classif.auc.html)
[‘classif.bacc‘](https://mlr3.mlr-org.com/reference/mlr_measures_classif.bacc.html)
[‘classif.bbrier‘](https://mlr3.mlr-org.com/reference/mlr_measures_classif.bbrier.html)
[‘classif.ce‘](https://mlr3.mlr-org.com/reference/mlr_measures_classif.ce.html)
[‘classif.costs‘](https://mlr3.mlr-org.com/reference/mlr_measures_classif.costs.html)
[‘classif.dor‘](https://mlr3.mlr-org.com/reference/mlr_measures_classif.dor.html)
[‘classif.fbeta‘](https://mlr3.mlr-org.com/reference/mlr_measures_classif.fbeta.html)
[‘classif.fdr‘](https://mlr3.mlr-org.com/reference/mlr_measures_classif.fdr.html)
[‘classif.fn‘](https://mlr3.mlr-org.com/reference/mlr_measures_classif.fn.html)
[‘classif.fnr‘](https://mlr3.mlr-org.com/reference/mlr_measures_classif.fnr.html)
[‘classif.fomr‘](https://mlr3.mlr-org.com/reference/mlr_measures_classif.fomr.html)
[‘classif.fp‘](https://mlr3.mlr-org.com/reference/mlr_measures_classif.fp.html)
[‘classif.fpr‘](https://mlr3.mlr-org.com/reference/mlr_measures_classif.fpr.html)
[‘classif.logloss‘](https://mlr3.mlr-org.com/reference/mlr_measures_classif.logloss.html)
[‘classif.mbrier‘](https://mlr3.mlr-org.com/reference/mlr_measures_classif.mbrier.html)
[‘classif.mcc‘](https://mlr3.mlr-org.com/reference/mlr_measures_classif.mcc.html)
[‘classif.npv‘](https://mlr3.mlr-org.com/reference/mlr_measures_classif.npv.html)
[‘classif.ppv‘](https://mlr3.mlr-org.com/reference/mlr_measures_classif.ppv.html)
[‘classif.prauc‘](https://mlr3.mlr-org.com/reference/mlr_measures_classif.prauc.html)
[‘classif.precision‘](https://mlr3.mlr-org.com/reference/mlr_measures_classif.precision.html)
[‘classif.recall‘](https://mlr3.mlr-org.com/reference/mlr_measures_classif.recall.html)
[‘classif.sensitivity‘](https://mlr3.mlr-org.com/reference/mlr_measures_classif.sensitivity.html)
[‘classif.specificity‘](https://mlr3.mlr-org.com/reference/mlr_measures_classif.specificity.html)
[‘classif.tn‘](https://mlr3.mlr-org.com/reference/mlr_measures_classif.tn.html)
[‘classif.tnr‘](https://mlr3.mlr-org.com/reference/mlr_measures_classif.tnr.html)
[‘classif.tp‘](https://mlr3.mlr-org.com/reference/mlr_measures_classif.tp.html)
[‘classif.tpr‘](https://mlr3.mlr-org.com/reference/mlr_measures_classif.tpr.html)
[‘debug‘](https://mlr3.mlr-org.com/reference/mlr_measures_debug.html)
[‘dens.logloss‘](https://mlr3proba.mlr-org.com/reference/mlr_measures_dens.logloss.html)
[‘oob_error‘](https://mlr3.mlr-org.com/reference/mlr_measures_oob_error.html)
[‘regr.bias‘](https://mlr3.mlr-org.com/reference/mlr_measures_regr.bias.html)
[‘regr.ktau‘](https://mlr3.mlr-org.com/reference/mlr_measures_regr.ktau.html)
[‘regr.mae‘](https://mlr3.mlr-org.com/reference/mlr_measures_regr.mae.html)
[‘regr.mape‘](https://mlr3.mlr-org.com/reference/mlr_measures_regr.mape.html)
[‘regr.maxae‘](https://mlr3.mlr-org.com/reference/mlr_measures_regr.maxae.html)
[‘regr.medae‘](https://mlr3.mlr-org.com/reference/mlr_measures_regr.medae.html)
[‘regr.medse‘](https://mlr3.mlr-org.com/reference/mlr_measures_regr.medse.html)
[‘regr.mse‘](https://mlr3.mlr-org.com/reference/mlr_measures_regr.mse.html)
[‘regr.msle‘](https://mlr3.mlr-org.com/reference/mlr_measures_regr.msle.html)
[‘regr.pbias‘](https://mlr3.mlr-org.com/reference/mlr_measures_regr.pbias.html)
[‘regr.rae‘](https://mlr3.mlr-org.com/reference/mlr_measures_regr.rae.html)
[‘regr.rmse‘](https://mlr3.mlr-org.com/reference/mlr_measures_regr.rmse.html)
[‘regr.rmsle‘](https://mlr3.mlr-org.com/reference/mlr_measures_regr.rmsle.html)
[‘regr.rrse‘](https://mlr3.mlr-org.com/reference/mlr_measures_regr.rrse.html)
[‘regr.rse‘](https://mlr3.mlr-org.com/reference/mlr_measures_regr.rse.html)
[‘regr.rsq‘](https://mlr3.mlr-org.com/reference/mlr_measures_regr.rsq.html)
[‘regr.sae‘](https://mlr3.mlr-org.com/reference/mlr_measures_regr.sae.html)
[‘regr.smape‘](https://mlr3.mlr-org.com/reference/mlr_measures_regr.smape.html)
[‘regr.srho‘](https://mlr3.mlr-org.com/reference/mlr_measures_regr.srho.html)
[‘regr.sse‘](https://mlr3.mlr-org.com/reference/mlr_measures_regr.sse.html)
[‘selected_features‘](https://mlr3.mlr-org.com/reference/mlr_measures_selected_features.html)
[‘sim.jaccard‘](https://mlr3.mlr-org.com/reference/mlr_measures_sim.jaccard.html)

10.3 Integrated Filter Methods

10.3.1 Standalone filter methods

Id	Packages
[‘anova‘](https://mlr3filters.mlr-org.com/reference/mlr_filters_anova.html)	stats
[‘auc‘](https://mlr3filters.mlr-org.com/reference/mlr_filters_auc.html)	[mlr3me
[‘carscore‘](https://mlr3filters.mlr-org.com/reference/mlr_filters_carscore.html)	[care](ht
[‘cmim‘](https://mlr3filters.mlr-org.com/reference/mlr_filters_cmim.html)	[praznik
[‘correlation‘](https://mlr3filters.mlr-org.com/reference/mlr_filters_correlation.html)	stats
[‘disr‘](https://mlr3filters.mlr-org.com/reference/mlr_filters_disr.html)	[praznik
[‘find_correlation‘](https://mlr3filters.mlr-org.com/reference/mlr_filters_find_correlation.html)	stats
[‘importance‘](https://mlr3filters.mlr-org.com/reference/mlr_filters_importance.html)	[mlr3](ht
[‘information_gain‘](https://mlr3filters.mlr-org.com/reference/mlr_filters_information_gain.html)	[FSelecto
[‘jmi‘](https://mlr3filters.mlr-org.com/reference/mlr_filters_jmi.html)	[praznik
[‘jmim‘](https://mlr3filters.mlr-org.com/reference/mlr_filters_jmim.html)	[praznik
[‘kruskal_test‘](https://mlr3filters.mlr-org.com/reference/mlr_filters_kruskal_test.html)	stats
[‘mim‘](https://mlr3filters.mlr-org.com/reference/mlr_filters_mim.html)	[praznik
[‘mrmr‘](https://mlr3filters.mlr-org.com/reference/mlr_filters_mrmr.html)	[praznik
[‘njmim‘](https://mlr3filters.mlr-org.com/reference/mlr_filters_njmim.html)	[praznik
[‘performance‘](https://mlr3filters.mlr-org.com/reference/mlr_filters_performance.html)	[mlr3](ht
[‘permutation‘](https://mlr3filters.mlr-org.com/reference/mlr_filters_permutation.html)	[mlr3](ht
[‘relief‘](https://mlr3filters.mlr-org.com/reference/mlr_filters_relief.html)	[FSelecto
[‘variance‘](https://mlr3filters.mlr-org.com/reference/mlr_filters_variance.html)	stats

10.3.2 Learners With Embedded Filter Methods

```
##  [1] "classif.featureless" "classif.ranger"      "classif.rpart"
##  [4] "classif.xgboost"     "regr.featureless"    "regr.ranger"
##  [7] "regr.rpart"          "regr.xgboost"        "surv.ranger"
## [10] "surv.rpart"          "surv.xgboost"
```

10.4 Integrated Pipe Operators

Id
[‘boxcox‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_boxcox.html)
[‘colapply‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_colapply.html)
[‘collapsefactors‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_collapsefactors.html)
[‘colroles‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_colroles.html)
[‘datefeatures‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_datefeatures.html)
[‘histbin‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_histbin.html)
[‘ica‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_ica.html)
[‘kernelpca‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_kernelpca.html)
[‘modelmatrix‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_modelmatrix.html)
[‘mutate‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_mutate.html)
[‘nmf‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_nmf.html)
[‘pca‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_pca.html)
[‘quantilebin‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_quantilebin.html)
[‘randomprojection‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_randomprojection.html)
[‘renamecolumns‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_renamecolumns.html)
[‘scale‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_scale.html)
[‘scalemaxabs‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_scalemaxabs.html)
[‘scalerange‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_scalerange.html)
[‘spatialsign‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_spatialsign.html)
[‘subsample‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_subsample.html)
[‘textvectorizer‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_textvectorizer.html)
[‘yeojohnson‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_yeojohnson.html)
[‘encode‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_encode.html)
[‘encodeimpact‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_encodeimpact.html)
[‘encodelmer‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_encodelmer.html)
[‘vtreat‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_vtreat.html)
[‘classifavg‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_classifavg.html)
[‘featureunion‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_featureunion.html)
[‘regravg‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_regravg.html)
[‘survavg‘](https://mlr3proba.mlr-org.com/reference/mlr_pipeops_survavg.html)
[‘filter‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_filter.html)
[‘select‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_select.html)
[‘classbalancing‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_classbalancing.html)
[‘classweights‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_classweights.html)
[‘smote‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_smote.html)
[‘learner‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_learner.html)
[‘learner_cv‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_learner_cv.html)
[‘imputeconstant‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_imputeconstant.html)
[‘imputehist‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_imputehist.html)
[‘imputelearner‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_imputelearner.html)
[‘imputemean‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_imputemean.html)
[‘imputemedian‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_imputemedian.html)
[‘imputemode‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_imputemode.html)
[‘imputeoor‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_imputeoor.html)
[‘imputesample‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_imputesample.html)
[‘missind‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_missind.html)
[‘multiplicityexply‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_multiplicityexply.html)
[‘multiplicityimply‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_multiplicityimply.html)
[‘ovrunite‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_ovrunite.html)
[‘replicate‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_replicate.html)
[‘fixfactors‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_fixfactors.html)
[‘removeconstants‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_removeconstants.html)
[‘ovrsplit‘](https://mlr3pipelines.mlr-org.com/reference/mlr_pipeops_ovrsplit.html)

10.5 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.

10.5.1 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 = tsk("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 `PipeOp`s can be obtained from

```
library("mlr3pipelines")
po()
```

```
## <DictionaryPipeOp> with 64 stored values
## Keys: boxcox, branch, chunk, classbalancing, classifavg, classweights,
##   colapply, collapsefactors, colroles, copy, datefeatures, encode,
##   encodeimpact, encodelmer, featureunion, filter, fixfactors, histbin,
##   ica, imputeconstant, imputehist, imputelearner, imputemean,
##   imputemedian, imputemode, imputeoor, imputesample, kernelpca,
##   learner, learner_cv, missind, modelmatrix, multiplicityexply,
##   multiplicityimply, mutate, nmf, nop, ovrsplit, ovrunite, pca, proxy,
##   quantilebin, randomprojection, randomresponse, regravg,
##   removeconstants, renamecolumns, replicate, scale, scalemaxabs,
##   scalerange, select, smote, spatialsign, subsample, targetinvert,
##   targetmutate, targettrafoscalerange, textvectorizer, threshold,
##   tunethreshold, unbranch, vtreat, yeojohnson
```

First we define the required PipeOps:

```
op1 = po("scale")
op2 = po("pca")
op3 = po("learner", learner = lrn("classif.rpart"))
```

10.5.1.1 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`.

10.5.1.2 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_ids      truth   response
##           6     setosa     setosa
##          82 versicolor versicolor
##          58 versicolor versicolor
## ---
##          27     setosa     setosa
##          42     setosa     setosa
##          12     setosa     setosa
```

10.5.2 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.

10.5.2.1 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, classes = list(factor = imputeConstant("missing")))

# Afterwards, we can train the resulting learner on a task
train(lrn, iris.task)
```

10.5.2.2 mlr3pipelines

```
library("mlr3")
library("mlr3pipelines")
library("mlr3filters")

impute = po("imputeoor")
filter = po("filter", filter = flt("variance"), filter.nfeat = 2L)
rpart = po("learner", lrn("classif.rpart"))

# Assemble the Pipeline
pipeline = impute %>>% filter %>>% rpart
# And convert to a 'GraphLearner'
learner = as_learner(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.

10.5.3 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.

10.5.3.1 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)
```

10.5.3.2 mlr3pipelines

```
library("mlr3verse")
iris = tsk("iris")

# Build the steps
copy = po("copy", 2)
pca = po("pca")
selection = po("filter", filter = flt("variance"))
union = po("featureunion", 2)
svm = po("learner", lrn("classif.svm", kernel = "linear", type = "C-classification"))

# Assemble the Pipeline
pipeline = copy %>>% gunion(list(pca, selection)) %>>% union %>>% svm
learner = as_learner(pipeline)

# For tuning, we define the resampling and the Parameter Space
resampling = rsmp("cv", folds = 5L)
```

```
library("paradox")
search_space = ps(
  classif.svm.cost = p_dbl(lower = 0.1, upper = 1),
  pca.rank. = p_int(lower = 1, upper = 3),
  variance.filter.nfeat = p_int(lower = 1, upper = 2)
)

instance = TuningInstanceSingleCrit$new(
  task = iris,
  learner = learner,
  resampling = resampling,
  measure = msr("classif.ce"),
  terminator = trm("none"),
  search_space = search_space
)

tuner = tnr("grid_search", resolution = 10)
tuner$optimize(instance)

Set the learner to the optimal values and train
learner$param_set$values = instance$result_learner_param_vals
```

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`.

10.5.4 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.

10.5.4.1 recipes

```
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:

```
# 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)
```

10.5.4.2 mlr3pipelines

The same analysis can be performed in mlr3pipelines. Note, that for now we do not impute via knn but instead via sampling.

```
library("data.table")
library("mlr3")
library("mlr3learners")
library("mlr3pipelines")
data("credit_data", package = "modeldata")
set.seed(55)

# Create the task
task = as_task_classif(credit_data, target = "Status")

# Build up the Pipeline:
g = po("imputesample", id = "impute") %>>%
  po("encode", method = "one-hot") %>>%
  po("scale") %>>%
  po("learner", lrn("classif.ranger", 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 = as_learner(g)
resample(task, glrn, rsmp("holdout"))
```

Citation Info

To cite this book, please use the following information:

```
@misc{
  title = {},
  author = {},
  url = {https://mlr3book.mlr-org.com},
  year = {2021},
  month = {11},
  day = {24},
}
```

For the package `mlr3`, please cite our JOSS paper:

```
## @Article{mlr3,
##   title = {{mlr3}: A modern object-oriented machine learning framework in {R}},
##   author = {Michel Lang and Martin Binder and Jakob Richter and Patrick Schratz and Floria
##   journal = {Journal of Open Source Software},
##   year = {2019},
##   month = {dec},
##   doi = {10.21105/joss.01903},
##   url = {https://joss.theoj.org/papers/10.21105/joss.01903},
## }
```

References

Bergstra, James, and Yoshua Bengio. 2012. "Random Search for Hyper-Parameter Optimization." *J. Mach. Learn. Res.* 13: 281–305.

Binder, Martin, Florian Pfisterer, Michel Lang, Lennart Schneider, Lars Kotthoff, and Bernd Bischl. 2021. "mlr3pipelines - Flexible Machine Learning Pipelines in r." *Journal of Machine Learning Research* 22 (184): 1–7. http://jmlr.org/papers/v22/21-0281.html.

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): 123–40.

Brenning, Alexander. 2012. "Spatial Cross-Validation and Bootstrap for the Assessment of Prediction Rules in Remote Sensing: The r Package Sperrorest." In *2012 IEEE International Geoscience and Remote Sensing Symposium*. IEEE. https://doi.org/10.1109/igarss.2012.6352393.

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.

Collett, David. 2014. *Modelling Survival Data in Medical Research*. 3rd ed. CRC.

Guyon, Isabelle, and André Elisseeff. 2003. "An Introduction to Variable and Feature Selection." *Journal of Machine Learning Research* 3 (Mar): 1157–82.

Hastie, Trevor, Jerome Friedman, and Robert Tibshirani. 2001. *The Elements of Statistical Learning*. Springer New York. https://doi.org/10.1007/978-0-387-21606-5.

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.

Legendre, Pierre. 1993. "Spatial Autocorrelation: Trouble or New Paradigm?" *Ecology* 74 (6): 1659–73. https://doi.org/10.2307/1939924.

Li, Lisha, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2016. "Efficient Hyperparameter Optimization and Infinitely Many Armed Bandits." *CoRR* abs/1603.06560. http://arxiv.org/abs/1603.06560.

Lovelace, Robin, Jakub Nowosad, and Jannes Muenchow. 2019. *Geocomputation with r*. CRC Press.

Muenchow, J., A. Brenning, and M. Richter. 2012. "Geomorphic Process Rates of Landslides Along a Humidity Gradient in the Tropical Andes." *Geomorphology* 139-140: 271–84. https://doi.org/https://doi.org/10.1016/j.geomorph.2011.10.029.

R Core Team. 2019. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. https://www.R-project.org/.

Schratz, Patrick, Jannes Muenchow, Eugenia Iturritxa, Jakob Richter, and Alexander Brenning. 2019. "Hyperparameter Tuning and Performance Assessment of Statistical and Machine-Learning Algorithms Using Spatial Data." *Ecological Modelling* 406 (August): 109–20.

https://doi.org/10.1016/j.ecolmodel.2019.06.002.

Silverman, Bernard W. 1986. *Density Estimation for Statistics and Data Analysis*. Vol. 26. CRC press.

Simon, Richard. 2007. "Resampling Strategies for Model Assessment and Selection." In *Fundamentals of Data Mining in Genomics and Proteomics*, edited by Werner Dubitzky, Martin Granzow, and Daniel Berrar, 173–86. Boston, MA: Springer US. https://doi.org/10.1007/978-0-387-47509-7_8.

Sonabend, Raphael, Franz J Király, Andreas Bender, Bernd Bischl, and Michel Lang. 2021. "Mlr3proba: An r Package for Machine Learning in Survival Analysis." *Bioinformatics*, February. https://doi.org/10.1093/bioinformatics/btab039.

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.