Flexible and Robust Machine Learning Using mlr3 in R

Getting Started

Main Title

Standard blurb goes here

%%Placeholder for Half title

Series page goes here (if applicable); otherwise blank

Flexible and Robust Machine Learning Using mlr3 in R

Imprint page here; PE will provide text

Table of contents

Editors

Editor Bios goes here

List of Figures

List of Tables

Contributors

Contributor 1
Affiliation 1
Affiliation 2

1

Introduction and Overview

Lars Kotthoff
Imperial College London

Raphael Sonabend
**

Natalie Foss
University of Wyoming

Bernd Bischl
University of Wyoming

Welcome to the **M**achine **L**earning in **R** universe (mlr3verse)! Before we begin, make sure you have installed `mlr3` if you want to follow along. We recommend installing the complete `mlr3verse`, which will install all of the important packages.

```
install.packages("mlr3verse")
```

Or you can install just the base package:

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

In our first example, we will show you some of the most basic functionality – training a model and making predictions.

```
library(mlr3)
task = tsk("penguins")
split = partition(task)
learner = lrn("classif.rpart")

learner$train(task, row_ids = split$train)
learner$model
```

```
n= 231

node), split, n, loss, yval, (yprob)
      * denotes terminal node

1) root 231 129 Adelie (0.441558442 0.199134199 0.359307359)
  2) flipper_length< 206.5 144  44 Adelie (0.694444444 0.298611111 0.006944444)
    4) bill_length< 43.05 98   3 Adelie (0.969387755 0.030612245 0.000000000) *
    5) bill_length>=43.05 46   6 Chinstrap (0.108695652 0.869565217 0.021739130) *
  3) flipper_length>=206.5 87   5 Gentoo (0.022988506 0.034482759 0.942528736) *
```

```
predictions = learner$predict(task, row_ids = split$test)
predictions
```

```
<PredictionClassif> for 113 observations:
    row_ids     truth  response
          1    Adelie    Adelie
          2    Adelie    Adelie
          3    Adelie    Adelie
---
        328 Chinstrap Chinstrap
        331 Chinstrap    Adelie
        339 Chinstrap Chinstrap
```

```
predictions$score(msr("classif.acc"))
```

```
classif.acc
  0.9557522
```

In this example, we trained a decision tree on a subset of the `penguins` dataset, made predictions on the rest of the data and then evaluated these with the accuracy measure. In Chapter 2 we will break this down in more detail.

`mlr3` makes training and predicting easy, but it also allows us to perform very complex operations in just a few lines of code:

```
library(mlr3verse)
library(mlr3pipelines)
library(mlr3benchmark)

tasks = tsks(c("breast_cancer", "sonar"))
tuned_rf = auto_tuner(
    tnr("grid_search", resolution = 5),
    lrn("classif.ranger", num.trees = to_tune(200, 500)),
    rsmp("holdout")
)
tuned_rf = pipeline_robustify(NULL, tuned_rf, TRUE) %>>%
    po("learner", tuned_rf)
stack_lrn = ppl(
    "stacking",
    base_learners = lrns(c("classif.rpart", "classif.kknn")),
    lrn("classif.log_reg"))
stack_lrn = pipeline_robustify(NULL, stack_lrn, TRUE) %>>%
    po("learner", stack_lrn)

learners = c(tuned_rf, stack_lrn)
bm = benchmark(benchmark_grid(tasks, learners, rsmp("holdout")))
```

```
bma = bm$aggregate(msr("classif.acc"))[, c("task_id", "learner_id",
  "classif.acc")]
```

```
bma$learner_id = rep(c("RF", "Stack"), 2)
bma
```

```
         task_id learner_id classif.acc
1: breast_cancer         RF   0.9780702
2: breast_cancer      Stack   0.9385965
3:         sonar         RF   0.8550725
4:         sonar      Stack   0.7246377
```

```
as.BenchmarkAggr(bm)$friedman_test()
```

```
Warning: 'as.BenchmarkAggr' is deprecated.
Use 'as_benchmark_aggr' instead.
See help("Deprecated")


    Friedman rank sum test

data:  ce and learner_id and task_id
Friedman chi-squared = 2, df = 1, p-value = 0.1573
```

In this (much more complex!) example we chose two tasks and two machine learning (ML) algorithms ("learners" in `mlr3` terms). We used automated tuning to optimize the number of trees in the random forest learner (Chapter 4) and a ML pipeline that imputes missing data, collapses factor levels, and creates stacked models (Chapter 6). We also showed basic features like loading learners (Chapter 2) and choosing resampling strategies for benchmarking (Chapter 3). Finally, we compared the performance of the models using the mean accuracy on the test set, and applied a statistical test to see if the learners performed significantly different (they did not!).

You will learn how to do all this and more in this book. We will walk through the functionality offered by `mlr3` and the packages in the `mlr3verse` step by step. There are a few different ways you can use this book, which we will discuss next.

1.1 Target audience and how to use this book

The mlr3 ecosystem is the result of many years of methodological and applied research and improving the design and implementation of the packages over the years. This book describes the resulting features of the `mlr3verse` and discusses best practices for ML, technical implementation details, extension guidelines, and in-depth considerations for optimizing ML. It is suitable for a wide range of readers and levels of ML expertise but we assume that users of `mlr3` have taken an introductory machine learning course or have the equivalent expertise and some basic experience with R. A background in computer science or statistics is beneficial for understanding the advanced functionality described in the later chapters of this book, but not required. A comprehensive introduction for those new to machine learning can be found in (James et al. 2014), and (Wickham and Grolemund 2017) gives a comprehensive introduction to data science in R. This book may also be helpful for 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.

Chapter 1, Chapter 2, and Chapter 3 cover the basics of mlr3. These chapters are essential to understanding the core infrastrucure of ML in mlr3. We recommend that all readers study these chapters to become familiar with basic mlr3 terminology, syntax, and style. Chapter 4, Chapter 5, Chapter 6, and **?@sec-preprocessing** contain more advanced implementation details and some ML theory. Chapter 7 delves into detail on domain-specific methods that are implemented in our extension packages. Readers may choose to selectively read sections in this chapter depending on your use cases (i.e., if you have domain-specific problems to tackle), or to use these as introductions to new domains to explore. Chapter 8 contains technical implementation details that are essential reading for advanced users who require parallelisation, custom error handling, and fine control over hyperparameters and large databases. Chapter 9 discusses packages that can be integrated with mlr3 to provide model-agnostic interpretability methods. Finally, anyone who would like to implement their own learners or measures should read Section 8.5.

Of course, you can also read the book cover to cover from start to finish. We have marked sections that are particularly complex, with respect to either technical or methodological detail, as 'optional'.

Each chapter includes tutorials, API references, explanations of methodologies, and exercises to test yourself on what you have learnt. You can find the solutions to these exercises in Appendix A.

If you want to reproduce any of the results in this book, note that at the start of each chapter we run `set.seed(123)` and the `sessionInfo` at the time of publication is printed in Appendix E.

1.2 Installation guidelines

All packages in the mlr3 ecosystem can be installed from GitHub and R-universe; the majority (but not all) packages can also be installed from CRAN. We recommend adding the mlr-org R-universe[1] to your R options so that you can install all packages with `install.packages()` without having to worry which package repository it comes from. To do this, install the `usethis` package and run the following:

```
usethis::edit_r_profile()
```

In the file that opens add or change the `repos` argument in `options` so it looks something like this (you might need to add the full code block below or just edit the existing `options` function).

```
options(repos = c(
  mlrorg = "https://mlr-org.r-universe.dev",
  CRAN = "https://cloud.r-project.org/"
```

[1] R-universe is an alternative package repository to CRAN. The bit of code below tells R to look at both R-universe and CRAN when trying to install packages. R will always install the latest version of a package.

```
))
```

Save the file, restart your R session, and you are ready to go!

```
install.packages("mlr3verse")
```

If you want the latest development version of any of our packages, run

```
remotes::install_github("mlr-org/{pkg}")
```

with `{pkg}` replaced with the name of the package you want to install (e.g., `remotes::install_github("mlr-org/mlr3tuning")`). You can see an up-to-date list of all our extension packages at https://github.com/mlr-org/mlr3/wiki/Extension-Packages.

1.3 Community links

The mlr community is open to all and we welcome everybody, from those completely new to ML and R to advanced coders and professional data scientists. You can reach us on our Mattermost[2].

For case studies and how-to guides, check out the mlr3gallery[3] for extended practical blog posts. For updates on mlr you might find our blog[4] a useful point of reference.

We appreciate all contributions, whether they are bug reports, feature requests, or pull requests that fix bugs or extend functionality. Each of our GitHub repositories includes issues and pull request templates to ensure we can help you as much as possible to get started. Please make sure you read our code of conduct[5] and contribution guidelines[6]. With so many packages in our universe it may be hard to keep track of where to open issues. As a general rule:

1. If you have a question about using any part of the mlr3 ecosystem, ask on StackOverflow and use the tag #mlr3 – one of our team will answer you there. Be sure to include a reproducible example (reprex) and if we think you found a bug then we will refer you to the relevant GitHub repository.
2. Bug reports or pull requests about core functionality (train, predict, etc.) should be opened in the mlr3 GitHub repository.
3. Bug reports or pull requests about learners should be opened in the mlr3extralearners GitHub repository.
4. Bug reports or pull requests about measures should be opened in the mlr3measures GitHub repository.
5. Bug reports or pull requests about domain specific functionality should be opened in the GitHub repository of the respective package (see Chapter 1).

[2] https://lmmisld-lmu-stats-slds.srv.mwn.de/signup_email?id=6n7n67tdh7d4bnfxydqomjqspo
[3] https://mlr-org.com/gallery.html
[4] https://mlr-org.com/blog.html
[5] https://github.com/mlr-org/mlr3/blob/main/.github/CODE_OF_CONDUCT.md
[6] https://github.com/mlr-org/mlr3/blob/main/CONTRIBUTING.md

Do not worry about opening an issue in the wrong place, we will transfer it to the right one.

1.4 mlr3book style guide

Throughout this book we will use our own style guide that can be found in the mlr3 wiki[7]. Below are the most important style choices relevant to the book.

1. We always use `=` instead of `<-` for assignment.
2. Class names are in `UpperCamelCase`
3. Function and method names are in `lower_snake_case`
4. When referencing functions, we will only include the package prefix (e.g., `pkg::function`) for functions outside the mlr3 universe or when there may be ambiguity about in which package the function lives. Note you can use `environment(function)` to see which namespace a function is loaded from.
5. We denote packages, fields, methods, and functions as follows:
   - `package` - With link (if online) to package CRAN, R-Universe, or GitHub page
   - `package::function()` (for functions *outside* the mlr-org ecosystem)
   - `function()` (for functions *inside* the mlr-org ecosystem)
   - `$field` for fields (data encapsulated in a R6 class)
   - `$method()` for methods (functions encapsulated in a R6 class)

1.5 MLR: Machine Learning in R

Regression Tasks
Classification Tasks
Learners

The (**M**achine **L**earning in **R**) `mlr3` (Lang et al. 2019) package and ecosystem provide a generic, object-oriented, and extensible framework for regression (Section 2.1), classification (Section 2.4), and other machine learning tasks (Chapter 7) for the R language (R Core Team 2019). This unified interface provides functionality to extend and combine existing machine learning algorithms (Learners (Section 2.2)), intelligently select and tune the most appropriate technique for a given machine learning task (Section 2.1), and perform large-scale comparisons that enable meta-learning. Examples of this advanced functionality include hyperparameter tuning (Chapter 4) and feature selection (Chapter 5). Parallelization of many operations is natively supported (Section 8.1).

`mlr3` has similar overall aims to `caret` and `tidymodels` for R, scikit-learn[8] for Python, and MLJ[9] for Julia. In general `mlr3` is designed to provide more flexibility than other machine learning frameworks while still offering easy ways to use advanced functionality. While tidymodels in particular makes it very easy to perform simple machine learning tasks, `mlr3`

[7] https://github.com/mlr-org/mlr3/wiki/Style-Guide
[8] https://scikit-learn.org/
[9] https://alan-turing-institute.github.io/MLJ.jl/dev/

is more geared towards advanced machine learning. To get a quick overview of how to do things in `mlr3`, see the mlr3 cheatsheets[10].

> **Note**
>
> `mlr3` provides a unified interface to existing learners in R. With few exceptions, we do not implement any learners ourselves, although we often augment the functionality provided by the underlying learners. This includes, in particular, the definition of hyperparameter spaces for tuning.

1.6 From mlr to mlr3

The `mlr` package (Bischl et al. 2016) was first released to CRAN[11] 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. In hindsight, we saw that some design and architecture choices in `mlr` made it difficult to support new features, in particular with respect to machine learning pipelines. Furthermore, the R ecosystem and helpful packages such as `data.table` have undergone major changes after the initial design of `mlr`.

It would have been difficult to integrate all of these changes into the original design of `mlr`. Instead, we decided to start working on a reimplementation in 2018, which resulted in the first release of `mlr3` on CRAN in July 2019.

The new design and the integration of further and newly-developed R packages (especially `R6`, `future`, and `data.table`) makes `mlr3` much easier to use, maintain, and in many regards more efficient than its predecessor `mlr`. The packages in the ecosystem are less tightly coupled, making them easier to maintain and easier to develop, especially very specialized packages.

1.7 Design principles

> **Advanced section**

We follow these general design principles in the `mlr3` package and `mlr3verse` ecosystem.

- **Object-oriented programming (OOP)**. We embrace `R6` for a clean, object-oriented design, object state-changes, and reference semantics. This means that the state of common objects (e.g. tasks (Section 2.1) and learners (Section 2.2)) is encapsulated within the object, for example to keep track of whether a model has been trained, without the user having to worry about this. We also use inheritance to specialize objects, e.g. all learners are derived from a common base class that provides basic functionality.

[10] https://cheatsheets.mlr-org.com/
[11] https://cran.r-project.org

- **Tabular data**. Embrace `data.table` for its top-notch computation performance as well as tabular data as a structure that can be easily processed further.
- **Unify input and output data formats.** 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). We use data.table which documents return types unlike other mechanisms popular in base R which "simplify" the result unpredictably (e.g., `sapply()` or the `drop` argument for indexing data.frames). And we have extensive unit tests!
- **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`, which makes installation and maintenance easier. We still provide the same functionality, but it is split into more packages that have fewer dependencies individually. One benefit of having the visualization functionality in a separate package, for example, is that the user can use mlr3 for ML operations without having to install graphical dependencies.
- **Separation of computation and presentation**. Most packages of the `mlr3` ecosystem focus on processing and transforming data, applying machine learning algorithms, and computing results. Our core packages do not provide visualizations because their dependencies would make installation unnecessarily complex, especially on headless servers (i.e. computers without a monitor where graphical libraries are not installed). For the same reason, visualizations of data and results are provided in the extra package `mlr3viz`, which avoids dependencies on `ggplot2`.

1.8 Package ecosystem

`mlr3` depends on the following popular and well-established packages that are not developed by core members of the `mlr3` team:

- `R6`: The class system predominantly used in mlr3.
- `data.table`: High-performance extension of R's `data.frame`.
- `digest`: Cryptographic hash functions.
- `uuid`: Generation of universally unique identifiers.
- `lgr`: Highly configurable logging library.
- `mlbench` and `palmerpenguins`: More machine learning data sets.
- `evaluate`: For capturing output, warnings, and exceptions (Section 8.2).
- `future` / `future.apply` / `parallelly`: For parallelization (Section 8.1).

The `mlr3` package itself provides the base functionality that the rest of ecosystem (`mlr3verse`) relies on and the fundamental building blocks for machine learning. **?@fig-mlr3verse** shows the packages in the `mlr3verse` that extend `mlr3` with capabilities for preprocessing, pipelining, visualizations, additional learners, additional task types, and more.

> **Tip**
>
> Instead of loading multiple extension packages individually, it is often more convenient to load the `mlr3verse` package instead. It makes the functions from most `mlr3`

packages that are used for common machine learning and data science tasks available.

> **Tip**
>
> A complete list with links to the repositories for the respective packages can be found on our package overview page[12].

We build on `R6` for object orientation and `data.table` to store and operate on tabular data. Both are core to `mlr3`; we briefly introduce both packages for beginners. While in-depth expertise with these packages is not necessary, a basic understanding is required to work effectively with `mlr3`.

1.9 How R6 & data.table apply to `mlr3`

1.9.1 Quick R6 introduction for beginners

`R6` is one of R's more recent paradigm for object-oriented programming (OOP). 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`.

Objects are created by calling the constructor of an `R6::R6Class()` object, specifically the initialization method `$new()`. For example, `foo = Foo$new(bar = 1)` creates a new object of class `Foo`, setting the `bar` argument of the constructor to the value `1`.

Objects have mutable state that is encapsulated in their fields, which can be accessed through the dollar operator. We can access the `bar` value in the `foo` variable from above through `foo$bar` and set its value by assigning the field, e.g. `foo$bar = 2`.

In addition to fields, objects expose methods that allow to inspect the object's state, retrieve information, or perform an action that changes the internal state of the object. For example, the `$train()` method of a learner changes the internal state of the learner by building and storing a model, which can then be used to make predictions.

Objects can have public and private fields and methods. The public fields and methods define the API to interact with the object. Private methods are only relevant for you if you want to extend `mlr3`, e.g. with new learners.

Technically, R6 objects are environments, and as such have reference semantics. For example, `foo2 = foo` does not create a copy of `foo` in `foo2`, but another reference to the same actual object. Setting `foo$bar = 3` will also change `foo2$bar` to `3` and vice versa.

To copy an object, use the `$clone()` method and the `deep = TRUE` argument for nested objects, for example, `foo2 = foo$clone(deep = TRUE)`.

[12] https://mlr-org.com/ecosystem.html

> **Tip**
>
> For more details on R6, have a look at the excellent R6 vignettes[13], especially the introduction[14]. For comprehensive R6 information, we refer to the R6 chapter from Advanced R[15].

1.9.2 Quick `data.table` introduction for beginners

The package `data.table` implements a popular alternative to R's `data.frame()`, i.e. an object to store tabular data. We decided to use `data.table` because it is blazingly fast and scales well to bigger data.

> **Note**
>
> Many `mlr3` functions return `data.table`s which can conveniently be subsetted or combined with other outputs. If you do not like the syntax or are feeling more comfortable with other tools, base `data.frame`s or `tibble`/`dplyr`s are just a `as.data.frame()` or `as_tibble()` away.

Data tables are constructed with the `data.table()` function (whose interface is similar to `data.frame()`) or by converting an object with `as.data.table()`.

```
library("data.table")
dt = data.table(x = 1:6, y = rep(letters[1:3], each = 2))
dt
```

```
   x y
1: 1 a
2: 2 a
3: 3 b
4: 4 b
5: 5 c
6: 6 c
```

`data.table`s can be used much like `data.frame`s, but they do provide additional functionality that makes complex operations easier. For example, data can be summarized by groups with the `[` operator:

```
dt[, mean(x), by = "y"]
```

```
   y  V1
1: a 1.5
2: b 3.5
3: c 5.5
```

There is also extensive support for many kinds of database join operations (see e.g. this

[15] https://r6.r-lib.org/
[15] https://r6.r-lib.org/articles/Introduction.html
[15] https://adv-r.hadley.nz/r6.html

RPubs post by Ronald Stalder[16]) that make it easy to combine multiple `data.tables` in different ways.

> **Tip**
>
> For an in-depth introduction, we refer the reader to the excellent data.table introduction vignette[17].

1.10 Essential mlr3 utilities

Helper Functions

Most objects in `mlr3` can be created through convenience functions called *helper functions* or *sugar functions*. They provide shortcuts for common code idioms, reducing the amount of code a user has to write. We heavily use helper functions throughout this book and give the equivalent "full form" for complete detail. In most cases, the helper functions will achieve what you want to do, and you only have to consider using the full R6 code if you use custom objects or extensions. For example `lrn("regr.rpart")` is the sugar version of `LearnerRegrRpart$new()`.

Dictionaries

`mlr3` uses dictionaries to store objects like learners or tasks. These are key-value stores that allow to associate a key with a value that can be an R6 object, much like paper dictionaries associate words with their definitions. Values in dictionaries are often accessed through sugar functions (i.e. `lrn()`) that automatically use the applicable dictionary (i.e. `mlr_learners`); only the key to be retrieved needs to be specified. Dictionaries are used to group relevant objects so that they can be listed and retrieved easily. For example, the "featureless" classification learner can be retrieved through `lrn("classif.featureless")` or directly from the `mlr_learners` dictionary using the key `"classif.featureless"` (`mlr_learners$get("classif.featureless")`). You can get an overview of all available learners by inspecting the dictionary, e.g. with `as.data.table(mlr_learners)`.

mlr3viz

`mlr3viz` is the package for all plotting functionality in the `mlr3` ecosystem. The package uses a common theme (`ggplot2::theme_minimal()`) so that all generated plots have a similar aesthetic. Under the hood, `mlr3viz` uses `ggplot2`. `mlr3viz` extends `fortify` and `autoplot` for use with common `mlr3` outputs including Prediction, Learner, and Benchmark objects (these objects will be introduced and covered in the next chapters). The most common use of `mlr3viz` is the `autoplot()` function, where the type of the object passed determines the type of the plot. Plot types are documented in the respective manual page that can be accessed through `?autoplot.X`. For example, the documentation of plots for regression tasks can be found by running `?autoplot.TaskRegr`.

[16] https://rstudio-pubs-static.s3.amazonaws.com/52230_5ae0d25125b544caab32f75f0360e775.html
[17] https://cran.r-project.org/web/packages/data.table/vignettes/datatable-intro.html

2

Fundamentals

Natalie Foss
University of Wyoming

Lars Kotthoff
University of Wyoming

In this chapter, we will introduce the `mlr3` objects and corresponding `R6` classes that implement the essential building blocks of machine learning. These building blocks include the data (and the methods of creating training and test sets), the machine learning algorithm (and its training and prediction process), the configuration of a machine learning algorithm through its hyperparameters, and evaluation measures to assess the quality of predictions.

In essence, machine learning means learning relationships from data. In supervised learning, datasets consist of observations (rows in tabular data) that are labeled, which means that each data point includes features (columns in tabular data) and a quantity that we are trying to predict, also called 'target'. For example, we might want to predict the miles per gallon a car gets based on features such as its horsepower and the number of gears. Data and information on what they represent, along with what quantities to predict are called "tasks" in `mlr3` (Section 2.1) – they can be thought of as machine learning tasks we are trying to solve. There can be more than one task per dataset, for example ones that include different sets of features, observations, or predict different target quantities.

Supervised learning can be further divided into regression (predicting numeric target values) and classification (predicting categorical target values/labels). In either case, the goal is to build a model that captures the relationship between features and target. We can build such models using machine learning algorithms, for example decision trees, support vector machines, neural networks, and many more. A machine learning algorithm, given training data, induces such a model. Machine learning algorithms are called "learners" in `mlr3` (Section 2.2) – given data, they learn models. Each learner has a parameterized space that potential models are drawn from and during the training process, these parameters are fitted to best match the data. For example, the parameters could be the weights given to individual features when predicting a quantity in linear regression. For other learners, the parameters are not as explicit, for example for decision tree learners where a fitted model corresponds to a particular decision tree. All learners optimize a so-called loss function during training, i.e. training a learner means finding the model that optimizes the loss. In general, a loss function quantifies the mismatch between ground truth target values in the training data and the predictions of the model. In addition to the parameters that are fit to the data, most learners also have hyperparameters that affect how the model is fit.

Given a model, we can make predictions (Section 2.2.2) on new data. A model is only useful though if it generalizes beyond the training data. Otherwise, we could build a perfect model by simply memorizing the training data. Therefore, separate test data is used to evaluate models in an unbiased way and to assess to what extent they have learned the true relationships that underlie the data (Chapter 3). We can evaluate models in `mlr3` in many ways (Section 2.3). In general, we use the same kind of loss function that the learner used to

build the model, but now with data that was not used during training. The performance of a model, quantified by the value of the loss function when evaluated on new data, is called the estimate of the generalization error – how well do we expect this model to do in general? `mlr3` calls loss functions "measures". We can use different measures for training and testing, although it makes most sense for the measures to be the same.

Much more information on (supervised) machine learning can be found in Hastie, Friedman, and Tibshirani (2001), James et al. (2014), or Bishop (2006).

The basic idea is illustrated in the following figure:

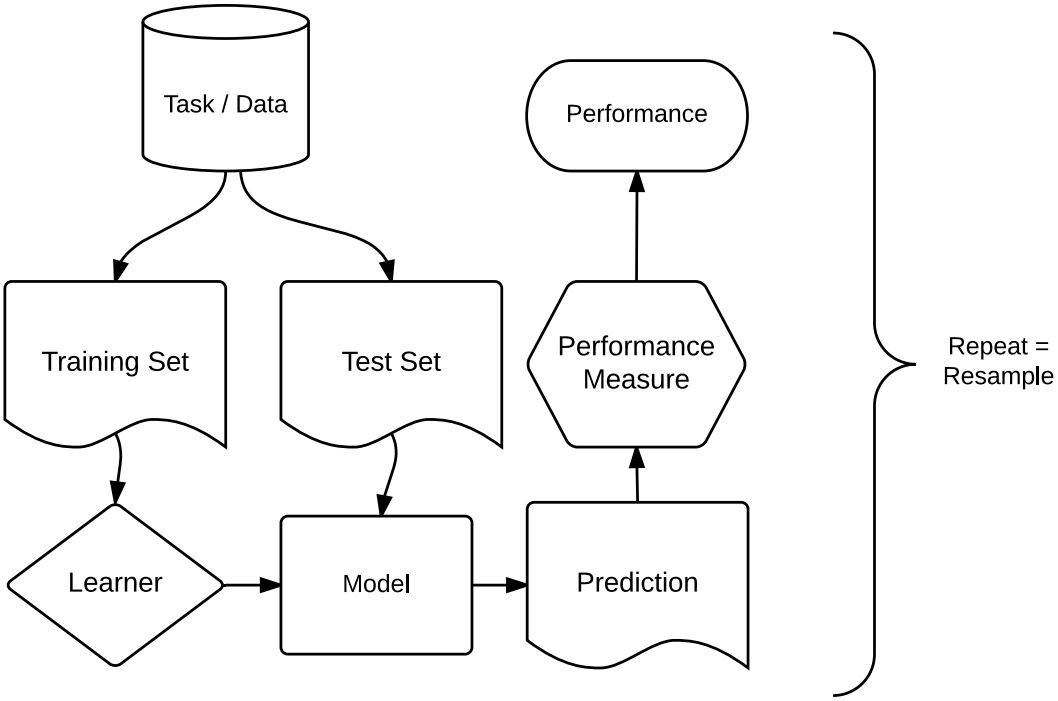


Figure 2.1: General overview of the machine learning process.

2.1 Tasks

Tasks are objects that contain the (usually tabular) data and additional meta-data that define a machine learning problem. The meta-data contain, for example, the name of the target feature for supervised machine learning problems. This information is used automatically by operations that can be performed on a task so that for example the user does not have to specify the prediction target every time a model is trained.

2.1.1 Constructing Tasks

mlr3 includes a few predefined machine learning tasks in an R6 `Dictionary` named mlr_tasks.

```
mlr_tasks
```

```
<DictionaryTask> with 20 stored values
Keys: bike_sharing, boston_housing, breast_cancer, german_credit, ilpd,
  iris, kc_housing, moneyball, mtcars, optdigits, penguins,
  penguins_simple, pima, ruspini, sonar, spam, titanic, usarrests,
  wine, zoo
```

To get a task from the dictionary, use the `tsk()` function and assign the return value to a new variable. Here, we retrieve the `mtcars regression task`, which is provided by the package `datasets`:

```
task_mtcars = tsk("mtcars")
task_mtcars
```

```
<TaskRegr:mtcars> (32 x 11): Motor Trends
* Target: mpg
* Properties: -
* Features (10):
  - dbl (10): am, carb, cyl, disp, drat, gear, hp, qsec, vs, wt
```

To get more information about a particular task, it is easiest to use the `help()` method that most mlr3 objects come with:

```
task_mtcars$help()
```

> **Tip**
>
> If you are familiar with R's help system (i.e. the `help()` and `?` functions), this may seem confusing. `task_mtcars` is the variable that holds the mtcars task, not a function, and hence we cannot use `help()` or `?`.

Alternatively, the corresponding man page can be found under `mlr_tasks_<id>`, e.g.

```
help("mlr_tasks_mtcars")
```

We can also load the data separately and convert it to a task, without using the `tsk()` function that mlr3 provides. If the data we want to use does not come with mlr3, it has to be done this way.

For example, the data for `mtcars` is also available separately, as a `data.frame()` and not a task. `mtcars` contains characteristics for different types of cars, along with their fuel consumption. We want to predict the numeric target feature stored in column `"mpg"` (miles per gallon).

```
data("mtcars", package = "datasets")
str(mtcars)
```

```
'data.frame':   32 obs. of  11 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 ...
 $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
 $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
 $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
 $ qsec: num  16.5 17 18.6 19.4 17 ...
 $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
 $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
 $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
 $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

We create the regression task, i.e. we construct a new instance of the R6 class `TaskRegr`. An easy way to do this is to use the function `as_task_regr()` to convert our `data.frame()` to a regression task, specifying the target feature in an additional argument. Before we give the data to `as_task_regr()`, we can process it using the usual R functions, for example to select a subset of data.

```
library("mlr3")
mtcars_subset = subset(mtcars, select = c("mpg", "cyl", "disp"))

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

> **Tip**
>
> The task constructors `as_task_regr()` and `as_task_classif()` will check for non-ASCII characters in the column names of your data. As many ML models do not work properly with arbitrary UTF8 names, `mlr3` defaults to throw an error if any of the column names contains either a non-ASCII character or does not comply with R's variable naming scheme. We generally recommend converting names with `make.names()` first, but you can also set the option `mlr3.allow_utf8_names` to true to relax the check (but do not be surprised if a model fails).

The data can be any rectangular data format, e.g. a `data.frame()`, `data.table()`, or `tibble()`. Internally, the data is converted and stored in a `DataBackend`. The `target` argument specifies the prediction target column. The `id` argument is optional and specifies an identifier for the task that is used in plots and summaries. If no `id` is provided, the deparsed name of the data will be used (an R way of turning data into strings).

```
task_mtcars
```

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

Printing a task gives a short summary: it has 32 observations and 3 columns, of which `mpg` is the target and 2 are features stored in double-precision floating point format.

We can plot the task using the `mlr3viz` package, which gives a graphical summary of the distribution of the target and feature values:

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

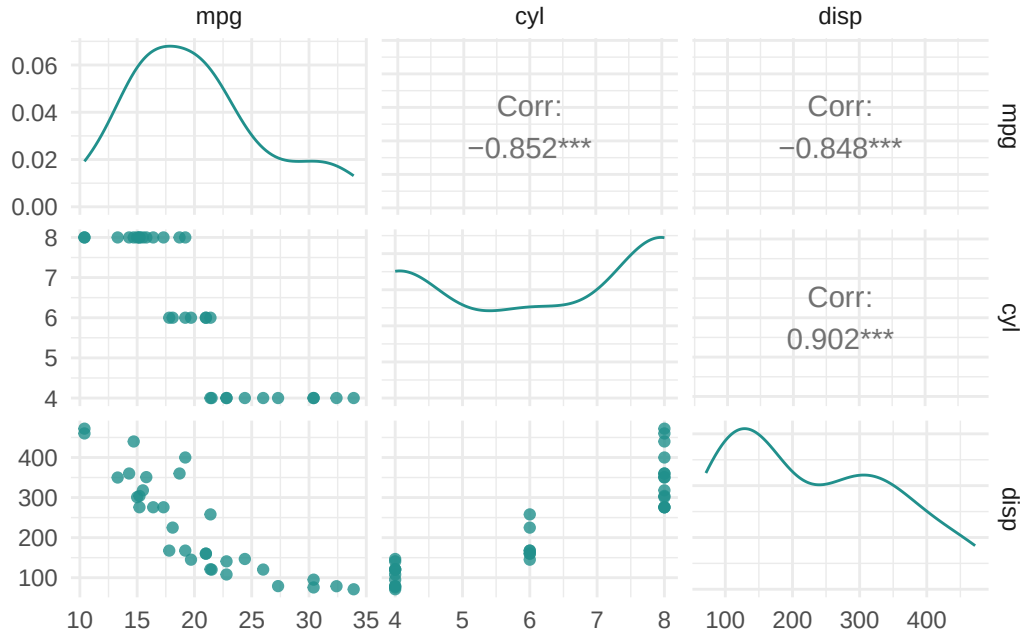


Figure 2.2: Overview of the mtcars dataset.

2.1.2 Retrieving Data

The `Task` object primarily represents a tabular dataset, combined with meta-data about which columns of that data should be used to predict which other columns in what way, and some more information about column data types.

Various fields can be used to retrieve meta-data about a task. The dimensions, for example, can be retrieved using `$nrow` and `$ncol`:

```
c(task_mtcars$nrow, task_mtcars$ncol)
```

```
[1] 32  3
```

The names of the feature and target columns are stored in the `$feature_names` and `$target_names` slots, respectively. Here, "target" refers to the feature we want to predict and "feature" to the predictors for the task.

```
list(task_mtcars$feature_names, task_mtcars$target_names)
```

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

[[2]]
[1] "mpg"
```

While the columns of a task have unique `character`-valued names, their rows are identified by unique natural numbers, called row IDs. They can be accessed through the `$row_ids` field:

```
head(task_mtcars$row_ids)
```

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

Row IDs are not used as features when training or predicting; they are meta-data that allows to access individual observations.

> **Warning**
>
> Although the row IDs are typically just the sequence from `1` to `nrow(data)`, they are only guaranteed to be unique natural numbers. It is possible that they do not start at `1`, that they are not increasing by 1 each, or that they are not even in increasing order. This allows to transparently operate on real database management systems, where uniqueness is the only requirement for primary keys.

The data contained in a task can be accessed through `$data()`, which returns a `data.table` object. It has optional `rows` and `cols` arguments to specify subsets of the data to retrieve. When a database backend is used, this avoids loading unnecessary data into memory, making it more efficient than retrieving the entire data first and then subsetting it using `[<rows>, <cols>]`.

```
task_mtcars$data()
```

```
     mpg cyl  disp
 1: 21.0   6 160.0
 2: 21.0   6 160.0
 3: 22.8   4 108.0
 4: 21.4   6 258.0
 5: 18.7   8 360.0
---
28: 30.4   4  95.1
29: 15.8   8 351.0
30: 19.7   6 145.0
31: 15.0   8 301.0
32: 21.4   4 121.0
```

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

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

A shortcut to extract all data from a task is to simply convert it to a `data.table`:

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

```
      mpg             cyl             disp
 Min.   :10.40   Min.   :4.000   Min.   : 71.1
 1st Qu.:15.43   1st Qu.:4.000   1st Qu.:120.8
 Median :19.20   Median :6.000   Median :196.3
 Mean   :20.09   Mean   :6.188   Mean   :230.7
 3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0
 Max.   :33.90   Max.   :8.000   Max.   :472.0
```

2.1.3 Task Mutators

It is often necessary to create tasks that encompass subsets of other tasks' data, for example to manually create train-test-splits, or to fit models on a subset of given features. Restricting tasks to a given set of features can be done by calling `$select()` with the desired feature names. Restriction to rows (observations) is done with `$filter()` with the row IDs.

```
task_mtcars_small = tsk("mtcars") # initialize with the full task
task_mtcars_small$select(c("am", "carb")) # keep only these features
task_mtcars_small$filter(2:4) # keep only these rows
task_mtcars_small$data()
```

```
    mpg am carb
1: 21.0  1    4
2: 22.8  1    1
3: 21.4  0    1
```

These methods are so-called *mutators*; they modify the given `Task` in place. If you want to have an unmodified version of the task, you need to use the `$clone()` method to create a copy first.

```
task_mtcars_smaller = task_mtcars_small$clone()
task_mtcars_smaller$filter(2)
task_mtcars_smaller$data()
```

```
   mpg am carb
1:  21  1    4
```

```
task_mtcars_small$data()  # the original task is unmodified
```

```
    mpg am carb
1: 21.0  1    4
2: 22.8  1    1
```

```
3: 21.4   0    1
```

Note also how the last call to `$filter(2)` did not select the second row of `task_mtcars_small`, but the row with ID 2, which is the *first* row of `task_mtcars_small`.

> **Tip**
>
> If you need to work with row numbers instead of row IDs, you can work on the vector of row IDs:

```
# keep the 2nd row:
keep = task_mtcars_small$row_ids[2] # extracts ID of 2nd row
task_mtcars_smaller$filter(keep)
```

The methods above allow to subset the data; the methods `$rbind()` and `$cbind()` allow to add extra rows and columns to a task.

```
task_mtcars_smaller$rbind( # add another row
  data.frame(mpg = 23, am = 0, carb = 3)
)
task_mtcars_smaller$data()
```

```
   mpg am carb
1:  21  1    4
2:  23  0    3
```

2.2 Learners

Objects of class `Learner` provide a unified interface to many popular machine learning algorithms in R. They are available through the `mlr_learners` dictionary. The list of learners supported in the base package `mlr3` is deliberately small to avoid dependencies; support for additional learners is provided by the `mlr3learners` and `mlr3extralearners` packages.

Learners encapsulate methods to train a model and make predictions using it given a `Task` and provide meta-data about the learners. The base class of each learner is `Learner`.

To retrieve a `Learner` from the `mlr_learners` dictionary, use the function `lrn()`:

```
learner_rpart = lrn("regr.rpart")
```

Each learner provides the following meta-data:

- `$feature_types`: the type of features the learner can deal with.
- `$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.

- `$predict_types`: possible prediction types. For example, a regression learner can predict numerical values ("response") and may be able to predict the standard error of a prediction ("se").
- `$param_set`: the set of hyperparameters. See Section 2.2.1.1.

This information can be queried through these slots, or seen at a glance when printing the learner:

```
learner_rpart
```

```
<LearnerRegrRpart:regr.rpart>: Regression Tree
* Model: -
* Parameters: xval=0
* Packages: mlr3, rpart
* Predict Types:  [response]
* Feature Types: logical, integer, numeric, factor, ordered
* Properties: importance, missings, selected_features, weights
```

All learners work in two stages:

- **Training**: A training task (features and target data) is passed to the learner's `$train()` function which trains and stores a model, i.e. the learned relationship of the features to the target.
- **Prediction**: New data, usually a different partition of the original dataset, is passed to the `$predict()` method of the trained learner. The model trained in the first step is used to predict the target values, e.g. the numerical value for regression problems.

> **Warning**
>
> A learner that has not been trained cannot make predictions and will throw an error if `$predict()` is called on it.

2.2.1 Training the learner

We train the model by giving a task to the learner. It is a good idea to hold back some data from the training process used to assess the quality of the predictions made by the trained model. The `partition()` function randomly splits the task into two disjoint sets: a training set (67% of the total data, the default) and a test set (33% of the total data, the data not part of the training set).

```
splits = partition(task_mtcars)
splits
```

```
$train
 [1]  1  2  3  4  5  8 25 30 32  6 11 13 17 22 23 24 29 31 19 20 28

$test
 [1]  9 10 21 27  7 12 14 15 16 18 26
```

We learn a regression tree by calling the `$train()` method of the learner, specifying the task and the part of it to use for training (`splits$train`). This operation adds the learned

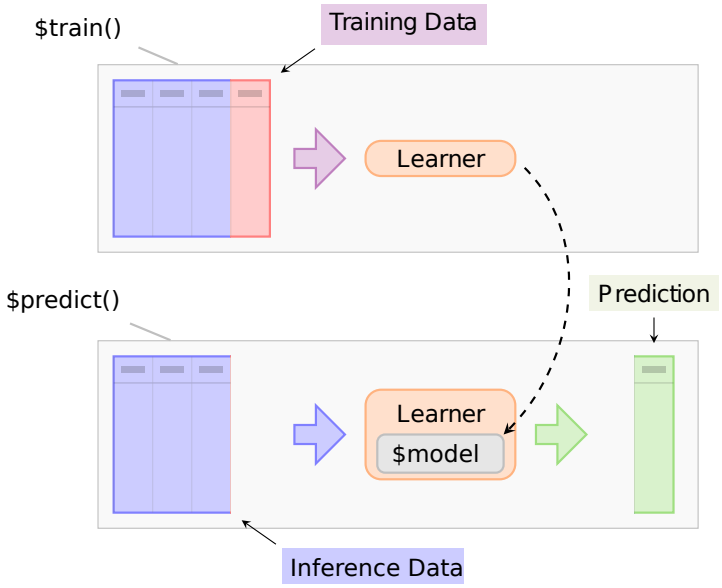


Figure 2.3: Overview of the different stages of a learner.

model to the existing `Learner` object. We can now access the stored model via the field `$model`.

```
learner_rpart$train(task_mtcars, splits$train)
learner_rpart$model
```

```
n= 21

node), split, n, deviance, yval
      * denotes terminal node

1) root 21 617.38670 20.33333
  2) disp>=153.35 14  88.36857 17.42857 *
  3) disp< 153.35 7 174.63710 26.14286 *
```

We see that the learner has identified features in the task that are predictive of the target (`mpg`) and uses them to partition observations in the tree. The textual representation of the model depends on the type of learner. For more information on this particular type of model and how it is printed, see `rpart::print.rpart()`.

2.2.1.1 Hyperparameters

The model seems rather simplistic, using only a single feature and a single set of branches. Each learner has hyperparameters that control its behavior and allow to influence the way a model is learned. Setting hyperparameters to values appropriate for a given machine learning task is crucial for good predictive performance. The field `param_set` stores a description of the hyperparameters the learner has, their ranges, defaults, and current values:

```
learner_rpart$param_set
```

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

The set of current hyperparameter values is stored in the `values` field of the `param_set` field. You can access and change the current hyperparameter values by accessing this field, which stores a named list:

```
learner_rpart$param_set$values
```

```
$xval
[1] 0
```

`mlr3` manages hyperparameters through the `paradox` package, developed by the mlr3 team. Hyperparameters affect the way a model is fit to the given data by a learner. For example, the `xval` hyperparameter for the `rpart` learner controls the number of internal cross-validations. The `mlr3` universe provides information on possible hyperparameter settings for all its learners, which has the following advantages:

1. Available hyperparameter settings, their data types, and accepted ranges can easily be listed without having to look through the respective manual pages.
2. Hyperparameters can be tuned automatically; covered in Hyperparameter Tuning.

The following table lists the available hyperparameter types, all of which derive from the `Param` base class that represents a single hyperparameter.

Hyperparameter Class	Description
`ParamDbl`	Real-valued (Numeric) Parameters
`ParamInt`	Integer Parameters
`ParamFct`	Categorical (Factor) Parameters
`ParamLgl`	Logical / Boolean Parameters
`ParamUty`	Untyped Parameters

Table 2.1: Hyperparameter Classes and the type of hyperparameter they represent.

Individual hyperparameters can be combined into `ParamSet` objects. Every learner has a `ParamSet` that defines the hyperparameter values to use when training a model. You can access this object using the `$param_set` field.

Continuing the example above, we focus on a subset of the hyperparameters for the sake of brevity:

```
all_params = learner_rpart$param_set
subset = all_params$subset(c("cp", "keep_model", "minsplit", "xval"))
subset
```

```
<ParamSet>
          id    class lower upper nlevels default value
1:        cp ParamDbl     0     1     Inf    0.01
2: keep_model ParamLgl    NA    NA       2   FALSE
3:   minsplit ParamInt     1   Inf     Inf      20
4:       xval ParamInt     0   Inf     Inf      10     0
```

The printed output above informs us what values are expected:

- `cp` must be a "double" between 0 and 1 with a default of 0.01.
- `keep_model` is a logical flag with default `FALSE`.
- `minsplit` must be an integer greater than 0 with a default of 20.
- `xval` must be 0 or a positive integer with a default of 10, but was set to 0.

Most learners do not have hyperparameter values set unless explicitly specified when instantiating the learner, i.e. the *value* column is empty. Some learners, such as `LearnerClassifRpart`, automatically initialize some of their hyperparameters to values that differ from the default, as shown above.

In more complex hyperparameter spaces, there may be additional columns. For example, the hyperparameter set of `LearnerClassifSVM` from the `mlr3learners` package has conditional dependencies:

```
lrn("classif.svm")$param_set
```

```
<ParamSet>
                id    class lower upper nlevels          default parents
 1:       cachesize ParamDbl  -Inf   Inf     Inf               40
 2:   class.weights ParamUty    NA    NA     Inf
 3:           coef0 ParamDbl  -Inf   Inf     Inf                0  kernel
 4:            cost ParamDbl     0   Inf     Inf                1    type
 5:           cross ParamInt     0   Inf     Inf                0
 6: decision.values ParamLgl    NA    NA       2            FALSE
 7:          degree ParamInt     1   Inf     Inf                3  kernel
 8:         epsilon ParamDbl     0   Inf     Inf              0.1
 9:          fitted ParamLgl    NA    NA       2             TRUE
10:           gamma ParamDbl     0   Inf     Inf   <NoDefault[3]>  kernel
11:          kernel ParamFct    NA    NA       4           radial
12:              nu ParamDbl  -Inf   Inf     Inf              0.5    type
13:           scale ParamUty    NA    NA     Inf             TRUE
14:       shrinking ParamLgl    NA    NA       2             TRUE
15:       tolerance ParamDbl     0   Inf     Inf            0.001
16:            type ParamFct    NA    NA       2 C-classification
1 variable not shown: [value]
```

The additional column *parents* indicate which other hyperparameters a `Param` depends. The field `$deps` lists the hyperparameters that have such dependencies, and the field `$cond` the conditional expressions:

```
lrn("classif.svm")$param_set$deps
```

```
       id     on           cond
1:   cost   type <CondEqual[9]>
2:     nu   type <CondEqual[9]>
3: degree kernel <CondEqual[9]>
4:  coef0 kernel <CondAnyOf[9]>
5:  gamma kernel <CondAnyOf[9]>
```

```
lrn("classif.svm")$param_set$deps$cond
```

```
[[1]]
CondEqual: x = C-classification

[[2]]
CondEqual: x = nu-classification

[[3]]
CondEqual: x = polynomial

[[4]]
CondAnyOf: x   {polynomial, sigmoid}

[[5]]
CondAnyOf: x   {polynomial, radial, sigmoid}
```

For example, the hyperparameter `cost` can only be set if `type` is set to `"C-classification"`; otherwise it does not apply. Similarly, the (polynomial) `degree` can only be specified if the `kernel` is `"polynomial"`.

2.2.1.2 Setting Hyperparameter Values

To change the hyperparameters of a learner, the `$values` field of the `ParamSet` must be changed. There are three ways to change the values of an existing hyperparameter set:

Method	Description
`param_set$values$<par> = <value>`	Changes only a single value
`param_set$values = list(<par1> = <value1>, ...)`	Replace all parameter values with the given list
`param_set$set_values(<par1> = <value1>, ...)`	Changes only the provided values

Table 2.2: Ways of setting hyperparameter values.

All of these methods check that the new hyperparameter setting is valid, i.e. if falls within the allowed range or options and satisfies the conditional expressions.

```
# Change a single hyperparameter
subset$values$minsplit = 10
```

```
3
4  # Change all hyperparameter values
5  subset$values = list(
6    cp = 0.05,
7    keep_model = TRUE,
8    minsplit = 10,
9    xval = 5)
10
11 # Change only the given values
12 subset$set_values(minsplit = 10)
```

We do not recommend using the second method as it clears all meta information (e.g. initial values; more on this here: paradox). We recommend the `$set_values()` method.

While there are different methods for changing parameter values, we recommend to use the `$set_values()` method.

```
1  learner_rpart$param_set$set_values(minsplit = 10)
2  learner_rpart$param_set$values
```

```
$cp
[1] 0.05

$keep_model
[1] TRUE

$minsplit
[1] 10

$xval
[1] 5
```

The `lrn()` function also accepts additional arguments to set hyperparameters when constructing it:

```
1  learner_rpart = lrn("regr.rpart", minsplit = 10)
2  learner_rpart$param_set$values
```

```
$xval
[1] 0

$minsplit
[1] 10
```

```
1  learner_rpart$train(task_mtcars, splits$train)
2  learner_rpart$model
```

```
n= 21

node), split, n, deviance, yval
      * denotes terminal node
```

```
1) root 21 617.386700 20.33333
  2) disp>=101.55 18 167.562800 18.46111
    4) cyl>=7 9  30.075560 16.07778 *
    5) cyl< 7 9  35.242220 20.84444 *
  3) disp< 101.55 3   8.166667 31.56667 *
```

With the changed hyperparameters, we have a more complex (and more reasonable) model.

> **Note**
>
> Details on the hyperparameters of our `rpart` learner can be found in the documentation of `rpart::rpart.control()`. Hyperparameters in general are discussed in more detail in the section on Hyperparameter Tuning.

2.2.2 Predicting

After the model has been created, we can use it to make predictions. We can give the test partition to the `$predict()` function with the second argument being the testing set (`splits$test`) defined in `splits` by `partition()` earlier in this chapter. This `$predict()` function takes the `row_ids` given by `splits$test` and returns predictions for those `row_ids` in the given task (first argument) and the model stored in the learner (`learner_rpart`).

```
predictions = learner_rpart$predict(task_mtcars, splits$test)
predictions
```

```
<PredictionRegr> for 11 observations:
    row_ids truth response
          9  22.8 20.84444
         10  19.2 20.84444
         21  21.5 20.84444
---
         16  10.4 16.07778
         18  32.4 31.56667
         26  27.3 31.56667
```

The `$predict()` method returns a `Prediction` object, in this case a `PredictionRegr` for predicting a numeric quantity. The "truth" column contains the ground truth data, which was not given to the model to get a prediction. The "response" column contains the value predicted by the model, allowing for easy comparison with the ground truth data.

The `Prediction` object can easily be converted into a `data.table` or `data.frame` using the appropriate function:

```
as.data.table(predictions)
```

```
    row_ids truth response
 1:       9  22.8 20.84444
 2:      10  19.2 20.84444
 3:      21  21.5 20.84444
 4:      27  26.0 20.84444
```

```
 5:       7  14.3 16.07778
 6:      12  16.4 16.07778
 7:      14  15.2 16.07778
 8:      15  10.4 16.07778
 9:      16  10.4 16.07778
10:      18  32.4 31.56667
11:      26  27.3 31.56667
```

We can also use separate data to make predictions, which can be part of a separate task or simply a separate `data.table` or `data.frame`:

```
mtcars_new = data.table(cyl = c(5, 6),
                        disp = c(100, 120),
                        hp = c(100, 150),
                        drat = c(4, 3.9),
                        wt = c(3.8, 4.1),
                        qsec = c(18, 19.5),
                        vs = c(1, 0),
                        am = c(1, 1),
                        gear = c(6, 4),
                        carb = c(3, 5))
mtcars_new
```

```
   cyl disp  hp drat  wt qsec vs am gear carb
1:   5  100 100  4.0 3.8 18.0  1  1    6    3
2:   6  120 150  3.9 4.1 19.5  0  1    4    5
```

The learner does not need to know more meta-data about this data to make predictions, as this was given when training the model. We can use the `$predict_newdata()` method to make predictions for our separate dataset:

```
predictions = learner_rpart$predict_newdata(mtcars_new)
predictions
```

```
<PredictionRegr> for 2 observations:
 row_ids truth response
       1    NA 31.56667
       2    NA 20.84444
```

Note that the "truth" column is now `NA`, as we did not give the ground truth data.

We can also access the predictions directly:

```
predictions$response
```

```
[1] 31.56667 20.84444
```

Similar to plotting tasks, `mlr3viz` provides an `autoplot()` method for `Prediction` objects.

```
library("mlr3viz")
predictions = learner_rpart$predict(task_mtcars, splits$test)
autoplot(predictions)
```

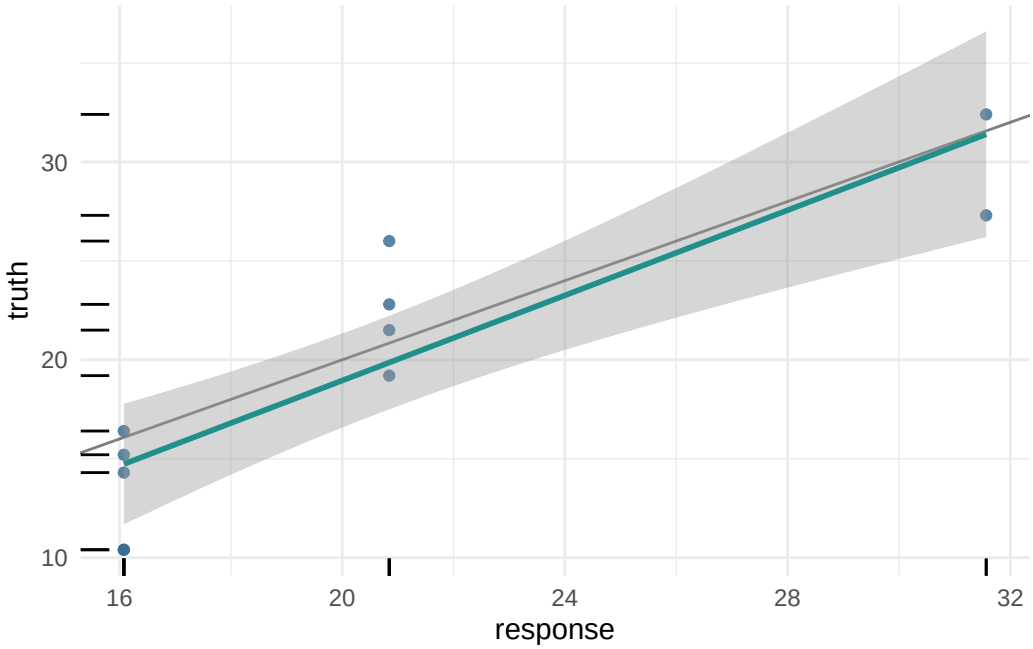


Figure 2.4: Comparing predicted and ground truth values for the mtcars dataset.

2.2.3 Changing the Prediction Type

Regression learners default to predicting a numeric quantity. However, many regression models can also give you bounds on the prediction by providing the standard error. To predict these standard errors, the `predict_type` field of a `LearnerRegr` must be changed from "response" (the default) to "se" before training. The `rpart` learner we used above does not support predicting standard errors, so we use the `lm` model from the `mlr3learners` package instead:

```
library(mlr3learners)

learner_lm = lrn("regr.lm")
learner_lm$predict_type = "se"
learner_lm$train(task_mtcars, splits$train)
predictions = learner_lm$predict(task_mtcars, splits$test)
predictions
```

```
<PredictionRegr> for 11 observations:
    row_ids truth response        se
          9  22.8 26.35691 1.4217357
         10  19.2 21.30664 0.9926847
         21  21.5 26.44570 1.2694455
---
         16  10.4 15.11714 2.1783078
         18  32.4 26.62327 1.1895495
         26  27.3 26.62199 1.1888914
```

> **Tip**
>
> Section 2.7.1 shows how to list learners that support the standard error prediction type.

The prediction object now contains the standard error for the predictions.

```
predictions
```

```
<PredictionRegr> for 11 observations:
    row_ids truth response        se
          9  22.8 26.35691 1.4217357
         10  19.2 21.30664 0.9926847
         21  21.5 26.44570 1.2694455
---
         16  10.4 15.11714 2.1783078
         18  32.4 26.62327 1.1895495
         26  27.3 26.62199 1.1888914
```

2.3 Evaluation

An important step of modeling is evaluating the performance of the trained model. We have seen how to inspect the model and plot its predictions above, but a more rigorous way that allows to compare different types of models more easily is to compute a performance measure. `mlr3` offers many performance measures, which can be created with the `msr()` function. Measures are stored in the dictionary `mlr_measures`, and a measure has to be supported by `mlr3` to be used, just like learners. For example, we can list all measures that are available for regression tasks:

```
mlr_measures$keys("regr")
```

```
 [1] "regr.bias"  "regr.ktau"  "regr.mae"   "regr.mape"  "regr.maxae"
 [6] "regr.medae" "regr.medse" "regr.mse"   "regr.msle"  "regr.pbias"
[11] "regr.rae"   "regr.rmse"  "regr.rmsle" "regr.rrse"  "regr.rse"
[16] "regr.rsq"   "regr.sae"   "regr.smape" "regr.srho"  "regr.sse"
```

For example, `regr.mape` is the mean absolute percent error, `regr.rmse` is the root of the mean squared error, and `regr.sse` is the sum of squared errors. The documentation for each measure, which contains its formula and more details, is available through the `$help()` function of the measure object.

Measure objects can be created with a single performance measure (`msr()`) or multiple (`msrs()`):

```
measure = msr("regr.rmse")
measures = msrs(c("regr.rmse", "regr.sse"))
```

At the core of all performance measures is a quantification of the difference between the predicted value and the ground truth value (except for unsupervised tasks, see Section 2.6). This means that in order to assess performance, we usually need the ground truth data – observations for which we do not know the true value cannot be used to assess the quality of the predictions of a model. This is why we make predictions on the data the model did not use during training (the test set).

As we have seen above, `mlr3`'s `Prediction` objects contain both predictions and ground truth. The `Measure` objects define how prediction and ground truth are compared, and how differences between them are quantified. We choose root mean squared error (`regr.rmse`) as our performance measure for this example. Once the measure is created, we can pass it to the `$score()` method of the `Prediction` object to quantify the predictive performance of our model.

```
measure = msr("regr.rmse")
measure
```

```
<MeasureRegrSimple:regr.rmse>: Root Mean Squared Error
* Packages: mlr3, mlr3measures
* Range: [0, Inf]
* Minimize: TRUE
* Average: macro
* Parameters: list()
* Properties: -
* Predict type: response
```

```
predictions$score(measure)
```

```
regr.rmse
 3.328844
```

> **Note**
>
> `$score()` can be called without a measure; in this case the default measure for the type of task is used. Regression defaults to mean squared error (`regr.mse`).

It is possible to calculate multiple measures at the same time by passing multiple measures to `$score()`. For example, to compute both root mean squared error `regr.rmse` and mean squared error `regr.mse`:

```
measures = msrs(c("regr.rmse", "regr.mse"))
predictions$score(measures)
```

```
regr.rmse  regr.mse
 3.328844 11.081203
```

`mlr3` also provides measures that do not quantify the quality of the predictions of a model, but other information we may be interested in, for example the time it took to train the model and make predictions:

```
measures = msrs(c("time_train", "time_predict"))
predictions$score(measures, learner = learner_lm)
```

```
time_train time_predict
     0.003        0.003
```

Note that these measures require a trained learner in addition to the predictions.

Some measures have hyperparameters themselves, for example `selected_features`. This measure gives information on the features the model used and is only supported by learners that have the "selected_features" property. It requires a task and a learner in addition to the predictions. The `lm` model does not support showing selected features; we use the `rpart` learner again and the full `mtcars` task.

```
task_mtcars = tsk("mtcars")
splits = partition(task_mtcars)
learner_rpart = lrn("regr.rpart", minsplit = 10)

learner_rpart$train(task_mtcars, splits$train)
predictions = learner_rpart$predict(task_mtcars, splits$test)
measure = msr("selected_features")
predictions$score(measure, task = task_mtcars, learner = learner_rpart)
```

```
selected_features
                2
```

The hyperparameter of the measure specifies whether the number of selected features should be normalized by the total number of features. The default is `FALSE`, giving the absolute number of features that, in this case, the trained decision tree uses. We can change the hyperparameter in the same way as for learners, for example:

```
measure = msr("selected_features", normalize = TRUE)
predictions$score(measure, task = task_mtcars, learner = learner_rpart)
```

```
selected_features
              0.2
```

We have now seen the basic building blocks of `mlr3` – creating and partitioning a task, instantiating a learner and setting its hyperparameters, training a model and inspecting it, making predictions, and assessing the quality of the model with a performance measure. So far, we have focused on regression, where we want to predict a numeric quantity. The rest of this chapter looks at other task types. The general procedure is the same, but some details are different.

2.4 Classification

Classification predicts a discrete, categorical target instead of the continuous numeric quantity for regression. The models that learn to classify data are different from regression models, and regression learners are not applicable for classification problems (although

for some learners, there are both regression and classification versions). `mlr3` distinguishes between the different tasks and learner types through different R6 classes and different prefixes – regression learners and measures start with `regr.`, whereas classification learners and measures start with `classif.`.

2.4.1 Classification Tasks

The `mlr_tasks` dictionary that comes with `mlr3` contains several classification tasks (`TaskClassif`). We can show only the classification tasks by converting the dictionary to a `data.table` and filtering on the `task_type`:

```
as.data.table(mlr_tasks)[task_type == "classif"]
```

```
                key                                      label task_type nrow
 1:   breast_cancer                  Wisconsin Breast Cancer   classif  683
 2:   german_credit                             German Credit   classif 1000
 3:           ilpd                Indian Liver Patient Data   classif  583
 4:           iris                              Iris Flowers   classif  150
 5:      optdigits Optical Recognition of Handwritten Digits   classif 5620
 6:       penguins                           Palmer Penguins   classif  344
 7: penguins_simple               Simplified Palmer Penguins   classif  333
 8:           pima                    Pima Indian Diabetes   classif  768
 9:          sonar                   Sonar: Mines vs. Rocks   classif  208
10:           spam                       HP Spam Detection   classif 4601
11:        titanic                                 Titanic   classif 1309
12:           wine                           Wine Regions   classif  178
13:            zoo                            Zoo Animals   classif  101
9 variables not shown: [ncol, properties, lgl, int, dbl, chr, fct, ord, pxc]
```

We will use the `penguins` dataset as a running example:

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

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

Just like for regression tasks, printing it gives an overview of the task, including the number of observations and features, and their types.

The target variable, species, is of type factor and has the following three classes or levels:

```
unique(task_penguins$data(cols = "species"))
```

```
     species
1:    Adelie
2:    Gentoo
```

```
3: Chinstrap
```

Classification tasks (`TaskClassif`) can also be plotted using `autoplot()`. Apart from the "pairs" plot type that we show here, "target" and "duo" are available. We refer the interested reader to the documentation of `mlr3viz::autoplot.TaskClassif` for an explanation of the other options. To keep the plot readable, we select only the first two features of the dataset.

```
library("mlr3viz")

task_penguins_small = task_penguins$clone()
# select the first feature, otherwise the individual plots are too small
task_penguins_small$select(head(task_penguins_small$feature_names, 1))
autoplot(task_penguins_small, type = "pairs")
```

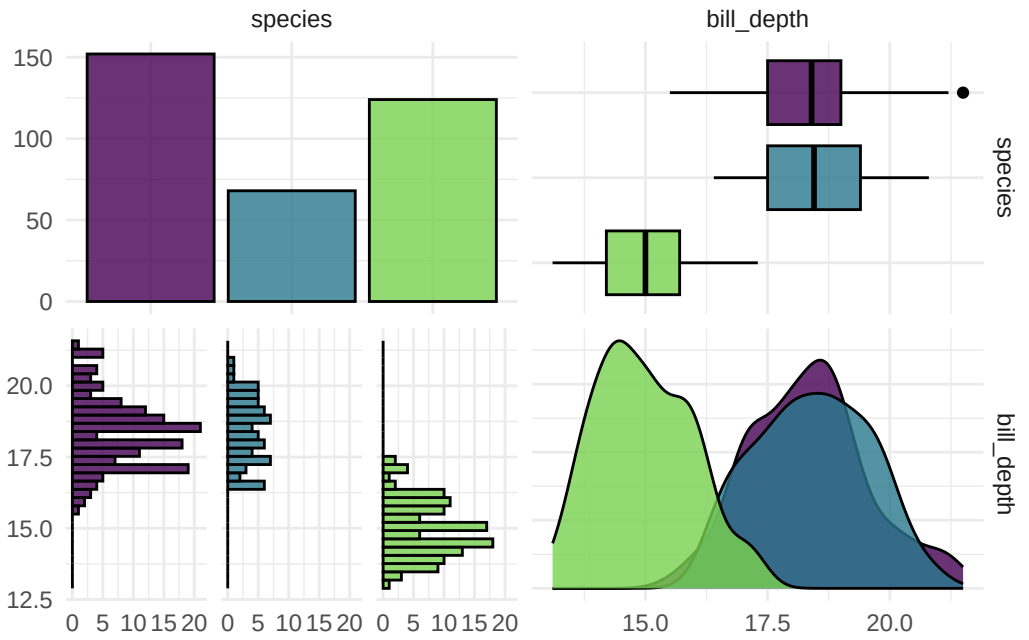


Figure 2.5: Overview of part of the penguins dataset.

2.4.2 Classification Learners

Classification learners (`LearnerClassif`) are a different R6 class than regression learners (`LearnerRegr`), but also inherit from the base class `Learner`. We can instantiate a classification learner in the same way as a regression learner, by retrieving it from the `mlr_learners` dictionary using `lrn()`. Note the "`classif.`" prefix to denote that we want a learner that classifies observations:

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

```
<LearnerClassifRpart:classif.rpart>: Classification Tree
```

```
* Model: -
* Parameters: xval=0
* Packages: mlr3, rpart
* Predict Types:  [response], prob
* Feature Types: logical, integer, numeric, factor, ordered
* Properties: importance, missings, multiclass, selected_features,
  twoclass, weights
```

Just like regression learners, classification learners have hyperparameters we can set to change their behavior, and printing the learner object gives some basic information about it. Training a model and making predictions works in the same way as for regression:

```
splits = partition(task_penguins)
learner_rpart$train(task_penguins, splits$train)
learner_rpart$model
```

```
n= 231

node), split, n, loss, yval, (yprob)
      * denotes terminal node

1) root 231 129 Adelie (0.441558442 0.199134199 0.359307359)
  2) flipper_length< 206.5 143  43 Adelie (0.699300699 0.293706294 0.006993007)
    4) bill_length< 44.2 101   3 Adelie (0.970297030 0.029702970 0.000000000) *
    5) bill_length>=44.2 42   3 Chinstrap (0.047619048 0.928571429 0.023809524) *
  3) flipper_length>=206.5 88   6 Gentoo (0.022727273 0.045454545 0.931818182)
    6) bill_depth>=17.2 7   3 Chinstrap (0.285714286 0.571428571 0.142857143) *
    7) bill_depth< 17.2 81   0 Gentoo (0.000000000 0.000000000 1.000000000) *
```

```
predictions = learner_rpart$predict(task_penguins, splits$test)
predictions
```

```
<PredictionClassif> for 113 observations:
    row_ids     truth  response
          2    Adelie    Adelie
          3    Adelie    Adelie
         10    Adelie    Adelie
---
        332 Chinstrap Chinstrap
        335 Chinstrap Chinstrap
        341 Chinstrap    Adelie
```

Just like predictions of regression models, we can plot classification predictions with `autoplot()`:

```
library("mlr3viz")
autoplot(predictions)
```

Figure 2.6: Comparing predicted and ground truth values for the penguins dataset.

2.4.2.1 Changing the Prediction Type

Classification problems support two types of predictions: the default "response", i.e. the class label, and "prob", which gives the probability for each class label. Not all learners support predicting probabilities.

The prediction type for a learner can be changed by setting `$predict_type`. After retraining the learner, all predictions have class probabilities (one for each class) in addition to the response, which is the class with the highest probability:

```
learner_rpart$predict_type = "prob"
learner_rpart$train(task_penguins, splits$train)
predictions = learner_rpart$predict(task_penguins, splits$test)
predictions
```

```
<PredictionClassif> for 113 observations:
    row_ids     truth  response prob.Adelie prob.Chinstrap prob.Gentoo
          2    Adelie    Adelie  0.97029703     0.02970297  0.00000000
          3    Adelie    Adelie  0.97029703     0.02970297  0.00000000
         10    Adelie    Adelie  0.97029703     0.02970297  0.00000000
---
        332 Chinstrap Chinstrap  0.04761905     0.92857143  0.02380952
        335 Chinstrap Chinstrap  0.04761905     0.92857143  0.02380952
        341 Chinstrap    Adelie  0.97029703     0.02970297  0.00000000
```

More information on how the probabilities are used to determine the predicted label and how to change this in Section 2.4.5.

> **Tip**
>
> Section 2.7.1 shows how to list learners that support the probability prediction type.

2.4.3 Classification Evaluation

Evaluation measures for classification problems that are supported by `mlr3` can be found in the `mlr_measures` dictionary:

```
mlr_measures$keys("classif")
```

```
 [1] "classif.acc"          "classif.auc"          "classif.bacc"
 [4] "classif.bbrier"       "classif.ce"           "classif.costs"
 [7] "classif.dor"          "classif.fbeta"        "classif.fdr"
[10] "classif.fn"           "classif.fnr"          "classif.fomr"
[13] "classif.fp"           "classif.fpr"          "classif.logloss"
[16] "classif.mauc_au1p"    "classif.mauc_au1u"    "classif.mauc_aunp"
[19] "classif.mauc_aunu"    "classif.mbrier"       "classif.mcc"
[22] "classif.npv"          "classif.ppv"          "classif.prauc"
[25] "classif.precision"    "classif.recall"       "classif.sensitivity"
[28] "classif.specificity" "classif.tn"           "classif.tnr"
[31] "classif.tp"           "classif.tpr"          "debug_classif"
```

For example, `classif.auc` is the area under the receiver operator characteristic (ROC) curve (see Section 3.4), `classif.acc` is the accuracy, and `classif.logloss` is the logarithmic loss. The documentation for each measure, which contains its formula and more details, is available through the `$help()` function of the measure object.

Some of these measures require the predictition type to be "prob" (e.g. `classif.auc`). As the default is "response", using those measures requires to change the prediction type, as shown above. You can check what prediction type a measure requires by looking at `$predict_type`.

```
measure = msr("classif.acc")
measure$predict_type
```

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

Once we have created a classification measure, we can give it to the `$score()` method to compute its value for a given `PredictionClassif` object:

```
predictions$score(measure)
```

```
classif.acc
  0.9557522
```

2.4.3.1 Confusion Matrix

A popular way to show the quality of prediction of a classification model is a confusion matrix. It gives a quick overview of what observations are misclassified, and how they are misclassified. The rows in a confusion matrix are the predicted class and the columns are

the true class. All off-diagonal entries are incorrectly classified observations, and all diagonal entries are correctly classified. More information can be found on Wikipedia[1].

`mlr3` supports confusion matrices through the `$confusion` property of the `PredictionClassif` object:

```
predictions$confusion
```

```
           truth
response    Adelie Chinstrap Gentoo
  Adelie        49         3      0
  Chinstrap      1        19      1
  Gentoo         0         0     40
```

In this case, our classifier does fairly well classifying the penguins.

2.4.4 Binary Classification and Positive Classes

Classification problems with a two-class target are called binary classification tasks. Binary Classification is special in the sense that one of these classes is denoted *positive* and the other one *negative*. You can specify the *positive class* for a classification task object during task creation. If not explicitly set during construction, the positive class defaults to the first level of the target feature.

```
# during construction
data("Sonar", package = "mlbench")
task_sonar = as_task_classif(Sonar, target = "Class", positive = "R")

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

2.4.5 Thresholding

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

Training a classifier on this data overpredicts the majority class, i.e. the more prevalent class is more likely to be predicted for any given observation.

```
task_credit = tsk("german_credit")
splits = partition(task_credit)
learner = lrn("classif.rpart", predict_type = "prob")
learner$train(task_credit)
predictions = learner$predict(task_credit)
predictions$confusion
```

[1] https://en.wikipedia.org/wiki/Confusion_matrix

```
        truth
response good bad
    good  627 130
    bad    73 170
```

```
autoplot(predictions)
```

Figure 2.7: Comparing predicted and ground truth values for the german_credit dataset.

Changing the prediction threshold allows to address this without having to adjust the hyperparameters of the learner or retrain the model.

```
predictions$set_threshold(0.7)
predictions$confusion
```

```
        truth
response good bad
    good  596 116
    bad   104 184
```

```
autoplot(predictions)
```

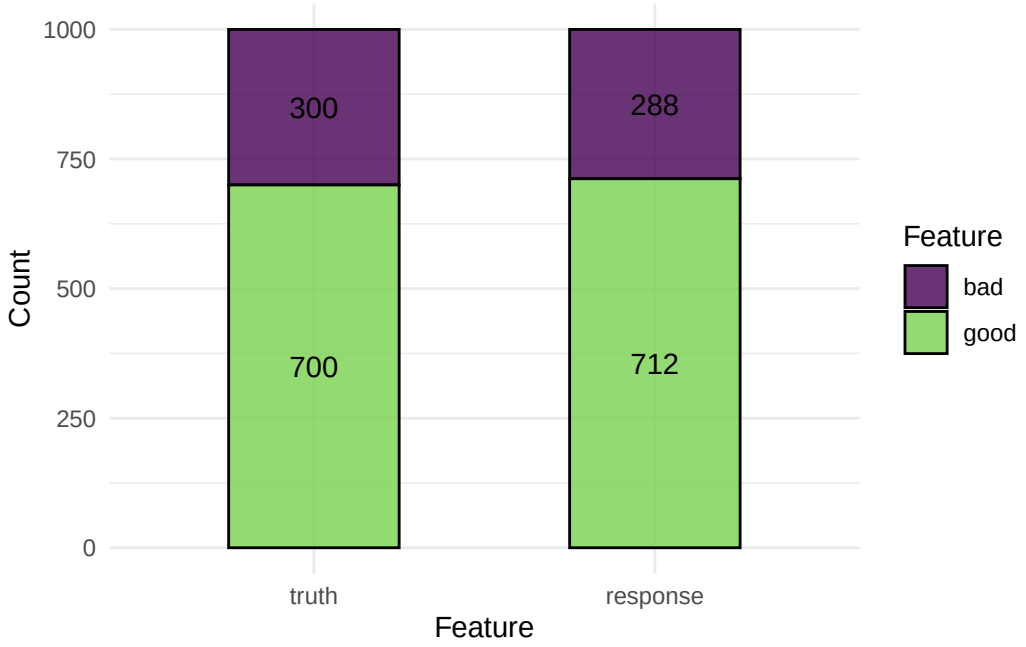


Figure 2.8: Comparing predicted and ground truth values for the german_credit dataset with adjusted threshold.

> **Tip**
>
> Thresholds can be tuned automatically with respect to prediction performance with the `mlr3pipelines` package using `PipeOpTuneThreshold`. This is covered in Chapter 6.

Thresholding For Multiple Classes

For classification tasks with more than two classes you can also adjust the prediction threshold, which is 0.5 for each class by default. Thresholds work slightly differently with multiple classes:

- The probability for a data point is divided by each class threshold resulting in `n` ratios for `n` classes.
- The highest ratio is selected (ties are random by default).

Lowering the threshold for a class means that it is more likely to be predicted and raising it has the opposite effect. The `zoo` dataset illustrates this concept nicely. When trained normally some classes are not predicted at all:

```
task = tsk("zoo")
learner = lrn("classif.rpart", predict_type = "prob")
learner$train(task)
preds = learner$predict(task)

```

```
autoplot(preds)
```

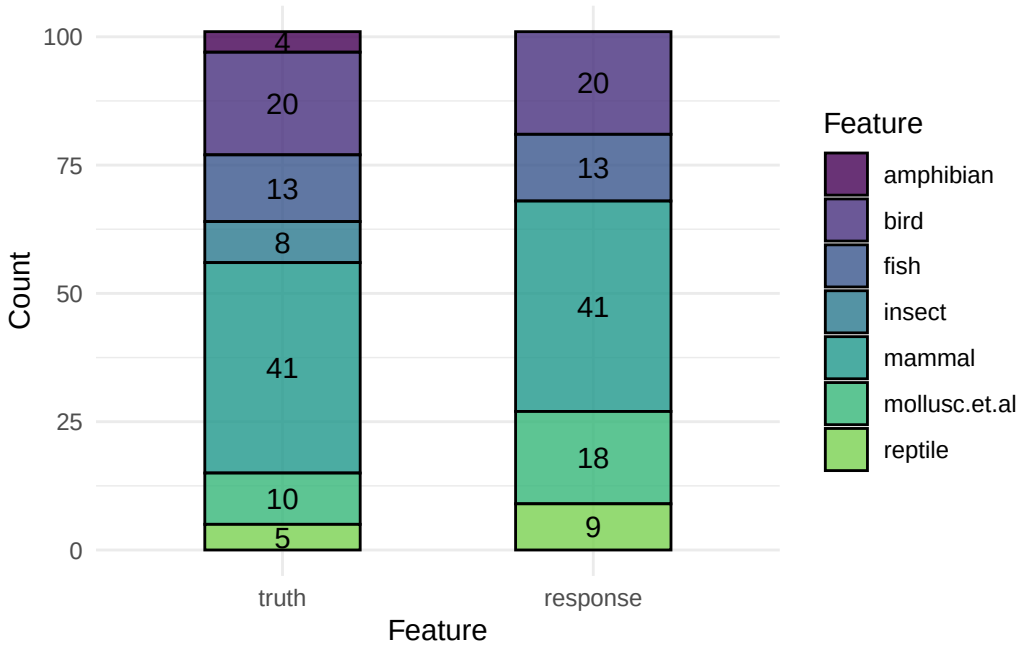


Figure 2.9: Comparing predicted and ground truth values for the zoo dataset.

The classes `amphibian` and `insect` are never predicted. On the other hand, the classes `mollusc` and `reptile` are predicted more often than they appear in the truth data. We can address this by lowering the threshold for `amphibian` and `insect`. `$set_threshold()` can be given a named list to set the threshold for all classes at once:

```
# c("mammal", "bird", "reptile", "fish", "amphibian", "insect", "mollusc.et.al")
new_thresh = c(0.5, 0.5, 0.5, 0.5, 0.4, 0.4, 0.5)
names(new_thresh) = task$class_names

autoplot(preds$set_threshold(new_thresh))
```

Figure 2.10: Comparing predicted and ground truth values for the zoo dataset with adjusted thresholds.

We can again see that adjusting the thresholds results in better predictive performance, without having to retrain a model.

2.5 Column Roles

> **!** **Advanced section**

We have seen that certain columns are designated as "targets" and "features" during task creation; `mlr3` calls this "roles". Target refers to the column(s) we want to predict and features are the predictors (also called co-variates or descriptors) for the target. Besides these two, there are other possible roles for columns. The roles affect the behavior of the task for different operations.

The `task_mtcars_small` task, for example, has the following column roles:

```
task_mtcars_small$col_roles[c("feature", "target")]
```

```
$feature
[1] "am"   "carb"

$target
[1] "mpg"
```

There are seven column roles. We can list all supported column roles by printing the names of the field `$col_roles`:

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

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

- `"feature"`: Regular feature used in the model fitting process.
- `"target"`: Target variable. Most tasks only accept a single target column.
- `"name"`: Row names / observation labels. To be used in plots. Can be queried with `$row_names`. Not more than a single column can be associated with this role.
- `"order"`: Data returned by `$data()` is ordered by this column (or these columns). Columns must be sortable with `order()`.
- `"group"`: During resampling, observations with the same value of the variable with role `"group"` are marked as "belonging together". For each resampling iteration, observations of the same group will be exclusively assigned to be either in the training set or in the test set. Not more than a single column can be associated with this role.
- `"stratum"`: Stratification variables. Multiple discrete columns may have this role.
- `"weight"`: Observation weights. Not more than one numeric column may have this role.

Columns can have multiple roles. It is also possible for a column to have no role at all, in which case they are ignored. This is, in fact, how `$select()` and `$filter()` operate: They unassign the `"feature"` (for columns) or `"use"` (for rows) role without modifying the data which is stored in an immutable backend:

```
task_mtcars_small$backend
```

```
<DataBackendDataTable> (32x13)
              model  mpg cyl disp  hp drat    wt  qsec vs am gear carb ..row_id
          Mazda RX4 21.0   6  160 110 3.90 2.620 16.46  0  1    4    4        1
      Mazda RX4 Wag 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4        2
         Datsun 710 22.8   4  108  93 3.85 2.320 18.61  1  1    4    1        3
     Hornet 4 Drive 21.4   6  258 110 3.08 3.215 19.44  1  0    3    1        4
  Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2        5
            Valiant 18.1   6  225 105 2.76 3.460 20.22  1  0    3    1        6
[...] (26 rows omitted)
```

There are two main ways to manipulate the col roles of a `Task` directly:

1. Use the `Task` method `$set_col_roles()` (recommended).
2. Directly modify the field `$col_roles`, which is a named list of vectors of column names. Each vector in this list corresponds to a column role, and the column names contained in that vector have the corresponding role.

Just as `$select()`/`$filter()`, these are in-place operations, i.e. the task object itself is modified. To retain an unmodified version of a task, use `$clone()`.

```
new_task = task_mtcars_small$clone()
```

2.5.1 Feature Role Example

Changing the column or row roles, whether through `$select()`/`$filter()` or directly, does not change the underlying data, it just updates the view on it. In a previous example we filtered the `"cyl"` column out of the task. Because the underlying data are still there (and accessible through `$backend`), we can add the `"cyl"` column back into the task by setting its column role to `"feature"`.

```
task_mtcars_small$set_col_roles("cyl", roles = "feature")
task_mtcars_small$feature_names  # cyl is now a feature again
```

```
[1] "am"   "carb" "cyl"
```

```
task_mtcars_small$data()
```

```
    mpg am carb cyl
1: 21.0  1    4   6
2: 22.8  1    1   4
3: 21.4  0    1   6
```

2.5.2 Weights Role Example

In some cases you may wish to weigh data points (rows) differently. For example, if your classification task has severe class imbalance (where the minority class is the class you are more interested in predicting accurately), weighting the minority class rows more heavily may improve the model's performance on that class.

For this example we will work with the built-in `breast_cancer` dataset. There are many more instances of the benign tumor class than the malignant tumor class. We are interested in predicting the malignant tumor class accurately so we will weight these instances.

```
task_cancer = tsk("breast_cancer")
summary(task_cancer$data()$class)
```

```
malignant    benign
      239       444
```

```
cancer_data = task_cancer$data()
# adding a column where the weight is 2 when the class == "malignant", and 1 otherwise
cancer_data$weights = ifelse(cancer_data$class == "malignant", 2, 1)
task_cancer = as_task_classif(cancer_data, target = "class")
task_cancer$set_col_roles("weights", roles = "weight")
task_cancer$col_roles[c("feature", "target", "weight")]
```

```
$feature
[1] "bare_nuclei"     "bl_cromatin"     "cell_shape"      "cell_size"
[5] "cl_thickness"    "epith_c_size"    "marg_adhesion"   "mitoses"
[9] "normal_nucleoli"

$target
[1] "class"
```

```
$weight
[1] "weights"
```

2.6 Additional Task Types

In addition to regression and classification, `mlr3` supports more types of tasks:

- Clustering (`mlr3cluster::TaskClust` in package `mlr3cluster`): An unsupervised task to identify similar groups within the feature space.
- Survival (`mlr3proba::TaskSurv` in package `mlr3proba`): The target is the (right-censored) time to an event.
- Density (`mlr3proba::TaskDens` in package `mlr3proba`): An unsupervised task to estimate the undetectable underlying probability distribution, based on observed data (as a numeric vector or a one-column matrix-like object).

Other task types that are less common are described in Chapter 7.

2.7 Additional Learners

As mentioned above, `mlr3` supports many learners. They can be accessed through three packages: the `mlr3` package, the `mlr3learners` package, and the `mlr3extralearners` package.

The list of learners included in the `mlr3` package is deliberately small to avoid large sets of dependencies for this core package:

- Featureless classifier `classif.featureless`: Simple baseline classification learner. Predicts the label that is most frequent in the training set. It can be used as a "fallback learner" to make predictions if another, more sophisticated, learner fails for some reason.
- Featureless regressor `regr.featureless`: Simple baseline regression learner. Predicts the mean of the target values in the training set.
- CART decision tree learner `classif.rpart`: Tree learner from `rpart`.
- CART regression tree learner `regr.rpart`: Tree learner from `rpart`.

The `mlr3learners` package contains cherry-picked implementations of the most popular machine learning methods:

- Linear (`regr.lm`) and logistic (`classif.log_reg`) regression.
- Penalized Generalized Linear Models (`regr.glmnet`, `classif.glmnet`), possibly with built-in optimization of the penalization parameter (`regr.cv_glmnet`, `classif.cv_glmnet`).
- (Kernelized) k-Nearest Neighbors regression (`regr.kknn`) and classification (`classif.kknn`).
- Kriging / Gaussian Process Regression (`regr.km`).
- Linear (`classif.lda`) and Quadratic (`classif.qda`) Discriminant Analysis.
- Naïve Bayes Classification (`classif.naive_bayes`).
- Support-Vector machines (`regr.svm`, `classif.svm`).

- Gradient Boosting (`regr.xgboost`, `classif.xgboost`).
- Random Forests for regression and classification (`regr.ranger`, `classif.ranger`).

A complete list of supported learners across all `mlr3` packages is hosted on our website[2].

The dictionary `mlr_learners` contains the supported learners and changes as packages are loaded. At the time of writing, `mlr3` supports six learners, `mlr3learners` 21 learners, `mlr3extralearners` 88 learners, `mlr3proba` five learners, and `mlr3cluster` 19 learners.

2.7.1 Listing Learners

You can list all learners by converting the `mlr_learners` dictionary into a `data.table`:

```
as.data.table(mlr_learners)
```

```
                    key                          label task_type
  1:  classif.AdaBoostM1             Adaptive Boosting   classif
  2:         classif.C50              Tree-based Model   classif
  3:         classif.IBk             Nearest Neighbour   classif
  4:         classif.J48              Tree-based Model   classif
  5:        classif.JRip Propositional Rule Learner.   classif
 ---
136: surv.priority_lasso                Priority Lasso      surv
137:         surv.ranger                 Random Forest      surv
138:          surv.rfsrc                 Random Forest      surv
139:             surv.svm      Support Vector Machine      surv
140:        surv.xgboost             Gradient Boosting      surv
4 variables not shown: [feature_types, packages, properties, predict_types]
```

The resulting `data.table` contains a lot of meta-data that is useful for identifying learners that have particular properties. For example, we can list all learners that support regression problems:

```
as.data.table(mlr_learners)[task_type == "regr"]
```

```
             key                                       label task_type
 1:      regr.IBk                       K-nearest neighbour      regr
 2:  regr.M5Rules                      Rule-based Algorithm      regr
 3:    regr.abess Fast Best Subset Selection for Regression      regr
 4:     regr.bart        Bayesian Additive Regression Trees      regr
 5: regr.catboost                          Gradient Boosting      regr
---
35:    regr.rpart                             Regression Tree      regr
36:      regr.rsm                      Response Surface Model      regr
37:      regr.rvm                    Relevance Vector Machine      regr
38:      regr.svm                                        <NA>      regr
39:  regr.xgboost                                        <NA>      regr
4 variables not shown: [feature_types, packages, properties, predict_types]
```

We can check multiple conditions, to for example find all learners that support regression problems and can predict standard errors:

[2] https://mlr-org.com/learners.html

```
as.data.table(mlr_learners)[task_type == "regr" &
    sapply(predict_types, function(x) "se" %in% x)]
```

```
               key                                    label task_type
1:       regr.debug             Debug Learner for Regression      regr
2:       regr.earth Multivariate Adaptive Regression Splines      regr
3: regr.featureless           Featureless Regression Learner      regr
4:         regr.gam    Generalized Additive Regression Model      regr
5:         regr.glm            Generalized Linear Regression      regr
6:          regr.km                                    <NA>      regr
7:          regr.lm                                    <NA>      regr
8:         regr.mob       Model-based Recursive Partitioning      regr
9:      regr.ranger                                    <NA>      regr
4 variables not shown: [feature_types, packages, properties, predict_types]
```

Or we can list all learners that support classification problems and missing feature values:

```
as.data.table(mlr_learners)[task_type == "classif" &
    sapply(properties, function(x) "missings" %in% x)]
```

```
                       key                              label task_type
 1:              classif.C50                 Tree-based Model   classif
 2:              classif.J48                 Tree-based Model   classif
 3:             classif.PART                 Tree-based Model   classif
 4:         classif.catboost                Gradient Boosting   classif
 5:            classif.debug Debug Learner for Classification   classif
 6:      classif.featureless Featureless Classification Learner   classif
 7:              classif.gbm                Gradient Boosting   classif
 8: classif.imbalanced_rfsrc          Imbalanced Random Forest   classif
 9:         classif.lightgbm                Gradient Boosting   classif
10:            classif.rfsrc                    Random Forest   classif
11:            classif.rpart               Classification Tree   classif
12:          classif.xgboost                              <NA>   classif
4 variables not shown: [feature_types, packages, properties, predict_types]
```

2.8 Exercises

1. Using the Sonar dataset, measure the classification error (`classif.ce`) of a classification tree model (`classif.rpart`) trained with default hyperparameters on 80% of the data and tested on the remaining 20%.
2. Give the true positive, false positive, true negative, and false negative rates of the predictions made by the model in exercise 1.
3. Change the threshold of the model from exercise 1 such that the false positive rate is lower than the false negative rate. Give a reason why you might do this.

3

Evaluation, Resampling and Benchmarking

Giuseppe Casalicchio
Ludwig-Maximilians-Universität München and Munich Center for Machine Learning (MCML), and Essential Data Science Training GmbH

Lukas Burk
Ludwig-Maximilians-Universität München, and Leibniz Institute for Prevention Research and Epidemiology - BIPS, and Munich Center for Machine Learning (MCML)

In supervised machine learning, a model which is deployed in practice is expected to generalize well to new, unseen data. Accurate estimation of this so-called generalization performance is crucial for many aspects of machine learning application and research — whether we want to fairly compare a novel algorithm with established ones or to find the best algorithm for a particular task after tuning — we always rely on this performance estimate. Hence, performance estimation is a fundamental concept used for model selection, model comparison, and hyperparameter tuning (which will be discussed in depth in Chapter 4) in supervised machine learning. To properly assess the generalization performance of a model, we must first decide on a performance measure that is appropriate for our given task and evaluation goal. A performance measure typically computes a numeric score indicating, e.g., how well the model predictions match the ground truth. However, it may also reflect other qualities such as the time for training a model. An overview of some common performance measures implemented in `mlr3`, including a short description and a basic mathematical definition, can be found by following the link provided in the overview table under *Measures overview* in Appendix Appendix D.

Once we have decided on a performance measure, the next step is to adopt a strategy that defines how to use the available data to estimate the generalization performance. Unfortunately, using the same data to train and test a model is a bad strategy as it would lead to an overly optimistic performance estimate. For example, an overfitted model may perfectly fit the data on which it was trained, but may not generalize well to new data. Assessing its performance using the same data it was trained would misleadingly suggest a well-performing model. It is therefore common practice to test a model on independent data not used to train a model. However, we typically train a deployed model on all available data, which leaves no data to assess its generalization performance. To address this issue, existing performance estimation strategies withhold a subset of the available data for evaluation purposes. This so-called test set serves as unseen data and is used to estimate the generalization performance.

A common simple strategy is the holdout method, which randomly partitions the data into a single training and test set using a pre-defined splitting ratio. The training set is used to create an intermediate model, whose sole purpose is to estimate the performance using the test set. This performance estimate is then used as a proxy for the performance of the final model trained on all available data and deployed in practice. Ideally, the training set

should be as large as all available data so that the intermediate model represents the final model well. If the training data is much smaller, the intermediate model learns less complex relationships compared to the final model, resulting in a pessimistically biased performance estimate. On the other hand, we also want as much test data as possible to reliably estimate the generalization performance. However, both goals are not possible if we have only access to a limited amount of data.

To address this issue, resampling strategies (see Section 3.2) repeatedly split all available data into multiple training and test sets, with one repetition corresponding to what is called a resampling iteration in `mlr3`. An intermediate model is then trained on each training set and the remaining test set is used to measure the performance in each resampling iteration. The generalization performance is finally estimated by the averaged performance over multiple resampling iterations (see Figure 3.1 for an illustration). Resampling methods allow using more data points for testing, while keeping the training sets as large as possible. Specifically, repeating the data splitting process allows using all available data points to assess the performance, as each data point can be ensured to be part of the test set in at least one resampling iteration. A higher number of resampling iterations can reduce the variance and result in a more reliable performance estimate. It also reduces the risk of the performance estimate being strongly affected by an unlucky split that does not reflect the original data distribution well, which is a known issue of the holdout method. However, since resampling strategies create multiple intermediate models trained on different parts of the available data and average their performance, they evaluate the performance of the learning algorithm that induced these models, rather than the performance of the final model which is deployed in practice. It is therefore important to train the intermediate models on nearly all data points from the same distribution so that the intermediate models and the final model are similar. If we only have access to a limited amount of data, the best we can do is to use the performance of the learning algorithm as a proxy for the performance of the final model. In Section 3.2, we will learn how to estimate the generalization performance of a `Learner` using the `mlr3` package.

3.1 Quick Start

In the previous chapter, we have applied the holdout method by manually partitioning the data contained in a `Task` object into a single training set (to train the model) and a single test set (to estimate the generalization performance). As a quick start into resampling and benchmarking with the `mlr3` package, we show a short example of how to do this with the `resample()` and `benchmark()` convenience functions. Specifically, we show how to estimate the generalization performance of a learner on a given task by the holdout method using `resample()` and how to use `benchmark()` to compare two learners on a task.

We first define the corresponding `Task` and `Learner` objects used throughout this chapter as follows:

```
task = tsk("penguins")
learner = lrn("classif.rpart", predict_type = "prob")
```

The next obvious step is to select a suitable performance measure, which can be done as explained in Section 2.3 using the `mlr_measures` dictionary. Passing the dictionary to the `as.data.table` function provides an overview of implemented measures with additional

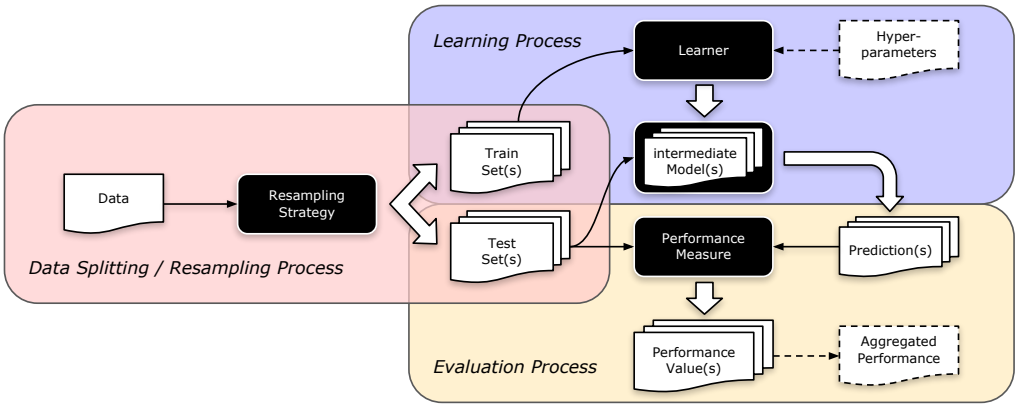


Figure 3.1: A general abstraction of the performance estimation process: The available data is (repeatedly) split into (a set of) training data and test data (data splitting / resampling process). The learner is applied to each training data and produces intermediate models (learning process). Each intermediate model along with its associated test data produces predictions. The performance measure compares these predictions with the associated actual target values from each test data and computes a performance value for each test data. All performance values are aggregated into a scalar value to estimate the generalization performance (evaluation process).

information from which we can select a suitable performance measure, which we print here in two parts for compactness:

```
msr_tbl = as.data.table(mlr_measures)
msr_tbl[1:5, .(key, label, task_type)]
```

```
            key                          label task_type
1:          aic   Akaike Information Criterion      <NA>
2:          bic Bayesian Information Criterion      <NA>
3:  classif.acc        Classification Accuracy   classif
4:  classif.auc       Area Under the ROC Curve   classif
5: classif.bacc              Balanced Accuracy   classif
```

```
msr_tbl[1:5, .(key, packages, predict_type, task_properties)]
```

```
            key          packages predict_type task_properties
1:          aic              mlr3         <NA>
2:          bic              mlr3         <NA>
3:  classif.acc mlr3,mlr3measures     response
4:  classif.auc mlr3,mlr3measures         prob        twoclass
5: classif.bacc mlr3,mlr3measures     response
```

Depending on our task at hand, we will look for a measure that fits our `"task_type"` (`"classif"` for `penguins`) and `"task_properties"`. The latter is important since measures like AUC `"classif.auc"` are only defined for binary tasks, which is indicated by `"twoclass"` in the `"task_properties"` column — multiclass-generalizations are available, but need to be selected explicitly. Similarly, some measures require the learner to predict

probabilities, while others require class predictions. In our learner above, we have already selected `predict_type = "prob"`, which is often required for measures that are not defined on class labels, such as the aforementioned AUC.

> **Tip**
>
> More information about a performance measure, including its mathematical definition, can be obtained using the `$help()` method of a `Measure` object, which opens the help page of the corresponding measure, e.g., `msr("classif.acc")$help()` provides all information about the classification accuracy.

The code example below shows how to apply holdout (specified using `rsmp("holdout")`) on the previously specified `mlr_tasks_penguins` task to estimate classification accuracy (using `msr("classif.acc")`) of the previously defined decision tree learner from the `rpart` package:

```
resampling = rsmp("holdout")
rr = resample(task = task, learner = learner, resampling = resampling)
rr$aggregate(msr("classif.acc"))
```

```
classif.acc
  0.9391304
```

The `benchmark()` function internally uses the `resample()` function to estimate the performance based on a resampling strategy. For illustration, we show a minimal code example that compares the classification accuracy of the decision tree against a featureless learner which always predicts the majority class:

```
lrns = c(learner, lrn("classif.featureless"))
d = benchmark_grid(task = task, learner = lrns, resampling = resampling)
bmr = benchmark(design = d)
acc = bmr$aggregate(msr("classif.acc"))
acc[, .(task_id, learner_id, classif.acc)]
```

```
    task_id          learner_id classif.acc
1: penguins       classif.rpart   0.9565217
2: penguins classif.featureless   0.4173913
```

Further details on resampling and benchmarking can be found in Section 3.2 and Section 3.3.

3.2 Resampling

Existing resampling strategies differ in how they partition the available data into training and test set, and a comprehensive overview can be found in Japkowicz and Shah (2011). For example, the k-fold cross-validation method randomly partitions the data into k subsets, called folds (see Figure 3.2). Then k models are trained on training data consisting of $k-1$ of the folds, with the remaining fold being used as test data exactly once in each of the k iterations. The k performance estimates resulting from each fold are then averaged to obtain

a more reliable performance estimate. This makes cross-validation a popular strategy, as each observation is guaranteed to be used in one of the test sets throughout the procedure, making efficient use of the available data for performance estimation. Several variations of cross-validation exist, including repeated k-fold cross-validation where the entire process illustrated in Figure 3.2 is repeated multiple times, and leave-one-out cross-validation where the test set in each fold consists of exactly one observation.

Other well-known resampling strategies include subsampling and bootstrapping. Subsampling — also known as repeated holdout — repeats the holdout method and creates multiple train-test splits, taking into account the ratio of observations to be included in the training sets. Bootstrapping creates training sets by randomly drawing observations from all available data with replacement. Some observations in the training sets may appear more than once, while the other observations that do not appear at all are used as test set. The choice of the resampling strategy usually depends on the specific task at hand and the goals of the performance assessment. Properties and pitfalls of different resampling techniques have been widely studied and discussed in the literature, see e.g., Bengio and Grandvalet (2003), Molinaro, Simon, and Pfeiffer (2005), Kim (2009), Bischl et al. (2012).

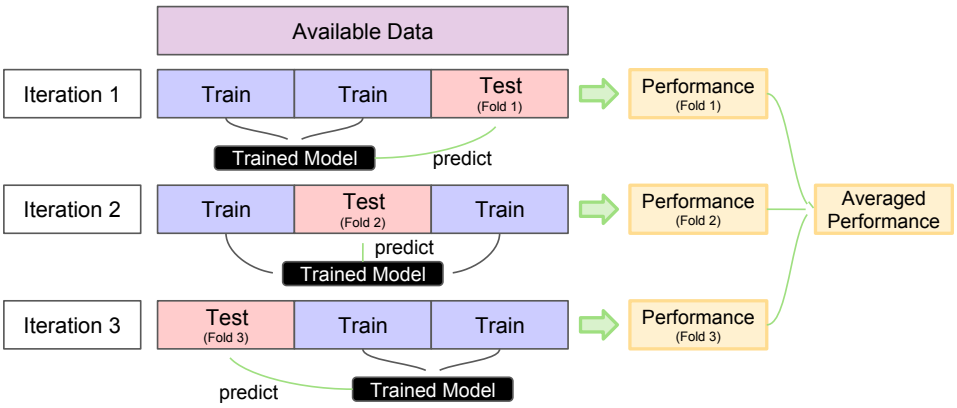


Figure 3.2: Illustration of a 3-fold cross-validation.

In `mlr3`, many resampling strategies have already been implemented so that users do not have to implement them from scratch, which can be tedious and error-prone. In this section, we cover how to use `mlr3` to

- query (Section 3.2.1) implemented resampling strategies,
- construct (Section 3.2.2) resampling objects for a selected resampling strategy,
- instantiate (Section 3.2.3) the train-test splits of a resampling object on a given task, and
- execute (Section 3.2.4) the selected resampling strategy on a learning algorithm to obtain resampling results.

3.2.1 Query

All implemented resampling strategies can be queried by looking at the `mlr_resamplings` dictionary (also listed in Appendix Appendix D). Passing the dictionary to the `as.data.table` function provides a more structured output with additional information:

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

```
          key                          label        params iters
1:   bootstrap                      Bootstrap ratio,repeats    30
2:      custom                   Custom Splits                  NA
3:   custom_cv Custom Split Cross-Validation                  NA
4:          cv              Cross-Validation         folds    10
5:     holdout                        Holdout         ratio     1
6:    insample           Insample Resampling                   1
7:         loo                  Leave-One-Out                  NA
8: repeated_cv     Repeated Cross-Validation folds,repeats   100
9: subsampling                   Subsampling ratio,repeats    30
```

For example, the column `params` shows the parameters of each resampling strategy (e.g., the train-test splitting `ratio` or the number of `repeats`) and the column `iters` shows the default value for the number of performed resampling iterations (i.e., the number of model fits).

3.2.2 Construction

Once we have decided on a resampling strategy, we have to construct a `Resampling` object via the function `rsmp()` which will define the resampling strategy we want to employ. For example, to construct a `Resampling` object for holdout, we use the value of the `key` column from the `mlr_resamplings` dictionary and pass it to the convenience function `rsmp()`:

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

```
<ResamplingHoldout>: Holdout
* Iterations: 1
* Instantiated: FALSE
* Parameters: ratio=0.6667
```

By default, the holdout method will use 2/3 of the data as training set and 1/3 as test set. We can adjust this by specifying the `ratio` parameter for holdout either during construction or by updating the `ratio` parameter afterwards. For example, we construct a `Resampling` object for holdout with a 80:20 split (see first line in the code below) then update to 50:50 (see second line in the code below):

```
resampling = rsmp("holdout", ratio = 0.8)
resampling$param_set$values = list(ratio = 0.5)
```

Holdout only estimates the generalization performance using a single test set. To obtain a more reliable performance estimate by making use of all available data, we may use other resampling strategies. For example, we could also set up a 10-fold cross-validation via

```
resampling = rsmp("cv", folds = 10)
```

By default, the `$is_instantiated` field of a `Resampling` object constructed as shown above is set to `FALSE`. This means that the resampling strategy is not yet applied to a task, i.e.,

the train-test splits are not contained in the `Resampling` object.

3.2.3 Instantiation

To generate the train-test splits for a given task, we need to instantiate a resampling strategy by calling the `$instantiate()` method of the previously constructed `Resampling` object on a `Task`. This will manifest a fixed partition and store the row indices for the training and test sets directly in the `Resampling` object. We can access these rows via the `$train_set()` and `$test_set()` methods:

```
resampling = rsmp("holdout", ratio = 0.8)
resampling$instantiate(task)
train_ids = resampling$train_set(1)
test_ids = resampling$test_set(1)
str(train_ids)
```

```
 int [1:275] 1 2 4 5 6 8 9 10 11 12 ...
```

```
str(test_ids)
```

```
 int [1:69] 3 7 23 33 35 39 41 49 50 55 ...
```

Instantiation is especially relevant is when the aim is to fairly compare multiple learners. Here, it is crucial to use the same train-test splits to obtain comparable results. That is, we need to ensure that all learners to be compared use the same training data to build a model and that they use the same test data to evaluate the model performance.

> **Tip**
>
> In Section 3.3, you will learn about the `ref ("benchmark()")` function, which automatically instantiates `Resampling` objects on all tasks to ensure a fair comparison by making use of the exact same training and test sets for learning and evaluating the fitted intermediate models.

3.2.4 Execution

Calling the function `resample()` on a task, learner, and constructed resampling object returns a `ResampleResult` object which contains all information needed to estimate the generalization performance. Specifically, the function will internally use the learner to train a model for each training set determined by the resampling strategy and store the model predictions of each test set. We can apply the `print` or `as.data.table` function to a `ResampleResult` object to obtain some basic information:

```
resampling = rsmp("cv", folds = 4)
rr = resample(task, learner, resampling)
print(rr)
```

```
<ResampleResult> with 4 resampling iterations
  task_id    learner_id resampling_id iteration warnings errors
 penguins classif.rpart            cv         1        0      0
```

```
 penguins classif.rpart            cv         2        0      0
 penguins classif.rpart            cv         3        0      0
 penguins classif.rpart            cv         4        0      0
```

```
as.data.table(rr)
```

```
               task                    learner         resampling iteration
1: <TaskClassif[51]> <LearnerClassifRpart[38]> <ResamplingCV[20]>         1
2: <TaskClassif[51]> <LearnerClassifRpart[38]> <ResamplingCV[20]>         2
3: <TaskClassif[51]> <LearnerClassifRpart[38]> <ResamplingCV[20]>         3
4: <TaskClassif[51]> <LearnerClassifRpart[38]> <ResamplingCV[20]>         4
1 variable not shown: [prediction]
```

Here, we used 4-fold cross-validation as resampling strategy. The resulting `ResampleResult` object (stored as `rr`) provides various methods to access the stored information. The two most relevant methods for performance assessment are `$score()` and `$aggregate()`.

In Section 2.3, we learned that `Prediction` objects contain both model predictions and ground truth values, which are used to calculate the performance measure using the `$score()` method. Similarly, we can use the `$score()` method of a `ResampleResult` object to calculate the performance measure for each resampling iteration separately. This means that the `$score()` method produces one value per resampling iteration that reflects the performance estimate of the intermediate model trained in the corresponding iteration. By default, `$score()` uses the test set in each resampling iteration to calculate the performance measure.

> **Tip**
>
> We are not limited to scoring predictions on the test set — if we set the argument `predict_sets = "train"` within the `$score()` method, we calculate the performance measure of each resampling iteration based on the training set instead of the test set.

In the code example below, we explicitly use the classification accuracy (`classif.acc`) as performance measure and pass it to the `$score()` method to obtain the estimated performance of each resampling iteration separately:

```
acc = rr$score(msr("classif.acc"))
acc[, .(iteration, classif.acc)]
```

```
   iteration classif.acc
1:         1   0.9069767
2:         2   0.9418605
3:         3   0.9418605
4:         4   0.9651163
```

> **Tip**
>
> If we do not explicitly pass a `Measure` object to the `$score()` method, the classification error (`classif.ce`) and the mean squared error (`regr.mse`) are used as defaults for classification and regression tasks respectively.

Similarly, we can pass `Measure` objects to the `$aggregate()` method to calculate an aggregated score across all resampling iterations. The type of aggregation is usually determined by the `Measure` object (see also the fields `$average` and `$aggregator` the in help page of `Measure` for more details). There are two approaches for aggregating scores across resampling iterations: The first is referred to as the macro average, which first calculates the measure in each resampling iteration separately, and then averages these scores across all iterations. The second approach is the micro average, which pools all predictions across resampling iterations into one `Prediction` object and computes the measure on this directly. The classification accuracy `msr("classif.acc")` uses the macro-average by default, but the micro-average can be computed as well by specifying the `average` argument:

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

```
classif.acc
  0.9389535
```

```
rr$aggregate(msr("classif.acc", average = "micro"))
```

```
classif.acc
  0.9389535
```

> **Tip**
>
> The classification accuracy compares the predicted class and the ground truth class of a single observation (point-wise loss) and calculates the proportion of correctly classified observations (average of point-wise loss). For performance measures that simply take the (unweighted) average of point-wise losses such as the classification accuracy, macro-averaging and micro-averaging will be equivalent unless the test sets in each resampling iteration have different sizes. For example, in the code example above, macro-averaging and micro-averaging yield the same classification accuracy because the `mlr_tasks_penguins` task (consisting of 344 observations) is split into 4 equally-sized test sets (consisting of 86 observations each) due to the 4-fold cross-validation. If we would use 5-fold cross-validation instead, macro-averaging and micro-averaging can lead to a (slightly) different performance estimate as the test sets can not have the exact same size:
>
> ```
> rr5 = resample(task, learner, rsmp("cv", folds = 5))
> rr5$aggregate(msr("classif.acc"))
> ```
>
> ```
> classif.acc
>   0.9504689
> ```
>
> ```
> rr5$aggregate(msr("classif.acc", average = "micro"))
> ```
>
> ```
> classif.acc
>   0.9505814
> ```
>
> For other performance measures that are not defined on observation level but rather on a set of observations such as the area under the ROC curve `msr("classif.auc")`, macro-averaging and micro-averaging will usually always lead to different values.

The aggregated score (as returned by `$aggregate()`) refers to the generalization performance of our selected learner on the given task estimated by the resampling strategy defined in the `Resampling` object. While we are usually interested in this aggregated score, it can be useful to look at the individual performance values of each resampling iteration (as returned by the `$score()` method) as well, e.g., to see if one (or more) of the iterations lead to very different performance results. Figure 3.3 visualizes the relationship between `$score()` and `$aggregate()` for a small example based on the `"penguins"` task.

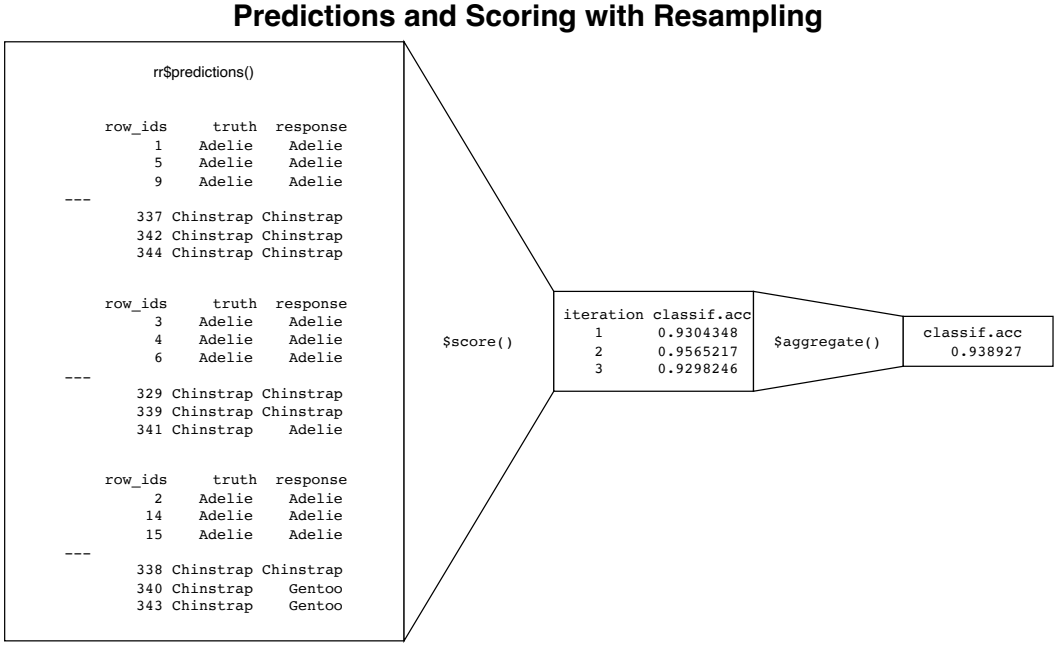


Figure 3.3: An example of the difference between `$score()` and `$aggregate()`: The former aggregates predictions to a single score within each resampling iteration, and the latter aggregates scores across all resampling folds

3.2.5 Inspect ResampleResult Objects

In this section, we show how to inspect some important fields and methods of a `ResampleResult` object. We first take a glimpse at what is actually contained in the object by converting it to a `data.table`:

```
rrdt = as.data.table(rr)
rrdt
```

```
                task                    learner         resampling iteration
1: <TaskClassif[51]> <LearnerClassifRpart[38]> <ResamplingCV[20]>         1
2: <TaskClassif[51]> <LearnerClassifRpart[38]> <ResamplingCV[20]>         2
3: <TaskClassif[51]> <LearnerClassifRpart[38]> <ResamplingCV[20]>         3
4: <TaskClassif[51]> <LearnerClassifRpart[38]> <ResamplingCV[20]>         4
1 variable not shown: [prediction]
```

We can see that the task, learner and resampling strategy which we previously passed to

the `resample()` function is stored in list columns of the `data.table`. In addition, we also have an integer column `iteration` that refers to the resampling iteration and another list column that contains the corresponding `Prediction` objects of each iteration. We can access the respective `prediction` column or directly use the `$predictions()` method of the `ResampleResult` object (without converting it to a `data.table` first) to obtain a list of `Prediction` objects of each resampling iteration:

```
rrdt$prediction
```

```
[[1]]
<PredictionClassif> for 86 observations:
    row_ids     truth  response prob.Adelie prob.Chinstrap prob.Gentoo
          2    Adelie    Adelie  0.97272727     0.02727273           0
          6    Adelie    Adelie  0.97272727     0.02727273           0
          8    Adelie    Adelie  0.97272727     0.02727273           0
---
        328 Chinstrap Chinstrap  0.03773585     0.96226415           0
        333 Chinstrap Chinstrap  0.03773585     0.96226415           0
        342 Chinstrap Chinstrap  0.03773585     0.96226415           0

[[2]]
<PredictionClassif> for 86 observations:
    row_ids     truth  response prob.Adelie prob.Chinstrap prob.Gentoo
         10    Adelie    Adelie  0.97413793     0.02586207  0.00000000
         11    Adelie    Adelie  0.97413793     0.02586207  0.00000000
         13    Adelie    Adelie  0.97413793     0.02586207  0.00000000
---
        325 Chinstrap Chinstrap  0.06818182     0.90909091  0.02272727
        331 Chinstrap    Adelie  0.97413793     0.02586207  0.00000000
        336 Chinstrap Chinstrap  0.06818182     0.90909091  0.02272727

[[3]]
<PredictionClassif> for 86 observations:
    row_ids     truth  response prob.Adelie prob.Chinstrap prob.Gentoo
          3    Adelie    Adelie  0.96491228     0.03508772   0.0000000
          5    Adelie    Adelie  0.96491228     0.03508772   0.0000000
          7    Adelie    Adelie  0.96491228     0.03508772   0.0000000
---
        330 Chinstrap Chinstrap  0.06382979     0.91489362   0.0212766
        338 Chinstrap Chinstrap  0.06382979     0.91489362   0.0212766
        340 Chinstrap    Gentoo  0.01030928     0.03092784   0.9587629

[[4]]
<PredictionClassif> for 86 observations:
    row_ids     truth  response prob.Adelie prob.Chinstrap prob.Gentoo
          1    Adelie    Adelie  0.96460177     0.03539823  0.00000000
          4    Adelie    Adelie  0.96460177     0.03539823  0.00000000
         12    Adelie    Adelie  0.96460177     0.03539823  0.00000000
---
        341 Chinstrap    Adelie  0.96460177     0.03539823  0.00000000
```

```
        343 Chinstrap Chinstrap  0.28571429     0.57142857  0.14285714
        344 Chinstrap Chinstrap  0.06521739     0.91304348  0.02173913
```

```
all.equal(rrdt$prediction, rr$predictions())
```

```
[1] TRUE
```

This allows to analyze the predictions of individual intermediate models from each resampling iteration and, e.g., to manually compute a macro-averaged performance estimate. Instead, we can use the `$prediction()` method to extract a single `Prediction` object that combines the predictions of each intermediate model arcoss all resampling iterations. The combined prediction object can be used to manually compute a micro-averaged performance estimate, for example:

```
pred = rr$prediction()
pred
```

```
<PredictionClassif> for 344 observations:
    row_ids     truth  response prob.Adelie prob.Chinstrap prob.Gentoo
          2    Adelie    Adelie  0.97272727     0.02727273  0.00000000
          6    Adelie    Adelie  0.97272727     0.02727273  0.00000000
          8    Adelie    Adelie  0.97272727     0.02727273  0.00000000
---
        341 Chinstrap    Adelie  0.96460177     0.03539823  0.00000000
        343 Chinstrap Chinstrap  0.28571429     0.57142857  0.14285714
        344 Chinstrap Chinstrap  0.06521739     0.91304348  0.02173913
```

```
pred$score(msr("classif.acc"))
```

```
classif.acc
  0.9389535
```

By default, the intermediate models produced at each resampling iteration are discarded after the prediction step to reduce memory consumption of the `ResampleResult` object and because only the predictions are required to calculate the performance measure. However, it can sometimes be useful to inspect, compare, or extract information from these intermediate models. To do so, we can configure the `resample()` function to keep the fitted intermediate models by setting the `store_models` argument to `TRUE`. Each model trained in a specific resampling iteration is then stored in the resulting `ResampleResult` object and can be accessed via `$learners[[i]]$model`, where `i` refers to the `i`-th resampling iteration:

```
rr = resample(task, learner, resampling, store_models = TRUE)
rr$learners[[1]]$model
```

```
n= 258

node), split, n, loss, yval, (yprob)
      * denotes terminal node

1) root 258 148 Adelie (0.42635659 0.21705426 0.35658915)
  2) flipper_length< 206.5 162  54 Adelie (0.66666667 0.32716049 0.00617284)
```

```
    4) bill_length< 43.35 111   5 Adelie (0.95495495 0.04504505 0.00000000) *
    5) bill_length>=43.35 51   3 Chinstrap (0.03921569 0.94117647 0.01960784) *
  3) flipper_length>=206.5 96   5 Gentoo (0.02083333 0.03125000 0.94791667) *
```

Here, we see the model output of a decision tree fitted by the `rpart` package. As models fitted by `rpart` provide information on how important features are, we can inspect if the importance varies across the resampling iterations:

```
lapply(rr$learners, function(x) x$model$variable.importance)
```

```
[[1]]
flipper_length    bill_length     bill_depth      body_mass         island
      96.74085       88.92280       70.09671       61.33462       53.31916

[[2]]
flipper_length    bill_length     bill_depth      body_mass         island
      98.11490       96.76265       75.80578       67.28828       61.40783

[[3]]
flipper_length    bill_length     bill_depth      body_mass         island
      96.68182       84.04033       69.72064       65.14820       46.16637

[[4]]
flipper_length      body_mass    bill_length     bill_depth         island
    105.123207      91.106307      89.932173      84.999767      84.304682
            year
      5.550757
```

Each resampling iteration involves a training step and a prediction step. Learner-specific error or warning messages may occur at each of these two steps. If the learner passed to the `resample()` function runs in an encapsulated framework that allows logging (see the `$encapsulate` field of a `Learner` object), all potential warning or error messages will be stored in the `$warnings` and `$errors` fields of the `ResampleResult` object.

3.2.6 Custom Resampling

> **!** **Advanced section**

Sometimes it is necessary to perform resampling with custom splits, e.g., to reproduce results reported in a study with pre-defined folds. A custom resampling strategy can be constructed using `rsmp("custom")`, where the row indices of the observations used for training and testing must be defined manually when instantiated in a task. In the example below, we construct a custom holdout resampling strategy by manually assigning row indices to the `$train` and `$test` fields.

```
resampling = rsmp("custom")
resampling$instantiate(task,
  train = list(c(1:50, 151:333)),
  test = list(51:150)
```

```
)
```

The resulting `Resampling` object can then be used like all other resampling strategies. To show that both sets contain the row indices we have defined, we can inspect the instantiated `Resampling` object:

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

```
 int [1:233] 1 2 3 4 5 6 7 8 9 10 ...
```

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

```
 int [1:100] 51 52 53 54 55 56 57 58 59 60 ...
```

The above is equivalent to a single custom train-test split analogous to the holdout strategy. A custom version of the cross-validation strategy can be constructed using `rsmp("custom_cv")`. The important difference is that we now have to specify either a custom `factor` variable (using the `f` argument of the `$instantiate()` method) or a `factor` column (using the `col` argument of the `$instantiate()` method) from the data to determine the folds.

In the example below, we instantiate a custom 4-fold cross-validation strategy using a `factor` variable called `folds` that contains 4 equally sized levels to define the 4 folds, each with one quarter of the total size of the `"penguin"` task:

```
custom_cv = rsmp("custom_cv")
folds = as.factor(rep(1:4, each = task$nrow/4))
custom_cv$instantiate(task, f = folds)
custom_cv
```

```
<ResamplingCustomCV>: Custom Split Cross-Validation
* Iterations: 4
* Instantiated: TRUE
* Parameters: list()
```

3.2.7 Resampling with Stratification and Grouping

> **!** **Advanced section**

In `mlr3`, we can assign a special role to a feature contained in the data by configuring the corresponding `$col_roles` field of a `Task`. The two relevant column roles that will affect behavior of a resampling strategy are `"group"` or `"stratum"`.

In some cases, it is desirable to keep observations together when the data is split into corresponding training and test sets, especially when a set of observations naturally belong to a group, e.g., when the data contains repeated measurements of individuals (longitudinal studies) or when dealing with spatial or temporal data. When observations belong to groups, we want to ensure that all observations of the same group belong to either the training set or the test set to prevent any potential leakage of information between training and testing sets. For example, in a longitudinal study, measurements of a person are usually taken at multiple

time points. Grouping ensures that the model is tested on data from each person that it has not seen during training, while maintaining the integrity of the person's measurements across different time points. In this context, the leave-one-out cross-validation strategy can be coarsened to the "leave-one-object-out" cross-validation strategy, where not only a single observation is left out, but all observations associated with a certain group (see Figure 3.4 for an illustration).

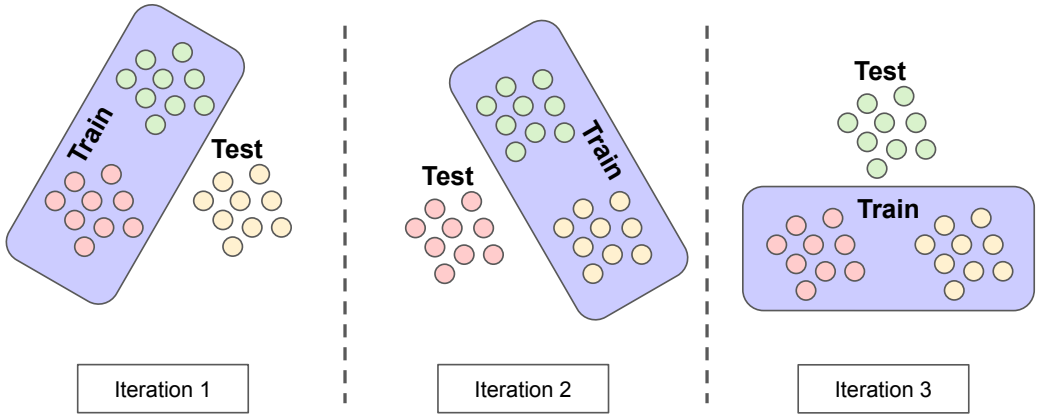


Figure 3.4: Illustration of the train-test splits of a leave-one-object-out cross-validation with 3 groups of observations (highlighted by different colors).

In `mlr3`, the column role `"group"` allows to specify the column in the data that defines the group structure of the observations (see also the help page of `Resampling` for more information on the column role `"group"`). The column role can be specified by assigning a feature to the `$col_roles$group` field which will then determine the group structure. The following code performs leave-one-object-out cross-validation using the feature `year` of the `mlr_tasks_penguins` task to determine the grouping. Since the feature `year` contains only three distinct values (i.e., 2007, 2008, and 2009), the corresponding test sets consist of observations from only one year:

```
task_grp = tsk("penguins")
task_grp$col_roles$group = "year"
r = rsmp("loo")
r$instantiate(task_grp)

table(task_grp$data(cols = "year"))
```

```
year
2007 2008 2009
 110  114  120
```

```
table(task_grp$data(rows = r$test_set(1), cols = "year"))
```

```
year
2007
 110
```

```
table(task_grp$data(rows = r$test_set(2), cols = "year"))
```

```
year
2008
 114
```

```
table(task_grp$data(rows = r$test_set(3), cols = "year"))
```

```
year
2009
 120
```

> **Tip**
>
> If there are many groups, say 100 groups, we can limit the number of resampling iterations using k-fold cross-validation (or any other resampling strategy with a previously definable number of resampling iterations) instead of performing leave-one-object-out cross-validation. In this case, each group is considered as a single observation, so that the division into training and test sets is done as determined by the resampling strategy

Another column role available in `mlr3` is `"stratum"`, which implements stratified sampling. Stratified sampling ensures that one or more discrete features within the training and test sets will have a similar distribution as in the original task containing all observations. This is especially useful when a discrete feature is highly imbalanced and we want to make sure that the distribution of that feature is similar in each resampling iteration. Stratification is commonly used for imbalanced classification tasks where the classes of the target feature are imbalanced (see Figure 3.5 for an illustration). Stratification by the target feature ensures that each intermediate model is fit on training data where the class distribution of the target is representative of the actual task. Otherwise it could happen that target classes are severely under- or over represented in individual resampling iterations, skewing the estimation of the generalization performance.

The `$col_roles$stratum` field of a `Task` can be set to one or multiple features (including the target in case of classification tasks). In case of multiple features, each combination of the values of all stratification features will form a strata. For example, the target column `species` of the `mlr_tasks_penguins` task is imbalanced:

```
prop.table(table(task$data(cols = "species")))
```

```
species
   Adelie Chinstrap    Gentoo
0.4418605 0.1976744 0.3604651
```

Without specifying a `"stratum"` column role, the `species` column may have quite different class distributions across the training and test sets of a 3-fold cross-validation strategy:

```
r = rsmp("cv", folds = 3)
r$instantiate(task)
prop.table(table(task$data(rows = r$test_set(1), cols = "species")))
```

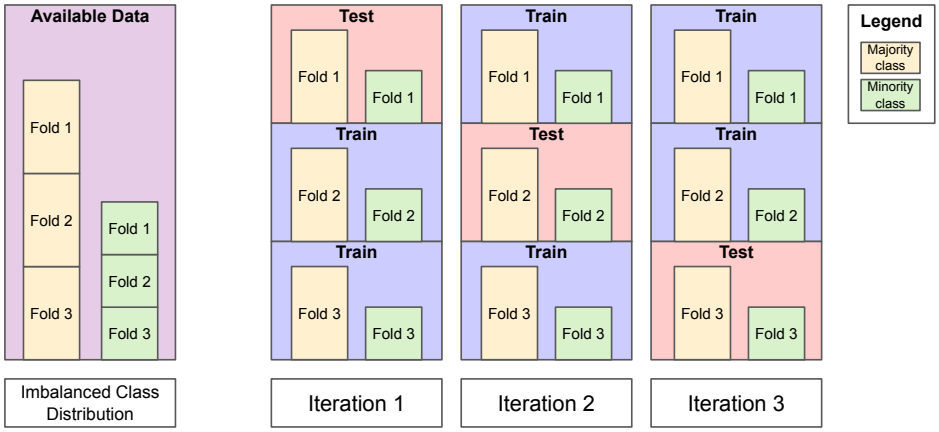


Figure 3.5: Illustration of a 3-fold cross-validation with stratification for an imbalanced binary classification task with a majority class that is about twice as large as the minority class. In each resampling iteration, the class distribution from the available data is preserved (which is not necessarily the case for cross-validation without stratification).

```
species
   Adelie Chinstrap    Gentoo
0.3913043 0.2347826 0.3739130
```

```
prop.table(table(task$data(rows = r$test_set(2), cols = "species")))
```

```
species
   Adelie Chinstrap    Gentoo
0.4434783 0.1478261 0.4086957
```

```
prop.table(table(task$data(rows = r$test_set(3), cols = "species")))
```

```
species
   Adelie Chinstrap    Gentoo
0.4912281 0.2105263 0.2982456
```

In the worst case, and especially for highly imbalanced classes, the minority class might be entirely left out of the training set in one or more resampling iterations. Consequently, the intermediate models within these resampling iterations will never predict the minority class, resulting in a misleading performance estimate for any resampling strategy without stratification. Relying on such a misleading performance estimate can have severe consequences for a deployed model, as it will perform poorly on the minority class in real-world scenarios. For example, misclassification of the minority class can have serious consequences in certain applications such as in medical diagnosis or fraud detection, where failing to identify the minority class may result in serious harm or financial losses. Therefore, it is important to be aware of the potential consequences of imbalanced class distributions in resampling and use stratification to mitigate highly unreliable performance estimates. The code below uses `species` as `"stratum"` column role to illustrate that the distribution of `species` in each test set will closely match the original distribution:

```
task_str = tsk("penguins")
task_str$col_roles$stratum = "species"
r = rsmp("cv", folds = 3)
r$instantiate(task_str)

prop.table(table(task_str$data(rows = r$test_set(1), cols = "species")))
```

```
species
   Adelie Chinstrap    Gentoo
0.4396552 0.1982759 0.3620690
```

```
prop.table(table(task_str$data(rows = r$test_set(2), cols = "species")))
```

```
species
   Adelie Chinstrap    Gentoo
0.4434783 0.2000000 0.3565217
```

```
prop.table(table(task_str$data(rows = r$test_set(3), cols = "species")))
```

```
species
   Adelie Chinstrap    Gentoo
0.4424779 0.1946903 0.3628319
```

Rather than assigning the `$col_roles$stratum` directly, it is also possible to use the `$set_col_roles()` method to add or remove columns to specific roles incrementally:

```
task_str$set_col_roles("species", remove_from = "stratum")
task_str$col_roles$stratum
```

```
character(0)
```

```
task_str$set_col_roles("species", add_to = "stratum")
task_str$col_roles$stratum
```

```
[1] "species"
```

We can further inspect the current stratification via the `$strata` field, which returns a `data.table` of the number of observations (`N`) and row indices (`row_id`) of each stratum. Since we stratified by the `species` column, we expect to see the same class frequencies as when we tabulate the task by the `species` column:

```
task_str$strata
```

```
     N                      row_id
1: 152             1,2,3,4,5,6,...
2: 124 153,154,155,156,157,158,...
3:  68 277,278,279,280,281,282,...
```

```
table(task$data(cols = "species"))
```

```
species
   Adelie Chinstrap    Gentoo
      152        68       124
```

Should we add another stratification column, the `$strata` field will show the same values as when we cross-tabulate the two variables of the task:

```
task_str$set_col_roles("year", add_to = "stratum")

task_str$strata
```

```
    N                       row_id
1: 50             1,2,3,4,5,6,...
2: 50       51,52,53,54,55,56,...
3: 52 101,102,103,104,105,106,...
4: 34 153,154,155,156,157,158,...
5: 46 187,188,189,190,191,192,...
6: 44 233,234,235,236,237,238,...
7: 26 277,278,279,280,281,282,...
8: 18 303,304,305,306,307,308,...
9: 24 321,322,323,324,325,326,...
```

```
table(task$data(cols = c("species", "year")))
```

```
           year
species     2007 2008 2009
  Adelie      50   50   52
  Chinstrap   26   18   24
  Gentoo      34   46   44
```

3.2.8 Plotting Resample Results

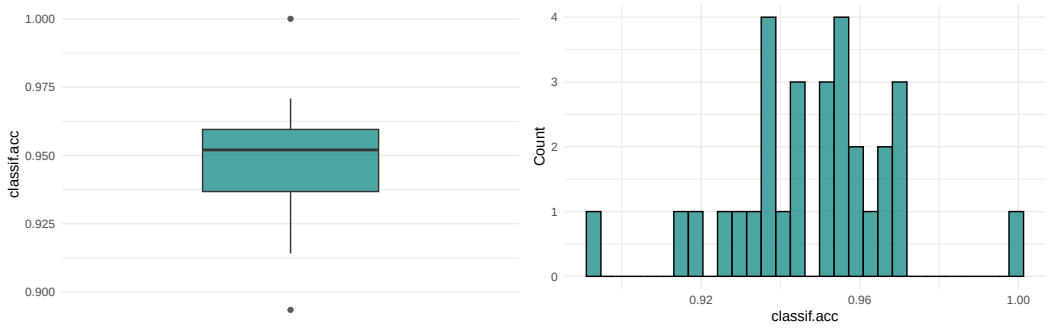
mlr3viz provides a autoplot() method to automatically visualize the resampling results either in a boxplot or histogram:

```
resampling = rsmp("bootstrap")
rr = resample(task, learner, resampling)

library(mlr3viz)
autoplot(rr, measure = msr("classif.acc"), type = "boxplot")
autoplot(rr, measure = msr("classif.acc"), type = "histogram")
```

```
`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

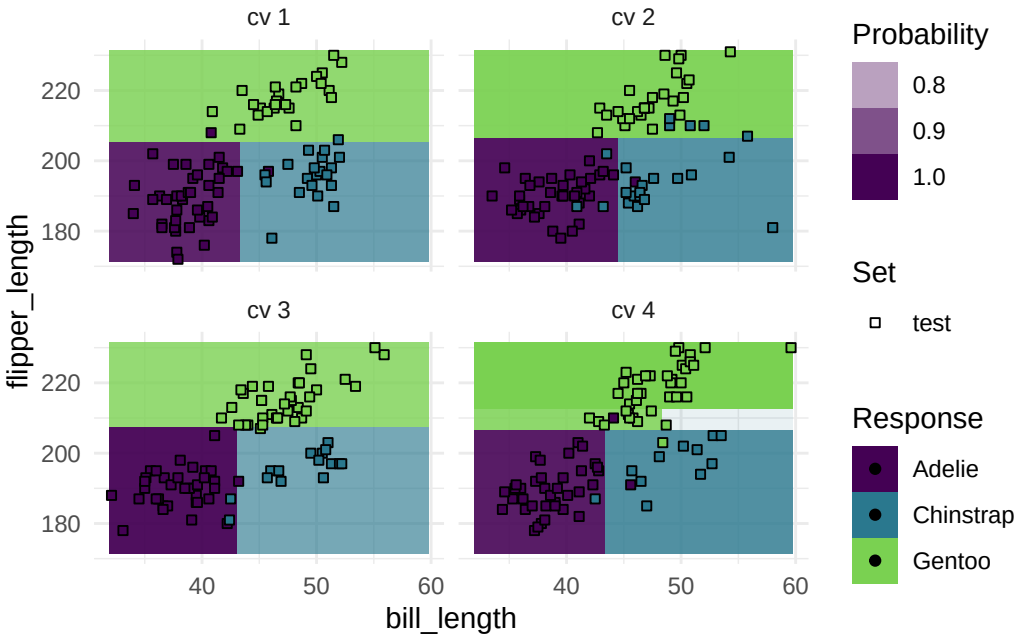
The histogram is useful to visually gauge the variance of the performance results across resampling iterations, whereas the boxplot is often used when multiple learners are compared side-by-side.

We can also visualize a 2-dimensional prediction surface of individual models in each resampling iteration if the task is restricted to two features:




```
task$select(c("bill_length", "flipper_length"))
resampling = rsmp("cv", folds = 4)
rr = resample(task, learner, resampling, store_models = TRUE)
autoplot(rr, type = "prediction")
```

```
Warning: Removed 2 rows containing missing values (`geom_point()`).
```



Prediction surfaces like this are a useful tool for model inspection, as they can help to identify the cause of unexpected performance result. Naturally, they are also popular for didactical purposes to illustrate the prediction behaviour of different learning algorithms, such as the classification tree in the example above with its characteristic orthogonal lines.

3.3 Benchmarking

Benchmarking in supervised machine learning refers to the comparison of different learners on a single task or multiple tasks. When comparing learners on a single task or on a domain consisting of multiple similar tasks, the main aim is often to rank the learners according to a pre-defined performance measure and to identify the best-performing learner for the considered task or domain. In an applied setting, benchmarking may be used to evaluate whether a deployed model used for a given task or domain can be replaced by a better alternative solution. When comparing multiple learners on multiple tasks, the main aim is often more of a scientific nature, e.g., to gain insights into how different learners perform in different data situations or whether there are certain data properties that heavily affect the performance of certain learners (or certain hyperparameter of learners). For example, it is common practice for algorithm designers to analyze the generalization performance or runtime of a newly proposed learning algorithm in a benchmark study where it has been compared with existing learners. In Section 3.3, we provide code examples for conducting benchmark studies and performing statistical analysis of benchmark results using the `mlr3` package.

The `mlr3` package offers the convenience function `benchmark()` to conduct a benchmark experiment. The function internally runs the `resample()` function on each task separately. The provided resampling strategy is instantiated on each task to ensure a fair comparison by training and evaluating multiple learners under the same conditions. This means that all provided learners use the same train-test splits for each task. In this section, we cover how to

- construct a benchmark design (Section 3.3.1) to define the benchmark experiments to be performed,
- run the benchmark experiments (Section 3.3.2) and aggregate their results, and
- convert benchmark objects (Section 3.3.3) to other types of objects that can be used for different purposes.

3.3.1 Constructing Benchmarking Designs

In `mlr3`, we can define a design to perform benchmark experiments via the `benchmark_grid()` convenience function. The design is essentially a table of scenarios to be evaluated and usually consists of unique combinations of `Task`, `Learner` and `Resampling` triplets.

The `benchmark_grid()` function constructs an exhaustive design to describe which combinations of learner, task and resampling should be used in a benchmark experiment. It properly instantiates the used resampling strategies so that all learners are evaluated on the same train-test splits for each task, ensuring a fair comparison. To construct a list of `Task`, `Learner` and `Resampling` objects, we can use the convenience functions `tsks()`, `lrns()`, and `rsmps()`.

We design an exemplary benchmark experiment and train a classification tree from the `rpart` package, a random forest from the `ranger` package and a featureless learner serving as a baseline on four different binary classification tasks. The constructed benchmark design is a `data.table` containing the task, learner, and resampling combinations in each row that should be performed:

```
library("mlr3verse")

tsks = tsks(c("german_credit", "sonar", "breast_cancer"))
lrns = lrns(c("classif.ranger", "classif.rpart", "classif.featureless"),
  predict_type = "prob")
rsmp = rsmps("cv", folds = 5)

design = benchmark_grid(tsks, lrns, rsmp)
head(design)
```

```
           task             learner resampling
1: german_credit      classif.ranger         cv
2: german_credit       classif.rpart         cv
3: german_credit classif.featureless         cv
4:         sonar      classif.ranger         cv
5:         sonar       classif.rpart         cv
6:         sonar classif.featureless         cv
```

Since the `data.table` contains R6 columns within list-columns, we unfortunately can not infer too much about `task` column, but the `ids` utility function can be used for quick inspection or subsetting:

```
mlr3misc::ids(design$task)
```

```
[1] "german_credit" "german_credit" "german_credit" "sonar"
[5] "sonar"         "sonar"         "breast_cancer" "breast_cancer"
[9] "breast_cancer"
```

```
design[mlr3misc::ids(task) == "sonar", ]
```

```
    task             learner resampling
1: sonar      classif.ranger         cv
2: sonar       classif.rpart         cv
3: sonar classif.featureless         cv
```

It is also possible to subset the design, e.g., to exclude a specific task-learner combination by manually removing a certain row from the design which is a `data.table`. Alternatively, we can also construct a custom benchmark design by manually defining a `data.table` containing task, learner, and resampling objects (see also the examples section in the help page of `benchmark_grid()`).

3.3.2 Execution of Benchmark Experiments

To run the benchmark experiment, we can pass the constructed benchmark design to the `benchmark()` function, which will internally call `resample()` for all the combinations of task, learner, and resampling strategy in our benchmark design:

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

```
<BenchmarkResult> of 45 rows with 9 resampling runs
 nr       task_id          learner_id resampling_id iters warnings errors
  1 german_credit      classif.ranger            cv     5        0      0
  2 german_credit       classif.rpart            cv     5        0      0
  3 german_credit classif.featureless            cv     5        0      0
  4         sonar      classif.ranger            cv     5        0      0
  5         sonar       classif.rpart            cv     5        0      0
  6         sonar classif.featureless            cv     5        0      0
  7 breast_cancer      classif.ranger            cv     5        0      0
  8 breast_cancer       classif.rpart            cv     5        0      0
  9 breast_cancer classif.featureless            cv     5        0      0
```

Once the benchmarking is finished (this can take some time, depending on the size of your design), we can aggregate the performance results with the `$aggregate()` method of the returned `BenchmarkResult`:

```
acc = bmr$aggregate(msr("classif.acc"))
acc[, .(task_id, learner_id, classif.acc)]
```

```
         task_id          learner_id classif.acc
1: german_credit      classif.ranger   0.7620000
2: german_credit       classif.rpart   0.7140000
3: german_credit classif.featureless   0.7000000
4:         sonar      classif.ranger   0.8322880
5:         sonar       classif.rpart   0.6491289
6:         sonar classif.featureless   0.5340302
7: breast_cancer      classif.ranger   0.9721447
8: breast_cancer       classif.rpart   0.9516531
9: breast_cancer classif.featureless   0.6500000
```

As the results are shown in a `data.table`, we can easily aggregate the results even further. For example, if we are interested in the learner that performed best across all tasks, we could average the performance of each individual learner across all tasks. Please note that averaging accuracy scores across multiple tasks as in this example is not always appropriate for comparison purposes. A more common alternative to compare the overall algorithm performance across multiple tasks is to first compute the ranks of each learner on each task separately and then compute the average ranks. For illustration purposes, we show how to average the performance of each individual learner across all tasks:

```
acc[, list(mean_accuracy = mean(classif.acc)), by = "learner_id"]
```

```
           learner_id mean_accuracy
1:      classif.ranger     0.8554776
2:       classif.rpart     0.7715940
3: classif.featureless     0.6280101
```

Ranking the performance scores can either be done via standard `data.table` syntax, or more conveniently with the `mlr3benchmark` package. We first use `as.BenchmarkAggr` to aggregate the `BenchmarkResult` using our measure, after which we use the `$rank_data()` method to convert the performance scores to ranks. The `minimize` argument is used to indicate that the classification accuracy should not be minimized, i.e. a higher score is better.

```
library("mlr3benchmark")

bma = as.BenchmarkAggr(bmr, measures = msr("classif.acc"))
```

```
Warning: 'as.BenchmarkAggr' is deprecated.
Use 'as_benchmark_aggr' instead.
See help("Deprecated")
```

```
bma$rank_data(minimize = FALSE)
```

```
            german_credit sonar breast_cancer
ranger                  1     1             1
rpart                   2     2             2
featureless             3     3             3
```

This results in per-task rankings of the three learners. Unsurprisingly, the featureless learner ranks last, as it always predicts the majority class. However, it is common practice to include it as a baseline in benchmarking experiments to easily gauge the relative performance of other algorithms. In this simple benchmark experiment, the random forest ranked first, outperforming a single classification tree as one would expect.

3.3.3 Inspect BenchmarkResult Objects

A `BenchmarkResult` object is a collection of multiple `ResampleResult` objects. We can analogously use `as.data.table` to take a look at the contents and compare them to the `data.table` of the `ResampleResult` from the previous section (`rrdt`):

```
bmrdt = as.data.table(bmr)

names(bmrdt)
```

```
[1] "uhash"      "task"       "learner"    "resampling" "iteration"
[6] "prediction"
```

```
names(rrdt)
```

```
[1] "task"       "learner"    "resampling" "iteration"  "prediction"
```

By the column names alone, we see that the general contents of a `BenchmarkResult` and `ResampleResult` which we specified in Section 3.2.5 is very similar, with the additional unique identification column `"uhash"` in the former being the only difference.

The stored `ResampleResult`s can be extracted via the `$resample_result(i)` method, where `i` is the index of the performed benchmark experiment. This allows us to investigate the extracted `ResampleResult` or individual resampling iterations as shown previously (see Section 3.2).

```
rr1 = bmr$resample_result(1)
rr2 = bmr$resample_result(2)
rr1
```

```
<ResampleResult> with 5 resampling iterations
       task_id     learner_id resampling_id iteration warnings errors
 german_credit classif.ranger            cv         1        0      0
 german_credit classif.ranger            cv         2        0      0
 german_credit classif.ranger            cv         3        0      0
 german_credit classif.ranger            cv         4        0      0
 german_credit classif.ranger            cv         5        0      0
```

```
rr2
```

```
<ResampleResult> with 5 resampling iterations
       task_id    learner_id resampling_id iteration warnings errors
 german_credit classif.rpart            cv         1        0      0
 german_credit classif.rpart            cv         2        0      0
 german_credit classif.rpart            cv         3        0      0
 german_credit classif.rpart            cv         4        0      0
 german_credit classif.rpart            cv         5        0      0
```

Multiple `ResampleResult` can be again converted to a `BenchmarkResult` with the function `as_benchmark_result()` and combined with `c()`:

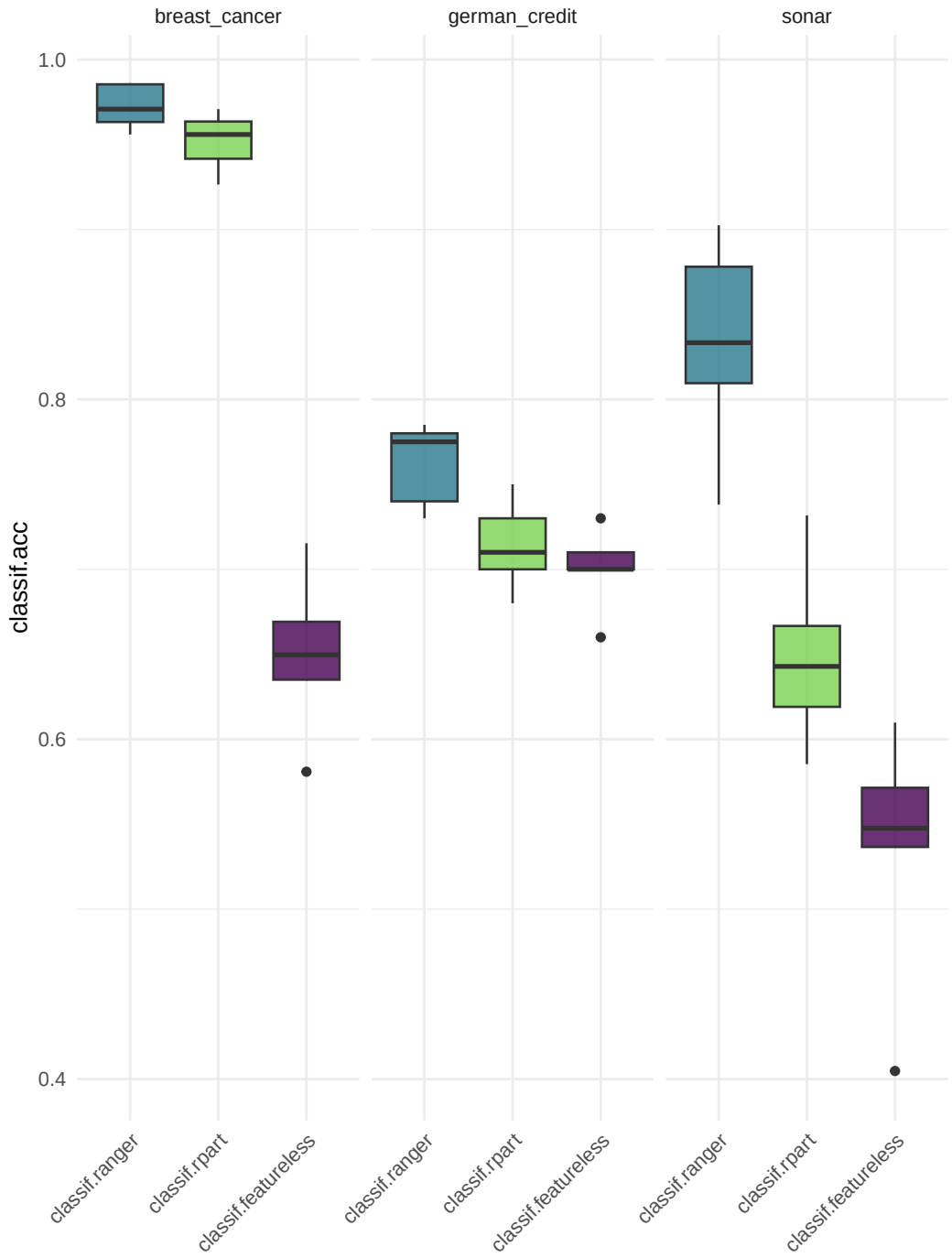
```
bmr1 = as_benchmark_result(rr1)
bmr2 = as_benchmark_result(rr2)

bmr_combined = c(bmr1, bmr2)
bmr_combined$aggregate(msr("classif.acc"))
```

```
   nr       task_id     learner_id resampling_id iters classif.acc
1:  1 german_credit classif.ranger            cv     5       0.762
2:  2 german_credit  classif.rpart            cv     5       0.714
Hidden columns: resample_result
```

Combining multiple `BenchmarkResult`s into a larger result object can be useful if related benchmarks where computed on different machines.

Similar to creating automated visualizations for tasks, predictions, or resample results, the `mlr3viz` package also provides a `autoplot()` method to visualize benchmark results, by default as a boxplot:

```
autoplot(bmr, measure = msr("classif.acc"))
```



Such a plot summarizes the benchmark experiment across all tasks and learners. Visualizing performance scores across all learners and tasks in a benchmark helps identifying potentially unexpected behavior, such as a learner performing reasonably well for most tasks, but yielding noticeably worse scores in one task. In the case of our example above, the three learners show consistent relative performance to each other, in the order we would expect.

3.3.4 Statistical Tests

> **!** **Advanced section**

The package `mlr3benchmark` we previously used for ranking also provides infrastructure for applying statistical significance tests on `BenchmarkResult` objects. Currently, Friedman tests and pairwise Friedman-Nemenyi tests (Demšar 2006) are supported to analyze benchmark experiments with at least two independent tasks and at least two learners.

`$friedman_posthoc()` can be used for a pairwise comparison:

```
bma = as.BenchmarkAggr(bmr, measures = msr("classif.acc"))
```

```
Warning: 'as.BenchmarkAggr' is deprecated.
Use 'as_benchmark_aggr' instead.
See help("Deprecated")
```

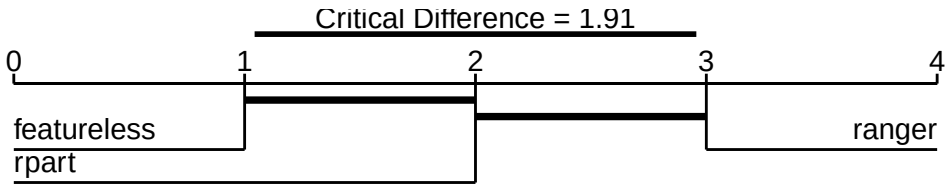
```
bma$friedman_posthoc()
```

```
    Pairwise comparisons using Nemenyi-Wilcoxon-Wilcox all-pairs test for a two-way balanced compl

data: acc and learner_id and task_id

           ranger rpart
rpart      0.438  -
featureless 0.038  0.438

P value adjustment method: single-step
```

These results would indicate a statistically significant difference between the `"featureless"` learner and `"ranger"`, assuming a 95% confidence level.

The results can be summarized in a critical difference plot which typically shows the mean rank of a learning algorithm on the x-axis along with a thick horizontal line that connects learners which are not significantly different:

```
autoplot(bma, type = "cd")
```



Similar to the test output before, this visualization leads to the conclusion that the `"featureless"` learner and `"ranger"` are significantly different, whereas the critical rank difference of 1.66 is not exceed for the comparison of the `"featureless"` learner, `"rpart"` and `"ranger"`, respectively.

3.4 ROC Analysis

ROC (Receiver Operating Characteristic) analysis is widely used to evaluate binary classifiers. Although extensions for multiclass classifiers exist (see e.g., Hand and Till (2001)), we will only cover the much easier binary classification case here. For binary classifiers that predict discrete classes, we can compute a confusion matrix which computes the following quantities (see also Figure 3.6):

- **True positives (TP)**: Instances that are actually positive and correctly classified as positive.
- **True negatives (TN)**: Instances that are actually negative and correctly classified as negative.
- **False positives (FP)**: Instances that are actually negative but incorrectly classified as positive.
- **False negatives (FN)**: Instances that are actually positive but incorrectly classified as negative.

There are a multitude of performance measures that can be derived from a confusion matrix. Unfortunately, many of them have different names for historical reasons, originating from different fields. For a good overview of common confusion matrix-based measures, see the

comprehensive table on Wikipedia[1] which also provides many common aliases for each measure.

3.4.1 Confusion Matrix-based Measures

Some common performance measures that are based on the confusion matrix and measure the ability of a classifier to separate the two classes (i.e., discrimination performance) include (see also Figure 3.6 for their definition based on TP, FP, TN and FN):

- **True Positive Rate (TPR)**, **Sensitivity** or **Recall**: How many of the true positives did we predict as positive?
- **True Negative Rate (TNR)** or **Specificity**: How many of the true negatives did we predict as negative?
- **False Positive Rate (FPR)**, or 1 - **Specificity**: How many of the true negatives did we predict as positive?
- **Positive Predictive Value (PPV)** or **Precision**: 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?
- **Accuracy (ACC)**: The proportion of correctly classified instances out of the total number of instances.
- **F1-score**: The harmonic mean of precision and recall, which balances the trade-off between precision and recall. It is calculated as $2 \times \frac{Precision \times Recall}{Precision + Recall}$.

		True Class y +	True Class y −	
Predicted Class $\hat{y}$	+	TP	FP	$\text{PPV} = \frac{\text{TP}}{\text{TP+FP}}$
	−	FN	TN	$\text{NPV} = \frac{\text{TN}}{\text{FN+TN}}$
		$\text{TPR} = \frac{\text{TP}}{\text{TP+FN}}$	$\text{TNR} = \frac{\text{TN}}{\text{FP+TN}}$	$\text{ACC} = \frac{\text{TP+TN}}{\text{TP+FP+FN+TN}}$

Figure 3.6: Binary confusion matrix of ground truth class vs. predicted class.

In the code example below, we first retrieve the `mlr_tasks_german_credit` task which is a binary classification task and construct a random forest learner using `classif.ranger` that predicts probabilities using the `predict_type = "prob"` option. Next, we use the `partition()` helper function which acts as a convenience shortcut function to the `"holdout"` resampling strategy to randomly partition the contained data into two disjoint set. We train the learner on the training set and use the trained model to generate predictions on the test set. Finally, we retrieve the confusion matrix from the resulting `Prediction` object by accessing the `$confusion` field (see also Section 2.4.3):

[1] https://en.wikipedia.org/wiki/Confusion_matrix#Table_of_confusion

```
task = tsk("german_credit")
learner = lrn("classif.ranger", predict_type = "prob")
splits = partition(task, ratio = 0.8)

learner$train(task, splits$train)
pred = learner$predict(task, splits$test)
pred$confusion
```

```
        truth
response good bad
    good  127  34
    bad    13  26
```

The `mlr3measures` package allows to additionally compute several common confusion matrix-based measures using the `confusion_matrix` function:

```
mlr3measures::confusion_matrix(truth = pred$truth,
  response = pred$response, positive = task$positive)
```

```
        truth
response good bad
    good  127  34
    bad    13  26
acc :  0.7650; ce  :  0.2350; dor :  7.4706; f1  :  0.8439
fdr :  0.2112; fnr :  0.0929; fomr:  0.3333; fpr :  0.5667
mcc :  0.3938; npv :  0.6667; ppv :  0.7888; tnr :  0.4333
tpr :  0.9071
```

If a binary classifier predicts probabilities instead of discrete classes, we could arbitrarily set a threshold to cut-off the probabilities and assign them to the positive and negative class. When it comes to classification performance, it is generally difficult to achieve a high TPR and low FPR simultaneously because there is often a trade-off between the two rates. Increasing the threshold for identifying the positive cases, leads to a higher number of negative predictions and fewer positive predictions. As a consequence, the FPR is usually better (lower), but at the cost of a worse (lower) TPR. For example, in the special case where the threshold is set too high and no instance is predicted as positive, the confusion matrix shows zero true positives (no instances that are actually positive and correctly classified as positive) and zero false positives (no instances that are actually negative but incorrectly classified as positive). Therefore, the FPR and TPR are also zero since there are zero false positives and zero true positives. Conversely, lowering the threshold for identifying positive cases may never predict the negative class and can increase (improve) TPR, but at the cost of a worse (higher) FPR. For example, below we set the threshold to `0.99` and `0.01` for the `mlr_tasks_german_credit` task to illustrate the two special cases explained above where zero positives and where zero negatives are predicted and inspect the resulting confusion matrix-based measures (some measures can not be computed due to division by 0 and therefore will produce `NaN` values):

```
pred$set_threshold(0.99)
mlr3measures::confusion_matrix(pred$truth, pred$response, task$positive)
```

```
        truth
```

```
response good bad
    good    0   0
    bad   140  60
acc :  0.3000; ce  :  0.7000; dor :  NaN; f1  :  NaN
fdr :  NaN; fnr :  1.0000; fomr:  0.7000; fpr :  0.0000
mcc :  0.0000; npv :  0.3000; ppv :  NaN; tnr :  1.0000
tpr :  0.0000
```

```
pred$set_threshold(0.01)
mlr3measures::confusion_matrix(pred$truth, pred$response, task$positive)
```

```
        truth
response good bad
    good  140  60
    bad     0   0
acc :  0.7000; ce  :  0.3000; dor :  NaN; f1  :  0.8235
fdr :  0.3000; fnr :  0.0000; fomr:  NaN; fpr :  1.0000
mcc :  0.0000; npv :  NaN; ppv :  0.7000; tnr :  0.0000
tpr :  1.0000
```

3.4.2 ROC Space

ROC analysis aims at evaluating the performance of classifiers by visualizing the trade-off between the TPR and the FPR which can be obtained from a confusion matrix. Each classifier that predicts discrete classes, will be a single point in the ROC space (see Figure 3.7, panel (a)). The best classifier lies on the top-left corner where the TPR is 1 and the FPR is 0. Classifiers on the diagonal predict class labels randomly (possibly with different class proportions). For example, if each positive instance will be randomly classified with 25% as to the positive class, we get a TPR of 0.25. If we assign each negative instance randomly to the positive class, we get a FPR of 0.25. In practice, we should never obtain a classifier clearly below the diagonal. Swapping the predicted classes of a classifier would results in points in the ROC space being mirrored at the diagonal baseline. A point in the ROC space below the diagonal might indicate that the positive and negative class labels have been switched by the classifier.

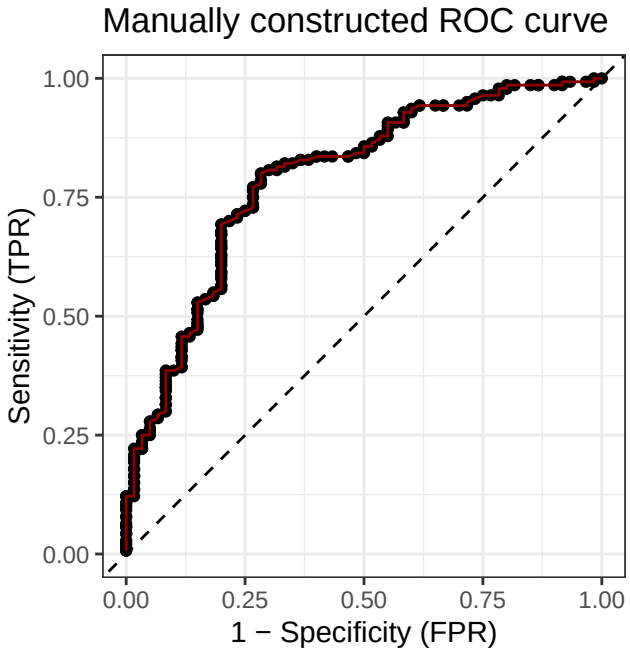
Using different thresholds to cut-off predicted probabilities and assign them to the positive and negative class may lead to different confusion matrices. In this case, we can characterize the behavior of a binary classifier for different thresholds by plotting the TPR and FPR values — this is the ROC curve. For example, we can use the previous `Prediction` object, compute all possible TPR and FPR combinations if we use all predicted probabilities as possible threshold, and visualize them to manually create a ROC curve:

```
thresholds = sort(pred$prob[,1])

rocvals = data.table::rbindlist(lapply(thresholds, function(t) {
  pred$set_threshold(t)
  data.frame(
    threshold = t,
    FPR = pred$score(msr("classif.fpr")),
    TPR = pred$score(msr("classif.tpr"))
```

```
  )
}))

head(rocvals)
```

```
   threshold       FPR       TPR
1: 0.2302968 1.0000000 1.0000000
2: 0.2579278 0.9833333 1.0000000
3: 0.2639405 0.9833333 0.9928571
4: 0.2657913 0.9666667 0.9928571
5: 0.2807516 0.9333333 0.9928571
6: 0.2961667 0.9166667 0.9928571
```

```
library(ggplot2)
ggplot(rocvals, aes(FPR, TPR)) +
  geom_point() +
  geom_path(color = "darkred") +
  geom_abline(linetype = "dashed") +
  coord_fixed(xlim = c(0, 1), ylim = c(0, 1)) +
  labs(
    title = "Manually constructed ROC curve",
    x = "1 - Specificity (FPR)",
    y = "Sensitivity (TPR)"
  ) +
  theme_bw()
```



A natural performance measure that can be derived from the ROC curve is the area under

the curve (AUC). The higher the AUC value, the better the performance, whereas a random classifier would result in an AUC of 0.5 (see Figure 3.7, panel (b) for an illustration). The AUC can be interpreted as the probability that a randomly chosen positive instance is ranked higher (in the sense that it gets a higher predicted probability of belonging to the positive class) by the classification model than a randomly chosen negative instance.

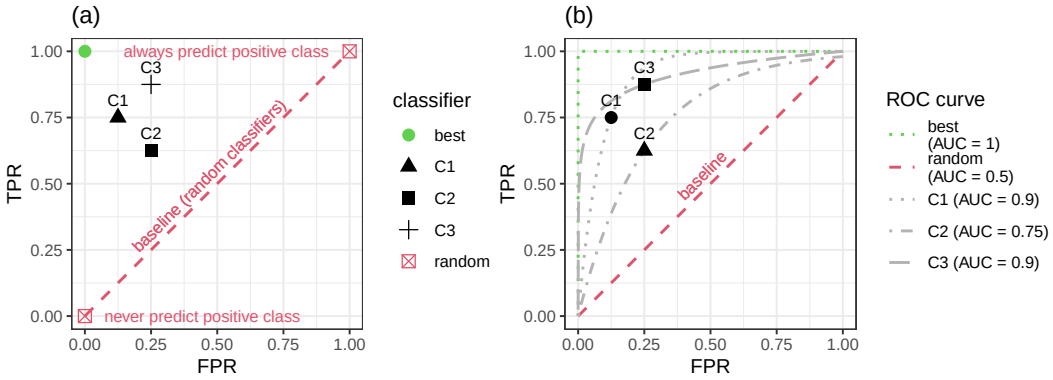
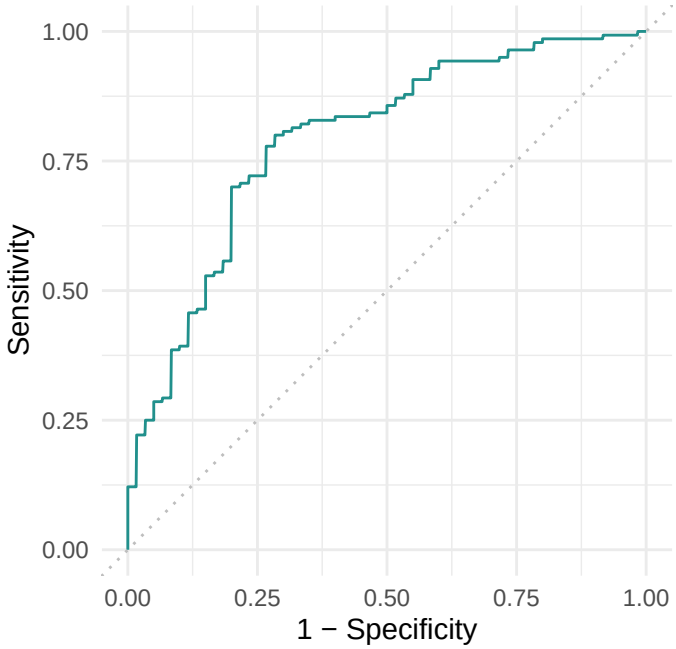


Figure 3.7: Panel (a): ROC space with best discrete classifier, two random guessing classifiers lying on the diagonal line (baseline), one that always predicts the positive class and one that never predicts the positive class, and three classifiers C1, C2, C3. We cannot say if C1 or C3 is better as both lie on a parallel line to the baseline. C2 is clearly dominated by C1, C3 as it is further away from the best classifier at (TPR = 1, FPR = 0). Panel (b): ROC curves of the best classifier (AUC = 1), of a random guessing classifier (AUC = 0.5), and the classifiers C1, C3, and C2.

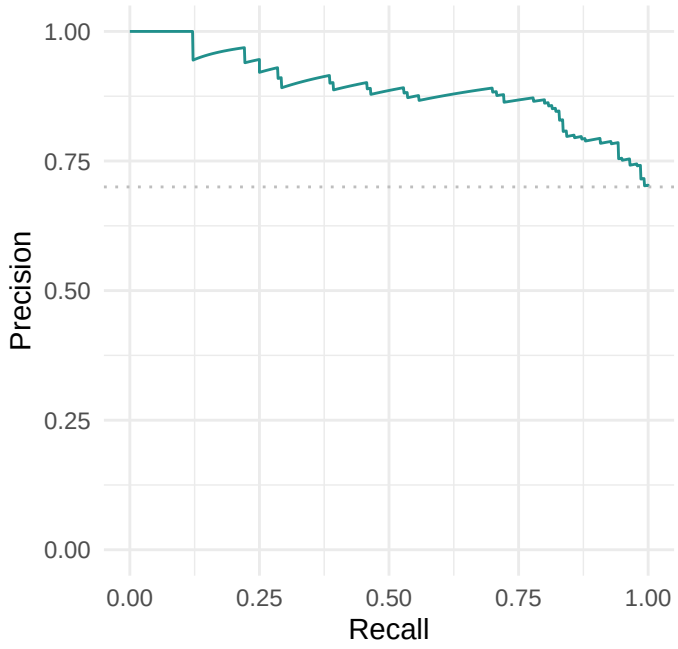
For `mlr3` prediction objects, the ROC curve can be constructed with the previously seen `autoplot.PredictionClassif` from `mlr3viz`. The x-axis showing the FPR is labelled "1 - Specificity" by convention, whereas the y-axis shows "Sensitivity" for the TPR.

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



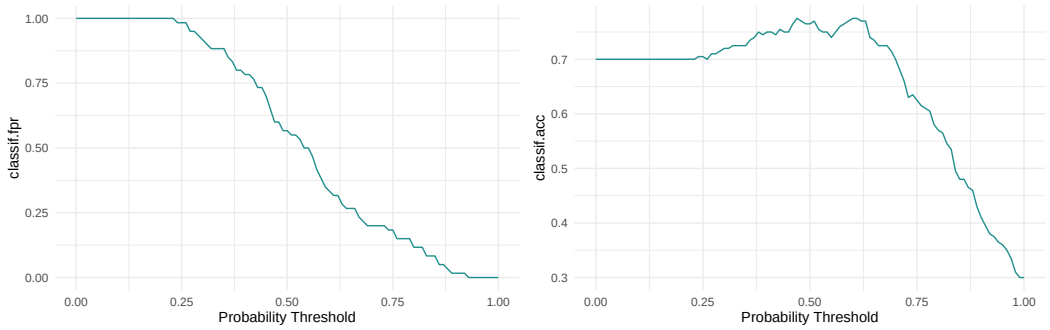
We can also plot the precision-recall (PR) curve which visualize the PPV vs. TPR. The main difference between ROC curves and PR curves is that the number of true-negatives are not used to produce a PR curve. PR curves are preferred over ROC curves for imbalanced populations. This is because the positive class is usually rare in imbalanced classification tasks. Hence, the FPR is often low even for a random classifier. As a result, the ROC curve may not provide a good assessment of the classifier's performance, because it does not capture the high rate of false negatives (i.e., misclassified positive observations). See also Davis and Goadrich (2006) for a detailed discussion about the relationship between the PRC and ROC curves.

```
autoplot(pred, type = "prc")
```



Another useful way to think about the performance of a classifier is to visualize the relationship of the set threshold with the performance metric at the given threshold. For example, if we want to see the FPR and accuracy across all possible thresholds:

```
autoplot(pred, type = "threshold", measure = msr("classif.fpr"))
autoplot(pred, type = "threshold", measure = msr("classif.acc"))
```

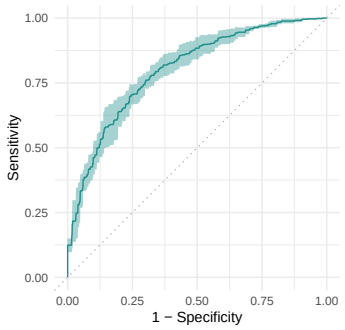


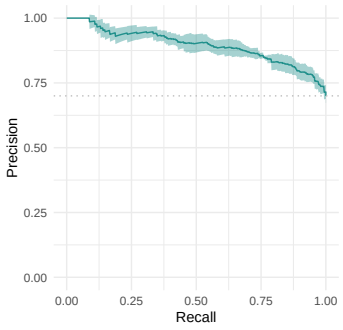
This visualization would show us that it would not matter if we picked a threshold of 0.5 or 0.75, since neither FPR nor accuracy changes in that range.

These visualizations are also available for `ResampleResult`. Here, the predictions of individual resampling iterations are merged prior to calculating a ROC or PR curve (micro-averaged):

```
rr = resample(
  task = tsk("german_credit"),
```

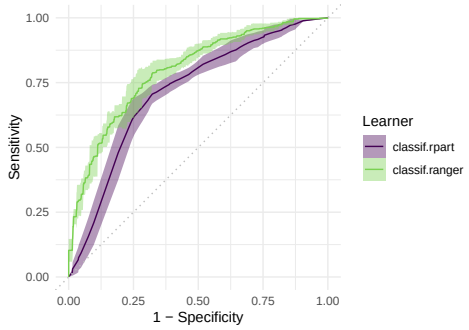
```
  learner = lrn("classif.ranger", predict_type = "prob"),
  resampling = rsmp("cv", folds = 5)
)
autoplot(rr, type = "roc")
autoplot(rr, type = "prc")
```

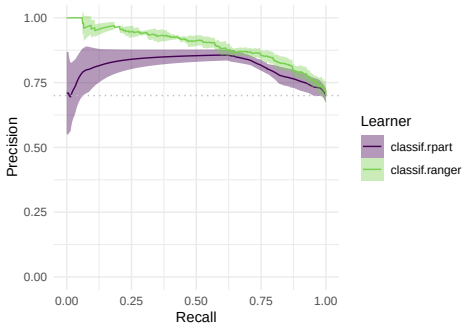




We can also visualize a `BenchmarkResult` to compare multiple learners on the same `Task`:

```
design = benchmark_grid(
  tasks = tsk("german_credit"),
  learners = lrns(c("classif.rpart", "classif.ranger"), predict_type = "prob"),
  resamplings = rsmp("cv", folds = 5)
)
bmr = benchmark(design)
autoplot(bmr, type = "roc")
autoplot(bmr, type = "prc")
```





3.5 Conclusion

In this chapter, we learned how to estimate the generalization performance of a model via resampling. We also learned about benchmarking to fairly compare the estimated generalization performance of different learners across multiple tasks. Performance calculations underpin these concepts, and we have seen some of them applied to classification tasks, with

a more in-depth look at the special case of binary classification and ROC analysis. We also learned how to visualize confusion matrix-based performance measures with regards to different thresholds as well as resampling and benchmark results with `mlr3viz`. The discussed topics belong to the fundamental concepts of supervised machine learning. Chapter 4 builds on these concepts and applies them for tuning (i.e., to automatically choose the optimal hyperparameters of a learner) through nested resampling (Section 4.3). In Chapter 7, we will also take a look at specialized tasks that require different resampling strategies. Finally, Table 3.1 provides an overview of some important `mlr3` functions and the corresponding R6 classes that were most frequently used throughout this chapter.

S3 function	R6 Class	Summary
`rsmp()`	`Resampling`	Assigns observations to train- and test sets
`resample()`	`ResampleResult`	Evaluates learners on given tasks using a resampling strategy
`benchmark_grid()`	-	Constructs a design grid of learners, tasks, and resamplings
`benchmark()`	`BenchmarkResult`	Evaluates learners on a given design grid

Table 3.1: Core S3 'sugar' functions for resampling and benchmarking in mlr3 with the underlying R6 class that are constructed when these functions are called (if applicable) and a summary of the purpose of the functions.

Resources

- Learn more about advanced resampling techniques in: Resampling - Stratified, Blocked and Predefined[2].
- Check out the blog post mlr3 Basics on "Iris" - Hello World![3] to see minimal examples on using resampling and benchmarking on the iris dataset.
- Use resampling and benchmarking for the comparison of decision boundaries of classification learners[4].
- Learn how to effectively pick thresholds by applying tuning and pipelines (Chapters 4 and 6) in this post on threshold tuning[5].

3.6 Exercises

1. Use the `spam` task and 5-fold cross-validation to benchmark Random Forest (`classif.ranger`), Logistic Regression (`classif.log_reg`), and XGBoost (`classif.xgboost`) with regards to AUC. Which learner appears to do best? How confident are you in your conclusion? How would you improve upon this?
2. A colleague claims to have achieved a 93.1% classification accuracy using the `classif.rpart` learner on the `penguins_simple` task. You want to reproduce

[2] https://mlr-org.com/gallery/basic/2020-03-30-stratification-blocking/
[3] https://mlr-org.com/gallery/basic/2020-03-18-iris-mlr3-basics/
[4] https://mlr-org.com/gallery/basic/2020-08-14-comparison-of-decision-boundaries/
[5] https://mlr-org.com/gallery/optimization/2020-10-14-threshold-tuning/index.html

their results and ask them about their resampling strategy. They said they used 3-fold cross-validation, and they assigned rows using the task's `row_id` modulo 3 to generate three evenly sized folds. Reproduce their results using the custom CV strategy.

4

Hyperparameter Optimization

Marc Becker
Ludwig-Maximilians-Universität München

Lennart Schneider
Ludwig-Maximilians-Universität München, and Munich Center for Machine Learning (MCML)

Hyperparameters

Machine learning algorithms usually include parameters and hyperparameters. Parameters are the model coefficients or weights or other information that are determined by the learning algorithm based on the training data. In contrast, hyperparameters, are configured by the user and determine how the model will fit its parameters, i.e., how the model is built. Examples include setting the number of trees in a random forest, penalty settings in support vector machines, or the learning rate in a neural network.

HPO: Hyperparameter Optimization

The goal of hyperparameter optimization (Section 4.1) or model tuning is to find the optimal configuration of hyperparameters of an ML algorithm for a given task. There is no closed-form mathematical representation (nor analytic gradient information) for model agnostic HPO. Instead, we follow a black-box optimization approach: an ML algorithm is configured with values chosen for one or more hyperparameters, this algorithm is then evaluated (using a resampling method) and its performance is measured. This process is repeated with multiple configurations and finally the configuration with the best performance is selected. HPO is very closely connected to the concepts of model evaluation and resampling (Chapter 3) as the objective is to find a hyperparameter configuration that maximizes the generalization performance. Broadly speaking, we could think of finding the optimal configuration in the same way as selecting a model from a benchmark experiment, where in this case each model uses different hyperparameter configurations. For example, we could try a support vector machine (SVM) with different `cost` values. However, human trial-and-error is time-consuming, often biased, error-prone, and computationally irreproducible. Instead, many sophisticated HPO methods (Section 4.1.4) (or 'tuners') have been developed over the past decades for robust and efficient HPO. Besides simple approaches such as a random search or grid search, most HPO methods employ iterative techniques that propose different configurations over time, often exhibiting adaptive behavior guided towards potentially optimal hyperparameter configurations. These methods continue to propose new configurations until a termination criterion is met, at which point the optimal configuration is returned. This iterative approach is depicted in a typical optimization loop as shown in Figure (Figure 4.1). For more details on HPO in general and more theoretical background, the reader is referred to Bischl et al. (2021) and Feurer and Hutter (2019).

4.1 Model Tuning

`mlr3tuning` is the hyperparameter optimization package of the mlr3 ecosystem. At the

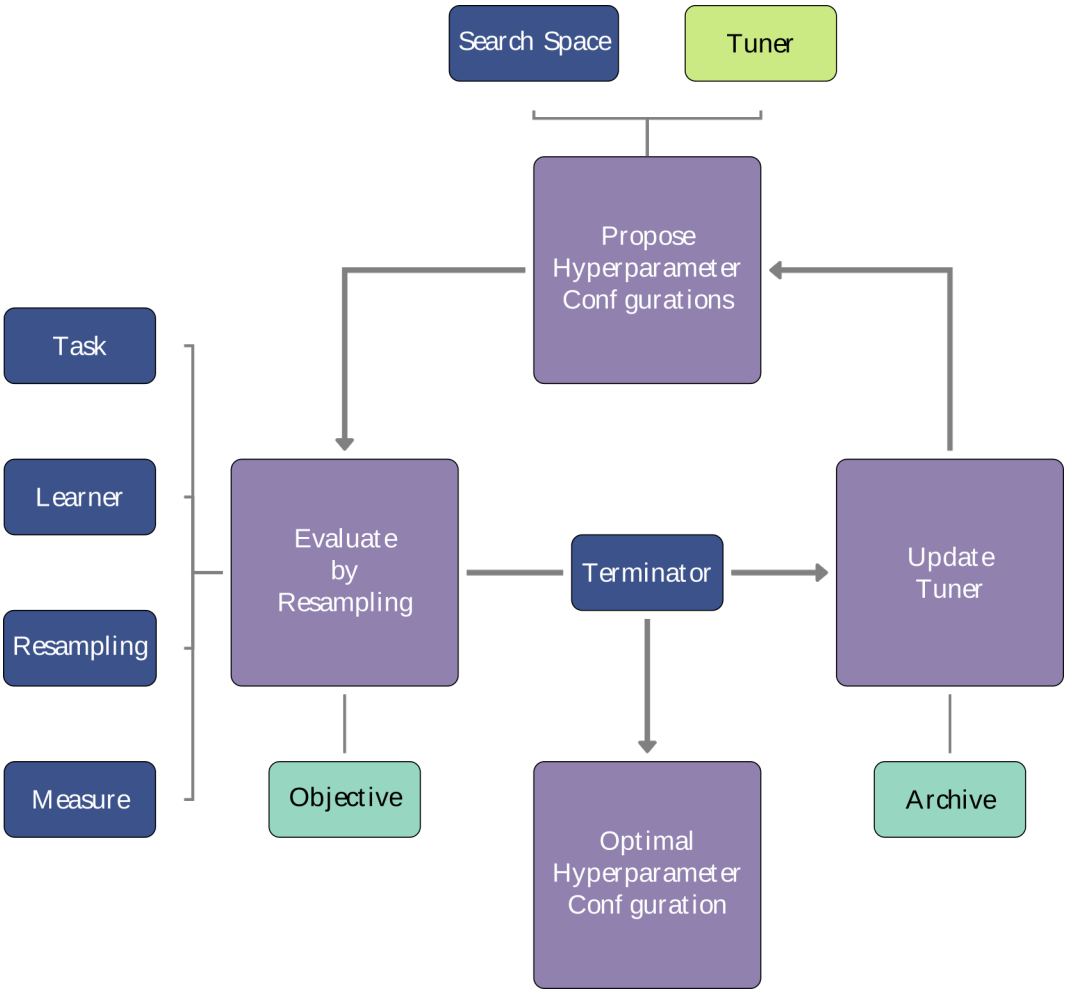


Figure 4.1: Representation of the hyperparameter optimization loop in mlr3tuning. Blue - Hyperparameter optimization loop. Purple - Objects of the tuning instance supplied by the user. Blue-Green - Internally created objects of the tuning instance. Green - Optimization Algorithm.

heart of the package (and indeed any optimization problem) are the R6 classes

- `TuningInstanceSingleCrit` and `TuningInstanceMultiCrit`, which are used to construct a tuning 'instance' which describes the optimization problem and stores the results; and
- `Tuner` which is used to get and set optimization algorithms.

In this section, we will cover these classes as well as other supporting functions and classes. Throughout this section, we will look at optimizing an SVM on the `sonar` data set as a running example.

4.1.1 Learner and Search Space

We begin by constructing a support vector machine from the `e1071` package with a radial kernel and specify that we want to tune this using `"C-classification"`.

```
learner = lrn("classif.svm", type = "C-classification", kernel = "radial")
```

Learner hyperparameter information is stored in the `$param_set` field, including parameter name, class (e.g., discrete or numeric), levels it can be tuned over, tuning limits, and more.

```
as.data.table(learner$param_set)[, .(id, class, lower, upper, nlevels)]
```

```
                id    class lower upper nlevels
 1:       cachesize ParamDbl  -Inf   Inf     Inf
 2:   class.weights ParamUty    NA    NA     Inf
 3:           coef0 ParamDbl  -Inf   Inf     Inf
 4:            cost ParamDbl     0   Inf     Inf
 5:           cross ParamInt     0   Inf     Inf
 6: decision.values ParamLgl    NA    NA       2
 7:          degree ParamInt     1   Inf     Inf
 8:         epsilon ParamDbl     0   Inf     Inf
 9:          fitted ParamLgl    NA    NA       2
10:           gamma ParamDbl     0   Inf     Inf
11:          kernel ParamFct    NA    NA       4
12:              nu ParamDbl  -Inf   Inf     Inf
13:           scale ParamUty    NA    NA     Inf
14:       shrinking ParamLgl    NA    NA       2
15:       tolerance ParamDbl     0   Inf     Inf
16:            type ParamFct    NA    NA       2
```

Note that `$param_set` also displays non-tunable parameters. Detailed information about parameters can be found in the help pages of the underlying implementation, for this example see `e1071::svm()`.

Given infinite resources, we could tune every single hyperparameter, but in reality that is not possible, so instead only a subset of hyperparameters can be tuned. This subset is referred to as the search space or tuning space. In this example we will tune the regularization and influence hyperparameters, `cost` and `gamma`.

Search Space

For numeric hyperparameters (we will explore others later) one must specify the bounds to tune over. We do this by constructing a learner and using `to_tune()` to set the lower and upper limits for the parameters we want to tune. This function allows us to construct

a learner in the usual way but to leave the hyperparameters of interest to be unspecified within a set range. This is best demonstrated by example:

```
learner = lrn("classif.svm",
  cost  = to_tune(1e-1, 1e5),
  gamma = to_tune(1e-1, 1),
  type  = "C-classification",
  kernel = "radial"
)
learner
```

```
<LearnerClassifSVM:classif.svm>
* Model: -
* Parameters: cost=<RangeTuneToken>, gamma=<RangeTuneToken>,
  type=C-classification, kernel=radial
* Packages: mlr3, mlr3learners, e1071
* Predict Types:  [response], prob
* Feature Types: logical, integer, numeric
* Properties: multiclass, twoclass
```

Here we have constructed a classification SVM by setting the type to `"C-classification"`, the kernel to `"radial"`, and not fully specifying the `cost` and `gamma` hyperparameters but instead indicating that we will tune these parameters. Note that calling `$train()` on a learner with tune token will throw an error.

> **Note**
>
> The `cost` and `gamma` hyperparameters are usually tuned on the logarithmic scale. You can find out more in Section 4.1.6.

Search spaces are usually chosen by experience. In some cases these can be quite complex. Section 4.5.1 and Section 4.5.2 and give a more detailed insight into the creation of tuning spaces. Section 4.5.3 introduces the `mlr3tuningspaces` extension package which allows loading of search spaces that have been established in published scientific articles.

4.1.2 Terminator

Theoretically, a tuner could search an entire search space exhaustively, however practically this is not possible and mathematically this is impossible for continuous hyperparameters. Therefore a core part of configuring tuning is to specify when to terminate the algorithm, this is also called the tuning budget. `mlr3tuning` includes many methods to specify when to terminate an algorithm, which are implemented in `Terminator` classes. Available terminators are listed in Table 4.1.

Tuning Budget Terminator

Terminator	Function call and default parameters
Clock Time	`trm("clock_time", stop_time = "2022-11-06 08:42:53 CET")`
Combo	`trm("combo", terminators = list(run_time_100, evals_200, any = TRUE))`
None	`trm("none")`

Terminator	Function call and default parameters
Number of Evaluations	`trm("evals", n_evals = 500)`
Performance Level	`trm("perf_reached", level = 0.1)`
Run Time	`trm("run_time", secs = 100)`
Stagnation	`trm("stagnation", iters = 5, threshold = 1e-5)`

Table 4.1: Terminators available in `mlr3tuning`, their function call and default parameters.

The most commonly used terminators are those that stop the tuning after a certain time (`trm("run_time")`) or the number of evaluations (`trm("evals")`). Choosing a runtime is often based on practical considerations and intuition. Using a time limit can be important on compute clusters where a maximum runtime for a compute job often needs to be specified. The `trm("perf_reached")` terminator stops the tuning when a certain performance level is reached, which can be helpful if a certain performance is seen as sufficient for the practical use of the model. However, one needs to be careful using this terminator since if the level is set too optimistically, the tuning might never terminate. The `trm("stagnation")` terminator stops when no progress is made for a number of iterations. The minimum progress is specified by the `threshold` argument. Note, `trm("stagnation")` could stop the optimization too aggressively if the search space is too complex i.e. the tuning terminates even though there is still potential for improvement. We use `trm("none")` when tuners, such as grid search and Hyperband (Section 4.6), control the termination themselves. Terminators can be freely combined with the `trm("combo")` terminator. The `any` argument determines if the optimization terminates when any or all included terminators stop. A complete and always up-to-date list of terminators can be found on our website[1]

4.1.3 Tuning Instance with `ti`

Tuning Instance

The tuning instance collects together the information required to optimize a model. A tuning instance can be constructed manually (Section 4.1.3) with the `ti()` function, or automated (Section 4.1.7) with the `tune()` function. We cover the manual approach first as this allows finer control of tuning and a more nuanced discussion about the design and use of `mlr3tuning`.

Now continuing our example, we will construct a single-objective tuning problem (i.e., tuning over one measure) by using the `ti()` function to create a `TuningInstanceSingleCrit`.

> **Note**
>
> Supplying more than one measure to `ti()` would result in a `TuningInstanceMultiCrit` (Section 4.7).

For this example we will use three-fold cross-validation and optimize the classification error measure. Note that we use `trm("none")` to perform an exhaustive grid search.

```
learner = lrn("classif.svm",
  cost  = to_tune(1e-1, 1e5),
```

[1] https://mlr-org.com/terminators.html

```
    gamma = to_tune(1e-1, 1),
    kernel = "radial",
    type = "C-classification"
  )

  instance = ti(
    task = tsk("sonar"),
    learner = learner,
    resampling = rsmp("cv", folds = 3),
    measures = msr("classif.ce"),
    terminator = trm("none")
  )
  instance
```

```
<TuningInstanceSingleCrit>
* State:  Not optimized
* Objective: <ObjectiveTuning:classif.svm_on_sonar>
* Search Space:
      id    class lower upper nlevels
1:  cost ParamDbl   0.1 1e+05     Inf
2: gamma ParamDbl   0.1 1e+00     Inf
* Terminator: <TerminatorNone>
```

4.1.4 Tuner

Tuner

After we created the tuning problem, we can look at *how* to tune. There are multiple `Tuner` classes in `mlr3tuning`, which implement different HPO (or more generally speaking black-box optimization) algorithms. While some algorithms are provided by external packages, others are implemented by the mlr3 team (see Table 4.2). Hyperband and Bayesian Optimization are included in the extension packages `mlr3hyperband` and `mlr3mbo`.

Tuner	Function call	Package
Random Search	`tnr("random_search")`	`mlr3tuning`
Grid Search	`tnr("grid_search")`	`mlr3tuning`
Bayesian Optimization	`tnr("mbo")`	`mlr3mbo`
CMA-ES	`tnr("cmaes")`	`adagio`
Iterative Racing	`tnr("irace")`	`irace`
Hyperband	`tnr("hyperband")`	`mlr3hyperband`
Generalized Simulated Annealing	`tnr("gensa")`	`GenSA`
Nonlinear Optimization	`tnr("nloptr")`	`nloptr`

Table 4.2: Tuning algorithms available in `mlr3tuning`, their function call and the package in which the algorithm is implemented.

Grid search and random search (Bergstra and Bengio 2012) are the most basic algorithms and are often selected first in initial experiments. The idea of grid search is to exhaustively evaluate every possible combination of given hyperparameter values determined by a resolution value, which is the number of distinct values to try per hyperparameter. Numeric and integer hyperparameters are spaced equidistantly in their box constraints (upper and

lower bounds), and categorical hyperparameters usually have all possible values considered. Random search involves randomly selecting values for each hyperparameter independently from a pre-specified distribution, usually uniform. Due to their simplicity, both grid search and random search can handle mixed search spaces (i.e., hyperparameters can be numeric, integer, or categorical) as well as hierarchical search spaces (i.e., some hyperparameters are inactive depending on other hyperparameters; for example when tuning an SVM we can consider different kernels, however, the gamma hyperparameter is only active if the kernel is given by the radial kernel and inactive for other kernels). Moreover, both are non-adaptive algorithms in the sense that they propose new configurations while ignoring performance from previous evaluations and are not guided towards potential optima during the optimization process.

In contrast, more sophisticated algorithms such as Bayesian optimization (also called model-based optimization), Covariance Matrix Adaptation Evolution Strategy (CMA-ES) or Iterative Racing learn from previously evaluated configurations to find good configurations more quickly.

Bayesian optimization (e.g., Snoek, Larochelle, and Adams 2012) describes a family of iterative optimization algorithms that use a so-called surrogate model to represent the unknown function that is to be optimized – in HPO the mapping from a hyperparameter configuration to the estimated generalization performance. Any Bayesian optimization algorithm starts by observing an initial design of observations and then trains the surrogate model on all data points and performance values observed so far. The algorithm then uses an acquisition function that usually relies on both the mean and variance prediction of the surrogate model to determine which points of the search space are promising candidates that should be evaluated next. By optimizing the acquisition function itself, the next candidate point is choosen for evaluation, evaluated and the process repeats itself by re-fitting or updating the surrogate model on the updated set of observed data points. Bayesian optimization is very flexible (e.g. mixed and hierarchical search spaces can easily be handled by choosing a suitable surrogate model), and highly sample efficient, i.e., compared to other algorithms, much less function evaluations are needed to find good configurations. On the downside, the optimization overhead of Bayesian Optimization is comparably large (e.g., in each iteration, training of the surrogate model and optimizing the acquisition function) and therefore really shines in the context of very costly function evaluations and tight optimization budget. For more details on Bayesian optimization, please see also **?@sec-bayesian-optimization**.

CMA-ES (Hansen and Auger 2011) is an evolutionary strategy that maintains a probability distribution over candidate points, with the distribution represented by a mean vector and covariance matrix. A new set of candidate points is generated by sampling from this distribution, with the probability of each candidate being proportional to its performance. The covariance matrix is adapted over time to reflect the performance landscape, with more promising regions of the search space receiving more focus.

The fundamental idea of racing algorithms is to discard configurations that show poor performance on the initial problem instances, for example, resampling folds. This is determined by running a statistical test and allows for an efficient allocation of computation time. Iterative Racing (López-Ibáñez et al. 2016) starts by racing down an initial population of randomly sampled configurations and then uses the surviving configurations of the race to stochastically initialize the population of the subsequent race to focus on promising regions of the search space.

Multi-fidelity HPO is also concerned with adaptive resource allocation by leveraging the predictive power of computationally cheap lower fidelity evaluations (for example using few

numbers of epochs when training a neural network or few boosting iterations when doing gradient boosting) to improve the overall optimization efficiency. This concept is used in Hyperband (Li et al. 2018, see also Section 4.6), a popular multi-fidelity HPO algorithm, which dynamically allocates increasingly more resources to promising configurations and terminates low-performing ones. By running different so-called brackets from different lower fidelities, Hyperband also reduces the risk of missing promising configurations. Hyperband usually results in better anytime performance (i.e., looking at the best performance obtained at any time point during optimization) compared to random search as the optimization budget is used much more efficiently.

Other implemented algorithms for numeric search spaces are Generalized Simulated Annealing (Xiang et al. 2013; Tsallis and Stariolo 1996) and various nonlinear optimization algorithms. These algorithms can be useful if evaluations are rather cheap as they require more function evaluations and are not that sample efficient as for example Bayesian optimization but also are more frequently used for general black-box optimization.

Advanced optimization algorithms within the mlr3 ecosystem are included in extension packages, for example the package `mlr3mbo` implements Bayesian optimization algorithms (see also **?@sec-bayesian-optimization**), and `mlr3hyperband` implements algorithms of the hyperband (Li et al. 2018) family (see Section 4.6). A complete and up-to-date list of tuners can be found on the website.

Note that the `param_classes` and `properties` fields of any tuner provide useful information regarding their use cases. `param_classes` tell us which classes of hyperparameters can be handled by the tuner, whereas `properties` for example state their ability to operate on hierarchical search spaces (as indicated by the `"dependencies"` property):

```
tnr("random_search")$param_classes
```

```
[1] "ParamLgl" "ParamInt" "ParamDbl" "ParamFct"
```

```
tnr("random_search")$properties
```

```
[1] "dependencies" "single-crit"  "multi-crit"
```

Besides making sure that a tuner can technically operate on a search spaces, choosing the *right* tuner is crucial for obtaining good results in HPO. As a rule of thumb, if the search space is small and consists of categorical hyperparameters, a grid search may be able to exhaustively evaluate the entire search space in a reasonable time. However, grid search is generally not recommended due to the curse of dimensionality and insufficient coverage of numeric search spaces. Grid search by construction also cannot evaluate a large number of unique values per hyperparameter, which is suboptimal when some hyperparameters have minimal impact on performance while others do.

In such scenarios, a random search is often a better choice as it considers more unique values per hyperparameter compared to grid search. Random search is also suitable for small optimization budgets and quick concept experiments. However, with larger optimization budgets, more guided optimization algorithms such as evolutionary strategies or Bayesian optimization tend to perform better and are more likely to result in peak performance.

When choosing between evolutionary strategies and Bayesian optimization, the cost of function evaluation is highly relevant. If hyperparameter configurations can be evaluated quickly, evolutionary strategies often find good configurations within a reasonable timeframe. On

the other hand, if model evaluations are time-consuming and the optimization budget is limited, Bayesian optimization is usually preferred over evolutionary strategies.

Finally, in cases where the HPO problem involves a meaningful fidelity parameter and optimization budget needs to be spent efficiently, multi-fidelity HPO algorithms like Hyperband may be worth considering. For further details on different tuners and practical recommendations, we refer to Bischl et al. (2021).

For our SVM example, we will use a simple grid search with a resolution of 5 for didactic reasons. Recall that the resolution is the number of distinct values to try *per hyperparameter*. The `batch_size` controls how many configurations are evaluated at the same time (see Section 8.1) and also determines the interval in which the terminator is checked. The `batch_size` parameter is only available for `tnr("random_search")` and `tnr("grid_search")` and is set to 1 by default. The other tuners set the batch size themselves.

```
tuner = tnr("grid_search", resolution = 5, batch_size = 10)
tuner
```

```
<TunerGridSearch>: Grid Search
* Parameters: resolution=5, batch_size=10
* Parameter classes: ParamLgl, ParamInt, ParamDbl, ParamFct
* Properties: dependencies, single-crit, multi-crit
* Packages: mlr3tuning
```

In our example we are tuning over two numeric parameters and `TunerGridSearch` will create an equidistant grid between the respective upper and lower bounds. This means our two-dimensional grid of resolution 5 consists of $5^2 = 25$ configurations. Each configuration is a distinct set of hyperparameter values that is evaluated on the given task using resampling (Figure 4.1). All configurations will be tried by the tuner (in random order) until either all configurations are evaluated or the terminator (Section 4.1.2) signals that the budget is exhausted.

Control Parameters

Similar to learners, which have hyperparameters, one can configure tuners through what we call control parameters. Unlike learners, the control parameters perform well enough in their default settings that they do not need to be changed often. However, changing them can still improve the performance of the tuner. Control parameters are stored in the `$param_set` field.

```
tuner$param_set
```

```
<ParamSet>
                  id    class lower upper nlevels        default value
1:        batch_size ParamInt     1   Inf     Inf <NoDefault[3]>    10
2:        resolution ParamInt     1   Inf     Inf <NoDefault[3]>     5
3: param_resolutions ParamUty    NA    NA     Inf <NoDefault[3]>
```

4.1.5 Run the Tuning

Now that we have all our components, we are ready to start tuning! To do this we simply pass the constructed `TuningInstanceSingleCrit` to the `$optimize()` method of the initialized `Tuner`. The tuner then proceeds with the HPO loop we discussed at the beginning of the

chapter (Figure 4.1).

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

```
       cost gamma learner_param_vals  x_domain classif.ce
1: 25000.08   0.1          <list[4]> <list[2]>  0.2448585
```

The optimizer returns the best hyperparameter configuration and the corresponding measured performance. This information is also stored in `instance$result`.

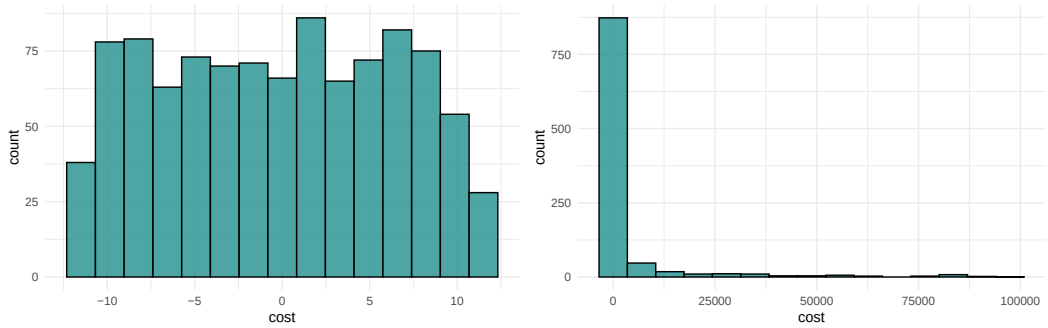
> **Note**
>
> The column `x_domain` contains transformed values (see Section 4.1.6) and `learner_param_vals` the transformed values and optional constants (none in this example). The result in `learner_param_vals` is usually passed to the final model (see Section 4.1.9).

4.1.6 Logarithmic Transformations

Logarithmic Scale

The SVM's `cost` and `gamma` parameters that we have tuned in the example above, are usually tuned on a logarithmic scale (as opposed to linear). The parameters vary over several orders of magnitude. Using a logarithmic scale allows us to efficiently search over a wide range of values. We can achieve this, by sampling uniformly from the interval $[log(1e-5), log(1e5)]$ and afterwards transforming the selected configuration with `exp()` before passing it to the learner. Using the log transformation emphasizes smaller values but can also result in large values. The code below demonstrates this more clearly. The histograms show how the algorithm searches within a narrow range but exponentiating then results in the majority of points being relatively small but a few being very large.

```
cost = runif(1000, log(1e-5), log(1e5))
```



(a) Values on the linear scale sampled by the tuner.

(b) Transformed values on the logarithmic scale as seen by the learner.

Figure 4.2: Histogram of uniformly sampled values from the interval $[log(1e-5), log(1e5)]$.

To add the `exp()` transformation to a hyperparameter, we pass `logscale = TRUE` to `to_tune()`.

```
learner = lrn("classif.svm",
  cost  = to_tune(1e-5, 1e5, logscale = TRUE),
  gamma = to_tune(1e-5, 1e5, logscale = TRUE),
  kernel = "radial",
  type = "C-classification"
)
instance = ti(
  task = tsk("sonar"),
  learner = learner,
  resampling = rsmp("cv", folds = 3),
  measures = msr("classif.ce"),
  terminator = trm("none")
)

tuner$optimize(instance)
```

```
       cost     gamma learner_param_vals  x_domain classif.ce
1: 5.756463 -5.756463          <list[4]> <list[2]>  0.1394065
```

The grid search with logarithmic transformation found a configuration with a much better performance. The column `x_domain` contains the hyperparameter values after the transformation i.e. `exp(5.76)` and `exp(-5.76)`:

```
instance$result$x_domain
```

```
[[1]]
[[1]]$cost
[1] 316.2278

[[1]]$gamma
[1] 0.003162278
```

You can learn more about transformations in Section 4.5.1.

4.1.7 Quick Tuning with `tune`

In the previous section, we looked at creating a tuning instance manually using `ti()`. However, you can also simplify this using the `tune()` helper function. Internally this creates a `TuningInstanceSingleCrit`, starts the tuning and returns the result with the instance. We have a little less control because we can't check the instance before the tuning starts.

```
learner = lrn("classif.svm",
  cost  = to_tune(1e-5, 1e5, logscale = TRUE),
  gamma = to_tune(1e-5, 1e5, logscale = TRUE),
  kernel = "radial",
  type = "C-classification"
)

instance = tune(
  tuner = tnr("grid_search", resolution = 5, batch_size = 5),
```

```
  task = tsk("sonar"),
  learner = learner,
  resampling = rsmp("cv", folds = 3),
  measures = msr("classif.ce")
)

instance$result
```

```
       cost     gamma learner_param_vals  x_domain classif.ce
1: 5.756463 -5.756463          <list[4]> <list[2]>  0.1443754
```

> **Note**
>
> The measured performance is different from the previous section because different train-test splits were used. The train-test splits are generated at the beginning of the optimization with the current seed (see Section 3.2.3).

4.1.8 Analyzing the Result

Regardless of using `ti` or `tune`, the output is the same and the archive lists all evaluated hyperparameter configurations:

```
as.data.table(instance$archive)[, .(cost, gamma, classif.ce)]
```

```
          cost      gamma classif.ce
 1: -11.512925 -11.512925  0.4663906
 2:   0.000000   0.000000  0.4663906
 3:   0.000000   5.756463  0.4663906
 4:   5.756463  11.512925  0.4663906
 5:  11.512925  11.512925  0.4663906
---
21: -11.512925  11.512925  0.4663906
22:  -5.756463  11.512925  0.4663906
23:   5.756463 -11.512925  0.2596273
24:   5.756463   0.000000  0.4663906
25:  11.512925   5.756463  0.4663906
```

Each row of the archive is a different evaluated configuration. The columns here show the tested configurations and the measure we optimize. If we only specify a single-objective criteria then the instance will return the configuration that optimizes this measure. However, we can manually inspect the archive to determine other important features. For example, when was the configuration evaluated? How long did the model take to run? Were there any errors or warnings while running?

```
as.data.table(instance$archive)[,
  .(timestamp, runtime_learners, errors, warnings)]
```

```
             timestamp runtime_learners errors warnings
 1: 2023-05-03 18:37:11            0.064      0        0
 2: 2023-05-03 18:37:11            0.083      0        0
```

```
 3: 2023-05-03 18:37:11            0.065      0        0
 4: 2023-05-03 18:37:11            0.077      0        0
 5: 2023-05-03 18:37:11            0.077      0        0
---
21: 2023-05-03 18:37:14            0.076      0        0
22: 2023-05-03 18:37:14            0.066      0        0
23: 2023-05-03 18:37:14            0.076      0        0
24: 2023-05-03 18:37:14            0.079      0        0
25: 2023-05-03 18:37:14            0.070      0        0
```

Another powerful feature of the instance is that we can score the internal `ResampleResult`s on a different performance measure, for example looking at false negative rate and false positive rate as well as classification error:

```
as.data.table(instance$archive,
  measures = msrs(c("classif.fpr", "classif.fnr")))[,
  .(cost, gamma, classif.ce, classif.fpr, classif.fnr)]
```

```
          cost      gamma classif.ce classif.fpr classif.fnr
 1: -11.512925 -11.512925  0.4663906   1.0000000   0.0000000
 2:   0.000000   0.000000  0.4663906   1.0000000   0.0000000
 3:   0.000000   5.756463  0.4663906   1.0000000   0.0000000
 4:   5.756463  11.512925  0.4663906   1.0000000   0.0000000
 5:  11.512925  11.512925  0.4663906   1.0000000   0.0000000
---
21: -11.512925  11.512925  0.4663906   1.0000000   0.0000000
22:  -5.756463  11.512925  0.4663906   1.0000000   0.0000000
23:   5.756463 -11.512925  0.2596273   0.2838196   0.2318296
24:   5.756463   0.000000  0.4663906   1.0000000   0.0000000
25:  11.512925   5.756463  0.4663906   1.0000000   0.0000000
```

You can access all the resamplings in a `BenchmarkResult` object with `instance$archive$benchmark_result`.

Finally, for more visually appealing results, you can use `mlr3viz` (Figure 4.3). We can observe nicely why a grid search is usually not the best tuner to choose. Although we evaluated 25 configurations in total, only 5 unique hyperparameter values were tried per hyperparameter (as the grid search by design considers all combinations of hyperparameter values).

```
autoplot(instance, type = "surface")
```

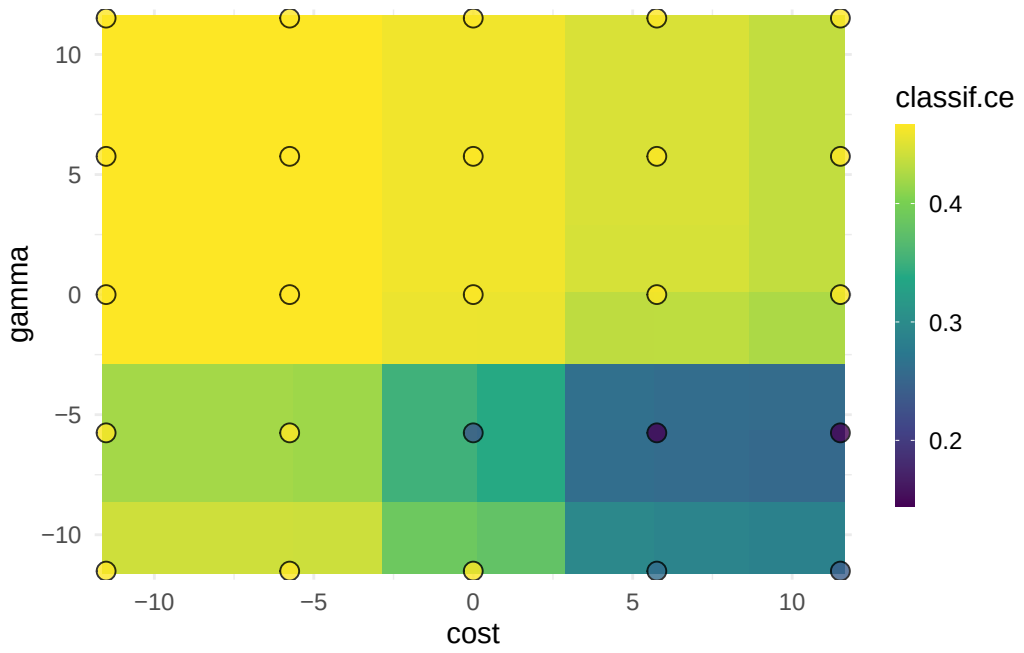


Figure 4.3: Model performance with different configurations for `cost` and `gamma`. Bright yellow regions represent the model performing worse and dark blue performing better. We can see that high `cost` values and low `gamma` values achieve the best performance. Note that we should not directly infer the performance of new unseen values from the heatmap since it is only an interpolation based on a surrogate model (`regr.ranger`). However, we can see the general interaction between the hyperparameters.

4.1.9 Using a tuned model

Once the learner has been tuned we can start to use it like any other model in the mlr3 universe. To do this we simply construct a new learner with the same underlying algorithm and set the learner hyperparameters to the optimal configuration:

```
svm_tuned = lrn("classif.svm")
svm_tuned$param_set$values = instance$result_learner_param_vals
```

Now we can train the learner on the full dataset and we are ready to make predictions. The trained model can then be used to predict new, external data:

```
svm_tuned$train(tsk("sonar"))
svm_tuned$model
```

```
Call:
svm.default(x = data, y = task$truth(), type = "C-classification",
    kernel = "radial", gamma = 0.00316227766016838, cost = 316.227766016838,
    probability = (self$predict_type == "prob"))
```

```
Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  radial
       cost:  316.2278

Number of Support Vectors:  93
```

> ⚠ Warning
>
> A common mistake when tuning is to report the performance estimated on the resampling sets on which the tuning was performed (`instance$result$classif.ce`) as an estimate of the model's performance. However, doing so would lead to bias and therefore nested resampling is required (Section 4.3). When tuning as above ensure that you do not make any statements about model performance without testing the model on more unseen data. We will come back to this in more detail in Section 4.2.

4.2 Automated Tuning with `AutoTuner`

One of the most useful classes in mlr3 is the `AutoTuner`. The `AutoTuner` wraps a learner and augments it with an automatic tuning process for a given set of hyperparameters – this allows transparent tuning of any learner, without the need to construct a new learner with the tuned configuration at the end. As the `AutoTuner` itself inherits from the `Learner` base class, it can be used like any other learner!

Let us see this in practice. We will run the exact same example as above but this time using the `AutoTuner` for automated tuning:

```
learner = lrn("classif.svm",
  cost  = to_tune(1e-5, 1e5, logscale = TRUE),
  gamma = to_tune(1e-5, 1e5, logscale = TRUE),
  kernel = "radial",
  type = "C-classification"
)

at = auto_tuner(
  tuner = tnr("grid_search", resolution = 5, batch_size = 5),
  learner = learner,
  resampling = rsmp("cv", folds = 3),
  measure = msr("classif.ce")
)

at
```

```
<AutoTuner:classif.svm.tuned>
* Model: list
* Search Space:
<ParamSet>
```

```
      id    class     lower    upper nlevels        default value
1:  cost ParamDbl -11.51293 11.51293     Inf <NoDefault[3]>
2: gamma ParamDbl -11.51293 11.51293     Inf <NoDefault[3]>
Trafo is set.
* Packages: mlr3, mlr3tuning, mlr3learners, e1071
* Predict Type: response
* Feature Types: logical, integer, numeric
* Properties: multiclass, twoclass
```

We can now use this like any other learner, calling the `$train()` and `$predict()` methods. The key difference to a normal learner is that calling `$train()` also tunes the learner's hyperparameters before fitting the model.

```
task = tsk("sonar")
split = partition(task)
at$train(task, row_ids = split$train)
at$predict(task, row_ids = split$test)$score()
```

```
classif.ce
 0.2173913
```

The `AutoTuner` contains a tuning instance that can be analyzed like any other instance (see Section 4.1.8).

```
at$tuning_instance
```

```
<TuningInstanceSingleCrit>
* State:  Optimized
* Objective: <ObjectiveTuning:classif.svm_on_sonar>
* Search Space:
      id    class     lower    upper nlevels
1:  cost ParamDbl -11.51293 11.51293     Inf
2: gamma ParamDbl -11.51293 11.51293     Inf
* Terminator: <TerminatorNone>
* Result:
       cost     gamma classif.ce
1: 5.756463 -5.756463  0.1726796
* Archive:
          cost      gamma classif.ce
 1:  -5.756463 -11.512925  0.4677767
 2:   0.000000  11.512925  0.4677767
 3:   5.756463  -5.756463  0.1726796
 4:  11.512925   0.000000  0.4677767
 5:  11.512925  11.512925  0.4677767
---
21: -11.512925  -5.756463  0.4677767
22: -11.512925   0.000000  0.4677767
23: -11.512925   5.756463  0.4677767
24:   5.756463   5.756463  0.4677767
25:  11.512925   5.756463  0.4677767
```

We could also pass the `AutoTuner` to `resample()` and `benchmark()`, which would result in

a nested resampling, discussed next.

4.3 Nested Resampling

Hyperparameter optimization generally requires an additional resampling to prevent a bias when estimating performance of the model. If the same data is used for determining the optimal configuration and the evaluation of the resulting model itself, the actual performance estimate of the model might be severely biased (Simon 2007). This is analogous to optimism of the training error described in James et al. (2014), which occurs when training error is taken as an estimate of out-of-sample performance.

Nested Resampling

Nested resampling separates model optimization from the process of estimating the performance of the tuned model by adding an additional resampling, i.e., while model performance is estimated using a resampling method in the 'usual way', tuning is then performed by resampling the resampled data (Figure 4.4). For more details and a formal introduction to nested resampling the reader is referred to Bischl et al. (2021).

A common confusion is how and when to use nested resampling. In the rest of this section we will answer the 'how' question but first the 'when'. A common mistake is to confuse nested resampling for model evaluation and comparison with tuning for model deployment. To put it differently, nested resampling is a statistical procedure to estimate the predictive performance of the model trained on the full dataset, it is *not* a procedure to select optimal hyperparameters. Nested resampling produces many hyperparameter configurations which should not be used to construct a final model (Simon 2007).

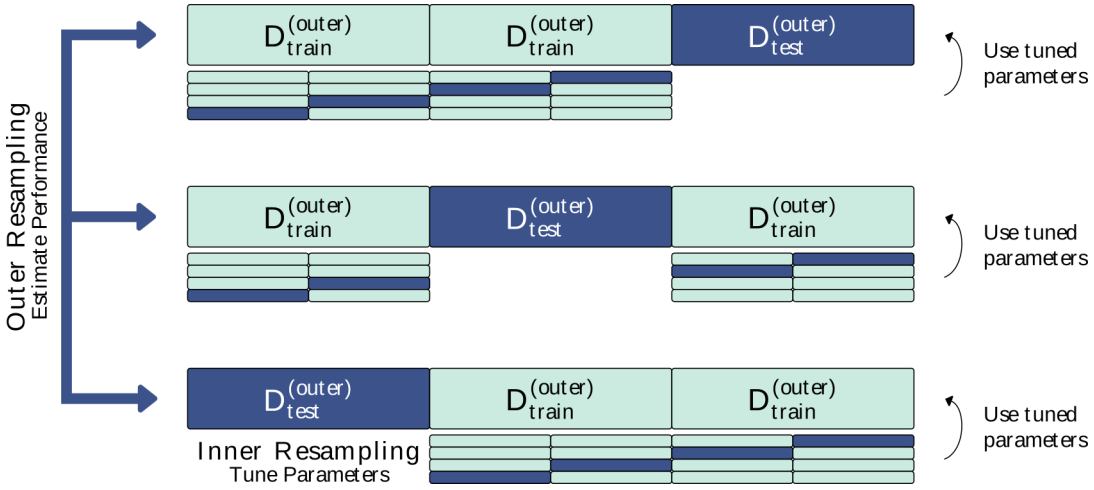


Figure 4.4: An illustration of nested resampling. The large blocks represent 3-fold cross-validation for the outer resampling for model evaluation and the small blocks represent 4-fold cross-validation for the inner resampling for HPO. The light blue blocks are the training sets and the dark blue blocks are the test sets.

An example for nested resampling looks like this:

1. Outer resampling – Instantiate 3-fold cross-validation to create different testing and training data sets.
2. Inner resampling – Within the outer training data instantiate 4-fold cross-validation to create different inner testing and training data sets.
3. HPO – Tune the hyperparameters on the outer training set (large, light blue blocks) using the inner data splits.
4. Training – Fit the learner on the outer training data set using the optimal hyperparameter configuration obtained from the inner resampling (small blocks).
5. Evaluation – Evaluate the performance of the learner on the outer testing data (large, dark blue block).
6. Cross-validation – Repeat (2)-(5) for each of the three folds.
7. Aggregation – Take the sample mean of the three performance values for an unbiased performance estimate.

The nested resampling loop runs three hyperparameter optimizations in total, one for each outer fold. The inner resampling produces generalization performance estimates based on which a single configuration is chosen to be evaluated on the outer resampling. The outer resampling produces generalization estimates for these optimal configurations. The outer resampling does not produce optimal configurations!

Let us take a look at how this works in mlr3.

4.3.1 Nested Resampling with `AutoTuner`

Nested resampling in mlr3 becomes quite simple with the `AutoTuner` (Section 4.2). We simply specify the inner-resampling and tuning setup with the `AutoTuner` and then pass this to `resample()` or `benchmark()`. Continuing with our previous example, we will use the auto-tuner to resample a support vector classifier with 3-fold cross-validation in the outer-resampling and 4-fold cross-validation in the inner resampling.

```
learner = lrn("classif.svm",
  cost  = to_tune(1e-5, 1e5, logscale = TRUE),
  gamma = to_tune(1e-5, 1e5, logscale = TRUE),
  kernel = "radial",
  type = "C-classification"
)

at = auto_tuner(
  tuner = tnr("grid_search", resolution = 5, batch_size = 10),
  learner = learner,
  resampling = rsmp("cv", folds = 4),
  measure = msr("classif.ce"),
)

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

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

rr
```

```
<ResampleResult> with 3 resampling iterations
 task_id        learner_id resampling_id iteration warnings errors
   sonar classif.svm.tuned            cv         1        0      0
   sonar classif.svm.tuned            cv         2        0      0
   sonar classif.svm.tuned            cv         3        0      0
```

Note that we set `store_models = TRUE` so that the `AutoTuner` models are stored to make it possible to investigate the inner tuning. In this example, we utilized the same resampling strategy (K-fold cross-validation), but the mlr3 infrastructure is not limited to this, you can freely combine different inner and outer resampling strategies as you choose. You can also mix-and-match parallelization methods for the process (Section 8.1.4).

There are some special functions for nested resampling available in addition to the methods described in Section 3.2.

The `extract_inner_tuning_results()` and `extract_inner_tuning_archives()` functions return the optimal configurations (across all outer folds) and full tuning archives, respectively.

```
extract_inner_tuning_results(rr)[,
  .(iteration, cost, gamma, classif.ce)]
```

```
   iteration      cost     gamma classif.ce
1:         1  5.756463 -5.756463  0.1449580
2:         2 11.512925 -5.756463  0.1869748
3:         3 11.512925 -5.756463  0.1945378
```

```
extract_inner_tuning_archives(rr)[,
  .(iteration, cost, gamma, classif.ce)]
```

```
    iteration       cost      gamma classif.ce
 1:         1 -11.512925 -11.512925  0.5575630
 2:         1 -11.512925   0.000000  0.5575630
 3:         1 -11.512925  11.512925  0.5575630
 4:         1  -5.756463   5.756463  0.5575630
 5:         1   0.000000 -11.512925  0.5575630
---
71:         3   5.756463  -5.756463  0.1945378
72:         3   5.756463   5.756463  0.4392857
73:         3  11.512925   0.000000  0.4392857
74:         3  11.512925   5.756463  0.4392857
75:         3  11.512925  11.512925  0.4392857
```

From the optimal results, we observe a trend toward larger `cost` and smaller `gamma` values. However, as we discussed earlier, these values should not be used to fit a final model as the selected hyperparameters might differ between the outer resampling iterations. The reason is that the different resampling splits can have different optimal hyperparameter configurations and these might be different from the optimal configuration on the full data set. To get an optimized hyperparameter configuration and a final model, use the steps in Section 4.1.

4.3.2 Performance comparison

Finally, we will compare the predictive performances estimated on the outer resampling to the inner resampling to gain an understanding of model overfitting and general performance.

```
extract_inner_tuning_results(rr)[,
  .(iteration, cost, gamma, classif.ce)]
```

```
   iteration      cost     gamma classif.ce
1:         1  5.756463 -5.756463  0.1449580
2:         2 11.512925 -5.756463  0.1869748
3:         3 11.512925 -5.756463  0.1945378
```

```
rr$score()[, .(iteration, classif.ce)]
```

```
   iteration classif.ce
1:         1  0.2000000
2:         2  0.1304348
3:         3  0.1449275
```

Significantly lower predictive performances on the outer resampling would indicate that the models with the optimized hyperparameters overfit the data.

It is therefore important to ensure that the estimated performance of a tuned model is reported as the aggregated performance of all outer resampling iterations, which is an unbiased estimate of future model performance.

```
rr$aggregate()
```

```
classif.ce
 0.1584541
```

As a final note, nested resampling is computationally expensive, as a simple example using three outer folds and four inner folds with a grid search of resolution 5 used to tune 2 parameters, results in `3*4*5*5 = 300` iterations of model training/testing. In practice, you may often see closer to three folds used in inner resampling or even holdout, or if you have the resources then we recommend parallelization (Section 8.1).

4.3.3 Detailed Example

We will now demonstrate that nested resampling better estimates the generalization performance of a tuned model. For this, we will tune the hyperparameters of an XGBoost model. We will compare the measured performance while tuning and the performance determined with nested resampling with the true performance of the tuned model. We do not run the code in this section because it takes several minutes to run. The results are given below the code chunks in the text.

```
learner = lrn("classif.xgboost",
  eta               = to_tune(1e-4, 1, logscale = TRUE),
  nrounds           = to_tune(1, 5000),
  max_depth         = to_tune(1, 20),
```

```
  colsample_bytree  = to_tune(1e-1, 1),
  colsample_bylevel = to_tune(1e-1, 1),
  lambda            = to_tune(1e-3, 1e3, logscale = TRUE),
  alpha             = to_tune(1e-3, 1e3, logscale = TRUE),
  subsample         = to_tune(1e-1, 1)
)
```

We use simulated data. This way we can generate as much data as we need. The task generator `tgen("moons")` creates two interleaving half circles ("moons") as a binary classification problem (Figure 4.5). `TaskGenerator` objects are used to simulate data and create tasks of arbitrary size.

```
generator = tgen("moons")
task = generator$generate(n = 100L)
```

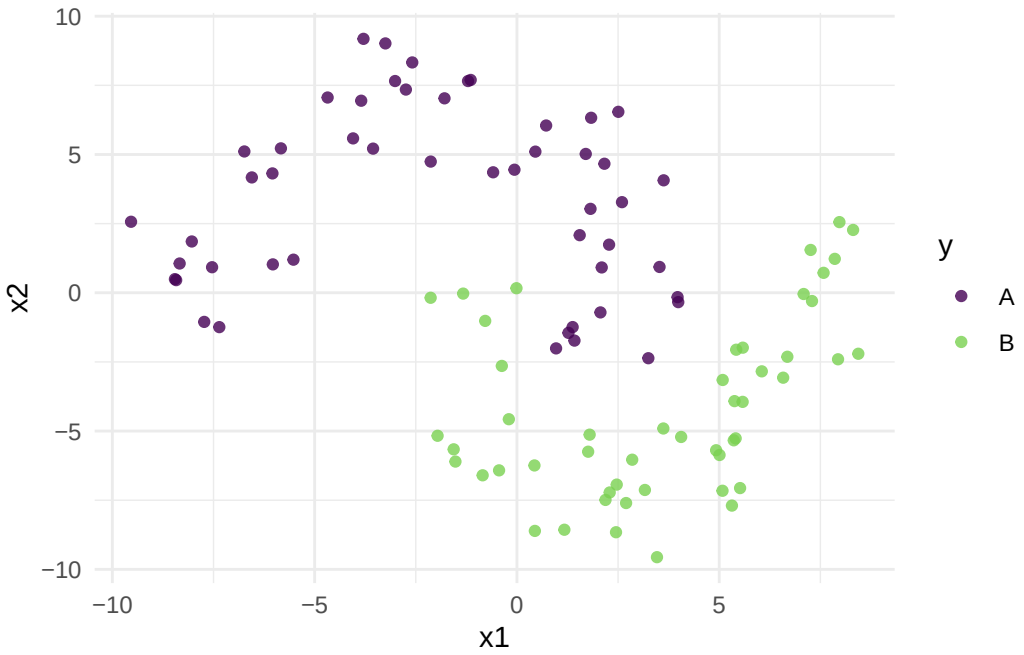


Figure 4.5: Two interleaving half circles ("moons") as a binary classification problem.

We start the tuning using a random search with 1000 evaluations.

```
instance = tune(
  tuner = tnr("random_search", batch_size = 10),
  task = task,
  learner = learner,
  resampling = rsmp("holdout"),
  measures = msr("classif.ce"),
  terminator = trm("evals", n_evals = 1000)
)
```

```
instance$result_y
```

The measured classification error of the best configuration is 9%. We train a model with the best configuration on the entire data set and predict 1 million observations.

```
tuned_learner = lrn("classif.xgboost")
tuned_learner$param_set$set_values(
  .values = instance$result_learner_param_vals)
tuned_learner$train(task)
pred = tuned_learner$predict(generator$generate(n = 1000000))
pred$score()
```

The classification error of 11% is the true generalization error of the tuned model. We see that the measured performance overestimates the performance of the tuned model.

Let's use nested resampling to estimate the performance of the tuned model.

```
at = auto_tuner(
  tuner = tnr("random_search", batch_size = 10),
  learner = learner,
  resampling = rsmp("holdout"),
  measure = msr("classif.ce"),
  terminator = trm("evals", n_evals = 1000)
)
rr = resample(task, at, rsmp("cv", folds = 5))
rr$aggregate()
```

Nested resampling estimate a classification error of 10%. The performance estimated by nested resampling is closer to the true performance of the tuned model.

4.4 Advanced Tuning

> **!** **Advanced section**

This section is devoted to advanced tuning techniques. The topics are not necessary for a basic understanding of tuning but make tuning more efficient and robust.

4.4.1 Encapsulation and Fallback Learner

Until now, we have focused on working examples, but to demonstrate encapsulation we will now consider 'broken' examples, e.g., where learners do not converge, run out of memory, or terminate with an error. To get an error in the following examples, we filter the `sonar` task to only one class.

```
task = tsk("sonar")
task$filter(seq(10))
```

Since tuning is an automated process, there is no opportunity for manual intervention, we therefore, mitigate errors by making use of encapsulation. Encapsulation allows errors in training to be isolated and handled, without disrupting the tuning process. In `mlr3`, encapsulation is controlled by passing arguments to the `encapsulate` field in any learner, as in the example below. The possible options are `evaluate` and `callr` named after the packages of the same name.

Encapsulation

```
learner = lrn("classif.svm",
  cost  = to_tune(1e-5, 1e5),
  gamma = to_tune(1e-5, 1e5),
  kernel = "radial",
  type = "C-classification"
)

learner$encapsulate = c(train = "evaluate", predict = "evaluate")
```

Note that encapsulation is set individually for training and predicting. This separation could be useful if, for example, we are happy to ignore errors in training but want to manually debug any errors in prediction. There are currently two options for encapsulating a learner, via the packages `evaluate` and `callr`. `evaluate` catches any errors that occur and allows the process to continue, whereas `callr` encapsulation spawns a separate R process (thus comes with more computational overhead) (see Section 8.2.1). This way `callr` also guards against segmentation faults which would tear down the complete R session. Both packages allow setting a timeout, which is useful when a learner does not converge. You can do this by setting the `timeout` field. Again this can be set for training and predicting individually:

```
learner$timeout = c(train = 30, predict = 30)
```

With encapsulation, exceptions and timeouts do not stop the tuning. Instead, the error message is recorded and a fallback learner is fitted.

Fallback learners allow scoring a result when no model was fitted during training. A common approach is to predict a weak baseline e.g. predicting the mean of the target (`lrn("regr.featureless")`), or the majority class (`lrn("classif.featureless")`). See Section 8.2.2 for more detailed information.

Below we set `mlr_learners_classif.featureless` as the featureless learner which always predicts the most frequent label.

```
learner$fallback = lrn("classif.featureless")
```

Errors and warnings that occurred during tuning are stored in the archive.

```
instance = tune(
  tuner = tnr("random_search", batch_size = 5),
  task = task,
  learner = learner,
```

```
  resampling = rsmp("holdout"),
  measures = msr("classif.ce"),
  term_evals = 10
)

as.data.table(instance$archive)[, .(cost, gamma, classif.ce, errors, warnings)]
```

```
        cost     gamma classif.ce errors warnings
 1: 50874.85 21806.528          0      1        0
 2: 40494.70 57489.455          0      1        0
 3: 42451.39  4065.314          0      1        0
 4: 91396.58 54059.751          0      1        0
 5: 90200.65 11282.846          0      1        0
 6: 20411.44 72762.945          0      1        0
 7: 76770.38 75284.619          0      1        0
 8: 42234.47 60073.895          0      1        0
 9: 54695.40 10106.674          0      1        0
10: 88051.88 82622.271          0      1        0
```

> **Note**
>
> When tuning, encapsulation should always be combined with a fallback learner, because the archive must contain a performance value for each configuration.

4.4.2 Memory Management

Running a large tuning experiment requires a lot of working memory, especially when using nested resampling. Most of the memory is consumed by the models since each resampling iteration creates one new model. The option `store_models` in the functions `ti()` and `auto_tuner()` allows us to enable the storage of the models. Storing the models is disabled by default and in most cases, it is not necessary to save the models.

The archive stores a `ResampleResult` for each evaluated hyperparameter configuration. The contained `Prediction` objects can take up a lot of memory, especially with large data sets and many resampling iterations. We can disable the storage of the resample results by setting `store_benchmark_result = FALSE` in the functions `ti()` and `auto_tuner()`. Note that without the resample results, it is no longer possible to score the configurations on another measure.

When we run nested resampling with many outer resampling iterations, additional memory can be saved if we set `store_tuning_instance = FALSE` in the `auto_tuner()` function. The functions `extract_inner_tuning_results()` and `extract_inner_tuning_archives()` will then no longer work.

The option `store_models = TRUE` sets `store_benchmark_result` and `store_tuning_instance` to `TRUE` because the models are stored in the benchmark results which in turn is part of the instance. This also means that `store_benchmark_result = TRUE` sets `store_tuning_instance` to `TRUE`.

Finally, we can set `store_models = FALSE` in the `resample()` or `benchmark()` functions to disable the storage of the auto tuners when running nested resampling. This way we can

still access the aggregated performance (`rr$aggregate()`) but do not have any information about the inner resampling anymore.

4.5 Defining Search Spaces

In this section, we will cover more advanced techniques for defining search spaces.

> **!** **Advanced section**

4.5.1 Defining Search Spaces from Scratch

In Section 4.1.3 we have seen how one can conveniently define the tuning space of a learner using a `TuneToken` that can be constructed with the `to_tune()` function. In this section, we will show how to define such search spaces from scratch and also cover more advanced techniques. To start, we will revisit an example from earlier, where we have tuned the `cost` and `gamma` parameters of an SVM classifier.

```
learner = lrn("classif.svm",
  cost  = to_tune(1e-1, 1e5),
  gamma = to_tune(1e-1, 1),
  kernel = "radial",
  type = "C-classification"
)
```

When an auto tuner is created from such a learner, the search space is automatically constructed from the tune tokens. This search space is an object of class `ParamSet`.

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

```
<ParamSet>
      id    class lower upper nlevels        default value
1:  cost ParamDbl   0.1 1e+05     Inf <NoDefault[3]>
2: gamma ParamDbl   0.1 1e+00     Inf <NoDefault[3]>
```

This object can also be created "by hand" and passed explicitly as the `search_space` argument of the `auto_tuner`. The `TuneToken` is merely a user-friendly mechanism that allows for a more convenient definition of search spaces.

We can define the search space from above by calling `paradox::ps()`. The function takes named `paradox::Domain` arguments and creates a `paradox::ParamSet` from them.

In order to define this search space, we have to pick the right type for each parameter. As of writing this book, there are five domain constructors that produce different parameters when passed to `ps()`.

Constructor	Description	Underlying Class
`p_dbl`	Real valued parameter ("double")	`ParamDbl`

Constructor	Description	Underlying Class
`p_int`	Integer parameter	`ParamInt`
`p_fct`	Discrete valued parameter ("factor")	`ParamFct`
`p_lgl`	Logical / Boolean parameter	`ParamLgl`
`p_uty`	Untyped parameter	`ParamUty`

Table 4.3: `Domain` Constructors and their resulting `Param`.

For the purpose of defining search spaces, the relevant arguments of the domain constructors fall into one of three categories:

1. Define the **range of values** over which to tune; these are e.g. `lower` and `upper` for `p_int` and `p_dbl` or `levels` for `p_fct`.
2. **Dependencies** between parameters via the argument `depends`. It is an expression that must involve other parameters and be of the form `<param> == <scalar>`, `<param> %in% <vector>`, or multiple of these chained by `&&`.
3. Parameter **transformations** via a `trafo`. This allows to modify the sampling distribution.

We can recreate the search space from earlier using the `p_dbl()` function.

```
search_space = ps(
  cost  = p_dbl(lower = 1e-1, upper = 1e5),
  gamma = p_dbl(lower = 1e-1, upper = 1)
)

search_space
```

```
<ParamSet>
      id    class lower upper nlevels        default value
1:  cost ParamDbl   0.1 1e+05     Inf <NoDefault[3]>
2: gamma ParamDbl   0.1 1e+00     Inf <NoDefault[3]>
```

> **Note**
>
> Because the `ParamSet` class is also used to represent the hyperparameter spaces of objects like learners or pipeops (see Chapter 6), the domain constructors also have other arguments. These will be covered in Section 8.5.

When creating a search space, one has to ensure that the resulting space is bounded. A search space is bounded if all parameters are bounded:

1. `ParamFct` and `ParamLgl` are always bounded
2. `ParamInt` and `ParamDbl` are bounded if `lower` and `upper` are finite
3. `ParamUty` is never bounded.

One can check whether a `ParamSet` (or `Param`) is bounded by accessing the `$is_bounded` field.

```
ps(cost = p_dbl(lower = 0.1, upper = 1))$is_bounded
```

```
[1] TRUE
```

```
ps(cost = p_dbl(lower = 0.1, upper = Inf))$is_bounded
```

```
[1] FALSE
```

Creating a search space with an unbounded parameter will result in an error since the tuner cannot sample from an unbounded space.

As a second example, we define a search space in which we search the optimal `cost` parameter in the range $[0.1, 1]$ and the best kernel of all values `"polynomial"` and `"radial"`.

```
search_space = ps(
  cost   = p_dbl(lower = 0.1, upper = 1),
  kernel = p_fct(levels = c("polynomial", "radial"))
)
search_space
```

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

It is also possible to specify dependencies between parameters. The SVM, for example, has the `degree` parameter that is only valid when the `kernel` is `"polynomial"`. We can specify this constraint by using the `depends` argument of the domain constructor. To tune the `degree` parameter, one would need to do the following:

```
search_space = ps(
  cost   = p_dbl(0, 1),
  kernel = p_fct(c("polynomial", "radial")),
  degree = p_int(1, 3, depends = (kernel == "polynomial"))
)
```

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

```
search_space = ps(
  cost   = p_dbl(-1, 1, trafo = function(x) exp(x)),
  kernel = p_fct(c("polynomial", "radial"))
)
```

Figure 4.6 visualizes the resulting design.

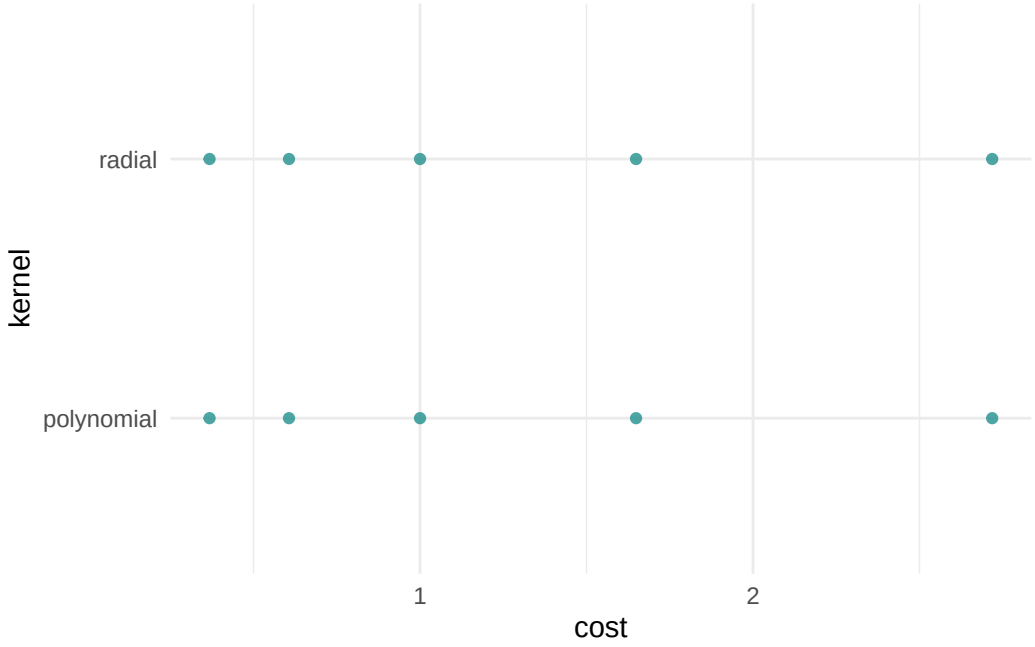


Figure 4.6: Design points from a grid search when tuning an SVM. The resolution is 5 and the cost parameter on a logarithmic scale.

> **Tip**
>
> Because the log-scale transformation is so common, the domain constructors `p_int()` and `p_dbl()` have a flag `logscale` that can be set to apply a logarithmic transformation (see Section 4.1.6).

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

In our example we now assume that the parameter `cost` should be set to a higher value when the `kernel` is `"polynomial"`.

```
search_space = ps(
  cost = p_dbl(-1, 1, trafo = function(x) exp(x)),
  kernel = p_fct(c("polynomial", "radial")),
  .extra_trafo = function(x, param_set) {
    if (x$kernel == "polynomial") {
      x$cost = x$cost + 2
    }
    x
```

```
    }
)
```

An exemplary design is depicted in Figure 4.7.

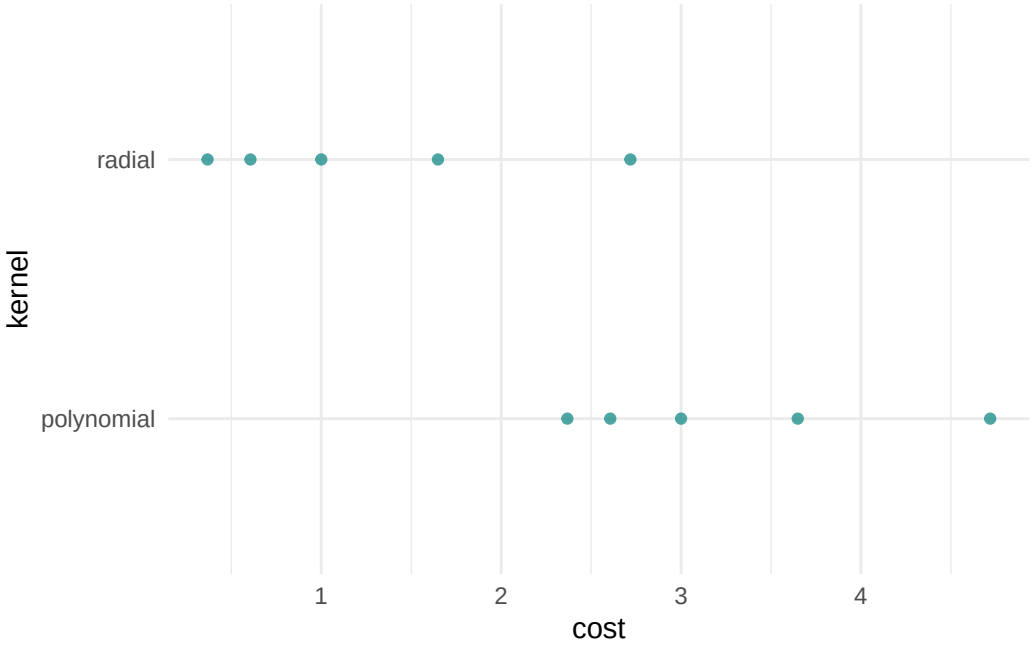


Figure 4.7: Design grid for tuning a SVM. The resolution is 5, the cost parameter is logarithmically transformed when points with a `kernel` equal to `"polynomial"` are shifted to the right by a value of 2.

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

An example is the `class.weights` parameter of the SVM, which takes a named vector of class weights with one entry for each target class. The transformation that would tune `class.weights` for the `sonar` data set could be:

```
search_space = ps(
  class.weights = p_dbl(lower = 0.1, upper = 0.9,
    trafo = function(x) c(M = x, R = 1 - x))
)
```

A common use-case is the necessity to specify a list of values that should all be tried (or sampled from). It may be the case that a hyperparameter accepts function objects as values and a certain list of functions should be tried. Or it may be that a choice of special numeric values should be tried. For this, the `p_fct` constructor's `level` argument may be a value

that is not a `character` vector, but something else. If, for example, only the values 0.1, 3, and 10 should be tried for the `cost` parameter, even when doing random search, then the following search space would achieve that:

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

This is equivalent to the following:

```
search_space = ps(
  cost   = p_fct(c("0.1", "0.3", "10"),
    trafo = function(x) list(`0.1` = 0.1, `3` = 3, `10` = 10)[[x]]),
  kernel = p_fct(c("polynomial", "radial"))
)
```

This makes sense when considering that factorial tuning parameters are always `character` values:

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

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

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

4.5.2 Creating Search Spaces from Learners

In Section 4.1.3 we have seen how one can conveniently define the tuning space of a learner using a `TuneToken`. We will now show how some of the advanced features from Section 4.5.1 can also be used through this mechanism.

A `TuneToken` can also be constructed with a `Domain` object, i.e. something constructed with a `p_<type>` call. This makes it possible to also specify dependencies and transformations of parameters. Dependencies are usually taken from the `ParamSet` of the learner, so we don't need to specify them. But being able to set a dependency with a `TuneToken` is helpful when we work with pipelines (see Chapter 6). However, to keep the example simple, let's show it here again on an SVM.

```
learner = lrn("classif.svm",
  kernel  = to_tune(c("linear", "polynomial")),
  gamma   = to_tune(p_dbl(1e-4, 1e4, depends = kernel == "polynomial"))
)

learner$param_set$search_space()
```

```
<ParamSet>
       id    class lower upper nlevels        default       parents value
1:  gamma ParamDbl 1e-04 10000     Inf <NoDefault[3]> kernel,kernel
2: kernel ParamFct    NA    NA       2 <NoDefault[3]>
```

Setting up a transformation works similarly.

```
learner = lrn("classif.svm",
  cost = to_tune(p_dbl(-1, 1, trafo = function(x) exp(x)))
)
```

It is even possible to define whole ParamSets that get tuned over for a single parameter. This may be especially useful for vector hyperparameters that should be searched along multiple dimensions. In this special case, the `.extra_trafo` must return a list with a single element, as it corresponds to a single hyperparameter that is being tuned. Suppose the `class.weights` hyperparameter should be tuned along two dimensions:

```
par = ps(
  M = p_dbl(0.1, 0.9),
  R = p_dbl(0.1, 0.9),
  .extra_trafo = function(x, param_set) {
    list(c(M = x$M, R = x$R))
  }
)
learner$param_set$set_values(class.weights = to_tune(par))
learner$param_set$search_space()
```

```
<ParamSet>
     id    class lower upper nlevels        default value
1: cost ParamDbl  -1.0   1.0     Inf <NoDefault[3]>
2:    M ParamDbl   0.1   0.9     Inf <NoDefault[3]>
3:    R ParamDbl   0.1   0.9     Inf <NoDefault[3]>
Trafo is set.
```

> **Tip**
>
> The `.extra_trafo` from `par` parameter set only has access to the parameters `M` and `R` and not the other parameters from the learner's search space.

4.5.3 Recommended Search Spaces

Selected search spaces can require a lot of background knowledge or expertise. The package `mlr3tuningspaces` tries to make HPO more accessible by providing implementations of

published search spaces for many popular machine learning algorithms. These search spaces should be applicable to a wide range of data sets, however, they may need to be adapted in specific situations. The search spaces are stored in the dictionary `mlr_tuning_spaces`.

```
as.data.table(mlr_tuning_spaces)
```

```
                      key                                  label        learner
 1: classif.glmnet.default    Classification GLM with Default classif.glmnet
 2:    classif.glmnet.rbv1  Classification GLM with RandomBot classif.glmnet
 3:    classif.glmnet.rbv2  Classification GLM with RandomBot classif.glmnet
 4:   classif.kknn.default   Classification KKNN with Default   classif.kknn
 5:      classif.kknn.rbv1 Classification KKNN with RandomBot   classif.kknn
---
32:          regr.svm.rbv1      Regression SVM with RandomBot       regr.svm
33:          regr.svm.rbv2      Regression SVM with RandomBot       regr.svm
34:   regr.xgboost.default    Regression XGBoost with Default   regr.xgboost
35:      regr.xgboost.rbv1  Regression XGBoost with RandomBot   regr.xgboost
36:      regr.xgboost.rbv2  Regression XGBoost with RandomBot   regr.xgboost
1 variable not shown: [n_values]
```

The tuning spaces are named according to the scheme `{learner-id}.{tuning-space-id}`. The `default` tuning spaces are published in Bischl et al. (2021). More tuning spaces are part of the random bot experiments `rbv1` and `rbv2` that are published in Kuehn et al. (2018) and Binder, Pfisterer, and Bischl (2020). The sugar function `lts()` is used to retrieve a `TuningSpace`.

```
lts("classif.rpart.default")
```

```
<TuningSpace:classif.rpart.default>: Classification Rpart with Default
          id lower upper levels logscale
1:  minsplit 2e+00 128.0           TRUE
2: minbucket 1e+00  64.0           TRUE
3:        cp 1e-04   0.1           TRUE
```

A tuning space can be passed to `ti()` as the `search_space`.

```
instance = ti(
  task = tsk("sonar"),
  learner = lrn("classif.rpart"),
  resampling = rsmp("cv", folds = 3),
  measures = msr("classif.ce"),
  terminator = trm("evals", n_evals = 20),
  search_space = lts("classif.rpart.rbv2")
)
```

Alternatively, we can explicitly set the search space of a learner with `TuneTokens`

```
vals = lts("classif.rpart.default")$values
vals[1]
```

```
$minsplit
```

```
Tuning over:
range [2, 128] (log scale)
```

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

When passing a learner to `lts()`, the default search space from the Bischl et al. (2021) article is applied.

```
lts(lrn("classif.rpart"))
```

```
<LearnerClassifRpart:classif.rpart>: Classification Tree
* Model: -
* Parameters: xval=0, minsplit=<RangeTuneToken>,
  minbucket=<RangeTuneToken>, cp=<RangeTuneToken>
* Packages: mlr3, rpart
* Predict Types:  [response], prob
* Feature Types: logical, integer, numeric, factor, ordered
* Properties: importance, missings, multiclass, selected_features,
  twoclass, weights
```

It is possible to simply overwrite a predefined tuning space in construction, for example here we change the range of the `maxdepth` hyperparameter in rpart:

```
lts("classif.rpart.rbv2", maxdepth = to_tune(1, 20))
```

```
<TuningSpace:classif.rpart.rbv2>: Classification Rpart with RandomBot
          id lower upper levels logscale
1:        cp 1e-04     1            TRUE
2:  maxdepth 1e+00    20           FALSE
3: minbucket 1e+00   100           FALSE
4:  minsplit 1e+00   100           FALSE
```

4.6 Multi-Fidelity Tuning via Hyperband

Increasingly large data sets and search spaces and costly to train models make hyperparameter optimization a time-consuming task. Recent HPO methods often also make use of evaluating a configuration at multiple so-called fidelity levels. For example, a neural network can be trained for an increasing number of epochs, gradient boosting can be performed for an increasing number of boosting iterations and training data can always be subsampled to a smaller fraction of all available data. The general idea of so-called *multi-fidelity* HPO is that the performance of a model obtained by using computationally cheap lower fidelity evaluation (few numbers of epochs or boosting iterations, only using a small sample of all available data for training) is predictive of the performance of the model obtained using computationally expensive full model evaluation and this concept can be leveraged to make HPO more efficient (e.g., only continuing to evaluate those configurations on higher fidelities that appear to be promising with respect to their performance). The fidelity parameter

is part of the search space and controls the trade-off between the runtime and preciseness of the performance approximation.

A popular multi-fidelity HPO algorithm is given by *Hyperband* (Li et al. 2018). After having evaluated randomly sampled configurations on low fidelities, Hyperband iteratively allocates more resources to promising configurations and terminates low-performing ones. In the following example, we will optimize XGBoost and use the number of boosting iterations as the fidelity parameter. This means Hyperband will allocate increasingly more boosting iterations to well-performing hyperparameter configurations. Increasing the number of boosting iterations increases the time to train a model but generally also the performance. It is therefore a suitable fidelity parameter. However, as already mentioned, Hyperband is not limited to machine learning algorithms that are trained iteratively. In the second example, we will tune a support vector machine and use the size of the training data as the fidelity parameter. Some prior knowledge about pipelines (Chapter 6) is beneficial but not necessary to fully understand the examples. In the following, the terms fidelity and budget are often used interchangeably.

4.6.1 Hyperband Tuner

Hyperband (Li et al. 2018) builds upon the Successive Halving algorithm by Jamieson and Talwalkar (2016). Successive Halving is initialized with the number of starting configurations n, the proportion of configurations discarded in each stage η, and the minimum r_{min} and maximum r_{max} budget of a single evaluation. The algorithm starts by sampling n random configurations and allocating the minimum budget r_{min} to them. The configurations are evaluated and $\frac{1}{\eta}$ of the worst-performing configurations are discarded. The remaining configurations are promoted to the next stage and evaluated on a larger budget. This continues until one or more configurations are evaluated on the maximum budget r_{max} and the best-performing configuration is selected. The total number of stages is calculated so that each stage consumes approximately the same overall budget. Successive Halving has the disadvantage that is not clear whether we should choose a large n and try many configurations on a small budget or choose a small n and train more configurations on the full budget.

Hyperband solves this problem by running Successive Halving with different numbers of stating configurations starting on different budget levels. The algorithm is initialized with the same parameters as Successive Halving except for n. Each run of Successive Halving is called a bracket and starts with a different budget r_0. A smaller starting budget means that more configurations can be evaluated. The most exploratory bracket is allocated the minimum budget r_{min}. The next bracket increases the starting budget by a factor of η. In each bracket, the starting budget increases further until the last bracket $s = 0$ essentially performs a random search with the full budget r_{max}. The number of brackets $s_{max} + 1$ is calculated with $s_{max} = \log_\eta \frac{r_{max}}{r_{min}}$. Under the condition that r_0 increases by η with each bracket, r_{min} sometimes has to be adjusted slightly in order not to use more than r_{max} resources in the last bracket. The number of configurations in the base stages is calculated so that each bracket uses approximately the same amount of budget. Table 4.4 shows a full run of the Hyperband algorithm. The bracket $s = 3$ is the most exploratory bracket and $s = 0$ essentially performs a random search using the full budget.

	$s=3$		$s=2$		$s=1$		$s=0$	
i	n_i	r_i	n_i	r_i	n_i	r_i	n_i	r_i
0	8	1	6	2	4	4	4	8

	$s = 3$		$s = 2$		$s = 1$		$s = 0$	
i	n_i	r_i	n_i	r_i	n_i	r_i	n_i	r_i
1	4	2	3	4	2	8		
2	2	4	1	8				
3	1	8						

Table 4.4: Hyperband schedule with the number of configurations n_i and resources r_i for each bracket s and stage i, when $\eta = 2$, $r_{min} = 1$ and $r_{max} = 8$

4.6.2 Example XGBoost

In this practical example, we will optimize the hyperparameters of XGBoost on the `spam` data set. We begin by constructing the learner.

```
library(mlr3hyperband)
```

```
Loading required package: mlr3tuning
```

```
Loading required package: paradox
```

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

As the next step we define the search space. The `nrounds` parameter controls the number of boosting iterations. We specify a range from 16 to 128 boosting iterations. This is used as r_{min} and r_{max} within Hyperband. We need to tag the parameter with `"budget"`to identify it as a fidelity parameter. For the other hyperparameters, we take the search space for XGBoost from Bischl et al. (2021). This search space usually work well for a wide range of data sets.

```
learner$param_set$set_values(
  nrounds           = to_tune(p_int(16, 128, tags = "budget")),
  eta               = to_tune(1e-4, 1, logscale = TRUE),
  max_depth         = to_tune(1, 20),
  colsample_bytree  = to_tune(1e-1, 1),
  colsample_bylevel = to_tune(1e-1, 1),
  lambda            = to_tune(1e-3, 1e3, logscale = TRUE),
  alpha             = to_tune(1e-3, 1e3, logscale = TRUE),
  subsample         = to_tune(1e-1, 1)
)
```

We proceed to construct the tuning instance. Note that we use `trm("none")` because Hyperband terminates itself after all brackets have been evaluated.

```
instance = ti(
  task = tsk("spam"),
  learner = learner,
  resampling = rsmp("holdout"),
  measures = msr("classif.ce"),
  terminator = trm("none")
)
```

```
)
```

We then construct the Hyperband tuner and specify `eta = 2`. In general, Hyperband can start all over from the beginning once the last bracket is evaluated. We control the number of Hyperband runs with the `repetition` argument. The setting `repetition = Inf` is useful when a terminator should stop the optimization, for example based on runtime.

```
tuner = tnr("hyperband", eta = 2, repetitions = 1)
```

Using `eta = 2` and 16 to 128 boosting iterations results in the following schedule. This only prints a data table with the schedule and does not modify the tuner.

```
hyperband_schedule(r_min = 16, r_max = 128, eta = 2)
```

```
    bracket stage budget n
 1:       3     0     16 8
 2:       3     1     32 4
 3:       3     2     64 2
 4:       3     3    128 1
 5:       2     0     32 6
 6:       2     1     64 3
 7:       2     2    128 1
 8:       1     0     64 4
 9:       1     1    128 2
10:       0     0    128 4
```

We can now proceed with the tuning:

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

The result is the configuration with the best performance.

```
instance$result[, .(classif.ce, nrounds)]
```

```
   classif.ce nrounds
1: 0.04563233     128
```

Note that the archive resulting of a Hyperband run contains the additional columns `bracket` and `stage`:

```
as.data.table(instance$archive)[,
  .(bracket, stage, classif.ce, eta, max_depth, colsample_bytree)]
```

```
    bracket stage classif.ce        eta max_depth colsample_bytree
 1:       3     0 0.35136897 -5.3546638        10        0.2386405
 2:       3     0 0.05801825 -3.9328372        17        0.8640443
 3:       3     0 0.08344198 -7.9741309        20        0.4712803
 4:       3     0 0.12646675 -7.1203222        13        0.2451142
 5:       3     0 0.49022164 -6.5252608        11        0.8071411
---
```

```
31:       0     0 0.06127771 -0.1248958        14        0.4152922
32:       3     3 0.05345502 -3.9328372        17        0.8640443
33:       2     2 0.04758801 -1.8496761        10        0.7461747
34:       1     1 0.05606258 -1.8823930         8        0.7920147
35:       1     1 0.07496741 -7.0610415        11        0.2516539
```

4.6.3 Example Support Vector Machine

In this example, we will optimize the hyperparameters of a support vector machine on the `sonar` data set. We begin by constructing the learner and setting `type` to `"C-classification"`.

```
learner = lrn("classif.svm", id = "svm", type = "C-classification")
```

The `mlr3pipelines` package features a `PipeOp` for subsampling data. This will be helpful when using the size of the training data as a fidelity parameter.

```
po("subsample")
```

```
PipeOp: <subsample> (not trained)
values: <frac=0.6321, stratify=FALSE, replace=FALSE>
Input channels <name [train type, predict type]>:
  input [Task,Task]
Output channels <name [train type, predict type]>:
  output [Task,Task]
```

This pipeline operator controls the size of the training data set with the `frac` parameter. We connect the `po("subsample")` with the learner and get a `GraphLearner`.

```
graph_learner = as_learner(
  po("subsample") %>>%
  learner
)
```

The graph learner subsamples and then fits a support vector machine on the data subset. The parameter set of the graph learner is a combination of the parameter sets of the pipeline operator and learner.

```
as.data.table(graph_learner$param_set)[, .(id, lower, upper, levels)]
```

```
                   id lower upper                              levels
 1:      subsample.frac     0   Inf
 2:  subsample.stratify    NA    NA                       TRUE,FALSE
 3:   subsample.replace    NA    NA                       TRUE,FALSE
 4:       svm.cachesize  -Inf   Inf
 5:   svm.class.weights    NA    NA
 6:            svm.coef0  -Inf   Inf
 7:             svm.cost     0   Inf
 8:            svm.cross     0   Inf
 9: svm.decision.values    NA    NA                       TRUE,FALSE
```

```
10:          svm.degree     1   Inf
11:         svm.epsilon     0   Inf
12:          svm.fitted    NA    NA                          TRUE,FALSE
13:           svm.gamma     0   Inf
14:          svm.kernel    NA    NA   linear,polynomial,radial,sigmoid
15:              svm.nu  -Inf   Inf
16:           svm.scale    NA    NA
17:       svm.shrinking    NA    NA                          TRUE,FALSE
18:       svm.tolerance     0   Inf
19:            svm.type    NA    NA C-classification,nu-classification
```

Next, we create the search space. We have to prefix the hyperparameters with the id of the pipeline operators, because this reflects the way how they are represented in the parameter set of the graph learner. The `subsample.frac` is the fidelity parameter that must be tagged with `"budget"` in the search space. In the following, the data set size is increased from 3.7% to 100%. For the other hyperparameters, we take the search space for support vector machines from Binder, Pfisterer, and Bischl (2020). This search space usually work well for a wide range of data sets.

```
graph_learner$param_set$set_values(
  subsample.frac  = to_tune(p_dbl(3^-3, 1, tags = "budget")),
  svm.kernel      = to_tune(c("linear", "polynomial", "radial")),
  svm.cost        = to_tune(1e-4, 1e3, logscale = TRUE),
  svm.gamma       = to_tune(1e-4, 1e3, logscale = TRUE),
  svm.tolerance   = to_tune(1e-4, 2, logscale = TRUE),
  svm.degree      = to_tune(2, 5)
)
```

Support vector machines can often crash during training or take an extensive time to train given certain hyperparameters. We therefore set a timeout of 30 seconds and specify a fallback learner (Section 4.4.1) to handle these cases.

```
graph_learner$encapsulate = c(train = "evaluate", predict = "evaluate")
graph_learner$timeout = c(train = 30, predict = 30)
graph_learner$fallback = lrn("classif.featureless")
```

Let's create the tuning instance. Again, we use `trm("none")` because Hyperband controls the termination itself.

```
instance = ti(
  task = tsk("sonar"),
  learner = graph_learner,
  resampling = rsmp("cv", folds = 3),
  measures = msr("classif.ce"),
  terminator = trm("none")
)
```

We create the tuner and set `eta = 3`.

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

Using `eta = 3` and a lower bound of 3.7% for the data set size results in the following Hyperband schedule.

```
hyperband_schedule(r_min = 3^-3, r_max = 1, eta = 3)
```

```
    bracket stage     budget  n
 1:       3     0 0.03703704 27
 2:       3     1 0.11111111  9
 3:       3     2 0.33333333  3
 4:       3     3 1.00000000  1
 5:       2     0 0.11111111 12
 6:       2     1 0.33333333  4
 7:       2     2 1.00000000  1
 8:       1     0 0.33333333  6
 9:       1     1 1.00000000  2
10:       0     0 1.00000000  4
```

We can now start the tuning.

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

We observe that the best model is a support vector machine with a polynomial kernel.

```
instance$result[, .(classif.ce, subsample.frac, svm.kernel)]
```

```
   classif.ce subsample.frac svm.kernel
1:  0.1536232              1 polynomial
```

The archive contains all evaluated configurations. We can proceed to further investigate the 8 configurations that were evaluated on the full data set. The configuration with the best classification error on the full data set was sampled in the second bracket. The classification error was estimated to be 30% on 33% of the data set and decreased to 14% on the full data set (see bright purple line in Figure 4.8).

4.7 Multi-Objective Tuning

Multi-objective

So far we have considered optimizing a model with respect to one metric, but multi-metric, or multi-objective optimization, is also possible. A simple example of multi-objective optimization might be optimizing a classifier to minimize false positive and false negative predictions. As a more complex example, consider the problem of deploying a classifier in a healthcare setting. One the one hand, we are usually interested in using a model that makes the best possible predictions. On the other hand, strong performing models such as deep neural networks or gradient boosted trees with many boosting iterations often can be very hard to interpret (e.g., understanding how a model arrives at a certain prediction can be difficult). Especially in healthcare applications interpretability or model complexity can be highly relevant (often a model is more likely to be trusted and adapted in practice if it can provide insight into its reasoning) and in this case, we may be interested in minimizing *both* the classification error and the complexity of the model.

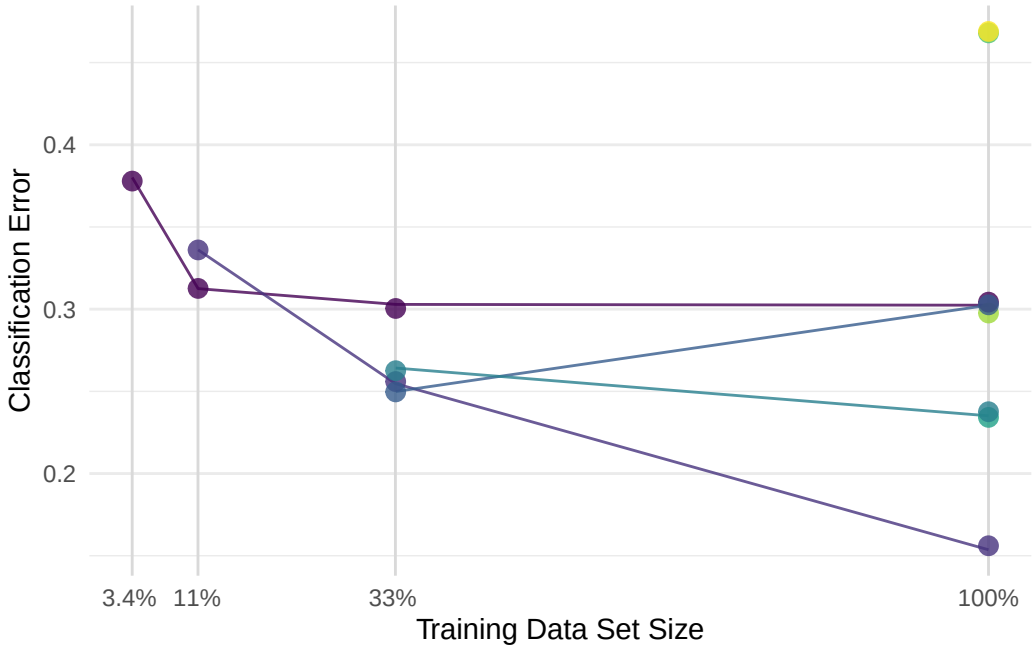


Figure 4.8: Optimization paths of the 8 configurations evaluated on the complete data set.

In general, when optimizing multiple metrics, these will be in competition (if they were not we would only need to optimize with respect to one of them) and so no single configuration exists that optimizes all metrics. Focus is therefore given to the concept of Pareto optimality. One hyperparameter configuration is said to Pareto-dominate another one if the resulting model is equal or better in all metrics and strictly better in at least one metric. All configurations that are not Pareto-dominated by any other configuration are called Pareto efficient and the set of all these configurations is the Pareto set. In contrast, the metric values coresponding to these non-dominated confgurations are referred to as the Pareto front.

Pareto Set

Pareto Front

The goal of multi-objective HPO is to approximate the true, unknown Pareto front. More methodological details on multi-objective HPO can be found in Karl et al. (2022).

We will now demonstrate multi-objective HPO by tuning a decision tree on the `sonar` data set with respect to the classification error, as a measure of model performance, and the number of selected features, as a measure of model complexity (in a decision tree the number of selected features is straightforward to obtain by simply counting the number of unique splitting variables). We will tune

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

```
learner = lrn("classif.rpart",
  cp = to_tune(1e-04, 1e-1),
  minsplit = to_tune(2, 64),
  maxdepth = to_tune(1, 30)
)

measures = msrs(c("classif.ce", "selected_features"))
```

Note that as we tune with respect to multiple measures, the function `ti` creates a `TuningInstanceMultiCrit` instead of a `TuningInstanceSingleCrit`. We also have to set `store_models = TRUE` because this is required by the selected features measure.

```
instance = ti(
  task = tsk("sonar"),
  learner = learner,
  resampling = rsmp("cv", folds = 5),
  measures = measures,
  terminator = trm("evals", n_evals = 30),
  store_models = TRUE
)
instance
```

```
<TuningInstanceMultiCrit>
* State:  Not optimized
* Objective: <ObjectiveTuning:classif.rpart_on_sonar>
* Search Space:
         id    class lower upper nlevels
1:       cp ParamDbl 1e-04   0.1     Inf
2: minsplit ParamInt 2e+00  64.0      63
3: maxdepth ParamInt 1e+00  30.0      30
* Terminator: <TerminatorEvals>
```

As before we will then select and run a tuning algorithm. Here we use a random search:

```
tuner = tnr("random_search", batch_size = 30)
tuner$optimize(instance)
```

Finally, we inspect the best-performing configurations, i.e., the Pareto set, and visualize the corresponding estimated Pareto front (Figure 4.9). Note that the number of selected features can be fractional, as in this example, it is determined through resampling and calculated as an average across the number of selected features per cross-validation fold.

```
instance$archive$best()[, .(cp, minsplit, maxdepth, classif.ce, selected_features)]
```

```
             cp minsplit maxdepth classif.ce selected_features
1: 0.0494610118       19       23  0.2407666              5.4
2: 0.0155207114       54        1  0.2599303              1.0
3: 0.0990043073       26        7  0.2599303              1.0
4: 0.0297009104       22       25  0.2598142              4.4
5: 0.0137808275       21       10  0.2550523              4.8
```

```
6: 0.0001712655       29         1  0.2599303                1.0
7: 0.0895633622       60         4  0.2599303                1.0
```

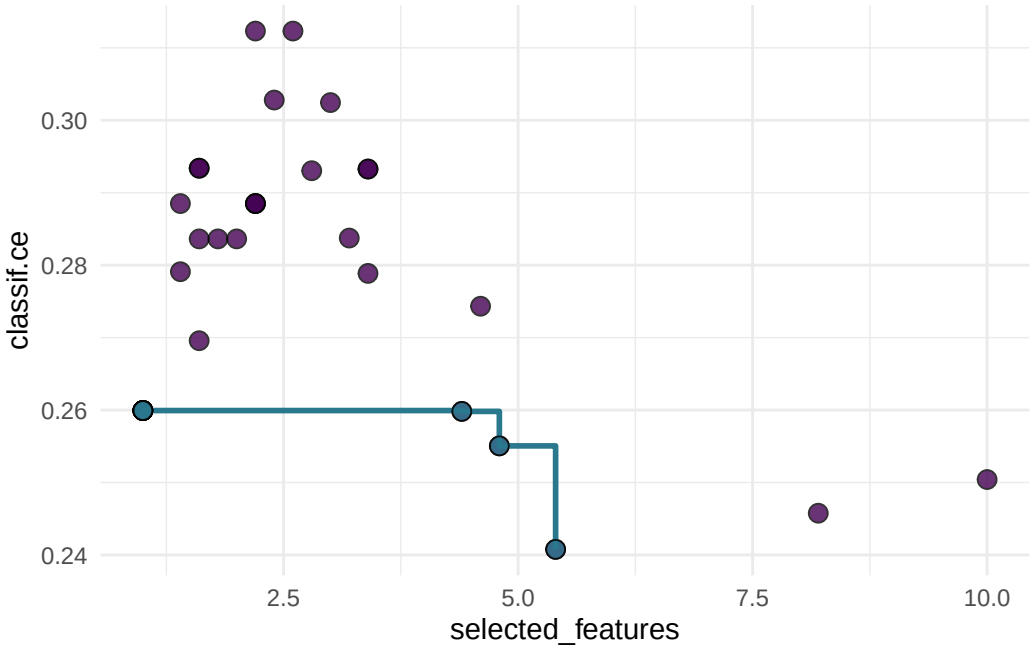


Figure 4.9: Pareto front of selected features and classification error. Purple dots represent tested configurations, each blue dot individually represents a Pareto-optimal configuration and all blue dots together represent the Pareto front.

4.8 Conclusion

In this chapter, we learned how to optimize a model using tuning instances, about different tuners and terminators, search spaces and transformations, how to make use of the automated methods for quicker implementation in larger experiments, and the importance of nested resampling. The most important functions and classes we learned about are in Table 4.5 alongside their R6 classes. If you are interested in learning more about the underlying R6 classes to gain finer control of these methods, then you are invited to take a look at the online documentation.

S3 function	R6 Class	Summary
`tnr()`	`Tuner`	Determines an optimisation algorithm
`trm()`	`Terminator`	Controls when to terminate the tuning algorithm
`ti()`	`TuningInstanceSingleCrit` or `TuningInstanceMultiCrit`	Stores tuning settings and save results

S3 function	R6 Class	Summary
`paradox::to_tune()`	`paradox::TuneToken`	Sets which parameters in a learner to tune and over what search space
`auto_tuner()`	`AutoTuner`	Automates the tuning process
`extract_inner_tuning_results()`	-	Extracts inner results from nested resampling
`extract_inner_tuning_archives()`	-	Extracts inner archives from nested resampling

Table 4.5: Core S3 'sugar' functions for model optimization in mlr3 with the underlying R6 class that are constructed when these functions are called (if applicable) and a summary of the purpose of the functions.

Resources

The mlr3tuning cheatsheet[2] summarizes the most important functions of mlr3tuning and the mlr3 gallery[3] features a collection of case studies and demonstrations about optimization, most notably learn how to:

- Apply advanced methods in the practical tuning series[4].
- Optimize an rpart classification tree with only a few lines of code[5].
- Tune an XGBoost model with early stopping[6].
- Quickly load and tune over search spaces that have been published in literature with mlr3tuningspaces[7].

4.9 Exercises

1. Tune the `mtry`, `sample.fraction`, `num.trees` hyperparameters of a random forest model (`regr.ranger`) on the `Motor Trend` data set (`mtcars`). Use a simple random search with 50 evaluations and select a suitable batch size. Evaluate with a 3-fold cross-validation and the root mean squared error.
2. Evaluate the performance of the model created in Question 1 with nested resampling. Use a holdout validation for the inner resampling and a 3-fold cross-validation for the outer resampling. Print the unbiased performance estimate of the model.
3. Tune and benchmark an XGBoost model against a logistic regression and determine which has the best Brier score. Use mlr3tuningspaces and nested resampling.

[2] https://cheatsheets.mlr-org.com/mlr3tuning.pdf
[3] https://mlr-org.com/gallery.html#category:tuning
[4] https://mlr-org.com/gallery.html#category:practical_tuning_series
[5] https://mlr-org.com/gallery/2022-11-10-hyperparameter-optimization-on-the-palmer-penguins/
[6] https://mlr-org.com/gallery/2022-11-04-early-stopping-with-xgboost/
[7] https://mlr-org.com/gallery/2021-07-06-introduction-to-mlr3tuningspaces/

5

Feature Selection

Marvin N. Wright
Leibniz Institute for Prevention Research and Epidemiology – BIPS, and University of Bremen, and University of Copenhagen

TODO
TODO

Feature selection, also known as variable or descriptor selection, is the process of finding a subset of features to use with a given task and learner. Using an *optimal set* of features can have several benefits:

- improved predictive performance, since we reduce overfitting on irrelevant features,
- robust models that do not rely on noisy features,
- simpler models that are easier to interpret,
- faster model fitting, e.g. for model updates,
- faster prediction, and
- no need to collect potentially expensive features.

However, these objectives will not necessarily be optimized by the same *optimal set* of features and thus feature selection is inherently multi-objective. In this chapter, we mostly focus on feature selection as a means of improving predictive performance, but also briefly cover optimization of multiple criteria (Section 5.2.5).

Reducing the amount of features can improve models across many scenarios, but it can be especially helpful in datasets that have a high number of features in comparison to the number of datapoints. Many learners perform implicit, also called embedded, feature selection, e.g. via the choice of variables used for splitting in a decision tree. Most other feature selection methods are model agnostic, i.e. they can be used together with any learner. Of the many different approaches to identifying relevant features, we will focus on two general concepts, which are described in detail below: Filter and Wrapper methods (Guyon and Elisseeff 2003; Chandrashekar and Sahin 2014).

For this chapter, the reader should know the basic concepts of mlr3 (Chapter 2), i.e. know about tasks (Section 2.1) and learners (Section 2.2). Basics about performance evaluation (Chapter 3), i.e. resampling (Section 3.2) and benchmarking (Section 3.3) are helpful but not strictly necessary.

5.1 Filters

Filter methods are preprocessing steps that can be applied before training a model. A very simple filter approach could look like this:

1. calculate the correlation coefficient ρ between each feature and a numeric target variable, and
2. select all features with $\rho > 0.2$ for further modelling steps.

This approach is a *univariate* filter because it only considers the univariate relationship between each feature and the target variable. Further, it can only be applied to regression tasks with continuous features and the threshold of $\rho > 0.2$ is quite arbitrary. Thus, more advanced filter methods, e.g. *multivariate* filters based on feature importance, usually perform better (Bommert et al. 2020). On the other hand, a benefit of univariate filters is that they are usually computationally cheaper than more complex filter or wrapper methods. In the following, it is described how to calculate univariate, multivariate and feature importance filters, how to access implicitly selected features, how to integrate filters in a machine learning pipeline and how to optimize filter thresholds.

Filter algorithms select features by assigning numeric scores to each feature, e.g. correlation between feature and target variables, use these to rank the features and select a feature subset based on the ranking. Features that are assigned lower scores can then be omitted in subsequent modeling steps. All filters are implemented via the package `mlr3filters`. Below, we cover how to

- instantiate a `Filter` object,
- calculate scores for a given task, and
- use calculated scores to select or drop features.

One special case of filters are feature importance filters (Section 5.1.2). They select features that are important according to the model induced by a selected `Learner`. Feature importance filters rely on the learner to extract information on feature importance from a trained model, for example, by inspecting a learned decision tree and returning the features that are used as split variables, or by computing model-agnostic feature importance (Chapter 9) values for each feature.

Many filter methods are implemented in `mlr3filters`, for example:

- Correlation, calculating Pearson or Spearman correlation between numeric features and numeric targets (`FilterCorrelation`)
- Information gain, i.e. mutual information of the feature and the target or the reduction of uncertainty of the target due to a feature (`FilterInformationGain`)
- Minimal joint mutual information maximization, minimizing the joint information between selected features to avoid redundancy (`FilterJMIM`)
- Permutation score, which calculates permutation feature importance (see Chapter 9) with a given learner for each feature (`FilterPermutation`)
- Area under the ROC curve calculated for each feature separately (`FilterAUC`)

Most of the filter methods have some limitations, e.g. the correlation filter can only be calculated for regression tasks with numeric features. For a full list of all implemented filter methods we refer the reader to the mlr3filters website[1], which also shows the supported task and features types. A benchmark of filter methods was performed by Bommert et al. (2020), who recommend to not rely on a single filter method but try several ones if the available computational resources allow. If only a single filter method is to be used, the authors recommend to use a feature importance filter using random forest permutation importance (see Section 5.1.2), similar to the permutation method described above, but also the JMIM and AUC filters performed well in their comparison.

[1] https://mlr3filters.mlr-org.com

5.1.1 Calculating Filter Values

The first step is to create a new R object using the class of the desired filter method. Similar to other instances in `mlr3`, these are registered in a dictionary (`mlr_filters`) with an associated shortcut function `flt()`. Each object of class `Filter` has a `$calculate()` method which computes the filter values and ranks them in a descending order. For example, we can use the information gain filter described above:

```
library("mlr3verse")
filter = flt("information_gain")
```

Such a `Filter` object can now be used to calculate the filter on the penguins data and get the results:

```
task = tsk("penguins")
filter$calculate(task)

as.data.table(filter)
```

```
          feature       score
1: flipper_length 0.581167901
2:    bill_length 0.544896584
3:     bill_depth 0.538718879
4:         island 0.520157171
5:      body_mass 0.442879511
6:            sex 0.007244168
7:           year 0.000000000
```

Some filters have hyperparameters, which can be changed similar to setting hyperparameters of a `Learner` using `$param_set$values`. For example, to calculate `"spearman"` instead of `"pearson"` correlation with the correlation filter:

```
filter_cor = flt("correlation")
filter_cor$param_set$values = list(method = "spearman")
filter_cor$param_set
```

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

5.1.2 Feature Importance Filters

To use feature importance filters, we can use a learner with integrated feature importance methods. All learners with the property "importance" have this functionality. A list of all learners with this property can be found with

```
as.data.table(mlr_learners)[sapply(properties, function(x) "importance" %in% x)]
```

```
                        key                            label task_type
1:         classif.catboost                 Gradient Boosting   classif
```

```
 2:      classif.featureless Featureless Classification Learner   classif
 3:              classif.gbm                   Gradient Boosting   classif
 4: classif.imbalanced_rfsrc           Imbalanced Random Forest   classif
 5:         classif.lightgbm                   Gradient Boosting   classif
---
22:                 surv.gbm                   Gradient Boosting      surv
23:              surv.mboost Boosted Generalized Additive Model      surv
24:              surv.ranger                         Random Forest      surv
25:               surv.rfsrc                         Random Forest      surv
26:             surv.xgboost                   Gradient Boosting      surv
4 variables not shown: [feature_types, packages, properties, predict_types]
```

or on the mlr3 website[2].

For some learners, the desired filter method needs to be set during learner creation. For example, learner `classif.ranger` comes with multiple integrated methods, c.f. the help page of `ranger::ranger()`. To use the feature importance method "impurity", select it during learner construction:

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

Now we can use the `FilterImportance` filter class:

```
task = tsk("penguins")

# Remove observations with missing data
task$filter(which(complete.cases(task$data())))

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

```
          feature     score
1:    bill_length 76.374739
2: flipper_length 45.348924
3:     bill_depth 36.305939
4:      body_mass 26.457564
5:         island 24.077990
6:            sex  1.597289
7:           year  1.215536
```

5.1.3 Embedded Methods

Many learners internally select a subset of the features which they find helpful for prediction, but ignore other features. For example, a decision tree might never select some features for splitting. These subsets can be used for feature selection, which we call embedded methods because the feature selection is embedded in the learner. The selected features (and those not selected) can be queried if the learner has the `"selected_features"` property, as the following example demonstrates:

[2] https://mlr-org.com/learners.html

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

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

learner = learner$train(task)
learner$selected_features()
```

```
[1] "flipper_length" "bill_length"    "island"
```

The features selected by the model can be extracted by a `Filter` object, where `$calculate()` corresponds to training the learner on the given task:

```
filter = flt("selected_features", learner = learner)
filter$calculate(task)
as.data.table(filter)
```

```
          feature score
1:         island     1
2: flipper_length     1
3:    bill_length     1
4:     bill_depth     0
5:            sex     0
6:           year     0
7:      body_mass     0
```

Contrary to other filter methods, embedded methods just return value of 1 (selected features) and 0 (dropped feature).

5.1.4 Filter-based Feature Selection

After calculating a score for each feature, one has to select the features to be kept or those to be dropped from further modelling steps. For the `"selected_features"` filter described in embedded methods (Section 5.1.3), this step is straight-forward since the methods assigns either a value of 1 for a feature to be kept or 0 for a feature to be dropped. With `task$select()` the features with a value of 1 can be selected:

```
task = tsk("penguins")
learner = lrn("classif.rpart")
filter = flt("selected_features", learner = learner)
filter$calculate(task)

# select all features used by rpart
keep = names(which(filter$scores == 1))
task$select(keep)
task$feature_names
```

```
[1] "bill_length"    "flipper_length" "island"
```

> **Tip**
>
> To select features, we use the function `task$select()` and not `task$filter()`, which is used to filter rows (not columns) of the data matrix, see task mutators (Section 2.1.3).

For filter methods which assign continuous scores, there are essentially two ways to select features:

- select the top k features, or
- select all features with a score above a threshold τ.

Where the first option is equivalent to dropping the bottom $p-k$ features. For both options, one has to decide on a threshold, which is often quite arbitrary. For example, to implement the first option with the information gain filter:

```
task = tsk("penguins")
filter = flt("information_gain")
filter$calculate(task)

# select top 3 features from information gain filter
keep = names(head(filter$scores, 3))
task$select(keep)
task$feature_names
```

```
[1] "bill_depth"     "bill_length"    "flipper_length"
```

Or, the second option with $\tau = 0.5$:

```
task = tsk("penguins")
filter = flt("information_gain")
filter$calculate(task)

# select all features with score >0.5 from information gain filter
keep = names(which(filter$scores > 0.5))
task$select(keep)
task$feature_names
```

```
[1] "bill_depth"     "bill_length"    "flipper_length" "island"
```

Filters can be integrated into pipelines. While pipelines are described in detail in Chapter 6, here is a brief preview:

```
library(mlr3pipelines)
task = tsk("penguins")

# combine filter (keep top 3 features) with learner
graph = po("filter", filter = flt("information_gain"), filter.nfeat = 3) %>>%
  po("learner", lrn("classif.rpart"))

# now it can be used as any learner, but it includes the feature selection
```

```
learner = as_learner(graph)
learner$train(task)
```

Pipelines can also be used to apply hyperparameter optimization (Chapter 4) to the filter, i.e. tune the filter threshold to optimize the feature selection regarding prediction performance. We first combine a filter with a learner,

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

and tune how many feature to include

```
library("mlr3tuning")
ps = ps(information_gain.filter.nfeat = p_int(lower = 1, upper = 7))
instance = TuningInstanceSingleCrit$new(
  task = task,
  learner = learner,
  resampling = rsmp("holdout"),
  measure = msr("classif.acc"),
  search_space = ps,
  terminator = trm("none")
)
tuner = tnr("grid_search")
tuner$optimize(instance)
```

```
   information_gain.filter.nfeat learner_param_vals  x_domain classif.acc
1:                             5          <list[2]> <list[1]>   0.9391304
```

The output above shows only the best result. To show the results of all tuning steps, retrieve them from the archive of the tuning instance:

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

```
   information_gain.filter.nfeat classif.acc
1:                             2   0.9304348
2:                             5   0.9391304
3:                             1   0.7478261
4:                             7   0.9391304
5:                             3   0.9391304
6:                             6   0.9391304
7:                             4   0.9391304
7 variables not shown: [x_domain_information_gain.filter.nfeat, runtime_learners, timestamp, batch
```

We can also plot the tuning results:

```
autoplot(instance)
```

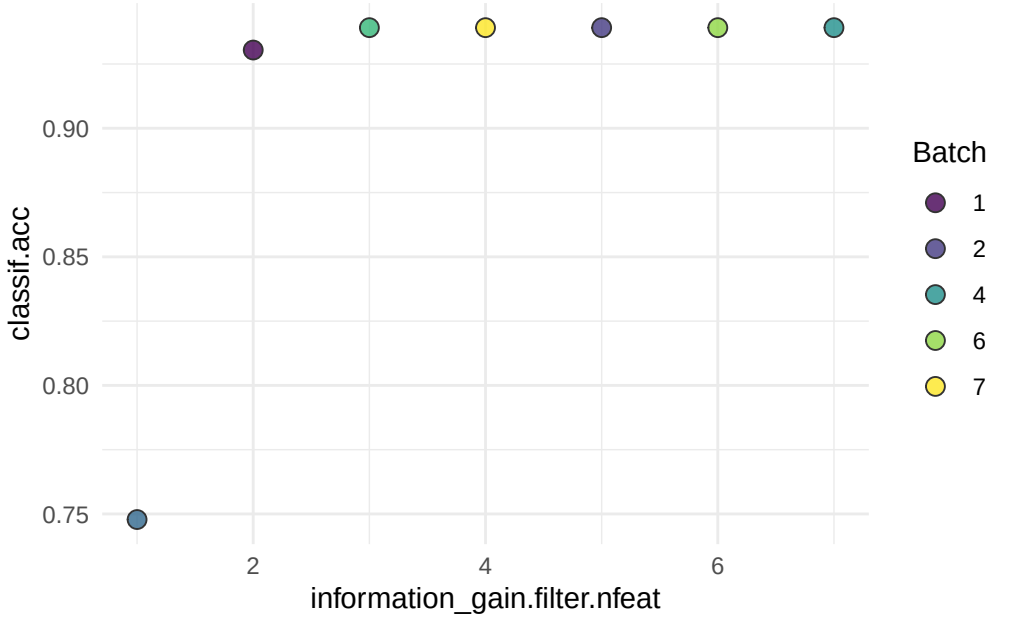


Figure 5.1: Model performance with different numbers of features, selected by an information gain filter.

For more details, see Pipelines (Chapter 6) and Hyperparameter Optimization (Chapter 4).

5.2 Wrapper Methods

Wrapper methods work by fitting models on selected feature subsets and evaluating their performance. This can be done in a sequential fashion, e.g. by iteratively adding features to the model in the so-called sequential forward selection, or in a parallel fashion, e.g. by evaluating random feature subsets in a random search. Below, the use of these simple approaches is described in a common framework along with more advanced methods such as genetic search. It is further shown how to select features by optimizing multiple performance measures and how to wrap a learner with feature selection to use it in pipelines or benchmarks.

In more detail, wrapper methods iteratively select features that optimize a performance measure. Instead of ranking features, a model is fit on a selected subset of features in each iteration and evaluated in resampling with respect to a selected performance measure. The strategy that determines which feature subset is used in each iteration is given by the `FSelector` object. A simple example is the sequential forward selection that starts with computing each single-feature model, selects the best one, and then iteratively adds the feature that leads to the largest performance improvement. Wrapper methods can be used with any learner but need to train the learner potentially many times, leading to a computationally intensive method. All wrapper methods are implemented via the package `mlr3fselect`. In this chapter, we cover how to

- instantiate an `FSelector` object,
- configure it, to e.g. respect a runtime limit or for different objectives,
- run it or fuse it with a `Learner` via an `AutoFSelector`.

> **Note**
>
> Wrapper-based feature selection is very similar to hyperparameter optimization (Chapter 4). The major difference is that we search for well-performing feature subsets instead of hyperparameter configurations. We will see below, that we can even use the same terminators, that some feature selection algorithms are similar to tuners and that we can also optimize multiple performance measures with feature selection.

5.2.1 Simple Forward Selection Example

We start with the simple example from above and do sequential forward selection with the penguins data:

```
library("mlr3fselect")

# subset features to ease visualization
task = tsk("penguins")
task$select(c("bill_depth", "bill_length", "body_mass", "flipper_length"))

instance = fselect(
  fselector = fs("sequential"),
  task =  task,
  learner = lrn("classif.rpart"),
  resampling = rsmp("holdout"),
  measure = msr("classif.acc")
)
```

To show all analyzed feature subsets and the corresponding performance, we use `as.data.table(instance$archive)`.

```
dt = as.data.table(instance$archive)
dt[batch_nr == 1, 1:5]
```

```
   bill_depth bill_length body_mass flipper_length classif.acc
1:       TRUE       FALSE     FALSE          FALSE   0.6956522
2:      FALSE        TRUE     FALSE          FALSE   0.7652174
3:      FALSE       FALSE      TRUE          FALSE   0.7043478
4:      FALSE       FALSE     FALSE           TRUE   0.7913043
```

We see that the feature `flipper_length` achieved the highest prediction performance in the first iteration and is thus selected. In the second round, adding `bill_length` improves performance to over 90%:

```
dt[batch_nr == 2, 1:5]
```

```
   bill_depth bill_length body_mass flipper_length classif.acc
```

```
1:       TRUE       FALSE     FALSE           TRUE   0.7652174
2:      FALSE        TRUE     FALSE           TRUE   0.9391304
3:      FALSE       FALSE      TRUE           TRUE   0.8173913
```

However, adding a third feature does not improve performance

```
dt[batch_nr == 3, 1:5]
```

```
   bill_depth bill_length body_mass flipper_length classif.acc
1:       TRUE        TRUE     FALSE           TRUE   0.9391304
2:      FALSE        TRUE      TRUE           TRUE   0.9391304
```

and the algorithm terminates. To directly show the best feature set, we can use:

```
instance$result_feature_set
```

```
[1] "bill_length"    "flipper_length"
```

> **Note**
>
> `instance$result_feature_set` shows features in alphabetical order and not in the order selected.

Internally, the `fselect` function creates an `FSelectInstanceSingleCrit` object and executes the feature selection with an `FSelector` object, based on the selected method, in this example an `FSelectorSequential` object. It uses the supplied resampling and measure to evaluate all feature subsets provided by the `FSelector` on the task.

At the heart of `mlr3fselect` are the R6 classes:

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

In the following two sections, these classes will be created manually, to learn more about the `mlr3fselect` package.

5.2.2 The `FSelectInstance` Classes

To create an `FSelectInstanceSingleCrit` object, we use the sugar function `fsi`, which is short for `FSelectInstanceSingleCrit$new()` or `FSelectInstanceMultiCrit$new()`, depending on the selected measure(s):

```
instance = fsi(
  task = tsk("penguins"),
  learner = lrn("classif.rpart"),
  resampling = rsmp("holdout"),
  measure = msr("classif.acc"),
  terminator = trm("evals", n_evals = 20)
)
```

Note that we have not selected a feature selection algorithm and thus did not select any features, yet. We have also supplied a so-called `Terminator`, which is used to stop the feature

selection. For the forward selection in the example above, we did not need a terminator because we simply tried all remaining features until the full model or no further performance improvement. However, for other feature selection algorithms such as `random search`, a terminator is required. The following terminator are available:

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

Above we used the sugar function `trm` to select `TerminatorEvals` with 20 evaluations.

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

5.2.3 The `FSelector` Class

The `FSelector` class is the base class for different feature selection algorithms. The following algorithms are currently implemented in `mlr3fselect`:

- Random search, trying random feature subsets until termination (`FSelectorRandomSearch`)
- Exhaustive search, trying all possible feature subsets (`FSelectorExhaustiveSearch`)
- Sequential search, i.e. sequential forward or backward selection (`FSelectorSequential`)
- Recursive feature elimination, which uses learner's importance scores to iteratively remove features with low feature importance (`FSelectorRFE`)
- Design points, trying all user-supplied feature sets (`FSelectorDesignPoints`)
- Genetic search, implementing a genetic algorithm which treats the features as a binary sequence and tries to find the best subset with mutations (`FSelectorGeneticSearch`)
- Shadow variable search, which adds permuted copies of all features (shadow variables) and stops when a shadow variable is selected (`FSelectorShadowVariableSearch`)

In this example, we will use a simple random search and retrieve it from the dictionary `mlr_fselectors` with the `fs()` sugar function, which is short for `FSelectorRandomSearch$new()`:

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

5.2.4 Starting the Feature Selection

To start the feature selection, we pass the `FSelectInstanceSingleCrit` object to the `$optimize()` method of the initialized `FSelector` object:

```
fselector$optimize(instance)
```

The algorithm proceeds as follows

1. The `FSelector` proposes at least one feature subset and may propose multiple subsets to improve parallelization, which can be controlled via the setting `batch_size`.

2. For each feature subset, the given learner is fitted on the task using the provided resampling and evaluated with the given measure.
3. All evaluations are stored in the archive of the `FSelectInstanceSingleCrit` object.
4. The terminator is queried if the budget is exhausted. If the budget is not exhausted, restart with 1) until it is.
5. Determine the feature subset with the best observed performance.
6. Store the best feature subset as the result in the instance object.

The best feature subset and the corresponding measured performance can be accessed from the instance:

```
as.data.table(instance$result)[, .(features, classif.acc)]
```

```
                                                          features classif.acc
1: bill_depth,bill_length,body_mass,flipper_length,island,sex,...   0.9391304
```

As in the forward selection example above, one can investigate all resamplings which were undertaken, as they are stored in the archive of the `FSelectInstanceSingleCrit` object and can be accessed by using `as.data.table()`:

```
as.data.table(instance$archive)[, .(bill_depth, bill_length, body_mass, classif.acc)]
```

```
    bill_depth bill_length body_mass classif.acc
 1:       TRUE        TRUE      TRUE   0.9391304
 2:      FALSE       FALSE      TRUE   0.7391304
 3:       TRUE        TRUE      TRUE   0.9391304
 4:       TRUE        TRUE      TRUE   0.9391304
 5:       TRUE        TRUE      TRUE   0.9391304
 6:      FALSE       FALSE     FALSE   0.6869565
 7:       TRUE       FALSE      TRUE   0.8086957
 8:      FALSE       FALSE      TRUE   0.7391304
 9:       TRUE       FALSE     FALSE   0.7739130
10:       TRUE       FALSE     FALSE   0.8000000
11:      FALSE       FALSE     FALSE   0.8086957
12:      FALSE       FALSE      TRUE   0.6869565
13:       TRUE        TRUE      TRUE   0.9391304
14:      FALSE       FALSE     FALSE   0.6173913
15:       TRUE        TRUE      TRUE   0.9217391
16:       TRUE        TRUE      TRUE   0.9391304
17:       TRUE        TRUE      TRUE   0.9043478
18:      FALSE       FALSE      TRUE   0.7391304
19:      FALSE       FALSE     FALSE   0.7739130
20:      FALSE       FALSE     FALSE   0.8086957
```

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

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

```

```
task$select(instance$result_feature_set)
learner$train(task)
```

The trained model can now be used to make a prediction on external data.

> **Warning**
>
> Predicting on observations present in the task used for feature selection should be avoided. The model has seen these observations already during feature selection and therefore performance evaluation results would be over-optimistic. Instead, to get unbiased performance estimates for the current task, nested resampling (see Section 5.2.6 and Section 4.3) is required.

5.2.5 Optimizing Multiple Performance Measures

You might want to use multiple criteria to evaluate the performance of the feature subsets. For example, you might want to select the subset with the highest classification accuracy and lowest time to train the model. However, these two subsets will generally not coincide, i.e. the subset with highest classification accuracy will probably be another subset than that with lowest training time. With `mlr3fselect`, the result is the pareto-optimal solution, i.e. the best feature subset for each of the criteria that is not dominated by another subset. For the example with classification accuracy and training time, a feature subset that is best in accuracy *and* training time will dominate all other subsets and thus will be the only pareto-optimal solution. If, however, different subsets are best in the two criteria, both subsets are pareto-optimal.

We will expand the previous example and perform feature selection on the penguins dataset, however, this time we will use `FSelectInstanceMultiCrit` to select the subset of features that has the highest classification accuracy and the one with the lowest time to train the model.

The feature selection process with multiple criteria is similar to that with a single criterion, except that we select two measures to be optimized:

```
instance = fsi(
  task = tsk("penguins"),
  learner = lrn("classif.rpart"),
  resampling = rsmp("holdout"),
  measure = msrs(c("classif.acc", "time_train")),
  terminator = trm("evals", n_evals = 5)
)
```

The function `fsi` creates an instance of `FSelectInstanceMultiCrit` if more than one measure is selected. We now create an `FSelector` and call the `$optimize()` function of the `FSelector` with the `FSelectInstanceMultiCrit` object, to search for the subset of features with the best classification accuracy and time to train the model. This time, we use `design points` to manually specify two feature sets to try: one with only the feature `sex` and one with all features except `island`, `sex` and `year`. We expect the sex-only model to train fast and the model including many features to be accurate.

```
  design = mlr3misc::rowwise_table(
    ~bill_depth, ~bill_length, ~body_mass, ~flipper_length, ~island, ~sex, ~year,
    FALSE, FALSE, FALSE, FALSE, FALSE, TRUE, FALSE,
    TRUE, TRUE, TRUE, TRUE, FALSE, FALSE, FALSE
  )

  fselector = fs("design_points", design = design)
  fselector$optimize(instance)
```

As above, the best feature subset and the corresponding measured performance can be accessed from the instance. However, in this simple case, if the fastest subset is not also the best performing subset, the result consists of two subsets: one with the lowest training time and one with the best classification accuracy:

```
  as.data.table(instance$result)[, .(features, classif.acc, time_train)]
```

```
                                          features classif.acc time_train
1:                                             sex   0.4347826      0.004
2: bill_depth,bill_length,body_mass,flipper_length   0.9304348      0.005
```

As explained above, the result is the pareto-optimal solution.

5.2.6 Automating the Feature Selection

The `AutoFSelector` class wraps a learner and augments it with an automatic feature selection for a given task. Because the `AutoFSelector` itself inherits from the `Learner` base class, it can be used like any other learner. Below, a new learner is created. This learner is then wrapped in a random search feature selector, which automatically starts a feature selection on the given task using an inner resampling, as soon as the wrapped learner is trained. Here, the function `auto_fselector` creates an instance of `AutoFSelector`, i.e. it is short for `AutoFSelector$new()`.

```
  at = auto_fselector(
    fselector = fs("random_search"),
    learner = lrn("classif.log_reg"),
    resampling = rsmp("holdout"),
    measure = msr("classif.acc"),
    terminator = trm("evals", n_evals = 10)
  )
  at
```

```
<AutoFSelector:classif.log_reg.fselector>
* Model: list
* Packages: mlr3, mlr3fselect, mlr3learners, stats
* Predict Type: response
* Feature Types: logical, integer, numeric, character, factor, ordered
* Properties: loglik, twoclass
```

We can now, as with any other learner, call the `$train()` and `$predict()` method. This time however, we pass it to `benchmark()` to compare the optimized feature subset to the complete feature set. This way, the `AutoFSelector` will do its resampling for feature selection on the

training set of the respective split of the outer resampling. The learner then undertakes predictions using the test set of the outer resampling. Here, the outer resampling refers to the resampling specified in `benchmark()`, whereas the inner resampling is that specified in `auto_fselector()`. This is called nested resampling (Section 4.3) and yields unbiased performance measures, as the observations in the test set have not been used during feature selection or fitting of the respective learner.

In the call to `benchmark()`, we compare our wrapped learner `at` with a normal logistic regression `lrn("classif.log_reg")`. For that, we create a benchmark grid with the task, the learners and a 3-fold cross validation on the `sonar` data.

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

bmr = benchmark(grid)
```

Now, we compare those two learners regarding classification accuracy and training time:

```
aggr = bmr$aggregate(msrs(c("classif.acc", "time_train")))
as.data.table(aggr)[, .(learner_id, classif.acc, time_train)]
```

```
                 learner_id classif.acc time_train
1: classif.log_reg.fselector   0.7351967 1.12933333
2:           classif.log_reg   0.6585231 0.04066667
```

We can see that, in this example, the feature selection improves prediction performance but also drastically increases the training time, since the feature selection (including resampling and random search) is part of the model training of the wrapped learner.

5.3 Conclusion

In this chapter, we learned how to perform feature selection with mlr3. We introduced filter and wrapper methods, combined feature selection with pipelines, learned how to automate the feature selection and covered the optimization of multiple performance measures. Table 5.1 gives an overview of the most important functions (S3) and classes (R6) used in this chapter.

S3 function	R6 Class	Summary
`flt()`	`Filter`	Selects features by calculating a score for each feature
`Filter$calculate()`	`Filter`	Calculates scores on a given task

S3 function	R6 Class	Summary
`fselect()`	`FSelectInstanceSingleCrit` or `FSelectInstanceMultiCrit`	Specifies a feature selection problem and stores the results
`fs()`	`FSelector`	Specifies a feature selection algorithm
`FSelector$optimize()`	`FSelector`	Executes the features selection specified by the `FSelectInstance` with the algorithm specified by the `FSelector`
`auto_fselector()`	`AutoFSelector`	Defines a learner that includes feature selection

Table 5.1: Core S3 'sugar' functions for feature selection in mlr3 with the underlying R6 class that are constructed when these functions are called (if applicable) and a summary of the purpose of the functions.

Resources

- A list of implemented filters in the `mlr3filters` package is provided on the mlr3filters website[3].
- A summary of wrapper-based feature selection with the `mlr3fselect` package is provided in the mlr3fselect cheatsheet[4].
- An overview of feature selection methods is provided by Chandrashekar and Sahin (2014).
- A more formal and detailed introduction to filters and wrappers is given in Guyon and Elisseeff (2003).
- Bommert et al. (2020) perform a benchmark of filter methods.
- Filters can be used as part of a machine learning pipeline (Chapter 6).
- Filters can be optimized with hyperparameter optimization (Chapter 4).

5.4 Exercises

1. Calculate a correlation filter on the `Motor Trend` data set (`mtcars`).
2. Use the filter from the first exercise to select the five best features in the `mtcars` data set.
3. Apply a backward selection to the `penguins` data set with a classification tree learner `"classif.rpart"` and holdout resampling by the measure classification accuracy. Compare the results with those in Section 5.2.1. Answer the following questions:
   a. Do the selected features differ?
   b. Which feature selection method achieves a higher classification accuracy?
   c. Are the accuracy values in b) directly comparable? If not, what has to be changed to make them comparable?

[3] https://mlr3filters.mlr-org.com
[4] https://cheatsheets.mlr-org.com/mlr3fselect.pdf

4. Automate the feature selection as in Section 5.2.6 with the `spam` data set and a logistic regression learner (`"classif.log_reg"`). Hint: Remember to call `library("mlr3learners")` for the logistic regression learner.

6

Pipelines

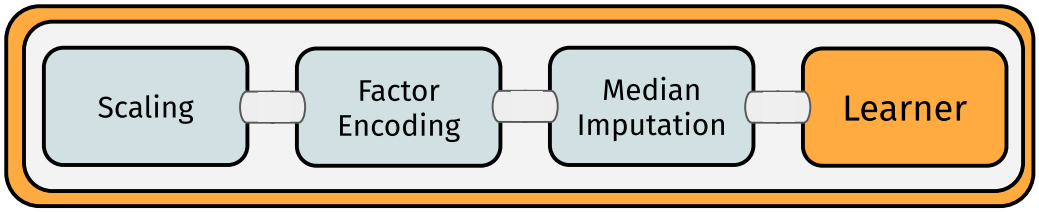
TODO
TODO

> **Note**
>
> This chapter is currently being re-written, in this PR. You can continue reading here if you want to learn about `mlr3pipelines`, but you currently don't need to bother checking this chapter for typos etc.

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

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

Below you can examine an example for such a graph:



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

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

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

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

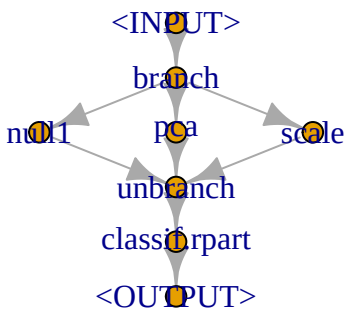
An extensive introduction to creating custom **PipeOps** (PO's) can be found in the technical introduction.

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

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

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

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



6.1 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                                          label
1:         boxcox Box-Cox Transformation of Numeric Features
2:         branch                                Path Branching
3:          chunk                 Chunk Input into Multiple Outputs
4: classbalancing                                Class Balancing
5:     classifavg                       Majority Vote Prediction
```

```
---
60:      threshold Change the Threshold of a Classification Prediction
61:  tunethreshold   Tune the Threshold of a Classification Prediction
62:       unbranch                              Unbranch Different Paths
63:         vtreat                    Interface to the vtreat Package
64:     yeojohnson      Yeo-Johnson Transformation of Numeric Features
9 variables not shown: [packages, tags, feature_types, input.num, output.num, input.type.train, in
```

Single POs can be created using the dictionary:

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

or using **syntactic sugar** `po(<name>)`:

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

Some POs require additional arguments for construction:

```
learner = po("learner")

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

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

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

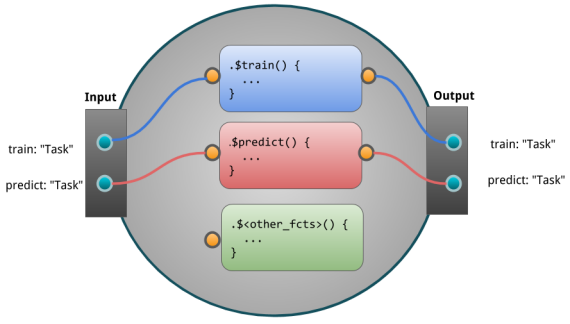
Hyperparameters of POs can be set through the `param_vals` argument. Here we set the fraction of features for a filter:

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

or in short notation:

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

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

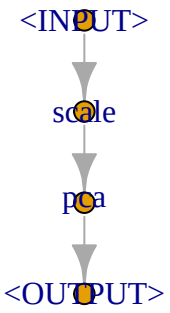


6.2 The Pipeline Operator: %>>%

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

This can be done using the **`%>>%`** ("double-arrow") operator. It takes either a `PipeOp` or a `Graph` on each of its sides and connects all of the outputs of its left-hand side to one of the inputs each of its right-hand side. The number of inputs therefore must match the number of outputs.

```
library("magrittr")

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



6.3 Nodes, Edges and Graphs

POs are combined into `Graph`s.

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

For this example we use the `pca` PO defined above and a new PO named "mutate". The

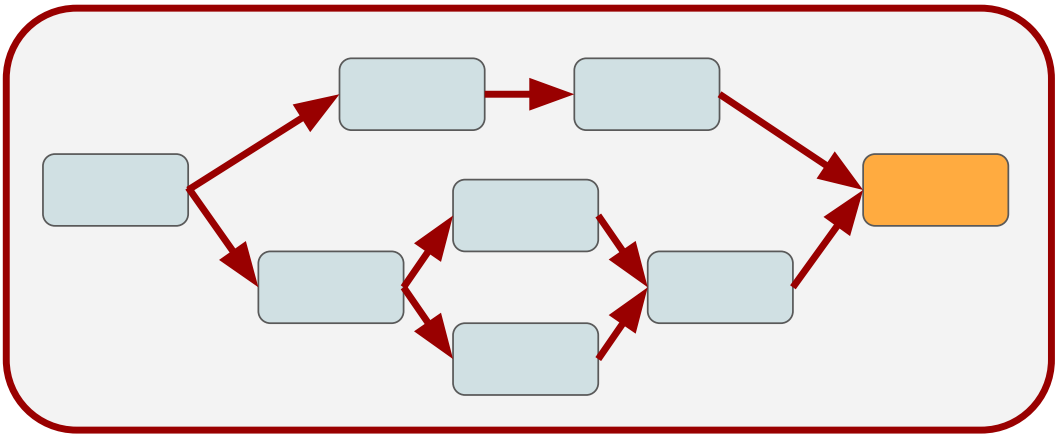
latter creates a new feature from existing variables. Additionally, we use the filter PO again.

```
mutate = po("mutate")

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

The recommended way to construct a graph is to use the `%>>%` operator to chain POs or `Graph`s.

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

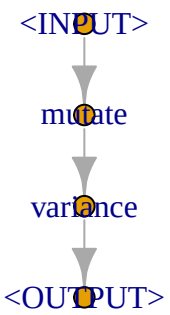
To illustrate how this sugar operator works under the surface we will include an example of the manual way (= hard way) to construct a `Graph`. This is done by creating an empty graph first. Then one fills the empty graph with POs, and connects edges between the POs. Conceptually, this may look like this:



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

The constructed `Graph` can be inspected using its `$plot()` function:

```
graph$plot()
```



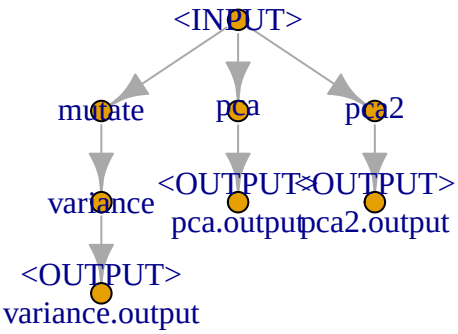
Chaining multiple POs of the same kind

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

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

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

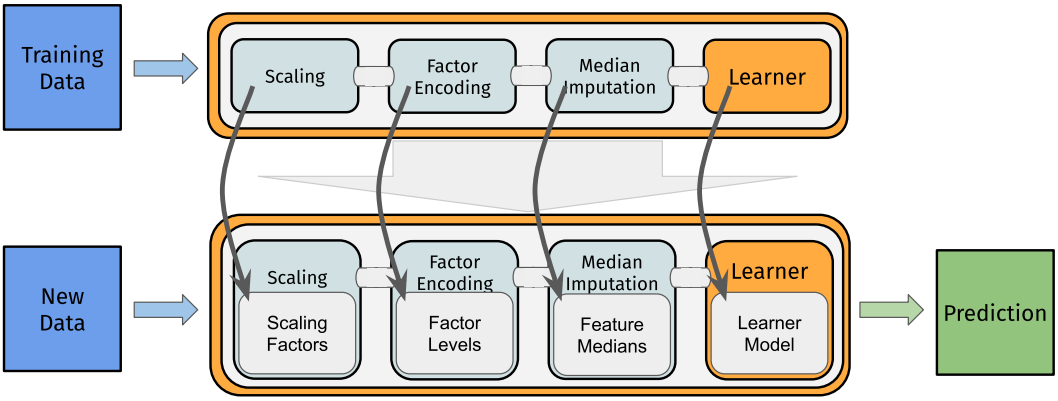
```
graph$plot()
```



6.4 Modeling

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

Conceptually, the process may be summarized as follows:



In the following we chain two preprocessing tasks:

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

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

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

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

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

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

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

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

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

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

```
glrn = as_learner(graph)
```

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

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

```
<ResampleResult> with 3 resampling iterations
 task_id                    learner_id resampling_id iteration warnings errors
    iris mutate.variance.classif.rpart            cv         1        0      0
    iris mutate.variance.classif.rpart            cv         2        0      0
```

```
    iris mutate.variance.classif.rpart            cv         3        0      0
```

6.4.1 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> with 3 resampling iterations
 task_id                    learner_id resampling_id iteration warnings errors
    iris mutate.variance.classif.rpart            cv         1        0      0
    iris mutate.variance.classif.rpart            cv         2        0      0
    iris mutate.variance.classif.rpart            cv         3        0      0
```

6.4.2 Tuning

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

```
library("paradox")
ps = ps(
  classif.rpart.cp = p_dbl(lower = 0, upper = 0.05),
  variance.filter.frac = p_dbl(lower = 0.25, upper = 1)
)
```

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

```
library("mlr3tuning")
instance = TuningInstanceSingleCrit$new(
  task = task,
  learner = glrn,
  resampling = rsmp("holdout"),
  measure = msr("classif.ce"),
  search_space = ps,
  terminator = trm("evals", n_evals = 20)
)
```

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

```
   classif.rpart.cp variance.filter.frac learner_param_vals  x_domain
1:       0.04886918            0.6870548          <list[5]> <list[2]>
1 variable not shown: [classif.ce]
```

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

```
instance$result_learner_param_vals
```

```
$mutate.mutation
list()

$mutate.delete_originals
[1] FALSE

$variance.filter.frac
[1] 0.6870548

$classif.rpart.xval
[1] 0

$classif.rpart.cp
[1] 0.04886918
```

```
instance$result_y
```

```
classif.ce
      0.02
```

6.5 Non-Linear Graphs

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

Possible types are:

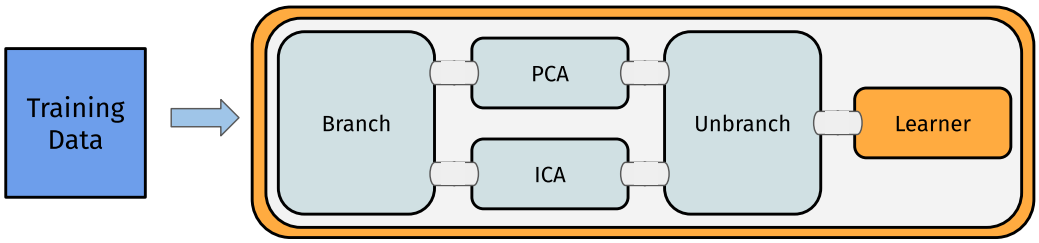
- Branching: Splitting of a node into several paths, e.g. useful when comparing multiple feature-selection methods (pca, filters). Only one path will be executed.
- Copying: Splitting of a node into several paths, all paths will be executed (sequentially). Parallel execution is not yet supported.
- Stacking: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

6.5.1 Branching & Copying

The `PipeOpBranch` and `PipeOpUnbranch` POs make it possible to specify multiple alternative paths. Only one path is actually executed, the others are ignored. The active path

is determined by a hyperparameter. This concept makes it possible to tune alternative preprocessing paths (or learner models).

Below a conceptual visualization of branching:



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

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

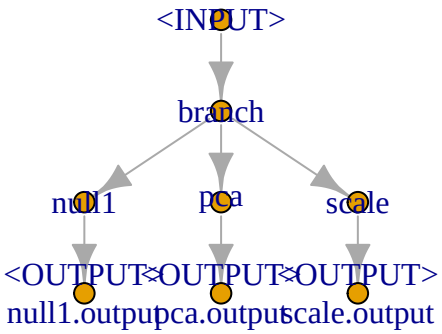
It is important to "unbranch" again after "branching", so that the outputs are merged into one result objects.

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

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

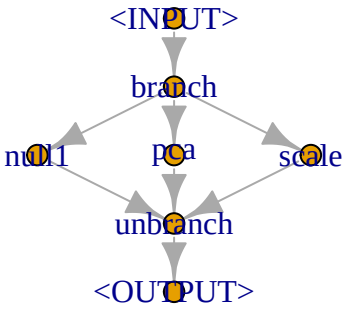
Without "unbranching" one creates the following graph:

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



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

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



The same can be achieved using a shorter notation:

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

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

6.5.2 Model Ensembles

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

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

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

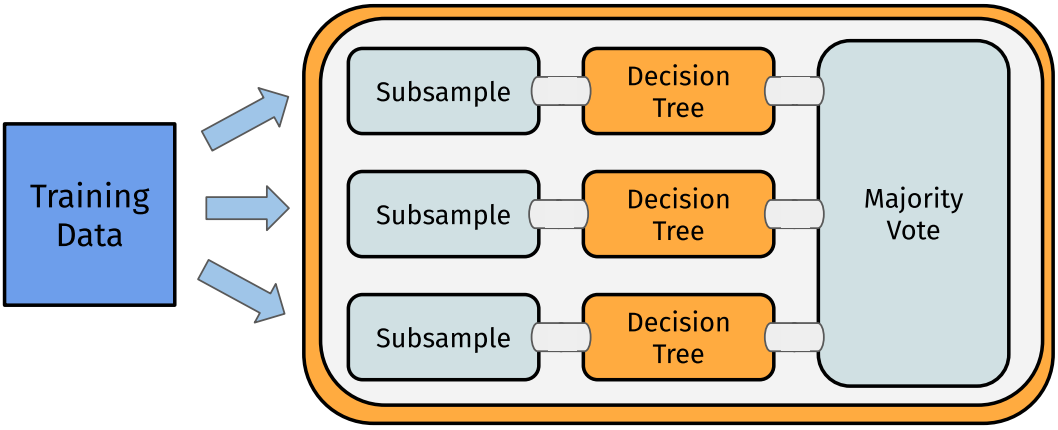
6.5.2.1 Bagging

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

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

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

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



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

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

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

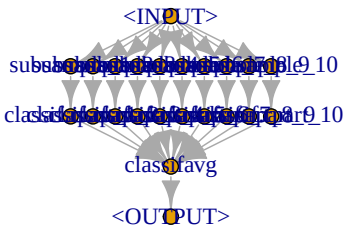
```
pred_set = ppl("greplicate", single_pred, 10L)
```

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

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

Now we can plot again to see what happens:

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



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

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

```
<PredictionClassif> for 30 observations:
    row_ids     truth   response prob.setosa prob.versicolor prob.virginica
         16    setosa     setosa           1             0.0            0.0
         18    setosa     setosa           1             0.0            0.0
         21    setosa     setosa           1             0.0            0.0
---
        129 virginica  virginica           0             0.0            1.0
        134 virginica versicolor           0             0.5            0.5
        149 virginica  virginica           0             0.0            1.0
```

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

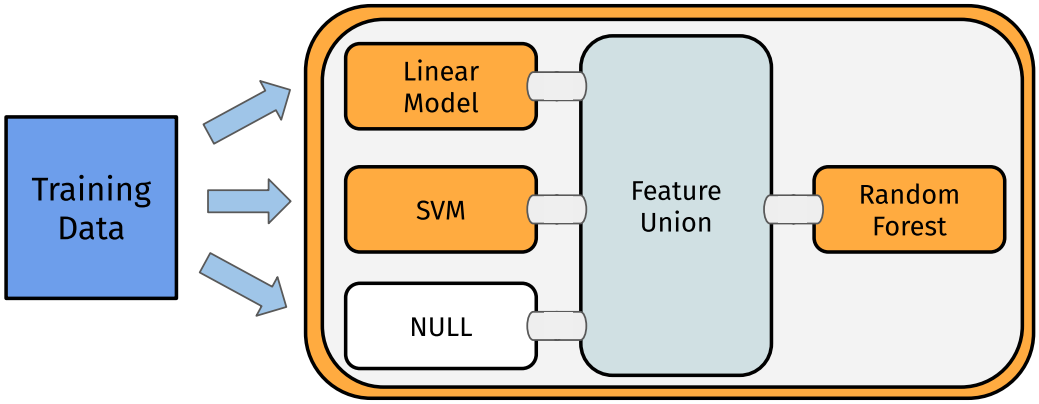
6.5.2.2 Stacking

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

Below an conceptual illustration of stacking:

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

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



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

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

```
lrn = lrn("classif.rpart")
lrn_0 = po("learner_cv", lrn$clone())
lrn_0$id = "rpart_cv"
```

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

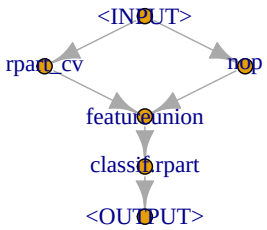
```
level_0 = gunion(list(lrn_0, po("nop")))
```

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

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

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

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




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

```
<PredictionClassif> for 30 observations:
    row_ids     truth   response
         16    setosa     setosa
         18    setosa     setosa
         21    setosa     setosa
---
        129 virginica  virginica
        134 virginica versicolor
        149 virginica  virginica
```

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

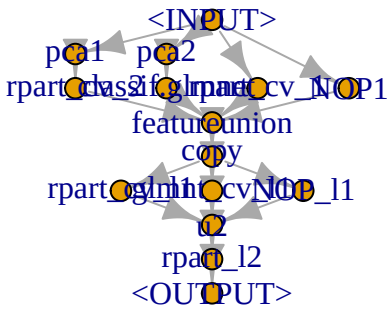
6.5.2.3 Multilevel Stacking

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

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

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

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

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

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

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

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

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

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




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

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

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

```
classif.ce
       0.1
```

6.6 Adding new PipeOps

This section showcases how the `mlr3pipelines` package can be extended to include custom `PipeOp`s. To run the following examples, we will need a `Task`; we are using the well-known "Iris" task:

```
library("mlr3")
task = tsk("iris")
task$data()
```

```
       Species Petal.Length Petal.Width Sepal.Length Sepal.Width
  1:    setosa          1.4         0.2          5.1         3.5
```

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

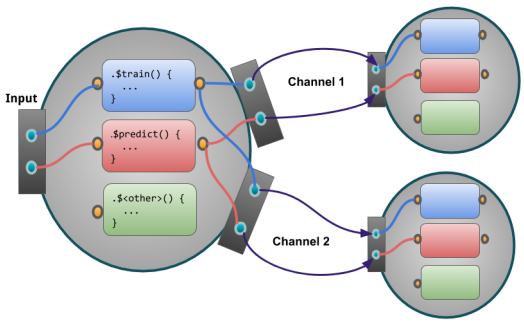
`mlr3pipelines` is fundamentally built around `R6`. When planning to create custom `PipeOp` objects, it can only help to familiarize yourself with it.

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

6.6.1 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.



6.6.1.1 First Steps: Inheriting from `PipeOp`

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

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

```
    ),
    private == list(
      .train = function(inputs) {
        ....
      },

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

Note, that **private** methods, e.g. `.train` and `.predict` etc are prefixed with a `.`.

6.6.1.2 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
    )
  }
```

6.6.1.3 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)
}
```

6.6.1.4 Putting it Together

The whole definition thus becomes

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

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

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

train it on something:

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

print(gr)
```

```
Graph with 1 PipeOps:
       ID         State sccssors prdcssors
 copy.two <<UNTRAINED>>
```

```
result = gr$train(task)

str(result)
```

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

6.6.2 Special Case: Preprocessing

Many `PipeOp`s 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 `PipeOp`s should use it if they fall in the kind of use-case named above.

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

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

6.6.2.1 Example: `PipeOpDropNA`

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

Because `mlr3` `"Task", text = "Tasks")` only contain a view to the underlying data, it is not necessary to modify data to remove rows with missing values. Instead, the rows can be removed using the `Task`'s `$filter` method, which modifies the `Task` in-place. This is done in the private method `.train_task()`. We take care that we also set the `$state` slot to

signal that the PipeOp was trained.

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

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

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

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

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

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

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

```
smalliris = iris[(1:5) * 30, ]
smalliris[1, 1] = NA
smalliris[2, 2] = NA
sitask = as_task_classif(smalliris, target = "Species")
print(sitask$data())
```

```
      Species Petal.Length Petal.Width Sepal.Length Sepal.Width
1:     setosa          1.6         0.2           NA         3.2
2: versicolor          3.9         1.4          5.2          NA
3: versicolor          4.0         1.3          5.5         2.5
4:  virginica          5.0         1.5          6.0         2.2
5:  virginica          5.1         1.8          5.9         3.0
```

We test this by feeding it to a new `Graph` that uses `PipeOpDropNA`.

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

filtered_task = gr$train(sitask)[[1]]
print(filtered_task$data())
```

```
      Species Petal.Length Petal.Width Sepal.Length Sepal.Width
1: versicolor          4.0         1.3          5.5         2.5
2:  virginica          5.0         1.5          6.0         2.2
3:  virginica          5.1         1.8          5.9         3.0
```

6.6.2.2 Example: `PipeOpScaleAlways`

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

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

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

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

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

  private = list(
    .select_cols = function(task) {
```

```
      task$feature_types[type == "numeric", id]
    },

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

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

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

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

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

result = gr$train(task)

result[[1]]$data()
```

```
        Species Petal.Length Petal.Width Sepal.Length Sepal.Width
  1:     setosa   -1.3357516  -1.3110521  -0.89767388  1.01560199
  2:     setosa   -1.3357516  -1.3110521  -1.13920048 -0.13153881
  3:     setosa   -1.3923993  -1.3110521  -1.38072709  0.32731751
  4:     setosa   -1.2791040  -1.3110521  -1.50149039  0.09788935
  5:     setosa   -1.3357516  -1.3110521  -1.01843718  1.24503015
 ---
146: virginica    0.8168591   1.4439941   1.03453895 -0.13153881
147: virginica    0.7035638   0.9192234   0.55148575 -1.27867961
148: virginica    0.8168591   1.0504160   0.79301235 -0.13153881
149: virginica    0.9301544   1.4439941   0.43072244  0.78617383
150: virginica    0.7602115   0.7880307   0.06843254 -0.13153881
```

6.6.3 Special Case: Preprocessing with Simple Train

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

may depend on training data).

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

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

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

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

6.6.3.1 Example: `PipeOpDropConst`

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

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

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

The full `PipeOp` could be written as follows:

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

  private = list(
    .get_state = function(task) {
```

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

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

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

```
irishead = task$clone()$filter(1:5)
irishead$data()
```

```
   Species Petal.Length Petal.Width Sepal.Length Sepal.Width
1:  setosa          1.4         0.2          5.1         3.5
2:  setosa          1.4         0.2          4.9         3.0
3:  setosa          1.3         0.2          4.7         3.2
4:  setosa          1.5         0.2          4.6         3.1
5:  setosa          1.4         0.2          5.0         3.6
```

```
gr = Graph$new()$add_pipeop(PipeOpDropConst$new())
dropped_task = gr$train(irishead)[[1]]

dropped_task$data()
```

```
   Species Petal.Length Sepal.Length Sepal.Width
1:  setosa          1.4          5.1         3.5
2:  setosa          1.4          4.9         3.0
3:  setosa          1.3          4.7         3.2
4:  setosa          1.5          4.6         3.1
5:  setosa          1.4          5.0         3.6
```

We can also see that the `$state` was correctly set. Calling `$.predict()` with this graph, even with different data (the whole Iris `Task`!) will still drop the `"Petal.Width"` column, as it should.

```
gr$pipeops$drop.const$state
```

```
$cnames
[1] "Petal.Length" "Sepal.Length" "Sepal.Width"

$affected_cols
[1] "Petal.Length" "Petal.Width"  "Sepal.Length" "Sepal.Width"

$intasklayout
             id    type
```

```
1: Petal.Length numeric
2:  Petal.Width numeric
3: Sepal.Length numeric
4:  Sepal.Width numeric

$outtasklayout
             id    type
1: Petal.Length numeric
2: Sepal.Length numeric
3:  Sepal.Width numeric

$outtaskshell
Empty data.table (0 rows and 4 cols): Species,Petal.Length,Sepal.Length,Sepal.Width
```

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

dropped_predict$data()
```

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

6.6.3.2 Example: `PipeOpScaleAlwaysSimple`

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

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

  private = list(
```

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

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

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

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

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

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

```
result_posa$data()
```

```
       Species Petal.Length Petal.Width Sepal.Length Sepal.Width
  1:    setosa   -1.3357516  -1.3110521  -0.89767388  1.01560199
  2:    setosa   -1.3357516  -1.3110521  -1.13920048 -0.13153881
  3:    setosa   -1.3923993  -1.3110521  -1.38072709  0.32731751
  4:    setosa   -1.2791040  -1.3110521  -1.50149039  0.09788935
  5:    setosa   -1.3357516  -1.3110521  -1.01843718  1.24503015
 ---
146: virginica    0.8168591   1.4439941   1.03453895 -0.13153881
147: virginica    0.7035638   0.9192234   0.55148575 -1.27867961
148: virginica    0.8168591   1.0504160   0.79301235 -0.13153881
149: virginica    0.9301544   1.4439941   0.43072244  0.78617383
150: virginica    0.7602115   0.7880307   0.06843254 -0.13153881
```

```
result_posa_simple$data()
```

```
       Species Petal.Length Petal.Width Sepal.Length Sepal.Width
  1:    setosa   -1.3357516  -1.3110521  -0.89767388  1.01560199
  2:    setosa   -1.3357516  -1.3110521  -1.13920048 -0.13153881
  3:    setosa   -1.3923993  -1.3110521  -1.38072709  0.32731751
  4:    setosa   -1.2791040  -1.3110521  -1.50149039  0.09788935
  5:    setosa   -1.3357516  -1.3110521  -1.01843718  1.24503015
 ---
```

```
146: virginica    0.8168591   1.4439941   1.03453895 -0.13153881
147: virginica    0.7035638   0.9192234   0.55148575 -1.27867961
148: virginica    0.8168591   1.0504160   0.79301235 -0.13153881
149: virginica    0.9301544   1.4439941   0.43072244  0.78617383
150: virginica    0.7602115   0.7880307   0.06843254 -0.13153881
```

6.6.4 Hyperparameters

`mlr3pipelines` uses the [paradox](https://paradox.mlr-org.com) package to define parameter spaces for `PipeOp`s. Parameters for `PipeOp`s 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 `Graph`s that modify the individual `PipeOp`'s behavior. The `Graph`s, when encapsulated in `GraphLearner`s, can even be tuned using the tuning functionality in `mlr3tuning`.

Hyperparameters are declared during initialization, when calling the `PipeOp`'s `$initialize()` function, by giving a `param_set` argument. The `param_set` must be a `ParamSet` from the `paradox` package; see the tuning chapter or **?@sec-paradox** for more information on how to define parameter spaces. After construction, the `ParamSet` can be accessed through the `$param_set` slot. While it is *possible* to modify this `ParamSet`, using e.g. the `$add()` and `$add_dep()` functions, *after* adding it to the `PipeOp`, it is strongly advised against.

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

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

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

6.6.4.1 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:

```
1  PipeOpScale$public_methods$initialize
```

```
function (id = "scale", param_vals = list())
.__PipeOpScale__initialize(self = self, private = private, super = super,
    id = id, param_vals = param_vals)
<environment: namespace:mlr3pipelines>
```

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

```
pss = po("scale")
print(pss$param_set)
```

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

```
pss$param_set$values$center = FALSE
print(pss$param_set$values)
```

```
$robust
[1] FALSE

$center
[1] FALSE
```

```
pss$param_set$values$scale = "TRUE" # bad input is checked!
```

```
Error in self$assert(xs): Assertion on 'xs' failed: scale: Must be of type 'logical flag', not 'ch
```

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

```
PipeOpScale$private_methods$.train_dt
```

```
function (dt, levels, target)
.__PipeOpScale__.train_dt(self = self, private = private, super = super,
    dt = dt, levels = levels, target = target)
<environment: namespace:mlr3pipelines>
```

Another change that is necessary compared to `PipeOpScaleAlways` is that the attributes `"scaled:scale"` and `"scaled:center"` are not always present, depending on parameters, and possibly need to be set to default values 1 or 0, respectively.

It is now even possible (if a bit pointless) to call `PipeOpScale` with both `scale` and `center` set to `FALSE`, which returns the original dataset, unchanged.

```
pss$param_set$values$scale = FALSE
pss$param_set$values$center = FALSE

gr = Graph$new()
gr$add_pipeop(pss)

```

```
result = gr$train(task)

result[[1]]$data()
```

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

6.7 Special Operators

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

6.7.1 Imputation: `PipeOpImpute`

Often you will be using data sets that have missing values. There are many methods of dealing with this issue, 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. These methods are called imputation.

The following PipeOps, PipeOpImpute:

- Add an indicator column marking whether a value for a given feature was missing or not (numeric only)
- Impute numeric values from a histogram
- Impute categorical values using a learner
- We use `po("featureunion")` and `po("nop")` to cbind the missing indicator features. In other words to combine the indicator columns with the rest of the data.

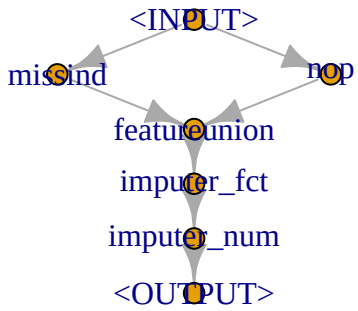
```
# Imputation example
task = tsk("penguins")
task$missings()
```

```
     species     bill_depth    bill_length      body_mass flipper_length
           0              2              2              2              2
      island            sex           year
           0             11              0
```

```
# Add missing indicator columns ("dummy columns") to the Task
pom = po("missind")
# Simply pushes the input forward
nop = po("nop")
# Imputes numerical features by histogram.
pon = po("imputehist", id = "imputer_num")
# combines features (used here to add indicator columns to original data)
pou = po("featureunion")
# Impute categorical features by fitting a Learner ("classif.rpart") for each feature.
pof = po("imputelearner", lrn("classif.rpart"), id = "imputer_fct", affect_columns = selector_t
```

Now we construct the graph.

```
impgraph = list(
  pom,
  nop
) %>>% pou %>>% pof %>>% pon

impgraph$plot()
```



Now we get the new task and we can see that all of the missing values have been imputed.

```
new_task = impgraph$train(task)[[1]]

new_task$missings()
```

```
          species missing_bill_depth missing_bill_length
```

```
                  0                       0                  0
missing_body_mass missing_flipper_length              island
                  0                       0                  0
               year                     sex        bill_depth
                  0                       0                  0
        bill_length             body_mass      flipper_length
                  0                       0                  0
```

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

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

6.7.2 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:

`iris` dataset looks like this:

```
task = task = tsk("iris")
head(as.data.table(task))
```

```
   Species Petal.Length Petal.Width Sepal.Length Sepal.Width
1:  setosa          1.4         0.2          5.1         3.5
2:  setosa          1.4         0.2          4.9         3.0
3:  setosa          1.3         0.2          4.7         3.2
4:  setosa          1.5         0.2          4.6         3.1
5:  setosa          1.4         0.2          5.0         3.6
6:  setosa          1.7         0.4          5.4         3.9
```

Once we do the mutations, you can see the new columns:

```
pom = po("mutate")

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

new_task = pom$train(list(task))[[1]]
head(as.data.table(new_task))
```

```
   Species Petal.Length Petal.Width Sepal.Length Sepal.Width Sepal.Sum
1:  setosa          1.4         0.2          5.1         3.5       8.6
```

```
2:  setosa          1.4         0.2          4.9         3.0       7.9
3:  setosa          1.3         0.2          4.7         3.2       7.9
4:  setosa          1.5         0.2          4.6         3.1       7.7
5:  setosa          1.4         0.2          5.0         3.6       8.6
6:  setosa          1.7         0.4          5.4         3.9       9.3
2 variables not shown: [Petal.Sum, Sepal.Petal.Ratio]
```

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`).

6.7.3 Training on data subsets: `PipeOpChunk`

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

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

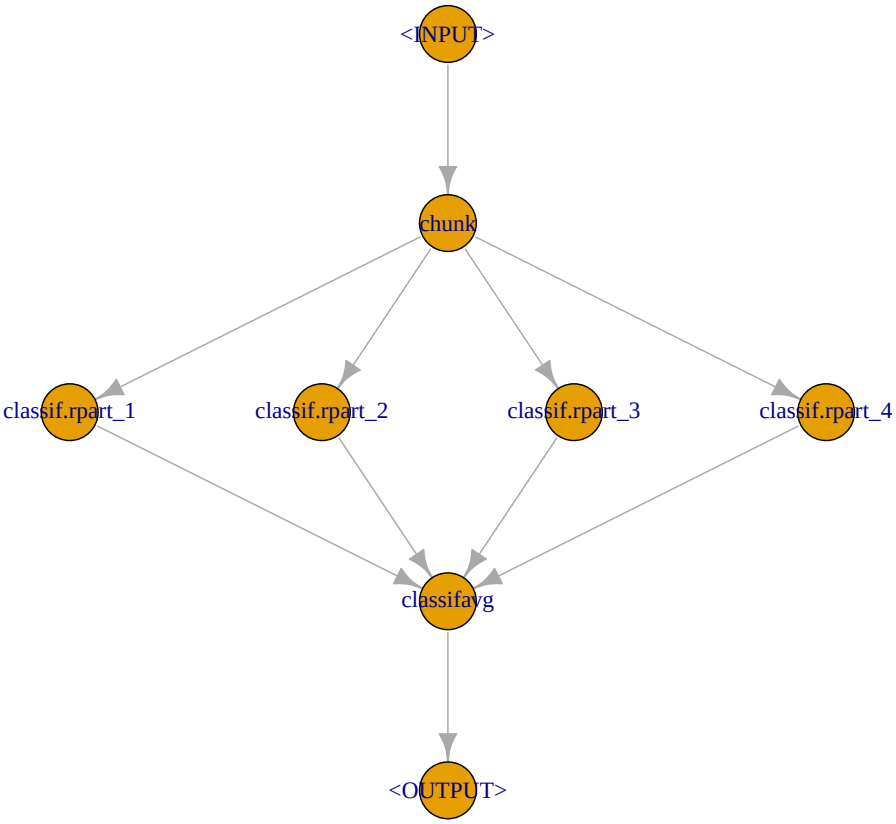
```
chks = po("chunk", 4)
lrns = ppl("greplicate", po("learner", lrn("classif.rpart")), 4)
```

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

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

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

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




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

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

```
classif.ce
 0.2083333
```

6.7.4 Feature Selection: `PipeOpFilter` and `PipeOpSelect`

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

A `PipeOp` for filters is `PipeOpFilter`:

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

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

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

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

6.8 In-depth look into mlr3pipelines

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

6.8.1 What's the Point

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

```
task = as_task_classif(iris, target = "Species")
lrn = lrn("classif.rpart")
rsmp = rsmp("holdout")
resample(task, lrn, rsmp)
```

```
<ResampleResult> with 1 resampling iterations
 task_id    learner_id resampling_id iteration warnings errors
```

```
    iris classif.rpart       holdout         1        0      0
```

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

6.8.2 `PipeOp`: Pipeline Operators

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

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

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

```
       Species       PC1         PC2         PC3          PC4
  1:    setosa -2.684126 -0.31939725  0.02791483 -0.002262437
  2:    setosa -2.714142  0.17700123  0.21046427 -0.099026550
  3:    setosa -2.888991  0.14494943 -0.01790026 -0.019968390
  4:    setosa -2.745343  0.31829898 -0.03155937  0.075575817
  5:    setosa -2.728717 -0.32675451 -0.09007924  0.061258593
 ---
146: virginica  1.944110 -0.18753230 -0.17782509 -0.426195940
147: virginica  1.527167  0.37531698  0.12189817 -0.254367442
148: virginica  1.764346 -0.07885885 -0.13048163 -0.137001274
149: virginica  1.900942 -0.11662796 -0.72325156 -0.044595305
150: virginica  1.390189  0.28266094 -0.36290965  0.155038628
```

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

```
   Species       PC1        PC2        PC3          PC4
1:  setosa -2.684126 -0.3193972 0.02791483 -0.002262437
```

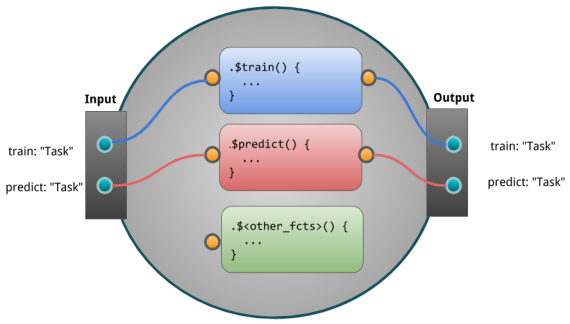
```
po$state
```

```
Standard deviations (1, .., p=4):
[1] 2.0562689 0.4926162 0.2796596 0.1543862
```

```
Rotation (n x k) = (4 x 4):
                    PC1         PC2         PC3         PC4
Petal.Length  0.85667061  0.17337266 -0.07623608  0.4798390
Petal.Width   0.35828920  0.07548102 -0.54583143 -0.7536574
Sepal.Length  0.36138659 -0.65658877  0.58202985 -0.3154872
Sepal.Width  -0.08452251 -0.73016143 -0.59791083  0.3197231
```

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

Schematically we can represent the `PipeOp` like so:



6.8.2.1 Why the `$state`

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

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

6.8.2.2 Where to get `PipeOp`s

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

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

```
              key input.num output.num
1:         boxcox         1          1
```

```
 2:         branch         1         NA
 3:          chunk         1         NA
 4: classbalancing         1          1
 5:     classifavg        NA          1
---
60:      threshold         1          1
61:  tunethreshold         1          1
62:       unbranch        NA          1
63:         vtreat         1          1
64:     yeojohnson         1          1
```

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

```
po("pca", rank. = 3)
```

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

6.8.3 PipeOp Channels

6.8.3.1 Input Channels

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

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

pofu = po("featureunion", innum = 2)

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

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

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

```
pofu$input
```

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

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

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

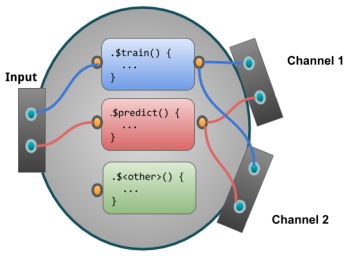
6.8.3.2 Output Channels

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

```
po("chunk", outnum = 3)$output
```

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

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



6.8.3.3 Channel Configuration

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

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

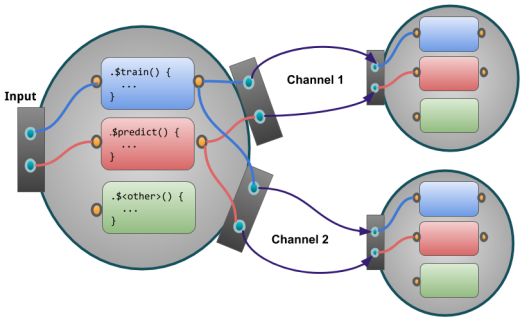
```
# po("backuplearner")
```

6.8.4 Graph: Networks of PipeOps

6.8.4.1 Basics

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

A schema of a simple graph of PipeOps:



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

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

```
print(gr)
```

```
Graph with 2 PipeOps:
```

```
        ID         State  sccssors prdcssors
     scale <<UNTRAINED>> subsample
 subsample <<UNTRAINED>>               scale
```

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

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

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

```
       Species Petal.Length   Petal.Width Sepal.Length Sepal.Width
 1:     setosa  -1.27910398 -1.3110521482  -1.01843718   0.7861738
 2:     setosa  -1.22245633 -1.3110521482  -1.25996379   0.7861738
```

```
 3:     setosa  -1.50569459 -1.4422448248  -1.86378030  -0.1315388
 4:     setosa  -1.16580868 -1.3110521482  -0.53538397   0.7861738
 5:     setosa  -1.44904694 -1.3110521482  -1.01843718   0.3273175
 6:     setosa  -1.39239929 -1.3110521482  -1.74301699  -0.1315388
 7: versicolor   0.42032558  0.3944526477   0.67224905   0.3273175
 8: versicolor  -0.14615094 -0.2615107354  -1.01843718  -2.4258204
 9: versicolor  -0.08950329  0.1320672944  -0.29385737  -0.3609670
10: versicolor   0.42032558  0.3944526477   0.43072244  -1.9669641
11: versicolor   0.42032558  0.3944526477   0.18919584  -0.3609670
12: versicolor   0.36367793  0.0008746178  -0.41462067  -1.0492515
13: versicolor   0.47697323  0.2632599711   0.30995914  -0.1315388
14: versicolor   0.13708732  0.0008746178  -0.05233076  -1.0492515
15:  virginica   1.04344975  1.1816087073   0.67224905  -0.5903951
```

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

```
   Species Petal.Length Petal.Width Sepal.Length Sepal.Width
1:  setosa    -1.335752   -1.311052   -0.8976739    1.015602
```

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

6.8.4.2 Networks

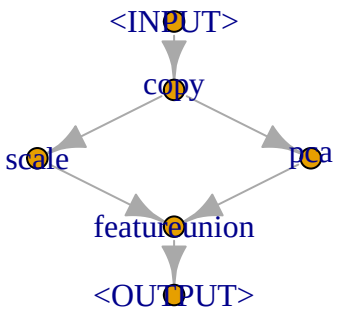
The example above showed a linear preprocessing pipeline, but it is in fact possible to build true "graphs" of operations, as long as no loops are introduced[1]. PipeOps with multiple output channels can feed their data to multiple different subsequent PipeOps, and PipeOps with multiple input channels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

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

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

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

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

```
12
13  gr$plot(html = FALSE)
```




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

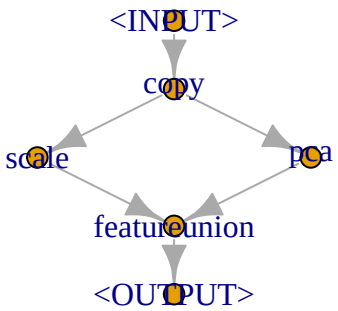
```
       Species Petal.Length Petal.Width       PC1          PC2
  1:    setosa   -1.3357516  -1.3110521 -2.561012 -0.006922191
  2:    setosa   -1.3357516  -1.3110521 -2.561012 -0.006922191
  3:    setosa   -1.3923993  -1.3110521 -2.653190  0.031849692
  4:    setosa   -1.2791040  -1.3110521 -2.468834 -0.045694073
  5:    setosa   -1.3357516  -1.3110521 -2.561012 -0.006922191
 ---
146: virginica    0.8168591   1.4439941  1.755953  0.455479438
147: virginica    0.7035638   0.9192234  1.416510  0.164312126
148: virginica    0.8168591   1.0504160  1.639637  0.178946130
149: virginica    0.9301544   1.4439941  1.940308  0.377935674
150: virginica    0.7602115   0.7880307  1.469915  0.033362474
```

6.8.4.3 Syntactic Sugar

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

next to each other akin to a (disjoint) graph union, the above network can more easily be constructed as follows:

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

gr$plot(html = FALSE)
```



6.8.4.4 `PipeOp` IDs and ID Name Clashes

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

```
po1 = po("scale")
po2 = po("scale")
po1 %>>% po2 # name clash
```

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

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

```
Graph with 2 PipeOps:
```

```
     ID         State sccssors prdcssors
  scale <<UNTRAINED>>   scale2
 scale2 <<UNTRAINED>>              scale
```

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

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

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

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

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

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

6.8.5 Learners in Graphs, Graphs in Learners

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

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

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

6.8.5.1 `PipeOpLearner`

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

The following is a very simple `Graph` that performs training and prediction on data after

performing principal component analysis.

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

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

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

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

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

6.8.5.2 GraphLearner

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

```
lrngrph = as_learner(gr)
resample(task, lrngrph, rsmp)
```

```
<ResampleResult> with 1 resampling iterations
 task_id        learner_id resampling_id iteration warnings errors
    iris pca.classif.rpart       holdout         1        0      0
```

6.8.6 Hyperparameters

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

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

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

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

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

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

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

```
op_pca = po("pca", center = TRUE)
op_pca$param_set$values
```

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

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

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

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

```
gr$param_set$values
```

```
$pca.center
[1] TRUE

$scale.robust
[1] FALSE
```

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

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

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

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

```
<ParamSetCollection>
                           id    class lower upper nlevels        default
 1:                  pca.center ParamLgl    NA    NA       2           TRUE
 2:                  pca.scale. ParamLgl    NA    NA       2          FALSE
 3:                   pca.rank. ParamInt     1   Inf     Inf
 4:           pca.affect_columns ParamUty    NA    NA     Inf  <Selector[1]>
 5:                scale.center ParamLgl    NA    NA       2           TRUE
 6:                 scale.scale ParamLgl    NA    NA       2           TRUE
 7:                scale.robust ParamLgl    NA    NA       2 <NoDefault[3]>
 8:         scale.affect_columns ParamUty    NA    NA     Inf  <Selector[1]>
 9:             classif.rpart.cp ParamDbl     0     1     Inf           0.01
10:     classif.rpart.keep_model ParamLgl    NA    NA       2          FALSE
11:     classif.rpart.maxcompete ParamInt     0   Inf     Inf              4
12:       classif.rpart.maxdepth ParamInt     1    30      30             30
13:   classif.rpart.maxsurrogate ParamInt     0   Inf     Inf              5
14:      classif.rpart.minbucket ParamInt     1   Inf     Inf <NoDefault[3]>
15:       classif.rpart.minsplit ParamInt     1   Inf     Inf             20
16: classif.rpart.surrogatestyle ParamInt     0     1       2              0
17:   classif.rpart.usesurrogate ParamInt     0     2       3              2
18:           classif.rpart.xval ParamInt     0   Inf     Inf             10
1 variable not shown: [value]
```

7

Beyond Regression and Classification

Raphael Sonabend
Imperial College London

Patrick Schratz
Friedrich-Schiller-University

Damir Pulatov
University of Wyoming

So far in this book we have only considered two tasks. In Chapter 2 we introduced deterministic regression as well as deterministic and probabilistic single-label classification (Table 7.1). But our infrastructure also works well for many other tasks, some of which are available in extension packages (**?@fig-mlr3verse**) and some are available by creating pipelines with `mlr3pipelines`. In this chapter, we will take you through just a subset of these new tasks, focusing on the ones that have a stable API. As we work through this chapter we will refer to the 'building blocks' of `mlr3`, this refers to the base classes that must be extended to create new tasks, these are `Prediction`, `Learner`, `Measure`, and `Task`.

Table 7.1 summarizes available extension tasks, including the package(s) they are implemented in and a brief description of the task.

Task	Package	Description
Deterministic regression	`mlr3`	Point prediction of a continuous variable.
Deterministic single-label classification	`mlr3`	Prediction of a single class for each observation.
Probabilistic single-label classification	`mlr3`	Prediction of the probability of an observation falling into one or more mutually exclusive categories.
Cost-sensitive classification	`mlr3` and `mlr3pipelines`	Classification predictions with unequal costs associated with misclassifications.
Survival analysis	`mlr3proba`	Time-to-event predictions with possible 'censoring'.
Density estimation	`mlr3proba`	Unsupervised estimation of probability density functions.
Spatiotemporal analysis	`mlr3spatiotempcv` and `mlr3spatial`	Supervised prediction of data with spatial (e.g., coordinates) and/or temporal outcomes.
Cluster analysis	`mlr3cluster`	Unsupervised estimation of homogeneous clusters of data points.

Task	Package	Description

Table 7.1: Table of extension tasks that can be used with `mlr3` infrastructure. As we have a growing community of contributors, this list is far from exhaustive and many 'experimental' task implementations exist; this list just represents the tasks that have a functioning interface.

7.1 Cost-Sensitive Classification

We begin by discussing a task that does not require any additional packages or infrastructure, only the tools we have already learned about from earlier chapters. In 'regular' classification, the aim is to optimize a metric (often the misclassification rate) whilst assuming all misclassification errors are deemed equally severe. A more general approach is cost-sensitive classification, in which costs caused by different kinds of errors may not be equal. The objective of cost-sensitive classification is to minimize the expected costs. As we discuss this task we will work with `mlr_tasks_german_credit` (Appendix C) as a running example.

Cost-sensitive Classification

Imagine you are trying to calculate if giving someone a loan of $5K will result in a profit after one year, assuming they are expected to pay back $6K. To make this calculation, you will need to be predict if the person will have good credit. This is a deterministic classification problem where we are predicting whether someone will be in class 'Good' or 'Bad'. Now we can define the potential costs associated with each prediction and the eventual truth:

```
costs = matrix(c(-1, 0, 5, 0), nrow = 2, dimnames =
  list("Predicted Credit" = c("good", "bad"),
    Truth = c("good", "bad")))
costs
```

```
                Truth
Predicted Credit good bad
            good   -1   5
            bad     0   0
```

If the model predicts that the individual has bad credit then there is no profit or loss, the loan is not provided. If the model predicts that the individual has good credit and they pay the loan and interest then you make a $1K profit, but if they default then you lose $5K. Note that cost-sensitive classification is a minimization problem where we assume lower costs mean higher profits/positive outcomes, hence above we wrote the profit and loss as `-1` and `+5` respectively.

7.1.1 Cost-sensitive Measure

We will now see how to implement a more nuanced approach to classification errors with `MeasureClassifCosts`. This measure takes one argument, which is a matrix with row and column names corresponding to the class labels in the task of interest. Let's put our insurance example into practice, notice that we have already named the cost matrix as required for the measure:

```
library(mlr3verse)

task = tsk("german_credit")

cost_measure = msr("classif.costs", costs = costs)
cost_measure
```

```
<MeasureClassifCosts:classif.costs>: Cost-sensitive Classification
* Packages: mlr3
* Range: [-Inf, Inf]
* Minimize: TRUE
* Average: macro
* Parameters: normalize=TRUE
* Properties: -
* Predict type: response
```

```
learners = lrns(c("classif.log_reg", "classif.featureless", "classif.ranger"))
bmr = benchmark(benchmark_grid(task, learners, rsmp("cv", folds = 3)))
bmr$aggregate(cost_measure)[, c(4, 7)]
```

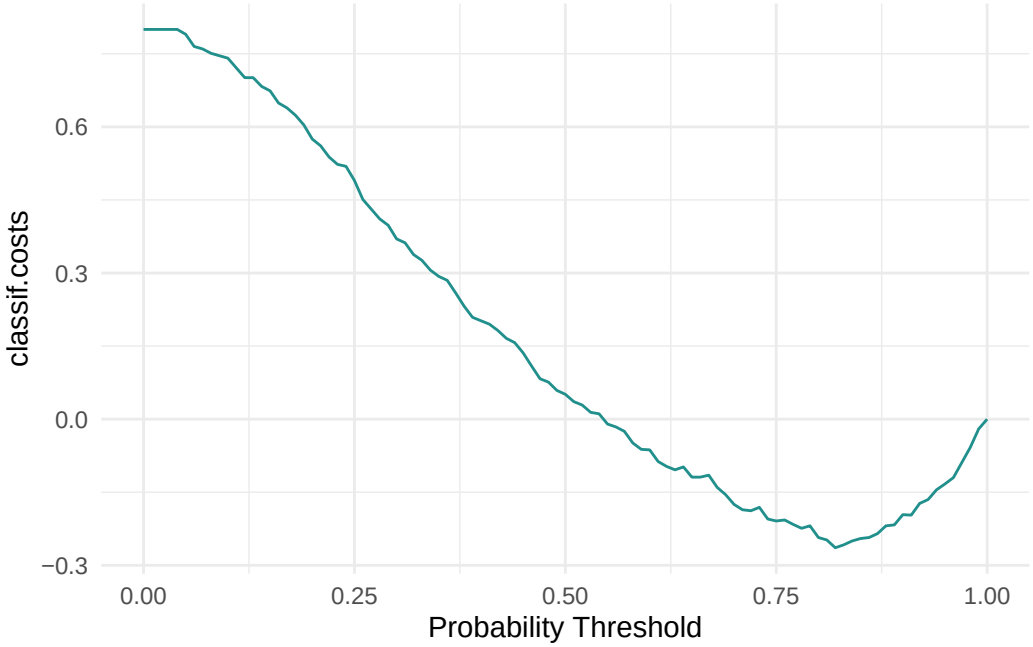
```
           learner_id classif.costs
1:     classif.log_reg     0.1790683
2: classif.featureless     0.8001654
3:      classif.ranger     0.2491294
```

In this experiment we find that the logistic regression learner happens to perform best as it minimizes the expected costs (and maximizes expected profits) and the featureless learner performs the worst. However all costs are positive, which actually means a loss is made, so let's now see if we can improve these models by using thresholding.

7.1.2 Thresholding

As we have discussed in Chapter 2, thresholding is a method to fine-tune the probability at which an observation will be predicted as one class label or another. Currently in our running example, the models above will predict a customer has good credit (in the class 'Good') if the probability of good credit is greater than 0.5. However, this may not be a sensible approach as the cost of a false positive and false negative is not equal. In fact, a false positive results in a cost of +5 whereas a false negative only results in a cost of 0, hence we would prefer a model with a higher false negative rates and lower false positive rates. This is highlighted in the `"threshold"` `autoplot`:

```
prediction = lrn("classif.log_reg",
  predict_type = "prob")$train(task)$predict(task)
autoplot(prediction, type = "threshold", measure = cost_measure)
```



As expected, the optimal threshold is greater than 0.5, indicating that to maximize profits, the majority of observations should be predicted to have bad credit.

To automate the process of optimizing the threshold, we can make use of `mlr3tuning` (Chapter 4) and `mlr3pipelines` (Chapter 6) by creating a graph with `PipeOpTuneThreshold`. Continuing the same example:

```
gr = po("learner_cv", lrn("classif.log_reg", predict_type = "prob")) %>>%
  po("tunethreshold", measure = cost_measure)

learners = list(as_learner(gr), lrn("classif.log_reg"))
bmr = benchmark(benchmark_grid(task, learners, rsmp("cv", folds = 3)))
bmr$aggregate(cost_measure)[, c(4, 7)]
```

```
                    learner_id classif.costs
1: classif.log_reg.tunethreshold    -0.1060282
2:               classif.log_reg     0.1481062
```

As expected, our tuned learner performs much better and now we expect a profit and not a loss.

7.2 Survival Analysis

Survival Analysis

Survival analysis is a field of statistics concerned with trying to predict/estimate the time until an event takes place. This predictive problem is unique as survival models are trained and tested on data that may include 'censoring', which occurs when the event of interest does

not take place. Survival analysis can be hard to explain in the abstract, so as a working example consider a marathon runner in a race. Here the 'survival problem' is trying to predict the time when the marathon runner finishes the race. However, if the event of interest does not take place (i.e., marathon runner gives up and does not finish the race), they are said to be censored. Instead of throwing away information about censored events, survival analysis datasets include a status variable that provides information about the 'status' of an observation. So in our example we might write the runner's outcome as $(4, 1)$ if they finish the race at 4 hours, otherwise if they give up at 2 hours we would write $(2, 0)$.

The key to modelling in survival analysis is that we assume there exists a hypothetical time the marathon runner would have finished if they had not been censored, it is then the job of a survival learner to estimate what the true survival time would have been for a similar runner, assuming they are *not* censored (see Figure 7.1). Mathematically, this is represented by the hypothetical event time, Y, the hypothetical censoring time, C, the observed outcome time, $T = min(Y, C)$, the event indicator $\Delta = (T = Y)$, and as usual some features, X. Learners are trained on (T, Δ) but, critically, make predictions of Y from previously unseen features. This means that unlike classification and regression, learners are trained on two variables, (T, Δ), which, in R, is often captured in a `survival::Surv` object. Relating to our example above the runner's outcome would then be $(T = 4, \Delta = 1)$ or $(T = 2, \Delta = 0)$. Another example is in the code below, where we randomly generate 6 survival times and 6 event indicators, an outcome with a `+` indicates the outcome is censored, otherwise the event of interest occurred.

```
library(survival)
Surv(runif(6), rbinom(6, 1, 0.5))
```

```
[1] 0.5522635+ 0.2905186  0.4404405+ 0.1184443  0.9216186+ 0.7325895
```

Readers familiar with survival analysis will recognize that the description above applies specifically to 'right censoring'. Currently, this is the only form of censoring available in the `mlr3` universe, hence restricting our discussion to that setting. For a good introduction to survival analysis see Collett (2014) or for machine learning in survival analysis specifically see R. Sonabend and Bender (2023).

For the remainder of this section we will look at how `mlr3proba` (R. Sonabend et al. 2021) extends the building blocks of `mlr3` for survival analysis. We will begin by looking at objects used to construct machine learning tasks for survival analysis, then we will turn to the learners we have implemented to solve these tasks, before looking at measures for evaluating survival analysis predictions, and then finally we will consider how to transform prediction types.

7.2.1 TaskSurv

As we saw in the introduction to this section, survival algorithms require two targets for training, this means the new `TaskSurv` object expects two targets. The simplest way to create a survival task is to use `as_task_surv`, as in the following code chunk. Note this has more arguments than `as_task_regr` to reflect multiple target and censoring types, `time` and `event` arguments expect strings representing column names where the 'time' and 'event' variables are stored, `type` refers to the censoring type (currently only right censoring supported so this is the default). Note how `as_task_surv` coerces the target columns into a `survival::Surv` object.

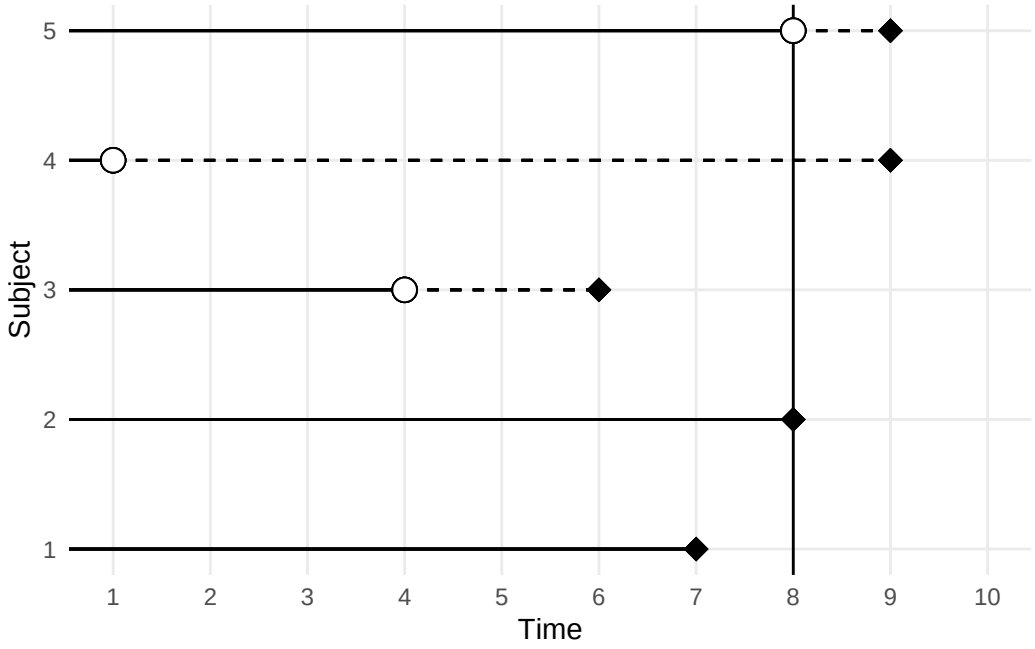


Figure 7.1: Plot illustrating different censoring types. Dead and censored subjects (y-axis) over time (x-axis). Black diamonds indicate true death times and white circles indicate censoring times. Vertical line is the study end time. Subjects 1 and 2 die in the study time. Subject 3 is censored in the study and (unknown) dies within the study time. Subject 4 is censored in the study and (unknown) dies after the study. Subject 5 dies after the end of the study. Figure and caption from R. E. B. Sonabend (2021).

```
library("mlr3verse")
library("mlr3proba")
library("survival")

task = as_task_surv(survival::rats, time = "time",
  event = "status", type = "right", id = "rats")

task$head()
```

```
   time status litter rx sex
1:  101      0      1  1   f
2:   49      1      1  0   f
3:  104      0      1  0   f
4:   91      0      2  1   m
5:  104      0      2  0   m
6:  102      0      2  0   m
```

Plotting the task with `autoplot` results in a Kaplan-Meier plot which is a non-parametric estimator of the probability of survival for the average observation in the training set.

```
autoplot(task)
```

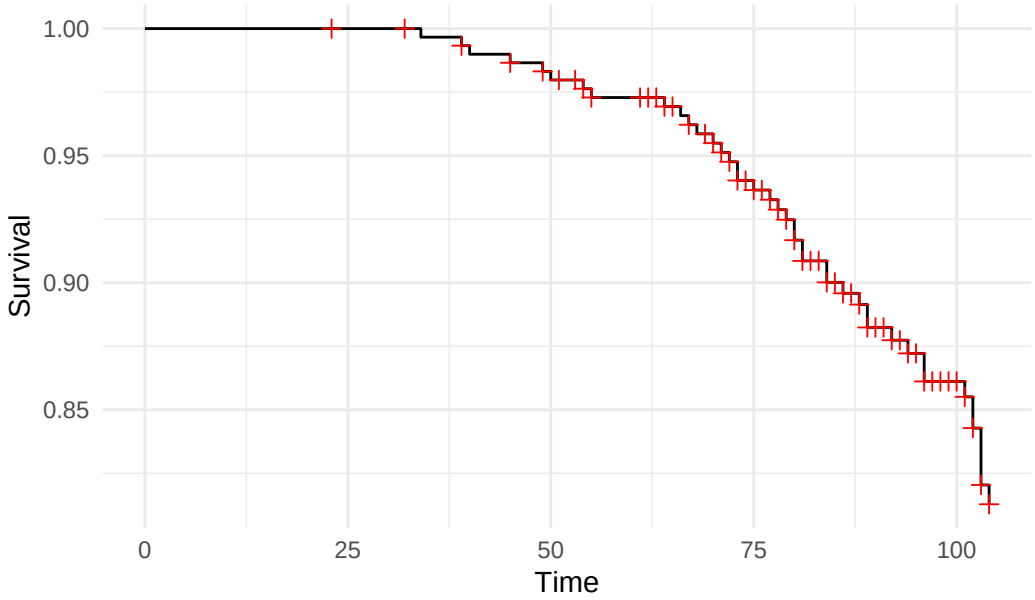


Figure 7.2: Kaplan-Meier plot of the `rats` task. x-axis is time variable and y-axis is survival function, S(T), defined by 1 - F(T) where F is the cumulative distribution function. Red crosses indicate points where censoring takes place.

In the above example we used the `survival::rats` dataset, which looks at predicting if a drug treatment was successful in preventing 150 rats from developing tumors. Note that the

dataset (by its own admission) is not perfect and should generally be treated as 'dummy' data, which is good for examples but not real-world analysis.

As well as creating your own tasks, you can load any of the tasks shipped with `mlr3proba`:

```
as.data.table(mlr_tasks)[task_type == "surv"]
```

```
            key                  label task_type nrow ncol properties lgl int
1:         actg                ACTG 320      surv 1151   13              0   3
2:         gbcs   German Breast Cancer      surv  686   10              0   4
3:        grace              GRACE 1000      surv 1000    8              0   2
4:         lung             Lung Cancer      surv  228   10              0   7
5:         rats                   Rats      surv  300    5              0   2
6: unemployment  Unemployment Duration      surv 3343    6              0   1
7:         whas Worcester Heart Attack      surv  481   11              0   4
5 variables not shown: [dbl, chr, fct, ord, pxc]
```

7.2.2 LearnerSurv, PredictionSurv and predict types

The interface for `LearnerSurv` and `PredictionSurv` objects is identical to the regression and classification settings discussed in Chapter 2. Similarly to these settings, survival learners are constructed with `"lrn"`; available learners are listed in Appendix D.

`mlr3proba` has a different predict interface to `mlr3` as all possible types of prediction ('predict types') are returned when possible for all survival models – i.e., if a model can compute a predict type then it is returned in `PredictionSurv`. The reason for this design decision is that all these predict types can be transformed to one another and it is therefore computationally simpler to return all at once instead of rerunning models to change predict type. In survival analysis, the following predictions can be made:

- `response` - Predicted survival time.
- `distr` - Predicted survival distribution, either discrete or continuous.
- `lp` - Linear predictor calculated as the fitted coefficients multiplied by the test data.
- `crank` - Continuous risk ranking.

We will go through each of these prediction types in more detail and with examples to make them less abstract. We will use the following setup for most of the examples. In this chunk we are partitioning the `survival::rats` and training a Cox Proportional Hazards model (`mlr_learners_surv.coxph`) on the training set and making predictions for the predict set. For this model, all predict types except `response` can be computed.

Cox Proportional Hazards

```
t = tsk("rats")
split = partition(t)
p = lrn("surv.coxph")$train(t, split$train)$predict(t, split$test)
p
```

```
<PredictionSurv> for 99 observations:
    row_ids time status      crank         lp     distr
          8  102  FALSE -0.1577286 -0.1577286 <list[1]>
         16   98  FALSE -1.9548867 -1.9548867 <list[1]>
         24   76  FALSE -2.7149884 -2.7149884 <list[1]>
---
```

```
241   72   TRUE  0.8826867  0.8826867 <list[1]>
247   73   TRUE  0.8896945  0.8896945 <list[1]>
249   66   TRUE  0.1225849  0.1225849 <list[1]>
```

7.2.2.1 `predict_type = "response"`

Counterintuitively for many, the `response` prediction of predicted survival times is actually the least common predict type in survival analysis. The likely reason for this is due to the presence of censoring. We rarely observe the true survival time for many observations and therefore it is unlikely any survival model can confidently make predictions for survival times. This is illustrated in the code below.

In the example below we train and predict from a survival support vector machine (`mlr_learners_surv.svm`), note we use `type = "regression"` to select the algorithm that optimizes survival time predictions and `gamma.mu = 1e-3` is selected arbitrarily as this is a required parameter (this parameter should usually be tuned). We then compare the predictions from the model to the true data.

```
library(mlr3extralearners)
pred = lrn("surv.svm", type = "regression", gamma.mu = 1e-3)$
  train(t, split$train)$predict(t, split$test)
data.frame(pred = pred$response[1:3], truth = pred$truth[1:3])
```

```
      pred truth
1 87.56067  102+
2 86.97710   98+
3 86.58935   76+
```

As can be seen from the output, our predictions are all less than the true observed time, which means we know our model definitely underestimated the truth. However, because each of the true values are censored times, we have absolutely no way of knowing if these predictions are slightly bad or absolutely terrible, (i.e., the true survival times could be $105, 99, 92$ or they could be $300, 1000, 200$). Hence, with no realistic way to evaluate these models, survival time predictions are rarely useful.

7.2.2.2 `predict_type = "distr"`

So unlike regression in which deterministic/point predictions are most common, in survival analysis distribution predictions are much more common. You will therefore find that the majority of survival models in `mlr3proba` will make distribution predictions by default. These predictions are implemented using the `distr6` package, which allows visualization and evaluation of survival curves (defined as 1 - cumulative distribution function). In the example below we train a Cox PH model on the `rats` dataset and then evaluate the survival function for three predictions at $t = 77$.

```
t = tsk("rats")
split = partition(t)
p = lrn("surv.coxph")$train(t, split$train)$predict(t, split$test)
p$distr[1:3]$survival(77)
```

```
       [,1]      [,2]      [,3]
77 0.959636 0.9925459 0.9586408
```

The output indicates that there is a 96%, 99.3%, 95.9%, chance of the first three predicted rats being alive at time 77 respectively.

7.2.2.3 `predict_type = "lp"`

`lp`, often written as η in academic writing, is computationally the simplest prediction and has a natural analogue in regression modelling. Readers familiar with linear regression will know that when fitting a simple linear regression model, $Y = X\beta$, we are actually estimating the values for β, and the estimated linear predictor (lp) is then $X\hat{\beta}$, where $\hat{\beta}$ are our estimated coefficients. In simple survival models, the linear predictor is the same quantity (but estimated in a slightly more complicated way). The learner implementations in `mlr3proba` are primarily machine-learning focused and few of these models have a simple linear form, which means that `lp` cannot be computed for most of these. In practice, when used for prediction, `lp` is a proxy for a relative risk/continuous ranking prediction, which is discussed next.

7.2.2.4 `predict_type = "crank"`

The final prediction type, `crank`, is the most common in survival analysis and perhaps also the most confusing. Academic texts will often refer to 'risk' predictions in survival analysis (hence why survival models are often known as 'risk prediction models'), without defining what 'risk' means. Often risk is defined as $exp(\eta)$ as this is a common quantity found in simple linear survival models. However, sometimes risk is defined as $exp(-\eta)$, and sometimes it can be an arbitrary quantity that does not have a meaningful interpretation. To prevent this confusion in `mlr3proba`, we define the predict type `crank`, which stands for **c**ontinuous **rank**ing. This is best explained by example.

> ⚠ Warning
>
> The interpretation of 'risk' for survival predictions differs across R packages and sometimes even between models in the same package. In `mlr3proba` there is one consistent interpretation of `crank`: lower values represent a lower risk of the event taking place and higher values represent higher risk.

Continuing from the previous example we output the first three `crank` predictions. The output tells us that the second rat is at the higher risk of death (larger values represent higher risk) and the third rat is at the lowest risk of death. The distance between predictions also tells us that the difference in risk between the first and second rat is smaller than the difference between the second and third. The actual values themselves are meaningless and therefore comparing `crank` values between samples (or papers or experiments) is not meaningful.

```
p$crank[1:3]
```

```
         1          2          3
-0.5893286 -2.2952962 -0.5644561
```

The `crank` prediction type is informative and common in practice because it allows identifying observations at lower/higher risk to each other, which is useful for resource allocation and prioritization (e.g., which patient should be given an expensive treatment), and clinical trials (e.g., are people in a treatment arm at lower risk of disease X than people in the control arm.).

7.2.3 MeasureSurv

Survival models in `mlr3proba` are evaluated with `MeasureSurv` objects, which are constructed in the usual way with `"msr"`; measures currently implemented are listed in see Appendix D.

In general survival measures can be grouped into the following:

1. Discrimination measures – Quantify if a model correctly identifies if one observation is at higher risk than another. Evaluate `crank` and/or `lp` predictions.
2. Calibration measures – Quantify if the average prediction is close to the truth (all definitions of calibration are unfortunately vague in a survival context). Evaluate `crank` and/or `distr` predictions.
3. Scoring rules – Quantify if probabilistic predictions are close to true values. Evaluates `distr` predictions.

```
head(as.data.table(mlr_measures)[
  task_type == "surv", c("key", "predict_type")])
```

```
                key predict_type
1:         surv.brier        distr
2:   surv.calib_alpha        distr
3:    surv.calib_beta           lp
4: surv.chambless_auc           lp
5:        surv.cindex        crank
6:        surv.dcalib        distr
```

There is a lot of debate in the literature around the 'best' survival measures to use to evaluate models, as a general rule we recommend RCLL (`mlr_measures_surv.rcll`) to evaluate the quality of `distr` predictions, concordance index (`mlr_measures_surv.cindex`) to evaluate a model's discrimination, and D-Calibration (`mlr_measures_surv.dcalib`) to evaluate a model's calibration.

We can now evaluate our predictions from the previous example. In the code below we use the recommend measures and find this model's performance seems okay as the RCLL and DCalib are relatively low 0 and the C-index is greater than 0.5.

```
p$score(msrs(c("surv.rcll", "surv.cindex", "surv.dcalib")))
```

```
surv.rcll surv.cindex surv.dcalib
3.7869591   0.7433128   1.2615224
```

7.2.4 Composition

Throughout `mlr3proba` documentation we refer to "native" and "composed" predictions. We define a 'native' prediction as the prediction made by a model without any post-processing, whereas a 'composed' prediction is one that is returned after post-processing.

7.2.4.1 Internal composition

In `mlr3proba` we make use of composition internally to return a `"crank"` prediction for every learner. This is to ensure that we can meaningfully benchmark all models according

to at least one criterion. The package uses the following rules to create `"crank"` predictions:

1. If a model returns a 'risk' prediction then `crank = risk` (we may multiply this by -1 to ensure the 'low value low risk' interpretation).
2. Else if a model returns a `response` prediction then we set `crank = -response`.
3. Else if a model returns a `lp` prediction then we set `crank = lp` (or `crank = -lp` if needed).
4. Else if a model returns a `distr` prediction then we set `crank` as the sum of the cumulative hazard function (see R. Sonabend, Bender, and Vollmer (2022) for full discussion as to why we picked this method).

7.2.4.2 Explicit composition and pipelines

At the start of this section we mentioned that it is possible to transform prediction types between each other. In `mlr3proba` this is possible with 'compositor' pipelines (Chapter 6). There are a number of pipelines implemented in the package but two in particular focus on predict type transformation:

1. `pipeline_crankcompositor()` - Transforms a `"distr"` prediction to `"crank"`; and
2. `pipeline_distrcompositor()` - Transforms a `"lp"` prediction to `"distr"`.

In practice, the second pipeline is more common as we internally use a version of the first pipeline whenever we return predictions from survival models (so only use the first pipeline to overwrite these ranking predictions), and so here we will just look at the second pipeline.

In the example below we load the `rats` dataset, remove factor columns, and then partition the data into training and testing. We construct the `distrcompositor` pipeline around a survival GLMnet learner (`mlr_learners_surv.glmnet`) which by default can only make predictions for `"lp"` and `"crank"`. In the pipeline we specify that we will estimate the baseline distribution with a Kaplan-Meier estimator (`estimator = "kaplan"`) and that we want to assume a proportional hazards form for our estimated distribution. We then train and predict in the usual way and in our output we can now see a `distr` prediction.

```
library(mlr3verse)
library(mlr3extralearners)

task = tsk("rats")
task$select(c("litter", "rx"))
split = partition(task)

learner = lrn("surv.glmnet")

# no distr output
learner$train(task, split$train)$predict(task, split$test)
```

```
<PredictionSurv> for 99 observations:
    row_ids time status     crank.1        lp.1
          1  101  FALSE 0.707318789 0.707318789
          3  104  FALSE 0.003141698 0.003141698
          4   91  FALSE 0.710460487 0.710460487
```

```
---
        230   84   TRUE 0.241910708 0.241910708
        235   80   TRUE 0.952371195 0.952371195
        293   75   TRUE 0.307886356 0.307886356
```

```
pipe = as_learner(ppl(
  "distrcompositor",
  learner = learner,
  estimator = "kaplan",
  form = "ph"
))

# now with distr
pipe$train(task, split$train)$predict(task, split$test)
```

```
<PredictionSurv> for 99 observations:
    row_ids time status     crank.1        lp.1     distr
          1  101  FALSE 0.707318789 0.707318789 <list[1]>
          3  104  FALSE 0.003141698 0.003141698 <list[1]>
          4   91  FALSE 0.710460487 0.710460487 <list[1]>
---
        230   84   TRUE 0.241910708 0.241910708 <list[1]>
        235   80   TRUE 0.952371195 0.952371195 <list[1]>
        293   75   TRUE 0.307886356 0.307886356 <list[1]>
```

Mathematically, we have done the following:

1. Assume our estimated distribution will have the form $h(t) = h_0(t)exp(\eta)$ where h is the hazard function and h_0 is the baseline hazard function.
2. Estimate $\hat{\eta}$ prediction using GLMnet
3. Estimate $\hat{h}_0(t)$ with the Kaplan-Meier estimator
4. Put this all together as $h(t) = \hat{h}_0(t)exp(\hat{\eta})$

For more details about prediction types and compositions we recommend Kalbfleisch and Prentice (2011).

7.2.5 Putting it all together

Finally, we will put all the above into practice in a small benchmark experiment. We first load the `grace` dataset and subset to the first 500 rows. We then select the RCLL, D-Calibration, and C-index to evaluate predictions, setup the same pipeline we used in the previous experiment, and load a Cox PH and Kaplan-Meier estimator. We run our experiment with 3-fold cross-validation and aggregate the results.

```
library(mlr3verse)
library(mlr3extralearners)

task = tsk("grace")$filter(1:500)
msr_txt = c("surv.rcll", "surv.cindex", "surv.dcalib")
measures = msrs(msr_txt)
```

```
7
8  pipe = as_learner(ppl(
9    "distrcompositor",
10   learner = lrn("surv.glmnet"),
11   estimator = "kaplan",
12   form = "ph"
13 ))
14 pipe$id = "Coxnet"
15 learners = c(lrns(c("surv.coxph", "surv.kaplan")), pipe)
16
17 bmr = benchmark(benchmark_grid(task, learners, rsmp("cv", folds = 3)))
18 bmr$aggregate(measures)[, c("learner_id", ..msr_txt)]
```

```
    learner_id surv.rcll surv.cindex surv.dcalib
1:  surv.coxph  2.937825   0.8016593    2.004290
2: surv.kaplan  3.094982   0.5000000    3.224777
3:      Coxnet  2.940040   0.8021212    5.658987
```

In this small experiment, Coxnet and Cox PH have the best discrimination with $C \approx 0.82$, Cox PH has the best calibration (as DCalib is closest to 0), and Coxnet and Cox PH have similar overall predictive accuracy (with lowest RCLL).

7.3 Density Estimation

Density estimation is a learning task to estimate the unknown distribution from which a univariate dataset is generated, or put more simply to estimate the probability density (or mass) function for a single variable. As with survival analysis, density estimation is implemented in `mlr3proba`, as both can make probability distribution predictions (hence the name "**mlr3proba**bilistic"). Unconditional density estimation (i.e. estimation of a target without any covariates) is viewed as an unsupervised task, which means the 'truth' is never known. For a good overview to density estimation see Silverman (1986).

The package `mlr3proba` extends `mlr3` with the following objects for density estimation:

- `TaskDens` to define density tasks.
- `LearnerDens` as the base class for density estimators.
- `PredictionDens` for density predictions.
- `MeasureDens` as a specialized class for density performance measures.

We will consider each in turn.

7.3.1 TaskDens

As density estimation is an unsupervised task, there is no target for prediction. In the code below we construct a density task using `as_task_dens` which takes one argument, a `data.frame` type object with exactly one column (which we will use to estimate the underlying distribution).

```
task = as_task_dens(data.table(x = rnorm(1000)))
task
```

```
<TaskDens:data.table(x = rnorm(1000))> (1000 x 1)
* Target: -
* Properties: -
* Features (1):
  - dbl (1): x
```

As with other tasks, we have included a couple of tasks that come shipped with `mlr3proba`:

```
as.data.table(mlr_tasks)[task_type == "dens", c(1:2, 4:5)]
```

```
       key                  label nrow ncol
1: faithful Old Faithful Eruptions  272    1
2:   precip   Annual Precipitation   70    1
```

7.3.2 LearnerDens and PredictionDens

Density learners can make one of three possible prediction types:

1. `distr` - probability distribution
2. `pdf` - probability density function
3. `cdf` - cumulative distribution function

All learners will return a `distr` and `pdf` prediction but only some can make `cdf` predictions. Again, the `distr` predict type is implemented using `distr6`.

```
learn = lrn("dens.hist")
p = learn$train(task, 1:900)$predict(task, 901:1000)
x = seq.int(-2, 2, 0.01)
ggplot(data.frame(x = x, y = p$distr$pdf(x)), aes(x = x, y = y)) +
  geom_line() + theme_minimal()
```

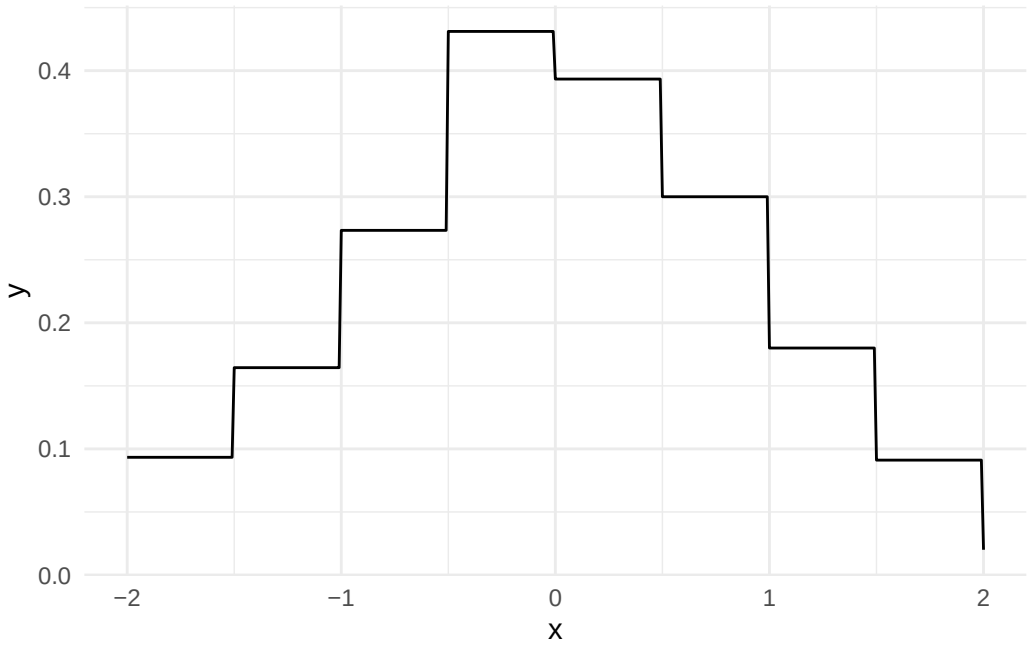


Figure 7.3: Predicted density from the histogram learner, as expected this closely resembles the underlying N(0, 1) data.

The `pdf` and `cdf` predict types are simply wrappers around `distr$pdf` and `distr$cdf` respectively, which is best demonstrated by example:

```
learn = lrn("dens.hist")
p = learn$train(task, 1:10)$predict(task, 11:13)
p
```

```
<PredictionDens> for 3 observations:
 row_ids pdf       cdf              distr
      11 0.2 0.5933786 <Distribution[39]>
      12 0.8 0.3599538 <Distribution[39]>
      13 0.0 0.0000000 <Distribution[39]>
```

```
cbind(p$distr$cdf(task$data()$x[11:13]), p$cdf[1:3])
```

```
          [,1]      [,2]
[1,] 0.5933786 0.5933786
[2,] 0.3599538 0.3599538
[3,] 0.0000000 0.0000000
```

The reason for returning `pdf` and `cdf` in this way is to support measures that can be used to evaluate the quality of our estimations, which we will return to in the next section.

7.3.3 MeasureDens

Currently the only measure implemented in `mlr3proba` for density estimation is logloss, which is defined in the same way as in classification, $L(y) = -log(\hat{f}_Y(y))$, where $\hat{f}_Y$ is our estimated probability density function. Putting this together with the above we are now ready to train a density learner, estimate a distribution, and evaluate our estimation:

```
meas = msr("dens.logloss")
meas
```

```
<MeasureDensLogloss:dens.logloss>: Log Loss
* Packages: mlr3, mlr3proba
* Range: [0, Inf]
* Minimize: TRUE
* Average: macro
* Parameters: eps=1e-15
* Properties: -
* Predict type: pdf
```

```
p$score(meas)
```

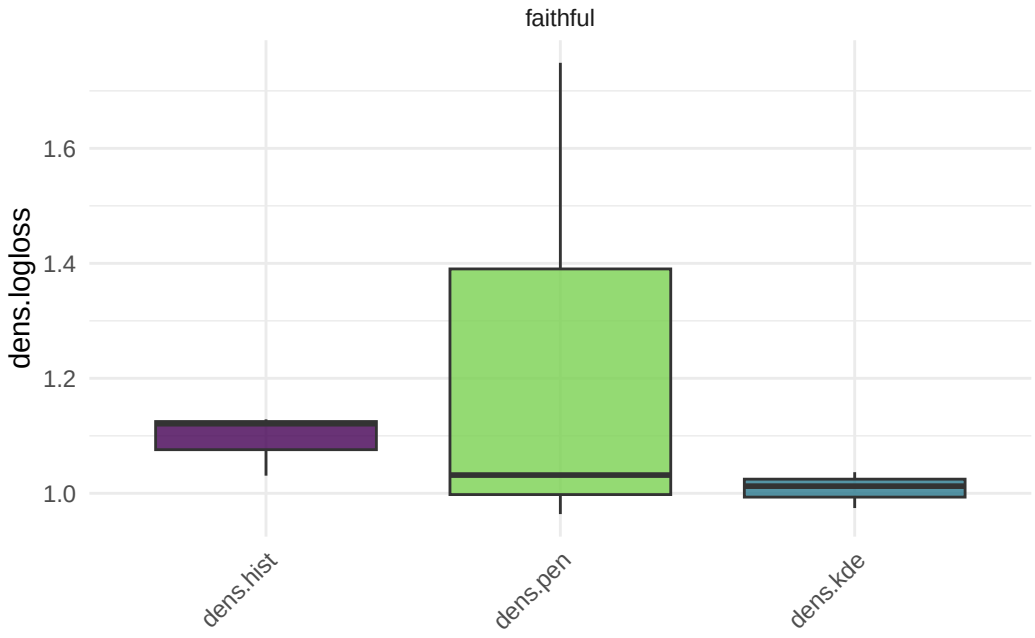
```
dens.logloss
    12.12379
```

As with any scoring rule this output cannot be interpreted by itself and is more useful in a benchmark experiment, which we will look at in the final part of this section.

7.3.4 Putting it all together

Finally, we conduct a small benchmark study on the `mlr_tasks_faithful` task using some of the integrated density learners:

```
library(mlr3extralearners)
task = tsk("faithful")
learners = lrns(c("dens.hist", "dens.pen", "dens.kde"))
measure = msr("dens.logloss")
bmr = benchmark(benchmark_grid(task, learners, rsmp("cv", folds = 3)))
bmr$aggregate(measure)
autoplot(bmr, measure = measure)
```



The results of this experiment show that the sophisticated Penalized Density Estimator does not outperform the baseline histogram, but that the Kernel Density Estimator has at least consistently better (i.e. lower logloss) results.

7.4 Cluster Analysis

Cluster Analysis

Cluster analysis is another unsupervised task implemented in `mlr3`. The objective of cluster analysis is to group data into clusters, where each cluster contains similar observations. The similarity is based on specified metrics that are task and application-dependent. Unlike classification where we try to predict a class for each observation, in cluster analysis there is no 'true' label or class to predict.

The package `mlr3cluster` extends `mlr3` with the following objects for cluster analysis:

- `TaskClust` to define clustering tasks
- `LearnerClust` as base class for clustering learners
- `PredictionClust` as specialized class for `Prediction` objects
- `MeasureClust` as specialized class for performance measures

We will consider each in turn.

7.4.1 TaskClust

Similarly to density estimation (Section 7.3), there is no target for prediction and so no `truth` field in `TaskClust`. Let's look at the `cluster::ruspini` dataset often used for cluster analysis examples.

```
library(cluster)
head(ruspini)
```

```
   x  y
1  4 53
2  5 63
3 10 59
4  9 77
5 13 49
6 13 69
```

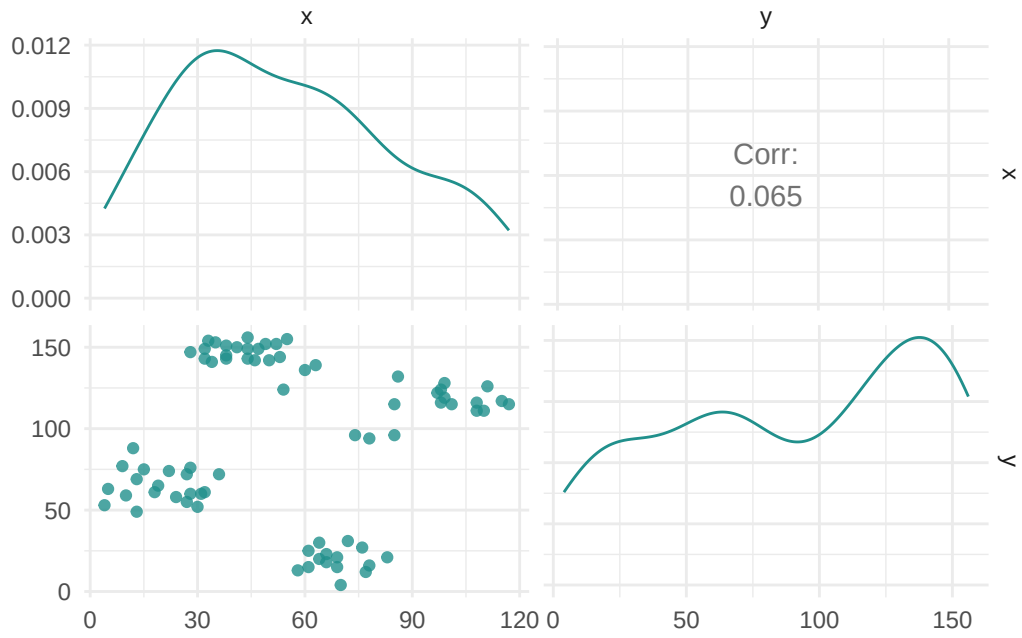
The dataset has 75 rows and two columns and was first introduced in Ruspini (1970) to illustrate different clustering techniques. As we will see from the plots below, the observations form four natural clusters.

In the code below we construct a cluster task using `as_task_clust` which only takes one argument, a `data.frame` type object.

```
library(mlr3verse)
library(cluster)
task = as_task_clust(ruspini)
task
```

```
<TaskClust:ruspini> (75 x 2)
* Target: -
* Properties: -
* Features (2):
  - int (2): x, y
```

```
autoplot(task)
```



Technically, we did not need to create a new task for ruspini dataset since it is already included in the package. Currently we have two clustering tasks shipped with `mlr3cluster`:

```
as.data.table(mlr_tasks)[task_type == "clust", c(1:2, 4:5)]
```

```
         key      label nrow ncol
1:   ruspini    Ruspini   75    2
2: usarrests US Arrests   50    4
```

7.4.2 LearnerClust and PredictionClust

As with density estimation, we refer to `training` and `predicting` for clustering to be consistent with the `mlr3` interface, but strictly speaking this should be `clustering` and `assigning` (the latter we will return to shortly). Two `predict_types` are available for clustering learners:

1. `partition` – estimate of which cluster an observation falls into
2. `prob` – probability of an observation belonging to each cluster

Hence, similarly to classification, prediction types of clustering learners are either deterministic (`partition`) or probabilistic (`prob`).

Below we construct a C-Means clustering learner with `prob` prediction type and 3 clusters, train it on the `cluster::ruspini` dataset and then return the cluster assignments (`$assignments`) for each observation.

```
learner = lrn("clust.cmeans", predict_type = "prob", centers = 3)
learner
```

```
<LearnerClustCMeans:clust.cmeans>: Fuzzy C-Means Clustering Learner
* Model: -
* Parameters: centers=3
* Packages: mlr3, mlr3cluster, e1071
* Predict Types:  partition, [prob]
* Feature Types: logical, integer, numeric
* Properties: complete, fuzzy, partitional
```

```
learner$train(task)
learner$assignments[1:6]
```

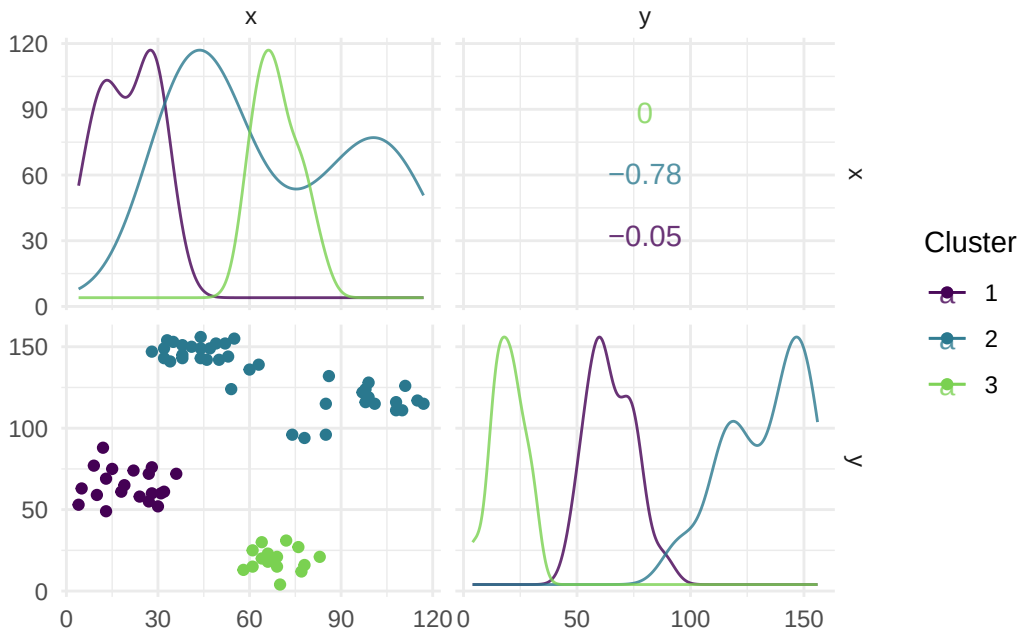
```
[1] 2 2 2 2 2 2
```

As clustering is unsupervised, it often does not make sense to use `predict` for new data however this is still possible using the `mlr3` interface.

```
# using different data for estimation (rare use case)
learner$train(task, 1:30)$predict(task, 31:32)
```

```
<PredictionClust> for 2 observations:
 row_ids partition    prob.1      prob.2    prob.3
      31         3 0.2746426 0.008320118 0.7170373
      32         3 0.3716548 0.006463253 0.6218819
```

```
# using same data for estimation (common use case)
prediction = learner$train(task)$predict(task)
autoplot(prediction, task)
```



Hierarchical Clustering

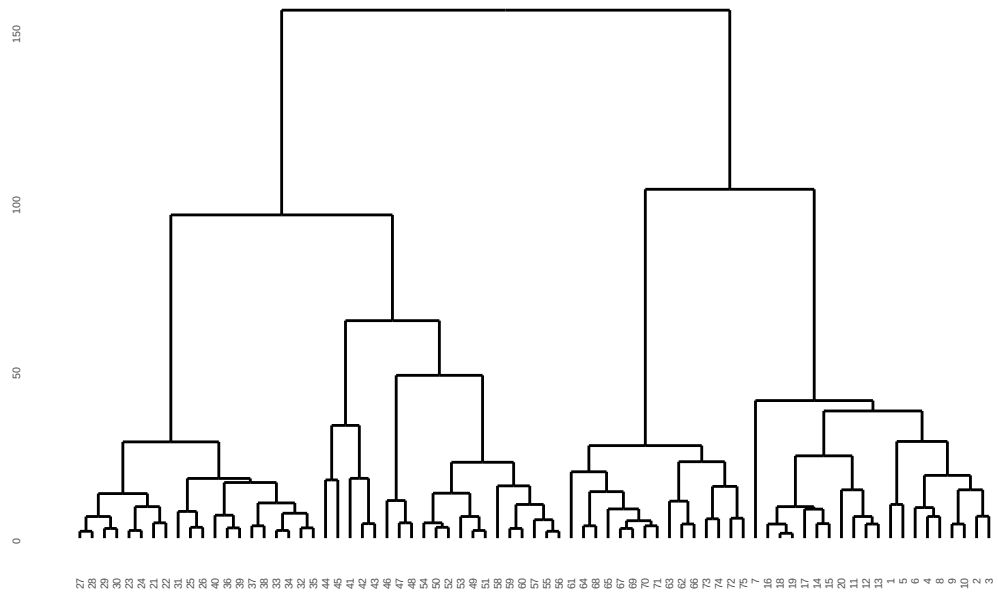
Whilst two prediction types are possible, there are some learners where 'prediction' can never make sense, for example in hierarchical clustering. In hierarchical clustering, the goal is to build a hierarchy of nested clusters by either splitting large clusters into smaller ones or merging smaller clusters into bigger ones. The final result is a tree or dendrogram which can change if a new data point is added. For consistency, `mlr3cluster` offers `predict` method for hierarchical clusters but with a warning:

```
learner = lrn("clust.hclust")
learner$train(task)
learner$predict(task)
```

```
Warning: Learner 'clust.hclust' doesn't predict on new data and predictions may
not make sense on new data
```

```
<PredictionClust> for 75 observations:
    row_ids partition
          1         1
          2         1
          3         1
---
         73         1
         74         1
         75         1
```

```
autoplot(learner) + theme(axis.text = element_text(size = 3.5))
```



In this case, the `predict` method simply cuts the dendrogram into the number of clusters specified by `k` parameter of the learner.

7.4.3 MeasureClust

As previously discussed, unsupervised tasks do not have ground truth data to compare to in model evaluation. However, we can still measure the quality of cluster assignments by quantifying how closely objects within the same cluster are related (cluster cohesion) as well as how distinct different clusters are from each other (cluster separation). There are a few built-in evaluation metrics available to assess the quality of clustering, see Appendix D.

Two common measures are the within sum of squares (WSS) measure, `mlr_measures_clust.wss`, and the silhouette coefficient, `mlr_measures_clust.silhouette`. WSS calculates the sum of squared differences between observations and centroids, which is a quantification of cluster cohesion (smaller values indicate clusters more compact). The silhouette coefficient quantifies how well each point belongs to its assigned cluster versus neighboring clusters, where scores closer to `1` indicate well clustered and scores closer to `-1` indicate poorly clustered. Note that the silhouette measure in `mlr3cluster` returns the mean silhouette score across all observations and when there is only a single cluster, the measure simply outputs 0.

Putting this together with the above we can now score our cluster estimation (note we must pass the `task` to `$score`):

```
meas = msrs(c("clust.wss", "clust.silhouette"))

prediction$score(meas, task = task)
```

```
      clust.wss clust.silhouette
   5.115541e+04     6.413923e-01
```

The very high WSS and middling mean silhouette coefficient indicate that our clusters could do with a bit more work. Often reducing an unsupervised task to a quantitative measure may not be useful (given no ground truth) and instead visualization (discussed below) may be a more effective tool for assessing the quality of the clusters.

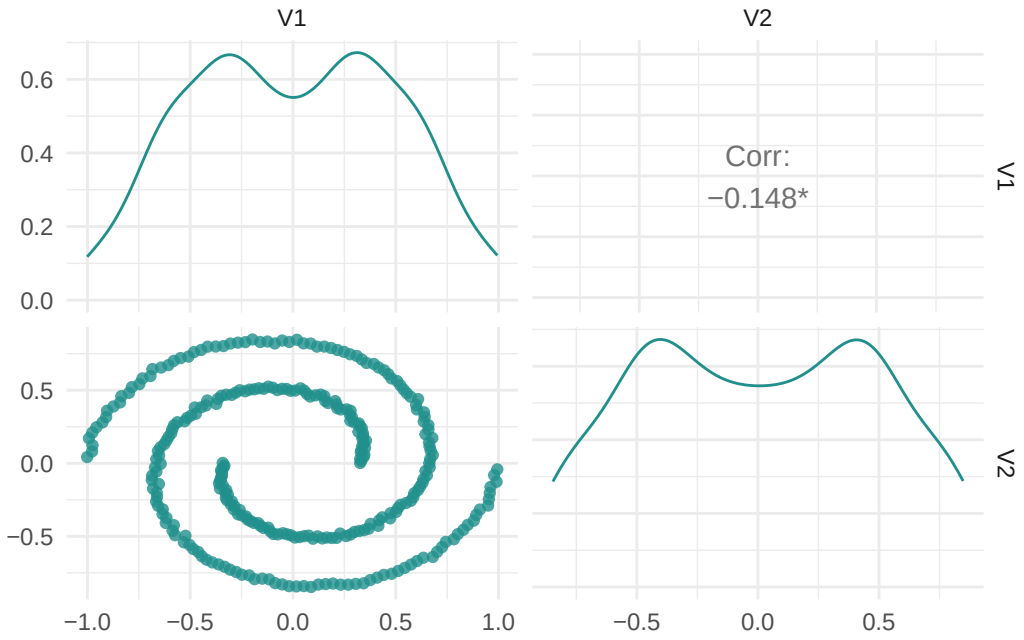
7.4.4 Visualization

As clustering is an unsupervised task, visualization can be essential not just for 'evaluating' models but also for determining if our learners are performing as expected for our task. The next section will look at visualizations for supporting clustering choices and following that we will consider plots for evaluating model performance.

7.4.4.1 Visualizing clusters

It is easy to rely on clustering measures to assess the quality of clustering. However, this should be done with some care, by example consider cluster analysis on the following dataset.

```
spirals = mlbench::mlbench.spirals(n = 300, sd = 0.01)
task = as_task_clust(as.data.frame(spirals$x))
autoplot(task)
```



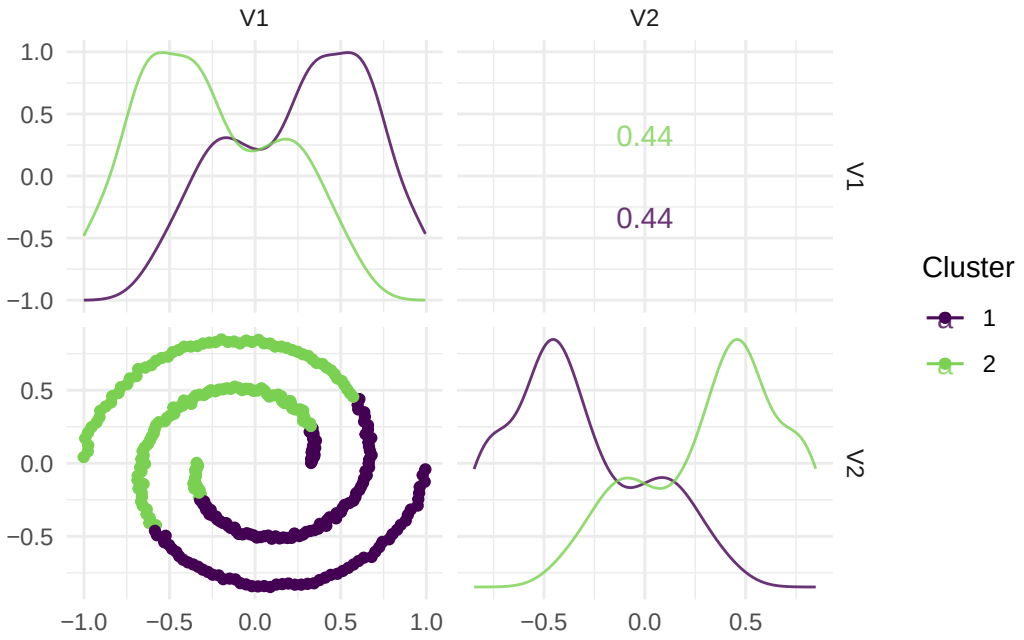
Now fitting our clustering learner.

```
learners = list(
  lrn("clust.kmeans"),
  lrn("clust.dbscan", eps = 0.1)
)
measures = list(msr("clust.silhouette"))
bmr = benchmark(benchmark_grid(task, learners, rsmp("insample")))
bmr$aggregate(measures)[, c(4, 7)]
```

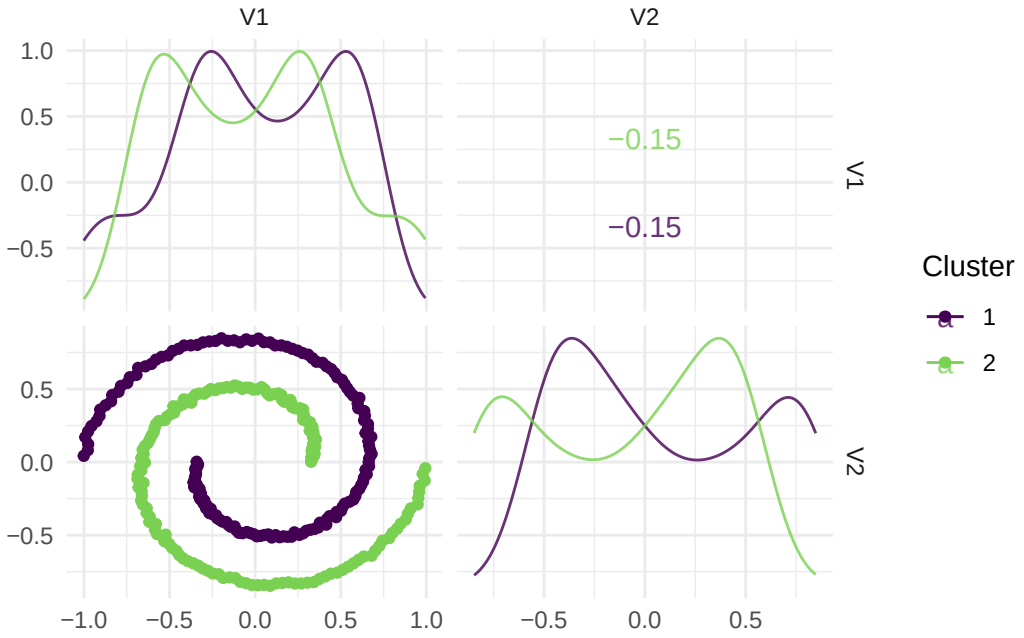
```
   learner_id clust.silhouette
1: clust.kmeans       0.37230996
2: clust.dbscan       0.02941168
```

We can see that K-means clustering gives us a higher average silhouette score and might assume that K-means learner with 2 centroids is a a better choice than DBSCAN method. However, now take a look at the cluster assignment plots.

```
prediction_kmeans = bmr$resample_results$resample_result[[1]]$prediction()
prediction_dbscan = bmr$resample_results$resample_result[[2]]$prediction()
autoplot(prediction_kmeans, task)
```




```
autoplot(prediction_dbscan, task)
```



The two learners arrived at different results – the K-means algorithm assigned points that are part of the same line into two different clusters whereas DBSCAN assigned each line to its own cluster. Which one of these approaches is correct? The answer is it depends on

your specific task and the goal of cluster analysis. If we had only relied on the silhouette score, then the details of how exactly the clustering was done would have been masked and we would not be able to decide which method was appropriate for the task.

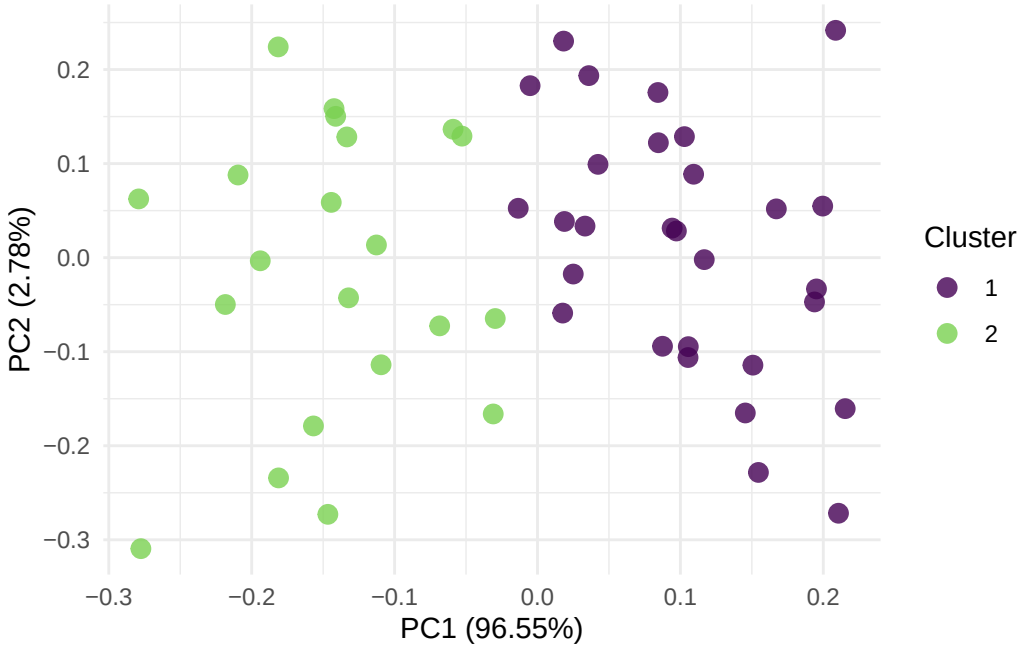
7.4.4.2 PCA and Silhouette plots

The two most important plots implemented in `mlr3viz` to support evaluation of cluster learners are principal components analysis (PCA) and silhouette plots.

PCA is a commonly used dimension reduction method in ML to reduce the number of variables in a dataset or to visualize the most important 'components', which are linear transformations of the dataset features. Components are considered more important if they have higher variance (and therefore more predictive power). In the context of clustering, by plotting observations against the first two components, and then coloring them by cluster, we could visualize our high dimensional dataset and we would expect to see observations in distinct groups.

Since our running example only has two features, PCA does not make sense to visualize the data. So we will use a task based on the `USArrests` dataset instead. By plotting the result of PCA, we see that our model has separated observations into two clusters along the first two principal components.

```
task = mlr_tasks$get("usarrests")
learner = lrn("clust.kmeans")
prediction = learner$train(task)$predict(task)
autoplot(prediction, task, type = "pca")
```

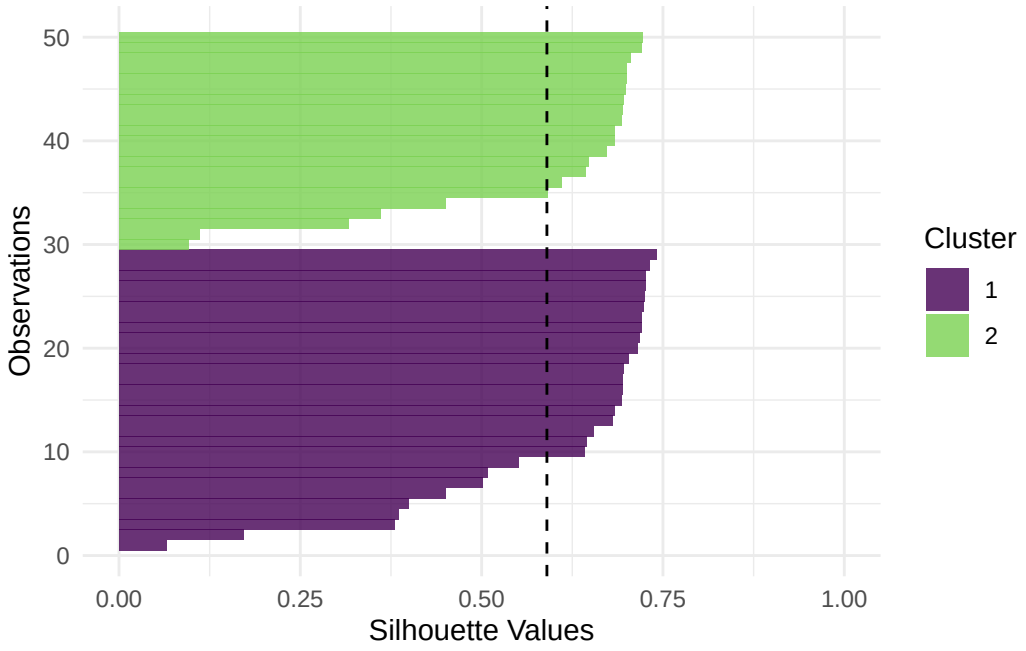


Silhouette plots visually assess the quality of the estimated clusters by visualizing if observations in a cluster are well-placed both individually and as a group. The plots include a

dotted line which visualizes the average silhouette coefficient across all data points and each data point's silhouette value is represented by a bar colored by cluster. In our particular case, the average silhouette index is 0.59. If the average silhouette value for a given cluster is below the average silhouette coefficient line then this implies that the cluster is not well defined.

Continuing with our new example, we find that a lot of observations are actually below the average line and close to zero, and therefore the quality of our cluster assignments is not very good, meaning that many observations are likely assigned to the wrong cluster.

```
autoplot(prediction, task, type = "sil")
```



7.4.5 Putting it all together

Finally, we conduct a small benchmark study using the `ruspini` data and with a few integrated cluster learners:

```
task = tsk("ruspini")
learners = list(
  lrn("clust.featureless"),
  lrn("clust.kmeans", centers = 4L),
  lrn("clust.cmeans", centers = 3L)
)
measures = list(msr("clust.wss"), msr("clust.silhouette"))
bmr = benchmark(benchmark_grid(task, learners, rsmp("insample")))
bmr$aggregate(measures)[, c(4, 7, 8)]
```

```
      learner_id clust.wss clust.silhouette
```

```
1: clust.featureless 244373.87        0.0000000
2:       clust.kmeans  48309.14        0.5537388
3:       clust.cmeans  51063.48        0.6327047
```

The experiment shows that using the K-means algorithm with four centers has the best cluster cohesion (lowest within sum of squares) and the best average silhouette score.

7.5 Spatial Analysis

Spatial Analysis

The final task we will discuss in this book is spatial analysis. Spatial analysis can be a subset of any other machine learning task (e.g., regression or classification) and is defined by the presence of spatial information in a dataset, usually stored as coordinates that are often named "x" and "y" or "lat" and "lon" (for 'latitude' and 'longitude' respectively.)

Autocorrelation

Spatial analysis is its own task as spatial data must be handled carefully due to the complexity of 'autocorrelation'. Where correlation is defined as a statistical association *between two* variables, autocorrelation is a statistical association *within one* variable. In machine learning terms, in a dataset with features and observations, correlation occurs when two or more features are statistically associated in some way, whereas autocorrelation occurs when two or more observations are statistically associated across one feature. Autocorrelation therefore violates one of the fundamental assumptions of ML that all observations in a dataset are independent, which results in lower confidence about the quality of a trained ML model and the resulting performance estimates (Hastie, Friedman, and Tibshirani 2001).

Autocorrelation is present in spatial data as there is implicit information encoded in coordinates, such as whether two observations (e.g., cities, countries, continents) are close together or far apart. For example, say we are predicting the number of cases of a disease two months after outbreak in Germany. Outbreaks radiate outwards from an epicenter and therefore countries closer to Germany will have higher numbers of cases, and countries further away will have lower numbers (Figure 7.4, bottom). Thus, looking at the data spatially shows clear signs of autocorrelation across nearby observations. Note in this example the autocorrelation is radial but in practice this will not always be the case.

Unlike other tasks we have looked at in this chapter, there is no underlying difference to the implemented learners or measures. Instead we provide additional resampling methods in `mlr3spatiotempcv` to account for the similarity in the train and test sets during resampling that originates from spatiotemporal autocorrelation.

Throughout this section we will use the `mlr3spatiotempcv::ecuador` dataset and task as a working example.

7.5.1 TaskClassifST and TaskRegrST

To make use of spatial resampling methods, we have implemented two extensions of `TaskClassif` and `TaskRegr` to accommodate spatial data, `TaskClassifST` and `TaskRegrST` respectively. Below we only show classification examples but regression follows trivially.

```
1 library(mlr3spatial)
2 library(mlr3spatiotempcv)
```

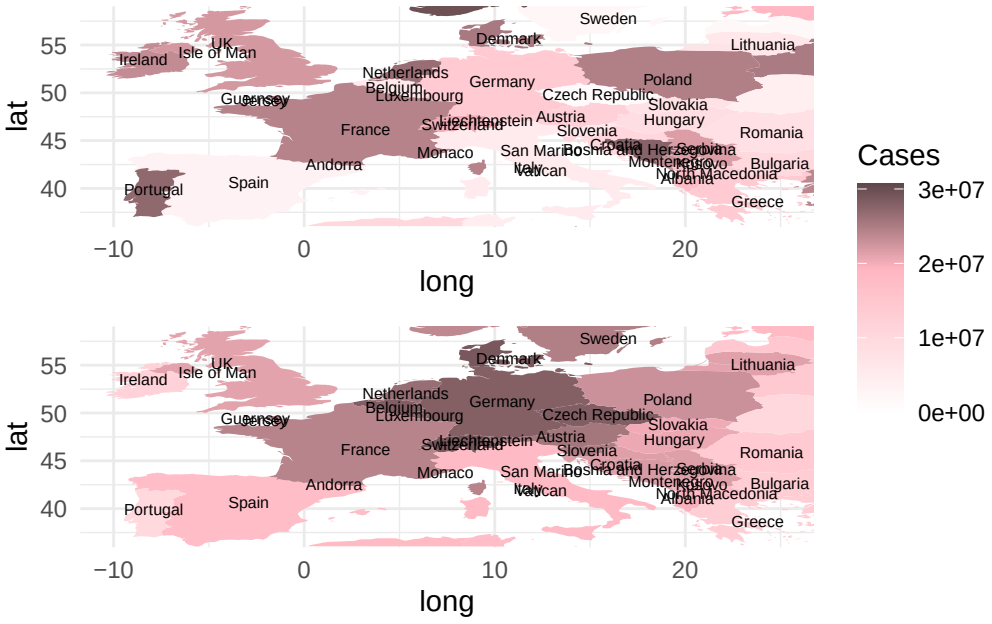


Figure 7.4: Heatmaps where darker countries indicate higher number of cases and lighter countries indicate lower number of cases of imaginary Disease X with epicenter in Germany. The top map imagines a world in which there is no spatial autocorrelation and the number of cases of a disease is randomly distributed. The bottom map shows a more accurate world in which the number of cases radiate outwards from the epicenter (Germany).

```
# create task from `data.frame`
task = as_task_classif_st(ecuador, id = "ecuador_task",
  target = "slides", positive = "TRUE",
  coordinate_names = c("x", "y"), crs = "32717")

# or create task from 'sf' object
data_sf = sf::st_as_sf(ecuador, coords = c("x", "y"), crs = "32717")
task = as_task_classif_st(data_sf, target = "slides", positive = "TRUE")
task
```

```
<TaskClassifST:data_sf> (751 x 11)
* Target: slides
* Properties: twoclass
* Features (10):
  - dbl (10): carea, cslope, dem, distdeforest, distroad,
    distslidespast, hcurv, log.carea, slope, vcurv
* Coordinates:
            X       Y
  1: 712882.5 9560002
  2: 715232.5 9559582
  3: 715392.5 9560172
  4: 715042.5 9559312
  5: 715382.5 9560142
 ---
747: 714472.5 9558482
748: 713142.5 9560992
749: 713322.5 9560562
750: 715392.5 9557932
751: 713802.5 9560862
```

Once a task is created, you can train and predict as normal.

```
lrn("classif.rpart")$train(task)$predict(task)
```

```
<PredictionClassif> for 751 observations:
    row_ids truth response
          1  TRUE     TRUE
          2  TRUE     TRUE
          3  TRUE     TRUE
---
        749 FALSE    FALSE
        750 FALSE    FALSE
        751 FALSE     TRUE
```

However as discussed above, it is best to use the specialized resampling methods to achieve bias-reduced estimates of model performance.

7.5.2 Spatiotemporal Cross-Validation

Before we look at the spatial resampling methods implemented in `mlr3spatiotempcv` we will first show what can go wrong if non-spatial resampling methods are used for spatial data. Below we benchmark a decision tree on the `mlr_tasks_ecuador` task using two different repeated cross-validation resampling methods, the first ("NSpCV" (non-spatial cross-validation)) is a non-spatial resampling method from `mlr3`, the second ("SpCV" (spatial cross-validation)) is from `mlr3spatiotempcv` and is optimized for spatial data. The example highlights how "NSpCV" makes it appear as if the decision tree is performing better than it is with significantly higher estimated performance, however this is an overconfident prediction due to the autocorrelation in the data.

```
learner = lrn("classif.rpart", predict_type = "prob")
rsmp_nsp = rsmp("repeated_cv", folds = 3, repeats = 2, id = "NSpCV")
rsmp_sp = rsmp("repeated_spcv_coords", folds = 3, repeats = 2, id = "SpCV")

design = benchmark_grid(task, learner, c(rsmp_nsp, rsmp_sp))
bmr = benchmark(design)
bmr$aggregate(msr("classif.acc"))[, c(5, 7)]
```

```
   resampling_id classif.acc
1:         NSpCV   0.6744197
2:          SpCV   0.5842370
```

In the above example, applying non-spatial resampling results in train and test sets that are very similar due to the underlying spatial autocorrelation. Hence there is little difference from testing a model on the same data it was trained on, which should be avoided for an honest performance result (see Chapter 2). In contrast, the spatial method has accommodated for autocorrelation and the test data is therefore less correlated (though still some association will remain) with the training data. Visually this can be seen using built-in `autoplot` methods. In Figure 7.5 we visualize how the task is partitioned according to the spatial resampling method (Figure 7.5, top) and non-spatial resampling method (Figure 7.5, bottom). There is a clear separation in space for the respective partitions when using the spatial resampling whereas the train and test splits overlap a lot (and are therefore more correlated) using the non-spatial method.

```
library(patchwork)
autoplot(rsmp_sp, task, fold_id = c(1:3), size = 0.7) /
  autoplot(rsmp_nsp, task, fold_id = c(1:3), size = 0.7) &
  scale_y_continuous(breaks = seq(-3.97, -4, -0.01)) &
  scale_x_continuous(breaks = seq(-79.06, -79.08, -0.02))
```

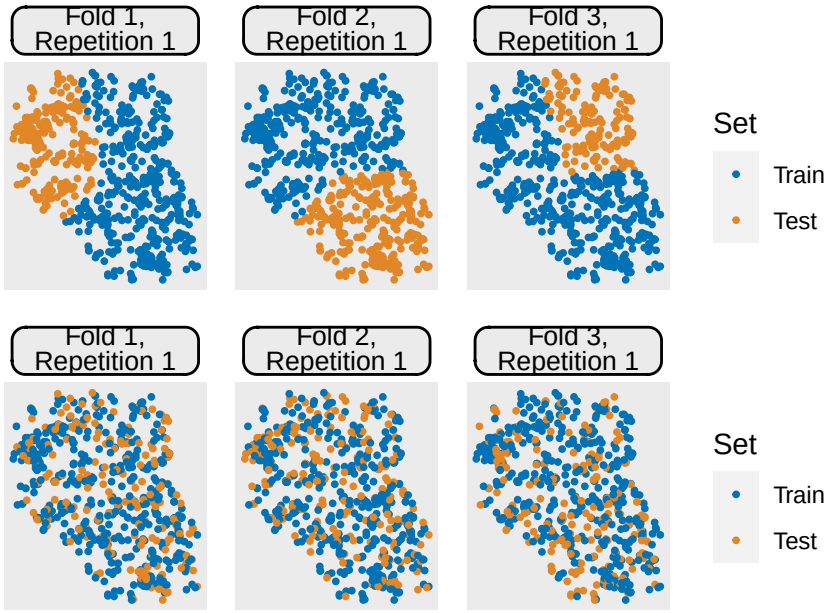


Figure 7.5: Scatterplots show separation of train (blue) and test (orange) data for the first three (left to right) folds of the first repetition of the cross-validation. The top row is spatial resampling where train and test data are clearly separated. The bottom row is non-spatial resampling where there is overlap in train and test data.

Now we have seen why spatial resampling matters we can take a look at what methods are available in `mlr3spatiotempcv`. The resampling methods we have added can be categorized into:

- Blocking – Create rectangular blocks in 2D or 3D space
- Buffering – Create buffering zones to remove observations between train and test sets
- Spatiotemporal clustering – Clusters based on coordinates (and/or time-points)
- Feature space clustering – Clusters based on feature space and not necessarily spatiotemporal
- Custom (partitioning) – Grouped by factor variables

The choice of method may depend on specific characteristics of the dataset and there is no easy rule to pick one method over another, full details of different methods can be found in Schratz et al. (2021) – the paper deliberately avoids recommending one method over another as the 'optimal' choice is highly dependent on the predictive task, autocorrelation in the data, and the spatial structure of the sampling design. The documentation for each of the implemented methods includes details of each method as well as references to original publications.

> Spatio*temporal* resampling
>
> We have focused on spatial analysis but referred to "spatiotemporal" and "spatiotemp". The spatial-only resampling methods discussed in this section can be extended to

temporal analysis (or spatial and temporal analysis combined) by setting the `"time"` `col_role` in the task (Chapter 3) – this is an advanced topic that may be added in future editions of this book. See the `mlr3spatiotempcv` visualization vignette[1] for specific details about 3D spatiotemporal visualization.

7.5.3 Spatial Prediction

Until now we have looked at resampling to accommodate spatiotemporal *features*, but what if you want to make spatiotemporal *predictions*? In this case the goal is to make classification or regression predictions at the a pixel level, i.e., for an area, defined by the geometric resolution, of a raster image.

To enable these predictions we have created a new function, `predict_spatial`, to allow spatial predictions using any of the following spatial data classes:

- `stars` (from package `stars`)
- `SpatRaster` (from package `terra`)
- `RasterLayer` (from package `raster`)
- `RasterStack` (from package `raster`)

To use a raster image for prediction, it must be wrapped in `TaskUnsupervised`. In the example below we load the `leipzig_points` dataset for training and coerce this to a spatiotemporal task with `as_task_classif_st`, and we load the `leipzig_raster` raster image and coerce this to an unsupervised task. Both files are included as example data in `mlr3spatial`.

```
library(sf)

# load sample points
leipzig_vector = sf::read_sf(system.file("extdata",
  "leipzig_points.gpkg", package = "mlr3spatial"),
  stringsAsFactors = TRUE)
# create training data
tsk_leipzig = as_task_classif_st(leipzig_vector, target = "land_cover")

# load raster image
leipzig_raster = terra::rast(system.file("extdata", "leipzig_raster.tif",
  package = "mlr3spatial"))
# create testing data
tsk_predict = as_task_unsupervised(leipzig_raster)
```

Now we can continue as normal to train and predict with a classification learner, in this case a random forest.

```
lrn = lrn("classif.ranger")$train(tsk_leipzig)
pred = predict_spatial(tsk_predict, lrn, format = "terra")
pred
```

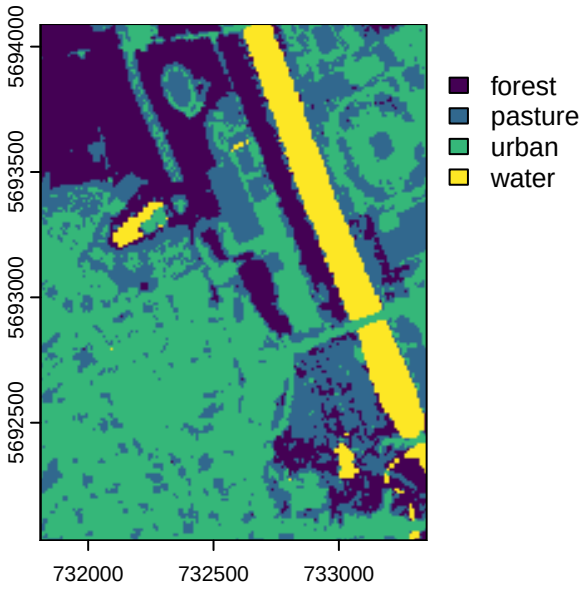
```
class       : SpatRaster
```

[1] https://mlr3spatiotempcv.mlr-org.com/articles/spatiotemp-viz.html

```
dimensions  : 206, 154, 1  (nrow, ncol, nlyr)
resolution  : 10, 10  (x, y)
extent      : 731810, 733350, 5692030, 5694090  (xmin, xmax, ymin, ymax)
coord. ref. : WGS 84 / UTM zone 32N (EPSG:32632)
source      : file25a158db9658.tif
categories  : categories
name        : land_cover
min value   :     forest
max value   :      water
```

In this example we specified creation of a `terra` object, which can be visualized with in-built plotting methods.

```
library(terra, exclude = "resample")
plot(pred, col = c("#440154FF", "#443A83FF", "#31688EFF",
  "#21908CFF", "#35B779FF", "#8FD744FF", "#FDE725FF"))
```



7.6 Conclusion

In this chapter, we explored going beyond deterministic regression and classification to see how functions in the `mlr3` can be used to implement other machine learning tasks. Cost-sensitive classification extends the 'normal' classification setting by assuming errors associated with false negatives and false positives are unequal. Running cost-sensitive classification experiments is possible just using features in `mlr3`. Survival analysis, available in `mlr3proba`, can be thought of as a regression problem when the outcome may be censored, which means

it may never be observed within a given time frame. The final task in `mlr3proba` is density estimation, the unsupervised task concerned with estimating univariate probability distributions. Using `mlr3cluster`, you can perform cluster analysis on observations, which involves grouping observations together according to similarities in their variables. Finally, with `mlr3spatial` and `mlr3spatiotempcv`, it is possible to perform spatial analysis to make predictions using co-ordinates as features and to make spatial predictions. The `mlr3` interface is highly extensible, which means future machine learning tasks can (and will) be added to our universe and we will add these to this chapter of the book in future editions.

7.7 Exercises

We will run `set.seed(11)` before each of our solutions so you can reproduce our results.

1. Run a benchmark experiment on the `german_credit` task with algorithms: `featureless`, `log_reg`, `ranger`. Tune the `featureless` model using `tunetreshold` and `learner_cv`. Use 2-fold CV and evaluate with `msr("classif.costs", costs = costs)` where you should make the parameter `costs` so that the cost of a true positive is -10, the cost of a true negative is -1, the cost of a false positive is 2, and the cost of a false negative is 3. Are your results surprising?
2. Train and predict a survival forest using `rfsrc` (from `mlr3extralearners`). Run this experiment using `task = tsk("rats"); split = partition(task)`. Evaluate your model with the RCLL measure.
3. Estimate the density of the `tsk("precip")` data using `logspline` (from `mlr3extralearners`). Run this experiment using `task = tsk("precip"); split = partition(task)`. Evaluate your model with the logloss measure.
4. Run a benchmark clustering experiment on the `wine` dataset without a label column. Compare the performance of `k-means` learner with `k` equal to 2, 3 and 4 using the silhouette measure. Use insample resampling technique. What value of `k` would you choose based on the silhouette scores?
5. Run a (spatially) unbiased classification benchmark experiment on the `ecuador` task with a featureless learner and xgboost, evaluate with the binary Brier score.

8

Technical

Michel Lang
Research Center Trustworthy Data Science and Security, and TU Dortmund University

In the previous chapters, we demonstrated how to turn ML concepts and ML methods into code. Unfortunately, we let out the more technical details yet, but they are utterly important for conducting applied ML experiments. This includes the following topics:

- Parallelization with the `future` framework (Section 8.1),
- how to handle errors and troubleshoot (Section 8.2),
- adjust the logger to your needs (Section 8.3),
- working with out-of-memory data, e.g., data stored in databases (Section 8.4), and
- adding new classes to mlr3 (Section 8.5).

8.1 Parallelization

The term "parallelization" refers to running multiple algorithms in parallel, i.e., executing them simultaneously on multiple CPU cores, CPUs, or computational nodes. Not all algorithms can be parallelized, but when they can, parallelization allows significant savings in computation time.

In general, there are many possibilities to parallelize, depending on the hardware to run the computations: If you only have a single CPU with multiple cores, threads[1] or forks[2] are ways to utilize all cores on a local machine. If you have multiple machines on the other hand, the machines need a way to communicate and exchange information, e.g. via protocols like network sockets or the Message Passing Interface (MPI)[3]. Larger computational sites rely on a scheduler to orchestrate the computation for multiple users and offer a shared network file system all machines can access. Interacting with scheduling systems on compute clusters is covered in **?@sec-large-benchmarking** using the R package `batchtools`. We do not want to delve too deep into such details here, but want to introduce some terminology which helps us to discuss parallelization on a more abstract level:

- We call the hardware to parallelize on together with the respective interface provided by an R package the parallelization backend. As many parallelization backends have different APIs, we are using the `future` package as an abstraction layer for many parallelization backends. `mlr3` just interfaces `future` while the user can control how the code is executed by configuring a backend prior to starting the computations. — Parallelization Backend
- The R session or process which orchestrates the computational work is called main, and it starts computational jobs. — Main Jobs

[1] https://en.wikipedia.org/wiki/Thread_(computing)
[2] https://en.wikipedia.org/wiki/Fork_(system_call)
[3] https://en.wikipedia.org/wiki/Message_Passing_Interface

- The R sessions, processes, forks or machines which receive the jobs, do the calculation and then send back the result are called workers.

Workers

An important step in parallel programming involves the identification of sections of the program flow which are both time-consuming (bottlenecks) and also can run independently of a different section. The key characteristic is that these sections do not depend on each other, i.e. section A can be ran without waiting for results from section B. Fortunately, these sections are comparably easy to spot for machine learning experiments:

Bottlenecks

1. The training of a learning algorithm (or other computationally intensive parts of a machine learning pipeline, c.f. Chapter 6) *may* contain independent sections which can run in parallel, e.g.
   - A single decision tree iterates over all features to find the best split point, for each feature independently.
   - A random forest usually fits hundreds of trees independently.
   - Many feature filters work in a univariate fashion, i.e. calculate a numeric score for each feature independently.

   Parallelization of learning algorithms is covered in Section 8.1.1.
2. A resampling consists of independent repetitions of train-test-splits (Section 8.1.2).
3. A benchmark consists of multiple independent resamplings (Section 8.1.3).
4. Tuning (Chapter 4) is repeated benchmarking, embedded in a sequential procedure which determines the hyperparameter configuration to try next. In addition to parallelization of the benchmark, some tuners propose multiple configurations to evaluate independently in each sequential step, which provides a second level for parallelization discussed in Section 8.1.4.
5. The predictions of a single learner for multiple observations is independent (Section 8.1.5).

When computational problems are so easy to parallelize like the examples listed in (1)-(4), they are often referred to as embarrassingly parallel. Whenever you can put the heavy lifting into a function and call it with a map-like function like `lapply()`, you are facing an embarrassingly parallel problem. Such problems are straightforward to parallelize, e.g., in R with the `furrr` package which provides parallel counterparts for popular sequential map-like functions from the `purrr` package.

Embarrassingly Parallel

However, it does not make practical sense to actually execute in parallel every operation that can be parallelized. Starting and terminating workers as well as possible communication between workers comes at a price in the form of additionally required runtime which is called parallelization overhead. The overhead strongly varies from parallelization backend to parallelization backend and must be carefully weighed against the runtime of the sequential execution to determine if parallelization is worth the effort. If the sequential execution is comparably fast, enabling parallelization often just introduces additional complexity for very little runtime savings or can even slow down the execution.

Parallelization Overhead

Granularity

Sometimes, it is possible to control the granularity of the parallelization to reduce the parallelization overhead. For example, if you want to parallelize a `for`-loop with 1000 iterations on 4 CPU cores, the overhead can be reduced by chunking the work of the 1000 jobs into 4 computational jobs performing 250 iterations each. So 4 bigger jobs are calculated instead of 1000 small ones.

This effect is illustrated in the following code chunk using a socket cluster. Note that this parallel backend already comes with an option to control the chunk size (`chunk.size`), but

for other backends you must chunk manually which is also demonstrated:

```
# set up a socket cluster with 4 workers on the local machine
library(parallel)
cores = 4
cl = makeCluster(cores)
print(cl)
```

```
socket cluster with 4 nodes on host 'localhost'
```

```
# vector to operate on
x = 1:1000

# fast function to parallelize
f = function(x) sqrt(x + 1)


# unchunked approach: 1000 jobs
system.time({
  parSapply(cl, x, f, chunk.size = 1)
})
```

```
   user  system elapsed
  0.240   0.020  11.013
```

```
# chunked approach: 4 jobs
system.time({
  parSapply(cl, x, f, chunk.size = 250)
})
```

```
   user  system elapsed
  0.002   0.000   0.093
```

```
# manual chunking: 4 jobs
chunks = rep(1:cores, each = length(x) %/% cores)
jobs = split(x, chunks)
system.time({
  parLapply(cl, jobs, function(chunk, .fun) sapply(chunk, .fun) ,
    .fun = f, chunk.size = 1)
})
```

```
   user  system elapsed
  0.002   0.000   0.049
```

Whenever you have the option to control the granularity by setting the chunk size, you should aim for at least as many jobs as workers and also the runtime of each worker should be at least several seconds. This ensures that you can fully utilize the system and that the parallelization overhead stays reasonable. If you have heterogeneous runtimes, also consider grouping jobs together so that the runtime of the chunks get homogeneous. If there is a good estimate for the runtime, `batchtools::binpack()` (create an arbitrary number of chunks with a specified maximum combined runtime) and `batchtools::lpt()` (pack a specified

number of chunks with arbitrary but homogeneous runtime) can prove useful. For unknown runtimes, randomizing the order of jobs sometimes helps if there is a systematic relationship between the order of the jobs and their runtime. This prevents the long jobs, for example, from all being executed at the end, which leads to underutilization. `mlr3misc` ships with the functions `chunk()` and `chunk_vector()` to conveniently chunk vectors and also shuffles them per default.

8.1.1 Parallelization of Learners

The most atomic part of `mlr3` which can be parallelized are calls to external code, i.e. the execution of certain `PipeOp` objects, `Filter` objects or `Learner` objects. For these objects, `mlr3` merely provides a unified interface to control the execution. The parallelization is implemented by the respective package authors of the (external) algorithms `mlr3` calls.

Most of these algorithms are parallelized via threading, e.g., the random forest implementation in `ranger` or the boosting implemented in `xgboost`. For example, while fitting a single decision tree, each split that divides the data into two disjoint partitions requires a search for the best cut point on all p features. So instead of iterating over all features sequentially, the search can be broken down into p threads, each searching for the best cut point on a single feature. These threads can easily be parallelized by the scheduler of the operating system, as there is no need for communication between the threads. After all threads have finished, the results are collected and merged before terminating the threads. I.e., for our example of the decision tree, (1) the p best cut points per feature are collected and then (2) aggregated to the single best cut point across all features by just iterating over the p results sequentially.

> **Note**
>
> Parallelization on GPUs is not covered in this book. `mlr3` only distributes the fitting of multiple learners, e.g., during resampling, benchmarking, or tuning. On this rather abstract level, GPU parallelization does not work efficiently. However, some learning procedures can be compiled against CUDA/OpenCL to utilize the GPU while fitting a single model. We refer to the respective documentation of the learner's implementation, e.g., https://xgboost.readthedocs.io/en/stable/gpu/ for XGBoost.

Threading is implemented in the compiled code of the package (e.g., in C or C++). The R interpreter calls the external code and waits for the results to be returned - without noticing that the computations are executed in parallel. Unfortunately, threading conflicts with certain parallel backends, causing the system to be overutilized in the best case and causing hangs or segfaults in the worst case. For this reason, we introduced the convention that threading parallelization is turned off per default. Hyperparameters that control the number of threads are tagged with the label `"threads"`:

```
library("mlr3learners") # for the ranger learner

# get the ranger learner
learner = lrn("classif.ranger")

# show all hyperparameters tagged with "threads"
learner$param_set$ids(tags = "threads")
```

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

```
# The number of threads is initialized to 1
learner$param_set$values$num.threads
```

```
[1] 1
```

To enable the parallelization for this learner, `mlr3` provides the helper function `set_threads()`:

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

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

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

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

In the last line, we did not set the number of threads, letting the package fall back to a heuristic to detect the correct number. This heuristic is sometimes flaky, and utilizing all available cores is occasionally counterproductive as overburdening the system often has negative effects on the overall runtime. The function which determines the number of CPUs for `mlr3` is implemented in `parallelly::availableCores()` and works well for many setups. See this blog post[4] for some background information about the implemented heuristic. However, there are still some scenarios where it is better to reduce the number of utilized CPUs manually:

- You want to simultaneously work on the same system, e.g., browse the web or watch a video.
- You are on a multi-user system and want to spare some resources for other users.
- You have a CPU with heterogeneous cores, for example, the energy-efficient "Icestorm" cores on a Mac M1 chip. These are comparably slower than the high-performance "Firestorm" cores and not well suited for heavy computations.

[4] https://www.jottr.org/2022/12/05/avoid-detectcores

- You have linked R to a threaded BLAS implementation like OpenBLAS, and your learners make heavy use of linear algebra.

You can manually set the number of CPUs to overrule the heuristic via option `"mc.cores"`:

```
options(mc.cores = 4)
```

We recommend setting this in your system's `.Rprofile` file, c.f. `Startup`.

There are some other approaches for parallelization of learners, e.g. by directly supporting one specific parallelization backend or a parallelization framework like `foreach`. If this is supported, parallelization must be explicitly activated, e.g. by setting a hyperparameter. If you need to parallelize on the learner level because a single model fit takes too much time, and you only fit a few of these models, consult the documentation of the respective learner. In many scenarios it makes more sense to parallelize on a different level like resampling or benchmarking which is covered in the following subsections.

8.1.2 Parallelization of Resamplings

In addition to parallel learners, most machine learning experiments include a very easy handle for parallelization: the resampling. By definition, resampling is performed to get an unbiased performance estimator by aggregating over **independent** repetitions of multiple train-test splits.

`mlr3` has "marked" this loop of independent iterations as parallelizable and uses an additional abstraction layer to support a broad range of parallel backends: the `future` package. The loop is executed via the `future` parallelization framework, using the parallel backend configured by the user via the `future::plan()` function.

In this section, we will use the spam task and a simple `lrn("classif.rpart")`. We use the `future::multisession` plan (which internally uses socket clusters from the `parallel` package, see `parallel::makeCluster()`) that should work on all operating systems.

```
# query the currently active plan
future::plan()

# define objects to perform a resampling
task = tsk("spam")
learner = lrn("classif.rpart")
resampling = rsmp("cv", folds = 3)

# select the multisession backend to use
future::plan("multisession")

# run the resampling in parallel
time = proc.time()[3]
resample(task, learner, resampling)
diff = proc.time()[3] - time
```

By default, all CPUs of your machine are used unless you specify the argument `workers` in `future::plan()` (possible problems with the value returned by the heuristic have already been discussed in the previous Section 8.1.1). If you compare runtimes between the

parallel backend and sequential execution (`plan("sequential")`) here, you should see a decrease in the reported elapsed time. However, in practice, you cannot expect the runtime to fall linearly as the number of cores increases (Amdahl's law[5]). In contrast to threads, the technical overhead for starting workers, communicating objects, sending back results, and shutting down the workers is quite large for the multisession backend. The multicore backend (`plan("multicore")`) comes with more overhead than threading, but considerably less overhead in comparison with the multisession backend. In fact, with the multicore backend, R objects are copied only when they are modified (copy-on-write), while with the multisession backend, objects are always copied to the respective session prior to any computation. In general, it is advised to only consider parallelization for resamplings where each iteration runs at least a few seconds.

Figure 8.1 illustrates the parallelization from the above example. From left to right:

1. The main process calls the `resample()` function.
2. The computational task is split into 3 parts for the 3-fold cross-validation.
3. The folds are passed to 3 workers, each fitting a model on the respective subset of the task and predicting on the left-out observations.
4. The predictions (and trained models) are communicated back to main process which combines them into a `ResampleResult`.

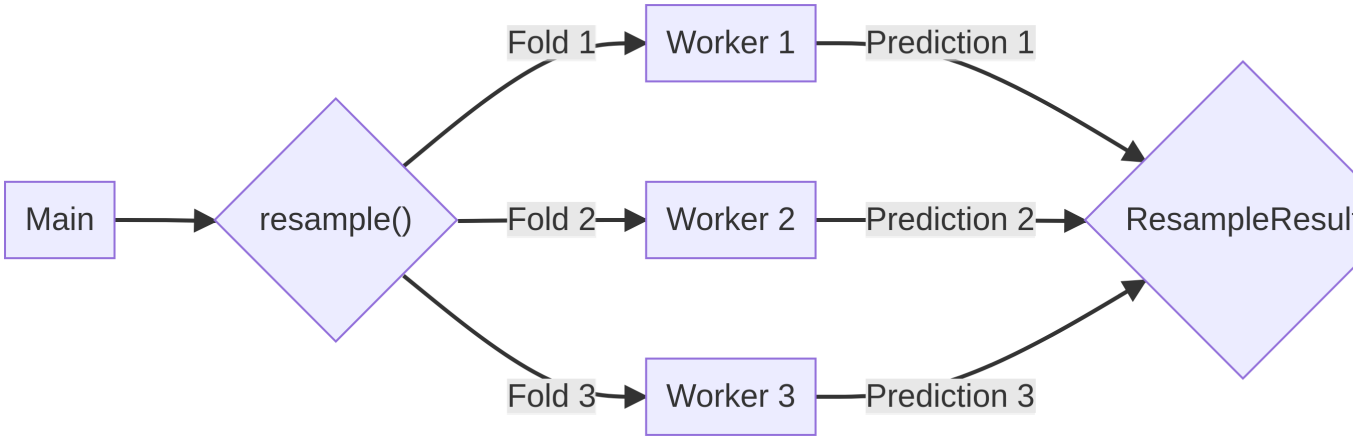


Figure 8.1: Parallelization of a resampling using a 3-fold cross-validation

8.1.3 Parallelization of Benchmarks

Benchmarks can be seen as a collection of multiple independent resamplings where a combination of a task, a learner, and a resampling strategy defines one resampling to perform. In pseudo-code, the calculation can be written down as

```
foreach combination of (task, learner, resampling strategy) {
    foreach resampling iteration {
        execute(resampling, j)
```

[5] https://en.wikipedia.org/wiki/Amdahl%2527s_law

```
    }
}
```

For parallelization, there are now two options:

1. Parallelize over all resamplings, execute each resampling sequentially (parallelize outer loop).
2. Iterate over all resamplings, execute each resampling in parallel (parallelize inner loop).

If you are transitioning from `mlr`, you might be used to selecting one of these parallelization levels before benchmarking. One major drawback of this approach becomes clear when both the outer and inner loop have fewer iterations than there are available workers, resulting in an underutilized system. In `mlr3`, the choice of level is no longer required (except occasionally for nested resampling, briefly described in the following Section 8.1.4). All experiments are rolled out on the same level, i.e., `benchmark()` iterates over the elements of the Cartesian product of the iterations of the outer and inner loops. Therefore, there is no need to decide whether you want to parallelize the tuning *or* the resampling, you always parallelize both. This approach makes the computation fine-grained and gives the `future` backend the opportunity to group the jobs into chunks of suitable size (depending on the number of workers).

8.1.4 Nested Resampling Parallelization

Like in benchmarking, nested resampling for tuning also translates into two nested resampling loops. But unlike benchmarking, the outer loop iterations are not necessarily independent of each other: depending on the result of the resampling in the first outer loop, different hyperparameters are suggested for the second iteration. Therefore, nested loops cannot be flattened, and the user instead has to choose which of the loops to parallelize. Let's consider the following example: You want to tune the `minsplit` argument of a classification tree using the `AutoTuner` of `mlr3tuning` (simplified version taken from Section 4.1):

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

```
Loading required package: paradox
```

```
learner = lrn("classif.rpart",
  minsplit  = to_tune(2, 128, logscale = TRUE)
)

at = auto_tuner(
  tuner = tnr("random_search"),
  learner = learner,
  resampling = rsmp("cv", folds = 2), # inner CV
  measure = msr("classif.ce"),
  term_evals = 20,
)
```

To evaluate the performance on an independent test set, resampling is used:

```
resample(
  task = tsk("penguins"),
  learner = at,
  resampling = rsmp("cv", folds = 5) # outer CV
)
```

```
<ResampleResult> with 5 resampling iterations
  task_id          learner_id resampling_id iteration warnings errors
 penguins classif.rpart.tuned            cv         1        0      0
 penguins classif.rpart.tuned            cv         2        0      0
 penguins classif.rpart.tuned            cv         3        0      0
 penguins classif.rpart.tuned            cv         4        0      0
 penguins classif.rpart.tuned            cv         5        0      0
```

Here, we have three opportunities to parallelize:

1. the inner cross-validation of the auto tuner with 2 folds,
2. the outer cross-validation of the resampling with 5 folds, and
3. the 20 points proposed by the random search in a single batch.

Because the third opportunity is not always applicable, especially for many advanced tuning algorithms which propose only a single points in each iteration, we will focus on the first two opportunities. Furthermore, we assume that we have a single CPU with four cores available.

If we opt to parallelize the outer CV, all four cores would be utilized first with the computation of the first 4 resampling iterations. The computation of the fifth iteration has to wait. The resulting CPU utilization of the nested resampling example on 4 CPUs is visualized in two Figures:

- Figure 8.2 as an example for parallelizing the outer 5-fold cross-validation.

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

  We assume that each fit during the inner resampling takes 4 seconds to compute and that there is no other significant overhead. First, each of the four workers starts with the computation of an inner 2-fold cross-validation. As there are more jobs than workers, the remaining fifth iteration of the outer resampling is queued on CPU1 **after** the first 4 iterations are finished after 8 secs. During the computation of the 5th outer resampling iteration, only CPU1 is utilized, the other 3 CPUs are idling.

- Figure 8.3 as an example for parallelizing the inner 2-fold cross-validation.

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

  Here, the outer loop runs sequentially and distributes the 2 computations for the inner resampling on 2 CPUs. Meanwhile, CPU3 and CPU4 are idling.

Both possibilities for parallelization are not exploiting the full potential of the 4 CPUs. With parallelization of the outer loop, all results are computed after 16s, in contrast to parallelization of the inner loop where the results are only available after 20s.

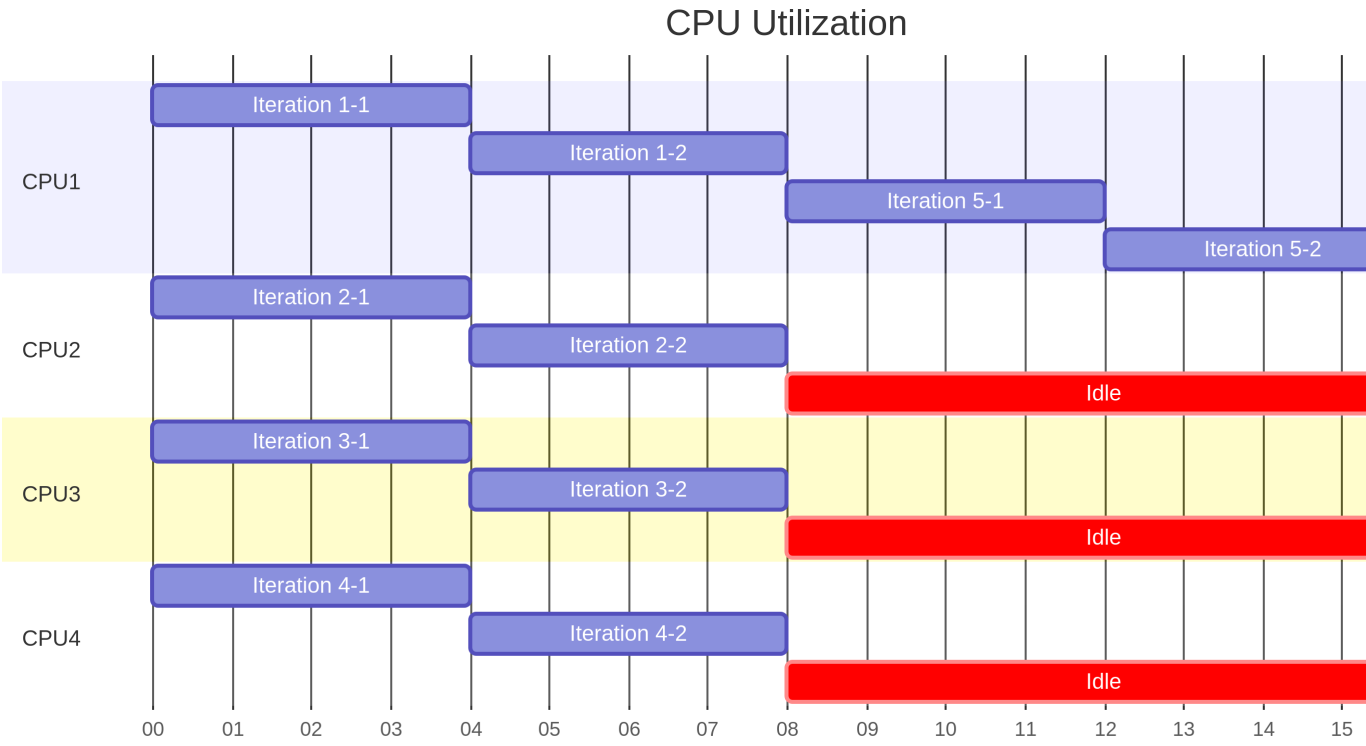


Figure 8.2: CPU utilization while parallelizing the outer resampling of a 2-fold cross-validation nested inside a 5-fold cross-validation on 4 CPUs.

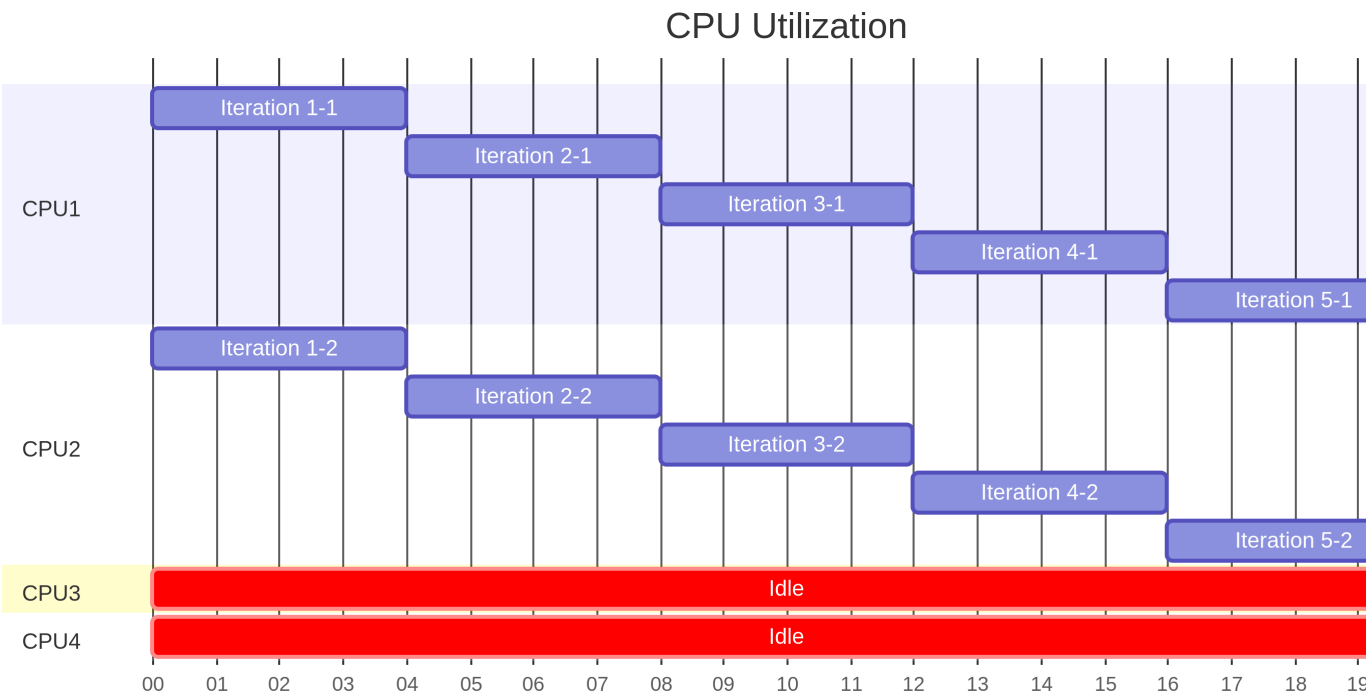


Figure 8.3: CPU utilization while parallelizing the inner resampling of a 2-fold cross-validation nested inside a 5-fold cross-validation on 4 CPUs.

If possible, the number of iterations can be adapted to the available hardware. There is no law set in stone that you have to select, e.g., 10 folds in cross-validation. If you have 4 CPUs and a reasonable variance, 8 iterations are often sufficient, or you do 12 iterations because you get the last two iterations basically for free.

Alternatively, you can also enable parallelization for both loops for nested parallelization, even on different parallelization backends. While nesting real parallelization backends is often unintended and causes unnecessary overhead, it is useful in some distributed computing setups. In this case, the number of workers must be manually tweaked so that the system does not get overburdened:

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

This example would run on 8 cores (`= 2 * 4`) on the local machine, parallelizing the outer resampling on 2, and the inner resampling on 4 workers. The vignette of the `future` package gives more insight into nested parallelization. For more background information about parallelization during tuning, see Section 6.7 of Bischl et al. (2021).

> **!** Important
>
> During tuning with `mlr3tuning`, you can often adjust the **batch size** of the `Tuner`, i.e., control how many hyperparameter configurations are evaluated in parallel. If you want full parallelization, make sure that the batch size multiplied by the number of (inner) resampling iterations is at least equal to the number of available workers. If you expect homogeneous runtimes, i.e., you are tuning over a single learner or linear pipeline and you have no hyperparameter which is likely to influence the runtime, aim for a multiple of the number of workers.
> In general, larger batches mean more parallelization, while smaller batches imply a more frequent evaluation of termination criteria. We default to a `batch_size` of 1 that ensures that all `Terminator`s work as intended, i.e., you cannot exceed the computational budget.

Heterogeneous runtimes add an extra layer of complexity to parallelization. This occurs frequently, especially in tuning, when a hyperparameter strongly influences the runtime of the learning procedure. Examples are the number of trees for random forests or the number of regularization values to be tested in penalized regression.

How efficient the parallelization turns out depends in particular on the scheduling strategy of the backend. After the first batch of jobs is sent to the worker, the next jobs are either started

(a) as soon as all results have been collectively reported back to the main process, or
(b) as soon as the first job reports back.

Method (a) usually comes with less synchronization overhead and is best suited for short jobs with homogeneous runtimes. Method (b) is faster if the runtimes are heterogeneous, especially if the parallelization overhead is neglectable in comparison with the runtime for the computation. E.g., for `parallel::mclapply()`, the behavior of the scheduler can

be controlled with the `mc.preschedule` option and `parallel::parSapply()` implements Method (a) while `parallel::parSapplyLB()` implements scheduling according to method (b). FIXME: future options

8.1.5 Parallelization of Predictions

Finally, also the prediction of a single learner can be parallelized as the predictions of two observations are independent. For most learners, training is the bottleneck and parallelizing the prediction is not a worthwhile endeavor, but of course there are exceptions.

Technically, the test data is first split into multiple groups and the predict-method of the learner is applied to each group in parallel using active backend configured via `future::plan()`. The resulting predictions are then combined internally in a second step. However, to avoid predicting in parallel accidentally, parallel predictions must first be enabled in the learner via the `parallel_predict` field:

```
# train random forest on spam task
task = tsk("spam")
learner = lrn("classif.ranger")
learner$train(task)

# set up parallel predict on 4 workers
future::plan("multisession", workers = 4)
learner$parallel_predict = TRUE

# perform prediction
prediction = learner$predict(task)
```

The resulting `Prediction` is identical to the one computed sequentially.

8.1.6 Reproducibility

Usually, reproducibility is a major concern during parallelization because special (PRNGs[6]) are required (see this blog post[7]. A simple `set.seed()` is not sufficient when parallelization is involved. One general recommendation here is switching from the default Mersenne Twister[8] to Pierre L'Ecuyer's RngStreams (see `Random` and `parallel::RNGstreams`). However, even this parallel PRNG comes with pitfalls w.r.t. reproducibility: if you change the number of workers, you still get different results after setting the seed with `set.seed()`.

Luckily, this and many other problems are already addressed by the excellent `future` parallelization framework which `mlr3` uses under the hood. `future` ensures that all workers will receive the exactly same PRNG streams, independent of the number of workers. Although correct seeding alone does not guarantee full reproducibility, it is one problem less to worry about. You can find more details about the used PRNG in this blog post[9].

[6] https://en.wikipedia.org/wiki/Pseudorandom_number_generator
[7] https://www.jottr.org/2020/09/22/push-for-statistical-sound-rng/
[8] https://en.wikipedia.org/wiki/Mersenne_Twister
[9] https://www.jottr.org/2020/09/22/push-for-statistical-sound-rng/

8.2 Error Handling

In large ML experiments, it is not uncommon that a model fit or prediction fails with an error. This is because the algorithms have to process arbitrary data, and not all eventualities can always be handled. While we try to identify obvious problems before execution, such as when missing values occur, but a learner cannot handle them, other problems are far more complex to detect. Examples include correlations or collinearity that make model fitting impossible, outliers that lead to numerical problems, or new levels of categorical variables appearing in the predict step. The learners behave quite differently when encountering such problems: some models signal a warning during the train step that they failed to fit but return a baseline model while other models stop the execution. During prediction, some learners just refuse to predict the response for observations they cannot handle while others predict a missing value. How to deal with these problems even in more complex setups like benchmarking or tuning is the topic of this section.

For illustration (and internal testing) of error handling, `mlr3` ships with the learners `"classif.debug"` and `"regr.debug"`. Here, we use the debug learner for classification to demonstrate the error handling:

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

```
<LearnerClassifDebug:classif.debug>: Debug Learner for Classification
* Model: -
* Parameters: list()
* Packages: mlr3
* Predict Types:  [response], prob
* Feature Types: logical, integer, numeric, character, factor, ordered
* Properties: hotstart_forward, missings, multiclass, twoclass
```

This learner comes with special parameters that let us simulate problems frequently encountered in ML. E.g., the debug learner comes with hyperparameters to 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 nlevels        default value
 1:        error_predict ParamDbl     0     1     Inf              0
 2:          error_train ParamDbl     0     1     Inf              0
 3:      message_predict ParamDbl     0     1     Inf              0
 4:        message_train ParamDbl     0     1     Inf              0
 5:      predict_missing ParamDbl     0     1     Inf              0
 6: predict_missing_type ParamFct    NA    NA       2             na
```

```
 7:           save_tasks ParamLgl    NA    NA       2          FALSE
 8:     segfault_predict ParamDbl     0     1     Inf              0
 9:       segfault_train ParamDbl     0     1     Inf              0
10:          sleep_train ParamUty    NA    NA     Inf <NoDefault[3]>
11:        sleep_predict ParamUty    NA    NA     Inf <NoDefault[3]>
12:              threads ParamInt     1   Inf     Inf <NoDefault[3]>
13:      warning_predict ParamDbl     0     1     Inf              0
14:        warning_train ParamDbl     0     1     Inf              0
15:                    x ParamDbl     0     1     Inf <NoDefault[3]>
16:                 iter ParamInt     1   Inf     Inf              1
```

With the learner's default settings, the learner will do nothing special: The learner remembers a random label and constantly predicts this label:

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

```
           truth
response    Adelie Chinstrap Gentoo
  Adelie         0         0      0
  Chinstrap      0         0      0
  Gentoo       152        68    124
```

We now set a hyperparameter to let the debug learner signal an error during the train step. By default, `mlr3` does not catch conditions such as warnings or errors raised while calling learners:

```
# set probability to signal an error to 1
learner$param_set$values = list(error_train = 1)

learner$train(tsk("penguins"))
```

```
Error in .__LearnerClassifDebug__.train(self = self, private = private, : Error from classif.debug
```

If this has been a regular learner, we could now start debugging with `traceback()` (or create a Minimal Reproducible Example (MRE)[10] to tackle the problem down or file a bug report upstream). However, due to the nature of the problem, it is likely that the cause of the error cannot be fixed - so you have to learn how to deal with the errors.

> **Note**
>
> If you start debugging, make sure you have disabled parallelization to avoid various pitfalls related to parallelization. It may also be helpful to set the option `mlr3.debug` to `TRUE`. If this flag is set, `mlr3` does not call into the `future` package, resulting in an easier-to-interpret program flow and `traceback()`.

8.2.1 Encapsulation

Since ML algorithms are confronted with arbitrary, often messy data, errors are not uncommon here, and we often just need to move on during benchmarking or tuning. Thus, we

[10] https://stackoverflow.com/help/minimal-reproducible-example

need a mechanism to

1. capture all signaled conditions such as messages, warnings and errors so that we can analyze them post-hoc (called "encapsulation", covered in this section),
2. deal with algorithms which do not terminate in a reasonable time, and
3. a statistically sound way to proceed while being able to aggregate over partial results (next Section 8.2.2).

Encapsulation ensures that signaled conditions (such as messages, warnings and errors) are intercepted: all conditions raised during the training or predict step are logged into the learner, and errors do not interrupt the program flow. I.e., the execution of the calling function or package (here: `mlr3`) continues as if there had been no error, though the result (fitted model during `train()`, predictions during `predict()`) are missing. Each `Learner` has a field `encapsulate` to control how the train or predict steps are wrapped. The easiest way to encapsulate the execution is provided by the package `evaluate` which evaluates R expressions while tracking conditions such as outputs, messages, warnings or errors (see the documentation of the `encapsulate()` helper function for more details):

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

# this learner throws a warning and then stops with an error during train()
learner$param_set$values = list(warning_train = 1, error_train = 1)

# enable encapsulation for train() and predict()
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. In contrast to `evaluate`, the computation is taken out in a separate R process. This guards the calling session against segmentation faults which otherwise would tear down the complete main R session. On the downside, starting new processes comes with comparably more 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"
```

With either of these encapsulation methods, we can now catch errors and post-hoc analyze the messages, warnings and error messages. Additionally, a timeout can be set so that learners do not run for an indefinite time but are terminated after a specified time. Interrupting learners works differently depending on the encapsulation (see `mlr3misc::encapsulate()`) and, when viewed from the outside, behave as if they would signal an error after reaching the timeout. The timeout can be set separately for training and prediction and must be provided in seconds:

```
# 5 minute timeout for training, no timeout for predict
learner$timeout = c(train = 5 * 60, predict = Inf)
```

Unfortunately, catching errors and ensuring an upper time limit is only half the battle. Without a model, it is not possible to get predictions:

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

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

To handle the missing predictions gracefully during `resample()`, `benchmark()` or tuning, fallback learners are introduced next.

8.2.2 Fallback learners

Fallback learners have the purpose of being able to score results in cases where a `Learner` completely failed to fit a model or refuses to provide predictions for some or all observations.

We will first handle the case that a learner fails to fit a model during training, e.g., if some convergence criterion is not met or the learner ran out of memory. There are in general three possibilities to proceed:

1. Ignore iterations with failed model fits. Although this is arguably the most frequent approach in practice, it is **not** statistically sound. For example, consider the case where a researcher wants a specific learner to look better in a benchmark study. To do this, the researcher takes an existing learner but introduces a small adaptation: If an internal goodness-of-fit measure is not achieved, an error is thrown. In other words, the learner only fits a model if the model can be reasonably well learned on the given training data. In comparison with the learning procedure without this adaptation and a good threshold, however, we now compare the mean over only the "easy" splits with the mean over all splits - an unfair advantage.
2. Penalize failing learners. Instead of ignoring failed iterations, we can simply impute the worst possible score (as defined by the `Measure`) and thereby heavily penalize the learner for failing. However, this often seems too harsh for many problems, and for some measures there is no reasonable value to impute.

3. Impute a value that corresponds to a (weak) baseline. Instead of imputing with the worst possible score, impute with a reasonable baseline, e.g., by just predicting the majority class or the mean of the response in the training data. Such simple baselines are implemented as featureless learners (`mlr_learners_classif.featureless` or `mlr_learners_regr.featureless`). Note that a reasonable baseline value is different in different training splits. Retrieving these values after a larger benchmark study has been conducted is possible, but tedious.

We strongly recommend option (3): it is statistically sound and very flexible. To make this procedure very convenient during resampling and benchmarking, we support fitting a proper baseline with a 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 single time with the given parametrization) and encapsulation is enabled, `mlr3` falls back to the predictions of the featureless learner internally:

```
task = tsk("penguins")

learner = lrn("classif.debug")
learner$param_set$values = list(error_train = 1)
learner$fallback = lrn("classif.featureless")

learner$train(task)
learner
```

```
<LearnerClassifDebug:classif.debug>: Debug Learner for Classification
* Model: -
* Parameters: error_train=1
* Packages: mlr3
* Predict Types:  [response], prob
* Feature Types: logical, integer, numeric, character, factor, ordered
* Properties: hotstart_forward, missings, multiclass, twoclass
* Errors: Error from classif.debug->train()
```

Note that encapsulation is not enabled explicitly; it is automatically set to `"evaluate"` for the training and the predict step while setting a fallback learner for a learner without encapsulation enabled. Furthermore, the log contains the captured error (which is also included in the print output), and although no model is stored, we can still get predictions:

```
learner$model
```

```
NULL
```

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

```
classif.ce
 0.5581395
```

In this stepwise train-predict procedure, the fallback learner is of limited use. However, it is invaluable for larger benchmark studies.

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("penguins"), list(learner, lrn("classif.rpart")), rsmp("cv")
aggr = bmr$aggregate(conditions = TRUE)
aggr[, .(learner_id, warnings, errors, classif.ce)]
```

```
      learner_id warnings errors classif.ce
1: classif.debug        0      3 0.59226891
2: classif.rpart        0      0 0.05218487
```

Even though the debug learner occasionally failed to provide predictions, we still got a statistically sound aggregated performance value which we can compare to the aggregated performance of the classification tree. It is also possible to split the benchmark up into separate `ResampleResult` objects which sometimes helps to get more context. E.g., if we only want to have a closer look into the debug learner, we can extract the errors from the corresponding resample results:

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

```
   iteration                                msg
1:         1 Error from classif.debug->train()
2:         5 Error from classif.debug->train()
3:         8 Error from classif.debug->train()
```

A problem similar to failed model fits emerges when a learner predicts only a subset of the observations in the test set (and predicts `NA` or no value for others). A typical case is, e.g., when new and unseen factor levels are encountered in the test data. Imagine again that our goal is to benchmark two algorithms using cross-validation on some binary classification task:

- Algorithm A is an ordinary logistic regression.
- Algorithm B is also an 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 returns a missing value otherwise.

At its core, this is the same problem as outlined before. If we measure the performance using only the non-missing predictions, Algorithm B would likely outperform algorithm A. However, this approach does not factor in that you can not generate predictions for all observations. Long story short, if a fallback learner is specified, 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("penguins")
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")
```

```
Adelie Chinstrap    Gentoo      <NA>
   172         0         0       172
```

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

```
Adelie Chinstrap    Gentoo      <NA>
   344         0         0         0
```

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

8.3 Logging

mlr3 internally uses the lgr package to control the verbosity of the output, e.g., suppress messages or enable additional debugging messages.

8.3.1 Changing mlr3 logging levels

All log messages have an associated level which encodes a number, e.g. informative messages have the level `"info"` which is associated with the number 400 while debugging messages have the level `"debug"` with number 500. A message is only displayed if the respective level exceeds the global logging threshold.

To change the setting for mlr3 for the current session, you need to retrieve the logger (which is a R6 object) from lgr, and then change the threshold of the like this:

```
requireNamespace("lgr")

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

The default log level is `"info"`. All available levels can be listed as follows:

```
getOption("lgr.log_levels")
```

```
fatal error  warn  info debug trace
```

```
  100   200   300   400   500   600
```

To increase verbosity, set the log level to a higher value, e.g. to `"debug"` with:

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

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

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

`lgr` comes with a global option called `"lgr.default_threshold"` which can be set via `options()` to make your choice permanent across sessions.

Also note that the optimization packages such as `mlr3tuning` or `mlr3fselect` use the logger of their base package `bbotk`. If you want do disable logging from `mlr3`, but keep the output from `mlr3tuning`, reduce the verbosity of the `mlr3` logger and change the `bbotk` logger to the desired level.

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

8.3.2 Redirecting output

Redirecting output is already extensively covered in the documentation and vignette of `lgr`. Here is just a short example that adds an appender to log events additionally to a temporary file using the JSON format:

```
tf = tempfile("mlr3log_", fileext = ".json")

# get the logger as R6 object
logger = lgr::get_logger("mlr")

# add Json appender
logger$add_appender(lgr::AppenderJson$new(tf), name = "json")

# signal a warning
logger$warn("this is a warning from mlr3")

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

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

8.3.3 Immediate Log Feedback

`mlr3` uses `future` and encapsulation to make evaluations fast, stable, and reproducible. However, this may lead to logs being delayed, out of order, or, in case of some errors, not present at all.

When it is necessary to have immediate access to log messages, for example to investigate problems, one may therefore choose to disable `future` and encapsulation. This can be done by enabling the debug mode using `options(mlr.debug = TRUE)`; the `$encapsulate` slot of learners should also be set to `"none"` (default) or `"evaluate"`, but not `"callr"`. Enabling the debug mode should only be done to investigate problems, however, and not for production use, because

1. this disables parallelization, and
2. this leads to different RNG behavior and therefore to results that are not fully reproducible.

8.4 Data Backends

> **!** **Advanced section**

In mlr3, `Task` objects store their data in an abstract data object, the `DataBackend`. A backend provides a unified API to retrieve subsets of the data or query information about it, regardless of how the data is actually stored on the system. The default backend uses `data.table` via the `DataBackendDataTable` as a very fast and efficient in-memory database. For example, we can query the dimensions of the penguins task:

```
task = tsk("penguins")
backend = task$backend
backend$nrow
```

```
[1] 344
```

```
backend$ncol
```

```
[1] 9
```

While storing the Task's data in memory is most efficient w.r.t. accessing it for model fitting, this has two major disadvantages:

1. Although only a small proportion of the data is required, the complete data frame sits in memory and consumes memory. This is especially a problem if you work with large tasks or many tasks simultaneously, e.g., for benchmarking.
2. During parallelization, the complete data needs to be transferred to the workers which can increase the overhead.

To avoid these drawbacks, especially for larger data, it can be necessary to interface out-of-memory data to reduce the memory requirements. This way, only the part of the data which is currently required by the learners will be placed in the main memory to operate on. There are multiple options to archive this:

1. `DataBackendDplyr` which interfaces the R package `dbplyr`, extending `dplyr` to

work on many popular databases like MariaDB[11], PostgresSQL[12] or SQLite[13].
2. `DataBackendDuckDB` for the impressive DuckDB[14] database connected via `duckdb`: a fast, zero-configuration alternative to SQLite.
3. `DataBackendDuckDB`, again, but for Parquet files[15]. The data does not need to be converted to DuckDB's native storage format, you can work directly on directories containing one or multiple files stored in the popular Parquet format.

In the following, we will show how to work with data backends that are available through `mlr3db`.

8.4.1 Databases with DataBackendDplyr

To demonstrate the `mlr3db::DataBackendDplyr` we use the NYC flights data set from the `nycflights13` package and move it into a SQLite database. Although `as_sqlite_backend()` provides a convenient function to perform this step, we construct the database manually here.

```
# load data
requireNamespace("DBI")
requireNamespace("RSQLite")
requireNamespace("nycflights13")
data("flights", package = "nycflights13")
str(flights)
```

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

[11] https://mariadb.org/
[12] https://www.postgresql.org/
[13] https://www.sqlite.org
[14] https://duckdb.org/
[15] https://parquet.apache.org/

```
# 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)
```

With the SQLite database stored in file `path`, we now re-establish a connection and switch to `dplyr`/`dbplyr` for some essential preprocessing. If you had a real database management system (DBMS), this would be the first step now:

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

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

```
Attaching package: 'dplyr'
The following objects are masked from 'package:data.table':

    between, first, last
The following objects are masked from 'package:stats':

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

    intersect, setdiff, setequal, union
```

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

```
Attaching package: 'dbplyr'
The following objects are masked from 'package:dplyr':

    ident, sql
```

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

As databases are intended to store large volumes of data, a natural first step is to slice and dice the data to suitable dimensions. Therefore, we build up an SQL query in a step-wise fashion using `dplyr` verbs and start by selecting 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, all_of(keep))
```

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

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

To reduce the runtime for this example, we also 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))
```

Next, the processed table is 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 about the data:

```
b$nrow
```

```
[1] 163707
```

```
b$ncol
```

```
[1] 13
```

```
b$head()
```

```
   row_id year month day hour minute dep_time arr_time carrier flight air_time
1:      2 2013     1   1    5     29      533      850      UA   1714      227
2:      4 2013     1   1    5     45      544     1004      B6    725      183
3:      6 2013     1   1    5     58      554      740      UA   1696      150
4:      8 2013     1   1    6      0      557      709      EV   5708       53
5:     10 2013     1   1    6      0      558      753      AA    301      138
6:     12 2013     1   1    6      0      558      853      B6     71      158
2 variables not shown: [distance, arr_delay]
```

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.

As we now have constructed a backend, we can switch over to `mlr3` for model fitting on a task based on the previously created `mlr3db::DataBackendDplyr`:

```
task = as_task_regr(b, id = "flights_sqlite", target = "arr_delay")
learner = lrn("regr.rpart")
resampling = rsmp("subsampling", ratio = 0.02, repeats = 3)
```

We pass all these objects to `resample()` to perform a subsampling on 2% of the observations three times. In each iteration, only the required subset of the data is queried from the SQLite database and passed to `rpart::rpart()`:

```
rr = resample(task, learner, resampling)
```

```
Loading required package: RSQLite
Loading required package: RSQLite
Loading required package: RSQLite
```

```
print(rr)
```

```
<ResampleResult> with 3 resampling iterations
        task_id learner_id resampling_id iteration warnings errors
 flights_sqlite regr.rpart   subsampling         1        0      0
 flights_sqlite regr.rpart   subsampling         2        0      0
 flights_sqlite regr.rpart   subsampling         3        0      0
```

```
measures = msrs(c("regr.mse", "time_train", "time_predict"))
rr$aggregate(measures)
```

```
    regr.mse   time_train time_predict
1238.555382     1.949667    22.906000
```

Note that we still have an active connection to the database. To properly close it, we remove the `tbl` object referencing the connection and then close the connection.

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

8.4.2 Parquet Files with DataBackendDuckDB

We have already demonstrated how to operate on a SQLite database. DuckDB databases (using `DataBackendDuckDB`) provide a modern alternative to SQLite, tailored to the needs of machine learning. To convert a `data.frame` to DuckDB, we provide the helper function `as_duckdb_backend()`. Only two arguments are required: the `data.frame` to convert, and a `path` to store the data.

While this is useful while working with many tasks simultaneously in order to keep the memory requirements reasonable, the more frequent use case for DuckDB are nowadays Parquet files. Parquet is a popular column-oriented data storage format supporting efficient compression, making it far superior to other popular data exchange formats such as CSV.

To demonstrate working with Parquet files, we first query the location of an example data set shipped with `mlr3db`:

```
path = system.file(file.path("extdata", "spam.parquet"), package = "mlr3db")
```

We can then create a `DataBackendDuckDB` based on this file and convert the backend to a classification task, all without loading the dataset into memory:

```
backend = as_duckdb_backend(path)
task = as_task_classif(backend, target = "type")
print(task)
```

```
<TaskClassif:backend> (4601 x 58)
* Target: type
* Properties: twoclass
* Features (57):
  - dbl (57): address, addresses, all, business, capitalAve,
    capitalLong, capitalTotal, charDollar, charExclamation, charHash,
    charRoundbracket, charSemicolon, charSquarebracket, conference,
    credit, cs, data, direct, edu, email, font, free, george, hp, hpl,
    internet, lab, labs, mail, make, meeting, money, num000, num1999,
    num3d, num415, num650, num85, num857, order, original, our, over,
    parts, people, pm, project, re, receive, remove, report, table,
    technology, telnet, will, you, your
```

Accessing the data internally triggers a query and data is fetched to be stored in an in-memory `data.frame`, but only the required subsets. After the retrieved data is processed, the garbage collector can release the occupied memory. The backend can also operate on a folder with multiple parquet files, which is documented in `as_duckdb_backend()`.

8.5 Extending mlr3

Hopefully having read the rest of this book you are now on the way to being an `mlr3` expert. Maybe you will even want to extend the universe with new classes for more learners, measures, tuners, pipeops, and more; if so, read on.

> **!** **Advanced section**

In this chapter we will show how to extend `mlr3` using the simple example of creating a custom `Measure`. If you are interested in implementing new learners, pipeops, and tuners, then check out the vignettes in the respective packages: `mlr3extralearners`, `mlr3pipelines`, or `mlr3tuning`. Or if you are considering adding a new machine learning task then please contact us on GitHub, email, or Mattermost. This section assumes good knowledge of `R6`, see Section 1.9.1 for a brief introduction and references to further resources.

We welcome contributions from all levels of developers and if you want to add any of your new classes to our universe then please make pull requests to aligning packages, for example tuners and filters would go to `mlr3tuning` and `mlr3filters` respectively, a new survival

measure would go to `mlr3proba`, and *all* new learners go to `mlr3extralearners`. Do not worry if you make a PR to the wrong repository, we will transfer it to the right one. Please read the mlr3 Wiki[16] for coding conventions that we use if you want to add code to our organisation.

We will now turn to extending the `Measure` class to implement new metrics. As an example, let's consider a regression measure that scores a prediction as 1 if the difference between the true and predicted values are less than one standard deviation of the truth, or scores the prediction as 0 otherwise. In maths this would be defined as $f(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n} \mathbb{I}(|y_i - \hat{y}_i| < \sigma(y))$, where y contains the true values and $\hat{y}$ the predicted values for observations $i = 1, ..., n$. In code this may be written as:

```
threshold_acc = function(truth, response) {
  mean(ifelse(abs(truth - response) < sd(truth), 1, 0))
}

threshold_acc(c(100, 0, 1), c(1, 11, 6))
```

```
[1] 0.6666667
```

This measure is then bounded in [0, 1] and a larger score is better.

To use this measure in mlr3, we need to create a new `R6::R6Class`, which will inherit from `Measure` and in this case specifically inheriting from `MeasureRegr`. We will now demonstrate what the final code for this new measure would look like and then explain each line, this can be used as a template for most performance measures.

```
MeasureRegrThresholdAcc = R6::R6Class("MeasureRegrThresholdAcc",
  inherit = mlr3::MeasureRegr, # regression measure
  public = list(
    initialize = function() { # initialize class
      super$initialize(
        id = "thresh_acc", # unique ID
        packages = character(), # no dependencies
        properties = character(), # no special properties
        predict_type = "response", # measures response prediction
        range = c(0, 1), # results in values between (0, 1)
        minimize = FALSE # larger values are better
      )
    }
  ),

  private = list(
    .score = function(prediction, ...) { # define score as private method
      # define loss
      threshold_acc = function(truth, response) {
        mean(ifelse(abs(truth - response) < sd(truth), 1, 0))
      }
      # call loss function
```

[16] https://github.com/mlr-org/mlr3/wiki

```
        threshold_acc(prediction$truth, prediction$response)
      }
    )
  )
```

1. In the first two lines we name the class, here `MeasureRegrThresholdAcc`, and then state this is a regression measure that inherits from `MeasureRegr`.
2. We initialize the class by stating its unique ID is "thresh_acc", that it does not require any external packages (`packages = character()`) and that it has no special properties (`properties = character()`).
3. We then pass specific details of the loss function which are: it measures the quality of a "response" type prediction, its values range between (0, 1), and that the loss is optimised as its maximum.
4. Finally, we define the score itself as a private method called `.score` and simply pass the predictions to the function we defined earlier.

Sometimes measures require data from the training set, the task, or the learner. These are usually complex edge-cases examples, so we will not go into detail here, for working examples we suggest looking at the code for `mlr3proba::MeasureSurvSongAUC` and `mlr3proba::MeasureSurvAUC`. You can also consult the manual page of the `Measure` for an overview of other properties and meta-data that can be specified.

Once you have defined your measure you can either use it with the `R6` constructor, or by adding it to the `mlr_measures` dictionary:

```
library(mlr3verse)

task = tsk("boston_housing")
split = partition(task)
learner = lrn("regr.featureless")$train(task, split$train)
prediction = learner$predict(task, split$test)
prediction$score(MeasureRegrThresholdAcc$new())
```

```
thresh_acc
 0.7185629
```

```
# or add to dictionary
mlr3::mlr_measures$add("regr.thresh_acc", MeasureRegrThresholdAcc)
prediction$score(msr("regr.thresh_acc"))
```

```
thresh_acc
 0.7185629
```

Even though we only showed how to create a custom measure, the process of adding other objects is in essence the same:

1. Find the right class to inherit from
2. Add methods that:
   a) Initialize the object with the correct properties (`$initialize()`).

b) Implement the public and private methods that do the actual computation. In the above example this was the private `$.score()` method.

As a lot of classes already exist in `mlr3`, we recommend looking at similar classes that are already available.

8.6 Conclusion

This chapter describes many advanced topics that are not relevant to each and every user and cannot be covered in full detail here. However, the sections presented deal with problems that many users will encounter with some regularity. Therefore, at least superficial knowledge is necessary to be able to identify the problem specifically and to be able to return to the correct section or, for more background information, the following references.

Resources

- Schmidberger et al. (2009) and Eddelbuettel (2020) give a more systematic and in-depth overview about the possibilities to parallelize with R
- The vignette[17] of the `lgr` package demonstrates advanced logging capabilities, e.g., logging to JSON files or retrieving logged objects for debugging.
- Extending and customizing objects is covered in the documentation of the respective packages:
  - for learners see the vignette of `mlr3learners`
  - for pipe operators see the vignette `mlr3pipelines`
- For an overview of available DBMS in R, see the CRAN task view on databases[18], and in particular the vignettes of the `dbplyr` package for DBMS readily available in mlr3. For working directly with a SQL database, we recommend a general purpose SQL Tutorial[19].

8.7 Exercises

Parallelization

Consider the following example where you resample a learner (debug learner, sleeps for 3 seconds during train) on 4 workers using the multisession backend:

```
task = tsk("penguins")
learner = lrn("classif.debug", sleep_train = function() 3)
resampling = rsmp("cv", folds = 6)


future::plan("multisession", workers = 4)
```

[17] https://cran.r-project.org/web/packages/lgr/vignettes/lgr.html
[18] https://cran.r-project.org/view=Databases
[19] https://www.w3schools.com/sql/

```
resample(task, learner, resampling)
```

- Assuming that the learner would actually calculate something and not just sleep: Would all CPUs be busy?
- Prove your point by measuring the elapsed time, e.g., using `system.time()`.
- What would you change in the setup and why?

Custom Measures

Create a new custom classification measure which scores predictions using the mean over the following classification costs:

1. If the learner predicted label "A" and the truth is "A", assign score 0
2. If the learner predicted label "B" and the truth is "B", assign score 0
3. If the learner predicted label "A" and the truth is "B", assign score 1
4. If the learner predicted label "B" and the truth is "A", assign score 10

Hint: You can implement it yourself as demonstrated in Section 8.5 or use the measure `mlr_measures_classif.costs`.

9

Model Interpretation

Przemysław Biecek
MI2.AI, Warsaw University of Technology

Predictive models have numerous applications in virtually every area of life. The increasing availability of data and frameworks to create models has allowed the widespread adoption of these solutions. However, this does not always go together with enough testing of the models and the consequences of incorrect predictions can be severe. The bestseller book „Weapons of Math Destruction'' (O'Neil 2016) discusses examples of deployed black-boxes that have led to wrong-headed decisions, sometimes on a massive scale. So what can we do to make our models more thoroughly tested? The answer is methods that allow deeper interpretation of predictive models. In this chapter, we will provide illustrations of how to perform the most popular of these methods (Holzinger et al. 2022).

In principle, all generic frameworks for model interpretation apply to the models fitted with `mlr3` by just extracting the fitted models from the `Learner` objects.

However, two of the most popular frameworks additionally come with some convenience for `mlr3`, these are

- `iml` presented in Section 9.2, and
- `DALEX` presented in Section 9.3.

Both these packages offer similar functionality, but they differ in design choices. `iml` is based on the R6 class system and for this reason working with it is more similar in style to working with the `mlr3` package. `DALEX` is based on the S3 class system and is mainly focused on the ability to compare multiple different models on the same graph for comparison and on the explainable model analysis process.

9.1 Penguin Task

To understand what model interpretation packages can offer, we start with a thorough example. The goal of this example is to figure out the species of penguins given a set of features. The `palmerpenguins::penguins` (Horst, Hill, and Gorman 2020) data set will be used which is an alternative to the `iris` data set. The `penguins` data sets contain 8 variables of 344 penguins:

```
data("penguins", package = "palmerpenguins")
str(penguins)
```

```
tibble [344 x 8] (S3: tbl_df/tbl/data.frame)
 $ species          : Factor w/ 3 levels "Adelie","Chinstrap",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ island           : Factor w/ 3 levels "Biscoe","Dream",..: 3 3 3 3 3 3 3 3 3 3 ...
```

```
 $ bill_length_mm   : num [1:344] 39.1 39.5 40.3 NA 36.7 39.3 38.9 39.2 34.1 42 ...
 $ bill_depth_mm    : num [1:344] 18.7 17.4 18 NA 19.3 20.6 17.8 19.6 18.1 20.2 ...
 $ flipper_length_mm: int [1:344] 181 186 195 NA 193 190 181 195 193 190 ...
 $ body_mass_g      : int [1:344] 3750 3800 3250 NA 3450 3650 3625 4675 3475 4250 ...
 $ sex              : Factor w/ 2 levels "female","male": 2 1 1 NA 1 2 1 2 NA NA ...
 $ year             : int [1:344] 2007 2007 2007 2007 2007 2007 2007 2007 2007 2007 ...
```

To get started run:

```
library("mlr3")
library("mlr3learners")
set.seed(1)
```

```
penguins = na.omit(penguins)
task_peng = as_task_classif(penguins, target = "species")
```

`penguins = na.omit(penguins)` is to omit the 11 cases with missing values. If not omitted, there will be an error when running the learner from the data points that have N/A for some features.

```
learner = lrn("classif.ranger")
learner$predict_type = "prob"
learner$train(task_peng)
learner$model
```

```
Ranger result

Call:
 ranger::ranger(dependent.variable.name = task$target_names, data = task$data(),      probability
```

```
Type:                             Probability estimation
Number of trees:                  500
Sample size:                      333
Number of independent variables:  7
Mtry:                             2
Target node size:                 10
Variable importance mode:         none
Splitrule:                        gini
OOB prediction error (Brier s.):  0.01790106
```

```
x = penguins[which(names(penguins) != "species")]
```

As explained in Section Learners, specific learners can be queried with `mlr_learners`. In Section Train/Predict it is recommended for some classifiers to use the `predict_type` as `prob` instead of directly predicting a label. This is what is done in this example. `penguins[which(names(penguins) != "species")]` is the data of all the features and `y` will be the penguins`species`. `learner$train(task_peng)` trains the model and `learner$model` stores the model from the training command. `Predictor` holds the machine learning model and the data. All interpretation methods in `iml` need the machine learning model and the data to be wrapped in the `Predictor` object.

9.2 iml

Author: Shawn Storm

`iml` is an R package that interprets the behaviour and explains predictions of machine learning models. The functions provided in the `iml` package are model-agnostic which gives the flexibility to use any machine learning model.

This chapter provides examples of how to use `iml` with `mlr3`. For more information refer to the IML github and the IML book

Next is the core functionality of `iml`. In this example, three separate interpretation methods will be used: FeatureEffects, FeatureImp and Shapley

- `FeatureEffects` computes the effects for all given features on the model prediction. Different methods are implemented: Accumulated Local Effect (ALE) plots, Partial Dependence Plots (PDPs) and Individual Conditional Expectation (ICE) curves.
- `Shapley` computes feature contributions for single predictions with the Shapley value – an approach from cooperative game theory (Shapley Value).
- `FeatureImp` computes the importance of features by calculating the increase in the model's prediction error after permuting the feature (more here).

9.2.1 FeatureEffects

In addition to the commands above the following two need to be run:

```
library("iml")

model = Predictor$new(learner, data = x, y = penguins$species)

num_features = c("bill_length_mm", "bill_depth_mm", "flipper_length_mm", "body_mass_g", "year")
effect = FeatureEffects$new(model)
plot(effect, features = num_features)
```

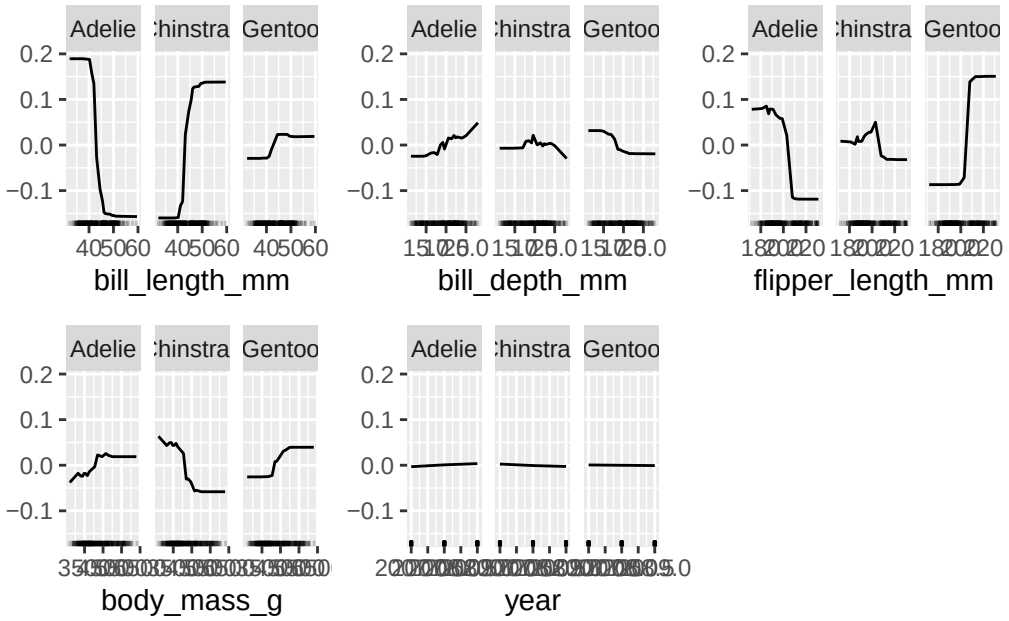


Figure 9.1: Plot of the results from FeatureEffects. FeatureEffects computes and plots feature effects of prediction models

`effect` stores the object from the `FeatureEffect` computation and the results can then be plotted. In this example, all of the features provided by the `penguins` data set were used.

All features except for `year` provide meaningful interpretable information. It should be clear why `year` doesn't provide anything of significance. `bill_length_mm` shows for example that when the bill length is smaller than roughly 40mm, there is a high chance that the penguin is an Adelie.

9.2.2 Shapley

```
x = penguins[which(names(penguins) != "species")]
model = Predictor$new(learner, data = penguins, y = "species")
x.interest = data.frame(penguins[1, ])
shapley = Shapley$new(model, x.interest = x.interest)
plot(shapley)
```

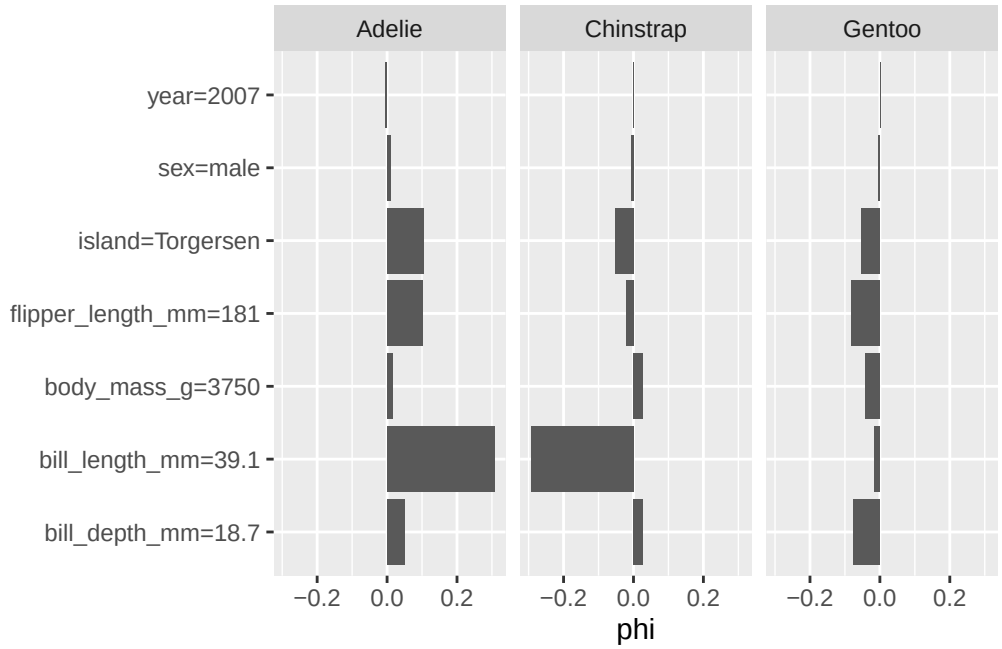


Figure 9.2: Plot of the results from Shapley. ϕ gives the increase or decrease in probability given the values on the vertical axis

The ϕ provides insight into the probability given the values on the vertical axis. For example, a penguin is less likely to be Gentoo if the bill_depth=18.7 is and much more likely to be Adelie than Chinstrap.

9.2.3 FeatureImp

```
effect = FeatureImp$new(model, loss = "ce")
effect$plot(features = num_features)
```

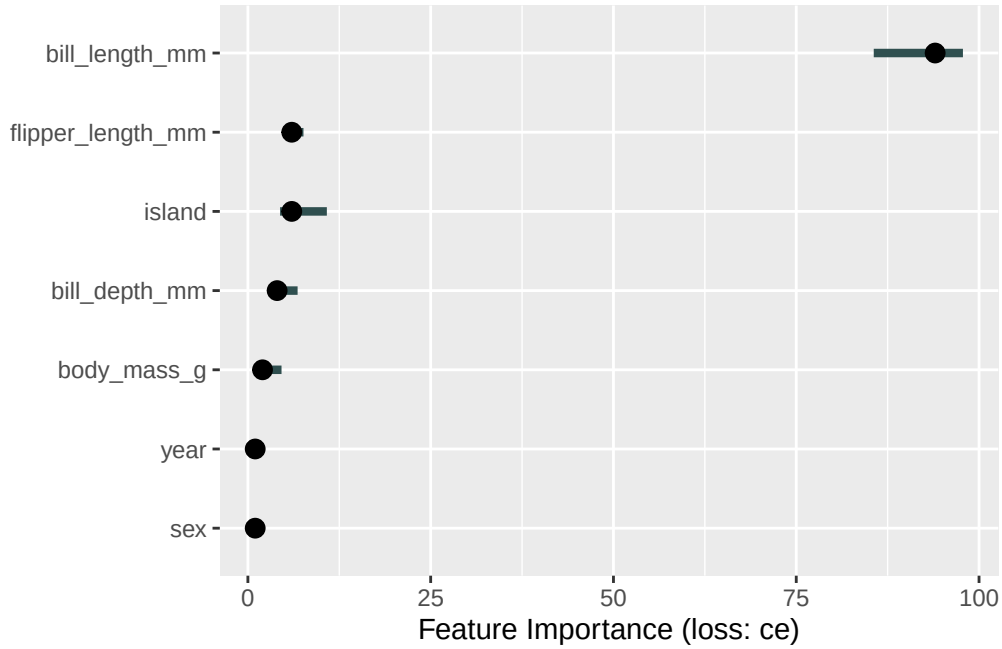


Figure 9.3: Plot of the results from FeatureImp. FeatureImp visualizes the importance of features given the prediction model

`FeatureImp` shows the level of importance of the features when classifying penguins. It is clear to see that the `bill_length_mm` is of high importance and one should concentrate on the different boundaries of this feature when attempting to classify the three species.

9.2.4 Independent Test Data

It is also interesting to see how well the model performs on a test data set. For this section, exactly as was recommended in Section Train/Predict, 80% of the penguin data set will be used for the training set and 20% for the test set:

```
train_set = sample(task_peng$nrow, 0.8 * task_peng$nrow)
test_set = setdiff(seq_len(task_peng$nrow), train_set)
learner$train(task_peng, row_ids = train_set)
prediction = learner$predict(task_peng, row_ids = test_set)
```

First, we compare the feature importance on training and test set

```
# plot on training
model = Predictor$new(learner, data = penguins[train_set, ], y = "species")
effect = FeatureImp$new(model, loss = "ce")
plot_train = plot(effect, features = num_features)

# plot on test data
model = Predictor$new(learner, data = penguins[test_set, ], y = "species")
```

```
effect = FeatureImp$new(model, loss = "ce")
plot_test = plot(effect, features = num_features)

# combine into single plot
library("patchwork")
plot_train + plot_test
```

Figure 9.4: FeatImp on train (left) and test (right)

The results of the train set for `FeatureImp` are very similar, which is expected. We follow a similar approach to compare the feature effects:

```
model = Predictor$new(learner, data = penguins[train_set, ], y = "species")
effect = FeatureEffects$new(model)
plot(effect, features = num_features)
```

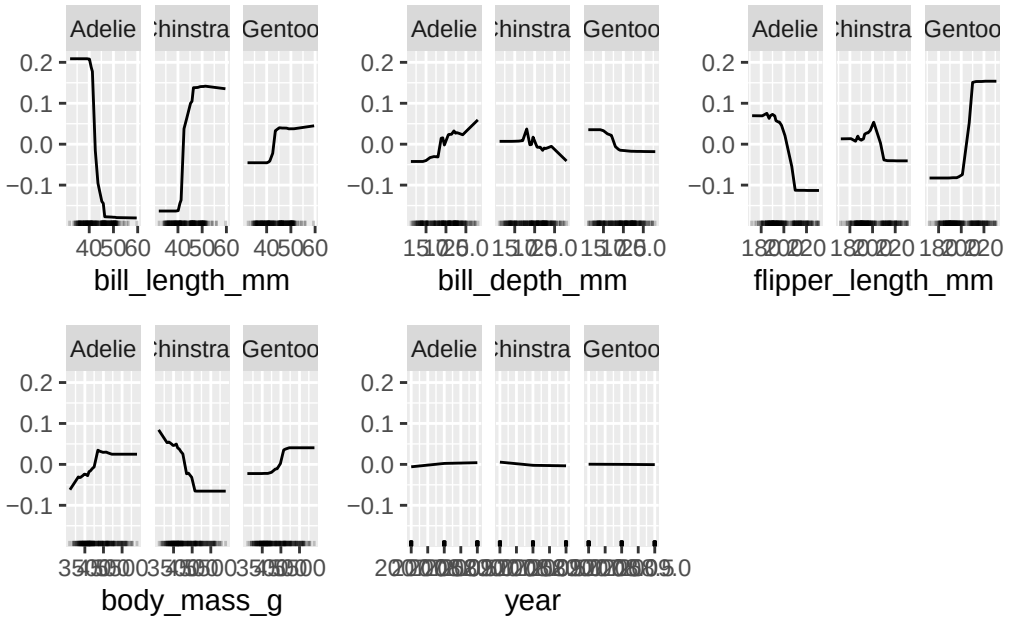


Figure 9.5: FeatEffect train data set

```
model = Predictor$new(learner, data = penguins[test_set, ], y = "species")
effect = FeatureEffects$new(model)
plot(effect, features = num_features)
```

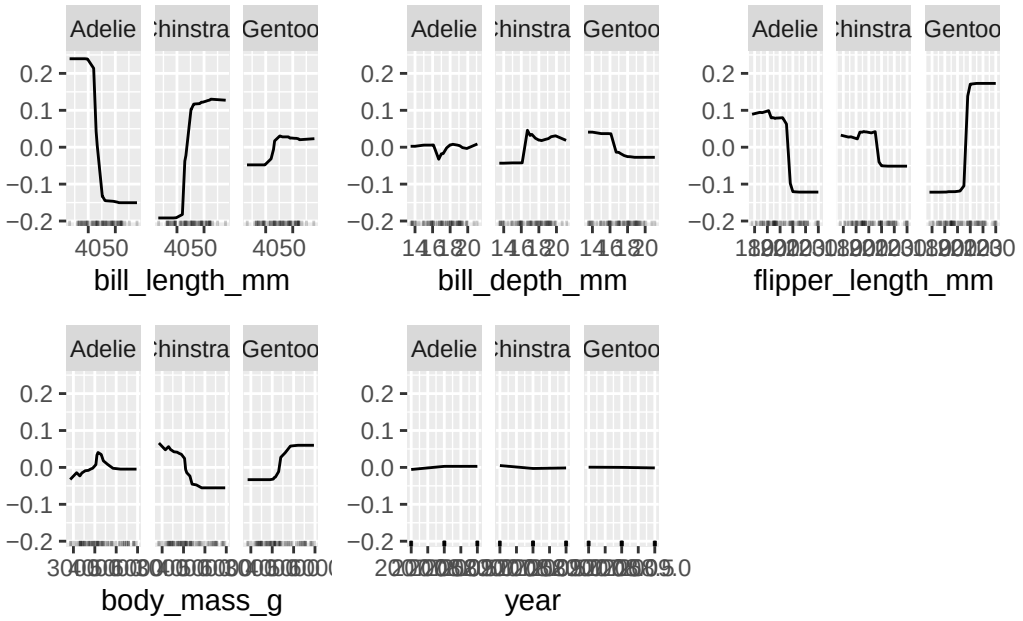


Figure 9.6: FeatEffect test data set

As is the case with `FeatureImp`, the test data results show either an over- or underestimate of feature importance / feature effects compared to the results where the entire penguin data set was used. This would be a good opportunity for the reader to attempt to resolve the estimation by playing with the amount of features and the amount of data used for both the test and train data sets of `FeatureImp` and `FeatureEffects`. Be sure to not change the line `train_set = sample(task_peng$nrow, 0.8 * task_peng$nrow)` as it will randomly sample the data again.

9.3 DALEX

The `DALEX` (Biecek 2018) package belongs to DrWhy family of solutions created to support the responsible development of machine learning models. It implements the most common methods for explaining predictive models using posthoc model agnostic techniques. You can use it for any model built with the `mlr3` package as well as with other frameworks in `R`. The counterpart in `Python` is the library `dalex` (Baniecki et al. 2021).

The philosophy of working with `DALEX` package is based on the process of explanatory model analysis described in the EMA book (Biecek and Burzykowski 2021). In this chapter, we present code snippets and a general overview of this package. For illustrative purposes, we reuse the `learner` model built in the Section 9.1 on `palmerpenguins::penguins` data.

Once you become familiar with the philosophy of working with the `DALEX` package, you can also use other packages from this family such as `fairmodels` (Wiśniewski and Biecek 2022) for detection and mitigation of biases, `modelStudio` (Baniecki and Biecek 2019) for inter-

active model exploration, `modelDown` (Romaszko et al. 2019) for the automatic generation of IML model documentation in the form of a report, `survex` (Krzyziński et al. 2023) for the explanation of survival models, or `treeshap` for the analysis of tree-based models.

9.3.1 Explanatory model analysis

The analysis of a model is usually an interactive process starting with a shallow analysis – usually a single-number summary. Then in a series of subsequent steps, one can systematically deepen understanding of the model by exploring the importance of single variables or pairs of variables to an in-depth analysis of the relationship between selected variables to the model outcome. See Bücker et al. (2022) for a broader discussion of what the model exploration process looks like.

This explanatory model analysis (EMA) process can focus on a single observation, in which case we speak of local model analysis, or for a set of observations, in which case we speak of global data analysis. Below, we will present these two scenarios in separate subsections. See Figure 9.7 for an overview of key functions that will be discussed.

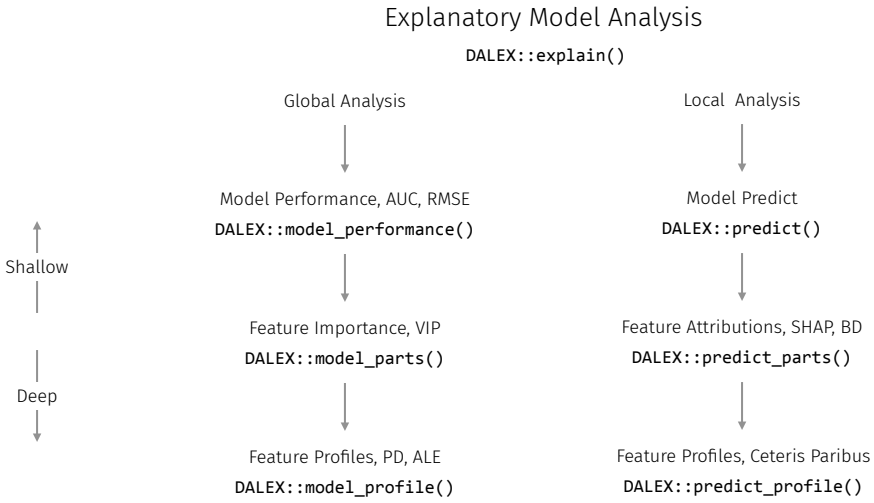


Figure 9.7: Taxonomy of methods for model exploration presented in this chapter. Left part overview methods for global level exploration while the right part is related to local level model exploration.

Predictive models in R have different internal structures. To be able to analyse them systematically, an intermediate object – a wrapper – is needed to provide a consistent interface for accessing the model. Working with explanations in the `DALEX` package always starts with the creation of such a wrapper with the use of the `DALEX::explain()` function. This function has several arguments that allow the model created by the various frameworks to be parameterised accordingly. For models created in the `mlr3` package, it is more convenient to use the `DALEXtra::explain_mlr3()`.

```
library("DALEX")
library("DALEXtra")

```

```
ranger_exp = DALEX::explain(learner,
  data = penguins[test_set, ],
  y = penguins[test_set, "species"],
  label = "Ranger Penguins",
  colorize = FALSE)
```

```
Preparation of a new explainer is initiated
  -> model label       :  Ranger Penguins
  -> data              :  67  rows  8  cols
  -> data              :  tibble converted into a data.frame
  -> target variable   :  Argument 'y' was a data frame. Converted to a vector. (  WARNING  )
  -> target variable   :  67  values
  -> predict function  :  yhat.LearnerClassif  will be used (  default  )
  -> predicted values  :  No value for predict function target column. (  default  )
  -> model_info        :  package mlr3 , ver. 0.15.0 , task multiclass (  default  )
  -> predicted values  :  predict function returns multiple columns:  3  (  default  )
  -> residual function :  difference between 1 and probability of true class (  default  )
  -> residuals         :  numerical, min =  0 , mean =  0.07756016 , max =  0.5380321
  A new explainer has been created!
```

The `DALEX::explain()` function performs a series of internal checks so the output is a bit verbose. Turn the `verbose = FALSE` argument to make it less wordy.

9.3.2 Global level exploration

The global model analysis aims to understand how a model behaves on average on a set of observations, most commonly a test set. In the `DALEX` package, functions for global analysis have names starting with the prefix `model_`.

9.3.2.1 Model Performance

As shown in Figure Figure 9.7, it starts by evaluating the performance of a model. This can be done with a variety of tools, in the `DALEX` package the default is to use the `DALEX::model_performance` function. Since the `explain` function checks what type of task is being analysed, it can select the appropriate performance measures for it. In our illustration, we have a multi-label classification, so measures such as micro-aggregated F1, macro-aggregated F1 etc. are calculated in the following snippet. One of the calculated measures is cross entropy and it will be used later in the following sections.

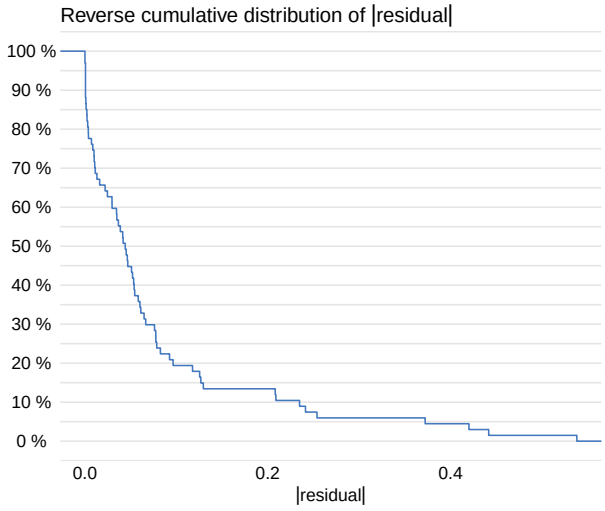
Each explanation can be drawn with the generic `plot()` function, for multi-label classification the distribution of residuals is drawn by default.

```
perf_penguin = model_performance(ranger_exp)
perf_penguin
```

```
Measures for:  multiclass
micro_F1   : 1
macro_F1   : 1
w_macro_F1 : 1
accuracy   : 1
w_macro_auc: 1
```

```
cross_entro: 6.034954

Residuals:
          0%          10%          20%          30%          40%          50%
0.0000000000 0.0005846154 0.0036863492 0.0111489133 0.0315985873 0.0440341048
         60%          70%          80%          90%         100%
0.0535907937 0.0683762754 0.0956176783 0.2191798413 0.5380321429
```

```
library("ggplot2")
old_theme = set_theme_dalex("ema")
plot(perf_penguin)
```



The task of classifying the penguin species is rather easy, which is why there are so many values of 1 in the performance assessment of this model.

9.3.2.2 Permutational Variable Importance

A popular technique for assessing variable importance in a model-agnostic manner is the permutation variable importance. It is based on the difference (or ratio) in the selected loss function after the selected variable or set of variables has been permuted. Read more about this technique in Variable-importance Measures chapter.

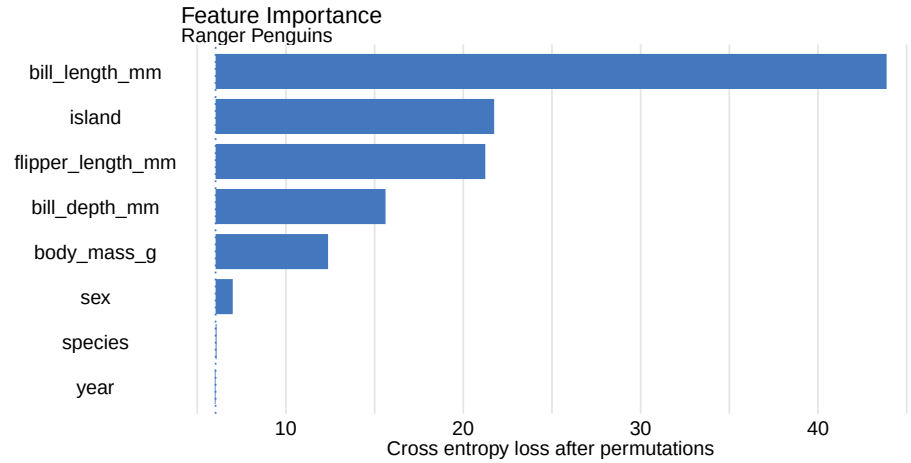
The `DALEX::model_parts()` function calculates the importance of variables and its results can be visualized with the generic `plot()` function.

```
ranger_effect = model_parts(ranger_exp)
head(ranger_effect)
```

```
      variable mean_dropout_loss           label
1  _full_model_          6.034954 Ranger Penguins
2          year          5.988560 Ranger Penguins
3       species          6.034954 Ranger Penguins
4           sex          7.002289 Ranger Penguins
```

```
5   body_mass_g         12.377824 Ranger Penguins
6 bill_depth_mm         15.617252 Ranger Penguins
```

```
plot(ranger_effect, show_boxplots = FALSE)
```



The bars start in loss (here cross-entropy loss) for the selected data and end in a loss for the data after the permutation of the selected variable. The more important the variable, the more the model will lose after its permutation.

9.3.2.3 Partial Dependence

Once we know which variables are most important, we can use Partial Dependence Plots to show how the model, on average, changes with changes in selected variables.

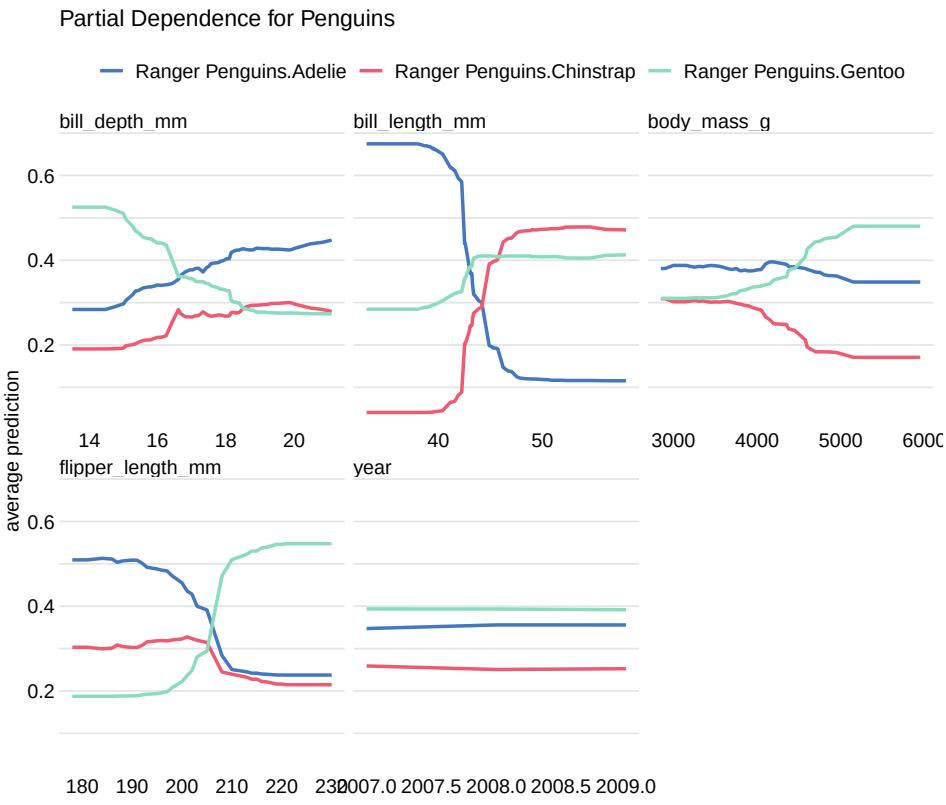
The `DALEX::model_profile()` function calculates the partial dependence profiles. The `type` argument of this function also allows *Marginal profiles* and *Accumulated Local profiles* to be calculated. Again, the result of the explanation can be model_profile with the generic function `plot()`.

```
ranger_profiles = model_profile(ranger_exp)
ranger_profiles
```

```
Top profiles    :
        _vname_                    _label_    _x_     _yhat_ _ids_
1 bill_depth_mm    Ranger Penguins.Adelie 13.500 0.2839077     0
2 bill_depth_mm Ranger Penguins.Chinstrap 13.500 0.1908264     0
3 bill_depth_mm    Ranger Penguins.Gentoo 13.500 0.5252659     0
4 bill_depth_mm    Ranger Penguins.Adelie 13.566 0.2839077     0
5 bill_depth_mm Ranger Penguins.Chinstrap 13.566 0.1908264     0
6 bill_depth_mm    Ranger Penguins.Gentoo 13.566 0.5252659     0
```

```
plot(ranger_profiles) +
  theme(legend.position = "top") +
```

```
    ggtitle("Partial Dependence for Penguins","")
```



For the multi-label classification model, profiles are drawn for each class separately by indicating them with different colours. We already know which variable is the most important, so now we can read how the model result changes with the change of this variable. In our example, based on `bill_length_mm` we can separate *Adelie* from *Chinstrap* and based on `flipper_length_mm` we can separate *Adelie* from *Gentoo*.

9.3.3 Local level explanation

The local model analysis aims to understand how a model behaves for a single observation. In the DALEX package, functions for local analysis have names starting with the prefix `predict_`.

We will carry out the following examples using Steve the penguin of the `Adelie` species as an example.

```
  steve = penguins[1,]
  steve
```

```
# A tibble: 1 x 8
  species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
```

```
  <fct>   <fct>              <dbl>         <dbl>             <int>       <int>
1 Adelie  Torgersen           39.1          18.7               181        3750
# i 2 more variables: sex <fct>, year <int>
```

9.3.3.1 Model Prediction

As shown in Figure Figure 9.7, the local analysis starts with the calculation of a model prediction.

For Steve, the species was correctly predicted as `Adelie` with high probability.

```
predict(ranger_exp, steve)
```

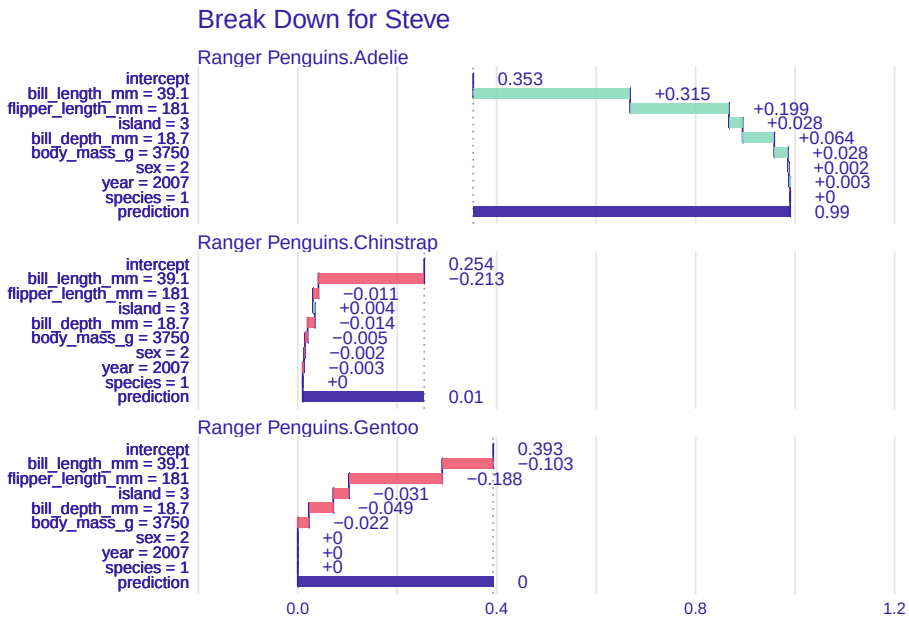
```
       Adelie   Chinstrap Gentoo
[1,] 0.9900897 0.009910317      0
```

9.3.3.2 Break Down

A popular technique for assessing the contributions of variables to model prediction is Break Down (see Introduction to Break Down chapter for more information about this method).

The function `DALEX::predict_parts()` function calculates the attributions of variables and its results can be visualized with the generic `plot()` function.

```
ranger_attributions = predict_parts(ranger_exp, new_observation = steve)
plot(ranger_attributions) + ggtitle("Break Down for Steve")
```



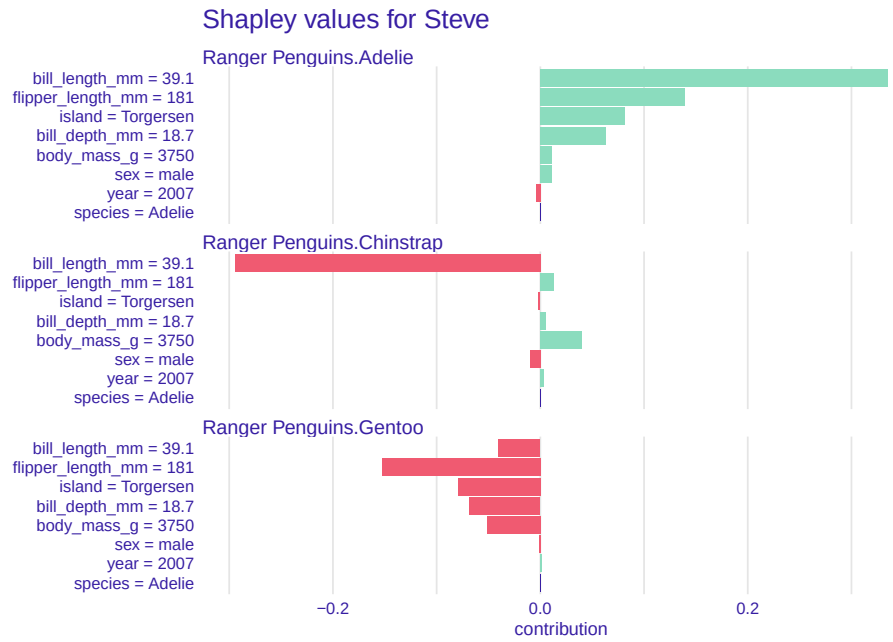
Looking at the plots above, we can read that the biggest contributors to the final prediction were for Steve the variables bill length and flipper.

9.3.3.3 Shapley Values

By far the most popular technique for local model exploration (Holzinger et al. 2022) is Shapley values and the most popular algorithm for estimating these values is the SHAP algorithm. Find a detailed description of the method and algorithm in the chapter SHapley Additive exPlanations (SHAP).

The function `DALEX::predict_parts()` calculates SHAP attributions, you just need to set `type = "shap"`. Its results can be visualized with a generic `plot()` function.

```
ranger_shap = predict_parts(ranger_exp, new_observation = steve,
              type = "shap")
plot(ranger_shap, show_boxplots = FALSE) +
              ggtitle("Shapley values for Steve", "")
```



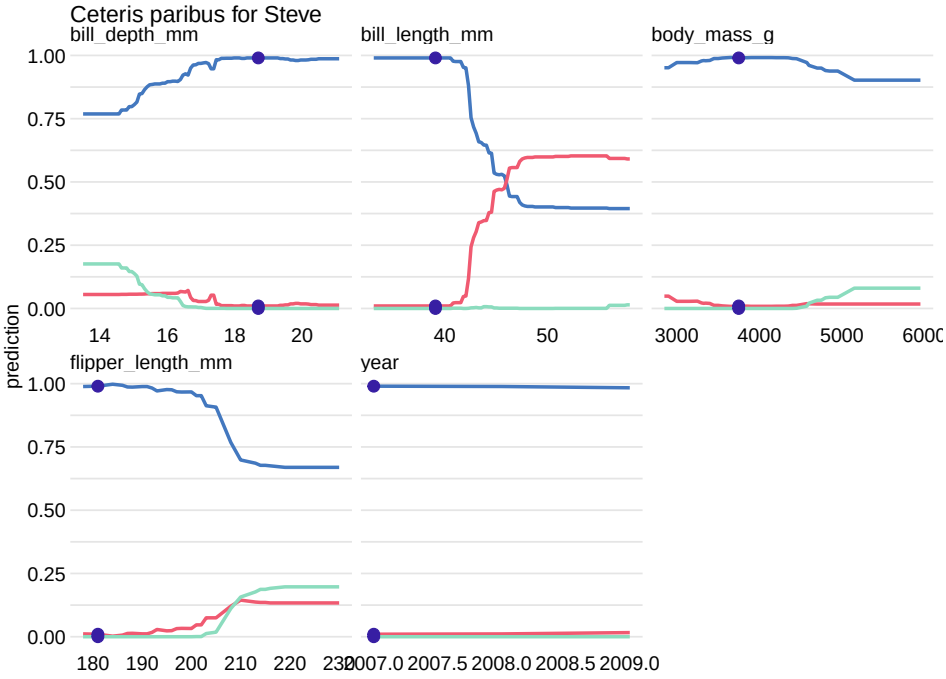
The results for Break Down and SHAP methods are generally similar. Differences will emerge if there are many complex interactions in the model.

9.3.3.4 Ceteris Paribus

In the previous section, we've introduced a global explanation – Partial Dependence plots. Ceteris Paribus plots are the local level version of that plot. Read more about this technique in the chapter Ceteris Paribus and note that these profiles are also called Individual Conditional Expectations (ICE). They show the response of a model when only one variable is changed while others stay unchanged.

The function `DALEX::predict_profile()` calculates Ceteris paribus profiles which can be visualized with the generic `plot()` function.

```
ranger_ceteris = predict_profile(ranger_exp, steve)
plot(ranger_ceteris) + ggtitle("Ceteris paribus for Steve", " ")
```



Blue dot stands for the prediction for Steve. Only a big change in bill length could convince the model of Steve's different species.

9.4 Exercises

Model explanation allows us to confront our expert knowledge related to the problem with relations learned by the model. Following tasks are based on predictions of the value of football players based on data from the FIFA game. It is a graceful example, as most people have some intuition about how a footballer's age or skill can affect their value. The latest FIFA statistics can be downloaded from kaggle.com, but also one can use the 2020 data avaliable in the `DALEX` packages(see `DALEX::fifa` dataset). The following exercises can be performed in both the `iml` and `DALEX` packages and we have provided solutions for both.

1. Prepare a `mlr3` regression task for `fifa` data. Select only variables describing the age and skills of footballers. Train any predictive model for this task, e.g. `regr.ranger`.
2. Use the permutation importance method to calculate variable importance ranking. Which variable is the most important? Is it surprising?

3. Use the Partial Dependence profile to draw the global behavior of the model for this variable. Is it aligned with your expectations?

4 Choose one of the football players. You can choose some well-known striker (e.g. Robert Lewandowski) or a well-known goalkeeper (e.g. Manuel Neuer). The following tasks are worth repeating for several different choices.

5. For the selected footballer, calculate and plot the Shapley values. Which variable is locally the most important and has the strongest influence on the valuation of the footballer?
6. For the selected footballer, calculate the Ceteris Paribus / Individual Conditional Expectatons profiles to draw the local behaviour of the model for this variable. Is it different from the global behaviour?

A

Solutions to exercises

A.1 Solutions to Chapter 2

1. Use the built in `sonar` task and the `classif.rpart` learner along with the partition function to train a model.

```
set.seed(124)
task = tsk("sonar")
learner = lrn("classif.rpart", predict_type = "prob")
measure = msr("classif.ce")
splits = partition(task, ratio=0.8)

learner$train(task, splits$train)
```

Once the model is trained, generate the predictions on the test set, define the performance measure (`classif.ce`), and score the predictions.

```
preds = learner$predict(task, splits$test)

measure = msr("classif.ce")
preds$score(measure)
```

```
classif.ce
 0.2195122
```

2. Generate a confusion matrix from the built in function.

```
preds$confusion
```

```
        truth
response  M  R
       M 20  7
       R  2 12
```

Since the rows represent predictions (response) and the columns represent the ground truth values, the TP, FP, TN, and FN rates are as follows:

- True Positive (TP) = 20
- False Positive (FP) = 2

- True Negative (TN) = 12
- False Positive (FN) = 7

3. Since in this case we want the model to predict the negative class more often, we will raise the threshold (note the `predict_type` for the learner must be `prob` for this to work).

```
# raise threshold from 0.5 default to 0.6
preds$set_threshold(0.6)

preds$confusion
```

```
        truth
response  M  R
       M 14  4
       R  8 15
```

One reason we might want the false positive rate to be lower than the false negative rate is if we felt it was worse for a positive prediction to be incorrect (meaning the true label was the negative label) than it was for a negative prediction to be incorrect (meaning the true label was the positive label).

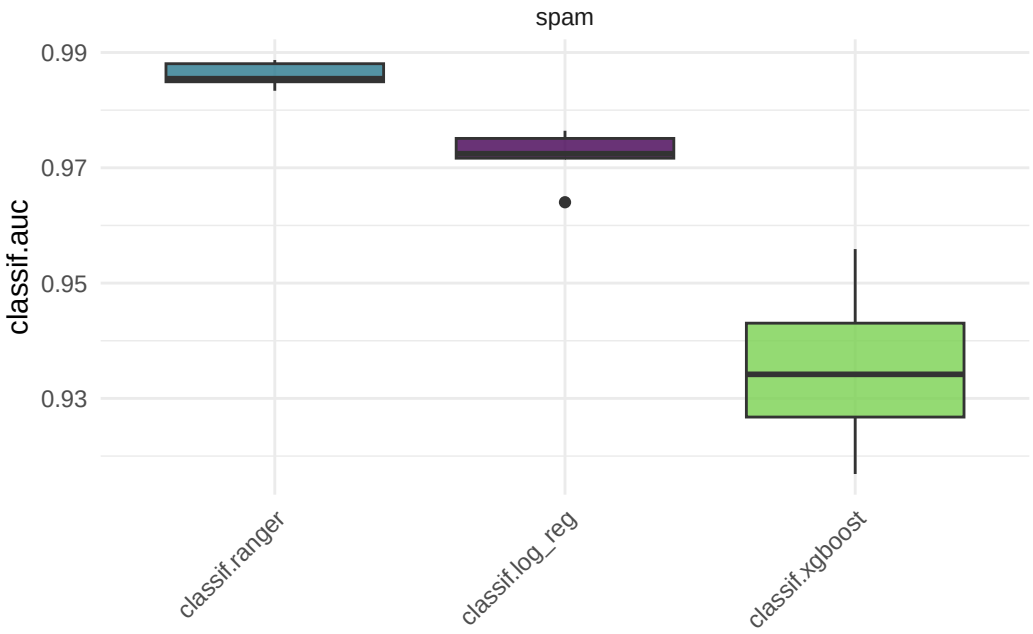
A.2 Solutions to Chapter 3

1. Use the `spam` task and 5-fold cross-validation to benchmark Random Forest (`classif.ranger`), Logistic Regression (`classif.log_reg`), and XGBoost (`classif.xgboost`) with regards to AUC. Which learner appears to do best? How confident are you in your conclusion? How would you improve upon this?

```
grid = benchmark_grid(
  tasks = tsk("spam"),
  learners = lrns(c("classif.ranger", "classif.log_reg", "classif.xgboost"), predict_type = "pr
  resamplings = rsmp("cv", folds = 5)
)

bmr = benchmark(grid)
```

```
Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

```
mlr3viz::autoplot(bmr, measure = msr("classif.auc"))
```



This is only a small example for a benchmark workflow, but without tuning (see Chapter 4), the results are naturally not suitable to make any broader statements about the superiority of either learner for this task.

2. A colleague claims to have achieved a 93.1% classification accuracy using the `classif.rpart` learner on the `penguins_simple` task. You want to reproduce their results and ask them about their resampling strategy. They said they used 3-fold cross-validation, and they assigned rows using the task's `row_id` modulo 3 to generate three evenly sized folds. Reproduce their results using the custom CV strategy.

```
task = tsk("penguins_simple")

resampling_customcv = rsmp("custom_cv")

resampling_customcv$instantiate(task = task, f = factor(task$row_ids %% 3))

rr = resample(
  task = task,
  learner = lrn("classif.rpart"),
  resampling = resampling_customcv
)

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

```
classif.acc
  0.9309309
```

A.3 Solutions to Chapter 4

1. Tune the `mtry`, `sample.fraction`, `num.trees` hyperparameters of a random forest model (`regr.ranger`) on the `Motor Trend` data set (`mtcars`). Use a simple random search with 50 evaluations and select a suitable batch size. Evaluate with a 3-fold cross-validation and the root mean squared error.

```
set.seed(4)
learner = lrn("regr.ranger",
  mtry.ratio      = to_tune(0, 1),
  sample.fraction = to_tune(1e-1, 1),
  num.trees       = to_tune(1, 2000)
)

instance = ti(
  task = tsk("mtcars"),
  learner = learner,
  resampling = rsmp("cv", folds = 3),
  measures = msr("regr.rmse"),
  terminator = trm("evals", n_evals = 50)
)

tuner = tnr("random_search", batch_size = 10)

tuner$optimize(instance)
```

```
   mtry.ratio sample.fraction num.trees learner_param_vals  x_domain regr.rmse
1:  0.2764274       0.9771886       556          <list[4]> <list[3]>  2.542653
```

2. Evaluate the performance of the model created in Question 1 with nested resampling. Use a holdout validation for the inner resampling and a 3-fold cross-validation for the outer resampling. Print the unbiased performance estimate of the model.

```
set.seed(4)
learner = lrn("regr.ranger",
  mtry.ratio      = to_tune(0, 1),
  sample.fraction = to_tune(1e-1, 1),
  num.trees       = to_tune(1, 2000)
)

at = auto_tuner(
  tuner = tnr("random_search", batch_size = 10),
```

```
  learner = learner,
  resampling = rsmp("holdout"),
  measure = msr("regr.rmse"),
  terminator = trm("evals", n_evals = 50)
)

task = tsk("mtcars")
outer_resampling = rsmp("cv", folds = 3)
rr = resample(task, at, outer_resampling, store_models = TRUE)

rr$aggregate()
```

```
regr.mse
8.322457
```

3. Tune and benchmark an XGBoost model against a logistic regression Spam data set and determine which has the best Brier score. Use mlr3tuningspaces and nested resampling.

```
library(mlr3tuningspaces)
```

```
Loading required package: mlr3tuning
```

```
Loading required package: paradox
```

```
learner_xgboost = lts(lrn("classif.xgboost", predict_type = "prob"))

at_xgboost = auto_tuner(
  tuner = tnr("random_search", batch_size = 1),
  learner = learner_xgboost,
  resampling = rsmp("cv", folds = 3),
  measure = msr("classif.bbrier"),
  term_evals = 2,
)

learner_logreg = lrn("classif.log_reg", predict_type = "prob")

at_logreg = auto_tuner(
  tuner = tnr("random_search", batch_size = 1),
  learner = learner_logreg,
  resampling = rsmp("cv", folds = 3),
  measure = msr("classif.bbrier"),
  term_evals = 2,
)

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

design = benchmark_grid(
```

```
  tasks = task,
  learners = list(at_xgboost, at_logreg),
  resamplings = outer_resampling
)

bmr = benchmark(design, store_models = TRUE)
```

```
Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

```
bmr
```

```
<BenchmarkResult> of 6 rows with 2 resampling runs
 nr task_id            learner_id resampling_id iters warnings errors
  1    spam classif.xgboost.tuned            cv     3        0      0
  2    spam classif.log_reg.tuned            cv     3        0      0
```

A.4 Solutions to Chapter 5

1. Calculate a correlation filter on the `Motor Trend` data set (`mtcars`).

```
library("mlr3verse")
filter = flt("correlation")

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

as.data.table(filter)
```

```
    feature     score
 1:      wt 0.8676594
 2:     cyl 0.8521620
 3:    disp 0.8475514
 4:      hp 0.7761684
 5:    drat 0.6811719
 6:      vs 0.6640389
 7:      am 0.5998324
 8:    carb 0.5509251
 9:    gear 0.4802848
10:    qsec 0.4186840
```

2. Use the filter from the first exercise to select the five best features in the `mtcars` data set.

```
keep = names(head(filter$scores, 5))
task$select(keep)
task$feature_names
```

```
[1] "cyl"  "disp" "drat" "hp"   "wt"
```

3. Apply a backward selection to the `penguins` data set with a classification tree learner `"classif.rpart"` and holdout resampling by the measure classification accuracy. Compare the results with those in Section 5.2.1.

```
library("mlr3fselect")
```

```
Attaching package: 'mlr3fselect'

The following object is masked from 'package:mlr3tuning':

    ContextEval
```

```
instance = fselect(
  fselector = fs("sequential", strategy = "sbs"),
  task =  tsk("penguins"),
  learner = lrn("classif.rpart"),
  resampling = rsmp("holdout"),
  measure = msr("classif.acc")
)
as.data.table(instance$result)[, .(bill_depth, bill_length, body_mass, classif.acc)]
```

```
   bill_depth bill_length body_mass classif.acc
1:      FALSE        TRUE      TRUE   0.9826087
```

```
instance$result_feature_set
```

```
[1] "bill_length" "body_mass"   "island"      "sex"         "year"
```

Answer the following questions:

a. Do the selected features differ?

Yes, the backward selection selects more features.

b. Which feature selection method achieves a higher classification accuracy?

In this example, the backwards example performs slightly better, but this depends heavily on the random seed and could look different in another run.

c. Are the accuracy values in b) directly comparable? If not, what has to be changed to make them comparable?

No, they are not comparable because the holdout sampling called with `rsmp("holdout")` creates a different holdout set for the two runs. A fair comparison would create a single resampling instance and use it for both feature selections (see Chapter 3 for details):

```
resampling = rsmp("holdout")
resampling$instantiate(tsk("penguins"))

sfs = fselect(
  fselector = fs("sequential", strategy = "sfs"),
  task =  tsk("penguins"),
  learner = lrn("classif.rpart"),
  resampling = resampling,
  measure = msr("classif.acc")
```

```
)
sbs = fselect(
  fselector = fs("sequential", strategy = "sbs"),
  task =  tsk("penguins"),
  learner = lrn("classif.rpart"),
  resampling = resampling,
  measure = msr("classif.acc")
)
as.data.table(sfs$result)[, .(bill_depth, bill_length, body_mass, classif.acc)]
```

```
   bill_depth bill_length body_mass classif.acc
1:       TRUE        TRUE     FALSE   0.9391304
```

```
as.data.table(sbs$result)[, .(bill_depth, bill_length, body_mass, classif.acc)]
```

```
   bill_depth bill_length body_mass classif.acc
1:       TRUE        TRUE      TRUE   0.9565217
```

Alternatively, one could automate the feature selection and perform a benchmark between the two wrapped learners.

4. Automate the feature selection as in Section 5.2.6 with the spam data set and a logistic regression learner (`"classif.log_reg"`). Hint: Remember to call `library("mlr3learners")` for the logistic regression learner.

```
library("mlr3fselect")
library("mlr3learners")

at = auto_fselector(
  fselector = fs("random_search"),
  learner = lrn("classif.log_reg"),
  resampling = rsmp("holdout"),
  measure = msr("classif.acc"),
  terminator = trm("evals", n_evals = 50)
)

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

bmr = benchmark(grid)

aggr = bmr$aggregate(msrs(c("classif.acc", "time_train")))
as.data.table(aggr)[, .(learner_id, classif.acc, time_train)]
```

```
                learner_id classif.acc time_train
1: classif.log_reg.fselector   0.9239290  6.0833333
2:           classif.log_reg   0.9263204  0.1103333
```

A.5 Solutions to Chapter 6

A.6 Solutions to Chapter 7

1. Run a benchmark experiment on the `german_credit` task with algorithms: `featureless`, `log_reg`, `ranger`. Tune the `featureless` model using `tunetreshold` and `learner_cv`. Use 2-fold CV and evaluate with `msr("classif.costs", costs = costs)` where you should make the parameter `costs` so that the cost of a true positive is -10, the cost of a true negative is -1, the cost of a false positive is 2, and the cost of a false negative is 3. Use `set.seed(11)` to make sure you get the same results as us. Are your results surprising?

```
library(mlr3verse)
set.seed(11)

costs = matrix(c(-10, 3, 2, -1), nrow = 2, dimnames =
  list("Predicted Credit" = c("good", "bad"),
    Truth = c("good", "bad")))
cost_measure = msr("classif.costs", costs = costs)

gr = po("learner_cv", lrn("classif.featureless", predict_type = "prob")) %>>%
  po("tunethreshold", measure = cost_measure)
task = tsk("german_credit")
learners = list(as_learner(gr), lrn("classif.log_reg"), lrn("classif.ranger"))
bmr = benchmark(benchmark_grid(task, learners, rsmp("cv", folds = 2)))
bmr$aggregate(cost_measure)[, c(4, 7)]
```

```
                         learner_id classif.costs
1: classif.featureless.tunethreshold        -6.400
2:                   classif.log_reg        -5.420
3:                    classif.ranger        -5.923
```

2. Train and predict a survival forest using `rfsrc` (from `mlr3extralearners`). Run this experiment using `task = tsk("rats"); split = partition(task)`. Evaluate your model with the RCLL measure.

```
library(mlr3verse)
library(mlr3proba)
library(mlr3extralearners)
set.seed(11)

task = tsk("rats")
split = partition(task)

```

```
  lrn("surv.rfsrc")$
    train(task, split$train)$
    predict(task, split$test)$
    score(msr("surv.rcll"))
```

```
surv.rcll
 4.030926
```

3. Estimate the density of the `tsk("precip")` data using `logspline` (from `mlr3extralearners`). Run this experiment using `task = tsk("precip"); split = partition(task)`. Evaluate your model with the logloss measure.

```
library(mlr3verse)
library(mlr3proba)
library(mlr3extralearners)
set.seed(11)

task = tsk("precip")
split = partition(task)

lrn("dens.logspline")$
  train(task, split$train)$
  predict(task, split$test)$
  score(msr("dens.logloss"))
```

```
dens.logloss
    3.979233
```

4. Run a benchmark clustering experiment on the `wine` dataset without a label column. Compare the performance of `k-means` learner with `k` equal to 2, 3 and 4 using the silhouette measure. Use insample resampling technique. What value of `k` would you choose based on the silhouette scores?

```
library(mlr3)
library(mlr3cluster)
set.seed(11)
learners = list(
  lrn("clust.kmeans", centers = 2L, id = "k-means, k=2"),
  lrn("clust.kmeans", centers = 3L, id = "k-means, k=3"),
  lrn("clust.kmeans", centers = 4L, id = "k-means, k=4")
)

task = as_task_clust(tsk("wine")$data()[, -1])
measure = msr("clust.silhouette")
bmr = benchmark(benchmark_grid(task, learners, rsmp("insample")))
bmr$aggregate(measure)[, c(4, 7)]
```

```
     learner_id clust.silhouette
1: k-means, k=2        0.6568537
```

```
2: k-means, k=3        0.5711382
3: k-means, k=4        0.5605941
```

Based on the silhouette score, we can choose `k = 2`.

5. Run a (spatially) unbiased classification benchmark experiment on the `ecuador` task with a featureless learner and xgboost, evaluate with the binary Brier score.

You can use any resampling method from `mlr3spatiotempcv`, in this solution we use 4-fold spatial environmental blocking.

```
library(mlr3verse)
library(mlr3spatial)
library(mlr3spatiotempcv)
```

```
Attaching package: 'mlr3spatiotempcv'

The following objects are masked from 'package:mlr3spatial':

    as_task_classif_st, as_task_classif_st.data.frame,
    as_task_classif_st.DataBackend, as_task_classif_st.sf,
    as_task_classif_st.TaskClassifST, as_task_regr_st,
    as_task_regr_st.data.frame, as_task_regr_st.DataBackend,
    as_task_regr_st.sf, as_task_regr_st.TaskClassifST,
    as_task_regr_st.TaskRegrST, TaskClassifST, TaskRegrST
```

```
set.seed(11)
learners = lrns(paste0("classif.", c("xgboost", "featureless")),
  predict_type = "prob")
rsmp_sp = rsmp("spcv_env", folds = 4)
design = benchmark_grid(tsk("ecuador"), learners, rsmp_sp)
bmr = benchmark(design)
bmr$aggregate(msr("classif.bbrier"))[, c(4, 7)]
```

```
            learner_id classif.bbrier
1:     classif.xgboost      0.2302815
2: classif.featureless      0.3838972
```

A.7 Solutions to Chapter 8

Parallel

Not all CPUs would be utilized in the example. All 4 of them are occupied for the first 4 iterations of the cross validation. The 5th iteration, however, only runs in parallel to the 6th fold, leaving 2 cores ilde. This is supported by the elapsed time of roughly 6 seconds for 6 jobs compared to also roughly 6 seconds for 8 jobs:

```
task = tsk("penguins")
learner = lrn("classif.debug", sleep_train = function() 3)

future::plan("multisession", workers = 4)

resampling = rsmp("cv", folds = 6)
system.time(resample(task, learner, resampling))

resampling = rsmp("cv", folds = 8)
system.time(resample(task, learner, resampling))
```

If possible, the number of resampling iterations should be an integer multiple of the number of workers. Therefore, a simple adaptation either increases the number of folds for improved accuracy of the error estimate or reduces the number of folds for improved runtime.

Custom Measures

The rules can easily be translated to R code where we expect `truth` and `prediction` to be factor vectors of the same length with levels `"A"` and `"B"`:

```
costsens = function(truth, prediction) {
    score = numeric(length(truth))
    score[truth == "A" & prediction == "B"] = 10
    score[truth == "B" & prediction == "A"] = 1

    mean(score)
}
```

This function can be embedded in the `Measure` class accordingly.

```
MeasureCustom = R6::R6Class("MeasureCustom",
  inherit = mlr3::MeasureClassif, # classification measure
  public = list(
    initialize = function() { # initialize class
      super$initialize(
        id = "custom", # unique ID
        packages = character(), # no dependencies
        properties = character(), # no special properties
        predict_type = "response", # measures response prediction
        range = c(0, Inf), # results in values between (0, 1)
        minimize = TRUE # smaller values are better
      )
    }
  ),

  private = list(
    .score = function(prediction, ...) { # define score as private method
      # define loss
      costsens = function(truth, prediction) {
```

```
      score = numeric(length(truth))
      score[truth == "A" & prediction == "B"] = 10
      score[truth == "B" & prediction == "A"] = 1

      mean(score)
    }

    # call loss function
    costsens(prediction$truth, prediction$response)
  }
 )
)
```

An alternative (as pointed to by the hint) can be constructed by first translating the rules to a matrix of misclassification costs, and then feeding this matrix to the constructor of the `mlr_measures_classif.costs` measure:

```
# truth in columns, prediction in rows
C = matrix(c(0, 10, 1, 0), nrow = 2)
rownames(C) = colnames(C) = c("A", "B")
print(C)
```

```
   A B
A  0 1
B 10 0
```

```
msr("classif.costs", costs = C)
```

```
<MeasureClassifCosts:classif.costs>: Cost-sensitive Classification
* Packages: mlr3
* Range: [0, Inf]
* Minimize: TRUE
* Average: macro
* Parameters: normalize=TRUE
* Properties: -
* Predict type: response
```

A.8 Solutions to Chapter 9

1. Prepare a `mlr3` regression task for `fifa` data. Select only variables describing the age and skills of footballers. Train any predictive model for this task, e.g. `regr.ranger`.

```
library("DALEX")
library("ggplot2")
data("fifa", package = "DALEX")
```

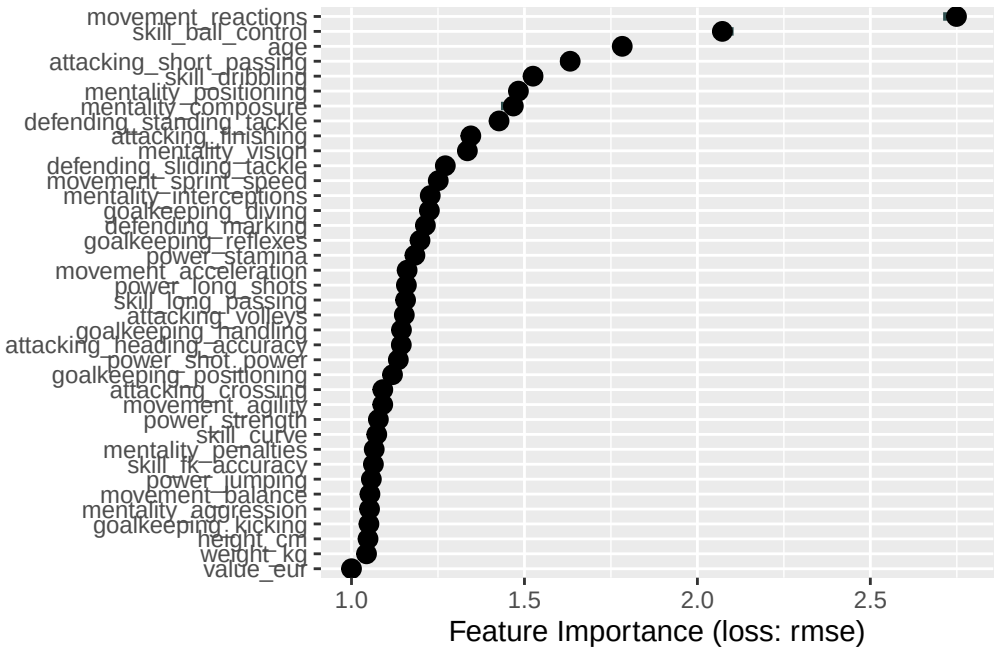
```
old_theme = set_theme_dalex("ema")

library("mlr3")
library("mlr3learners")
set.seed(1)

fifa20 <- fifa[,5:42]
task_fifa = as_task_regr(fifa20, target = "value_eur", id = "fifa20")

learner = lrn("regr.ranger")
learner$train(task_fifa)
learner$model
```

```
Ranger result

Call:
 ranger::ranger(dependent.variable.name = task$target_names, data = task$data(),      case.weights

Type:                             Regression
Number of trees:                  500
Sample size:                      5000
Number of independent variables:  37
Mtry:                             6
Target node size:                 5
Variable importance mode:         none
Splitrule:                        variance
OOB prediction error (MSE):       1.022805e+13
R squared (OOB):                  0.869943
```

2. Use the permutation importance method to calculate variable importance ranking. Which variable is the most important? Is it surprising?

With `iml`

```
library(iml)
model = Predictor$new(learner,
                      data = fifa20,
                      y = fifa$value_eur)

effect = FeatureImp$new(model,
                      loss = "rmse")
effect$plot()
```

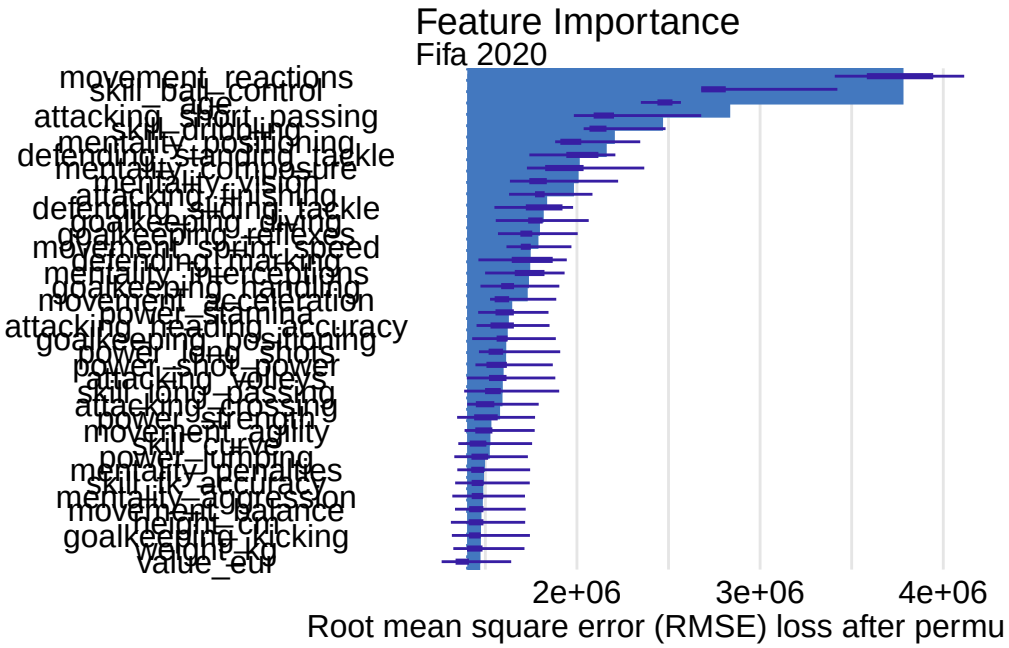


With DALEX

```
library("DALEX")
ranger_exp = DALEX::explain(learner,
  data = fifa20,
  y = fifa$value_eur,
  label = "Fifa 2020",
  verbose = FALSE)

ranger_effect = model_parts(ranger_exp, B = 5)
head(ranger_effect)
```

```
            variable mean_dropout_loss     label
1        _full_model_           1402526 Fifa 2020
2           value_eur           1402526 Fifa 2020
3           weight_kg           1471865 Fifa 2020
4 goalkeeping_kicking           1472795 Fifa 2020
5           height_cm           1474859 Fifa 2020
6    movement_balance           1475618 Fifa 2020
```

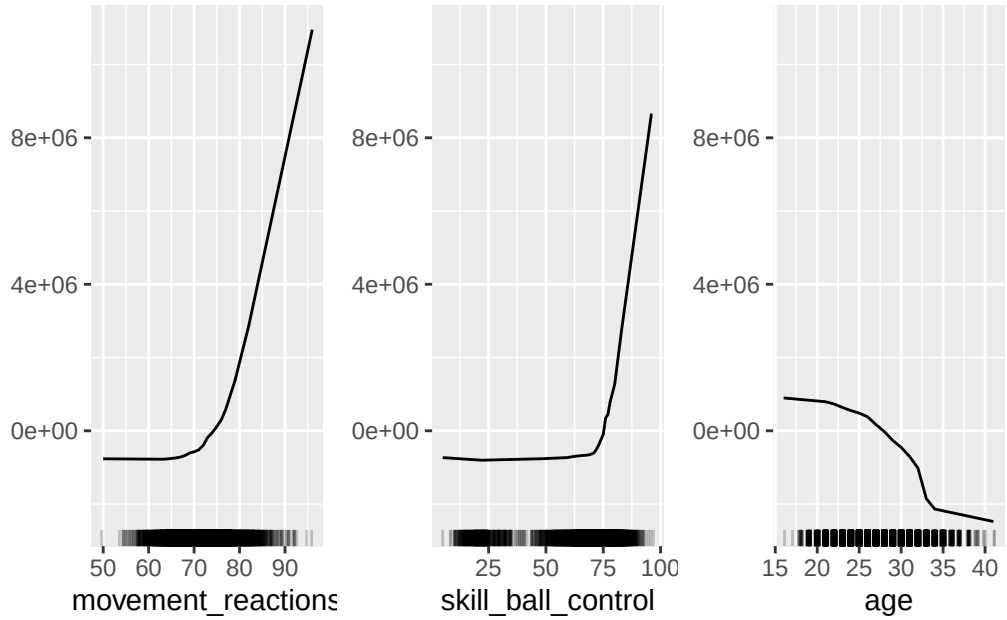
```
plot(ranger_effect)
```



3. Use the Partial Dependence profile to draw the global behavior of the model for this variable. Is it aligned with your expectations?

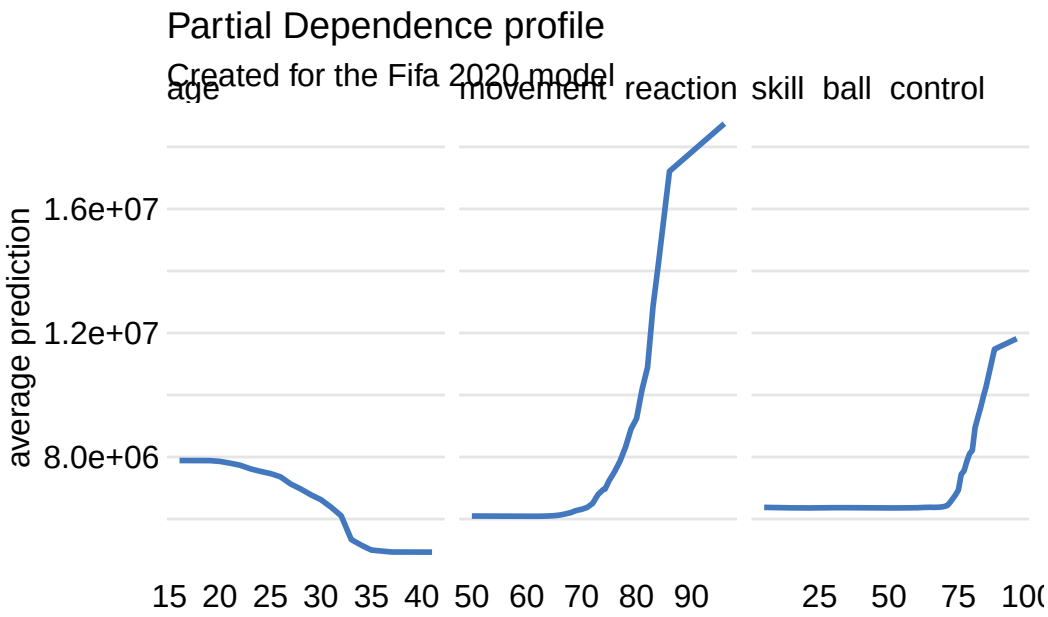
With `iml`

```
num_features = c("movement_reactions", "skill_ball_control", "age")

effect = FeatureEffects$new(model)
plot(effect, features = num_features)
```



With DALEX

```
num_features = c("movement_reactions", "skill_ball_control", "age")

ranger_profiles = model_profile(ranger_exp, variables = num_features)
plot(ranger_profiles)
```



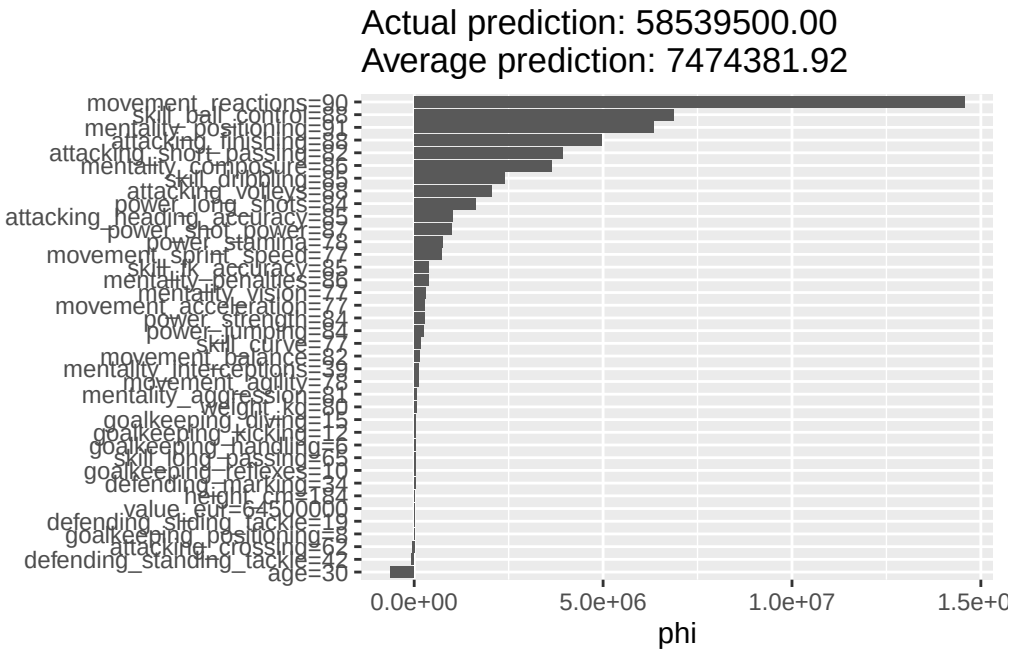
4 Choose one of the football players. You can choose some well-known striker (e.g. Robert Lewandowski) or a well-known goalkeeper (e.g. Manuel Neuer). The following tasks are worth repeating for several different choices.

```
player_1 <- fifa["R. Lewandowski", 5:42]
```

5. For the selected footballer, calculate and plot the Shapley values. Which variable is locally the most important and has the strongest influence on the valuation of the footballer?

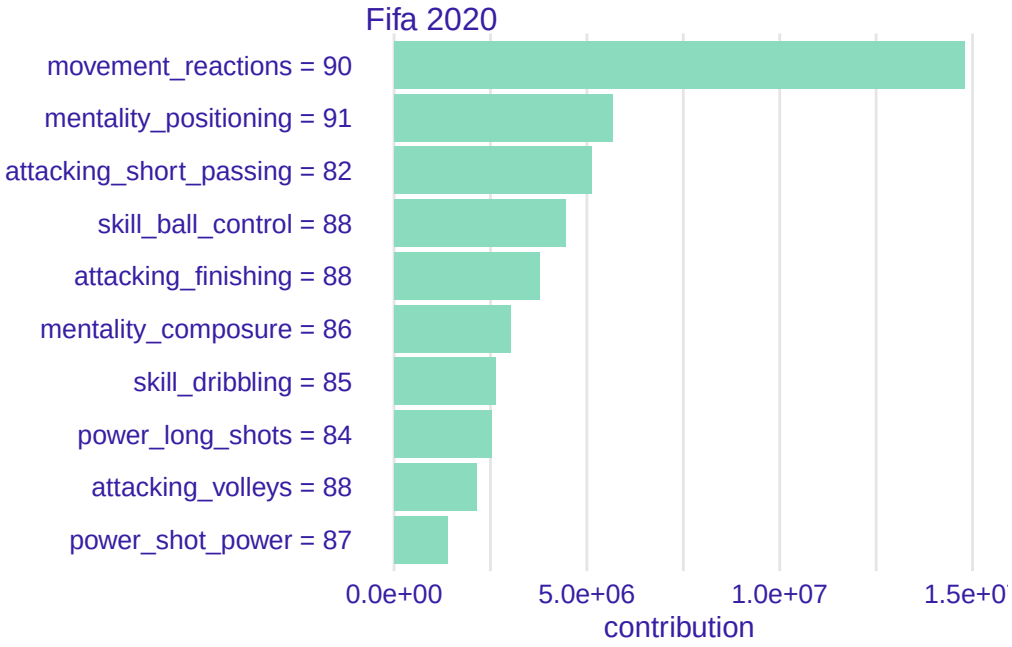
With `iml`

```
shapley = Shapley$new(model, x.interest = player_1)
plot(shapley)
```



With DALEX

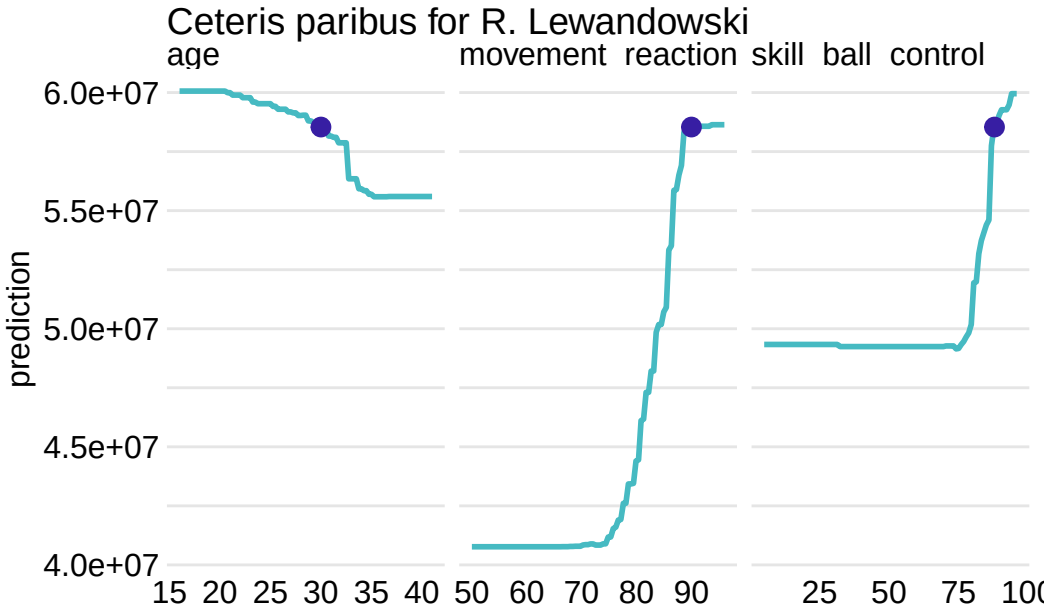
```
ranger_shap = predict_parts(ranger_exp,
              new_observation = player_1,
              type = "shap", B = 1)
plot(ranger_shap, show_boxplots = FALSE)
```



6. For the selected footballer, calculate the Ceteris Paribus / Individual Conditional Expectation profiles to draw the local behavior of the model for this variable. Is it different from the global behavior?

With DALEX

```
num_features = c("movement_reactions", "skill_ball_control", "age")

ranger_ceteris = predict_profile(ranger_exp, player_1)
plot(ranger_ceteris, variables = num_features) +
  ggtitle("Ceteris paribus for R. Lewandowski", " ")
```


Ceteris paribus for R. Lewandowski
age
movement
reaction
skill
ball
control
prediction
6.0e+07
5.5e+07
5.0e+07
4.5e+07
4.0e+07
15 20 25 30 35 40
50 60 70 80 90
25 50 75 100

B

Citation information

Every package in the mlr3verse has its own citation details that can be found on the respective GitHub repository.

To reference the whole book please use:

```
Bischl B, Sonabend R, Kotthoff L, Lang M. 2023.
"Flexible and Robust Machine Learning Using mlr3 in R". https://mlr3book.mlr-org.com.

@book{Bischl2023
    title = Flexible and Robust Machine Learning Using mlr3 in R
    editor = {Bernd Bischl, Raphael Sonabend, Lars Kotthoff, Michel Lang},
    url = {https://mlr3book.mlr-org.com},
    year = 2023
}
```

When possible please always reference specific chapters to ensure our authors receive appropriate credit. To reference a specific chapter in the book please use:

```
Chapter Author Surname(s) and Initial(s). 2023.
"Title of Chapter", in Bischl B, Sonabend R, Kotthoff L, Lang M, (ed[s])
Flexible and Robust Machine Learning Using mlr3 in R.
https://mlr3book.mlr-org.com.

@incollection{Bischl2023ChapterN
    author = {Chapter Author First Name(s) and Last Name(s).},
    booktitle = Flexible and Robust Machine Learning Using mlr3 in R,
    publisher = "",
    year = {2023}
    editor = {Bernd Bischl, Raphael Sonabend, Lars Kotthoff, Michel Lang},
    chapter = {Title of Chapter},
    pages = "",
    note = {https://mlr3book.mlr-org.com}
}
```

To reference the mlr3 package, please cite our JOSS paper:

```
Lang M, Binder M, Richter J, Schratz P, Pfisterer F, Coors S, Au Q,
Casalicchio G, Kotthoff L, Bischl B (2019). "mlr3: A modern object-oriented
machine learning framework in R." Journal of Open Source Software.
doi: 10.21105/joss.01903.

@Article{mlr3,
  title = {{mlr3}: A modern object-oriented machine learning framework in {R}},
  author = {Michel Lang and Martin Binder and Jakob Richter and Patrick Schratz and
```

```
  Florian Pfisterer and Stefan Coors and Quay Au and Giuseppe Casalicchio and
  Lars Kotthoff and Bernd Bischl},
  journal = {Journal of Open Source Software},
  year = {2019},
  month = {dec},
  doi = {10.21105/joss.01903},
  url = {https://joss.theoj.org/papers/10.21105/joss.01903},
}
```

C

Tasks

The key features of the tasks that we use throughout the book are explained below as well as a plot of the target variable(s).

C.1 Regression Tasks

C.1.1 `mtcars`

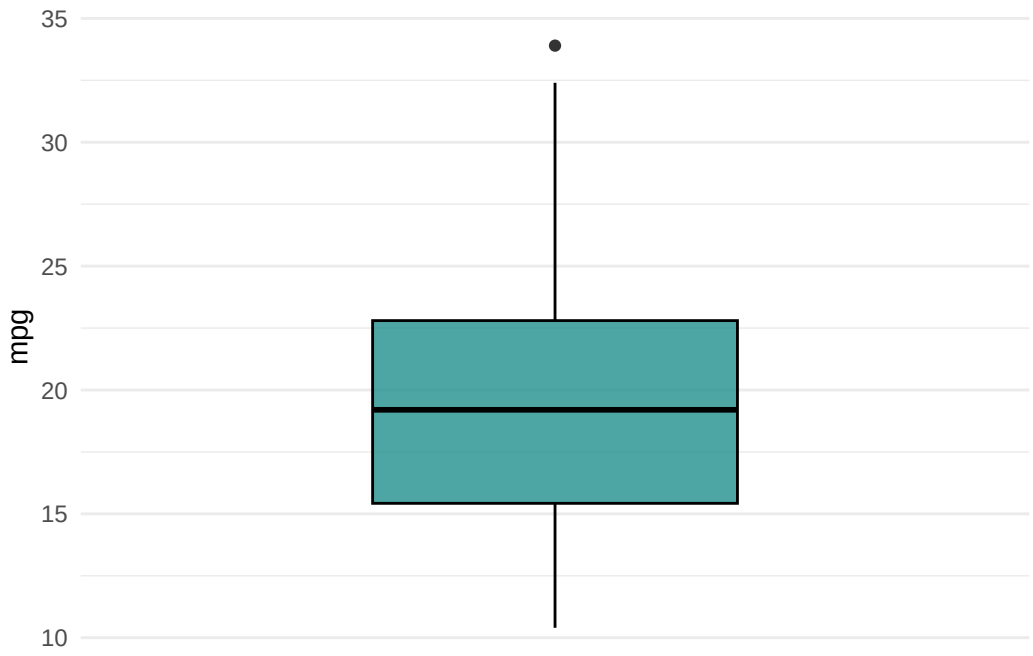
```
tsk("mtcars")
```

```
<TaskRegr:mtcars> (32 x 11): Motor Trends
* Target: mpg
* Properties: -
* Features (10):
  - dbl (10): am, carb, cyl, disp, drat, gear, hp, qsec, vs, wt
```

```
tsk("mtcars")$head()
```

```
    mpg am carb cyl disp drat gear  hp  qsec vs    wt
1: 21.0  1    4   6  160 3.90    4 110 16.46  0 2.620
2: 21.0  1    4   6  160 3.90    4 110 17.02  0 2.875
3: 22.8  1    1   4  108 3.85    4  93 18.61  1 2.320
4: 21.4  0    1   6  258 3.08    3 110 19.44  1 3.215
5: 18.7  0    2   8  360 3.15    3 175 17.02  0 3.440
6: 18.1  0    1   6  225 2.76    3 105 20.22  1 3.460
```

```
autoplot(tsk("mtcars"))
```



See more at `?mlr_tasks_mtcars`.

C.2 Classification Tasks

C.2.1 german_credit

```
tsk("german_credit")
```

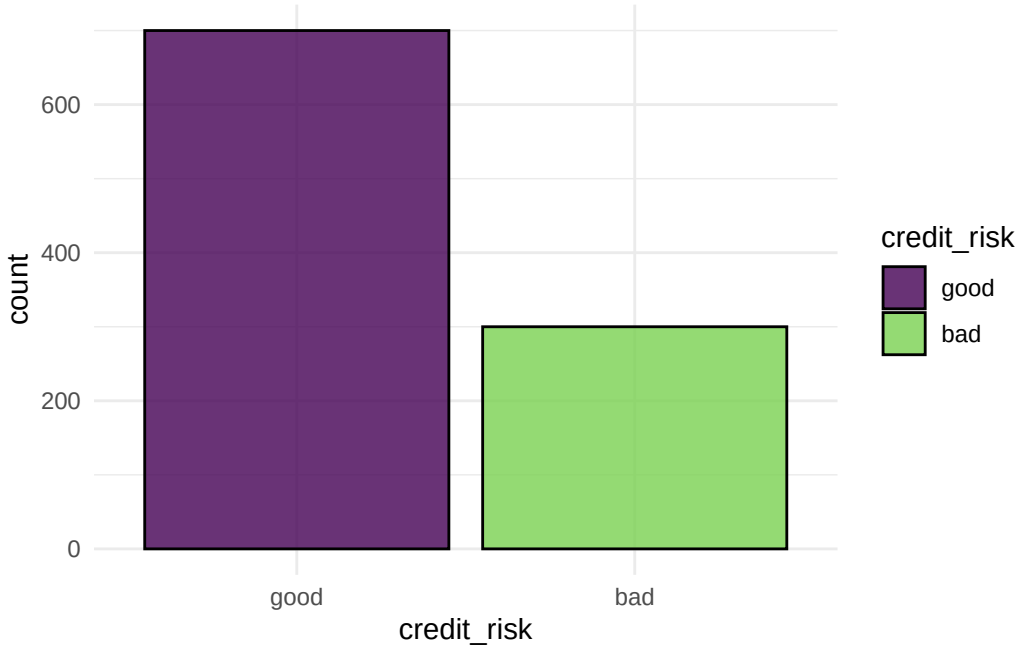
```
<TaskClassif:german_credit> (1000 x 21): German Credit
* Target: credit_risk
* Properties: twoclass
* Features (20):
  - fct (14): credit_history, employment_duration, foreign_worker,
    housing, job, other_debtors, other_installment_plans,
    people_liable, personal_status_sex, property, purpose, savings,
    status, telephone
  - int (3): age, amount, duration
  - ord (3): installment_rate, number_credits, present_residence
```

```
tsk("german_credit")$head()
```

```
   credit_risk age amount                            credit_history duration
1:        good  67   1169     all credits at this bank paid back duly        6
2:         bad  22   5951 no credits taken/all credits paid back duly       48
3:        good  49   2096     all credits at this bank paid back duly       12
```

```
4:        good  45   7882 no credits taken/all credits paid back duly       42
5:         bad  53   4870    existing credits paid back duly till now       24
6:        good  35   9055 no credits taken/all credits paid back duly       36
16 variables not shown: [employment_duration, foreign_worker, housing, installment_rate, job, numb
```

```
autoplot(tsk("german_credit"))
```



See more at `?mlr_tasks_german_credit`.

C.2.2 penguins

```
tsk("penguins")
```

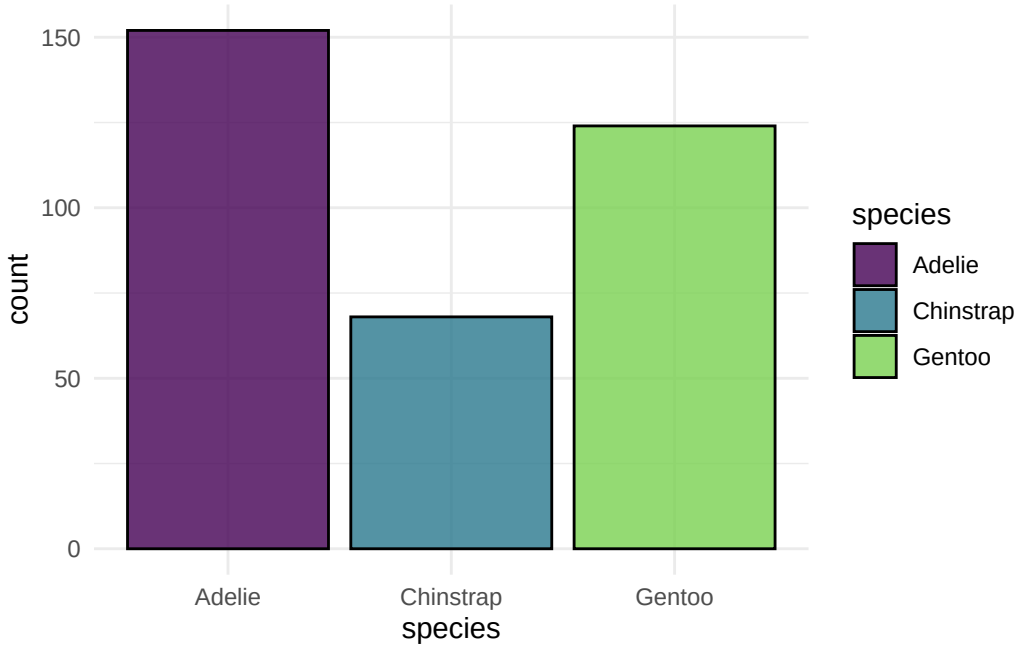
```
<TaskClassif:penguins> (344 x 8): Palmer Penguins
* Target: species
* Properties: multiclass
* Features (7):
  - int (3): body_mass, flipper_length, year
  - dbl (2): bill_depth, bill_length
  - fct (2): island, sex
```

```
tsk("penguins")$head()
```

```
   species bill_depth bill_length body_mass flipper_length    island    sex
1:  Adelie       18.7        39.1      3750            181 Torgersen   male
2:  Adelie       17.4        39.5      3800            186 Torgersen female
```

```
3:  Adelie       18.0        40.3      3250            195 Torgersen female
4:  Adelie         NA          NA        NA             NA Torgersen   <NA>
5:  Adelie       19.3        36.7      3450            193 Torgersen female
6:  Adelie       20.6        39.3      3650            190 Torgersen   male
1 variable not shown: [year]
```

```
autoplot(tsk("penguins"))
```



See more at `?mlr_tasks_penguins`.

C.2.3 penguins_simple

```
tsk("penguins_simple")
```

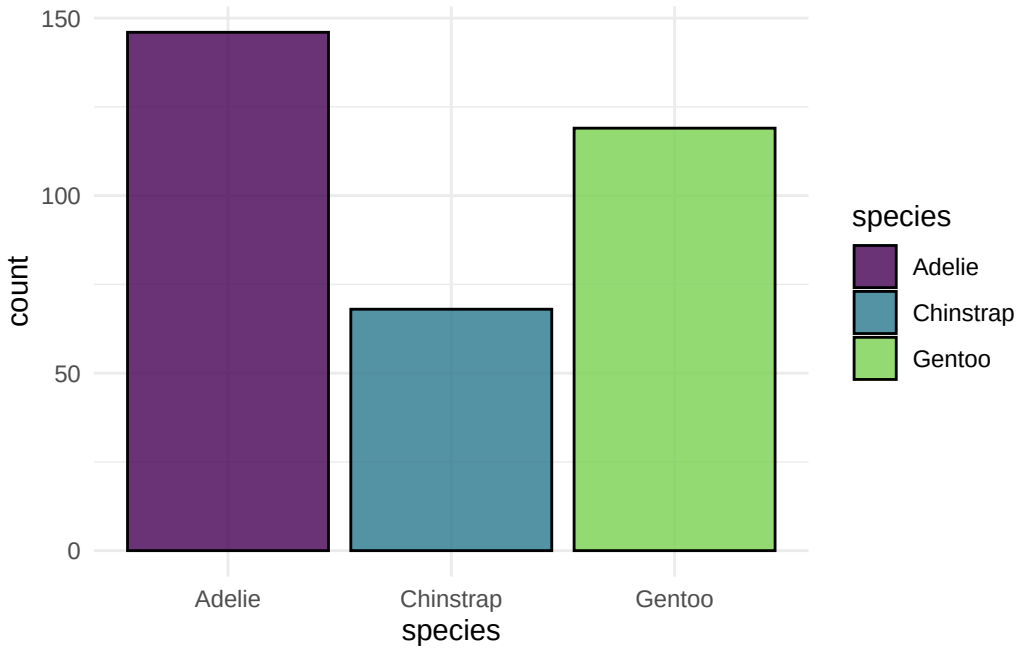
```
<TaskClassif:penguins> (333 x 11): Simplified Palmer Penguins
* Target: species
* Properties: multiclass
* Features (10):
  - dbl (7): bill_depth, bill_length, island.Biscoe, island.Dream,
    island.Torgersen, sex.female, sex.male
  - int (3): body_mass, flipper_length, year
```

```
tsk("penguins_simple")$head()
```

```
   species bill_depth bill_length body_mass flipper_length island.Biscoe
1:  Adelie       18.7        39.1      3750            181             0
```

```
2:  Adelie       17.4        39.5       3800            186            0
3:  Adelie       18.0        40.3       3250            195            0
4:  Adelie       19.3        36.7       3450            193            0
5:  Adelie       20.6        39.3       3650            190            0
6:  Adelie       17.8        38.9       3625            181            0
5 variables not shown: [island.Dream, island.Torgersen, sex.female, sex.male, year]
```

```
autoplot(tsk("penguins_simple"))
```



See more at `?mlr3data::mlr_tasks_penguins_simple`.

C.2.4 sonar

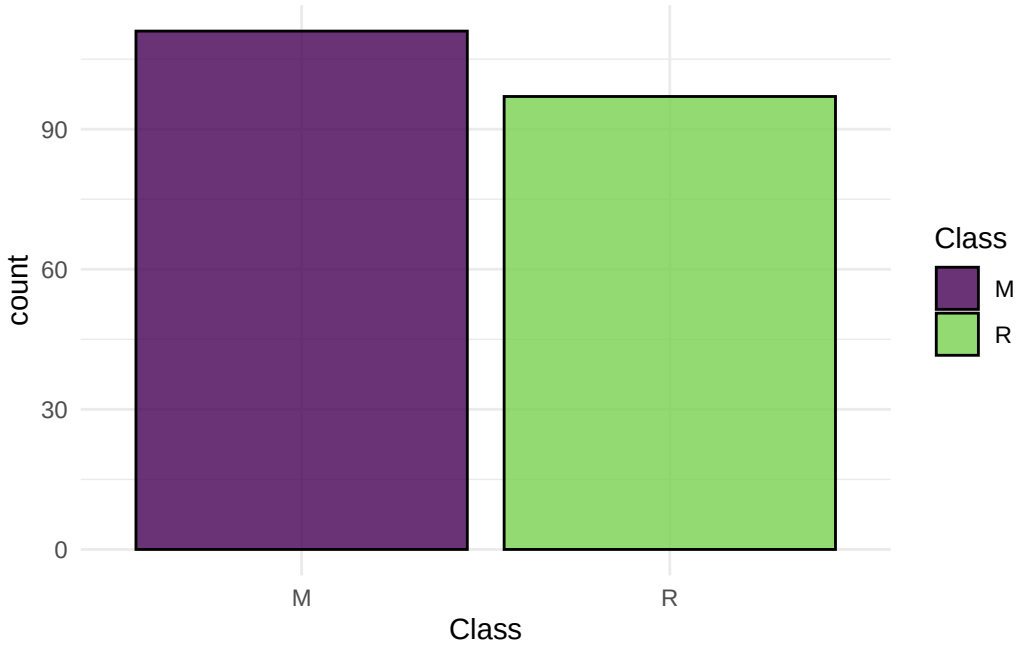
```
tsk("sonar")
```

```
<TaskClassif:sonar> (208 x 61): Sonar: Mines vs. Rocks
* Target: Class
* Properties: twoclass
* Features (60):
  - dbl (60): V1, V10, V11, V12, V13, V14, V15, V16, V17, V18, V19, V2,
    V20, V21, V22, V23, V24, V25, V26, V27, V28, V29, V3, V30, V31,
    V32, V33, V34, V35, V36, V37, V38, V39, V4, V40, V41, V42, V43,
    V44, V45, V46, V47, V48, V49, V5, V50, V51, V52, V53, V54, V55,
    V56, V57, V58, V59, V6, V60, V7, V8, V9
```

```
tsk("sonar")$head()
```

```
   Class     V1    V10    V11    V12    V13    V14    V15    V16    V17    V18
1:     R 0.0200 0.2111 0.1609 0.1582 0.2238 0.0645 0.0660 0.2273 0.3100 0.2999
2:     R 0.0453 0.2872 0.4918 0.6552 0.6919 0.7797 0.7464 0.9444 1.0000 0.8874
3:     R 0.0262 0.6194 0.6333 0.7060 0.5544 0.5320 0.6479 0.6931 0.6759 0.7551
4:     R 0.0100 0.1264 0.0881 0.1992 0.0184 0.2261 0.1729 0.2131 0.0693 0.2281
5:     R 0.0762 0.4459 0.4152 0.3952 0.4256 0.4135 0.4528 0.5326 0.7306 0.6193
6:     R 0.0286 0.3039 0.2988 0.4250 0.6343 0.8198 1.0000 0.9988 0.9508 0.9025
50 variables not shown: [V19, V2, V20, V21, V22, V23, V24, V25, V26, V27, ...]
```

```
1  autoplot(tsk("sonar"))
```



See more at `?mlr_tasks_sonar`.

C.2.5 `spam`

```
1  tsk("spam")
```

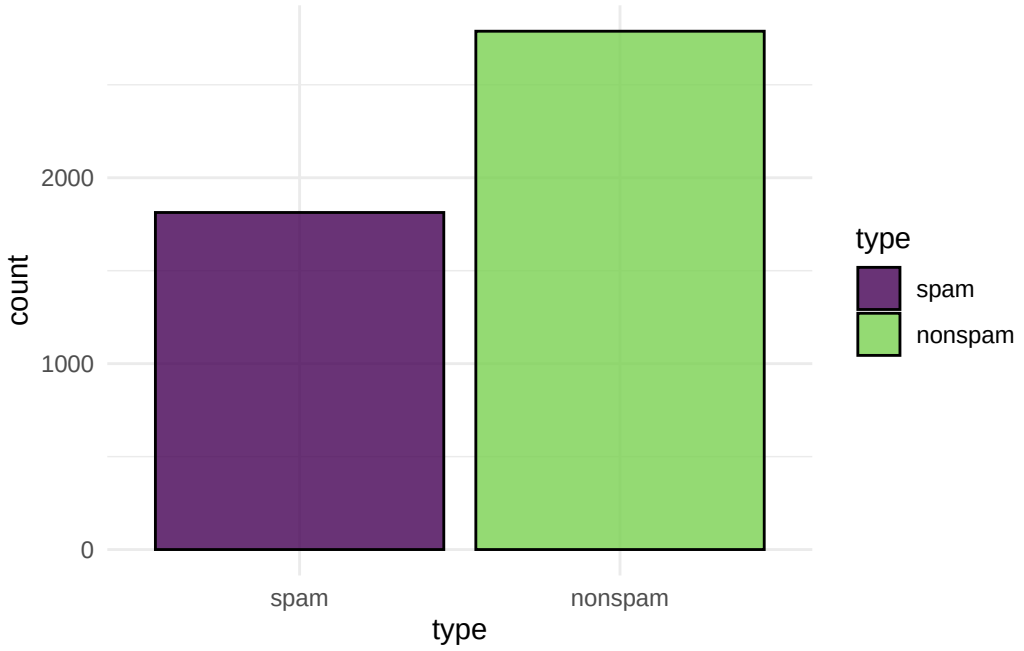
```
<TaskClassif:spam> (4601 x 58): HP Spam Detection
* Target: type
* Properties: twoclass
* Features (57):
  - dbl (57): address, addresses, all, business, capitalAve,
    capitalLong, capitalTotal, charDollar, charExclamation, charHash,
    charRoundbracket, charSemicolon, charSquarebracket, conference,
    credit, cs, data, direct, edu, email, font, free, george, hp, hpl,
    internet, lab, labs, mail, make, meeting, money, num000, num1999,
    num3d, num415, num650, num85, num857, order, original, our, over,
```

```
    parts, people, pm, project, re, receive, remove, report, table,
    technology, telnet, will, you, your
```

```
tsk("spam")$head()
```

```
   type address addresses  all business capitalAve capitalLong capitalTotal
1: spam    0.64      0.00 0.64     0.00      3.756          61          278
2: spam    0.28      0.14 0.50     0.07      5.114         101         1028
3: spam    0.00      1.75 0.71     0.06      9.821         485         2259
4: spam    0.00      0.00 0.00     0.00      3.537          40          191
5: spam    0.00      0.00 0.00     0.00      3.537          40          191
6: spam    0.00      0.00 0.00     0.00      3.000          15           54
50 variables not shown: [charDollar, charExclamation, charHash, charRoundbracket, charSemicolon, c
```

```
autoplot(tsk("spam"))
```



See more at `?mlr_tasks_spam`.

C.3 Survival Tasks

C.3.1 `rats`

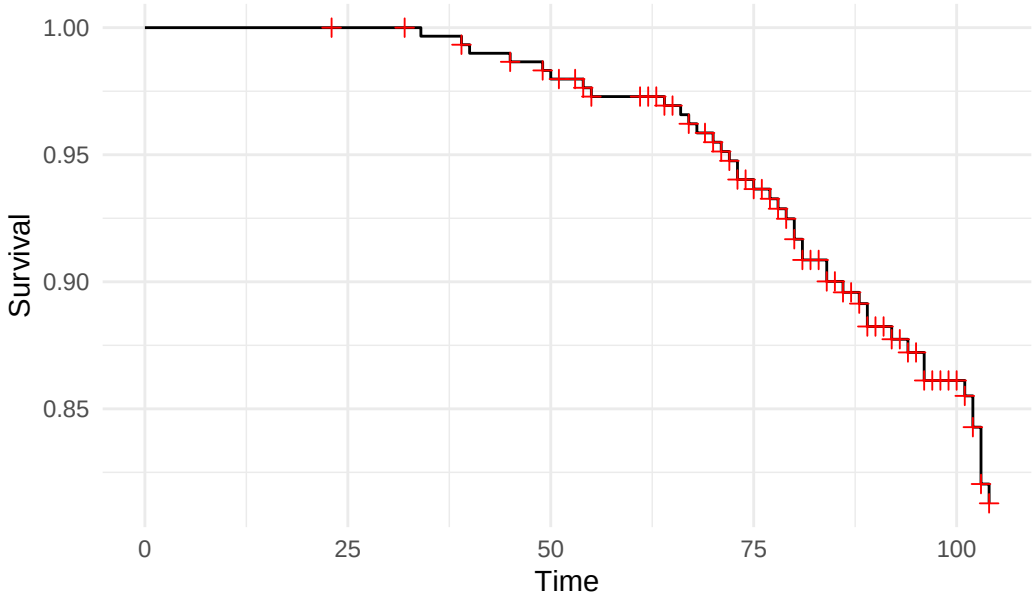
```
tsk("rats")
```

```
<TaskSurv:rats> (300 x 5): Rats
* Target: time, status
* Properties: -
* Features (3):
  - int (2): litter, rx
  - fct (1): sex
```

```
tsk("rats")$head()
```

```
   time status litter rx sex
1:  101      0      1  1   f
2:   49      1      1  0   f
3:  104      0      1  0   f
4:   91      0      2  1   m
5:  104      0      2  0   m
6:  102      0      2  0   m
```

```
autoplot(tsk("rats"))
```



See more at `?mlr3proba::mlr_tasks_rats`.

C.4 Density Tasks

C.4.1 `precip`

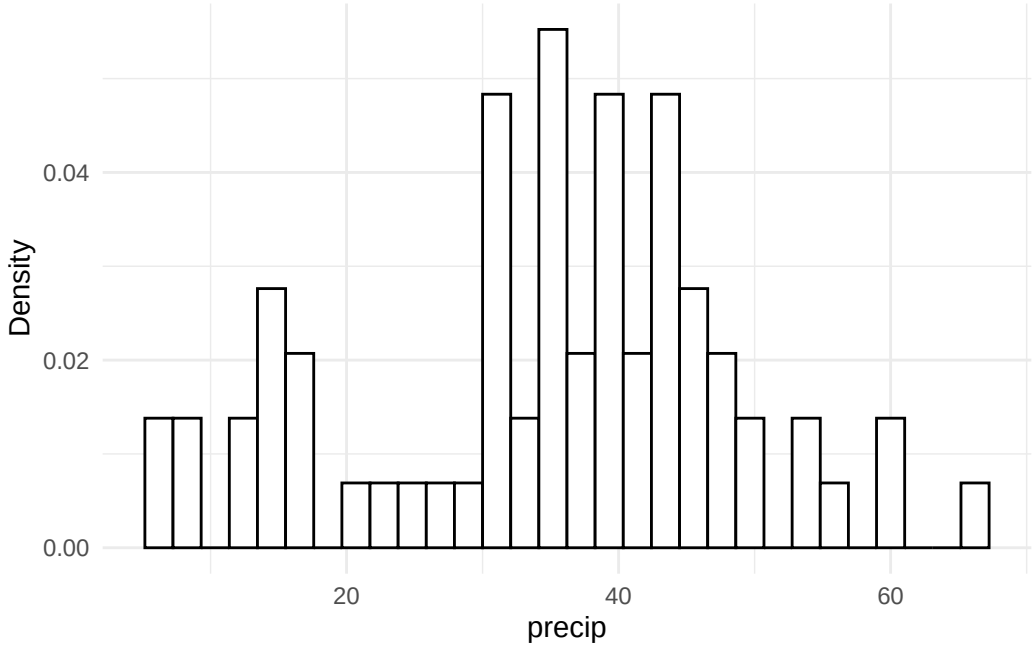
```
tsk("precip")
```

```
<TaskDens:precip> (70 x 1): Annual Precipitation
* Target: -
* Properties: -
* Features (1):
  - dbl (1): precip
```

```
tsk("precip")$head()
```

```
   precip
1:   67.0
2:   54.7
3:    7.0
4:   48.5
5:   14.0
6:   17.2
```

```
autoplot(tsk("precip"))
```



See more at `?mlr3proba::mlr_tasks_precip`.

C.5 Spatiotemporal Tasks

C.5.1 `ecuador`

```
tsk("ecuador")
```

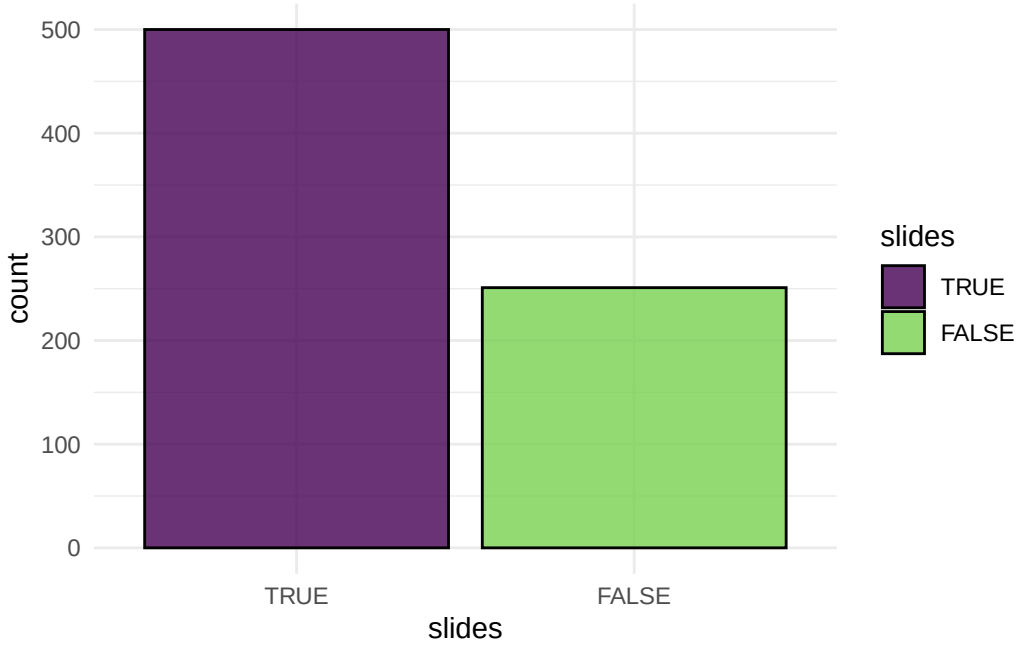
```
<TaskClassifST:ecuador> (751 x 11): Ecuador landslides
* Target: slides
* Properties: twoclass
* Features (10):
  - dbl (10): carea, cslope, dem, distdeforest, distroad,
    distslidespast, hcurv, log.carea, slope, vcurv
* Coordinates:
           x       y
  1: 712882.5 9560002
  2: 715232.5 9559582
  3: 715392.5 9560172
  4: 715042.5 9559312
  5: 715382.5 9560142
 ---
747: 714472.5 9558482
748: 713142.5 9560992
749: 713322.5 9560562
```

```
750: 715392.5 9557932
751: 713802.5 9560862
```

```
tsk("ecuador")$head()
```

```
   slides       carea   cslope     dem distdeforest distroad distslidespast
1:   TRUE   5577.3916 34.42789 1911.52        15.00      300              9
2:   TRUE   1399.2329 30.71569 2198.66       300.00      300             21
3:   TRUE 351155.1250 32.81444 1988.71       300.00      300             40
4:   TRUE    500.5027 33.90592 2320.49       300.00      300            100
5:   TRUE    671.1807 41.60017 2021.07       300.00      300             21
6:   TRUE    634.3320 30.29457 1838.40         9.15      300              2
4 variables not shown: [hcurv, log.carea, slope, vcurv]
```

```
autoplot(tsk("ecuador"))
```



See more at `?mlr3spatiotempcv::mlr_tasks_ecuador`.

C.6 Clustering Tasks

C.6.1 `usarrests`

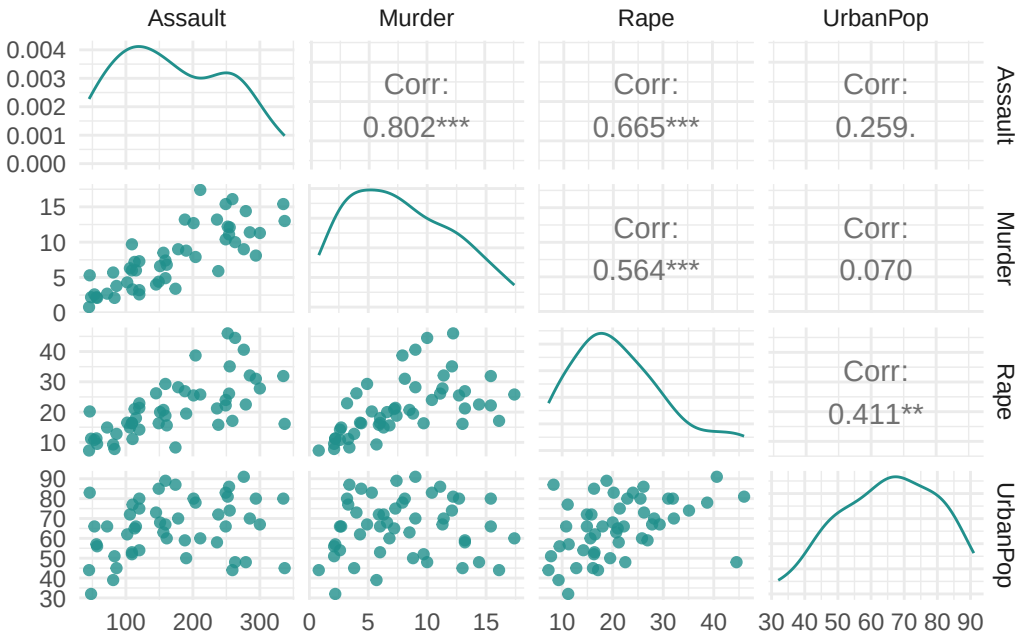
```
tsk("usarrests")
```

```
<TaskClust:usarrests> (50 x 4): US Arrests
* Target: -
* Properties: -
* Features (4):
  - int (2): Assault, UrbanPop
  - dbl (2): Murder, Rape
```

```
tsk("usarrests")$head()
```

```
   Assault Murder Rape UrbanPop
1:     236   13.2 21.2       58
2:     263   10.0 44.5       48
3:     294    8.1 31.0       80
4:     190    8.8 19.5       50
5:     276    9.0 40.6       91
6:     204    7.9 38.7       78
```

```
autoplot(tsk("usarrests"))
```



See more at `?mlr3cluster::mlr_tasks_usarrests`.

D

Overview Tables

Our homepage provides overviews and tables of the following objects:

Description	Link
Packages overview	https://mlr-org.com/packages.html
Task overview	https://mlr-org.com/tasks.html
Learner overview	https://mlr-org.com/learners.html
Resampling overview	https://mlr-org.com/resamplings.html
Measure overview	https://mlr-org.com/measures.html
PipeOp overview	https://mlr-org.com/pipeops.html
Graph overview	https://mlr-org.com/graphs.html
Tuner overview	https://mlr-org.com/tuners.html
Terminator overview	https://mlr-org.com/terminators.html
Tuning space overview	https://mlr-org.com/tuning_spaces.html
Filter overview	https://mlr-org.com/filters.html
FSelector overview	https://mlr-org.com/fselectors.html

```
library(mlr3verse)
library(mlr3proba)
library(mlr3spatiotempcv)
library(mlr3spatial)
library(mlr3extralearners)
library(mlr3hyperband)
library(mlr3mbo)
```

```
mlr_tasks
```

```
<DictionaryTask> with 33 stored values
Keys: actg, bike_sharing, boston_housing, breast_cancer, cookfarm_mlr3,
  diplodia, ecuador, faithful, gbcs, german_credit, grace, ilpd, iris,
  kc_housing, leipzig, lung, moneyball, mtcars, optdigits, penguins,
  penguins_simple, pima, precip, rats, ruspini, sonar, spam, titanic,
  unemployment, usarrests, whas, wine, zoo
```

```
mlr_learners
```

```
<DictionaryLearner> with 145 stored values
Keys: classif.abess, classif.AdaBoostM1, classif.bart, classif.C50,
  classif.catboost, classif.cforest, classif.ctree, classif.cv_glmnet,
```

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

```
1  mlr_resamplings
```

```
<DictionaryResampling> with 24 stored values
Keys: bootstrap, custom, custom_cv, cv, holdout, insample, loo,
  repeated_cv, repeated_spcv_block, repeated_spcv_coords,
  repeated_spcv_disc, repeated_spcv_env, repeated_spcv_tiles,
  repeated_sptcv_cluto, repeated_sptcv_cstf, spcv_block, spcv_buffer,
  spcv_coords, spcv_disc, spcv_env, spcv_tiles, sptcv_cluto,
  sptcv_cstf, subsampling
```

```
1  mlr_measures
```

```
<DictionaryMeasure> with 92 stored values
Keys: aic, bic, classif.acc, classif.auc, classif.bacc, classif.bbrier,
  classif.ce, classif.costs, classif.dor, classif.fbeta, classif.fdr,
  classif.fn, classif.fnr, classif.fomr, classif.fp, classif.fpr,
  classif.logloss, classif.mauc_au1p, classif.mauc_au1u,
  classif.mauc_aunp, classif.mauc_aunu, classif.mbrier, classif.mcc,
```

```
  classif.npv, classif.ppv, classif.prauc, classif.precision,
  classif.recall, classif.sensitivity, classif.specificity, classif.tn,
  classif.tnr, classif.tp, classif.tpr, clust.ch, clust.dunn,
  clust.silhouette, clust.wss, debug_classif, dens.logloss, oob_error,
  regr.bias, regr.ktau, regr.logloss, regr.mae, regr.mape, regr.maxae,
  regr.medae, regr.medse, regr.mse, regr.msle, regr.pbias, regr.rae,
  regr.rmse, regr.rmsle, regr.rrse, regr.rse, regr.rsq, regr.sae,
  regr.smape, regr.srho, regr.sse, selected_features, sim.jaccard,
  sim.phi, surv.brier, surv.calib_alpha, surv.calib_beta,
  surv.chambless_auc, surv.cindex, surv.dcalib, surv.graf,
  surv.hung_auc, surv.intlogloss, surv.logloss, surv.mae, surv.mse,
  surv.nagelk_r2, surv.oquigley_r2, surv.rcll, surv.rmse, surv.schmid,
  surv.song_auc, surv.song_tnr, surv.song_tpr, surv.uno_auc,
  surv.uno_tnr, surv.uno_tpr, surv.xu_r2, time_both, time_predict,
  time_train
```

```
1 mlr_pipeops
```

```
<DictionaryPipeOp> with 74 stored values
Keys: boxcox, branch, chunk, classbalancing, classifavg, classweights,
  colapply, collapsefactors, colroles, compose_crank, compose_distr,
  compose_probregr, copy, crankcompose, datefeatures, distrcompose,
  encode, encodeimpact, encodelmer, featureunion, filter, fixfactors,
  histbin, ica, imputeconstant, imputehist, imputelearner, imputemean,
  imputemedian, imputemode, imputeoor, imputesample, kernelpca,
  learner, learner_cv, missind, modelmatrix, multiplicityexply,
  multiplicityimply, mutate, nmf, nop, ovrsplit, ovrunite, pca, proxy,
  quantilebin, randomprojection, randomresponse, regravg,
  removeconstants, renamecolumns, replicate, scale, scalemaxabs,
  scalerange, select, smote, spatialsign, subsample, survavg,
  targetinvert, targetmutate, targettrafoscalerange, textvectorizer,
  threshold, trafopred_regrsurv, trafopred_survregr,
  trafotask_regrsurv, trafotask_survregr, tunethreshold, unbranch,
  vtreat, yeojohnson
```

```
1 mlr_graphs
```

```
<DictionaryGraph> with 13 stored values
Keys: bagging, branch, crankcompositor, distrcompositor, greplicate,
  ovr, probregr, robustify, stacking, survaverager, survbagging,
  survtoregr, targettrafo
```

```
1 mlr_tuners
```

```
<DictionaryTuner> with 10 stored values
Keys: cmaes, design_points, gensa, grid_search, hyperband, irace, mbo,
  nloptr, random_search, successive_halving
```

```
1 mlr_terminators
```

```
<DictionaryTerminator> with 8 stored values
Keys: clock_time, combo, evals, none, perf_reached, run_time,
  stagnation, stagnation_batch
```

```
mlr_tuning_spaces
```

```
<DictionaryTuningSpaces> with 36 stored values
Keys: classif.glmnet.default, classif.glmnet.rbv1, classif.glmnet.rbv2,
  classif.kknn.default, classif.kknn.rbv1, classif.kknn.rbv2,
  classif.ranger.default, classif.ranger.rbv1, classif.ranger.rbv2,
  classif.rpart.default, classif.rpart.rbv1, classif.rpart.rbv2,
  classif.svm.default, classif.svm.rbv1, classif.svm.rbv2,
  classif.xgboost.default, classif.xgboost.rbv1, classif.xgboost.rbv2,
  regr.glmnet.default, regr.glmnet.rbv1, regr.glmnet.rbv2,
  regr.kknn.default, regr.kknn.rbv1, regr.kknn.rbv2,
  regr.ranger.default, regr.ranger.rbv1, regr.ranger.rbv2,
  regr.rpart.default, regr.rpart.rbv1, regr.rpart.rbv2,
  regr.svm.default, regr.svm.rbv1, regr.svm.rbv2, regr.xgboost.default,
  regr.xgboost.rbv1, regr.xgboost.rbv2
```

```
mlr_filters
```

```
<DictionaryFilter> with 21 stored values
Keys: anova, auc, carscore, carsurvscore, cmim, correlation, disr,
  find_correlation, importance, information_gain, jmi, jmim,
  kruskal_test, mim, mrmr, njmim, performance, permutation, relief,
  selected_features, variance
```

```
mlr_fselectors
```

```
<DictionaryFSelector> with 8 stored values
Keys: design_points, exhaustive_search, genetic_search, random_search,
  rfe, rfecv, sequential, shadow_variable_search
```

E

Session Info

If you would like to reproduce the results in this book, note the seed set at the top of each chapter (which is the same as the chapter number) and the `sessionInfo` at the time of publication:

```
R version 4.3.0 (2023-04-21)
Platform: x86_64-pc-linux-gnu (64-bit)
Running under: Ubuntu 22.04.2 LTS

Matrix products: default
BLAS:   /usr/lib/x86_64-linux-gnu/openblas-pthread/libblas.so.3
LAPACK: /usr/lib/x86_64-linux-gnu/openblas-pthread/libopenblasp-r0.3.20.so;  LAPACK version 3.10.0

locale:
 [1] LC_CTYPE=C.UTF-8       LC_NUMERIC=C           LC_TIME=C.UTF-8
 [4] LC_COLLATE=C.UTF-8     LC_MONETARY=C.UTF-8    LC_MESSAGES=C.UTF-8
 [7] LC_PAPER=C.UTF-8       LC_NAME=C              LC_ADDRESS=C
[10] LC_TELEPHONE=C         LC_MEASUREMENT=C.UTF-8 LC_IDENTIFICATION=C

time zone: UTC
tzcode source: system (glibc)

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

loaded via a namespace (and not attached):
 [1] compiler_4.3.0  fastmap_1.1.1   cli_3.6.1       tools_4.3.0
 [5] htmltools_0.5.5 rmarkdown_2.21  knitr_1.42      jsonlite_1.8.4
 [9] xfun_0.39       digest_0.6.31   rlang_1.1.0     evaluate_0.20
```

E

References

Baniecki, Hubert, and Przemyslaw Biecek. 2019. "modelStudio: Interactive Studio with Explanations for ML Predictive Models." *Journal of Open Source Software* 4 (43): 1798. https://doi.org/10.21105/joss.01798.

Baniecki, Hubert, Wojciech Kretowicz, Piotr Piątyszek, Jakub Wiśniewski, and Przemysław Biecek. 2021. "dalex: Responsible Machine Learning with Interactive Explainability and Fairness in Python." *Journal of Machine Learning Research* 22 (214): 1–7. http://jmlr.org/papers/v22/20-1473.html.

Bengio, Yoshua, and Yves Grandvalet. 2003. "No Unbiased Estimator of the Variance of k-Fold Cross-Validation." *Advances in Neural Information Processing Systems* 16.

Bergstra, James, and Yoshua Bengio. 2012. "Random Search for Hyper-Parameter Optimization." *Journal of Machine Learning Research* 13 (10): 281–305. http://jmlr.org/papers/v13/bergstra12a.html.

Biecek, Przemyslaw. 2018. "DALEX: Explainers for complex predictive models in R." *Journal of Machine Learning Research* 19 (84): 1–5. http://jmlr.org/papers/v19/18-416.html.

Biecek, Przemyslaw, and Tomasz Burzykowski. 2021. *Explanatory Model Analysis*. Chapman; Hall/CRC, New York. https://ema.drwhy.ai/.

Binder, Martin, Florian Pfisterer, and Bernd Bischl. 2020. "Collecting Empirical Data about Hyperparameters for Data Driven AutoML." In *AutoML Workshop at ICML 2020*. https://www.automl.org/wp-content/uploads/2020/07/AutoML_2020_paper_63.pdf.

Binder, Martin, Florian Pfisterer, Michel Lang, Lennart Schneider, Lars Kotthoff, and Bernd Bischl. 2021. "mlr3pipelines - Flexible Machine Learning Pipelines in R." *Journal of Machine Learning Research* 22 (184): 1–7. http://jmlr.org/papers/v22/21-0281.html.

Bischl, Bernd, Martin Binder, Michel Lang, Tobias Pielok, Jakob Richter, Stefan Coors, Janek Thomas, et al. 2021. "Hyperparameter Optimization: Foundations, Algorithms, Best Practices, and Open Challenges." *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, e1484.

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.

Bischl, Bernd, Olaf Mersmann, Heike Trautmann, and Claus Weihs. 2012. "Resampling Methods for Meta-Model Validation with Recommendations for Evolutionary Computation." *Evolutionary Computation* 20 (2): 249–75.

Bishop, Christopher M. 2006. *Pattern Recognition and Machine Learning*. Springer.

Bommert, Andrea, Xudong Sun, Bernd Bischl, Jörg Rahnenführer, and Michel Lang. 2020. "Benchmark for Filter Methods for Feature Selection in High-Dimensional Classification Data." *Computational Statistics & Data Analysis* 143: 106839. https://doi.org/https://doi.org/10.1016/j.csda.2019.106839.

Breiman, Leo. 1996. "Bagging Predictors." *Machine Learning* 24 (2): 123–40.

Bücker, Michael, Gero Szepannek, Alicja Gosiewska, and Przemyslaw Biecek. 2022. "Trans-

parency, Auditability, and Explainability of Machine Learning Models in Credit Scoring." *Journal of the Operational Research Society* 73 (1): 70–90. https://doi.org/10.1080/01605682.2021.1922098.

Chandrashekar, Girish, and Ferat Sahin. 2014. "A Survey on Feature Selection Methods." *Computers and Electrical Engineering* 40 (1): 16–28. https://doi.org/https://doi.org/10.1016/j.compeleceng.2013.11.024.

Collett, David. 2014. *Modelling Survival Data in Medical Research*. 3rd ed. CRC.

Davis, Jesse, and Mark Goadrich. 2006. "The Relationship Between Precision-Recall and ROC Curves." In *Proceedings of the 23rd International Conference on Machine Learning*, 233–40.

Demšar, Janez. 2006. "Statistical Comparisons of Classifiers over Multiple Data Sets." *Journal of Machine Learning Research* 7 (1): 1–30. https://jmlr.org/papers/v7/demsar06a.html.

Eddelbuettel, Dirk. 2020. "Parallel Computing with R: A Brief Review." *WIREs Computational Statistics* 13 (2). https://doi.org/10.1002/wics.1515.

Feurer, Matthias, and Frank Hutter. 2019. "Hyperparameter Optimization." In *Automated Machine Learning: Methods, Systems, Challenges*, edited by Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren, 3–33. Cham: Springer International Publishing. https://doi.org/10.1007/978-3-030-05318-5_1.

Guyon, Isabelle, and André Elisseeff. 2003. "An Introduction to Variable and Feature Selection." *Journal of Machine Learning Research* 3 (Mar): 1157–82.

Hand, David J, and Robert J Till. 2001. "A Simple Generalisation of the Area Under the ROC Curve for Multiple Class Classification Problems." *Machine Learning* 45: 171–86.

Hansen, Nikolaus, and Anne Auger. 2011. "CMA-ES: Evolution Strategies and Covariance Matrix Adaptation." In *Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation*, 991–1010.

Hastie, Trevor, Jerome Friedman, and Robert Tibshirani. 2001. *The Elements of Statistical Learning*. Springer New York. https://doi.org/10.1007/978-0-387-21606-5.

Holzinger, Andreas, Anna Saranti, Christoph Molnar, Przemyslaw Biecek, and Wojciech Samek. 2022. "Explainable AI Methods - a Brief Overview." *International Workshop on Extending Explainable AI Beyond Deep Models and Classifiers*, 13–38. https://doi.org/10.1007/978-3-031-04083-2_2.

Horst, Allison Marie, Alison Presmanes Hill, and Kristen B Gorman. 2020. *palmerpenguins: Palmer Archipelago (Antarctica) penguin data*. https://doi.org/10.5281/zenodo.3960218.

James, Gareth, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2014. *An Introduction to Statistical Learning: With Applications in r*. Springer Publishing Company, Incorporated.

Jamieson, Kevin, and Ameet Talwalkar. 2016. "Non-Stochastic Best Arm Identification and Hyperparameter Optimization." In *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*, edited by Arthur Gretton and Christian C. Robert, 51:240–48. Proceedings of Machine Learning Research. Cadiz, Spain: PMLR. http://proceedings.mlr.press/v51/jamieson16.html.

Japkowicz, Nathalie, and Mohak Shah. 2011. *Evaluating Learning Algorithms: A Classification Perspective*. Cambridge University Press.

Kalbfleisch, John D, and Ross L Prentice. 2011. *The statistical analysis of failure time data*. Vol. 360. John Wiley & Sons.

Karl, Florian, Tobias Pielok, Julia Moosbauer, Florian Pfisterer, Stefan Coors, Martin Binder, Lennart Schneider, et al. 2022. "Multi-Objective Hyperparameter Optimization - an Overview." https://doi.org/10.48550/ARXIV.2206.07438.

Kim, Ji-Hyun. 2009. "Estimating Classification Error Rate: Repeated Cross-Validation, Repeated Hold-Out and Bootstrap." *Computational Statistics & Data Analysis* 53 (11):

3735–45.

Krzyziński, Mateusz, Mikołaj Spytek, Hubert Baniecki, and Przemysław Biecek. 2023. "SurvSHAP(t): Time-dependent explanations of machine learning survival models." *Knowledge-Based Systems* 262: 110234. https://doi.org/https://doi.org/10.1016/j.knosys.2022.110234.

Kuehn, Daniel, Philipp Probst, Janek Thomas, and Bernd Bischl. 2018. "Automatic Exploration of Machine Learning Experiments on OpenML." https://arxiv.org/abs/1806.10961.

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.

Li, Lisha, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2018. "Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization." *Journal of Machine Learning Research* 18 (185): 1–52. https://jmlr.org/papers/v18/16-558.html.

López-Ibáñez, Manuel, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. 2016. "The Irace Package: Iterated Racing for Automatic Algorithm Configuration." *Operations Research Perspectives* 3: 43–58.

Molinaro, Annette M, Richard Simon, and Ruth M Pfeiffer. 2005. "Prediction Error Estimation: A Comparison of Resampling Methods." *Bioinformatics* 21 (15): 3301–7.

O'Neil, Cathy. 2016. *Weapons of Math Destruction: How Big Data Increases Inequality and Threatens Democracy*. New York, NY: Crown Publishing Group.

R Core Team. 2019. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. https://www.R-project.org/.

Romaszko, Kamil, Magda Tatarynowicz, Mateusz Urbański, and Przemysław Biecek. 2019. "modelDown: Automated Website Generator with Interpretable Documentation for Predictive Machine Learning Models." *Journal of Open Source Software* 4 (38): 1444. https://doi.org/10.21105/joss.01444.

Ruspini, Enrique H. 1970. "Numerical Methods for Fuzzy Clustering." *Information Sciences* 2 (3): 319–50. https://doi.org/https://doi.org/10.1016/S0020-0255(70)80056-1.

Schmidberger, Markus, Martin Morgan, Dirk Eddelbuettel, Hao Yu, Luke Tierney, and Ulrich Mansmann. 2009. "State of the Art in Parallel Computing with R." *Journal of Statistical Software* 31 (1). https://doi.org/10.18637/jss.v031.i01.

Schratz, Patrick, Marc Becker, Michel Lang, and Alexander Brenning. 2021. "mlr3spatiotempcv: Spatiotemporal resampling methods for machine learning in R," October. http://arxiv.org/abs/2110.12674.

Silverman, Bernard W. 1986. *Density Estimation for Statistics and Data Analysis*. Vol. 26. CRC press.

Simon, Richard. 2007. "Resampling Strategies for Model Assessment and Selection." In *Fundamentals of Data Mining in Genomics and Proteomics*, edited by Werner Dubitzky, Martin Granzow, and Daniel Berrar, 173–86. Boston, MA: Springer US. https://doi.org/10.1007/978-0-387-47509-7_8.

Snoek, Jasper, Hugo Larochelle, and Ryan P Adams. 2012. "Practical Bayesian Optimization of Machine Learning Algorithms." In *Advances in Neural Information Processing Systems*, edited by F. Pereira, C. J. Burges, L. Bottou, and K. Q. Weinberger. Vol. 25.

Sonabend, Raphael Edward Benjamin. 2021. "A Theoretical and Methodological Framework for Machine Learning in Survival Analysis: Enabling Transparent and Accessible Predictive Modelling on Right-Censored Time-to-Event Data." PhD, University College

London (UCL). https://discovery.ucl.ac.uk/id/eprint/10129352/.

Sonabend, Raphael, and Andreas Bender. 2023. *Machine Learning in Survival Analysis*. https://www.mlsabook.com.

Sonabend, Raphael, Andreas Bender, and Sebastian Vollmer. 2022. "Avoiding C-hacking when evaluating survival distribution predictions with discrimination measures." Edited by Zhiyong Lu. *Bioinformatics* 38 (17): 4178–84. https://doi.org/10.1093/bioinformatics/btac451.

Sonabend, Raphael, Franz J Király, Andreas Bender, Bernd Bischl, and Michel Lang. 2021. "mlr3proba: An R Package for Machine Learning in Survival Analysis." *Bioinformatics*, February. https://doi.org/10.1093/bioinformatics/btab039.

Tsallis, Constantino, and Daniel A Stariolo. 1996. "Generalized Simulated Annealing." *Physica A: Statistical Mechanics and Its Applications* 233 (1-2): 395–406.

Wickham, Hadley, and Garrett Grolemund. 2017. *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data*. 1st ed. O'Reilly Media. http://r4ds.had.co.nz/.

Wiśniewski, Jakub, and Przemysław Biecek. 2022. "The r Journal: Fairmodels: A Flexible Tool for Bias Detection, Visualization, and Mitigation in Binary Classification Models." *The R Journal* 14: 227–43. https://doi.org/10.32614/RJ-2022-019.

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.

Xiang, Yang, Sylvain Gubian, Brian Suomela, and Julia Hoeng. 2013. "Generalized Simulated Annealing for Global Optimization: The GenSA Package." *R J.* 5 (1): 13.

Index