The mlr3 Manual

presented by
The mlr-org Team

Contents

Table of Contents 2

Quickstart 6

1 Introduction and Overview 8

2 Basics 10
- 2.1 Quick R6 Intro for Beginners . . . 12
- 2.2 Tasks . . . 12
  - 2.2.1 Task Types . . . 12
  - 2.2.2 Task Creation . . . 13
  - 2.2.3 Predefined tasks . . . 14
  - 2.2.4 Task API . . . 15
  - 2.2.5 Plotting Tasks . . . 20
- 2.3 Learners . . . 25
  - 2.3.1 Predefined Learners . . . 26
- 2.4 Train and Predict . . . 28
  - 2.4.1 Creating Task and Learner Objects . . . 28
  - 2.4.2 Setting up the train/test splits of the data . . . 29
  - 2.4.3 Training the learner . . . 29
  - 2.4.4 Predicting . . . 30
  - 2.4.5 Changing the Predict Type . . . 31
  - 2.4.6 Plotting Predictions . . . 31
  - 2.4.7 Performance assessment . . . 34
- 2.5 Resampling . . . 35
  - 2.5.1 Settings . . . 36
  - 2.5.2 Instantiation . . . 37
  - 2.5.3 Execution . . . 37
  - 2.5.4 Custom resampling . . . 39
  - 2.5.5 Plotting Resample Results . . . 40
- 2.6 Benchmarking . . . 41
  - 2.6.1 Design Creation . . . 41
  - 2.6.2 Execution and Aggregation of Results . . . 43
  - 2.6.3 Plotting Benchmark Results . . . 44
  - 2.6.4 Extracting ResampleResults . . . 46
  - 2.6.5 Converting and Merging ResampleResults . . . 47
- 2.7 Binary classification . . . 48
  - 2.7.1 ROC Curve and Thresholds . . . 48
  - 2.7.2 Threshold Tuning . . . 51

3 Model Optimization 52
- 3.1 Hyperparameter Tuning . . . 53
  - 3.1.1 The `TuningInstance` Class . . . 53
  - 3.1.2 The `Tuner` Class . . . 55

3.1.3 Triggering the Tuning . . . 55
3.1.4 Automating the Tuning . . . 56
3.2 Feature Selection / Filtering . . . 58
3.2.1 Filters . . . 58
3.2.2 Calculating filter values . . . 58
3.2.3 Variable Importance Filters . . . 59
3.2.4 Ensemble Methods . . . 60
3.2.5 Wrapper Methods . . . 60
3.3 Nested Resampling . . . 60
3.3.1 Execution . . . 62
3.3.2 Evaluation . . . 62

4 Pipelines **64**
4.1 The Building Blocks: PipeOps . . . 65
4.2 The Pipeline Operator: `%>>%` . . . 69
4.3 Nodes, Edges and Graphs . . . 70
4.4 Modeling . . . 72
4.4.1 Setting Hyperparameters . . . 73
4.4.2 Tuning . . . 73
4.5 Non-Linear Graphs . . . 74
4.5.1 Branching & Copying . . . 74
4.5.2 Model Ensembles . . . 76
4.6 Special Operators . . . 81
4.6.1 Imputation: `PipeOpImpute` . . . 81
4.6.2 Feature Engineering: `PipeOpMutate` . . . 81
4.6.3 Training on data subsets: `PipeOpChunk` . . . 82
4.6.4 Feature Selection: `PipeOpFilter` and `PipeOpSelect` . . . 84

5 Technical **85**
5.1 Parallelization . . . 85
5.2 Error Handling . . . 86
5.2.1 Encapsulation . . . 88
5.2.2 Fallback learners . . . 89
5.3 Database Backends . . . 91
5.3.1 Use Case: NYC Flights . . . 91
5.3.2 Preprocessing with `dplyr` . . . 92
5.3.3 DataBackendDplyr . . . 93
5.3.4 Model fitting . . . 94
5.3.5 Cleanup . . . 95
5.4 Parameters (using paradox) . . . 95
5.4.1 Reference Based Objects . . . 95
5.4.2 Defining a Parameter Space . . . 96
5.4.3 Parameter Sampling . . . 102
5.4.4 Parameter Transformation . . . 105
5.5 Logging and Verbosity . . . 108
5.5.1 Available logging levels . . . 108
5.5.2 Global Setting . . . 108
5.5.3 Changing mlr3 logging levels . . . 108
5.6 mlr -> mlr3 Transition Guide . . . 109

6 Extending **116**
6.1 Extending with Learners . . . 116
6.1.1 Meta-information . . . 117
6.1.2 Train function . . . 118
6.1.3 Predict function . . . 120
6.1.4 Final learner . . . 120
6.2 Extending with mlr3pipelines . . . 122
6.2.1 General Case Example: `PipeOpCopy` . . . 123
6.2.2 Special Case: Preprocessing . . . 126
6.2.3 Special Case: Preprocessing with Simple Train . . . 130
6.2.4 Hyperparameters . . . 134

7 Special Tasks **138**
7.1 Survival Analysis . . . 139
7.2 Spatial Analysis . . . 141
7.3 Ordinal Analysis . . . 141
7.4 Functional Analysis . . . 141
7.4.1 How to model functional data? . . . 141
7.5 Multilabel Classification . . . 141
7.6 Cost-Sensitive Classification . . . 141
7.6.1 A First Model . . . 143
7.6.2 Cost-sensitive Measure . . . 143
7.6.3 Thresholding . . . 144
7.6.4 Threshold Tuning . . . 146

8 Model Interpretation **147**
8.1 IML . . . 147
8.2 Dalex . . . 147

9 Use Cases **148**
9.1 House Price Prediction in King County . . . 148
9.1.1 Exploratory Data Analysis . . . 148
9.1.2 Splitting into train and test data . . . 155
9.1.3 A first model: Decision Tree . . . 155
9.1.4 A first baseline: Decision Tree . . . 158
9.1.5 Many Trees: Random Forest . . . 159
9.1.6 A better baseline: `AutoTuner` . . . 160
9.1.7 Engineering Features: Mutating ZIP-Codes . . . 161
9.1.8 Obtaining a sparser model . . . 162

10 Appendix **163**
10.1 Integrated Learners . . . 164
10.2 Integrated Performance Measures . . . 165
10.3 Integrated Filter Methods . . . 167
10.3.1 Standalone filter methods . . . 167
10.3.2 Algorithms With Embedded Filter Methods . . . 167

11 References **168**

Contents

Quickstart

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

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

# train a model of this learner for a subset of the task
learner$train(task, row_ids = 1:120)
# this is what the decision tree looks like
learner$model
## n= 120
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
## 1) root 120 70 setosa (0.41667 0.41667 0.16667)
##   2) Petal.Length< 2.45 50  0 setosa (1.00000 0.00000 0.00000) *
##   3) Petal.Length>=2.45 70 20 versicolor (0.00000 0.71429 0.28571)
##     6) Petal.Length< 4.95 49  1 versicolor (0.00000 0.97959 0.02041) *
##     7) Petal.Length>=4.95 21  2 virginica (0.00000 0.09524 0.90476) *

predictions = learner$predict(task, row_ids = 121:150)
predictions
## <PredictionClassif> for 30 observations:
##     row_id     truth   response
##        121 virginica  virginica
##        122 virginica versicolor
##        123 virginica  virginica
## ---
##        148 virginica  virginica
##        149 virginica  virginica
##        150 virginica  virginica
# accuracy of our model on the test set of the final 30
# rows
predictions$score(msr("classif.acc"))
## classif.acc
##      0.8333
```

The code examples in this book use a few additional packages that are not installed by default if you install mlr3. To run all examples in the book, install the `mlr-org/mlr3book` package using the `remotes` package:

```
remotes::install_github("mlr-org/mlr3book", dependencies = TRUE)
```

1 Introduction and Overview

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

Target Audience

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

Why a Rewrite?

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

Design Principles

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

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

mlr3 requires the following packages:

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

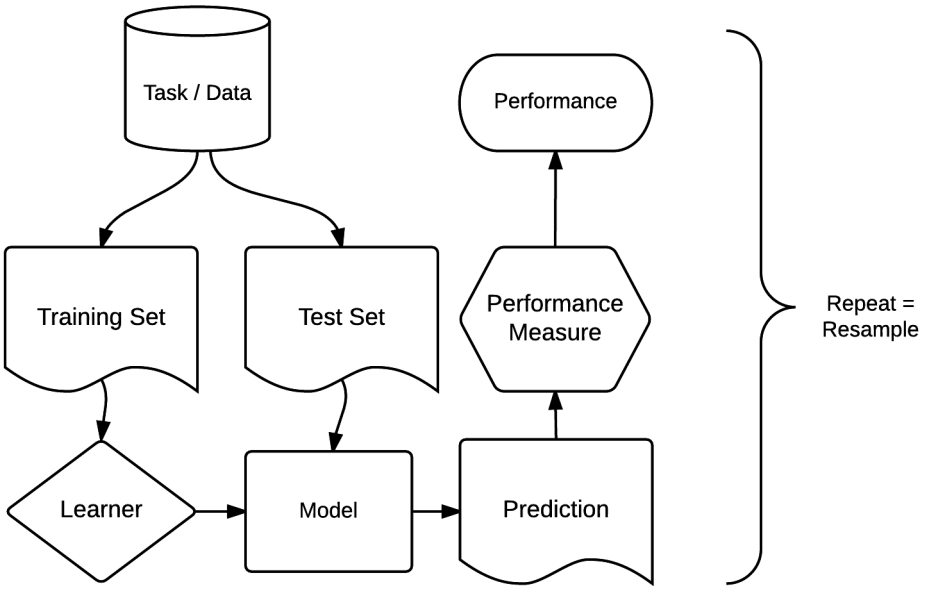
None of these packages adds any extra recursive dependencies to mlr3.

mlr3 provides additional functionality through extra packages:

- For parallelization, mlr3 utilizes the `future` and `future.apply` packages.
- To capture output, warnings, and exceptions, `evaluate` and `callr` can be used.

2 Basics

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



The data, which mlr3 encapsulates in tasks, is split into non-overlapping training and test sets. We are interested in models that generalize to new data rather than just memorizing the training data, and separate test data allows to objectively evaluate models with respect to that. The training data is given to a machine learning algorithm, which we call a learner in mlr3. The learner uses the training data to build a model of the relationship of in 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 does based on the difference between predicted and actual values. Usually this measure is a numeric score.

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

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

This chapter covers the following subtopics:

Tasks

Tasks encapsulate the data with meta-information, such as what the prediction target is. 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 collums 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 `mlr3::Task`. They are provided by R and other packages. We cover how to

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

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

Train and predict

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

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

Resampling

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

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

Additional information on resampling can be found in section about nested resampling 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 schemes. We cover how to

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

Binary classification

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

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

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

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

2.1 Quick R6 Intro for Beginners

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

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

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

2.2 Tasks

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

2.2.1 Task Types

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

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

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

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

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

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

Spatial Task: Observations in the task have spatio-temporal information (e.g. coordinates). ⟶ Not yet implemented, but started in add-on package mlr3spatiotempcv

2.2.2 Task Creation

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

First, we load and prepare the data.

```
data("mtcars", package = "datasets")
data = mtcars[, 1:3]
str(data)
## 'data.frame':    32 obs. of  3 variables:
##  $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
##  $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
##  $ disp: num  160 160 108 258 360 ...
```

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

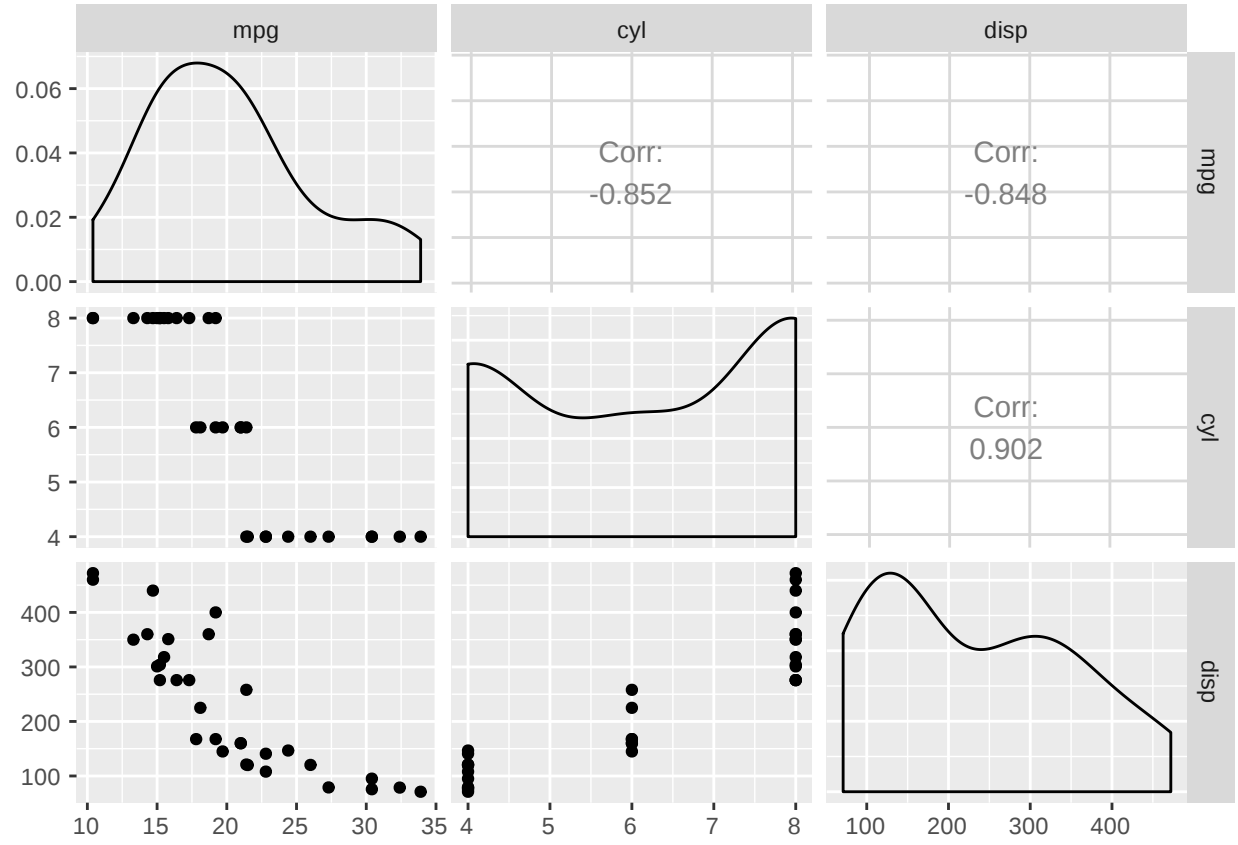
1. `id`: An arbitrary identifier for the task, used in plots and summaries.
2. `backend`: This parameter allows fine-grained control over how data is accessed. Here, we simply provide the dataset which is automatically converted to a `DataBackendDataTable`. We could also construct a `DataBackend` manually.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

```
library(mlr3)

task_mtcars = TaskRegr$new(id = "cars", backend = data, target = "mpg")
print(task_mtcars)
## <TaskRegr:cars> (32 x 3)
## * Target: mpg
## * Properties: -
## * Features (2):
##   - dbl (2): cyl, disp
```

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

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

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



2.2.3 Predefined tasks

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

```
mlr_tasks
## <DictionaryTask> with 12 stored values
## Keys: boston_housing, german_credit, iris, lung,
##   mtcars, pima, rats, sonar, spam, unemployment,
##   wine, zoo
```

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

```
library(data.table)
as.data.table(mlr_tasks)
##                key task_type nrow ncol lgl int dbl chr
##  1: boston_housing      regr  506   19   0   3  13   0
##  2:  german_credit   classif 1000   21   0   0   7   0
##  3:           iris   classif  150    5   0   0   4   0
##  4:           lung      surv  228   10   0   7   0   0
##  5:         mtcars      regr   32   11   0   0  10   0
##  6:           pima   classif  768    9   0   0   8   0
##  7:           rats      surv  300    5   0   2   0   0
```

```
##  8:          sonar   classif  208   61   0   0  60   0
##  9:           spam   classif 4601   58   0   0  57   0
## 10:   unemployment      surv 3343    6   0   1   2   0
## 11:           wine   classif  178   14   0   2  11   0
## 12:            zoo   classif  101   17  15   1   0   0
##     fct ord pxc
##  1:   2   0   0
##  2:  12   1   0
##  3:   0   0   0
##  4:   1   0   0
##  5:   0   0   0
##  6:   0   0   0
##  7:   1   0   0
##  8:   0   0   0
##  9:   0   0   0
## 10:   1   0   0
## 11:   0   0   0
## 12:   0   0   0
```

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

```
task_iris = mlr_tasks$get("iris")
print(task_iris)
## <TaskClassif:iris> (150 x 5)
## * Target: Species
## * Properties: multiclass
## * Features (4):
##   - dbl (4): Petal.Length, Petal.Width,
##     Sepal.Length, Sepal.Width
```

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

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

2.2.4 Task API

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

2.2.4.1 Retrieving Data

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

```
task_iris$nrow
## [1] 150
task_iris$ncol
## [1] 5
```

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

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

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

The *iris* task uses integer `row_ids`:

```
# iris uses integer row_ids
head(task_iris$row_ids)
## [1] 1 2 3 4 5 6

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

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

The *mtcars* task on the other hand uses names for its `row_ids`, encoded as `character`:

```
task_mtcars = tsk("mtcars")
head(task_mtcars$row_ids)
## [1] "AMC Javelin"        "Cadillac Fleetwood"
## [3] "Camaro Z28"         "Chrysler Imperial"
## [5] "Datsun 710"         "Dodge Challenger"

# retrieve data for rows with id 'Datsun 710'
task_mtcars$data(rows = "Datsun 710")
##     mpg am carb cyl disp drat gear hp  qsec vs   wt
## 1: 22.8  1    1   4  108 3.85    4 93 18.61  1 2.32
```

Note that the method `$data()` only allows to read the data and does not modify it.

Similarly, each column has an identifier or name. These names are stored in the public slots `feature_names` and `target_names`. Here "target" refers to the variable we want to predict and "feature" to the predictors for the task.

```
task_iris$feature_names
## [1] "Petal.Length" "Petal.Width"  "Sepal.Length"
## [4] "Sepal.Width"
task_iris$target_names
## [1] "Species"
```

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

```
# retrieve data for rows 1, 51, and 101 and only select
# column 'Species'
task_iris$data(rows = c(1, 51, 101), cols = "Species")
##       Species
## 1:     setosa
## 2: versicolor
## 3:  virginica
```

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

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

2.2.4.2 Roles (Rows and Columns)

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

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

```
print(task_mtcars$col_roles)
## $feature
##  [1] "am"   "carb" "cyl"  "disp" "drat" "gear" "hp"
##  [8] "qsec" "vs"   "wt"
##
## $target
## [1] "mpg"
##
## $name
## character(0)
##
## $order
## character(0)
##
## $stratum
## character(0)
##
## $group
## character(0)
##
## $weight
## character(0)
```

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

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

# we now have integer row_ids
task$row_ids
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
## [18] 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32

# there is a new feature called 'rn'
task$feature_names
## [1] "cyl"  "disp" "rn"
```

The row names are now a feature whose values are stored in the column "rn". We include this column here for educational purposes only. Generally speaking, there is no point in having a feature that uniquely identifies each row. Furthermore, the character data type will cause problems with many types of machine learning algorithms. The identifier may be useful to label points in plots and identify outliers however. To use the new column for only this purpose, we will change the role of the "rn" column and remove it from the set of active features.

```
task$feature_names
## [1] "cyl"  "disp" "rn"

# working with a list of column vectors
task$col_roles$name = "rn"
task$col_roles$feature = setdiff(task$col_roles$feature,
  "rn")

# 'rn' not listed as feature anymore
task$feature_names
## [1] "cyl"  "disp"

# does also not appear when we access the data anymore
task$data(rows = 1:2)
##    mpg cyl disp
## 1:  21   6  160
## 2:  21   6  160
task$head(2)
##    mpg cyl disp
## 1:  21   6  160
## 2:  21   6  160
```

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

Just like columns, it is also possible to assign different roles to rows. Rows can have two different roles:

1. Role `use` Rows that are generally available for model fitting (although they may also be used as test set in resampling). This role is the default role.

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

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

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

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

2.2.4.3 Task Mutators

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

```
task = tsk("iris")
task$select(c("Sepal.Width", "Sepal.Length"))  # keep only these features
task$filter(1:3)  # keep only these rows
task$head()
##    Species Sepal.Length Sepal.Width
## 1:  setosa          5.1         3.5
## 2:  setosa          4.9         3.0
## 3:  setosa          4.7         3.2
```

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

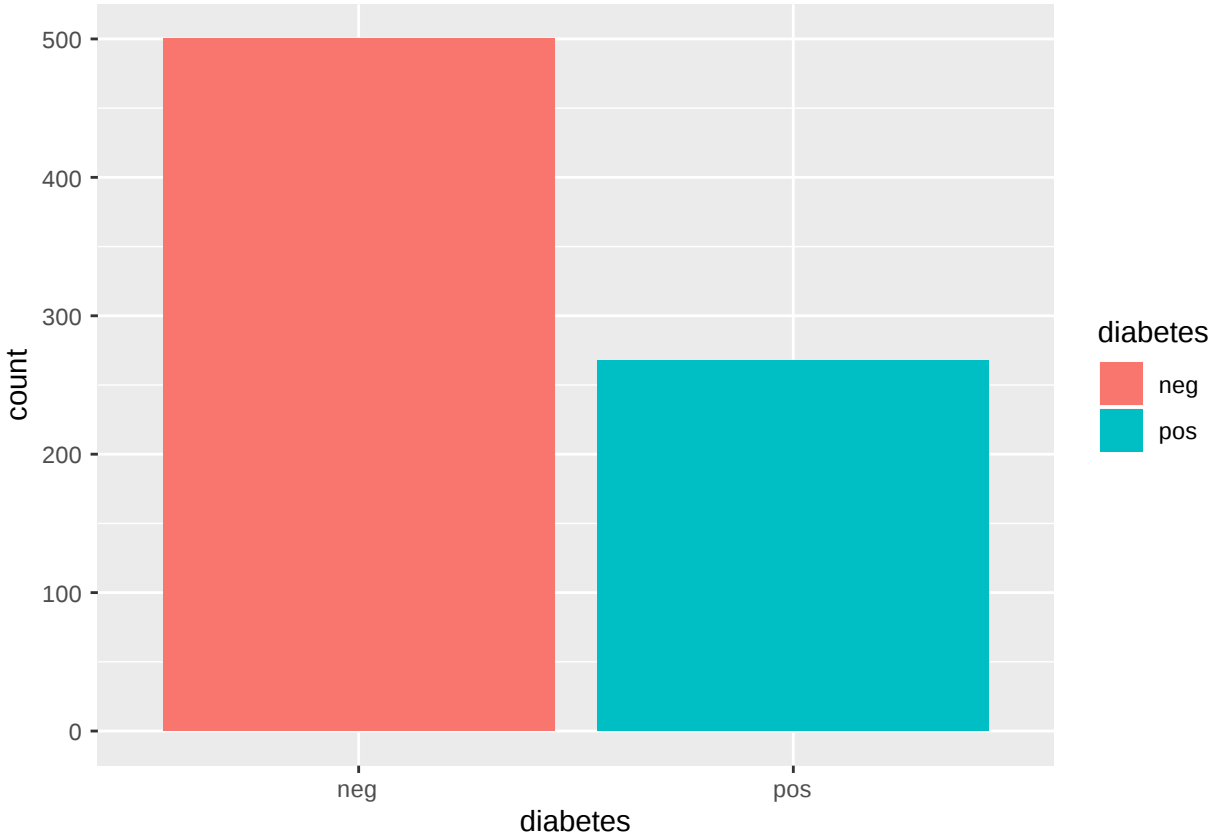
```
task$cbind(data.table(foo = letters[1:3]))  # add column foo
task$head()
##    Species Sepal.Length Sepal.Width foo
## 1:  setosa          5.1         3.5   a
## 2:  setosa          4.9         3.0   b
## 3:  setosa          4.7         3.2   c
```

2.2.5 Plotting Tasks

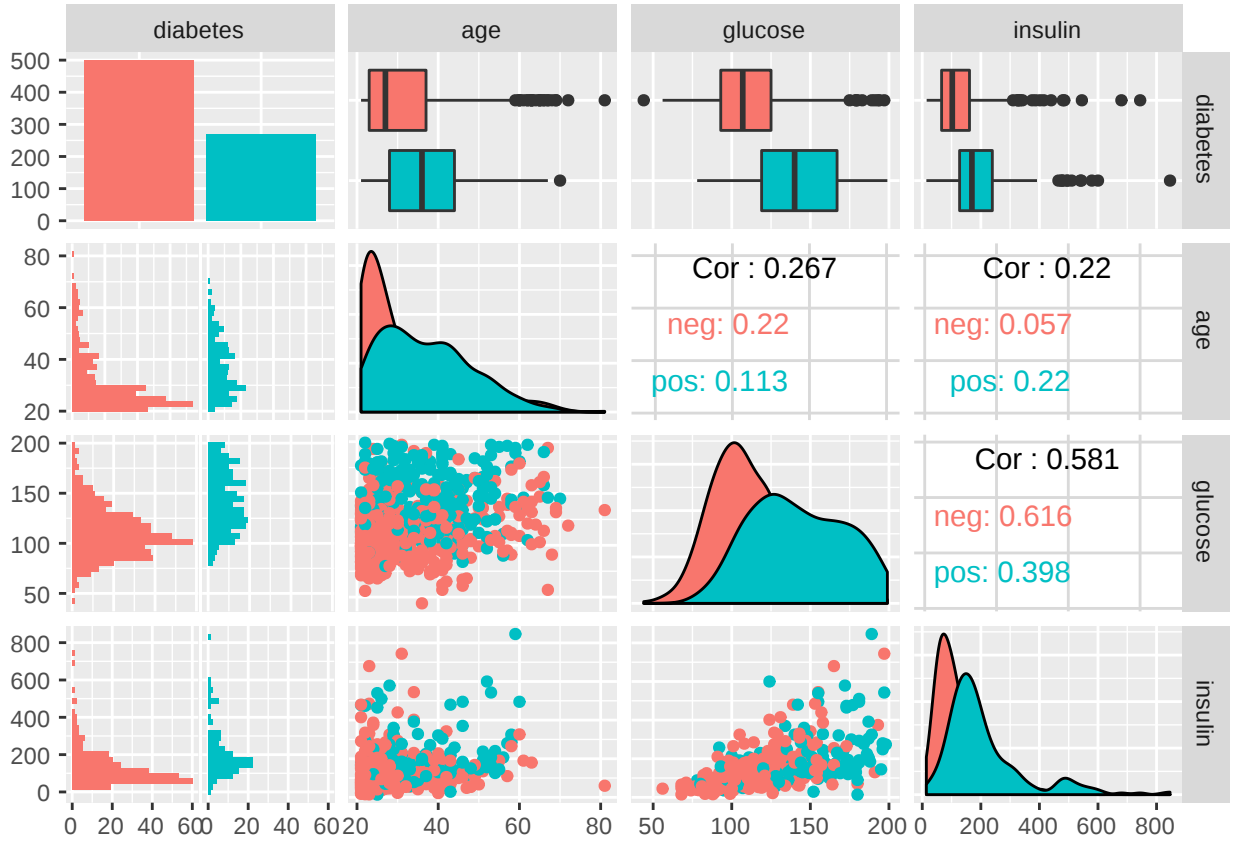
The mlr3viz provides plotting facilities for many classes implemented in mlr3. The types of plot depend on the inherited class, but all plots are returned as `ggplot2` objects which can be easily customized.

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

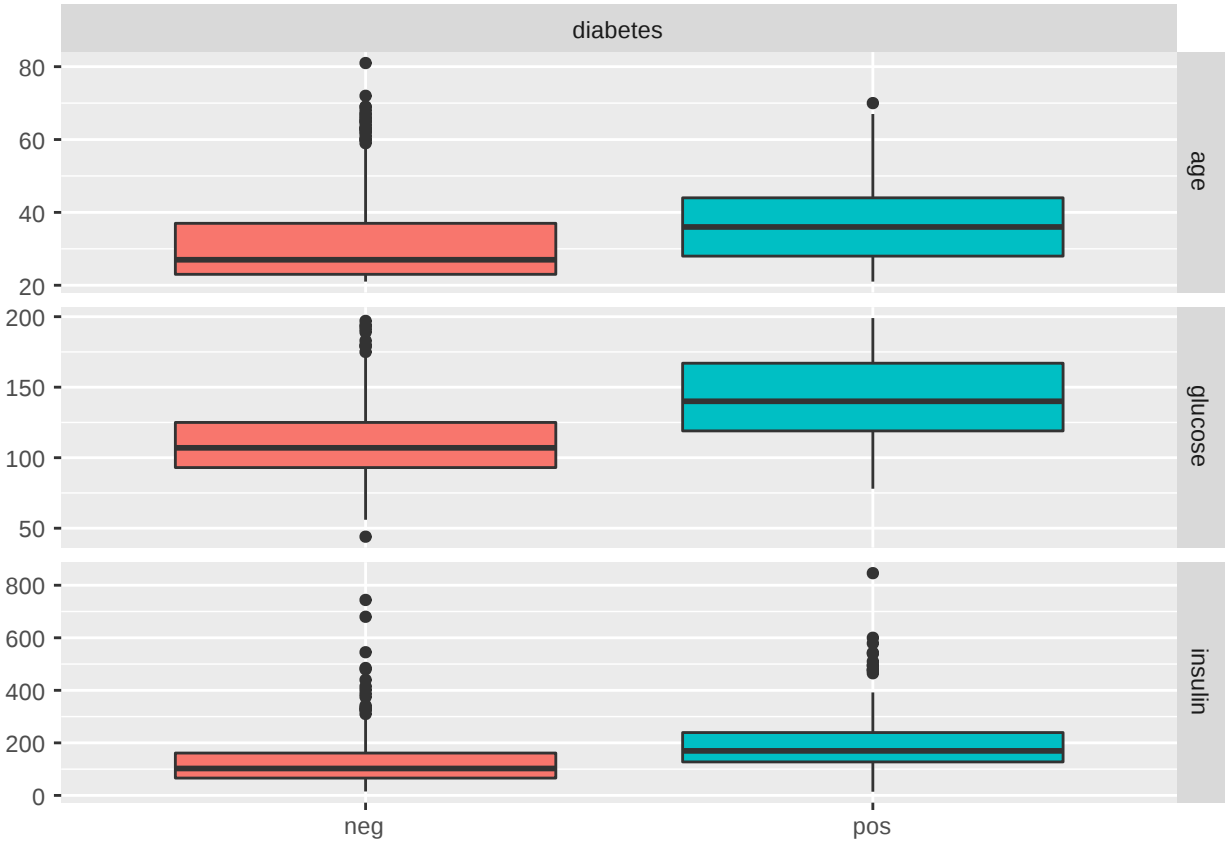
```
library(mlr3viz)

# 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")
```

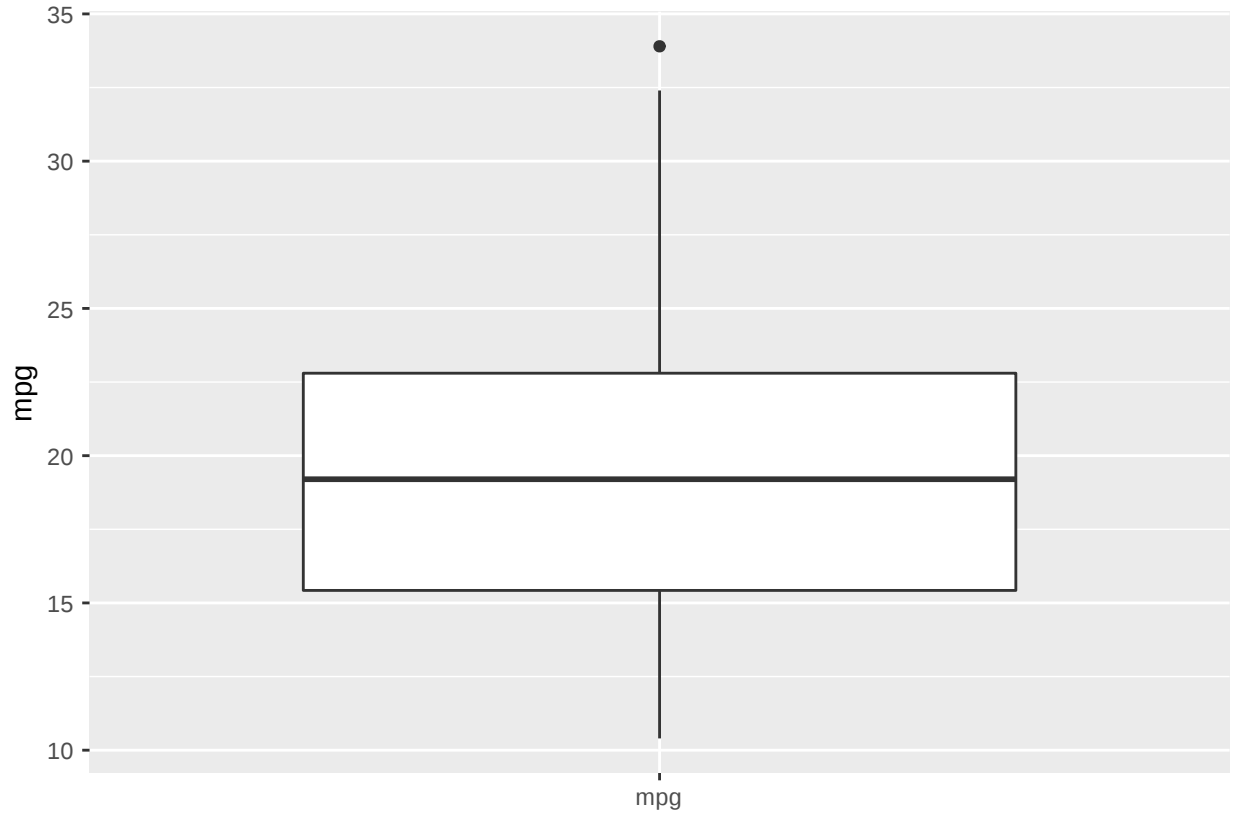



```
# duo plot (requires package GGally)
autoplot(task, type = "duo")
```

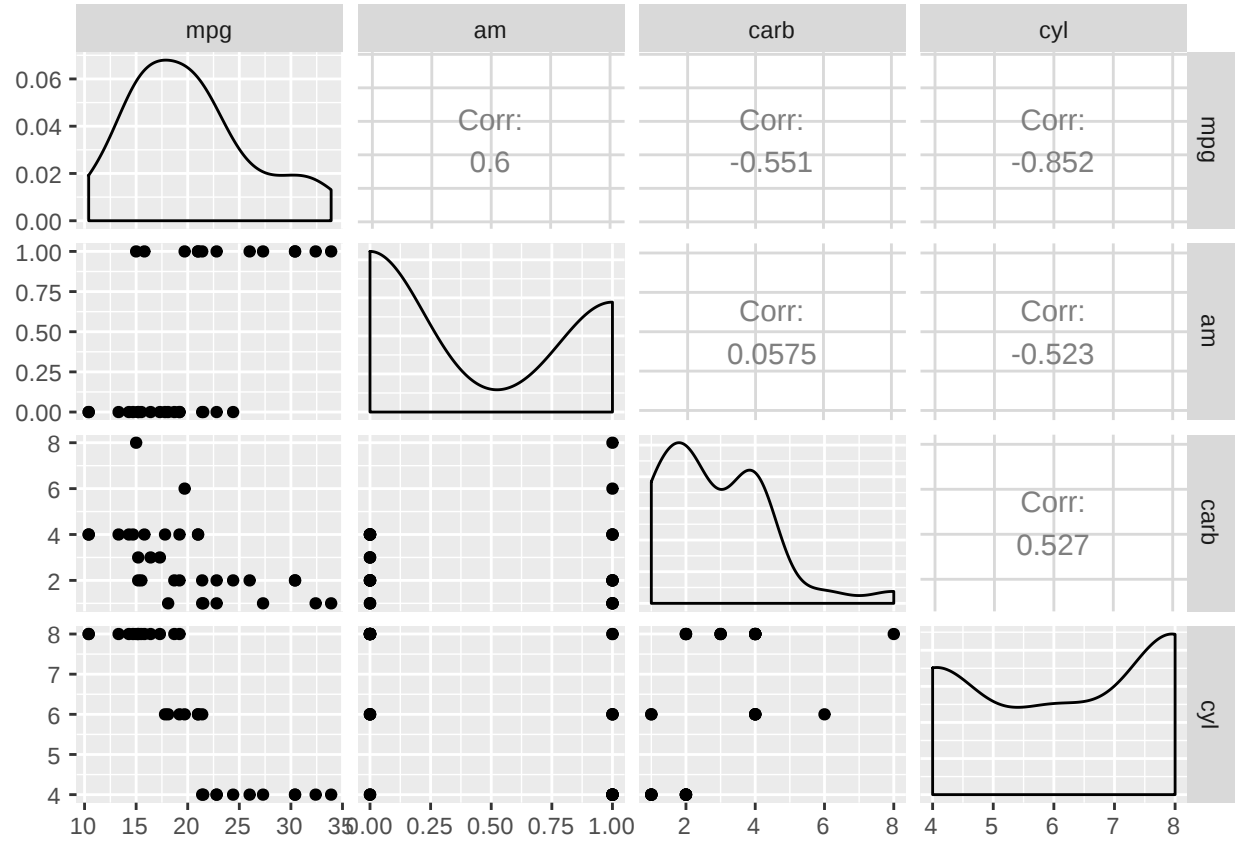


Of course, you can do the same for regression tasks (inheriting from `TaskRegr`) as documented in `mlr3viz::autoplot.TaskRegr`:

```
library(mlr3viz)

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

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

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

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

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

More learners are collected on GitHub in the mlr3learners organization. Their state is also listed on the wiki of the mlr3learners repository.

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

2.3.1 Predefined Learners

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

```
library(mlr3learners)
mlr_learners
## <DictionaryLearner> with 38 stored values
## Keys: classif.debug, classif.featureless,
##   classif.glmnet, classif.kknn, classif.lda,
##   classif.log_reg, classif.naive_bayes,
##   classif.qda, classif.ranger, classif.rpart,
##   classif.svm, classif.xgboost, regr.featureless,
##   regr.glmnet, regr.kknn, regr.km, regr.lm,
##   regr.ranger, regr.rpart, regr.svm,
##   regr.xgboost, surv.blackboost, surv.coxph,
##   surv.cvglmnet, surv.flexible, surv.gamboost,
##   surv.gbm, surv.glmboost, surv.glmnet,
##   surv.kaplan, surv.mboost, surv.nelson,
##   surv.parametric, surv.penalized,
##   surv.randomForestSRC, surv.ranger, surv.rpart,
##   surv.svm
```

Each learner has the following information:

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

For a tabular overview of integrated learners, see Section 10.1.

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

```
learner = mlr_learners$get("classif.rpart")
print(learner)
## <LearnerClassifRpart:classif.rpart>
## * Model: -
## * Parameters: xval=0
```

```
## * Packages: rpart
## * Predict Type: response
## * Feature types: logical, integer, numeric,
##   factor, ordered
## * Properties: importance, missings, multiclass,
##   selected_features, twoclass, weights
```

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

```
learner$param_set
## ParamSet:
##              id    class lower upper levels default
## 1:     minsplit ParamInt     1   Inf             20
## 2:           cp ParamDbl     0     1           0.01
## 3:   maxcompete ParamInt     0   Inf              4
## 4: maxsurrogate ParamInt     0   Inf              5
## 5:     maxdepth ParamInt     1    30             30
## 6:         xval ParamInt     0   Inf             10
##    value
## 1:
## 2:
## 3:
## 4:
## 5:
## 6:     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: rpart
## * Predict Type: response
## * Feature types: logical, integer, numeric,
##   factor, ordered
## * Properties: importance, missings, multiclass,
##   selected_features, twoclass, weights
```

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

```
learner$param_set$values = mlr3misc::insert_named(learner$param_set$values,
  list(cp = 0.02, minsplit = 2))
learner
## <LearnerClassifRpart:classif.rpart>
## * Model: -
## * Parameters: cp=0.02, xval=0, minsplit=2
## * Packages: rpart
## * Predict Type: response
## * Feature types: logical, integer, numeric,
##   factor, ordered
## * Properties: importance, missings, multiclass,
##   selected_features, twoclass, weights
```

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

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

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

If you pass hyperparameters here, it is added to the default parameters in a `mlr3misc::insert_named()", text = "insert_named()`-fashion.

For further information on how to customize learners using mlr3, see the section on extending learners.

2.4 Train and Predict

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

Training a learner means fitting a model to a given data set. Subsequently, we want to predict the target value for new observations. These predictions are compared to the ground truth values to assess the quality of the model. In sum, the goal of training and predicting is to evaluate the predictive power of different models.

2.4.1 Creating Task and Learner Objects

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

1. The classification task:

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

2. A learner for the classification tree:

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

2.4.2 Setting up the train/test splits of the data

It is common to train on a majority of the data. Here we use 80% of all available observations and predict on the remaining 20% observations. 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)
```

2.4.3 Training the learner

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

```
learner$model
## NULL
```

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

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

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

```
print(learner$model)
## n= 166
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
##  1) root 166 74 M (0.55422 0.44578)
##    2) V11>=0.198 96 20 M (0.79167 0.20833)
##      4) V17< 0.5512 58  4 M (0.93103 0.06897) *
##      5) V17>=0.5512 38 16 M (0.57895 0.42105)
##       10) V23>=0.7846 16  1 M (0.93750 0.06250) *
```

```
##       11) V23< 0.7846 22  7 R (0.31818 0.68182)
##         22) V41< 0.1595 7  1 M (0.85714 0.14286) *
##         23) V41>=0.1595 15  1 R (0.06667 0.93333) *
##    3) V11< 0.198 70 16 R (0.22857 0.77143)
##      6) V4>=0.05095 18  7 M (0.61111 0.38889) *
##      7) V4< 0.05095 52  5 R (0.09615 0.90385) *
```

2.4.4 Predicting

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

```
prediction = learner$predict(task, row_ids = test_set)
print(prediction)
## <PredictionClassif> for 42 observations:
##     row_id truth response
##          1     R        R
##          6     R        R
##         13     R        R
## ---
##        197     M        M
##        203     M        M
##        204     M        M
```

The `$predict()` method of the `Learner` returns a `Prediction` object. More precise, as the learner is specialized for classification, 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 to a `data.table()`:

```
head(as.data.table(prediction))
##    row_id truth response
## 1:      1     R        R
## 2:      6     R        R
## 3:     13     R        R
## 4:     19     R        R
## 5:     21     R        M
## 6:     25     R        R
```

For classification, you can also extract the confusion matrix:

```
prediction$confusion
##         truth
## response  M  R
##        M 17  9
##        R  2 14
```

2.4.5 Changing the Predict Type

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

```
learner$predict_type = "prob"

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

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

The prediction object now contains probabilities for all class labels:

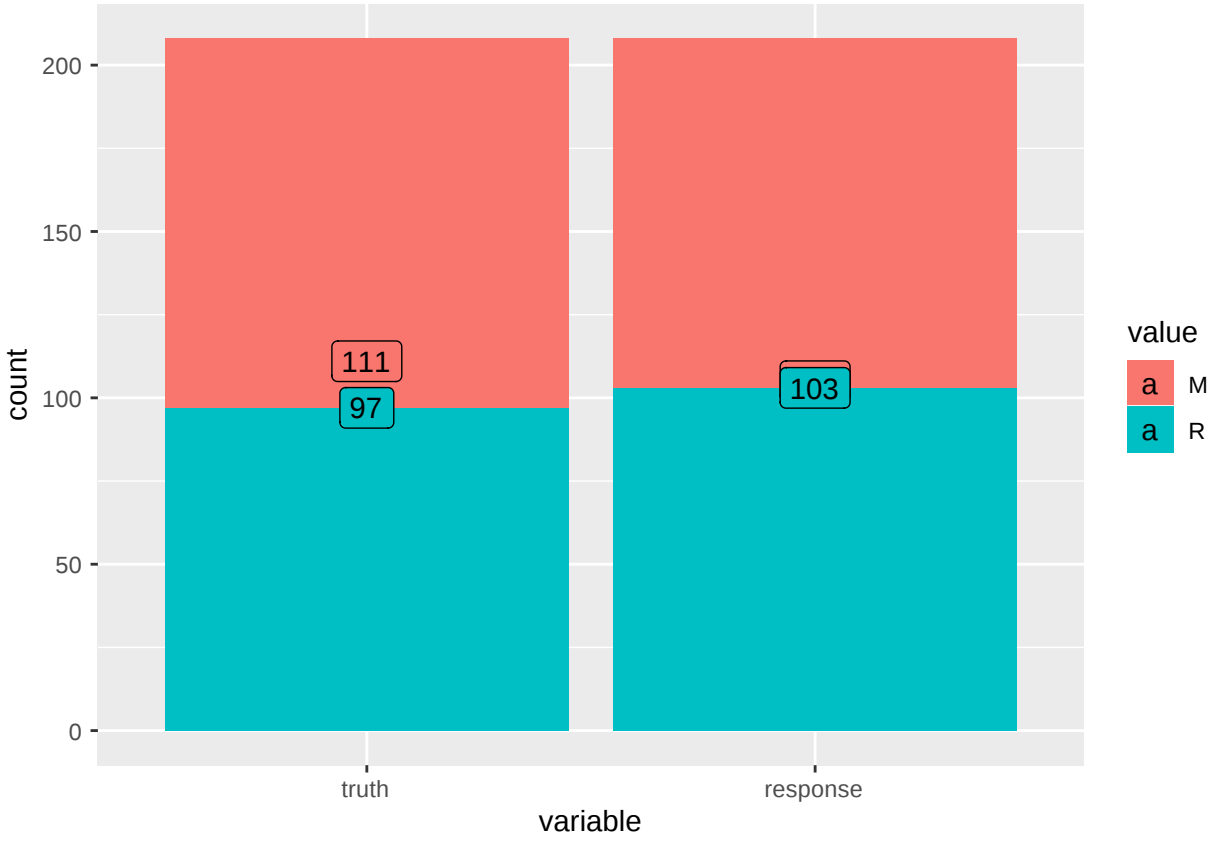
```
# data.table conversion
head(as.data.table(prediction))
##    row_id truth response  prob.M  prob.R
## 1:      1     R        R 0.09615 0.90385
## 2:      6     R        R 0.06667 0.93333
## 3:     13     R        R 0.09615 0.90385
## 4:     19     R        R 0.09615 0.90385
## 5:     21     R        M 0.93103 0.06897
## 6:     25     R        R 0.09615 0.90385

# directly access the predicted labels:
head(prediction$response)
## [1] R R R R M R
## Levels: M R

# directly access the matrix of probabilities:
head(prediction$prob)
##            M       R
## [1,] 0.09615 0.90385
## [2,] 0.06667 0.93333
## [3,] 0.09615 0.90385
## [4,] 0.09615 0.90385
## [5,] 0.93103 0.06897
## [6,] 0.09615 0.90385
```

Analogously to predicting probabilities, many `LearnerRegr", text = "regression learners` support the extracting of a standard error estimates by setting the predict type to `"se"`.

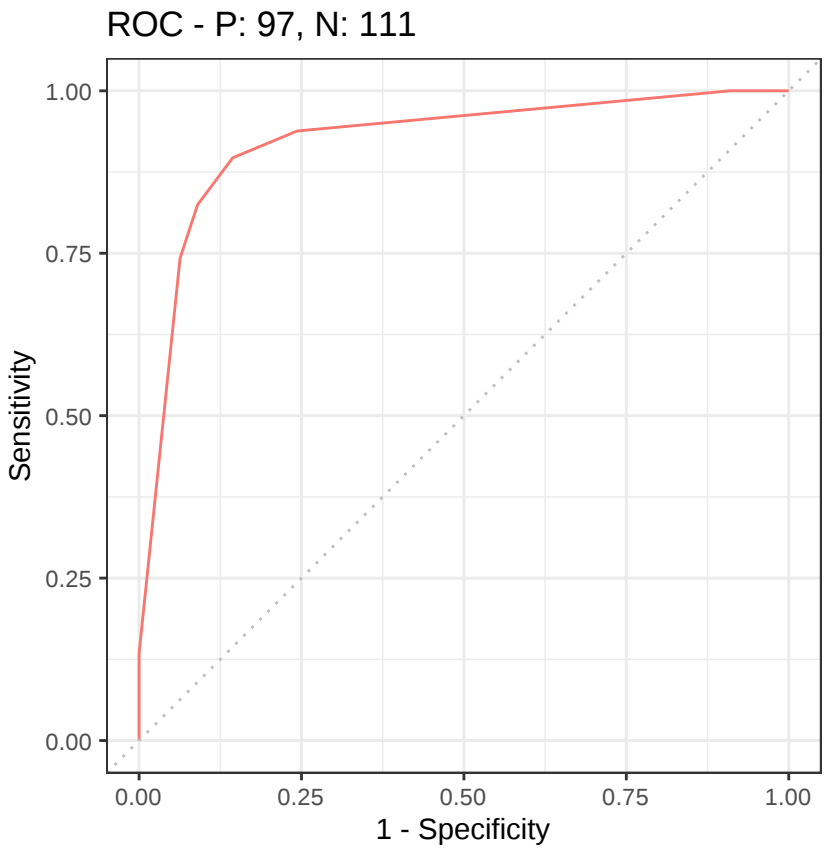
2.4.6 Plotting Predictions

Analogously to plotting tasks, mlr3viz provides a `ggplot2::autoplot()", text = "autoplot()` method. All available types are listed on the manual page of `autoplot.PredictionClassif()` or `autoplot.PredictionClassif()`, respectively.

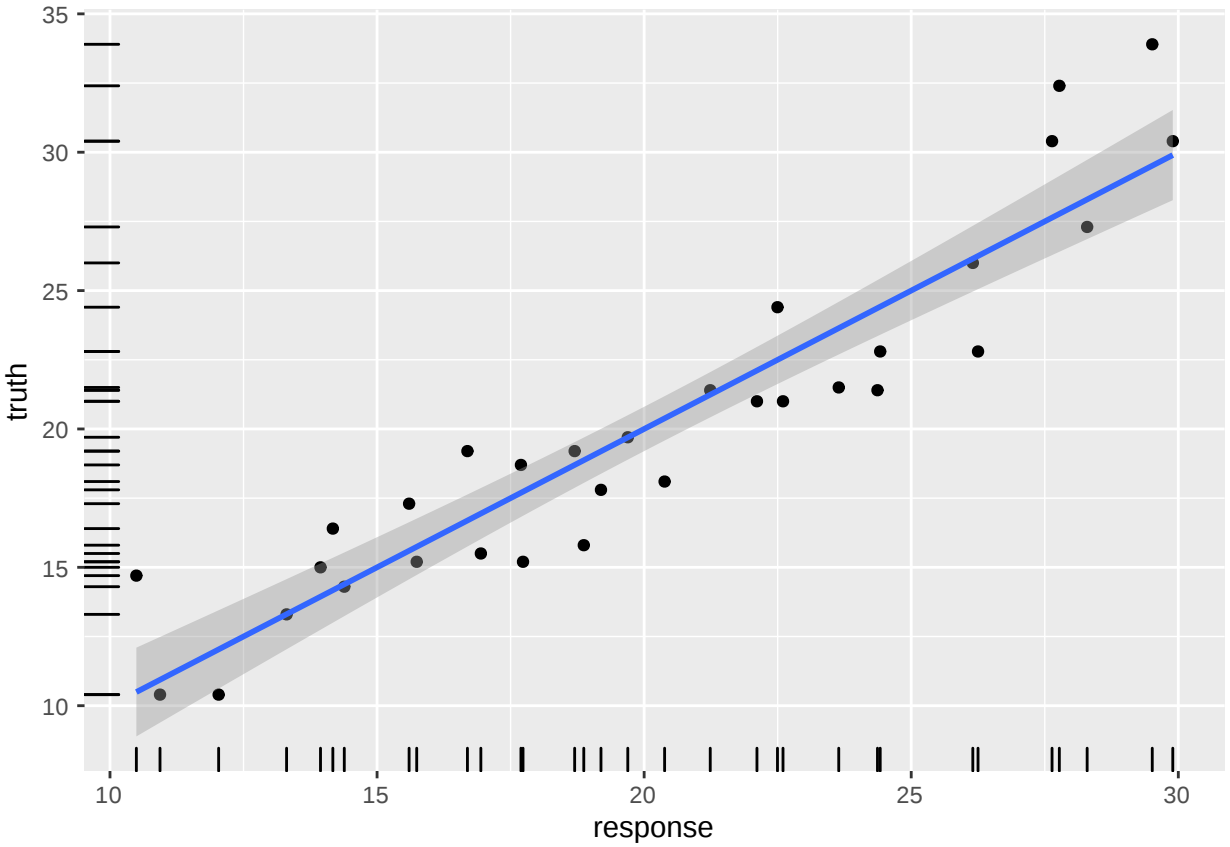
```
library(mlr3viz)

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




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




```
library(mlr3viz)
library(mlr3learners)
local({
  # we do this locally to not overwrite the objects from
  # previous chunks
  task = tsk("mtcars")
  learner = lrn("regr.lm")
  learner$train(task)
  prediction = learner$predict(task)
  autoplot(prediction)
})
```



2.4.7 Performance assessment

The last step of an modeling is usually the performance assessment. The quality of the predictions of a model in mlr3 can be assessed with respect to a number of different performance measures. At the performance assessment we choose a specific performance measure to quantify the predictions. This is done by comparing the predicted labels with the true labels. Predefined available measures are stored in `mlr_measures` (with convenience getter `msr()`):

```
mlr_measures
## <DictionaryMeasure> with 72 stored values
## Keys: classif.acc, classif.auc, classif.bacc,
##   classif.ce, classif.costs, classif.dor,
##   classif.fbeta, classif.fdr, classif.fn,
##   classif.fnr, classif.fomr, classif.fp,
##   classif.fpr, classif.logloss, classif.mcc,
##   classif.npv, classif.ppv, classif.precision,
##   classif.recall, classif.sensitivity,
##   classif.specificity, classif.tn, classif.tnr,
##   classif.tp, classif.tpr, debug, oob_error,
##   regr.bias, regr.ktau, regr.mae, regr.mape,
##   regr.maxae, regr.medae, regr.medse, regr.mse,
##   regr.msle, regr.pbias, regr.rae, regr.rmse,
##   regr.rmsle, regr.rrse, regr.rse, regr.rsq,
##   regr.sae, regr.smape, regr.srho, regr.sse,
##   selected_features, surv.beggC,
```

```
##   surv.chamblessAUC, surv.gonenC, surv.graf,
##   surv.grafSE, surv.harrellC, surv.hungAUC,
##   surv.intlogloss, surv.intloglossSE,
##   surv.logloss, surv.loglossSE, surv.nagelkR2,
##   surv.oquigleyR2, surv.songAUC, surv.songTNR,
##   surv.songTPR, surv.unoAUC, surv.unoC,
##   surv.unoTNR, surv.unoTPR, surv.xuR2, time_both,
##   time_predict, time_train
```

We select the accuracy (`mlr_measures_classif.acc", text = "classif.acc`) and call the method `$score()` of the `Prediction` object.

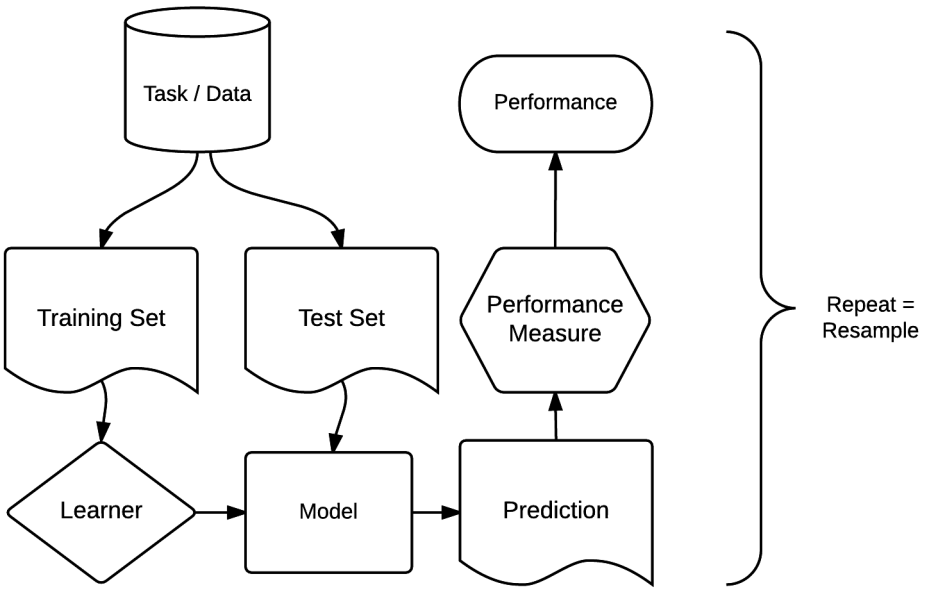
```
measure = msr("classif.acc")
prediction$score(measure)
## classif.acc
##       0.875
```

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

2.5 Resampling

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

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



2.5.1 Settings

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

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

When performing resampling with a dataset, we first need to define which approach should be used. The resampling strategies of *mlr3* can be queried using the `.$keys()` method of the `mlr_resamplings` dictionary.

```
mlr_resamplings
## <DictionaryResampling> with 6 stored values
## Keys: bootstrap, custom, cv, holdout,
##   repeated_cv, subsampling
```

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

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

```
resampling = rsmp("holdout")
print(resampling)
## <ResamplingHoldout> with 1 iterations
## * Instantiated: FALSE
## * Parameters: ratio=0.6667
```

Note that the `Instantiated` field is set to `FALSE`. This means we did not actually apply the strategy on a dataset yet, but just performed a dry-run. Applying the strategy on a dataset is done in section next Instantiation.

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

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

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

2. Specifying the resampling parameters directly during construction:

```
rsmp("holdout", ratio = 0.8)
## <ResamplingHoldout> with 1 iterations
## * Instantiated: FALSE
## * Parameters: ratio=0.8
```

2.5.2 Instantiation

So far we just set the stage and selected the resampling strategy. To actually perform the splitting, the resampling needs a `Task`. By calling the method `instantiate()`, splits into training and test set are calculated and stored in the `Resampling` object:

```
resampling = rsmp("cv", folds = 3L)
resampling$instantiate(task)
resampling$iters
## [1] 3
str(resampling$train_set(1))
##  int [1:100] 1 7 9 11 12 15 16 17 18 19 ...
str(resampling$test_set(1))
##  int [1:50] 3 10 20 21 26 27 28 29 39 43 ...
```

2.5.3 Execution

With a `Task`, a `Learner` and `Resampling` object we can call `resample()` and create a `ResampleResult` object.

Before we go into more detail, let's change the resampling to a "3-fold cross-validation" to better illustrate what operations are possible with a `ResampleResult`. Additionally, we tell `resample()` to keep the fitted models via the flag `store_models`:

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

rr = resample(task, learner, resampling, store_models = TRUE)
print(rr)
## <ResampleResult> of 3 iterations
## * Task: pima
## * Learner: classif.rpart
## * Warnings: 0 in 0 iterations
## * Errors: 0 in 0 iterations
```

The following operations are supported with `ResampleResult` objects:

- Calculate the average performance:

  ```
  rr$aggregate(msr("classif.ce"))
  ## classif.ce
  ##     0.2617
  ```

- Extract the performance for the individual resampling iterations:

```
rr$score(msr("classif.ce"))
##             task task_id               learner
## 1: <TaskClassif>    pima <LearnerClassifRpart>
## 2: <TaskClassif>    pima <LearnerClassifRpart>
## 3: <TaskClassif>    pima <LearnerClassifRpart>
##       learner_id     resampling resampling_id iteration
## 1: classif.rpart <ResamplingCV>            cv         1
## 2: classif.rpart <ResamplingCV>            cv         2
## 3: classif.rpart <ResamplingCV>            cv         3
##    prediction classif.ce
## 1:     <list>     0.2539
## 2:     <list>     0.2734
## 3:     <list>     0.2578
```

- Check for warnings or errors:

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

- Extract and inspect the resampling splits:

```
rr$resampling
## <ResamplingCV> with 3 iterations
## * Instantiated: TRUE
## * Parameters: folds=3
rr$resampling$iters
## [1] 3
str(rr$resampling$test_set(1))
##  int [1:256] 2 6 15 16 17 19 24 27 39 42 ...
str(rr$resampling$train_set(1))
##  int [1:512] 1 3 4 5 7 10 12 18 23 25 ...
```

- Retrieve the learner of a specific iteration and inspect it:

```
lrn = rr$learners[[1]]
lrn$model
## n= 512
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
##  1) root 512 175 neg (0.6582 0.3418)
##    2) glucose< 127.5 331  63 neg (0.8097 0.1903) *
##    3) glucose>=127.5 181  69 pos (0.3812 0.6188)
##      6) mass< 29.95 49  13 neg (0.7347 0.2653)
##       12) glucose< 154.5 35   5 neg (0.8571 0.1429) *
```

```
##       13) glucose>=154.5 14   6 pos (0.4286 0.5714) *
##      7) mass>=29.95 132  33 pos (0.2500 0.7500) *
```

- Extract the predictions:

```
rr$prediction()  # all predictions merged into a single Prediction
## <PredictionClassif> for 768 observations:
##     row_id truth response prob.pos prob.neg
##          2   neg      neg   0.1903   0.8097
##          6   neg      neg   0.1903   0.8097
##         15   pos      pos   0.5714   0.4286
## ---
##        762   pos      pos   0.8889   0.1111
##        763   neg      neg   0.1763   0.8237
##        767   pos      pos   0.5840   0.4160
rr$predictions()[[1]]  # prediction of first resampling iteration
## <PredictionClassif> for 256 observations:
##     row_id truth response prob.pos prob.neg
##          2   neg      neg   0.1903   0.8097
##          6   neg      neg   0.1903   0.8097
##         15   pos      pos   0.5714   0.4286
## ---
##        758   pos      neg   0.1903   0.8097
##        759   neg      neg   0.1903   0.8097
##        765   neg      neg   0.1903   0.8097
```

Note that if you want to compare multiple learners, you should ensure to that each learner operates on the same resampling instance by manually instantiating beforehand. This reduces the variance of the performance estimation.

If your aim is to compare different `Task`, `Learner` or `Resampling`, you are better off using the `benchmark()` function, covered in the next section. It is basically a wrapper around `resample()` simplifying the handling of multiple settings. If you discover this only after you've run multiple `resample()` calls, don't worry. You can combine multiple `ResampleResult` objects into a `BenchmarkResult` (also explained in the next section).

2.5.4 Custom resampling

Sometimes it is necessary to perform resampling with custom splits. If you want to do that because you are coming from a specific modeling field, first take a look at the *mlr3* extension packages. It is important to make sure that your custom resampling method has not been implemented already.

If your custom resampling method is widely used in your field, feel welcome to integrate it into one of the existing *mlr3* extension packages. You could also create your own extension package.

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

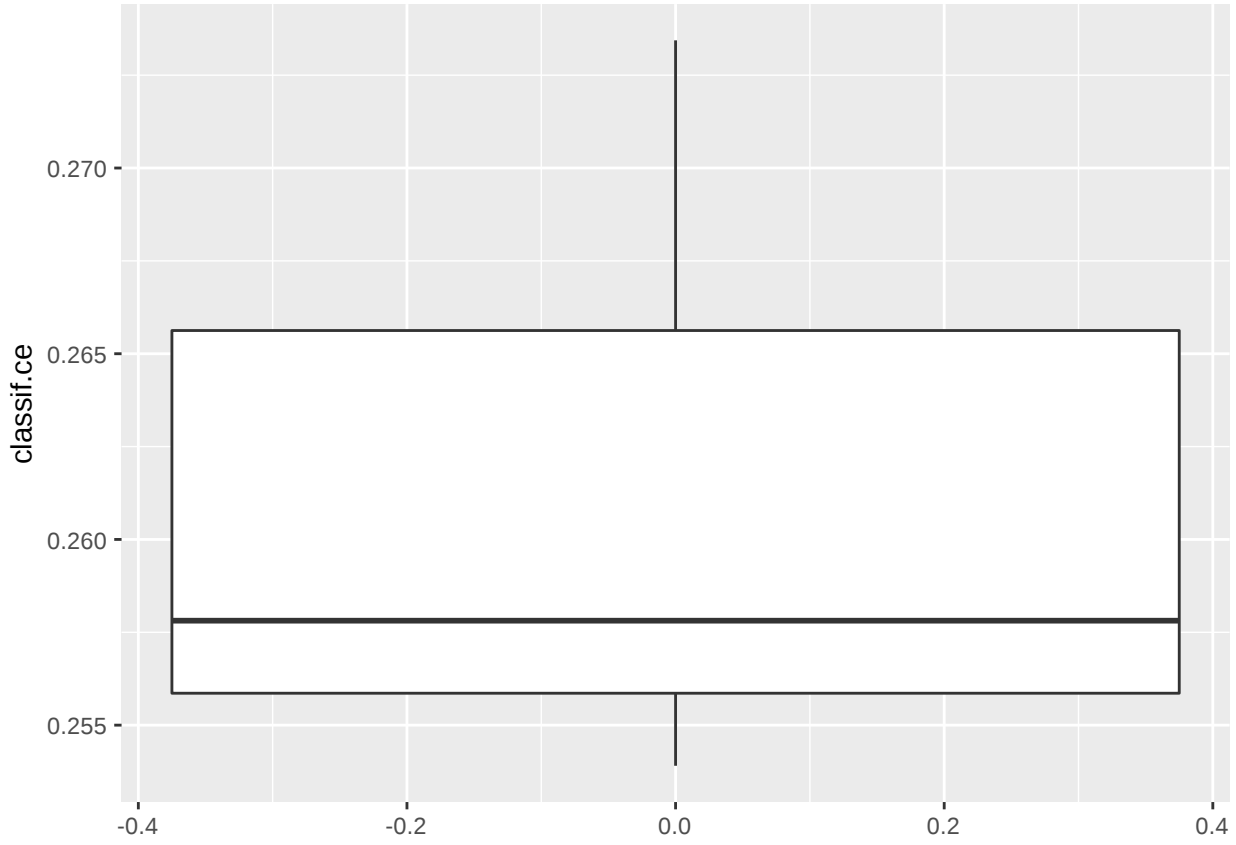
```
resampling = rsmp("custom")
resampling$instantiate(task, train = list(c(1:10, 51:60,
  101:110)), test = list(c(11:20, 61:70, 111:120)))
resampling$iters
```

```
## [1] 1
resampling$train_set(1)
##  [1]   1   2   3   4   5   6   7   8   9  10  51  52  53
## [14]  54  55  56  57  58  59  60 101 102 103 104 105 106
## [27] 107 108 109 110
resampling$test_set(1)
##  [1]  11  12  13  14  15  16  17  18  19  20  61  62  63
## [14]  64  65  66  67  68  69  70 111 112 113 114 115 116
## [27] 117 118 119 120
```

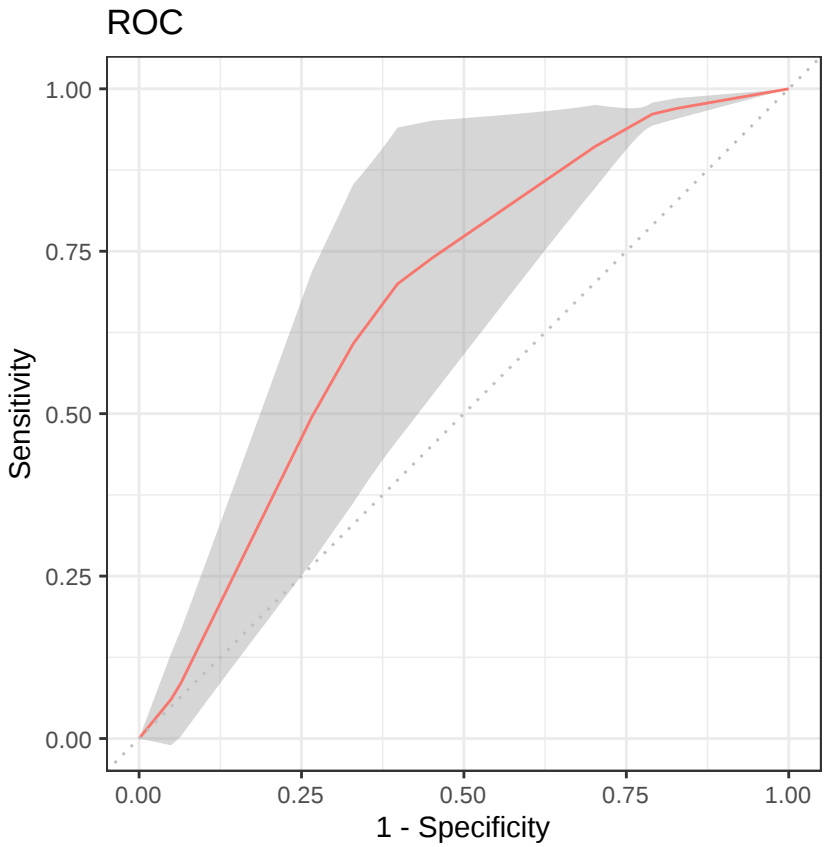
2.5.5 Plotting Resample Results

Again, mlr3viz provides a `ggplot2::autoplot()", text = "autoplot()` method.

```
library(mlr3viz)

autoplot(rr)
```




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



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

2.6 Benchmarking

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

2.6.1 Design Creation

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

Here, we call `benchmark()` to perform a single holdout split on a single task and two learners. We use the `benchmark_grid()` function to create an exhaustive design and properly instantiate the resampling:

```
library(data.table)
design = benchmark_grid(tasks = tsk("iris"), learners = list(lrn("classif.rpart"),
  lrn("classif.featureless")), resamplings = rsmp("holdout"))
print(design)
##             task                     learner
## 1: <TaskClassif>       <LearnerClassifRpart>
## 2: <TaskClassif> <LearnerClassifFeatureless>
```

```
##             resampling
## 1: <ResamplingHoldout>
## 2: <ResamplingHoldout>
bmr = benchmark(design)
```

Note that the holdout splits have been automatically instantiated for each row of the design. As a result, the `rpart` learner used a different training set than the `featureless` learner. However, for comparison of learners you usually want the learners to see the same splits into train and test sets. To overcome this issue, the resampling strategy needs to be **manually instantiated** before creating the design.

The interface of `benchmark()` allows for full flexibility. Irrespective, the creation of such design tables can be tedious. Therefore, mlr3 provides a convenience function to quickly generate design tables and instantiate resampling strategies in an exhaustive grid fashion: `benchmark_grid()`.

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

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

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

# create a BenchmarkDesign object
design = benchmark_grid(tasks, learners, resamplings)
print(design)
##             task                     learner
## 1: <TaskClassif> <LearnerClassifFeatureless>
## 2: <TaskClassif>       <LearnerClassifRpart>
## 3: <TaskClassif>      <LearnerClassifRanger>
## 4: <TaskClassif>        <LearnerClassifKKNN>
## 5: <TaskClassif> <LearnerClassifFeatureless>
## 6: <TaskClassif>       <LearnerClassifRpart>
## 7: <TaskClassif>      <LearnerClassifRanger>
## 8: <TaskClassif>        <LearnerClassifKKNN>
##        resampling
## 1: <ResamplingCV>
## 2: <ResamplingCV>
## 3: <ResamplingCV>
## 4: <ResamplingCV>
## 5: <ResamplingCV>
## 6: <ResamplingCV>
## 7: <ResamplingCV>
## 8: <ResamplingCV>
```

2.6.2 Execution and Aggregation of Results

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

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

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

After the benchmark, one can calculate and aggregate the performance with `.$aggregate()`:

```
# measures: * area under the curve (auc) on training *
# area under the curve (auc) on test
measures = list(msr("classif.auc", id = "auc_train", predict_sets = "train"),
  msr("classif.auc", id = "auc_test"))
bmr$aggregate(measures)
##    nr  resample_result       task_id
## 1:  1 <ResampleResult> german_credit
## 2:  2 <ResampleResult> german_credit
## 3:  3 <ResampleResult> german_credit
## 4:  4 <ResampleResult> german_credit
## 5:  5 <ResampleResult>         sonar
## 6:  6 <ResampleResult>         sonar
## 7:  7 <ResampleResult>         sonar
## 8:  8 <ResampleResult>         sonar
##             learner_id resampling_id iters auc_train
## 1: classif.featureless            cv     3    0.5000
## 2:       classif.rpart            cv     3    0.8041
## 3:      classif.ranger            cv     3    0.9987
## 4:        classif.kknn            cv     3    0.9889
## 5: classif.featureless            cv     3    0.5000
## 6:       classif.rpart            cv     3    0.9369
## 7:      classif.ranger            cv     3    1.0000
## 8:        classif.kknn            cv     3    0.9982
##    auc_test
## 1:   0.5000
## 2:   0.7200
## 3:   0.7948
## 4:   0.7179
## 5:   0.5000
## 6:   0.8001
## 7:   0.9004
## 8:   0.9194
```

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

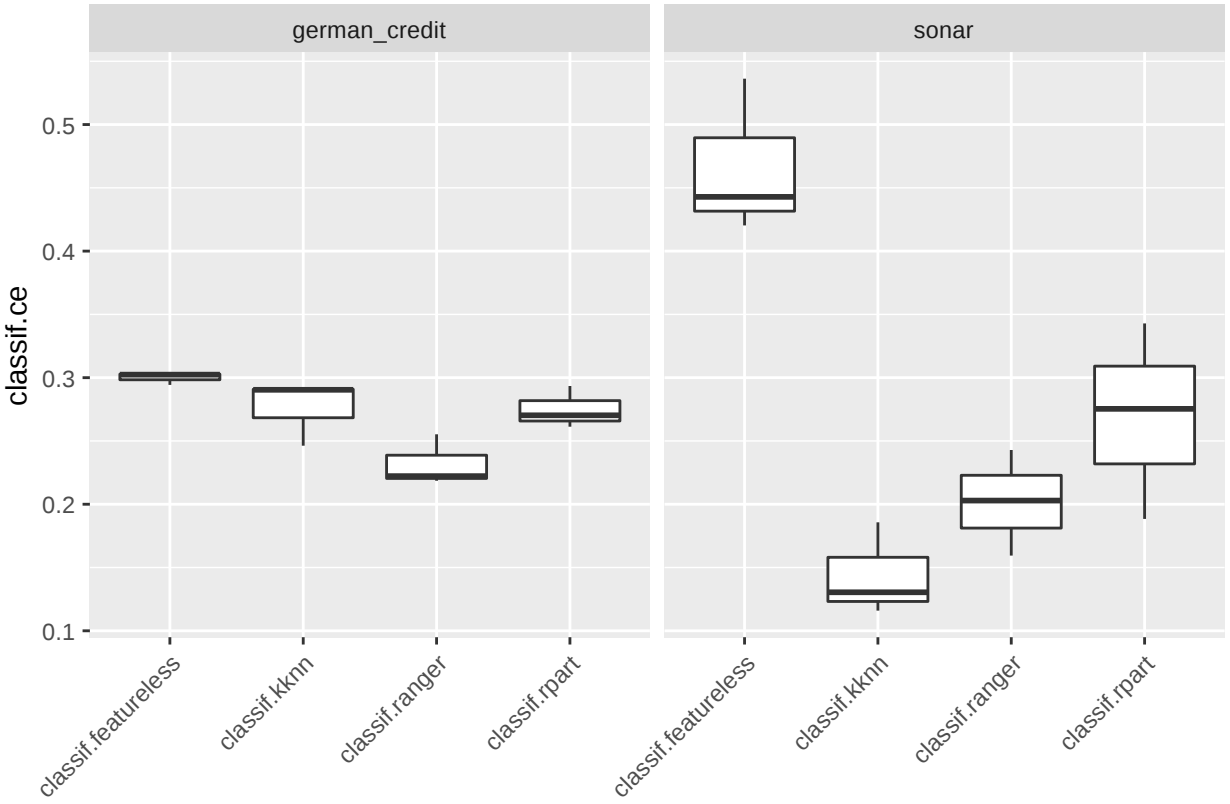
```
tab = bmr$aggregate(measures)
ranks = tab[, .(learner_id, rank_train = rank(-auc_train),
  rank_test = rank(-auc_test)), by = task_id]
print(ranks)
##          task_id          learner_id rank_train
## 1: german_credit classif.featureless          4
## 2: german_credit       classif.rpart          3
## 3: german_credit      classif.ranger          1
## 4: german_credit        classif.kknn          2
## 5:         sonar classif.featureless          4
## 6:         sonar       classif.rpart          3
## 7:         sonar      classif.ranger          1
## 8:         sonar        classif.kknn          2
##    rank_test
## 1:         4
## 2:         2
## 3:         1
## 4:         3
## 5:         4
## 6:         3
## 7:         2
## 8:         1

ranks[, .(mrank_train = mean(rank_train), mrank_test = mean(rank_test)),
  by = learner_id][order(mrank_test)]
##             learner_id mrank_train mrank_test
## 1:      classif.ranger           1        1.5
## 2:        classif.kknn           2        2.0
## 3:       classif.rpart           3        2.5
## 4: classif.featureless           4        4.0
```

Unsurprisingly, the featureless learner is outperformed.

2.6.3 Plotting Benchmark Results

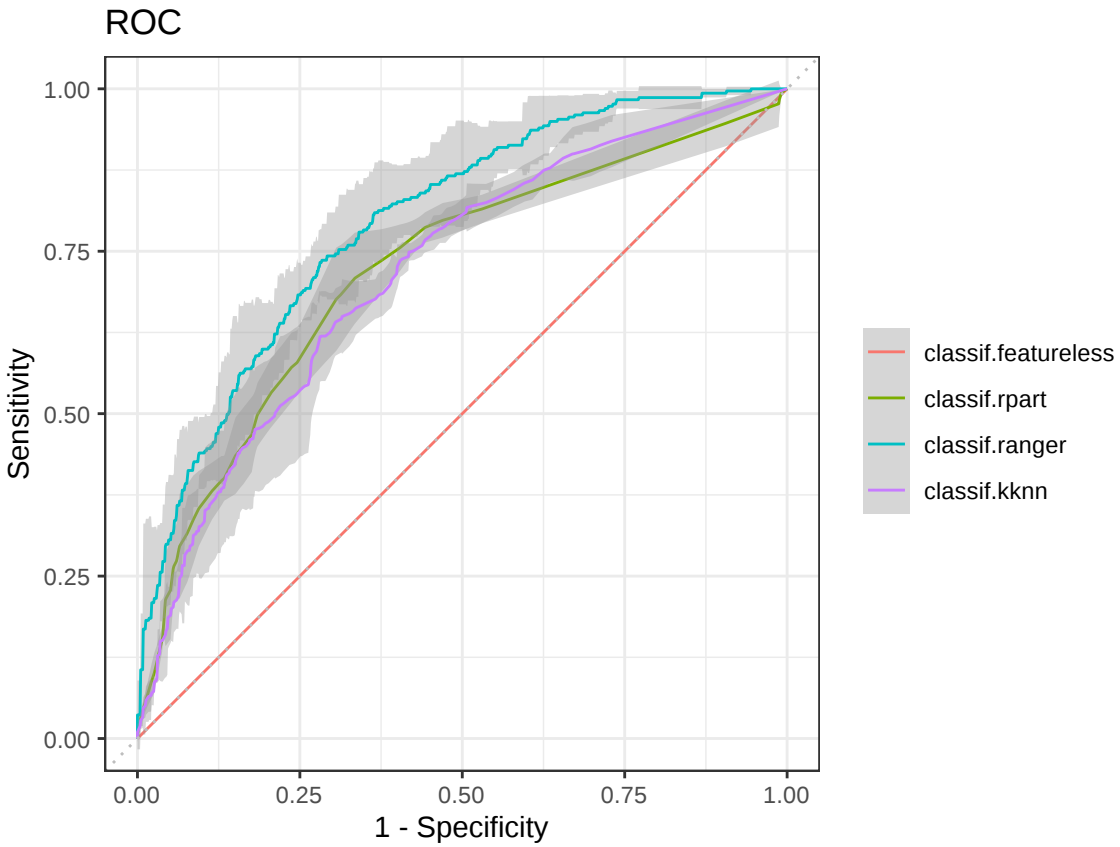
Analogously to plotting tasks, predictions or resample results, mlr3viz also provides a `ggplot2::autoplot()", text = "autoplot()` method for benchmark results.

```
library(mlr3viz)
library(ggplot2)

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



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

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



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

2.6.4 Extracting ResampleResults

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

```
tab = bmr$aggregate(measures)
rr = tab[task_id == "sonar" & learner_id == "classif.ranger"]$resample_result[[1]]
print(rr)
## <ResampleResult> of 3 iterations
## * Task: sonar
## * Learner: classif.ranger
## * Warnings: 0 in 0 iterations
## * Errors: 0 in 0 iterations
```

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

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

```
##      0.9004

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

# get the corresponding learner and train set
print(rr$learners[[i]])
## <LearnerClassifRanger:classif.ranger>
## * Model: -
## * Parameters: list()
## * Packages: ranger
## * Predict Type: prob
## * Feature types: logical, integer, numeric,
##   character, factor, ordered
## * Properties: importance, multiclass, oob_error,
##   twoclass, weights
head(rr$resampling$train_set(i))
## [1]  7 12 13 14 17 18
```

2.6.5 Converting and Merging ResampleResults

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

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

rr1 = resample(task, lrn("classif.rpart"), resampling)
rr2 = resample(task, lrn("classif.featureless"), resampling)

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

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

bmr1
## <BenchmarkResult> of 2 rows with 2 resampling runs
##  nr task_id          learner_id resampling_id iters
##   1    iris       classif.rpart       holdout     1
##   2    iris classif.featureless       holdout     1
##  warnings errors
##         0      0
##         0      0
```

2.7 Binary classification

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

```
# during construction
data("Sonar", package = "mlbench")
task = TaskClassif$new(id = "Sonar", Sonar, target = "Class",
  positive = "R")
## Warning: Using character row ids although the rownames
## of 'data' looks like integers

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

2.7.1 ROC Curve and Thresholds

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

```
learner = lrn("classif.rpart", predict_type = "prob")
pred = learner$train(task)$predict(task)
C = pred$confusion
print(C)
##          truth
## response  M  R
##        M 95 10
##        R 16 87
```

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

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

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

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

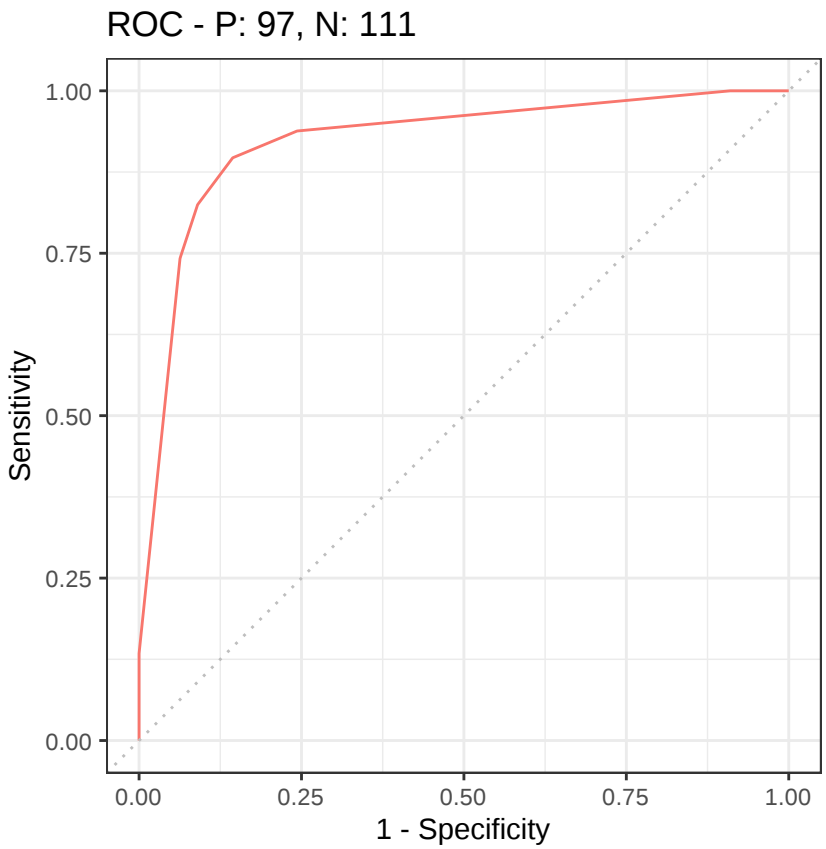
Source: Wikipedia

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

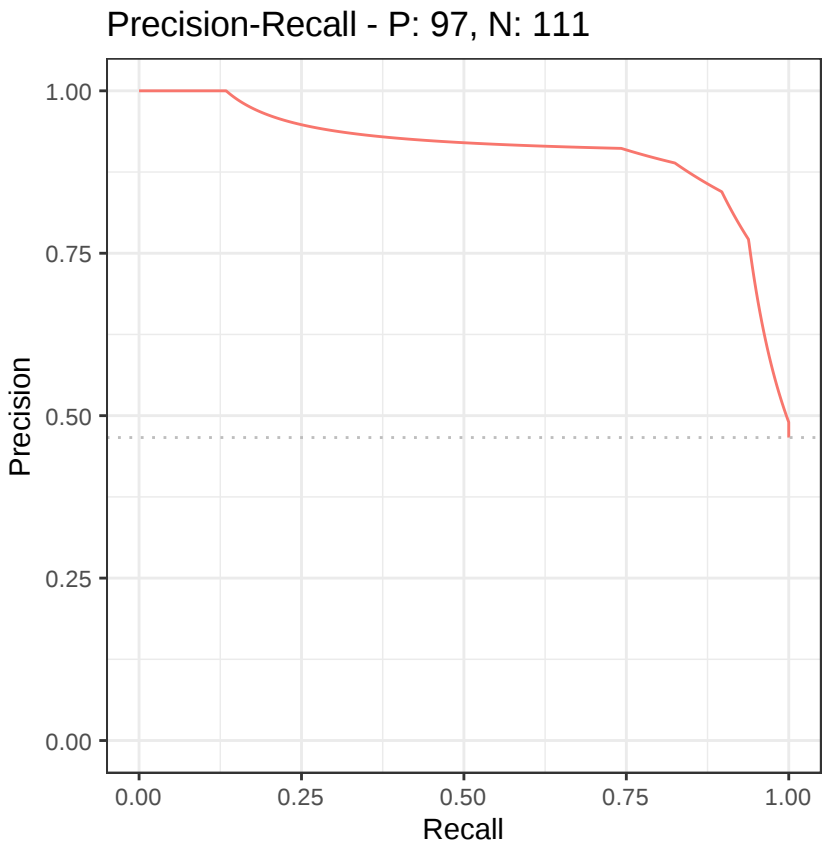
A scoring classifier is a model which produces scores or probabilities, instead of discrete labels. Nearly all modern classifiers can do that. Thresholding flexibly converts measured probabilities to labels. Predict 1 (positive class) if $\hat{f}(x) > \tau$ else predict 0. Normally, one could use $\tau = 0.5$ to convert probabilities to labels, but for imbalanced or cost-sensitive situations another threshold could be more suitable. After thresholding, any metric defined on labels can be used.

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

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




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



2.7.2 Threshold Tuning

3 Model Optimization

Model Tuning

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

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

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

This sub-chapter requires the package `mlr3-tuning`, an extension package which supports hyperparameter tuning.

Feature Selection

The second part of this chapter explains 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 model fitting and improve the learner performance by reducing noise in the data. Different approaches exist to identify the relevant features. In the sub-chapter on feature selection, three approaches are emphasized:

- Feature selection using filter algorithms
- Feature selection via variable importance filters
- Feature selection by employing the so called wrapper methods

A fourth approach, feature selection via ensemble filters, is introduced subsequently. The implementation of all four approaches in mlr3 is showcased using the extension-package mlr3filters.

Nested Resampling

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

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

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

3.1 Hyperparameter Tuning

Hyperparameter tuning is supported via the extension package mlr3tuning. The heart of mlr3tuning are the R6 classes:

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

3.1.1 The `TuningInstance` Class

The following sub-section examines the optimization of a simple classification tree on the `mlr_tasks_pima", text = "Pima Indian Diabetes` data set.

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

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

```
learner = lrn("classif.rpart")
learner$param_set
## ParamSet:
##              id    class lower upper levels default
## 1:     minsplit ParamInt     1   Inf             20
## 2:           cp ParamDbl     0     1           0.01
## 3:   maxcompete ParamInt     0   Inf              4
## 4: maxsurrogate ParamInt     0   Inf              5
## 5:     maxdepth ParamInt     1    30             30
## 6:         xval ParamInt     0   Inf             10
##    value
## 1:
## 2:
## 3:
## 4:
## 5:
## 6:     0
```

Here, we opt to tune two parameters namely on the one hand the complexity `cp` and second of all the termination criterion `minsplit`. As the tuning space has to be bound,one needs to set lower and upper bounds:

```
library(paradox)
tune_ps = ParamSet$new(list(ParamDbl$new("cp", lower = 0.001,
  upper = 0.1), ParamInt$new("minsplit", lower = 1, upper = 10)))
tune_ps
## ParamSet:
##          id    class lower upper levels     default
## 1:       cp ParamDbl 0.001   0.1        <NoDefault>
## 2: minsplit ParamInt 1.000  10.0        <NoDefault>
##    value
## 1:
## 2:
```

Next, we are able to define how to evaluate the performance. In order to determine the evaluation method, one has to choose a `Resampling", text = "resampling strategy` and a `Measure", text = "performance measure`.

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

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

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

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

```
library(mlr3tuning)

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

instance = TuningInstance$new(task = task, learner = learner,
  resampling = hout, measures = measure, param_set = tune_ps,
  terminator = evals20)
print(instance)
## <TuningInstance>
## * Task: <TaskClassif:pima>
## * Learner: <LearnerClassifRpart:classif.rpart>
## * Measures: classif.ce
## * Resampling: <ResamplingHoldout>
## * Terminator: <TerminatorEvals>
## * bm_args: list()
## ParamSet:
##          id    class lower upper levels     default
## 1:       cp ParamDbl 0.001   0.1        <NoDefault>
```

```
## 2: minsplit ParamInt 1.000  10.0        <NoDefault>
##    value
## 1:
## 2:
## Archive:
## Empty data.table (0 rows and 11 cols): nr,batch_nr,resample_result,task_id,learner_id,resampling_id...
```

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

3.1.2 The `Tuner` Class

The following algorithms are currently implemented in mlr3tuning:

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

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

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

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

3.1.3 Triggering the Tuning

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

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

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

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

```
instance$archive(unnest = "params")[, c("cp", "minsplit",
  "classif.ce")]
##          cp minsplit classif.ce
##  1: 0.00100        3     0.3008
##  2: 0.10000        8     0.3008
##  3: 0.10000       10     0.3008
##  4: 0.07525        1     0.3008
##  5: 0.00100        8     0.2773
##  6: 0.02575        1     0.2305
##  7: 0.10000        3     0.3008
##  8: 0.05050       10     0.3008
##  9: 0.10000        5     0.3008
## 10: 0.07525       10     0.3008
## 11: 0.07525        8     0.3008
## 12: 0.07525        3     0.3008
## 13: 0.02575        8     0.2305
## 14: 0.05050        5     0.3008
## 15: 0.02575        3     0.2305
## 16: 0.07525        5     0.3008
## 17: 0.00100        1     0.3164
## 18: 0.00100        5     0.2969
## 19: 0.05050        1     0.3008
## 20: 0.05050        3     0.3008
```

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

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

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

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

3.1.4 Automating the Tuning

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

```
library(paradox)
library(mlr3tuning)

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

at = AutoTuner$new(learner = learner, resampling = resampling,
  measures = measures, tune_ps = tune_ps, terminator = terminator,
  tuner = tuner)
at
## <AutoTuner:classif.rpart.tuned>
## * Model: -
## * Parameters: xval=0
## * Packages: rpart
## * Predict Type: response
## * Feature types: logical, integer, numeric,
##   factor, ordered
## * Properties: importance, missings, multiclass,
##   selected_features, twoclass, weights
```

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

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

```
grid = benchmark_grid(task = tsk("pima"), learner = list(at,
  lrn("classif.rpart")), resampling = rsmp("cv", folds = 3))
bmr = benchmark(grid)
bmr$aggregate(measures)
##    nr  resample_result task_id          learner_id
## 1:  1 <ResampleResult>    pima classif.rpart.tuned
## 2:  2 <ResampleResult>    pima       classif.rpart
##    resampling_id iters classif.ce
## 1:            cv     3     0.2643
## 2:            cv     3     0.2617
```

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

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

3.2 Feature Selection / Filtering

Often, data sets include a large number of features. The technique of extracting a subset of relevant features is called "feature selection". The objective of feature selection is to fit the sparse dependent of a model on a subset of available data features in the most suitable manner. Feature selection can enhance the interpretability of the model, speed up the learning process and improve the learner performance. Different approaches exist to identify the relevant features. In the literature two distinct approaches are emphasized: One is called Filtering and the other approach is often referred to as feature subset selection or wrapper methods.

What are the differences (Chandrashekar and Sahin 2014)?

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

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

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

All functionality that is related to feature selection is implemented via the extension package `mlr-org/mlr3filters`.

3.2.1 Filters

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

3.2.2 Calculating filter values

Currently, only classification and regression tasks are supported.

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

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

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

as.data.table(filter)
##         feature  score
## 1: Sepal.Length 1.0401
## 2:  Petal.Width 0.9894
```

```
## 3: Petal.Length 0.9881
## 4:  Sepal.Width 0.8314
```

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

```
filter_cor = FilterCorrelation$new()
filter_cor$param_set
## ParamSet:
##        id    class lower upper
## 1:    use ParamFct    NA    NA
## 2: method ParamFct    NA    NA
##                                                                    levels
## 1: everything,all.obs,complete.obs,na.or.complete,pairwise.complete.obs
## 2:                                             pearson,kendall,spearman
##       default value
## 1: everything
## 2:    pearson

# change parameter 'method'
filter_cor$param_set$values = list(method = "spearman")
filter_cor$param_set
## ParamSet:
##        id    class lower upper
## 1:    use ParamFct    NA    NA
## 2: method ParamFct    NA    NA
##                                                                    levels
## 1: everything,all.obs,complete.obs,na.or.complete,pairwise.complete.obs
## 2:                                             pearson,kendall,spearman
##       default    value
## 1: everything
## 2:    pearson spearman
```

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

```
filter = flt("cmim")
filter
## <FilterCMIM:cmim>
## Task Types: classif, regr
## Task Properties: -
## Packages: praznik
## Feature types: integer, numeric, factor, ordered
```

3.2.3 Variable Importance Filters

All `Learner` with the property "importance" come with integrated feature selection methods.

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

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

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

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

```
library(mlr3learners)

task = tsk("iris")
filter = flt("importance", learner = lrn)
filter$calculate(task)
head(as.data.table(filter), 3)
##         feature score
## 1:  Petal.Width 43.60
## 2: Petal.Length 43.51
## 3: Sepal.Length 10.01
```

3.2.4 Ensemble Methods

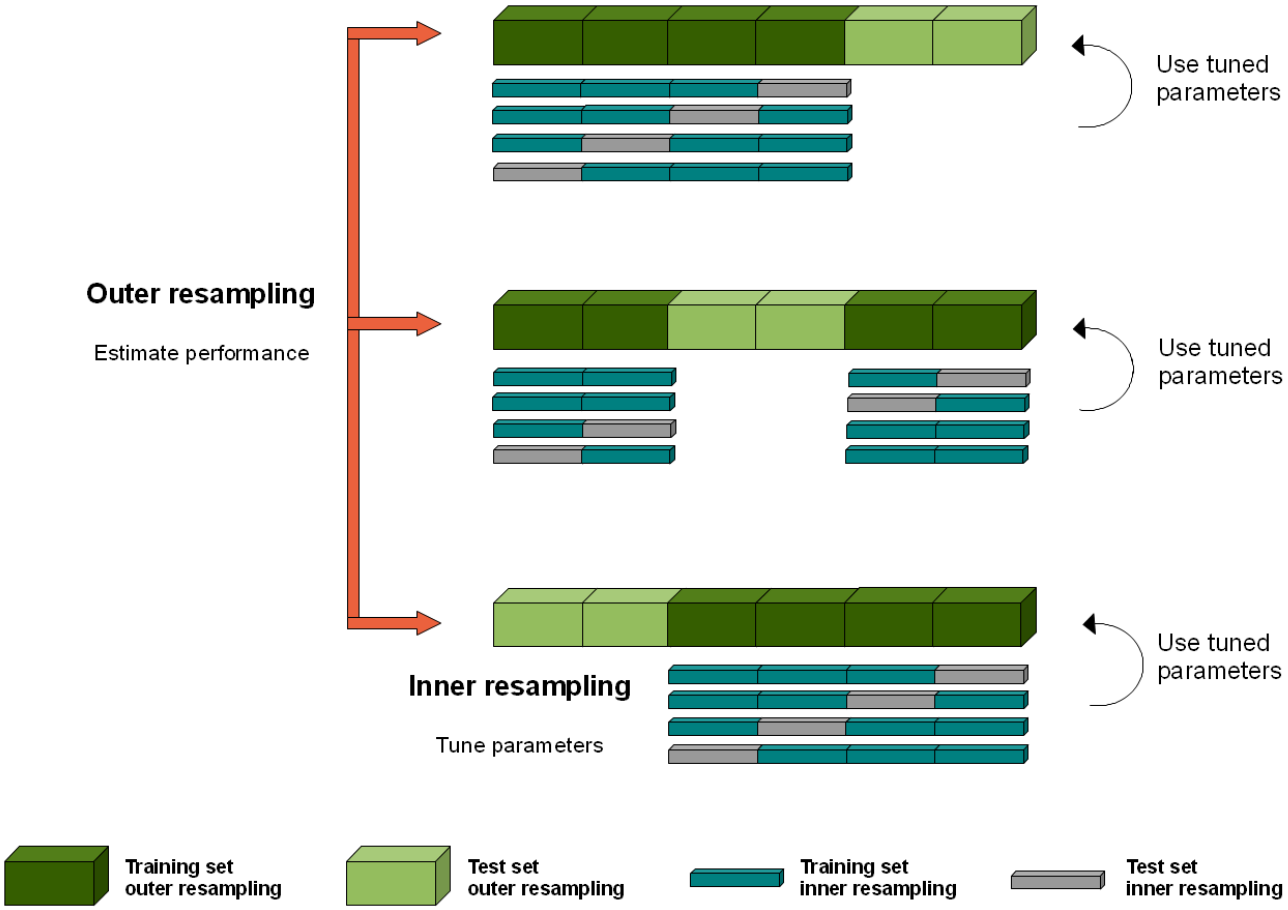
Work in progress :)

3.2.5 Wrapper Methods

Work in progress :) - via package *mlr3fswrap*

3.3 Nested Resampling

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



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

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

In `mlr-org/mlr3`, you can get nested resampling for free without programming any looping by using the `mlr3tuning::AutoTuner` class. This works as follows:

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

You can freely combine different inner and outer resampling strategies.

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

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

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

3.3.1 Execution

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

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

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

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

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

Now construct the `resample()` call:

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

3.3.2 Evaluation

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

For example, we can query the aggregated performance result:

```
rr$aggregate()
## classif.ce
##    0.07333
```

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

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

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

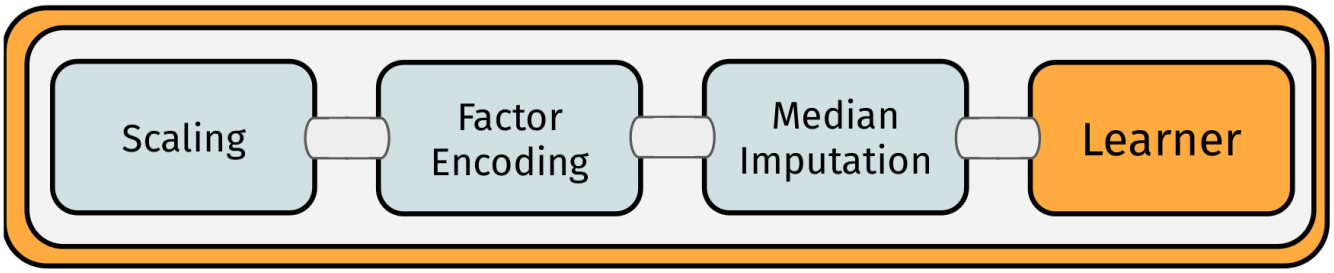
```
rr$prediction()$confusion
##             truth
## response     setosa versicolor virginica
##   setosa         50          0         0
##   versicolor      0         44         5
##   virginica       0          6        45
```

4 Pipelines

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

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

An example for such a graph can be found below:



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

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

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

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

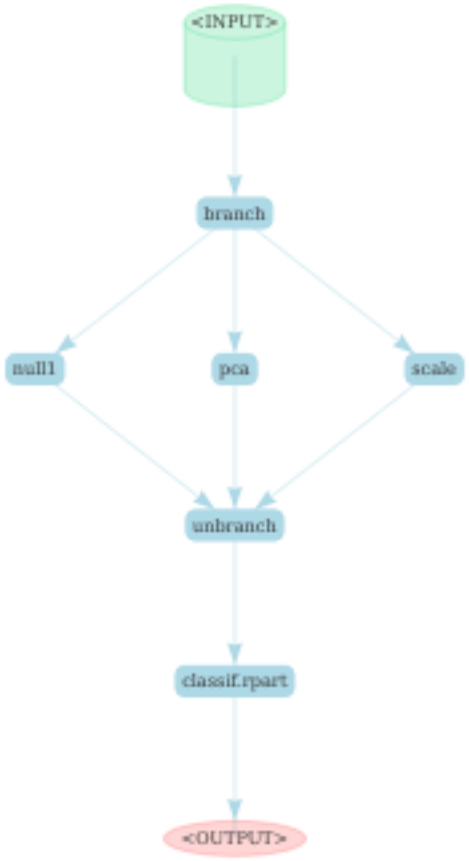
An extensive introduction to creating custom **PipeOps** (PO's) can be found in the technical introduction.

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

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

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

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



4.1 The Building Blocks: PipeOps

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

```
library("mlr3pipelines")
as.data.table(mlr_pipeops)
##                 key      packages input.num output.num
##  1:          boxcox bestNormalize         1          1
##  2:          branch                       1         NA
##  3:           chunk                       1         NA
##  4:  classbalancing                       1          1
```

```
##  5:      classifavg         stats        NA          1
##  6:    classweights                      1          1
##  7:        colapply                      1          1
##  8: collapsefactors                      1          1
##  9:            copy                      1         NA
## 10:    crankcompose        distr6        1          1
## 11:    distrcompose        distr6        2          1
## 12:          encode         stats        1          1
## 13:    encodeimpact                      1          1
## 14:      encodelmer   lme4,nloptr        1          1
## 15:    featureunion                     NA          1
## 16:          filter                      1          1
## 17:      fixfactors                      1          1
## 18:         histbin      graphics        1          1
## 19:             ica       fastICA        1          1
## 20:      imputehist      graphics        1          1
## 21:      imputemean                      1          1
## 22:    imputemedian         stats        1          1
## 23:    imputenewlvl                      1          1
## 24:    imputesample                      1          1
## 25:       kernelpca       kernlab        1          1
## 26:         learner                      1          1
## 27:      learner_cv                      1          1
## 28:         missind                      1          1
## 29:     modelmatrix         stats        1          1
## 30:          mutate                      1          1
## 31:             nop                      1          1
## 32:             pca                      1          1
## 33:     quantilebin         stats        1          1
## 34:         regravg                     NA          1
## 35: removeconstants                      1          1
## 36:           scale                      1          1
## 37:     scalemaxabs                      1          1
## 38:      scalerange                      1          1
## 39:          select                      1          1
## 40:           smote   smotefamily        1          1
## 41:     spatialsign                      1          1
## 42:       subsample                      1          1
## 43:        unbranch                     NA          1
## 44:      yeojohnson bestNormalize        1          1
##                key      packages input.num output.num
##     input.type.train            input.type.predict
##  1:             Task                          Task
##  2:                *                             *
##  3:             Task                          Task
##  4:      TaskClassif                   TaskClassif
##  5:             NULL             PredictionClassif
##  6:      TaskClassif                   TaskClassif
##  7:             Task                          Task
##  8:             Task                          Task
```

```
##  9:               *                           *
## 10:            NULL               PredictionSurv
## 11:       NULL,NULL PredictionSurv,PredictionSurv
## 12:            Task                        Task
## 13:            Task                        Task
## 14:            Task                        Task
## 15:            Task                        Task
## 16:            Task                        Task
## 17:            Task                        Task
## 18:            Task                        Task
## 19:            Task                        Task
## 20:            Task                        Task
## 21:            Task                        Task
## 22:            Task                        Task
## 23:            Task                        Task
## 24:            Task                        Task
## 25:            Task                        Task
## 26:     TaskClassif                 TaskClassif
## 27:     TaskClassif                 TaskClassif
## 28:            Task                        Task
## 29:            Task                        Task
## 30:            Task                        Task
## 31:               *                           *
## 32:            Task                        Task
## 33:            Task                        Task
## 34:            NULL               PredictionRegr
## 35:            Task                        Task
## 36:            Task                        Task
## 37:            Task                        Task
## 38:            Task                        Task
## 39:            Task                        Task
## 40:            Task                        Task
## 41:            Task                        Task
## 42:            Task                        Task
## 43:               *                           *
## 44:            Task                        Task
##     input.type.train           input.type.predict
##     output.type.train output.type.predict
##  1:             Task                Task
##  2:                *                   *
##  3:             Task                Task
##  4:      TaskClassif         TaskClassif
##  5:             NULL   PredictionClassif
##  6:      TaskClassif         TaskClassif
##  7:             Task                Task
##  8:             Task                Task
##  9:                *                   *
## 10:             NULL      PredictionSurv
## 11:             NULL      PredictionSurv
## 12:             Task                Task
```

```
## 13:              Task                Task
## 14:              Task                Task
## 15:              Task                Task
## 16:              Task                Task
## 17:              Task                Task
## 18:              Task                Task
## 19:              Task                Task
## 20:              Task                Task
## 21:              Task                Task
## 22:              Task                Task
## 23:              Task                Task
## 24:              Task                Task
## 25:              Task                Task
## 26:              NULL   PredictionClassif
## 27:       TaskClassif         TaskClassif
## 28:              Task                Task
## 29:              Task                Task
## 30:              Task                Task
## 31:                 *                   *
## 32:              Task                Task
## 33:              Task                Task
## 34:              NULL      PredictionRegr
## 35:              Task                Task
## 36:              Task                Task
## 37:              Task                Task
## 38:              Task                Task
## 39:              Task                Task
## 40:              Task                Task
## 41:              Task                Task
## 42:              Task                Task
## 43:                 *                   *
## 44:              Task                Task
##     output.type.train output.type.predict
```

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

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

Some POs require additional arguments for construction:

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

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

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

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

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

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

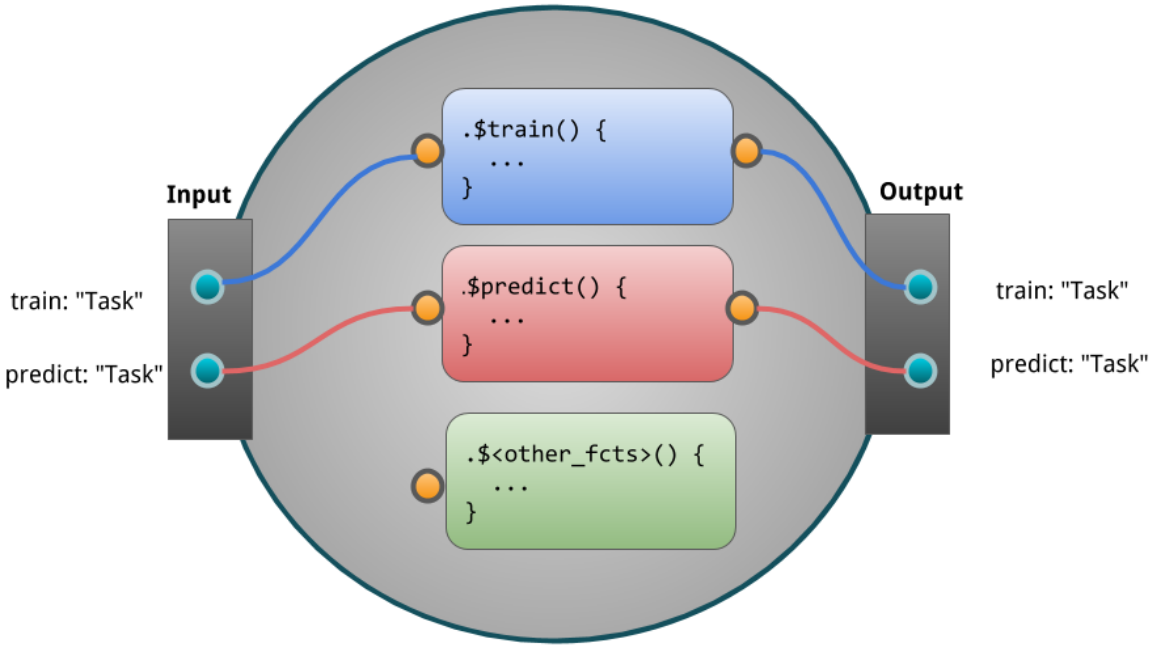
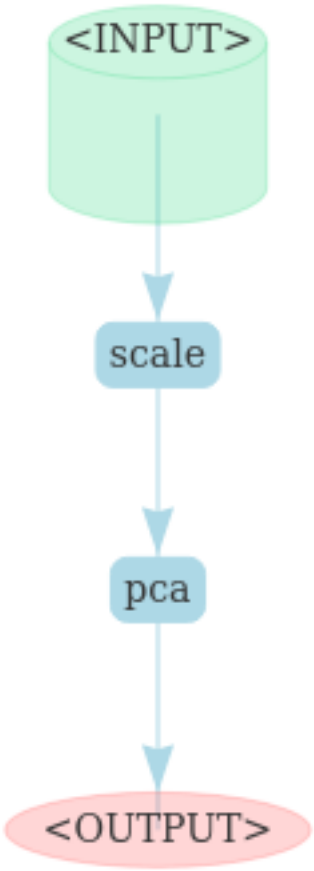


Figure 4.1: A PipeOperator

4.2 The Pipeline Operator: %>>%

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. 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.

```
gr = mlr_pipeops$get("scale") %>>% mlr_pipeops$get("pca")
gr$plot(html = TRUE) %>% visInteraction(zoomView = FALSE)  # disable zoom
```



4.3 Nodes, Edges and Graphs

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

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

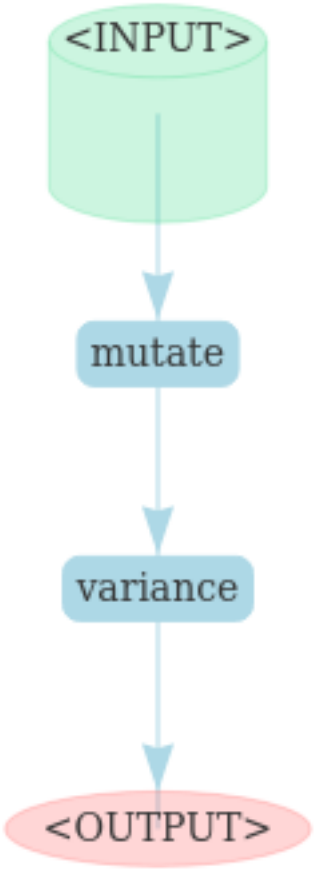
```
graph = Graph$new()$add_pipeop(mutate)$add_pipeop(filter)$add_edge("mutate",
  "variance")  # add connection mutate -> filter
```

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(html = TRUE) %>% visInteraction(zoomView = FALSE)  # disable zoom
```



Chaining multiple POs of the same kind

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

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

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

4.4 Modeling

The main purpose of a `Graph` is to build combined preprocessing and model fitting pipelines that can be used as mlr3 `Learner`. In the following we chain two preprocessing tasks:

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

and then chain a PO learner to train and predict on the modified dataset.

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

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

```
task = mlr_tasks$get("iris")
graph$train(task)
## $classif.rpart.output
## NULL
graph$predict(task)
## $classif.rpart.output
## <PredictionClassif> for 150 observations:
##     row_id     truth  response
##          1    setosa    setosa
##          2    setosa    setosa
##          3    setosa    setosa
## ---
##        148 virginica virginica
##        149 virginica virginica
##        150 virginica virginica
```

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

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

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

```
cv3 = rsmp("cv", folds = 3)
resample(task, glrn, cv3)
## <ResampleResult> of 3 iterations
## * Task: iris
## * Learner: mutate.variance.classif.rpart
## * Warnings: 0 in 0 iterations
## * Errors: 0 in 0 iterations
```

4.4.1 Setting Hyperparameters

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

```
glrn$param_set$values$variance.filter.frac = 0.25
cv3 = rsmp("cv", folds = 3)
resample(task, glrn, cv3)
## <ResampleResult> of 3 iterations
## * Task: iris
## * Learner: mutate.variance.classif.rpart
## * Warnings: 0 in 0 iterations
## * Errors: 0 in 0 iterations
```

4.4.2 Tuning

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

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

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

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

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

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

```
instance$result
```

4.5 Non-Linear Graphs

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

Possible types are:

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

4.5.1 Branching & Copying

The `PipeOpBranch` and `PipeOpUnbranch` POs make it possible to specify multiple alternative paths. Only one 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).

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

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

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

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

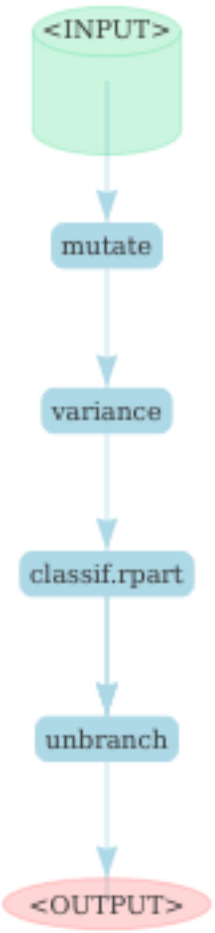
```
graph = mlr_pipeops$get("branch", c("nop", "pca", "scale")) %>>%
  gunion(list(mlr_pipeops$get("nop", id = "null1"), mlr_pipeops$get("pca"),
    mlr_pipeops$get("scale")))
```

Without "unbranching" one creates the following graph:

```
graph$plot(html = TRUE) %>% visInteraction(zoomView = FALSE)  # disable zoom
```

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

```
(graph %>>% mlr_pipeops$get("unbranch", c("nop", "pca", "scale")))$plot(html = TRUE) %>%
  visInteraction(zoomView = FALSE)  # disable zoom
```



4.5.2 Model Ensembles

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

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

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

4.5.2.1 Bagging

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

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

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

We can achieve this by downsampling our data before training a learner, repeating this e.g. 10 times and then performing a majority vote on the predictions.

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

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

We can now copy this operation 10 times using `greplicate` .

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

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

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

Now we can plot again to see what happens:

```
vn = bagging$plot(html = TRUE)
visNetwork::visInteraction(vn, zoomView = FALSE)  # disable zoom
```

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

```
baglrn = GraphLearner$new(bagging)
baglrn$train(task, train.idx)
baglrn$predict(task, test.idx)
## <PredictionClassif> for 30 observations:
##     row_id     truth  response
##          2    setosa    setosa
##          5    setosa    setosa
##         12    setosa    setosa
## ---
##        144 virginica virginica
##        145 virginica virginica
##        147 virginica virginica
```

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

4.5.2.2 Stacking

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

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

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

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

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

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

We use a 'FIXME' in parallel to the "level 0" learner, in order to send the unchanged Task to the next level, where it is then combined with the predictions from our decision tree learner.

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

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

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

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

```
stack = combined %>>% PipeOpLearner$new(lrn$clone())
vn = stack$plot(html = TRUE)
visNetwork::visInteraction(vn, zoomView = FALSE)  # disable zoom
```

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

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

4.5.2.3 Multilevel Stacking

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

In this case, we train a `glmnet` and 2 different `rpart` models (some transform its inputs using `PipeOpPCA`) on our task in the "level 0" and concatenate them with the original features (via 'FIXME'). 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 'FIXME') and pass this on to "level 2". In "level 2" we simply concatenate all "level 1" outputs and train a final decision tree.

```
library(mlr3learners)  # for classif.glmnet
rprt = lrn("classif.rpart", predict_type = "prob")
glmn = lrn("classif.glmnet", predict_type = "prob")

# Create Learner CV Operators
lrn_0 = PipeOpLearnerCV$new(rprt, id = "rpart_cv_1")
lrn_0$values$maxdepth = 5L
lrn_1 = PipeOpPCA$new(id = "pca1") %>>% PipeOpLearnerCV$new(rprt,
  id = "rpart_cv_2")
lrn_1$values$rpart_cv_2.maxdepth = 1L
lrn_2 = PipeOpPCA$new(id = "pca2") %>>% PipeOpLearnerCV$new(glmn)

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

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

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

# Plot the resulting graph
vn = level_2$plot(html = TRUE)
visNetwork::visInteraction(vn, zoomView = FALSE)  # disable zoom

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

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

4.6 Special Operators

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

4.6.1 Imputation: `PipeOpImpute`

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

The following `PipeOp`, `PipeOpImpute`, imputes numeric values from a histogram, adds a new level for factors and additionally adds a column marking whether a value for a given feature was missing or not.

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

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

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

4.6.2 Feature Engineering: `PipeOpMutate`

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

```
pom = PipeOpMutate$new()

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

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

4.6.3 Training on data subsets: `PipeOpChunk`

In cases, where data is too big to fit into the machine's memory, an often-used technique is to split the data into several parts. Subsequently, the parts are trained on each part of the data. After undertaking these steps, we aggregate the models. In this example, we split our data into 4 parts using `PipeOpChunk` . Additionally, we create 4 `PipeOpLearner` POS, which are then trained on each split of the data.

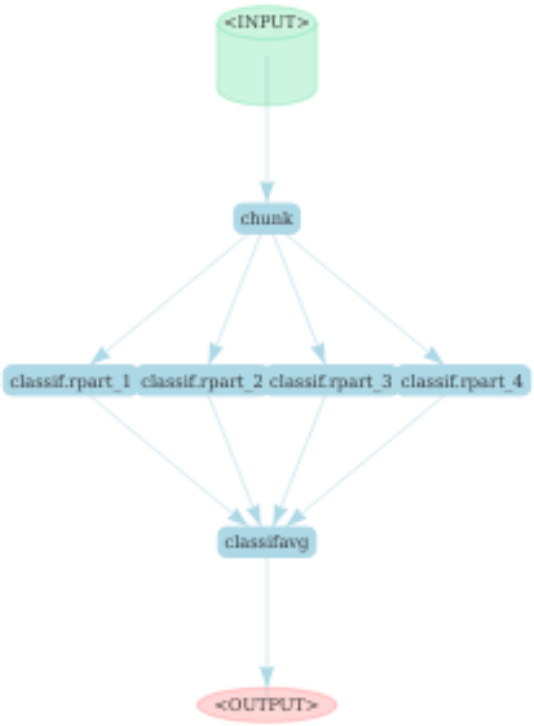
```
chks = PipeOpChunk$new(4)
lrns = greplicate(PipeOpLearner$new(mlr_learners$get("classif.rpart")),
  4)
```

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

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

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

```
pipeline = chks %>>% lrns %>>% mjv
pipeline$plot(html = TRUE) %>% visInteraction(zoomView = FALSE)  # disable zoom
```




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

```
## classif.ce
##      0.025
```

4.6.4 Feature Selection: `PipeOpFilter` and `PipeOpSelect`

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

A `PipeOp` for filters is `PipeOpFilter`:

```
PipeOpFilter$new(mlr3filters::FilterInformationGain$new())
## PipeOp: <information_gain> (not trained)
## values: <list()>
## Input channels <name [train type, predict type]>:
##   input [Task,Task]
## Output channels <name [train type, predict type]>:
##   output [Task,Task]
```

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

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

5 Technical

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

Parallelization

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

Database Backends

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

Parameters

In the section Parameters instructions are given on how to:

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

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

Logging and Verbosity

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

Transition Guide

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

5.1 Parallelization

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

`mlr-org/mlr3` uses the `future` backends for parallelization. Make sure you have installed the required packages `future` and `future.apply`:

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

In the following section, we will use the *spam* task and a simple classification tree (`"classif.rpart"`) to showcase parallelization. We use the `future` package to parallelize the resampling by selecting a backend via the function `future::plan()`. We use the `"multiprocess"` backend here which uses threads on UNIX based systems and a "Socket" cluster on Windows.

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

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

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

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

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

Choosing the parallelization level

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

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

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

5.2 Error Handling

To demonstrate how to properly deal with misbehaving learners, `mlr-org/mlr3` ships with the learner `mlr_learners_classif.debug`
`"classif.debug`:

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

This learner comes with special hyperparameters that let us control

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

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

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

```
task = tsk("iris")
learner$train(task)$predict(task)$confusion
##             truth
## response     setosa versicolor virginica
##   setosa         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,`mlr-org/mlr3` does not catch conditions such as warnings or errors raised by third-party code like learners:

```
learner$param_set$values = list(error_train = 1)
learner$train(tsk("iris"))
## Error in learner$train_internal(task = task): Error from classif.debug->train()
```

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

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

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

These two mechanisms are explained in the following subsections.

5.2.1 Encapsulation

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

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

learner$train(task)
```

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

```
learner$log
##    stage   class                                msg
## 1: train warning Warning from classif.debug->train()
## 2: train   error   Error from classif.debug->train()
learner$warnings
## [1] "Warning from classif.debug->train()"
learner$errors
## [1] "Error from classif.debug->train()"
```

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

```
learner$encapsulate = c(train = "callr", predict = "callr")
learner$param_set$values = list(segfault_train = 1)
learner$train(task = task)
learner$errors
## [1] "callr process exited with status -11"
```

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

```
learner$predict(task)
## Error: Cannot predict, Learner 'classif.debug' has not been trained yet
```

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

5.2.2 Fallback learners

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

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

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

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

```
task = tsk("iris")
learner = lrn("classif.debug")
learner$param_set$values = list(error_train = 1)
learner$encapsulate = c(train = "evaluate")
learner$fallback = lrn("classif.featureless")
learner$train(task)
learner
## <LearnerClassifDebug:classif.debug>
## * Model: -
## * Parameters: error_train=1
## * Packages: -
## * Predict Type: response
## * Feature types: logical, integer, numeric,
##   character, factor, ordered
## * Properties: missings, multiclass, twoclass
## * Errors: Error from classif.debug->train()
```

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

```
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
## 1:  1 <ResampleResult>    iris classif.debug
## 2:  2 <ResampleResult>    iris classif.rpart
##    resampling_id iters warnings errors classif.ce
## 1:            cv    10        0      1    0.67333
## 2:            cv    10        0      0    0.07333
```

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

```
rr = aggr[learner_id == "classif.debug"]$resample_result[[1L]]
rr$errors
##    iteration                                msg
## 1:         3 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>
##          0          0         75         75

# with fallback
learner$fallback = lrn("classif.featureless")
p = learner$train(task)$predict(task)
table(p$response, useNA = "always")
##
##     setosa versicolor  virginica       <NA>
##          0         75         75          0
```

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

5.3 Database Backends

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

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

5.3.1 Use Case: NYC Flights

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

```
# load data
requireNamespace("DBI")
requireNamespace("RSQLite")
requireNamespace("nycflights13")
data("flights", package = "nycflights13")
str(flights)
## Classes 'tbl_df', 'tbl' and 'data.frame':    336776 obs. of  19 variables:
##  $ year          : int  2013 2013 2013 2013 2013 2013 2013 2013 2013 2013 ...
##  $ month         : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ day           : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ dep_time      : int  517 533 542 544 554 554 555 557 557 558 ...
##  $ sched_dep_time: int  515 529 540 545 600 558 600 600 600 600 ...
##  $ dep_delay     : num  2 4 2 -1 -6 -4 -5 -3 -3 -2 ...
##  $ arr_time      : int  830 850 923 1004 812 740 913 709 838 753 ...
##  $ sched_arr_time: int  819 830 850 1022 837 728 854 723 846 745 ...
##  $ arr_delay     : num  11 20 33 -18 -25 12 19 -14 -8 8 ...
##  $ carrier       : chr  "UA" "UA" "AA" "B6" ...
##  $ flight        : int  1545 1714 1141 725 461 1696 507 5708 79 301 ...
##  $ tailnum       : chr  "N14228" "N24211" "N619AA" "N804JB" ...
##  $ origin        : chr  "EWR" "LGA" "JFK" "JFK" ...
##  $ dest          : chr  "IAH" "IAH" "MIA" "BQN" ...
##  $ air_time      : num  227 227 160 183 116 150 158 53 140 138 ...
##  $ distance      : num  1400 1416 1089 1576 762 ...
##  $ hour          : num  5 5 5 5 6 5 6 6 6 6 ...
##  $ minute        : num  15 29 40 45 0 58 0 0 0 0 ...
##  $ time_hour     : POSIXct, format: "2013-01-01 05:00:00" ...

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

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

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

5.3.2 Preprocessing with `dplyr`

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

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

# select the 'flights' table, enter dplyr
```

```
library(dplyr)
library(dbplyr)
tbl = tbl(con, "flights")
```

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

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

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

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

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

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

The factor levels of the feature `carrier` are merged so that infrequent carriers are replaced by level "other":

```
tbl = mutate(tbl, carrier = case_when(carrier %in% c("OO",
  "HA", "YV", "F9", "AS", "FL", "VX", "WN") ~ "other",
  TRUE ~ carrier))
```

5.3.3 DataBackendDplyr

The processed table is now used to create a `mlr3db::DataBackendDplyr` from `mlr3db`:

```
library("mlr3db")
b = as_data_backend(tbl, primary_key = "row_id")
```

We can now use the interface of `DataBackend` to query some basic information of the data:

```
b$nrow
## [1] 163707
b$ncol
## [1] 13
b$head()
##    row_id year month day hour minute dep_time arr_time
```

```
## 1:      2 2013     1   1    5     29      533      850
## 2:      4 2013     1   1    5     45      544     1004
## 3:      6 2013     1   1    5     58      554      740
## 4:      8 2013     1   1    6      0      557      709
## 5:     10 2013     1   1    6      0      558      753
## 6:     12 2013     1   1    6      0      558      853
##    carrier flight air_time distance arr_delay
## 1:      UA   1714      227     1416        20
## 2:      B6    725      183     1576       -18
## 3:      UA   1696      150      719        12
## 4:      EV   5708       53      229       -14
## 5:      AA    301      138      733         8
## 6:      B6     71      158     1005        -3
```

Note that the `DataBackendDplyr` does not know about any rows or columns we have filtered out with `dplyr` before, it just operates on the view we provided.

5.3.4 Model fitting

We create the following mlr3 objects:

- A `TaskRegr", text = "regression task`, based on the previously created `mlr3db::DataBackendDplyr`.
- A regression learner (`mlr_learners_regr.rpart", text = "regr.rpart`).
- A resampling strategy: 3 times repeated subsampling using 2% of the observations for training (“`mlr_resamplings_subsampli` `text = "subsampling`”)
- Measures “`mlr_measures_regr.mse", text = "mse`”, “`mlr_measures_time_train", text = "time_predict`” and “`mlr_measures_time_predict", text = "time_predict`”

```
task = TaskRegr$new("flights_sqlite", b, target = "arr_delay")
learner = lrn("regr.rpart")
measures = mlr_measures$mget(c("regr.mse", "time_train",
  "time_predict"))
resampling = rsmp("subsampling")
resampling$param_set$values = list(repeats = 3, ratio = 0.02)
```

We pass all these objects to `resample()` to perform a simple resampling with three iterations. In each iteration, only the required subset of the data is queried from the SQLite data base and passed to `rpart::rpart()`:

```
rr = resample(task, learner, resampling)
print(rr)
## <ResampleResult> of 3 iterations
## * Task: flights_sqlite
## * Learner: regr.rpart
## * Warnings: 0 in 0 iterations
## * Errors: 0 in 0 iterations
rr$aggregate(measures)
##     regr.mse   time_train time_predict
##    1167.4481       0.1483       1.6047
```

5.3.5 Cleanup

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

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

5.4 Parameters (using paradox)

The paradox package offers a language for the description of *parameter spaces*, as well as tools for useful operations on these parameter spaces. A parameter space is often useful when describing:

- A set of sensible input values for an R function
- The set of possible values that slots of a configuration object can take
- The search space of an optimization process

The tools provided by paradox therefore relate to:

- **Parameter checking**: Verifying that a set of parameters satisfies the conditions of a parameter space
- **Parameter sampling**: Generating parameter values that lie in the parameter space for systematic exploration of program behavior depending on these parameters

paradox is, by nature, an auxiliary package that derives its usefulness from other packages that make use of it. It is heavily utilized in other mlr-org packages such as mlr3, mlr3pipelines, and mlr3tuning.

5.4.1 Reference Based Objects

paradox is the spiritual successor to the `ParamHelpers` package and was written from scratch using the `R6` class system. The most important consequence of this is that all objects created in paradox are "reference-based", unlike most other objects in R. When a change is made to a `ParamSet` object, for example by adding a parameter using the `$add()` function, all variables that point to this `ParamSet` will contain the changed object. To create an independent copy of a `ParamSet`, the `$clone()` method needs to be used:

```
library("paradox")

ps = ParamSet$new()
ps2 = ps
ps3 = ps$clone(deep = TRUE)
print(ps)  # the same for ps2 and ps3
## ParamSet:
## Empty.
```

```
ps$add(ParamLgl$new("a"))
```

```
print(ps)  # ps was changed
## ParamSet:
##    id    class lower upper      levels     default
## 1:  a ParamLgl    NA    NA  TRUE,FALSE <NoDefault>
##    value
## 1:
print(ps2)  # contains the same reference as ps
## ParamSet:
##    id    class lower upper      levels     default
## 1:  a ParamLgl    NA    NA  TRUE,FALSE <NoDefault>
##    value
## 1:
print(ps3)  # is a 'clone' of the old (empty) ps
## ParamSet:
## Empty.
```

5.4.2 Defining a Parameter Space

5.4.2.1 Single Parameters

The basic building block for describing parameter spaces is the **`Param`** class. It represents a single parameter, which usually can take a single atomic value. Consider, for example, trying to configure the `rpart` package's `rpart.control` object. It has various components (`minsplit`, `cp`, ...) that all take a single value, and that would all be represented by a different instance of a `Param` object.

The `Param` class has various sub-classes that represent different value types:

- `ParamInt`: Integer numbers
- `ParamDbl`: Real numbers
- `ParamFct`: String values from a set of possible values, similar to R `factors`
- `ParamLgl`: Truth values (`TRUE` / `FALSE`), as `logicals` in R
- `ParamUty`: Parameter that can take any value

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

```
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", "?Param`.

```
parB$lower
## [1] 0
parA$levels
## [1]  TRUE FALSE
parE$class
## [1] "ParamUty"
```

It is also possible to get all information of a `Param` as `data.table` by calling `as.data.table`.

```
as.data.table(parA)
##    id    class lower upper      levels nlevels is_bounded special_vals     default storage_type tags
## 1:  A ParamLgl    NA    NA  TRUE,FALSE       2       TRUE       <list> <NoDefault>      logical
```

Type / Range Checking A `Param` object offers the possibility to check whether a value satisfies its condition, i.e. is of the right type, and also falls within the range of allowed values, using the `$test()`, `$check()`, and `$assert()` functions. `test()` should be used within conditional checks and returns `TRUE` or `FALSE`, while `check()` returns an error description when a value does not conform to the parameter (and thus plays well with the `checkmate::assert()` function). `assert()` will throw an error whenever a value does not fit.

```
parA$test(FALSE)
## [1] TRUE
parA$test("FALSE")
## [1] FALSE
parA$check("FALSE")
## [1] "Must be of type 'logical flag', not 'character'"
```

Instead of testing single parameters, it is often more convenient to check a whole set of parameters using a `ParamSet`.

5.4.2.2 Parameter Sets

The ordered collection of parameters is handled in a `ParamSet`[1]. It is initialized using the `$new()` function and optionally takes a list of `Params` as argument. Parameters can also be added to the constructed `ParamSet` using the `$add()` function. It is even possible to add whole `ParamSets` to other `ParamSets`.

```
ps = ParamSet$new(list(parA, parB))
ps$add(parC)
ps$add(ParamSet$new(list(parD, parE)))
print(ps)
## ParamSet:
##    id    class lower upper      levels     default
## 1:  A ParamLgl    NA    NA  TRUE,FALSE <NoDefault>
## 2:  B ParamInt     0    10             <NoDefault>
## 3:  C ParamDbl     0     4             <NoDefault>
## 4:  D ParamFct    NA    NA       x,y,z           y
## 5:  E ParamUty    NA    NA             <NoDefault>
##    value
## 1:
## 2:
## 3:
## 4:
## 5:
```

The individual parameters can be accessed through the `$params` slot. It is also possible to get information about all parameters in a vectorized fashion using mostly the same slots as for individual `Params` (i.e. `$class`, `$levels` etc.), see `?ParamSet` for details.

It is possible to reduce `ParamSets` using the **`$subset`** method. Be aware that it modifies a ParamSet in-place, so a "clone" must be created first if the original `ParamSet` should not be modified.

```
psSmall = ps$clone()
psSmall$subset(c("A", "B", "C"))
print(psSmall)
## ParamSet:
##    id    class lower upper      levels     default
## 1:  A ParamLgl    NA    NA  TRUE,FALSE <NoDefault>
## 2:  B ParamInt     0    10             <NoDefault>
## 3:  C ParamDbl     0     4             <NoDefault>
##    value
## 1:
## 2:
## 3:
```

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`.

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

```
as.data.table(ps)
##    id    class lower upper      levels nlevels is_bounded special_vals     default storage_type      tags
## 1:  A ParamLgl    NA    NA TRUE,FALSE       2       TRUE       <list> <NoDefault>      logical
## 2:  B ParamInt     0    10                 11       TRUE       <list> <NoDefault>      integer tag1,tag2
## 3:  C ParamDbl     0     4                Inf       TRUE       <list> <NoDefault>      numeric
## 4:  D ParamFct    NA    NA       x,y,z       3       TRUE       <list>           y    character
## 5:  E ParamUty    NA    NA                Inf      FALSE       <list> <NoDefault>         list
```

Type / Range Checking Similar to individual `Params`, the `ParamSet` provides `$test()`, `$check()` and `$assert()` functions that allow for type and range checking of parameters. Their argument must be a named list with values that are checked against the respective parameters. It is possible to check only a subset of parameters.

```
ps$check(list(A = TRUE, B = 0, E = identity))
## [1] TRUE
ps$check(list(A = 1))
## [1] "A: Must be of type 'logical flag', not 'double'"
ps$check(list(Z = 1))
## [1] "Parameter 'Z' not available. Did you mean 'A' / 'B' / 'C'?"
```

Values in a `ParamSet` Although a `ParamSet` fundamentally represents a value space, it also has a slot `$values` that can contain a point within that space. This is useful because many things that define a parameter space need similar operations (like parameter checking) that can be simplified. The `$values` slot contains a named list that is always checked against parameter constraints. When trying to set parameter values, e.g. for mlr3 `Learners`, it is the `$values` slot of its `$param_set` that needs to be used.

```
ps$values = list(A = TRUE, B = 0)
ps$values$B = 1
print(ps$values)
## $A
## [1] TRUE
##
## $B
## [1] 1
```

The parameter constraints are automatically checked:

```
ps$values$B = 100
## Error in (function (xs) : Assertion on 'xs' failed: B: Element 1 is not <= 10.
```

Dependencies It is often the case that certain parameters are irrelevant or should not be given depending on values of other parameters. An example would be a parameter that switches a certain algorithm feature (for example regularization) on or off, combined with another parameter that controls the behavior of that feature (e.g. a regularization parameter). The second parameter would be said to *depend* on the first parameter having the value `TRUE`.

A dependency can be added using the `$add_dep` method, which takes both the ids of the "depender" and "dependee" parameters as well as a `Condition` object. The `Condition` object represents the check to be performed on the "dependee". Currently it can be created using `CondEqual$new()` and `CondAnyOf$new()`. Multiple dependencies can be added, and parameters that depend on others can again be depended on, as long as no cyclic dependencies are introduced.

The consequence of dependencies are twofold: For one, the `$check()`, `$test()` and `$assert()` tests will not accept the presence of a parameter if its dependency is not met. Furthermore, when sampling or creating grid designs from a `ParamSet`, the dependencies will be respected (see Parameter Sampling, in particular Hierarchical Sampler).

The following example makes parameter `D` depend on parameter `A` being `FALSE`, and parameter `B` depend on parameter `D` being one of `"x"` or `"y"`. This introduces an implicit dependency of `B` on `A` being `FALSE` as well, because `D` does not take any value if `A` is `TRUE`.

```
ps$add_dep("D", "A", CondEqual$new(FALSE))
ps$add_dep("B", "D", CondAnyOf$new(c("x", "y")))
```

```
ps$check(list(A = FALSE, D = "x", B = 1))  # OK: all dependencies met
## [1] TRUE
ps$check(list(A = FALSE, D = "z", B = 1))  # B's dependency is not met
## [1] "Condition for 'B' not ok: D anyof x, y; instead: D=z"
ps$check(list(A = FALSE, B = 1))  # B's dependency is not met
## [1] "Condition for 'B' not ok: D anyof x, y; instead: D=<not-there>"
ps$check(list(A = FALSE, D = "z"))  # OK: B is absent
## [1] TRUE
ps$check(list(A = TRUE))  # OK: neither B nor D present
## [1] TRUE
ps$check(list(A = TRUE, D = "x", B = 1))  # D's dependency is not met
## [1] "Condition for 'D' not ok: A equal FALSE; instead: A=TRUE"
ps$check(list(A = TRUE, B = 1))  # B's dependency is not met
## [1] "Condition for 'B' not ok: D anyof x, y; instead: D=<not-there>"
```

Internally, the dependencies are represented as a `data.table`, which can be accessed listed in the **`$deps`** slot. This `data.table` can even be mutated, to e.g. remove dependencies. There are no sanity checks done when the `$deps` slot is changed this way. Therefore it is advised to be cautious.

```
ps$deps
##    id on        cond
## 1:  D  A <CondEqual>
## 2:  B  D <CondAnyOf>
```

5.4.2.3 Vector Parameters

Unlike in the old `ParamHelpers` package, there are no more vectorial parameters in paradox. Instead, it is now possible to create multiple copies of a single parameter using the `$rep` function. This creates a `ParamSet` consisting of multiple copies of the parameter, which can then (optionally) be added to another `ParamSet`.

```
ps2d = ParamDbl$new("x", lower = 0, upper = 1)$rep(2)
print(ps2d)
## ParamSet:
##         id    class lower upper levels     default
## 1: x_rep_1 ParamDbl     0     1        <NoDefault>
## 2: x_rep_2 ParamDbl     0     1        <NoDefault>
##    value
## 1:
## 2:
```

```
ps$add(ps2d)
print(ps)
## ParamSet:
##         id    class lower upper      levels     default
## 1:       A ParamLgl    NA    NA  TRUE,FALSE <NoDefault>
## 2:       B ParamInt     0    10             <NoDefault>
## 3:       C ParamDbl     0     4             <NoDefault>
## 4:       D ParamFct    NA    NA       x,y,z           y
## 5:       E ParamUty    NA    NA             <NoDefault>
## 6: x_rep_1 ParamDbl     0     1             <NoDefault>
## 7: x_rep_2 ParamDbl     0     1             <NoDefault>
##    parents value
## 1:          TRUE
## 2:       D     1
## 3:
## 4:       A
## 5:
## 6:
## 7:
```

It is also possible to use a `ParamUty` to accept vectorial parameters, which also works for parameters of variable length. A `ParamSet` containing a `ParamUty` can be used for parameter checking, but not for sampling. To sample values for a method that needs a vectorial parameter, it is advised to use a parameter transformation function that creates a vector from atomic values.

Assembling a vector from repeated parameters is aided by the parameter's `$tags`: Parameters that were generated by the `$rep()` command automatically get tagged as belonging to a group of repeated parameters.

```
ps$tags
## $A
## character(0)
##
## $B
## [1] "tag1" "tag2"
##
## $C
## character(0)
```

```
##
## $D
## character(0)
##
## $E
## character(0)
##
## $x_rep_1
## [1] "x_rep"
##
## $x_rep_2
## [1] "x_rep"
```

5.4.3 Parameter Sampling

It is often useful to have a list of possible parameter values that can be systematically iterated through, for example to find parameter values for which an algorithm performs particularly well (tuning). paradox offers a variety of functions that allow creating evenly-spaced parameter values in a "grid" design as well as random sampling. In the latter case, it is possible to influence the sampling distribution in more or less fine detail.

A point to always keep in mind while sampling is that only numerical and factorial parameters that are bounded can be sampled from, i.e. not `ParamUty`. Furthermore, for most samplers `ParamInt` and `ParamDbl` must have finite lower and upper bounds.

5.4.3.1 Parameter Designs

Functions that sample the parameter space fundamentally return an object of the `Design` class. These objects contain the sampled data as a `data.table` under the `$data` slot, and also offer conversion to a list of parameter-values using the **`$transpose()`** function.

5.4.3.2 Grid Design

The `generate_design_grid()` function is used to create grid designs that contain all combinations of parameter values: All possible values for `ParamLgl` and `ParamFct`, and values with a given resolution for `ParamInt` and `ParamDbl`. The resolution can be given for all numeric parameters, or for specific named parameters through the `param_resolutions` parameter.

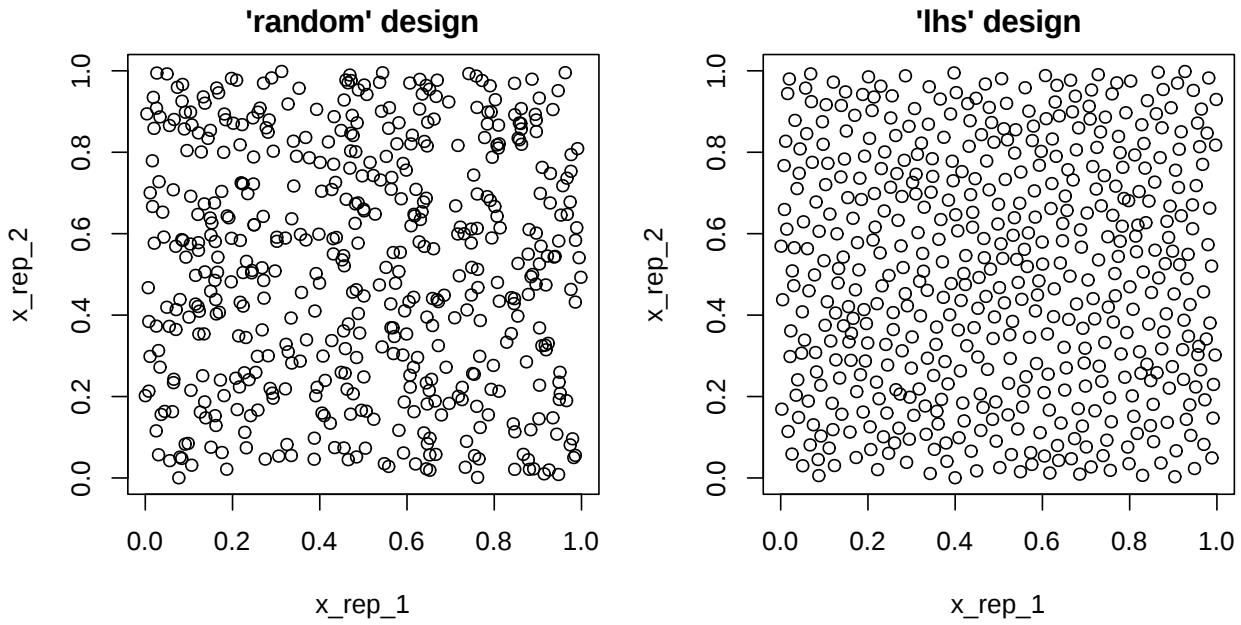
```
design = generate_design_grid(psSmall, 2)
print(design)
## <Design> with 8 rows:
##        A  B C
## 1:  TRUE  0 0
## 2:  TRUE  0 4
## 3:  TRUE 10 0
## 4:  TRUE 10 4
## 5: FALSE  0 0
## 6: FALSE  0 4
## 7: FALSE 10 0
## 8: FALSE 10 4
```

```
generate_design_grid(psSmall, param_resolutions = c(B = 1,
  C = 2))
## <Design> with 4 rows:
##    B C     A
## 1: 0 0  TRUE
## 2: 0 0 FALSE
## 3: 0 4  TRUE
## 4: 0 4 FALSE
```

5.4.3.3 Random Sampling

paradox offers different methods for random sampling, which vary in the degree to which they can be configured. The easiest way to get a uniformly random sample of parameters is `generate_design_random`. It is also possible to create "latin hypercube" sampled parameter values using **`generate_design_lhs`**, which utilizes the `lhs` package. LHS-sampling creates low-discrepancy sampled values that cover the parameter space more evenly than purely random values.

```
pvrand = generate_design_random(ps2d, 500)
pvlhs = generate_design_lhs(ps2d, 500)
```



5.4.3.4 Generalized Sampling: The `Sampler` Class

It may sometimes be desirable to configure parameter sampling in more detail. paradox uses the `Sampler` abstract base class for sampling, which has many different sub-classes that can be parameterized and combined to control the sampling process. It is even possible to create further sub-classes of the `Sampler` class (or of any of *its* subclasses) for even more possibilities.

Every `Sampler` object has a `sample()` function, which takes one argument, the number of instances to sample, and returns a `Design` object.

1D-Samplers There is a variety of samplers that sample values for a single parameter. These are `Sampler1DUnif` (uniform sampling), `Sampler1DCateg` (sampling for categorical parameters), `Sampler1DNormal` (normally distributed sampling, truncated at parameter bounds), and `Sampler1DRfun` (arbitrary 1D sampling, given a random-function). These are initialized with a single `Param`, and can then be used to sample values.

```
sampA = Sampler1DCateg$new(parA)
sampA$sample(5)
## <Design> with 5 rows:
##        A
## 1:  TRUE
## 2: FALSE
## 3:  TRUE
## 4: FALSE
## 5:  TRUE
```

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
##   FALSE   0   0   0   0   0   0   0   0   0   0   0
##   TRUE   55  54  41  49  38  54  35  42  47  46  43
##        B
## A       <NA>
##   FALSE  496
##   TRUE     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: 0.7068 0.7112
```

```
## 2: 0.9676 0.1795
## 3: 0.2373 0.5154
## 4: 0.9031 0.2450
## 5: 0.8957 0.7116
```

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

5.4.4 Parameter Transformation

While the different `Samplers` allow for a wide specification of parameter distributions, there are cases where the simplest way of getting a desired distribution is to sample parameters from a simple distribution (such as the uniform distribution) and then transform them. This can be done by assigning a function to the `$trafo` slot of a `ParamSet`. The `$trafo` function is called with two parameters:

- The list of parameter values to be transformed as `x`
- The `ParamSet` itself as `param_set`

The `$trafo` function must return a list of transformed parameter values.

The transformation is performed when calling the `$transpose` function of the `Design` object returned by a `Sampler` with the `trafo` ParamSet to `TRUE` (the default). The following, for example, creates a parameter that is exponentially distributed:

```
psexp = ParamSet$new(list(ParamDbl$new("par", 0, 1)))
psexp$trafo = function(x, param_set) {
  x$par = -log(x$par)
  x
}
design = generate_design_random(psexp, 2)
print(design)
## <Design> with 2 rows:
##       par
## 1: 0.5500
## 2: 0.5163
design$transpose()  # trafo is TRUE
## [[1]]
## [[1]]$par
## [1] 0.5978
##
##
## [[2]]
## [[2]]$par
## [1] 0.6611
```

Compare this to `$transpose()` without transformation:

```
design$transpose(trafo = FALSE)
## [[1]]
## [[1]]$par
## [1] 0.55
##
##
## [[2]]
## [[2]]$par
## [1] 0.5163
```

5.4.4.1 Transformation between Types

Usually the design created with one `ParamSet` is then used to configure other objects that themselves have a `ParamSet` which defines the values they take. The `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 levels     default value
## 1: fun ParamUty    NA    NA        <NoDefault>
```

If one wanted to sample this method, using one of four functions, a way to do this would be:

```
samplingPS = ParamSet$new(list(ParamFct$new("fun", c("mean",
  "median", "min", "max"))))

samplingPS$trafo = function(x, param_set) {
  # x$fun is a `character(1)`, in particular one of 'mean',
  # 'median', 'min', 'max'.  We want to turn it into a
  # function!
  x$fun = get(x$fun, mode = "function")
  x
}
```

```
design = generate_design_random(samplingPS, 2)
print(design)
## <Design> with 2 rows:
##    fun
## 1: min
## 2: min
```

Note that the `Design` only contains the column "`fun`" as a `character` column. To get a single value as a *function*, the `$transpose` function is used.

```
xvals = design$transpose()
print(xvals[[1]])
## $fun
## function (..., na.rm = FALSE)  .Primitive("min")
```

We can now check that it fits the requirements set by `methodPS`, and that `fun` it is in fact a function:

```
methodPS$check(xvals[[1]])
## [1] TRUE
xvals[[1]]$fun(1:10)
## [1] 1
```

Imagine now that a different kind of parametrization of the function is desired: The user wants to give a function that selects a certain quantile, where the quantile is set by a parameter. In that case the `$transpose` function could generate a function in a different way. For interpretability, the parameter is called "`quantile`" before transformation, and the "`fun`" parameter is generated on the fly.

```
samplingPS2 = ParamSet$new(list(ParamDbl$new("quantile",
  0, 1)))

samplingPS2$trafo = function(x, param_set) {
  # x$quantile is a `numeric(1)` between 0 and 1.  We want
  # to turn it into a function!
  list(fun = function(input) quantile(input, x$quantile))
}
```

```
design = generate_design_random(samplingPS2, 2)
print(design)
## <Design> with 2 rows:
##    quantile
## 1: 0.004436
## 2: 0.660597
```

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: 0x12912160>
methodPS$check(xvals[[1]])
## [1] TRUE
xvals[[1]]$fun(1:10)
## 0.4436%
##    1.04
```

5.5 Logging and Verbosity

We use the `lgr` package for logging and progress output.

Because *lgr* comes with its own exhaustive vignette, we will just briefly give examples how you can change the most important settings related to logging in mlr3.

5.5.1 Available logging levels

lgr comes with certain numeric thresholds which correspond to verbosity levels of the logging. For mlr3 the default is set to 400 which corresponds to level "info". The following ones are available:

```
library("lgr")
getOption("lgr.log_levels")
## fatal error  warn  info debug trace
##   100   200   300   400   500   600
```

5.5.2 Global Setting

lgr comes with a global option called `"lgr.default_threshold"` which can be set via `options()`. You can set a specific level in your `.Rprofile` which is then used for all packages that use the *lgr* package. This approach may not be desirable if you want to only change the logging level for mlr3.

5.5.3 Changing mlr3 logging levels

To change the setting for mlr3 only, you need to change the threshold of the mlr3 logger like this:

```
lgr::get_logger("mlr3")$set_threshold("<level>")
```

Remember that this change only applies to the current R session.

5.6 mlr -> mlr3 Transition Guide

In case you have already worked with `mlr`, you may want to quickstart with mlr3 by looking up the specific equivalent of an element of `mlr` in the new version mlr3. For this, you can use the following table. This table is not complete but should give you an overview about how mlr3 is organized.

Category	mlr	mlr3	Note
General / Helper	getCacheDir() / deleteCacheDir()	Not yet implemented	---
	configureMlr()	---	---
	getMlrOptions()	---	---
	createDummyFeatures()	Not yet implemented	mlr3pipelines
	crossover()	---	---
	downsample()	Not yet implemented	---
	generateCalibrationData()	Not yet implemented	---
	generateCritDifferencesData()	Not yet implemented	---
	generateLearningCurveData()	Not yet implemented	mlr3viz
	generatePartialDependenceData()	Not yet implemented	mlr3viz
	generateThreshVsPerfData()	Not yet implemented	mlr3viz
	getCaretParamSet()	Not used anymore	---
	reimpute() / impute()	Not yet implemented	mlr3pipelines
	fn() / fnr() / fp() / fpr()	???	
	tn() / tnr() / tp() / tpr()	???	
	summarizeColumns()	???	
	summarizeLevels()	???	

Category	mlr	mlr3	Note
Task	Task	mlr_tasks / Task	---
	SurvTask	TaskSurv	mlr3proba
	ClusterTask	mlr_tasks	---
	MultilabelTask	mlr_tasks	---
	SpatialTask	Not yet implemented	mlr3spatiotemporal
	Example tasks (iris.task,mtcars.task)	mlr_tasks$get('iris') / tsk('iris')	---
	convertMLBenchObjToTask()	Not yet implemented	mlr3
	dropFeatures()	Task$select()	---
	getTaskCosts()	Not yet implemented	---
	getTaskData()	Task$data()	---
	getTaskDesc() / getTaskDescription()	Task$print()	---
	getTaskFeatureNames()	Task$feature_names	---
	getTaskFormula()	Task$formula	---
	getTaskId()	Task$id	---
	getTaskNFeats()	length(Task$feature_names)	---
	getTaskSize()	Task$nrow()	---
	getTaskTargetNames()	Task$target_names	---
	getTaskTargets()	as.data.table(Task)[,Task$feature_names,with = FALSE]	---
	getTaskType()	Task$task_type	---
	oversample() / undersample()		---

Category	mlr	mlr3	Note
Learner	helpLearner()	Not yet implemented	---
	helpLearnerParam()	Not yet implemented	---
	getLearnerId()	Learner$id	---
	setLearnerId()	Learner$id	---
	getLearnerModel()	Learner$model	---
	getLearnerNote()	Not used anymore	---
	getLearnerPackages()	Learner$packages	---
	getLearnerParVals() / getLearnerParamSet()	Learner$param_set	---
	getLearnerPredictType()	Learner$predict_type	---
	getLearnerShortName()	Learner$predict_type	---
	getLearnerType()	Learner$Type	---
	setPredictType()	Learner$Type	---
	getLearnerProperties	???	---
	getParamSet()	Learner$param_set	---
	trainLearner()	Learner$train()	---
	predictLearner()	Learner$predict()	---
	makeRLearner*()	Learner	---
	generateLearningCurveData()	Not yet implemented	mlr3viz
	FailureModel	---	---
	getFailureModelDump()	---	---
	getFailureModelMsg()	---	---
	isFailureModel()	---	---
	makeLearner() / makeLearners()	???	---

Category	mlr	mlr3	Note
Train/Predict/Resample	train()	Experiment$train()	---
	predict()	Experiment$predict()	---
	performance()	Experiment$score()	---
	makeResampleDesc()	Resampling	mlr_resamplings
	resample()	resample()	---
	ResamplePrediction	ResampleResult	---
	Aggregation / makeAggregation	Not yet implemented	---
	asROCRPrediction()	Not yet implemented	---
	ConfusionMatrix / getConfMatrix() / calculateConfusionMatrix()	Not yet implemented	---
	calculateROCMeasures()	Not yet implemented	---
	estimateRelativeOverfitting()	Not yet implemented	---
	estimateResidualVariance()	Not yet implemented	---
	getDefaultMeasure()		---
	getMeasureProperties()	???	---
	getPredictionResponse() / getPredictionSE() / getPredictionTruth()	???	---
	getPredictionDump()	???	---
	getPredictionTaskDesc()	???	---
	getRRDump()	???	---
	getRRPredictionList()	???	---
	getRRPredictions()	ResampleResult$prediction	---
	getRRTaskDesc() / getRRTaskDescription()	ResampleResult$task$print()	---

Category	mlr	mlr3	Note
Benchmark	benchmark()	benchmark()	---
	batchmark() / reduceBatchmarkResults()	not used anymore	---
	BenchmarkResult	BenchmarkResult	---
	convertBMRToRankMatrix()	Not yet implemented	---
	convertMLBenchObjToTask()	Not yet implemented	---
	getBMRAggrPerformances()	BenchmarkResult$aggregated()	---
	getBMRFeatSelResults()	Not yet implemented	mlr3filters
	getBMRFilteredFeatures()	Not yet implemented	mlr3filters
	getBMRLearners() / getBMRLearnerIds() / getBMRLearnerShortNames()	BenchmarkResult$learners	---
	getBMRMeasures() / getBMRMeasureIds()	BenchmarkResult$measures	---
	getBMRModels()	BenchmarkResult$data$learner[[1]]$model	---
	getBMRPerformances()	BenchmarkResult$data$performance	---
	getBMRTaskDescriptions() / getBMRTaskDescs() / getBMRTaskIds()	BenchmarkResult$tasks	---
	getBMRTuneResults()	Not yet implemented	---
	getBMRPredictions()	Not yet implemented	---
	friedmanTestBMR()	Not yet implemented	---
	mergeBenchmarkResults()	BenchmarkResult$combine()	---
	plotBMRBoxplots()	Not yet implemented	mlr3viz
	plotBMRRanksAsBarChart()	Not yet implemented	mlr3viz
	plotBMRSummary()	Not yet implemented	mlr3viz
	plotResiduals()	Not yet implemented	mlr3viz
Parameter Specification	ParamHelpers::makeNumericParam()	ParamDbl$new()	paradox
	ParamHelpers::makeNumericVectorParam()	ParamDbl$new()	paradox
	ParamHelpers::makeIntegerParam()	paradox::ParamInt$new()	paradox
	ParamHelpers::makeIntegerVectorParam()	paradox::ParamInt$new()	paradox
	ParamHelpers::makeDiscreteParam()	paradox::ParamFct$new()	paradox
	ParamHelpers::makeDiscreteVectorParam()	paradox::ParamFct$new()	paradox
	ParamHelpers::makeLogicalParam()	paradox::ParamLgl$new()	paradox
	ParamHelpers::makeLogicalVectorParam()	paradox::ParamLgl$new()	paradox
Preprocessing	---	---	---
	---	---	---

Category	mlr	mlr3	Note
Feature Selection	makeFeatSelControlExhaustive()	Not yet implemented	mlr3filters
	makeFeatSelControlRandom()	Not yet implemented	mlr3filters
	makeFeatSelControlSequential()	Not yet implemented	mlr3filters
	makeFeatSelControlGA()	Not yet implemented	mlr3filters
	makeFilter()	Filter$new()	mlr3filters
	FeatSelResult	Not yet implemented	mlr3filters
	listFilterMethods()	mlr_filters	mlr3filters
	analyzeFeatSelResult()	Not yet implemented	mlr3filters
	getBMRFeatSelResults()	Not yet implemented	mlr3filters
	getBMRFilteredFeatures()	Not yet implemented	mlr3filters
	getFeatSelResult()	Not yet implemented	mlr3filters
	getFeatureImportance()	Not yet implemented	mlr3filters
	getFilteredFeatures()	Not yet implemented	mlr3filters
	makeFeatSelWrapper()	Not used anymore	mlr3filters
	makeFilterWrapper()	Not used anymore	mlr3filters
	getResamplingIndices()	Not yet implemented	
	selectFeatures()	Not yet implemented	mlr3filters
	filterFeatures()	Filter$filter_*()	mlr3filters
	generateFilterValuesData()	Filter$calculate()	mlr3filters
Tuning	getTuneResult()	Not yet implemented	mlr3tuning
	getTuneResultOptPath()	Not yet implemented	mlr3tuning
	makeTuneControl*()	Tuner	mlr3tuning
	makeTuneMultiCritControl*()	Tuner	mlr3tuning

Category	mlr	mlr3	Note
Parallelization	ParallelMap::parallelStart*(), parallelMap::parallelStop()	future::plan() / future	
Plotting	plotBMRBoxplots()	Not yet implemented	mlr3viz
	plotBMRRanksAsBarChart()	Not yet implemented	mlr3viz
	plotBMRSummary()	Not yet implemented	mlr3viz
	plotCalibration()	Not yet implemented	mlr3viz
	plotCritDifferences()	Not yet implemented	mlr3viz
	plotFilterValues()	Not yet implemented	mlr3viz
	plotHyperParsEffect()	Not yet implemented	mlr3viz
	plotLearnerPrediction()	Not yet implemented	mlr3viz
	plotLearningCurve()	Not yet implemented	mlr3viz
	plotPartialDependence()	Not yet implemented	mlr3viz
	plotResiduals()	Not yet implemented	mlr3viz
	plotROCCurves()	Not yet implemented	mlr3viz
	plotThreshVsPerf()	Not yet implemented	mlr3viz
	plotTuneMultiCritResult()	Not yet implemented	mlr3viz
FDA	extractFDAFPCA()	Not yet implemented	mlr3fda
	extractFDAFourier()	Not yet implemented	mlr3fda
	extractFDAMultiResFeatures()	Not yet implemented	mlr3fda
	extractFDAWavelets()	Not yet implemented	mlr3fda

6 Extending

This chapter gives instructions on how to extend mlr3 and its extension packages with custom objects.

The approach is always the same:

1. determine the base class you want to inherit from,
2. extend the class with your custom functionality,
3. test your implementation
4. (optionally) add new object to the respective `Dictionary`.

The chapter Create a new learner illustrates the steps needed to create a custom learner in mlr3.

6.1 Extending with Learners

Here, we show how to create a custom `LearnerClassif` step-by-step.

Preferably, you checkout our template package for new learners. Alternatively, here is a template snippet for a new classification learner:

```
LearnerClassifYourLearner = R6::R6Class("LearnerClassifYourLearner",
  inherit = LearnerClassif,
  public = list(
    initialize = function(id = "classif.yourlearner") {
      super$initialize(
        id = id,
        param_set = ParamSet$new(),
        predict_types = ,
        feature_types = ,
        properties = ,
        packages = ,
      )
    },

    train = function(task) {

    },
    predict = function(task) {

    }
  )
)
```

In the first line of the template, we create a new R6 class with class "LearnerClassifYourLearner". The next line determines the parent class: As we want to create a classification learner, we obviously want to inherit from LearnerClassif.

A learner consists of three parts:

1. Meta information about the learners
2. A train_internal() function which takes a (filtered) TaskClassif and returns a model
3. A predict_internal() function which operates on the model in self$model (stored during $train()) and a (differently subsetted) TaskClassif to return a named list of predictions.

6.1.1 Meta-information

In the constructor function initialize() the constructor of the super class LearnerClassif is called with meta information about the learner we want to construct. This includes:

- id: The id of the new learner.
- packages: Set of required packages to run the learner.
- param_set: A set of hyperparameters and their description, provided as paradox::ParamSet. It is perfectly fine to add no parameters here for a first draft. For each hyperparameter you want to add, you have to select the appropriate class:
  - paradox::ParamLgl for scalar logical hyperparameters.
  - paradox::ParamInt for scalar integer hyperparameters.
  - paradox::ParamDbl for scalar numeric hyperparameters.
  - paradox::ParamFct for scalar factor hyperparameters (this includes characters).
  - paradox::ParamUty for everything else.
- predict_types: Set of predict types the learner is capable of. These differ depending on the type of the learner.
  - LearnerClassif
    - response: Only predicts a class label for each observation in the test set.
    - prob: Also predicts the posterior probability for each class for each observation in the test set.
  - LearnerRegr
    - response: Only predicts a numeric response for each observation in the test set.
    - se: Also predicts the standard error for each value of response for each observation in the test set.
- feature_types: Set of feature types the learner can handle. See mlr_reflections", text = "mlr_reflections$task_feature_ty for feature types supported by mlr3.
- properties: Set of properties of the learner. Possible properties include:
  - "twoclass": The learner works on binary classification problems.
  - "multiclass": The learner works on multi-class classification problems.
  - "missings": The learner can natively handle missing values.
  - "weights": The learner can work on tasks which have observation weights / case weights.
  - "parallel": The learner can be parallelized, e.g. via threading.
  - "importance": The learner supports extracting importance values for features. If this property is set, you must also implement a public method importance() to retrieve the importance values from the model.
  - "selected_features": The learner supports extracting the features which where used. If this property is set, you must also implement a public method selected_features() to retrieve the set of used features from the model.

For a simplified rpart::rpart(), the initialization could look like this:

```
initialize = function(id = "classif.rpart") {
  ps = ParamSet$new(list(ParamDbl$new(id = "cp", default = 0.01,
    lower = 0, upper = 1, tags = "train"), ParamInt$new(id = "xval",
    default = 10L, lower = 0L, tags = "train")))
  ps$values = list(xval = 0L)

  super$initialize(id = id, packages = "rpart", feature_types = c("logical",
    "integer", "numeric", "factor"), predict_types = c("response",
    "prob"), param_set = ps, properties = c("twoclass",
    "multiclass", "weights", "missings"))
}
```

We only have specified a small subset of the available hyperparameters:

- The complexity `"cp"` is numeric, its feasible range is `[0,1]`, it defaults to `0.01` and the parameter is used during `"train"`.
- The complexity `"xval"` is integer, its lower bound `0`, its default is `0` and the parameter is also used during `"train"`. Note that we have changed the default here from `10` to `0` to save some computation time. This is **not** done by setting a different `default` in `ParamInt$new()`, but instead by setting the value explicitly.

6.1.2 Train function

We continue the to adept the template for a `rpart::rpart()` learner, and now tackle the `train_internal()` function. The train function takes a `Task` as input and must return an arbitrary model. First, we write something down that works completely without mlr3:

```
data = iris
model = rpart::rpart(Species ~ ., data = iris, xval = 0)
```

In the next step, we replace the data frame `data` with a `Task`:

```
task = tsk("iris")
model = rpart::rpart(Species ~ ., data = task$data(), xval = 0)
```

The target variable `"Species"` is still hard-coded and specific to the task. This is unnecessary, as the information about the target variable is stored in the task:

```
task$target_names
## [1] "Species"
task$formula()
## Species ~ .
## NULL
```

We can adapt our code accordingly:

```
rpart::rpart(task$formula(), data = task$data(), xval = 0)
## n= 150
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
## 1) root 150 100 setosa (0.33333 0.33333 0.33333)
##   2) Petal.Length< 2.45 50   0 setosa (1.00000 0.00000 0.00000) *
##   3) Petal.Length>=2.45 100  50 versicolor (0.00000 0.50000 0.50000)
##     6) Petal.Width< 1.75 54   5 versicolor (0.00000 0.90741 0.09259) *
##     7) Petal.Width>=1.75 46   1 virginica (0.00000 0.02174 0.97826) *
```

The last thing missing is the handling of hyperparameters. Instead of the hard-coded `xval`, we query the hyperparameter settings from the `Learner` itself.

To illustrate this, we quickly construct the tree learner from the mlr3 package, and use the method `get_value()` from the `ParamSet` to retrieve all set hyperparameters with tag `"train"`.

```
self = lrn("classif.rpart")
self$param_set$get_values(tags = "train")
## $xval
## [1] 0
```

To pass all hyperparameters down to the model fitting function, we recommend to use either `do.call` or the function `mlr3misc::invoke()`.

```
pars = self$param_set$get_values(tags = "train")
mlr3misc::invoke(rpart::rpart, task$formula(), data = task$data(),
  .args = pars)
## n= 150
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
## 1) root 150 100 setosa (0.33333 0.33333 0.33333)
##   2) Petal.Length< 2.45 50   0 setosa (1.00000 0.00000 0.00000) *
##   3) Petal.Length>=2.45 100  50 versicolor (0.00000 0.50000 0.50000)
##     6) Petal.Width< 1.75 54   5 versicolor (0.00000 0.90741 0.09259) *
##     7) Petal.Width>=1.75 46   1 virginica (0.00000 0.02174 0.97826) *
```

In the final learner, `self` will of course reference the learner itself. In the last step, we wrap everything in a function.

```
train_internal = function(task) {
  pars = self$param_set$get_values(tags = "train")
  mlr3misc::invoke(rpart::rpart, task$formula(), data = task$data(),
    .args = pars)
}
```

6.1.3 Predict function

The internal predict function `predict_internal` also operates on a `Task` as well as on the model stored during `train()` in `self$model`. The return value is a `Prediction` object. We proceed analogously to the section on the train function. We start with a version without any mlr3 objects and continue to replace objects until we have reached the desired interface:

```
# inputs:
task = tsk("iris")
self = list(model = rpart::rpart(task$formula(), data = task$data()))

data = iris
response = predict(self$model, newdata = data, type = "class")
prob = predict(self$model, newdata = data, type = "prob")
```

The `rpart::predict.rpart()` function predicts class labels if argument `type` is set to to `"class"`, and class probabilities if set to `"prob"`.

Next, we transition from `data` to a `task` again and construct a proper `PredictionClassif` object to return. Additionally, as we do not want to run the prediction twice, we differentiate what type of prediction is requested by querying the set predict type of the learner. The complete `predict_internal` function looks like this:

```
predict_internal = function(task) {
  self$predict_type = "response"
  response = prob = NULL

  if (self$predict_type == "response") {
    response = predict(self$model, newdata = task$data(),
      type = "class")
  } else {
    prob = predict(self$model, newdata = task$data(),
      type = "prob")
  }

  PredictionClassif$new(task, response = response, prob = prob)
}
```

Note that if the learner would need to handle hyperparameters during the predict step, we would proceed analogously to the `train()` step and use `self$params("predict` in combination with `mlr3misc::invoke()`.

Also note that you cannot rely on the column order of the data returned by `task$data()`, i.e. the order of columns may be different from the order of the columns during `$train()`. You have to make sure that your learner accesses columns by name, not by position (like some algorithms with a matrix interface do). You may have to restore the order manually here, see "classif.svm" for an example.

6.1.4 Final learner

```
LearnerClassifYourRpart = R6::R6Class("LearnerClassifYourRpart",
  inherit = LearnerClassif,
  public = list(
    initialize = function(id = "classif.rpart") {
      ps = ParamSet$new(list(
        ParamDbl$new(id = "cp", default = 0.01, lower = 0, upper = 1, tags = "train"),
        ParamInt$new(id = "xval", default = 0L, lower = 0L, tags = "train")
      ))
      ps$values = list(xval = 0L)

      super$initialize(
        id = id,
        packages = "rpart",
        feature_types = c("logical", "integer", "numeric", "factor"),
        predict_types = c("response", "prob"),
        param_set = ps,
        properties = c("twoclass", "multiclass", "weights", "missings")
      )
    },

    train_internal = function(task) {
      pars = self$param_set$get_values(tag = "train")
      mlr3misc::invoke(rpart::rpart, task$formula(), data = task$data(), .args = pars)
    },

    predict_internal = function(task) {
      self$predict_type = "response"
      response = prob = NULL

      if (self$predict_type == "response") {
        response = predict(self$model, newdata = task$data(), type = "class")
      } else {
        prob = predict(self$model, newdata = task$data(), type = "prob")
      }
      PredictionClassif$new(task, response = response, prob = prob)
    }
  )
)

lrn = LearnerClassifYourRpart$new()
print(lrn)
## <LearnerClassifYourRpart:classif.rpart>
## * Model: -
## * Parameters: xval=0
## * Packages: rpart
## * Predict Type: response
## * Feature types: logical, integer, numeric,
##   factor
## * Properties: missings, multiclass, twoclass,
```

```
##   weights
```

To run some basic tests:

```
task = tsk("iris")
lrn$train(task)
p = lrn$predict(task)
p$confusion
##             truth
## response     setosa versicolor virginica
##   setosa         50          0         0
##   versicolor      0         49         5
##   virginica       0          1        45
```

To run a bunch of automatic tests, you may source some auxiliary scripts from the unit tests of mlr3:

```
helper = list.files(system.file("testthat", package = "mlr3"),
  pattern = "^helper.*\\.[rR]", full.names = TRUE)
ok = lapply(helper, source)
stopifnot(run_autotest(lrn))
```

6.2 Extending with mlr3pipelines

This tutorial 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 = mlr_tasks$get("iris")
task$data()
##        Species Petal.Length Petal.Width Sepal.Length
##   1:    setosa          1.4         0.2          5.1
##   2:    setosa          1.4         0.2          4.9
##   3:    setosa          1.3         0.2          4.7
##   4:    setosa          1.5         0.2          4.6
##   5:    setosa          1.4         0.2          5.0
##  ---
## 146: virginica          5.2         2.3          6.7
## 147: virginica          5.0         1.9          6.3
## 148: virginica          5.2         2.0          6.5
## 149: virginica          5.4         2.3          6.2
## 150: virginica          5.1         1.8          5.9
##      Sepal.Width
##   1:         3.5
##   2:         3.0
```

```
##   3:         3.2
##   4:         3.1
##   5:         3.6
##  ---
## 146:         3.0
## 147:         2.5
## 148:         3.0
## 149:         3.4
## 150:         3.0
```

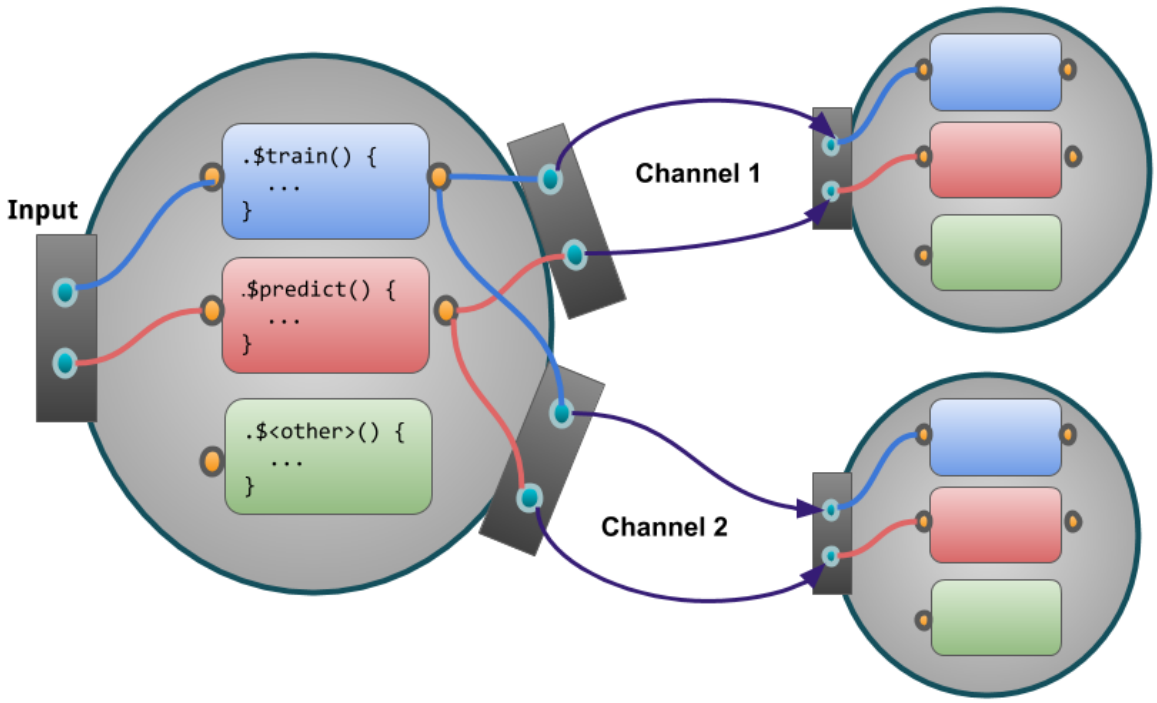
mlr3pipelines is fundamentally built around R6. When planning to create custom PipeOp objects, it can only help to familiarize yourself with it.

In principle, all a PipeOp must do is inherit from the PipeOp R6 class and implement the train() and predict() functions. There are, however, several auxiliary subclasses that can make the creation of *certain* operations much easier.

6.2.1 General Case Example: PipeOpCopy

A very simple yet useful PipeOp is PipeOpCopy, which takes a single input and creates a variable number of output channels, all of which receive a copy of the input data. It is a simple example that showcases the important steps in defining a custom PipeOp. We will show a simplified version here, **PipeOpCopyTwo**, that creates exactly two copies of its input data.

The following figure visualizes how our PipeOp is situated in the Pipeline and the significant in- and outputs.



6.2.1.1 First Steps: Inheriting from `PipeOp`

The first part of creating a custom `PipeOp` is inheriting from `PipeOp`. We make a mental note that we need to implement a `train()` and a `predict()` function, and that we probably want to have an `initialize()` as well:

```
PipeOpCopyTwo = R6::R6Class("PipeOpCopyTwo",
  inherit = mlr3pipelines::PipeOp,
  public = list(
    initialize = function(id = "copy.two") {
      ....
    },

    train = function(inputs) {
      ....
    },

    predict = function(inputs) {
      ....
    }
  )
)
```

6.2.1.2 Channel Definitions

We need to tell the `PipeOp` the layout of its channels: How many there are, what their names are going to be, and what types are acceptable. This is done on initialization of the `PipeOp` (using a `super$initialize` call) by giving the `input` and `output` `data.table` objects. These must have three columns: a `"name"` column giving the names of input and output channels, and a `"train"` and `"predict"` column naming the class of objects we expect during training and prediction as input / output. A special value for these classes is `"*"`, which indicates that any class will be accepted; our simple copy operator accepts any kind of input, so this will be useful. We have only one input, but two output channels.

By convention, we name a single channel `"input"` or `"output"`, and a group of channels [`"input1"`, `"input2"`, ...], unless there is a reason to give specific different names. Therefore, our `input` `data.table` will have a single row `<"input", "*", "*">`, and our `output` table will have two rows, `<"output1", "*", "*">` and `<"output2", "*", "*">`.

All of this is given to the `PipeOp` creator. Our `initialize()` will thus look as follows:

```
initialize = function(id = "copy.two") {
  input = data.table::data.table(name = "input", train = "*",
    predict = "*")
  # the following will create two rows and automatically
  # fill the `train` and `predict` cols with '*'
  output = data.table::data.table(name = c("output1", "output2"),
    train = "*", predict = "*")
  super$initialize(id, input = input, output = output)
}
```

6.2.1.3 Train and Predict

Both `train()` and `predict()` will receive a `list` as input and must give a `list` in return. According to our `input` and `output` definitions, we will always get a list with a single element as input, and will need to return a list with two elements. Because all we want to do is create two copies, we will just create the copies using `c(inputs, inputs)`.

Two things to consider:

- The `train()` function must always modify the `self$state` variable to something that is not `NULL` or `NO_OP`. This is because the `$state` slot is used as a signal that `PipeOp` has been trained on data, even if the state itself is not important to the `PipeOp` (as in our case). Therefore, our `train()` will set `self$state = list()`.
- It is not necessary to "clone" our input or make deep copies, because we don't modify the data. However, if we were changing a reference-passed object, for example by changing data in a `Task`, we would have to make a deep copy first. This is because a `PipeOp` may never modify its input object by reference.

Our `train()` and `predict()` functions are now:

```
train = function(inputs) {
  self$state = list()
  c(inputs, inputs)
}
```

```
predict = function(inputs) {
  c(inputs, inputs)
}
```

6.2.1.4 Putting it Together

The whole definition thus becomes

```
PipeOpCopyTwo = R6::R6Class("PipeOpCopyTwo",
  inherit = mlr3pipelines::PipeOp,
  public = list(
    initialize = function(id = "copy.two") {
      super$initialize(id,
        input = data.table::data.table(name = "input", train = "*", predict = "*"),
        output = data.table::data.table(name = c("output1", "output2"),
                            train = "*", predict = "*")
      )
    },

    train = function(inputs) {
      self$state = list()
      c(inputs, inputs)
    },
```

```
    predict = function(inputs) {
      c(inputs, inputs)
    }
  )
)
```

We can create an instance of our `PipeOp`, put it in a graph, and see what happens when we train it on something:

```
library(mlr3pipelines)
poct = PipeOpCopyTwo$new()
gr = Graph$new()
gr$add_pipeop(poct)

print(gr)
## Graph with 1 PipeOps:
##        ID         State sccssors prdcssors
##  copy.two <<UNTRAINED>>

result = gr$train(task)

str(result)
## List of 2
##  $ copy.two.output1:Classes 'TaskClassif', 'TaskSupervised', 'Task', 'R6' <TaskClassif:iris>
##  $ copy.two.output2:Classes 'TaskClassif', 'TaskSupervised', 'Task', 'R6' <TaskClassif:iris>
```

6.2.2 Special Case: Preprocessing

Many PipeOps perform an operation on exactly one `Task`, and return exactly one `Task`. They may even not care about the "Target" / "Outcome" variable of that task, and only do some modification of some input data. However, it is usually important to them that the `Task` on which they perform prediction has the same data columns as the `Task` on which they train. For these cases, the auxiliary base class `PipeOpTaskPreproc` exists. It inherits from `PipeOp` itself, and other PipeOps should use it if they fall in the kind of use-case named above.

When inheriting from `PipeOpTaskPreproc`, one must either implement the `train_task` and `predict_task` functions, or the `train_dt`, `predict_dt` functions, depending on whether wants to operate on a `Task` object or on `data.tables`. In the second case, one can optionally also overload the `select_cols` function, which chooses which of the incoming `Task`'s features are given to the `train_dt` / `predict_dt` functions.

The following will show two examples: `PipeOpDropNA`, which removes a `Task`'s rows with missing values during training (and implements `train_task` and `predict_task`), and `PipeOpScale`, which scales a `Task`'s numeric columns (and implements `train_dt`, `predict_dt`, and `select_cols`).

6.2.2.1 Example: `PipeOpDropNA`

Dropping rows with missing values may be important when training a model that can not handle them.

Because mlr3 `Tasks` only contain a view to the underlying data, it is not necessary to modify data to remove rows with missing values. Instead, the rows can be removed using the `Task`'s `$filter` method, which modifies the `Task` in-place.

This is done in the `train_task` function. We take care that we also set the `$state` slot to signal that the `PipeOp` was trained.

The `predict_task` function does not need to do anything; removing missing values during prediction is not as useful, since learners that cannot handle them will just ignore the respective rows. Furthermore, mlr3 expects a `Learner` to always return just as many predictions as it was given input rows, so a `PipeOp` that removes `Task` rows during training can not be used inside a `GraphLearner`.

When we inherit from `PipeOpTaskPreproc`, it sets the `input` and `output` `data.tables` for us to only accept a single `Task`. The only thing we do during `initialize()` is therefore to set an `id` (which can optionally be changed by the user).

The complete `PipeOpDropNA` can therefore be written as follows. Note that it inherits from `PipeOpTaskPreproc`, unlike the `PipeOpCopyTwo` example from above:

```
PipeOpDropNA = R6::R6Class("PipeOpDropNA",
  inherit = mlr3pipelines::PipeOpTaskPreproc,
  public = list(
    initialize = function(id = "drop.na") {
      super$initialize(id)
    },

    train_task = function(task) {
      self$state = list()
      featuredata = task$data(cols = task$feature_names)
      exclude = apply(is.na(featuredata), 1, any)
      task$filter(task$row_ids[!exclude])
    },

    predict_task = function(task) {
      # nothing to be done
      task
    }
  )
)
```

To test this `PipeOp`, we create a small task with missing values:

```
smalliris = iris[(1:5) * 30, ]
smalliris[1, 1] = NA
smalliris[2, 2] = NA
sitask = TaskClassif$new("smalliris", as_data_backend(smalliris),
  "Species")
print(sitask$data())
##       Species Petal.Length Petal.Width Sepal.Length
## 1:     setosa          1.6         0.2           NA
## 2: versicolor          3.9         1.4          5.2
## 3: versicolor          4.0         1.3          5.5
## 4:  virginica          5.0         1.5          6.0
## 5:  virginica          5.1         1.8          5.9
##    Sepal.Width
```

```
## 1:         3.2
## 2:          NA
## 3:         2.5
## 4:         2.2
## 5:         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
## 1: versicolor          4.0         1.3          5.5
## 2:  virginica          5.0         1.5          6.0
## 3:  virginica          5.1         1.8          5.9
##    Sepal.Width
## 1:         2.5
## 2:         2.2
## 3:         3.0
```

6.2.2.2 Example: `PipeOpScaleAlways`

An often-applied preprocessing step is to simply **center** and/or **scale** the data to mean 0 and standard deviation 1. This fits the `PipeOpTaskPreproc` pattern quite well. Because it always replaces all columns that it operates on, and does not require any information about the task's target, it only needs to overload the `train_dt` and `predict_dt` functions. This saves some boilerplate-code from getting the correct feature columns out of the task, and replacing them after modification.

Because scaling only makes sense on numeric features, we want to instruct `PipeOpTaskPreproc` to give us only these numeric columns. We do this by overloading the `select_cols` function: It is called by the class to determine which columns to give to `train_dt` and `predict_dt`. Its input is the `Task` that is being transformed, and it should return a `character` vector of all features to work with. When it is not overloaded, it uses all columns; instead, we will set it to only give us numeric columns. Because the `levels()` of the data table given to `train_dt` and `predict_dt` may be different from the levels `task`'s levels, these functions must also take a `levels` argument that is a named list of column names indicating their levels. When working with numeric data, this argument can be ignored, but it should be used instead of `levels(dt[[column]])` for factorial or character columns.

This is the first `PipeOp` where we will be using the `$state` slot for something useful: We save the centering offset and scaling coefficient and use it in `$predict()`!

For simplicity, we are not using hyperparameters and will always scale and center all data. Compare this `PipeOpScaleAlways` operator to the one defined inside the mlr3pipelines package, `PipeOpScale`, defined in `PipeOpScale.R`.

```
PipeOpScaleAlways = R6::R6Class("PipeOpScaleAlways",
  inherit = mlr3pipelines::PipeOpTaskPreproc,
  public = list(
    initialize = function(id = "scale.always") {
      super$initialize(id = id)
    },

    select_cols = function(task) {
      task$feature_types[type == "numeric", id]
    },

    train_dt = function(dt, levels, target) {
      sc = scale(as.matrix(dt))
      self$state = list(
        center = attr(sc, "scaled:center"),
        scale = attr(sc, "scaled:scale")
      )
      sc
    },

    predict_dt = function(dt, levels) {
      t((t(dt) - self$state$center) / self$state$scale)
    }
  )
)
```

(Note for the observant: If you check `PipeOpScale.R` from the mlr3pipelines package, you will notice that is uses "`get("type"`" and "`get("id"`" instead of "`type`" and "`id`", because the static code checker on CRAN would otherwise complain about references to undefined variables. This is a "problem" with `data.table` and not exclusive to mlr3pipelines.)

We can, again, create a new `Graph` that uses this `PipeOp` to test it. Compare the resulting data to the original "iris" `Task` data printed at the beginning:

```
gr = Graph$new()
gr$add_pipeop(PipeOpScaleAlways$new())

result = gr$train(task)

result[[1]]$data()
##        Species Petal.Length Petal.Width Sepal.Length
##   1:    setosa      -1.3358     -1.3111     -0.89767
##   2:    setosa      -1.3358     -1.3111     -1.13920
##   3:    setosa      -1.3924     -1.3111     -1.38073
##   4:    setosa      -1.2791     -1.3111     -1.50149
##   5:    setosa      -1.3358     -1.3111     -1.01844
##  ---
## 146: virginica       0.8169      1.4440      1.03454
## 147: virginica       0.7036      0.9192      0.55149
## 148: virginica       0.8169      1.0504      0.79301
```

```
## 149: virginica       0.9302      1.4440      0.43072
## 150: virginica       0.7602      0.7880      0.06843
##      Sepal.Width
##   1:     1.01560
##   2:    -0.13154
##   3:     0.32732
##   4:     0.09789
##   5:     1.24503
##  ---
## 146:    -0.13154
## 147:    -1.27868
## 148:    -0.13154
## 149:     0.78617
## 150:    -0.13154
```

6.2.3 Special Case: Preprocessing with Simple Train

It is possible to make even further simplifications for many `PipeOps` that perform mostly the same operation during training and prediction. The point of `Task` preprocessing is often to modify the training data in mostly the same way as prediction data (but in a way that *may* depend on training data).

Consider constant feature removal, for example: The goal is to remove features that have no variance, or only a single factor level. However, what features get removed must be decided during *training*, and may only depend on training data. Furthermore, the actual process of removing features is the same during training and prediction.

A simplification to make is therefore to have a function `get_state(task)` which sets the `$state` slot during training, and a `transform(task)` function, which gets called both during training *and* prediction. This is done in the `PipeOpTaskPreprocSimple` class. Just like `PipeOpTaskPreproc`, one can inherit from this and overload these functions to get a `PipeOp` that performs preprocessing with very little boilerplate code.

Just like `PipeOpTaskPreproc`, `PipeOpTaskPreprocSimple` offers the possibility to instead overload the `get_state_dt(dt, levels)` and `transform_dt(dt, levels)` functions (and optionally, again, the `select_cols(task)` function) to operate on `data.table` feature data instead of the whole `Task`.

Even some methods that do not use `PipeOpTaskPreprocSimple` *could* work in a similar way: The `PipeOpScaleAlways` example from above will be shown to also work with this paradigm.

6.2.3.1 Example: `PipeOpDropConst`

A typical example of a preprocessing operation that does almost the same operation during training and prediction is an operation that drops features depending on a criterion that is evaluated during training. One simple example of this is dropping constant features. Because the mlr3 `Task` class offers a flexible view on underlying data, it is most efficient to drop columns from the task directly using its `$select()` function, so the `get_state_dt(dt, levels)` / `transform_dt(dt, levels)` functions will *not* get used; instead we overload the `get_state(task)` and `transform(task)` functions.

The `get_state()` function's result is saved to the `$state` slot, so we want to return something that is useful for dropping features. We choose to save the names of all the columns that have nonzero variance. For brevity, we use `length(unique(column)) > 1` to check whether more than one distinct value is present; a more sophisticated version could have a tolerance parameter for numeric values that are very close to each other.

The `transform()` function is evaluated both during training *and* prediction, and can rely on the `$state` slot being present. All it does here is call the `Task$select` function with the columns we chose to keep.

The full `PipeOp` could be written as follows:

```
PipeOpDropConst = R6::R6Class("PipeOpDropConst",
  inherit = mlr3pipelines::PipeOpTaskPreprocSimple,
  public = list(
    initialize = function(id = "drop.const") {
      super$initialize(id = id)
    },

    get_state = function(task) {
      data = task$data(cols = task$feature_names)
      nonconst = sapply(data, function(column) length(unique(column)) > 1)
      list(cnames = colnames(data)[nonconst])
    },

    transform = function(task) {
      task$select(self$state$cnames)
    }
  )
)
```

This can be tested using the first five rows of the "Iris" `Task`, for which one feature (`"Petal.Width"`) is constant:

```
irishead = task$clone()$filter(1:5)
irishead$data()
##    Species Petal.Length Petal.Width Sepal.Length
## 1:  setosa          1.4         0.2          5.1
## 2:  setosa          1.4         0.2          4.9
## 3:  setosa          1.3         0.2          4.7
## 4:  setosa          1.5         0.2          4.6
## 5:  setosa          1.4         0.2          5.0
##    Sepal.Width
## 1:         3.5
## 2:         3.0
## 3:         3.2
## 4:         3.1
## 5:         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"
## [4] "Sepal.Width"
##
## $intasklayout
##              id    type
## 1: Petal.Length numeric
## 2:  Petal.Width numeric
## 3: Sepal.Length numeric
## 4:  Sepal.Width numeric
##
## $outtasklayout
##              id    type
## 1: Petal.Length numeric
## 2: Sepal.Length numeric
## 3:  Sepal.Width numeric
```

```
dropped_predict = gr$predict(task)[[1]]

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

6.2.3.2 Example: `PipeOpScaleAlwaysSimple`

This example will show how a `PipeOpTaskPreprocSimple` can be used when only working on feature data in form of a `data.table`. Instead of calling the `scale()` function, the `center` and `scale` values are calculated directly and saved to the `$state` slot. The `transform_dt` function will then perform the same operation during both training and prediction: subtract the `center` and divide by the `scale` value. As in the `PipeOpScaleAlways` example above, we use `select_cols()` so that we only work on numeric columns.

```
PipeOpScaleAlwaysSimple = R6::R6Class("PipeOpScaleAlwaysSimple",
  inherit = mlr3pipelines::PipeOpTaskPreprocSimple,
  public = list(
    initialize = function(id = "scale.always.simple") {
      super$initialize(id = id)
    },

    select_cols = function(task) {
      task$feature_types[type == "numeric", id]
    },

    get_state_dt = function(dt, levels, target) {
      list(
        center = sapply(dt, mean),
        scale = sapply(dt, sd)
      )
    },

    transform_dt = function(dt, levels) {
      t((t(dt) - self$state$center) / self$state$scale)
    }
  )
)
```

We can compare this `PipeOp` to the one above to show that it behaves the same.

```
gr = Graph$new()$add_pipeop(PipeOpScaleAlways$new())
result_posa = gr$train(task)[[1]]

gr = Graph$new()$add_pipeop(PipeOpScaleAlwaysSimple$new())
result_posa_simple = gr$train(task)[[1]]
```

```
result_posa$data()
##        Species Petal.Length Petal.Width Sepal.Length
##   1:    setosa      -1.3358     -1.3111     -0.89767
##   2:    setosa      -1.3358     -1.3111     -1.13920
##   3:    setosa      -1.3924     -1.3111     -1.38073
##   4:    setosa      -1.2791     -1.3111     -1.50149
##   5:    setosa      -1.3358     -1.3111     -1.01844
##  ---
## 146: virginica       0.8169      1.4440      1.03454
## 147: virginica       0.7036      0.9192      0.55149
## 148: virginica       0.8169      1.0504      0.79301
## 149: virginica       0.9302      1.4440      0.43072
## 150: virginica       0.7602      0.7880      0.06843
##      Sepal.Width
##   1:     1.01560
```

```
##   2:    -0.13154
##   3:     0.32732
##   4:     0.09789
##   5:     1.24503
##  ---
## 146:    -0.13154
## 147:    -1.27868
## 148:    -0.13154
## 149:     0.78617
## 150:    -0.13154
```

```
result_posa_simple$data()
##        Species Petal.Length Petal.Width Sepal.Length
##   1:    setosa      -1.3358     -1.3111     -0.89767
##   2:    setosa      -1.3358     -1.3111     -1.13920
##   3:    setosa      -1.3924     -1.3111     -1.38073
##   4:    setosa      -1.2791     -1.3111     -1.50149
##   5:    setosa      -1.3358     -1.3111     -1.01844
##  ---
## 146: virginica       0.8169      1.4440      1.03454
## 147: virginica       0.7036      0.9192      0.55149
## 148: virginica       0.8169      1.0504      0.79301
## 149: virginica       0.9302      1.4440      0.43072
## 150: virginica       0.7602      0.7880      0.06843
##      Sepal.Width
##   1:     1.01560
##   2:    -0.13154
##   3:     0.32732
##   4:     0.09789
##   5:     1.24503
##  ---
## 146:    -0.13154
## 147:    -1.27868
## 148:    -0.13154
## 149:     0.78617
## 150:    -0.13154
```

6.2.4 Hyperparameters

mlr3pipelines uses the paradox package to define parameter spaces for `PipeOps`. Parameters for `PipeOps` can modify their behavior in certain ways, e.g. switch centering or scaling off in the `PipeOpScale` operator. The unified interface makes it possible to have parameters for whole `Graphs` that modify the individual `PipeOp`'s behavior. The `Graphs`, when encapsulated in `GraphLearners`, can even be tuned using the tuning functionality in mlr3tuning.

Hyperparameters are declared during initialization, when calling the `PipeOp`'s `$initialize()` function, by giving a `param_set` argument. The `param_set` must be a `ParamSet` from the paradox package; see the mlr3book for more information on how to define parameter spaces. After construction, the `ParamSet` can be accessed through the `$param_set`

slot. While it is *possible* to modify this `ParamSet`, using e.g. the `$add()` and `$add_dep()` functions, *after* adding it to the `PipeOp`, it is strongly advised against.

Hyperparameters can be set and queried through the `$values` slot. When setting hyperparameters, they are automatically checked to satisfy all conditions set by the `$param_set`, so it is not necessary to type check them. Be aware that it is always possible to *remove* hyperparameter values.

When a `PipeOp` is initialized, it usually does not have any parameter values—`$values` takes the value `list()`. It is possible to set initial parameter values in the `$initialize()` constructor; this must be done *after* the `super$initialize()` call where the corresponding `ParamSet` must be supplied. This is because setting `$values` checks against the current `$param_set`, which would fail if the `$param_set` was not set yet.

When using an underlying library function (the `scale` function in `PipeOpScale`, say), then there is usually a "default" behavior of that function when a parameter is not given. It is good practice to use this default behavior whenever a parameter is not set (or when it was removed). This can easily be done when using the mlr3misc library's `invoke()` function, which has functionality similar to `base::do.call()`.

6.2.4.1 Hyperparameter Example: `PipeOpScale`

How to use hyperparameters can best be shown through the example of `PipeOpScale`, which is very similar to the example above, `PipeOpScaleAlways`. The difference is made by the presence of hyperparameters. `PipeOpScale` constructs a `ParamSet` in its `$initialize` function and passes this on to the `super$initialize` function:

```
PipeOpScale$public_methods$initialize
## function (id = "scale", param_vals = list())
## {
##     ps = ParamSet$new(params = list(ParamLgl$new("center", default = TRUE,
##         tags = c("train", "scale")), ParamLgl$new("scale", default = TRUE,
##         tags = c("train", "scale"))))
##     super$initialize(id = id, param_set = ps, param_vals = param_vals)
## }
## <bytecode: 0x1968a1d0>
## <environment: namespace:mlr3pipelines>
```

The user has access to this and can set and get parameters. Types are automatically checked:

```
pss = PipeOpScale$new()
print(pss$param_set)
## ParamSet: scale
##                id    class lower upper      levels
## 1:         center ParamLgl    NA    NA  TRUE,FALSE
## 2:          scale ParamLgl    NA    NA  TRUE,FALSE
## 3: affect_columns ParamUty    NA    NA
##        default value
## 1:        TRUE
## 2:        TRUE
## 3: <NoDefault>
```

```
pss$param_set$values$center = FALSE
print(pss$param_set$values)
## $center
## [1] FALSE
```

```
pss$param_set$values$scale = "TRUE"  # bad input is checked!
## Error in (function (xs) : Assertion on 'xs' failed: scale: Must be of type 'logical flag', not 'character'.
```

How `PipeOpScale` handles its parameters can be seen in its `$train` method: It gets the relevant parameters from its `$values` slot and uses them in the `mlr3misc::invoke` call. This has the advantage over calling `scale()` directly that if a parameter is not given, its default value from the `base::scale` function will be used.

```
PipeOpScale$public_methods$train
## function (dt, levels, target)
## {
##     sc = invoke(scale, as.matrix(dt), .args = self$param_set$get_values(tags = "scale"))
##     self$state = list(center = attr(sc, "scaled:center") %??%
##         0, scale = attr(sc, "scaled:scale") %??% 1)
##     constfeat = self$state$scale == 0
##     self$state$scale[constfeat] = 1
##     sc[, constfeat] = 0
##     sc
## }
## <bytecode: 0x19687d38>
## <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
##   1:    setosa          1.4         0.2          5.1
##   2:    setosa          1.4         0.2          4.9
```

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

7 Special Tasks

This chapter explores the different functions of mlr3 when dealing with specific data sets that require further statistical modification to undertake sensible analysis. Following topics are discussed:

Survival Analysis

This sub-chapter explains how to conduct sound survival analysis in mlr3. Survival analysis is used to monitor the period of time until a specific event takes places. This specific event could be e.g. death, transmission of a disease, marriage or divorce. Two considerations are important when conducting survival analysis:

- Whether the event occurred within the frame of the given data
- How much time it took until the event occurred

In summary, this sub-chapter explains how to account for these considerations and conduct survival analysis using the mlr3proba extension package.

Spatial Analysis

Spatial analysis data observations entail reference information about spatial characteristics. One of the largest shortcomings of spatial data analysis is the inevitable auto-correlation in spatial data. Auto-correlation is especially severe in data with marginal spatial variation. The sub-chapter on spatial analysis provides instructions on how to handle the problems associated with spatial data accordingly.

Ordinal Analysis

This is work in progress. See `mlr-org/mlr3ordinal` for the current state.

Functional Analysis

Functional analysis contains data that consists of curves varying over a continuum e.g. time, frequency or wavelength. This type of analysis is frequently used when examining measurements over various time points. Steps on how to accommodate functional data structures in mlr3 are explained in the functional analysis sub-chapter.

Multilabel Classification

Multilabel classification deals with objects that can belong to more than one category at the same time. Numerous target labels are attributed to a single observation. Working with multilabel data requires one to use modified algorithms, to accommodate data specific characteristics. Two approaches to multilabel classification are prominently used:

- The problem transformation method
- The algorithm adaption method

Instructions on how to deal with multilabel classification in mlr3 can be found in this sub-chapter.

Cost Sensitive Classification

This sub-chapter deals with the implementation of cost-sensitive classification. Regular classification aims to minimize the misclassification rate and thus all types of misclassification errors are deemed equally severe. Cost-sensitive classification is a setting where the costs caused by different kinds of errors are not assumed to be equal. The objective is to minimize the expected costs.

Analytical data for a big credit institution is used as a use case to illustrate the different features. Firstly, the sub-chapter provides guidance on how to implement a first model. Subsequently, the sub-chapter contains instructions on how to modify cost sensitivity measures, thresholding and threshold tuning.

7.1 Survival Analysis

Survival analysis examines data on whether a specific event of interest takes place and how long it takes till this event occurs. One cannot use ordinary regression analysis when dealing with survival analysis data sets. Firstly, survival data contains solely positive values and therefore needs to be transformed to avoid biases. Secondly, ordinary regression analysis cannot deal with censored observations accordingly. Censored observations are observations in which the event of interest has not occurred, yet. Survival analysis allows the user to handle censored data with limited time frames that sometimes do not entail the event of interest. Note that survival analysis accounts for both censored and uncensored observations while adjusting respective model parameters.

The package mlr3proba extends mlr3 with the following objects for survival analysis:

- `mlr3proba::TaskSurv", text = "TaskSurv` to define (right-censored) survival tasks
- `mlr3proba::LearnerSurv", text = "LearnerSurv` as base class for survival learners
- `mlr3proba::PredictionSurv", text = "PredictionSurv` as specialized class for `Prediction` objects
- `mlr3proba::MeasureSurv", text = "MeasureSurv` as specialized class for performance measures

In this example we demonstrate the basic functionality of the package on the `survival::rats", text = "rats` data from the `survival` package. This task ships as pre-defined `TaskSurv` with mlr3proba.

```
library(mlr3proba)
task = tsk("rats")
print(task)
## <TaskSurv:rats> (300 x 5)
## * Target: time, status
## * Properties: -
## * Features (3):
##   - int (2): litter, rx
##   - fct (1): sex

# the target column is a survival object:
head(task$truth())
## [1] 101+  49  104+  91+ 104+ 102+
```

Now, we conduct a small benchmark study on the `mlr_tasks_rats", text = "rats` task using some of the integrated survival learners:

```
# integrated learners
learners = lapply(c("surv.coxph", "surv.kaplan", "surv.ranger"),
  lrn)
print(learners)
## [[1]]
## <LearnerSurvCoxPH:surv.coxph>
## * Model: -
```

```
## * Parameters: list()
## * Packages: survival, distr6
## * Predict Type: distr
## * Feature types: logical, integer, numeric,
##   factor
## * Properties: importance
##
## [[2]]
## <LearnerSurvKaplan:surv.kaplan>
## * Model: -
## * Parameters: list()
## * Packages: survival, distr6
## * Predict Type: crank
## * Feature types: logical, integer, numeric,
##   character, factor, ordered
## * Properties: missings
##
## [[3]]
## <LearnerSurvRanger:surv.ranger>
## * Model: -
## * Parameters: list()
## * Packages: ranger, distr6
## * Predict Type: distr
## * Feature types: logical, integer, numeric,
##   character, factor, ordered
## * Properties: importance, oob_error, weights

measure = msr("surv.harrellC")
print(measure)
## <MeasureSurvHarrellC:surv.harrellC>
## * Packages: -
## * Range: [0, 1]
## * Minimize: FALSE
## * Properties: -
## * Predict type: crank

set.seed(1)
bmr = benchmark(benchmark_grid(task, learners, rsmp("cv",
  folds = 3)))
print(bmr)
## <BenchmarkResult> of 9 rows with 3 resampling runs
##  nr task_id  learner_id resampling_id iters warnings
##   1    rats  surv.coxph            cv     3        0
##   2    rats surv.kaplan            cv     3        0
##   3    rats surv.ranger            cv     3        0
##  errors
##       0
##       0
##       0
```

7.2 Spatial Analysis

Spatial data observations entail reference information about spatial characteristics. This information is frequently stored as coordinates named 'x' and 'y'. Treating spatial data using non-spatial data methods could lead to over-optimistic treatment. This is due to the underlying auto-correlation in spatial data.

See `mlr-org/mlr3spatiotemporal` for the current state of the implementation.

7.3 Ordinal Analysis

This is work in progress. See `mlr-org/mlr3ordinal` for the current state of the implementation.

7.4 Functional Analysis

Functional data is data containing an ordering on the dimensions. This implies that functional data consists of curves varying over a continuum, such as time, frequency, or wavelength.

7.4.1 How to model functional data?

There are two ways to model functional data:

- Modification of the learner, so that the learner is suitable for the functional data
- Modification of the task, so that the task matches the standard- or classification-learner

More following soon!

7.5 Multilabel Classification

Multilabel deals with objects that can belong to more than one category at the same time.

More following soon!

7.6 Cost-Sensitive Classification

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

Imagine you are an analyst for a big credit institution. Let's also assume that a correct decision of the bank would result in 35% of the profit at the end of a specific period. A correct decision means that the bank predicts that a customer will pay their bills (hence would obtain a loan), and the customer indeed has good credit. On the other hand, a wrong decision means that the bank predicts that the customer's credit is in good standing, but the opposite is true. This would result in a loss of 100% of the given loan.

	Good Customer (truth)	Bad Customer (truth)
Good Customer (predicted)	+ 0.35	- 1.0

	Good Customer (truth)	Bad Customer (truth)
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 `mlr_tasks_german_credit", text = "German Credit` task:

```
library(mlr3)
task = tsk("german_credit")
table(task$truth())
##
## good  bad
##  700  300
```

The data has 70% customers who are able to pay back their credit, and 30% bad customers who default on the debt. A manager, who doesn't have any model, could decide to give either everybody a credit or to give nobody a credit. The resulting costs for the German credit data are:

```
# nobody:
(700 * costs[2, 1] + 300 * costs[2, 2])/1000
## [1] 0

# everybody
(700 * costs[1, 1] + 300 * costs[1, 2])/1000
## [1] 0.055
```

If the average loan is $20,000, the credit institute would lose more than one million dollar if it would grant everybody a credit:

```
# average profit * average loan * number of customers
0.055 * 20000 * 1000
## [1] 1100000
```

Our goal is to find a model which minimizes the costs (and thereby maximizes the expected profit).

7.6.1 A First Model

For our first model, we choose an ordinary logistic regression (implemented in the add-on package mlr3learners). We first create a classification task, then resample the model using a 10-fold cross validation and extract the resulting confusion matrix:

```
library(mlr3learners)
learner = lrn("classif.log_reg")
rr = resample(task, learner, rsmp("cv"))

confusion = rr$prediction()$confusion
print(confusion)
##         truth
## response good bad
##     good  602 154
##     bad    98 146
```

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] -0.0567
```

With an average loan of $20,000, the logistic regression yields the following costs:

```
avg_costs * 20000 * 1000
## [1] -1134000
```

Instead of losing over $1,000,000, the credit institute now can expect a profit of more than $1,000,000.

7.6.2 Cost-sensitive Measure

Our natural next step would be to further improve the modeling step in order to maximize the profit. For this purpose we first create a cost-sensitive classification measure which calculates the costs based on our cost matrix. This allows us to conveniently quantify and compare modeling decisions. Fortunately, there already is a predefined measure `Measure` for this purpose: `MeasureClassifCosts`:

```
cost_measure = msr("classif.costs", costs = costs)
print(cost_measure)
## <MeasureClassifCosts:classif.costs>
## * Packages: -
## * Range: [-Inf, Inf]
## * Minimize: TRUE
## * Properties: requires_task
## * Predict type: response
```

If we now call `resample()` or `benchmark()`, the cost-sensitive measures will be evaluated. We compare the logistic regression to a simple featureless learner and to a random forest from package `ranger` :

```
learners = list(lrn("classif.log_reg"), lrn("classif.featureless"),
  lrn("classif.ranger"))
cv3 = rsmp("cv", folds = 3)
bmr = benchmark(benchmark_grid(task, learners, cv3))
bmr$aggregate(cost_measure)
##    nr  resample_result       task_id
## 1:  1 <ResampleResult> german_credit
## 2:  2 <ResampleResult> german_credit
## 3:  3 <ResampleResult> german_credit
##             learner_id resampling_id iters
## 1:     classif.log_reg            cv     3
## 2: classif.featureless            cv     3
## 3:      classif.ranger            cv     3
##    classif.costs
## 1:      -0.05436
## 2:       0.05498
## 3:      -0.04000
```

As expected, the featureless learner is performing comparably bad. The logistic regression and the random forest work equally well.

7.6.3 Thresholding

Although we now correctly evaluate the models in a cost-sensitive fashion, the models themselves are unaware of the classification costs. They assume the same costs for both wrong classification decisions (false positives and false negatives). Some learners natively support cost-sensitive classification (e.g., XXX). However, we will concentrate on a more generic approach which works for all models which can predict probabilities for class labels: thresholding.

Most learners can calculate the probability p for the positive class. If p exceeds the threshold 0.5, they predict the positive class, and the negative class otherwise.

For our binary classification case of the credit data, the we primarily want to minimize the errors where the model predicts "good", but truth is "bad" (i.e., the number of false positives) as this is the more expensive error. If we now increase the threshold to values > 0.5, we reduce the number of false negatives. Note that we increase the number of false positives simultaneously, or, in other words, we are trading false positives for false negatives.

```
# fit models with probability prediction
learner = lrn("classif.log_reg", predict_type = "prob")
rr = resample(task, learner, rsmp("cv"))
p = rr$prediction()
print(p)
## <PredictionClassif> for 1000 observations:
##     row_id truth response prob.good prob.bad
##         14   bad     good    0.6210 0.379048
##         20  good     good    0.9046 0.095355
##         25  good     good    0.9939 0.006067
```

```
## ---
##        980   bad      bad    0.3350 0.665042
##        983  good     good    0.6665 0.333486
##        999   bad      bad    0.3552 0.644800

# 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  604 159
##     bad    96 141
##
## $costs
## classif.costs
##       -0.0524
with_threshold(p, 0.75)
## $confusion
##         truth
## response good bad
##     good  468  69
##     bad   232 231
##
## $costs
## classif.costs
##       -0.0948
with_threshold(p, 1)
## $confusion
##         truth
## response good bad
##     good    1   1
##     bad   699 299
##
## $costs
## classif.costs
##       0.00065

# 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] 0.7175
##
## $objective
## classif.costs
##      -0.09795

# optimized confusion matrix:
with_threshold(p, best$minimum)$confusion
##          truth
## response good bad
##     good  497  76
##     bad   203 224
```

The function `optimize()` is intended for unimodal functions and therefore may converge to a local optimum here. See below for better alternatives to find good threshold values.

7.6.4 Threshold Tuning

More following soon!

Upcoming sections will entail:

- threshold tuning as pipeline operator
- joint hyperparameter optimization

More following soon!

8 Model Interpretation

8.1 IML

8.2 Dalex

9 Use Cases

This chapter is a collection of use cases to showcase mlr3. The first use case shows different functions, using house price data, housing price data in King Country.

Following features are illustrated:

- Summarizing the data set
- Converting data to treat it as a numeric feature/factor
- Generating new variables
- Splitting data into train and test data sets
- Computing a first model (decision tree)
- Building many trees (random forest)
- Visualizing price data across different region
- Optimizing the baseline by implementing a tuner
- Engineering features
- Creating a sparser model

Further use cases are following soon!

9.1 House Price Prediction in King County

We use the `house_sales_prediction` dataset contained in this book in order to provide a use-case for the application of mlr3 on real-world data.

```
library(mlr3book)
data("house_sales_prediction", package = "mlr3book")
```

9.1.1 Exploratory Data Analysis

In order to get a quick impression of our data, we perform some initial *Exploratory Data Analysis*. This helps us to get a first impression of our data and might help us arrive at additional features that can help with the prediction of the house prices.

We can get a quick overview using R's summary function:

```
summary(house_sales_prediction)
##        id                 date
##  Min.   :   1000102   Length:21613
##  1st Qu.:2123049194   Class :character
##  Median :3904930410   Mode  :character
##  Mean   :4580301520
```

```
##  3rd Qu.:7308900445
##  Max.   :9900000190
##      price            bedrooms       bathrooms
##  Min.   :  75000   Min.   : 0.00   Min.   :0.00
##  1st Qu.: 321950   1st Qu.: 3.00   1st Qu.:1.75
##  Median : 450000   Median : 3.00   Median :2.25
##  Mean   : 540088   Mean   : 3.37   Mean   :2.12
##  3rd Qu.: 645000   3rd Qu.: 4.00   3rd Qu.:2.50
##  Max.   :7700000   Max.   :33.00   Max.   :8.00
##   sqft_living       sqft_lot           floors
##  Min.   :  290   Min.   :    520   Min.   :1.00
##  1st Qu.: 1427   1st Qu.:   5040   1st Qu.:1.00
##  Median : 1910   Median :   7618   Median :1.50
##  Mean   : 2080   Mean   :  15107   Mean   :1.49
##  3rd Qu.: 2550   3rd Qu.:  10688   3rd Qu.:2.00
##  Max.   :13540   Max.   :1651359   Max.   :3.50
##    waterfront          view         condition
##  Min.   :0.0000   Min.   :0.000   Min.   :1.00
##  1st Qu.:0.0000   1st Qu.:0.000   1st Qu.:3.00
##  Median :0.0000   Median :0.000   Median :3.00
##  Mean   :0.0075   Mean   :0.234   Mean   :3.41
##  3rd Qu.:0.0000   3rd Qu.:0.000   3rd Qu.:4.00
##  Max.   :1.0000   Max.   :4.000   Max.   :5.00
##      grade          sqft_above   sqft_basement
##  Min.   : 1.00   Min.   : 290   Min.   :   0
##  1st Qu.: 7.00   1st Qu.:1190   1st Qu.:   0
##  Median : 7.00   Median :1560   Median :   0
##  Mean   : 7.66   Mean   :1788   Mean   : 292
##  3rd Qu.: 8.00   3rd Qu.:2210   3rd Qu.: 560
##  Max.   :13.00   Max.   :9410   Max.   :4820
##     yr_built     yr_renovated       zipcode
##  Min.   :1900   Min.   :   0.0   Min.   :98001
##  1st Qu.:1951   1st Qu.:   0.0   1st Qu.:98033
##  Median :1975   Median :   0.0   Median :98065
##  Mean   :1971   Mean   :  84.4   Mean   :98078
##  3rd Qu.:1997   3rd Qu.:   0.0   3rd Qu.:98118
##  Max.   :2015   Max.   :2015.0   Max.   :98199
##       lat            long      sqft_living15
##  Min.   :47.2   Min.   :-123   Min.   : 399
##  1st Qu.:47.5   1st Qu.:-122   1st Qu.:1490
##  Median :47.6   Median :-122   Median :1840
##  Mean   :47.6   Mean   :-122   Mean   :1987
##  3rd Qu.:47.7   3rd Qu.:-122   3rd Qu.:2360
##  Max.   :47.8   Max.   :-121   Max.   :6210
##    sqft_lot15
##  Min.   :   651
##  1st Qu.:  5100
##  Median :  7620
##  Mean   : 12768
##  3rd Qu.: 10083
```

```
##  Max.   :871200
dim(house_sales_prediction)
## [1] 21613    21
```

Our dataset has 21613 observations and 21 columns. The variable we want to predict is `price`. In addition to the price column, we have several other columns:

- `id`: A unique identifier for every house.
- `date`: A date column, indicating when the house was sold. This column is currently not encoded as a `date` and requires some preprocessing.
- `zipcode`: A column indicating the ZIP code. This is a categorical variable with many factor levels.
- `long, lat` The longitude and latitude of the house
- `...` several other numeric columns providing information about the house, such as number of rooms, square feet etc.

Before we continue with the analysis, we preprocess some features so that they are stored in the correct format.

First we convert the `date` column to `numeric` to be able to treat it as a numeric feature:

```
library(lubridate)
house_sales_prediction$date = ymd(substr(house_sales_prediction$date,
  1, 8))
house_sales_prediction$date = as.numeric(as.Date(house_sales_prediction$date,
  origin = "1900-01-01"))
house_sales_prediction$date = house_sales_prediction$date
```

Afterwards, we convert the zip code to a factor:

```
house_sales_prediction$zipcode = as.factor(house_sales_prediction$zipcode)
```

And add a new column **renovated** indicating whether a house was renovated at some point.

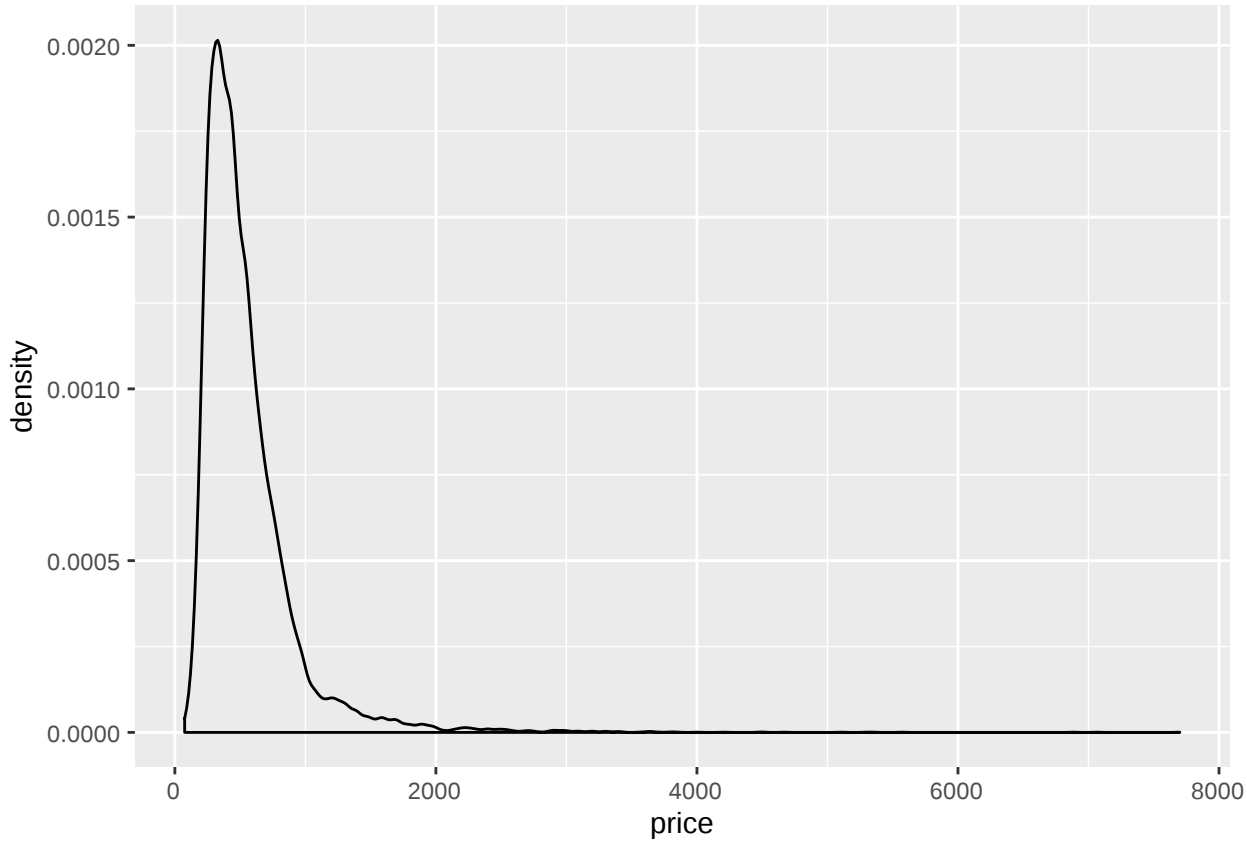
```
house_sales_prediction$renovated = as.numeric(house_sales_prediction$yr_renovated >
  0)
# And drop the id column:
house_sales_prediction$id = NULL
```

Additionally we convert the price from Dollar to units of 1000 Dollar to improve readability.

```
house_sales_prediction$price = house_sales_prediction$price/1000
```

We can now plot the density of the **price** to get a first impression on its distribution.

```
library(ggplot2)
ggplot(house_sales_prediction, aes(x = price)) + geom_density()
```



We can see that the prices for most houses lie between 75.000 and 1.5 million dollars. There are few extreme values of up to 7.7 million dollars.

Feature engineering often allows us to incorporate additional knowledge about the data and underlying processes. This can often greatly enhance predictive performance. A simple example: A house which has `yr_renovated == 0` means that is has not been renovated yet. Additionally we want to drop features which should not have any influence (`id column`).

After those initial manipulations, we load all required packages and create a Task containing our data.

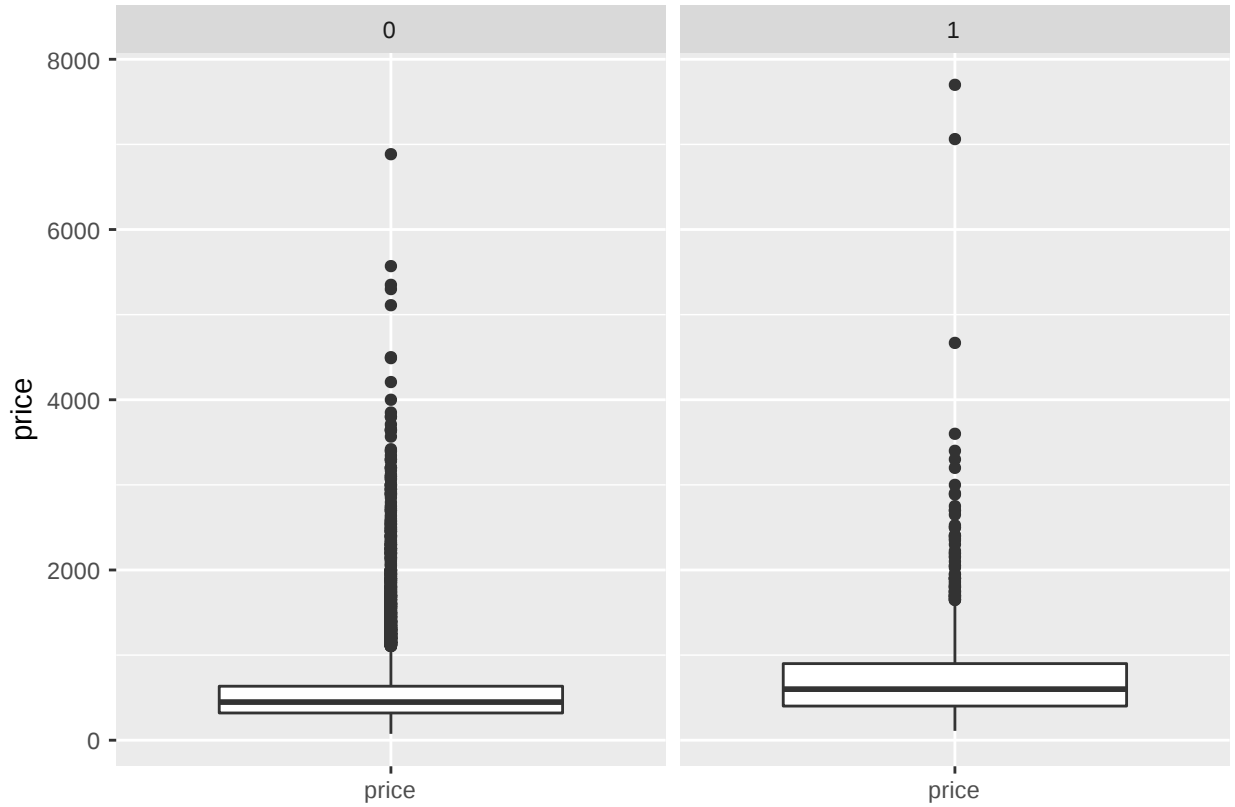
```
library(mlr3)
library(mlr3viz)
tsk = TaskRegr$new("sales", house_sales_prediction, target = "price")
```

We can inspect associations between variables using mlr3viz's `autoplot` function in order to get some good first impressions for our data. Note, that this does in no way prevent us from using other powerful plot functions of our choice on the original data.

9.1.1.1 Distribution of the price:

The outcome we want to predict is the **price** variable. The `autoplot` function provides a good first glimpse on our data. As the resulting object is a `ggplot2` object, we can use `faceting` and other functions from **ggplot2** in order to enhance plots.

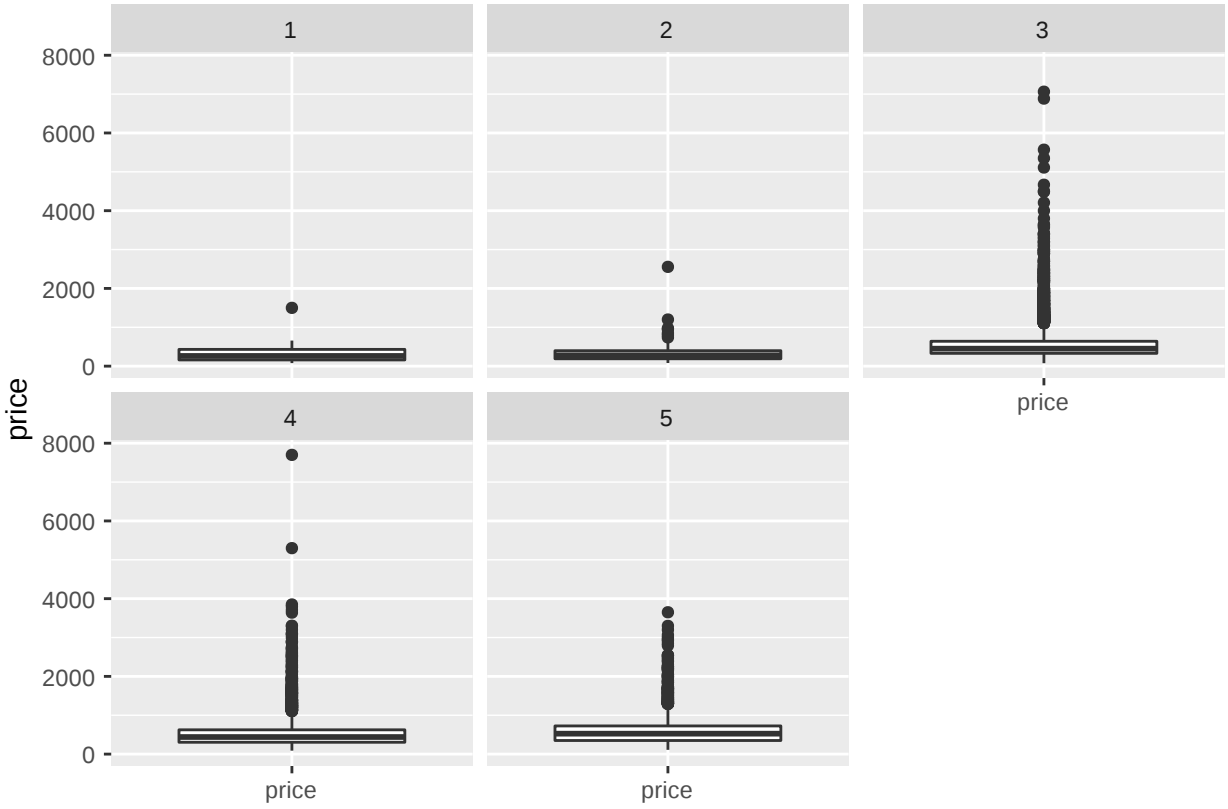
```
library(ggplot2)
autoplot(tsk) + facet_wrap(~renovated)
```



We can observe that renovated flats seem to achieve higher sales values, and this might thus be a relevant feature.

Additionally, we can for example look at the condition of the house. Again, we clearly can see that the price rises with increasing condition.

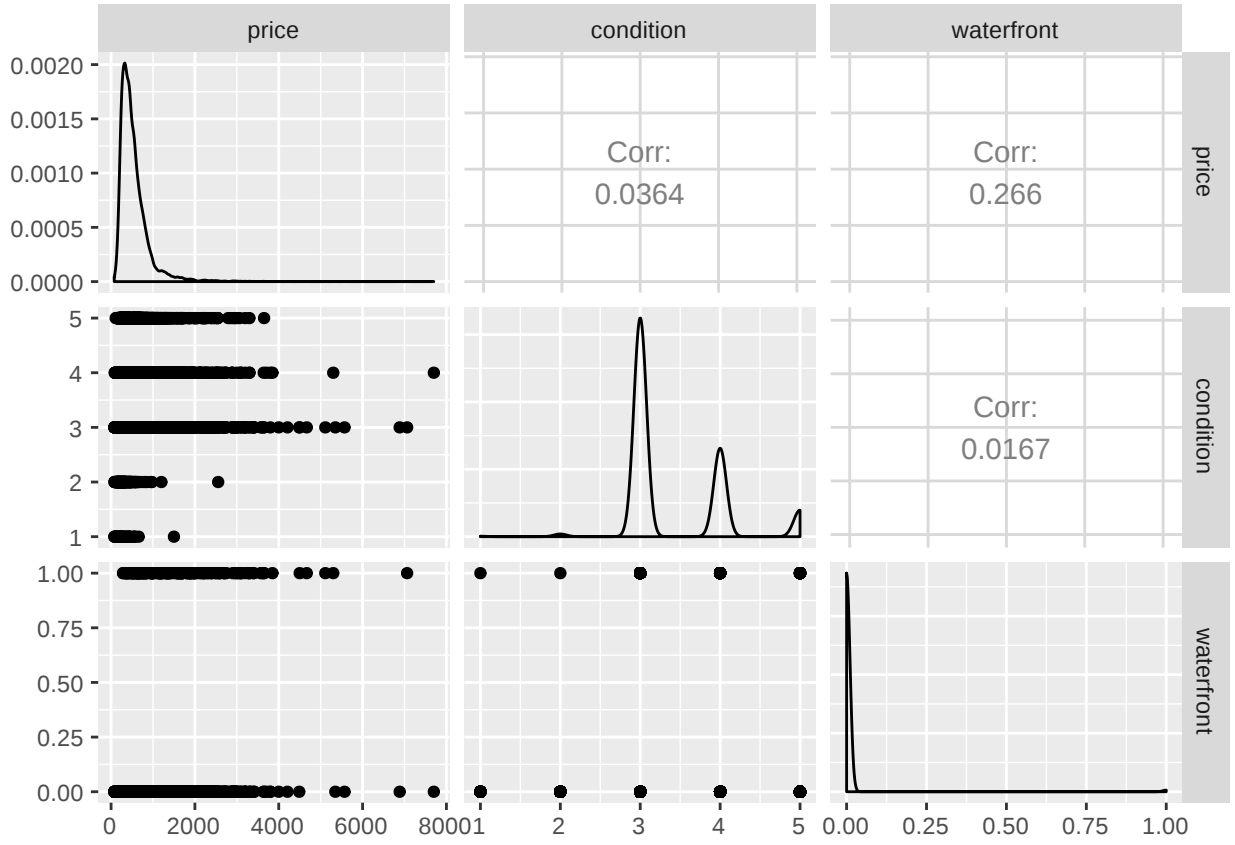
```
autoplot(tsk) + facet_wrap(~condition)
```



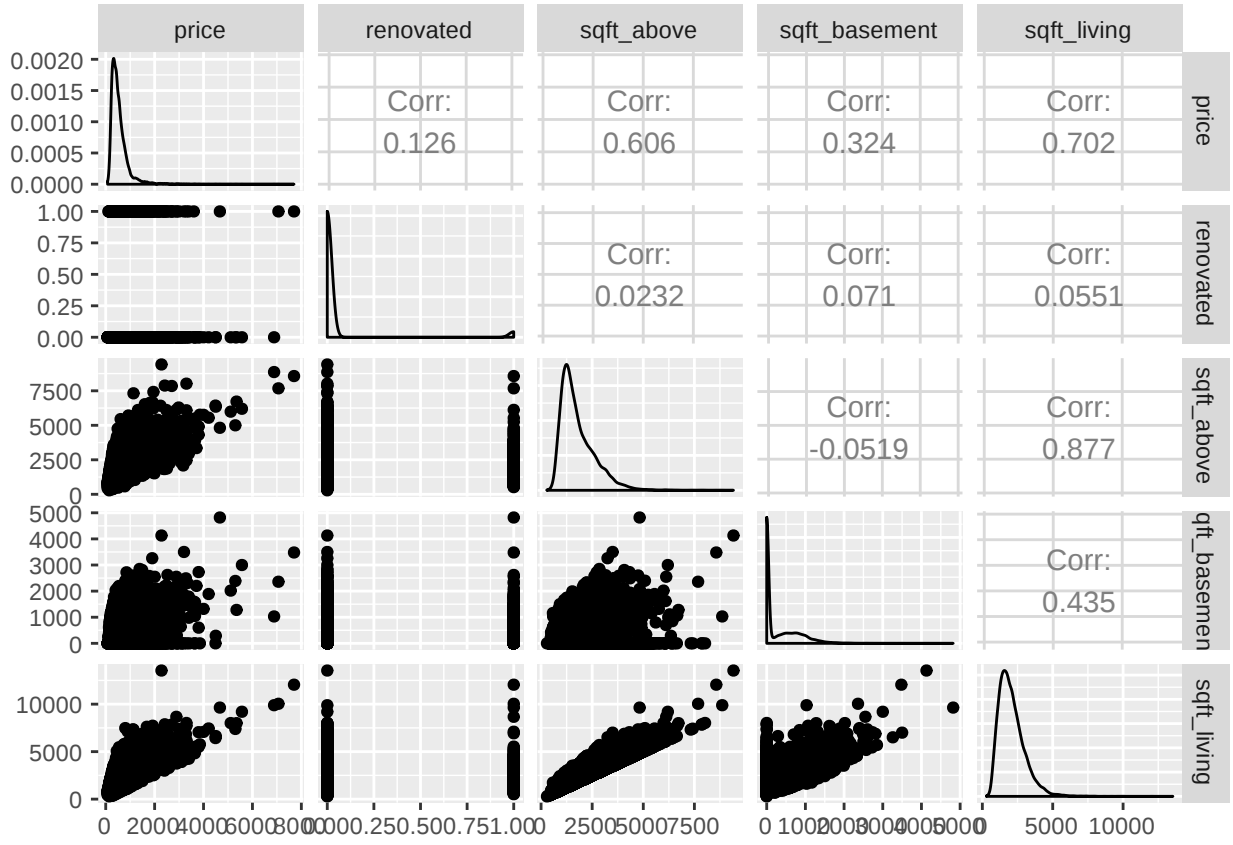
9.1.1.2 Association between variables

In addition to the association with the target variable, the association between the features can also lead to interesting insights. We investigate using variables associated with the quality and size of the house. Note that we use `$clone()` and `$select()` to clone the task and select only a subset of the features for the `autoplot` function, as `autoplot` per default uses all features. The task is cloned before we select features in order to keep the original task intact.

```
# Variables associated with quality
autoplot(tsk$clone()$select(tsk$feature_names[c(3, 17)]),
  type = "pairs")
```




```
autoplot(tsk$clone()$select(tsk$feature_names[c(9:12)]),
  type = "pairs")
```



9.1.2 Splitting into train and test data

In mlr3, we do not create `train` and `test` data sets, but instead keep only a vector of train and test indices.

```
set.seed(4411)
train.idx = sample(seq_len(tsk$nrow), 0.7 * tsk$nrow)
test.idx = setdiff(seq_len(tsk$nrow), train.idx)
```

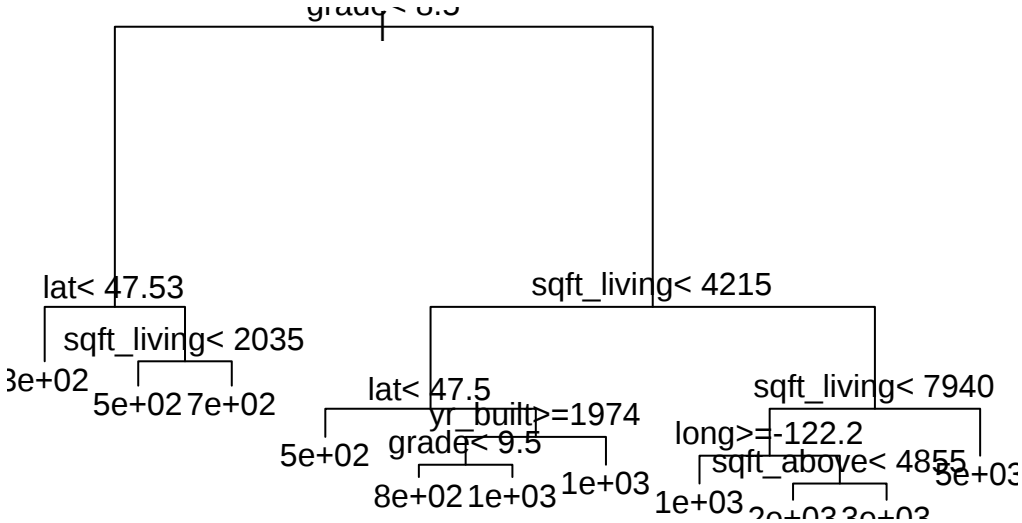
9.1.3 A first model: Decision Tree

Decision trees cannot only be used as a powerful tool for predictive models but also for exploratory data analysis. In order to fit a decision tree, we first get the `regr.rpart` learner from the `mlr_learners` dictionary by using the sugar function `lrn`.

For now we leave out the `zipcode` variable, as we also have the `latitude` and `longitude` of each house.

```
tsk_nozip = tsk$clone()$select(setdiff(tsk$feature_names,
  "zipcode"))
# Get the learner
lrn = lrn("regr.rpart")
# And train on the task
lrn$train(tsk_nozip, row_ids = train.idx)
```

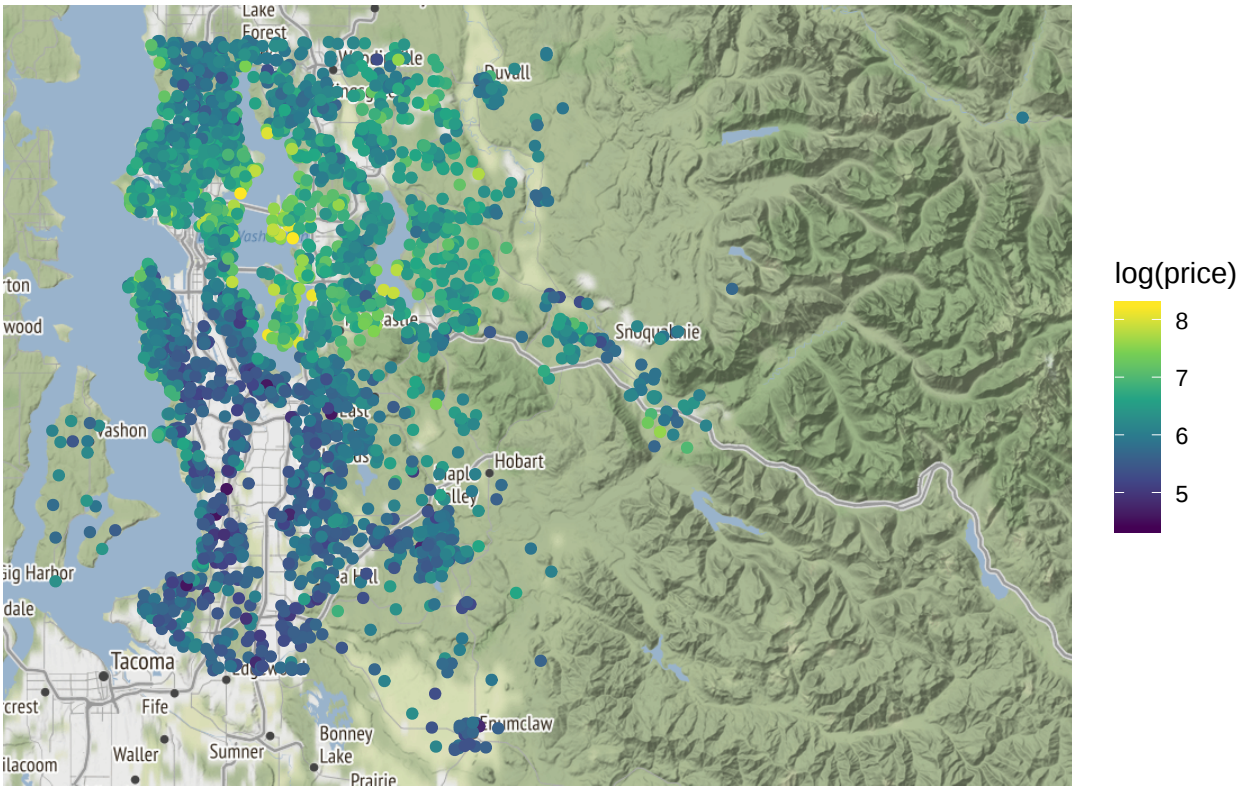
```
plot(lrn$model)
text(lrn$model)
```



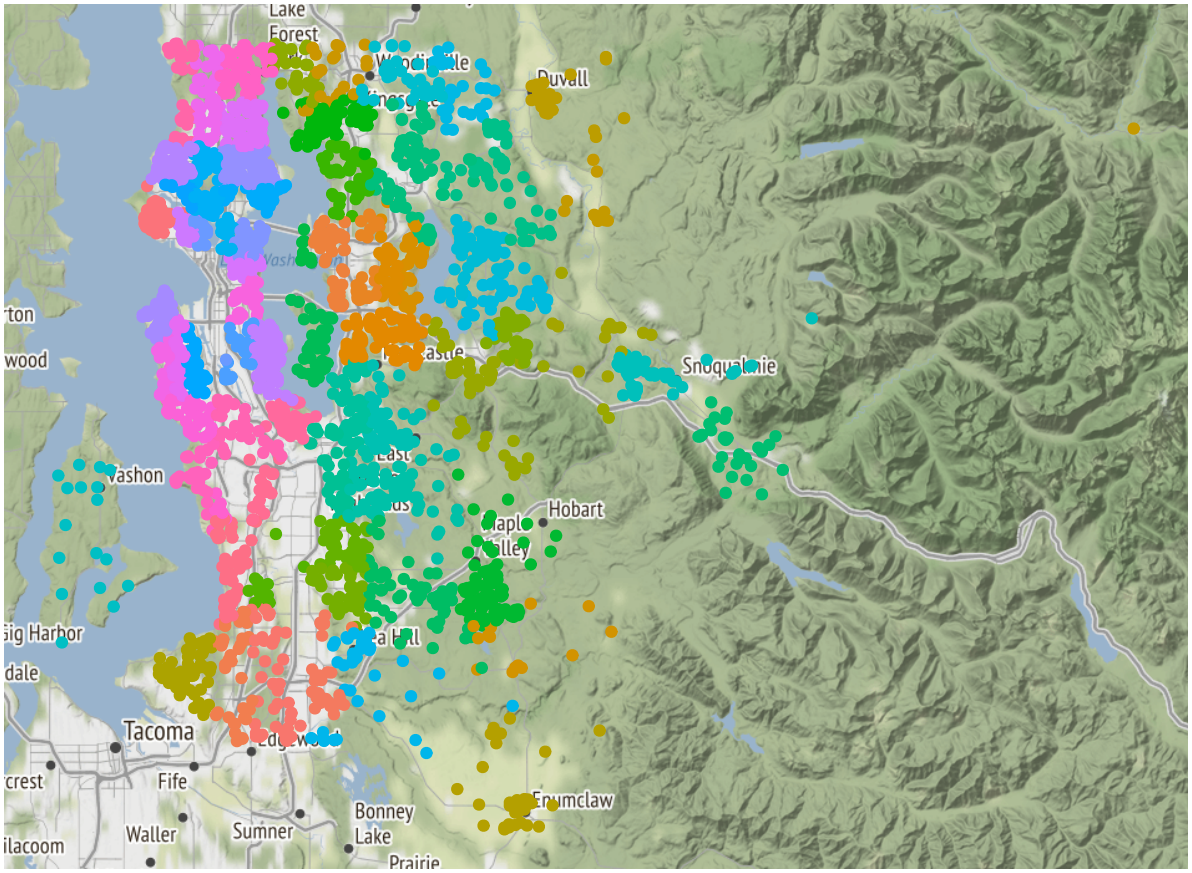
The learned tree relies on several variables in order to distinguish between cheaper and pricier houses. The features we split along are **grade**, **sqft_living**, but also some features related to the area (longitude and latitude).

We can visualize the price across different regions in order to get more info:

```
# Load the ggmap package in order to visualize on a map
library(ggmap)

# And create a quick plot for the price
qmplot(long, lat, maptype = "watercolor", color = log(price),
  data = house_sales_prediction[train.idx[1:3000], ]) +
  scale_colour_viridis_c()
```




```
# And the zipcode
qmplot(long, lat, maptype = "watercolor", color = zipcode,
  data = house_sales_prediction[train.idx[1:3000], ]) +
  guides(color = FALSE)
```



We can see that the price is clearly associated with the zipcode when comparing then two plots. As a result, we might want to indeed use the **zipcode** column in our future endeavours.

9.1.4 A first baseline: Decision Tree

After getting an initial idea for our data, we might want to construct a first baseline, in order to see what a simple model already can achieve.

We use `resample` with `3-fold cross-validation` on our training data in order to get a reliable estimate of the algorithm's performance on future data. Before we start with defining and training learners, we create a `Resampling` in order to make sure that we always compare on exactly the same data.

```
library(mlr3learners)
cv3 = rsmp("cv", folds = 3)
cv3$instantiate(tsk$clone()$filter(train.idx))
```

For the cross-validation we only use the **training data** by cloning the task and selecting only observations from the training set.

```
lrn_rpart = lrn("regr.rpart")
res = resample(task = tsk$clone()$filter(train.idx), lrn_rpart,
  cv3)
res$score(msr("regr.mse"))
```

```
##          task task_id            learner learner_id
## 1: <TaskRegr>   sales <LearnerRegrRpart> regr.rpart
## 2: <TaskRegr>   sales <LearnerRegrRpart> regr.rpart
## 3: <TaskRegr>   sales <LearnerRegrRpart> regr.rpart
##        resampling resampling_id iteration prediction
## 1: <ResamplingCV>            cv         1     <list>
## 2: <ResamplingCV>            cv         2     <list>
## 3: <ResamplingCV>            cv         3     <list>
##    regr.mse
## 1:    45416
## 2:    49599
## 3:    42518
sprintf("RMSE of the simple rpart: %s", round(sqrt(res$aggregate()),
  2))
## [1] "RMSE of the simple rpart: 214.11"
```

9.1.5 Many Trees: Random Forest

We might be able to improve upon the **RMSE** using more powerful learners. We first load the mlr3learners package, which contains the **ranger** learner (a package which implements the "Random Forest" algorithm).

```
lrn_ranger = lrn("regr.ranger", num.trees = 15L)
res = resample(task = tsk$clone()$filter(train.idx), lrn_ranger,
  cv3)
res$score(msr("regr.mse"))
##          task task_id             learner  learner_id
## 1: <TaskRegr>   sales <LearnerRegrRanger> regr.ranger
## 2: <TaskRegr>   sales <LearnerRegrRanger> regr.ranger
## 3: <TaskRegr>   sales <LearnerRegrRanger> regr.ranger
##        resampling resampling_id iteration prediction
## 1: <ResamplingCV>            cv         1     <list>
## 2: <ResamplingCV>            cv         2     <list>
## 3: <ResamplingCV>            cv         3     <list>
##    regr.mse
## 1:    23892
## 2:    22455
## 3:    18899
sprintf("RMSE of the simple ranger: %s", round(sqrt(res$aggregate()),
  2))
## [1] "RMSE of the simple ranger: 147.48"
```

Often tuning **RandomForest** methods does not increase predictive performances substantially. If time permits, it can nonetheless lead to improvements and should thus be performed. In this case, we resort to tune a different kind of model: **Gradient Boosted Decision Trees** from the package `xgboost`.

9.1.6 A better baseline: `AutoTuner`

Tuning can often further improve the performance. In this case, we *tune* the xgboost learner in order to see whether this can improve performance. For the `AutoTuner` we have to specify a **Termination Criterion** (how long the tuning should run) a **Tuner** (which tuning method to use) and a **ParamSet** (which space we might want to search through). For now we do not use the **zipcode** column, as `xgboost` cannot naturally deal with categorical features. The **AutoTuner** automatically performs nested cross-validation.

```
set.seed(444L)
library(mlr3tuning)
library(paradox)
lrn_xgb = lrn("regr.xgboost")

# Define the ParamSet
ps = ParamSet$new(params = list(ParamDbl$new(id = "eta",
  lower = 0.2, upper = 0.4), ParamDbl$new(id = "min_child_weight",
  lower = 1, upper = 20), ParamDbl$new(id = "subsample",
  lower = 0.7, upper = 0.8), ParamDbl$new(id = "colsample_bytree",
  lower = 0.9, upper = 1), ParamDbl$new(id = "colsample_bylevel",
  lower = 0.5, upper = 0.7), ParamInt$new(id = "nrounds",
  lower = 1L, upper = 25)))

# Define the Terminator
terminator = TerminatorEvaluations$new(10)
cv3 = rsmp("cv", folds = 3)
at = AutoTuner$new(lrn_xgb, cv3, measures = msr("regr.mse"),
  ps, terminator, tuner = TunerRandomSearch, tuner_settings = list())
```

```
res$score(msr("regr.mse"))
sprintf("RMSE of the tuned xgboost: %s", round(sqrt(res$aggregate()),
  2))
```

We can obtain the resulting params in the respective splits by accessing the `ResampleResult`.

```
sapply(res$learners, function(x) x$param_set$values)
## $num.trees
## [1] 15
##
## $num.trees
## [1] 15
##
## $num.trees
## [1] 15
```

NOTE: To keep runtime low, we only tune parts of the hyperparameter space of `xgboost` in this example. Additionally, we only allow for 10 random search iterations, which is usually to little for real-world applications. Nonetheless, we are able to obtain an improved performance when comparing to the `ranger` model.

In order to further improve our results we have several options:

- Find or engineer better features
- Remove Features to avoid overfitting
- Obtain additional data (often prohibitive)
- Try more models
- Improve the tuning
  - Increase the tuning budget
  - Enlarge the tuning search space
  - Use a more efficient tuning algorithm
- Stacking and Ensembles (see Pipelines)

Below we will investigate some of those possibilities and investigate whether this improves performance.

9.1.7 Engineering Features: Mutating ZIP-Codes

In order to better cluster the zip codes, we compute a new feature: **med_price**: It computes the median price in each zip-code. This might help our model to improve the prediction.

```
# Create a new feature and append it to the task
zip_price = house_sales_prediction[, .(med_price = median(price)),
  by = zipcode]

# Join on the original data to match with original
# columns, then cbind to the task
tsk$cbind(house_sales_prediction[zip_price, on = "zipcode"][,
  "med_price"])
```

Again, we run `resample` and compute the **RMSE**.

```
lrn_ranger = lrn("regr.ranger")
res = resample(task = tsk$clone()$filter(train.idx), lrn_ranger,
  cv3)
res$score(msr("regr.mse"))
##          task task_id             learner  learner_id
## 1: <TaskRegr>   sales <LearnerRegrRanger> regr.ranger
## 2: <TaskRegr>   sales <LearnerRegrRanger> regr.ranger
## 3: <TaskRegr>   sales <LearnerRegrRanger> regr.ranger
##        resampling resampling_id iteration prediction
## 1: <ResamplingCV>            cv         1     <list>
## 2: <ResamplingCV>            cv         2     <list>
## 3: <ResamplingCV>            cv         3     <list>
##    regr.mse
## 1:    22719
## 2:    20384
## 3:    17799
sprintf("RMSE of ranger with med_price: %s", round(sqrt(res$aggregate()),
```

```
  2))
## [1] "RMSE of ranger with med_price: 142.48"
```

9.1.8 Obtaining a sparser model

In many cases, we might want to have a sparse model. For this purpose we can use a `mlr3filters::Filter` implemented in mlr3filters. This can prevent our learner from overfitting make it easier for humans to interpret models as fewer variables influence the resulting prediction.

```
library(mlr3filters)
filter = FilterMRMR$new()$calculate(tsk)
tsk_ftsel = tsk$clone()$select(head(names(filter$scores),
  12))
```

The resulting **RMSE** is slightly higher, and at the same time we only use 12 features.

```
lrn_ranger = lrn("regr.ranger")
res = resample(task = tsk_ftsel$clone()$filter(train.idx),
  lrn_ranger, cv3)
res$score(msr("regr.mse"))
##          task task_id             learner  learner_id
## 1: <TaskRegr>   sales <LearnerRegrRanger> regr.ranger
## 2: <TaskRegr>   sales <LearnerRegrRanger> regr.ranger
## 3: <TaskRegr>   sales <LearnerRegrRanger> regr.ranger
##        resampling resampling_id iteration prediction
## 1: <ResamplingCV>            cv         1     <list>
## 2: <ResamplingCV>            cv         2     <list>
## 3: <ResamplingCV>            cv         3     <list>
##    regr.mse
## 1:    34720
## 2:    29056
## 3:    25907
sprintf("RMSE of ranger with filtering: %s", round(sqrt(res$aggregate()),
  2))
## [1] "RMSE of ranger with filtering: 172.9"
```

10 Appendix

10.1 Integrated Learners

Id	Feature Types	Required packages	Properties	Predict Types
classif.debug	lgl, int, dbl, chr, fct, ord		Missings, Multiclass, Twoclass	response, prob
classif.featureless	lgl, int, dbl, chr, fct, ord		Importance, Missings, Multiclass, Selected Features, Twoclass	response, prob
classif.glmnet	int, dbl	glmnet	Multiclass, Twoclass, Weights	response, prob
classif.kknn	lgl, int, dbl, fct, ord	withr, kknn	Multiclass, Twoclass	response, prob
classif.lda	lgl, int, dbl, fct, ord	MASS	Multiclass, Twoclass, Weights	response, prob
classif.log_reg	lgl, int, dbl, chr, fct, ord	stats	Twoclass, Weights	response, prob
classif.naive_bayes	lgl, int, dbl, fct	e1071	Multiclass, Twoclass	response, prob
classif.qda	lgl, int, dbl, fct, ord	MASS	Multiclass, Twoclass, Weights	response, prob
classif.ranger	lgl, int, dbl, chr, fct, ord	ranger	Importance, Multiclass, Oob Error, Twoclass, Weights	response, prob
classif.rpart	lgl, int, dbl, fct, ord	rpart	Importance, Missings, Multiclass, Selected Features, Twoclass, Weights	response, prob
classif.svm	int, dbl	e1071	Multiclass, Twoclass	response, prob
classif.xgboost	int, dbl	xgboost	Importance, Missings, Multiclass, Twoclass, Weights	response, prob
regr.featureless	lgl, int, dbl, chr, fct, ord	stats	Importance, Missings, Selected Features	response, se
regr.glmnet	int, dbl	glmnet	Weights	response
regr.kknn	lgl, int, dbl, fct, ord	withr, kknn		response
regr.km	int, dbl	DiceKriging		response, se
regr.lm	int, dbl, fct	stats	Weights	response, se
regr.ranger	lgl, int, dbl, chr, fct, ord	ranger	Importance, Oob Error, Weights	response, se
regr.rpart	lgl, int, dbl, fct, ord	rpart	Importance, Missings, Selected Features, Weights	response
regr.svm	int, dbl	e1071		response
regr.xgboost	int, dbl	xgboost	Importance, Missings, Weights	response
surv.blackboost	int, dbl, fct	mboost , distr6 , survival, partykit, mvtnorm		distr, crank, lp
surv.coxph	lgl, int, dbl, fct	survival, distr6	Importance	distr, crank, lp
surv.cvglmnet	int, dbl, fct	glmnet , survival	Weights	crank, lp
surv.flexible	lgl, int, fct, dbl	flexsurv, survival, distr6	Weights	distr, lp, crank
surv.gamboost	int, dbl, fct, lgl	mboost , distr6 , survival		distr, crank, lp
surv.gbm	int, dbl, fct, ord	gbm	Importance, Missings, Weights	crank, lp
surv.glmboost	int, dbl, fct, lgl	mboost , distr6 , survival		distr, crank, lp
surv.glmnet	int, dbl, fct	glmnet , survival	Weights	crank, lp
surv.kaplan	lgl, int, dbl, chr, fct, ord	survival, distr6	Missings	crank, distr
surv.mboost	int, dbl, fct, lgl	mboost , distr6 , survival		distr, crank, lp
surv.nelson	lgl, int, dbl, chr, fct, ord	survival, distr6	Missings	crank, distr
surv.parametric	lgl, int, dbl, fct	survival, distr6	Weights	distr, lp, crank
surv.penalized	int, dbl, fct, ord	penalized, distr6	Importance	distr, crank
surv.randomForestSRC	lgl, int, dbl, fct, ord	randomForestSRC, distr6	Importance, Missings, Weights	crank, distr
surv.ranger	lgl, int, dbl, chr, fct, ord	ranger, distr6	Importance, Oob Error, Weights	distr, crank
surv.rpart	lgl, int, dbl, chr, fct, ord	rpart , distr6 , survival	Importance, Missings, Selected Features, Weights	crank, distr
surv.svm	int, dbl	survivalsvm		crank

10.2 Integrated Performance Measures

Also see the overview on the website of mlr3measures.

Id	Task Type	Required packages	Task Properties	Predict Type
classif.acc	classif	mlr3measures		response
classif.auc	classif	mlr3measures	twoclass	prob
classif.bacc	classif	mlr3measures		response
classif.ce	classif	mlr3measures		response
classif.costs	classif			response
classif.dor	classif	mlr3measures	twoclass	response
classif.fbeta	classif	mlr3measures	twoclass	response
classif.fdr	classif	mlr3measures	twoclass	response
classif.fn	classif	mlr3measures	twoclass	response
classif.fnr	classif	mlr3measures	twoclass	response
classif.fomr	classif	mlr3measures	twoclass	response
classif.fp	classif	mlr3measures	twoclass	response
classif.fpr	classif	mlr3measures	twoclass	response
classif.logloss	classif	mlr3measures		prob
classif.mcc	classif	mlr3measures	twoclass	response
classif.npv	classif	mlr3measures	twoclass	response
classif.ppv	classif	mlr3measures	twoclass	response
classif.precision	classif	mlr3measures	twoclass	response
classif.recall	classif	mlr3measures	twoclass	response
classif.sensitivity	classif	mlr3measures	twoclass	response
classif.specificity	classif	mlr3measures	twoclass	response
classif.tn	classif	mlr3measures	twoclass	response
classif.tnr	classif	mlr3measures	twoclass	response
classif.tp	classif	mlr3measures	twoclass	response
classif.tpr	classif	mlr3measures	twoclass	response
debug				response
oob_error				response
regr.bias	regr	mlr3measures		response
regr.ktau	regr	mlr3measures		response
regr.mae	regr	mlr3measures		response
regr.mape	regr	mlr3measures		response
regr.maxae	regr	mlr3measures		response
regr.medae	regr	mlr3measures		response
regr.medse	regr	mlr3measures		response
regr.mse	regr	mlr3measures		response
regr.msle	regr	mlr3measures		response
regr.pbias	regr	mlr3measures		response
regr.rae	regr	mlr3measures		response
regr.rmse	regr	mlr3measures		response
regr.rmsle	regr	mlr3measures		response
regr.rrse	regr	mlr3measures		response
regr.rse	regr	mlr3measures		response
regr.rsq	regr	mlr3measures		response
regr.sae	regr	mlr3measures		response
regr.smape	regr	mlr3measures		response
regr.srho	regr	mlr3measures		response
regr.sse	regr	mlr3measures		response
selected_features				response
surv.beggC	surv	survAUC		lp
surv.chamblessAUC	surv	survAUC		lp
surv.gonenC	surv	survAUC		lp
surv.graf	surv	distr6		distr
surv.grafSE	surv	distr6		distr
surv.harrellC	surv			crank
surv.hungAUC	surv	survAUC		lp
surv.intlogloss	surv	distr6		distr
surv.intloglossSE	surv	distr6		distr
surv.logloss	surv	distr6		distr
surv.loglossSE	surv	distr6		distr
surv.nagelkR2	surv	survAUC		lp
surv.oquigleyR2	surv	survAUC		lp
surv.songAUC	surv	survAUC		lp
surv.songTNR	surv	survAUC		lp
surv.songTPR	surv	survAUC		lp
surv.unoAUC	surv	survAUC		lp
surv.unoC	surv	survAUC		crank
surv.unoTNR	surv	survAUC		lp
surv.unoTPR	surv	survAUC		lp
surv.xuR2	surv	survAUC		lp
time_both				response
time_predict				response
time_train				response

10.3 Integrated Filter Methods

10.3.1 Standalone filter methods

Id	Task	Task Properties	Features	Package
anova	classif		int, dbl	stats
auc	classif	twoclass	int, dbl	mlr3measures
carscore	regr		dbl	care
cmim	classif, regr		int, dbl, fct, ord	praznik
correlation	regr		int, dbl	stats
disr	classif		int, dbl, fct, ord	praznik
importance	classif		lgl, int, dbl, fct, ord	rpart
information_gain	classif, regr		int, dbl, fct, ord	FSelectorRcpp
jmi	classif		int, dbl, fct, ord	praznik
jmim	classif		int, dbl, fct, ord	praznik
kruskal_test	classif		int, dbl	stats
mim	classif		int, dbl, fct, ord	praznik
mrmr	classif, regr		dbl, fct, int, chr, lgl	praznik
njmim	classif		int, dbl, fct, ord	praznik
performance	classif		lgl, int, dbl, fct, ord	rpart
variance	classif, regr		int, dbl	stats

10.3.2 Algorithms With Embedded Filter Methods

```
##  [1] "classif.featureless"  "classif.ranger"
##  [3] "classif.rpart"        "classif.xgboost"
##  [5] "regr.featureless"     "regr.ranger"
##  [7] "regr.rpart"           "regr.xgboost"
##  [9] "surv.coxph"           "surv.gbm"
## [11] "surv.penalized"       "surv.randomForestSRC"
## [13] "surv.ranger"          "surv.rpart"
```

11 References

Bergstra, James, and Yoshua Bengio. 2012. "Random Search for Hyper-Parameter Optimization." *J. Mach. Learn. Res.* 13. JMLR.org: 281–305.

Bischl, Bernd, Michel Lang, Lars Kotthoff, Julia Schiffner, Jakob Richter, Erich Studerus, Giuseppe Casalicchio, and Zachary M. Jones. 2016. "mlr: Machine Learning in R." *Journal of Machine Learning Research* 17 (170): 1–5. http://jmlr.org/papers/v17/15-066.html.

Breiman, Leo. 1996. "Bagging Predictors." *Machine Learning* 24 (2). Springer: 123–40.

Chandrashekar, Girish, and Ferat Sahin. 2014. "A Survey on Feature Selection Methods." *Computers and Electrical Engineering* 40 (1): 16–28. https://doi.org/https://doi.org/10.1016/j.compeleceng.2013.11.024.

Lang, Michel. 2017. "checkmate: Fast Argument Checks for Defensive R Programming." *The R Journal* 9 (1): 437–45. https://doi.org/10.32614/RJ-2017-028.

Lang, Michel, Martin Binder, Jakob Richter, Patrick Schratz, Florian Pfisterer, Stefan Coors, Quay Au, Giuseppe Casalicchio, Lars Kotthoff, and Bernd Bischl. 2019. "mlr3: A Modern Object-Oriented Machine Learning Framework in R." *Journal of Open Source Software*, December. https://doi.org/10.21105/joss.01903.

R Core Team. 2019. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. https://www.R-project.org/.

Wolpert, David H. 1992. "Stacked Generalization." *Neural Networks* 5 (2): 241–59. https://doi.org/https://doi.org/10.1016/S0893-6080(05)80023-1.