Flexible and Robust Machine Learning Using mlr3 in R

Table of contents

Ge	tting Started	1
Pr	eface	3
	How to use this book	. 5
	Installation guidelines	. 6
	Community links	. 7
	Citation info	. 7
	mlr3book style guide	. 8
1.	Introduction and Overview	11
	1.1. Target audience	. 11
	1.2. From mlr to mlr3	. 12
	1.3. Design principles	. 12
	1.4. Package ecosystem	. 13
	1.5. Quick R6 introduction for beginners	. 13
	1.6. Quick data.table introduction for beginners	. 14
	1.7. Essential mlr3 utilities	. 15
2.	Basics	17
	2.1. Tasks	. 17
	2.2. Learners	. 26
	2.3. Evaluation	. 33
	2.4. Classification	. 35
	2.5. Additional Task Types	. 45
	2.6. Additional Learners	. 45
3.	Performance Evaluation and Comparison	49
	3.1. ROC Curve and Thresholds	. 50
	3.2. Resampling	. 53
	3.3. Benchmarking	. 62
4.	Hyperparameter Optimization	71
	4.1. Model Tuning	. 73
	4.2. Advanced Tuning	. 84
	4.3. Multi-Objective Tuning	. 89
	4.4. Automated Tuning with AutoTuner	. 92
	4.5. Nested Resampling	. 93
	4.6. Conclusion	. 97
	4.7 Evergises	

Table of contents

5.	eature Selection		99
	.1. Introduction	 	99
	.2. Filters	 	100
	.3. Wrapper Methods	 	106
6.	Pipelines		113
	.1. The Building Blocks: PipeOps	 	114
	.2. The Pipeline Operator: %>>%		
	.3. Nodes, Edges and Graphs		117
	.4. Modeling	 	119
	.5. Non-Linear Graphs	 	122
	.6. Adding new PipeOps	 	131
	.7. Special Operators	 	146
	.8. In-depth look into mlr3pipelines	 	152
7.	Preprocessing		169
8.	pecial Tasks		171
	.1. Survival Analysis		
	.2. Density Estimation		
	.3. Spatiotemporal Analysis		
	.4. Cost-Sensitive Classification		
	.5. Cluster Analysis	 •	203
9.	- echnical		21 3
	.1. Parallelization	 	213
	.2. Error Handling	 	222
	.3. Data Backends		
	.4. Parameters (using paradox)	 	235
	.5. Logging	 	261
10	Nodel Interpretation		265
	0.1. Penguin Task		
	0.2. iml		
	0.3. DALEX		
11	extending		28 3
	1.1. Adding new Learners		
	1.2. Adding new Measures		
	1.3. Adding new PipeOps		
	1.4. Adding new Tuners		
Re	erences		319
	. 0000		J-13
Αį	endices		32 1
Δ	Overview Tables		32 3
	/ TOI TIOTE 1 40 100		

В.	Solutions to exercises	327
	B.1. Solutions to Chapter 2	327
	B.2. Solutions to Chapter 3	327
	B.3. Solutions to Chapter 4	327
	B.4. Solutions to Chapter 5	328
	B.5. Solutions to Chapter 6	328
	B.6. Solutions to Chapter 8	328
	B.7. Solutions to Chapter 9	328
	B.8. Solutions to Chapter 10	328
	B.9. Solutions to Chapter 11	328
_		220
C.	Session Info	329

List of Figures

2.1.	Overview of the different stages of a learner	. 28
4.1.	In this code example we benchmark three random forest models with 1, 10, and 100 trees respectively, using 3-fold resampling, classification error loss, and tested on the simplified penguin dataset. The plot shows that the models with 10 and 100 trees are better performing across all three folds and 100 trees may be better than 10	. 72
4.2.	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 -	7 0
4.3.	Optimization Algorithm	
	performance.	
4.4. 4.5.	Histogram of sampled cost values. Pareto front of selected features and classification error. Black dots represent tested configurations, each red dot individually represents a Pareto-optimal configuration	. 85
4.6.	and all red dots together represent the Pareto front. Illustration of biased tuning. An algorithm is tuned by training on the Train dataset and then the optimal configuration is selected by evaluation on the Test data. The model's performance is then evaluated with the optimal configuration on the same	. 91
4.7.	data	. 93
	blocks represent 4-fold cross-validation for the inner resampling for HPO	. 94
9.1. 9.2.	Parallelization of a resampling using a 3-fold cross-validation	. 217
	nested inside a 5-fold cross-validation on 4 CPUs	. 220
9.3.	CPU utilization while parallelizing the inner resampling of a 2-fold cross-validation nested inside a 5-fold cross-validation on 4 CPUs	. 221
10.1	. Plot of the results from FeatureEffects. FeatureEffects computes and plots feature	0.00
10.2	effects of prediction models	
10.9	the values on the vertical axis	. 269
10.5	given the prediction model	. 270
10.4	FeatImp on train (left) and test (right)	
	FeatEffect train data set	
10.6	. FeatEffect test data set	272

List of Figures

10.7. 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

List of Tables

4.1.	Terminators available in mlr3tuning, their function call and default parameters	7 6
4.2.	Tuning algorithms available in mlr3tuning, their function call and the methodology.	7 9
4.3.	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	98

Getting Started

Editors

Lars Kotthoff, Raphael Sonabend, Michel Lang, Bernd Bischl

Contributing authors

- Marc Becker
- Przemysław Biecek
- Martin Binder
- Bernd Bischl
- Lukas Burk
- Giuseppe Casalicchio
- Sebastian Fischer
- Natalie Foss
- Lars Kotthoff
- Michel Lang
- Florian Pfisterer
- Damir Pulatov
- Lennart Schneider
- Patrick Schratz
- Raphael Sonabend
- Marvin Wright

Welcome to the Machine Learning in R universe. This is the electronic version of the upcoming book Flexible and Robust Machine Learning Using mlr3 in R. This book will teach you about the mlr3 universe of packages, from some machine learning methodology to implementations of complex algorithmic pipelines. We will cover how to use the mlr3 family of packages for data processing, fitting and training of machine learning models, tuning and hyperparameter optimization, feature selection, pipelines, data preprocessing, and model interpretability. In addition we will look at how our interface works beyond classification and regression settings to other fields including survival analysis, clustering, and more. Finally we will demonstrate how you can contribute to our universe by creating packages, learners, measures, pipelines, and other features.

We hope you enjoy reading our book and always welcome comments and feedback. If you notice any mistakes in the book we would appreciate if you could open an issue in the mlr3book issue tracker. All content in this book is licenced under CC BY-NC 4.0.

Preface

Welcome to the Machine Learning 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.

```
1 library(mlr3)
task = tsk("penguins")
 split = partition(task)
  learner = lrn("classif.rpart")
  learner$train(task, row_ids = split$train)
  learner$model
n = 231
node), split, n, loss, yval, (yprob)
      * denotes terminal node
1) root 231 129 Adelie (0.441558442 0.199134199 0.359307359)
  2) flipper_length< 207.5 145  44 Adelie (0.696551724 0.296551724 0.006896552)
    4) bill_length< 44.65 100
                                2 Adelie (0.980000000 0.020000000 0.000000000) *
    5) bill_length>=44.65 45
                               4 Chinstrap (0.066666667 0.911111111 0.022222222) *
  3) flipper_length>=207.5 86
                                4 Gentoo (0.011627907 0.034883721 0.953488372) *
 predictions = learner$predict(task, row_ids = split$test)
 predictions
```

3 Adelie Adelie

0.9380531

```
4 Adelie Adelie
5 Adelie Adelie
---

341 Chinstrap Adelie
343 Chinstrap Gentoo
344 Chinstrap Chinstrap

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

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")
10
   tuned_rf = pipeline_robustify(NULL, tuned_rf, TRUE) %>>%
11
       po("learner", tuned_rf)
12
   stack_lrn = ppl(
13
       "stacking",
14
       base_learners = lrns(c("classif.rpart", "classif.kknn")),
15
       lrn("classif.log_reg"))
16
   stack_lrn = pipeline_robustify(NULL, stack_lrn, TRUE) %>>%
17
       po("learner", stack_lrn)
18
19
   learners = c(tuned_rf, stack_lrn)
20
   bm = benchmark(benchmark_grid(tasks, learners, rsmp("holdout")))
21
   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 0.9605263

2: breast_cancer Stack 0.9122807

3: sonar RF 0.7681159

4: sonar Stack 0.7101449
```

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

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.

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.

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, and Chapter 6 contain more advanced implementation details and some ML theory. Chapter 8 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 9 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 10 discusses packages that can be integrated with mlr3 to provide model-agnostic interpretability methods. Finally, anyone who would like to contribute to our ecosystem should read Chapter 11.

Of course, you can also read the book cover to cover from start to finish. We have marked any section that contains complex technical information with an exclamation mark (!). You may wish to skip these sections if you are only interested in basic functionality. Similarly, we have marked sections that are optional, such as parts that are more methodological focused and do not discuss the software implementation, with an asterisk (*). Readers that are interested in the more technical detail will likely want to pay attention to the tables at the end of each chapter that show the relationship between our S3 'sugar' functions and the underlying R6 classes; this is explained in more detail in Chapter 1.

This book tries to follow the Diátaxis framework for documentation and so we include tutorials, how-to guides, API references, and explanations. This means that the conclusion of each chapter includes a short reference to the core functions learnt in the chapter, links to relevant posts in the mlr3gallery¹, and a few exercises that will cover content introduced in the chapter. You can find the solutions to these exercises in Appendix B.

Finally, if you want to reproduce any of the results in this book, note that the random seed is set as the chapter number and the sessionInfo printed in Appendix C.

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-universe² 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 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/"
    ))
```

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

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

If you want latest development versions of any of our packages, run

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

 $^{^{1}}$ https://mlr-org.com/gallery.html

²R-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.

with {pkg} replaced with the name of the package you want to install. You can see an up-to-date list of all our extension packages at https://github.com/mlr-org/mlr3/wiki/Extension-Packages.

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 Mattermost³.

For case studies and how-to guides, check out the mlr3gallery⁴ for extended practical blog posts. For updates on mlr you might find our blog⁵ 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 conduct⁶ and contribution guidelines⁷. 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).

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

Citation info

Every package in the mlr3verse has its own citation details that can be found on the respective GitHub repository.

To reference this book please use:

Becker M, Binder M, Bischl B, Foss N, Kotthoff L, Lang M, Pfisterer F, Reich N G, Richter J, Schratz P, Sonabend R, Pulatov D. 2023. "Flexible and Robust Machine Learning Using mlr3 in R". https://mlr3book.mlr-org.com.

³https://lmmisld-lmu-stats-slds.srv.mwn.de/signup_email?id=6n7n67tdh7d4bnfxydqomjqspo

⁴https://mlr-org.com/gallerv.html

⁵https://mlr-org.com/blog.html

⁶https://github.com/mlr-org/mlr3/blob/main/.github/CODE OF CONDUCT.md

⁷https://github.com/mlr-org/mlr3/blob/main/CONTRIBUTING.md

```
@misc{
    title = Flexible and Robust Machine Learning Using mlr3 in R
    author = {Marc Becker, Martin Binder, Bernd Bischl, Natalie Foss,
    Lars Kotthoff, Michel Lang, Florian Pfisterer, Nicholas G. Reich,
    Jakob Richter, Patrick Schratz, Raphael Sonabend, Damir Pulatov},
    url = {https://mlr3book.mlr-org.com},
    year = \{2023\}
}
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},
}
```

mlr3book style guide

Throughout this book we will use our own style guide that can be found in the mlr3 wiki⁸. 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

 $^{^8 \}rm https://github.com/mlr-org/mlr3/wiki/Style-Guide$

- package::function() (for functions *outside* the mlr-org ecosystem)
- \bullet function() (for functions inside the mlr-org ecosystem) With link to function documentation page
- \$field
- \$method()

1. Introduction and Overview

The (Machine Learning in R) mlr3 (Lang et al. 2019) package and ecosystem provide a generic, object-oriented, and extensible framework for classification, regression, survival analysis, and other machine learning tasks for the R language (R Core Team 2019) (task types are discussed in detail in ?@sec-tasks-types). This unified interface provides functionality to extend and combine existing machine learning algorithms (learners), intelligently select and tune the most appropriate technique for a specific machine learning task, 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 9.1).

mlr3 has similar overall aims to caret, tidymodels, scikit-learn for Python, and MLJ 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 in particular tidymodels makes it very easy to perform simple machine learning tasks, mlr3 is more geared towards advanced machine learning. To get a quick overview of how to do things in the mlr3verse, see the mlr3 cheatsheets¹.

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.

1.1. Target audience

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. (James et al. 2014) provides a comprehensive introduction for those new to machine learning.

mlr3 provides a domain-specific language for machine learning in R that allows to do everything from simple exercises to complex projects. We target both **practitioners** who want to quickly apply machine learning algorithms and **researchers** who want to implement, benchmark, and compare their new methods in a structured environment.

¹https://cheatsheets.mlr-org.com/

1.2. From mlr to mlr3

The mlr package (Bischl et al. 2016) was first released to CRAN² 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 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 impossible to integrate all of these changes into the original design of mlr. Instead, we decided to start working on a reimplementation in 2018, which resulted in the first release of mlr3 on CRAN in July 2019.

The new design and the integration of further and newly-developed R packages (especially R6, future, and data.table) makes mlr3 much easier to use, maintain, and in many regards more efficient than its predecessor mlr.

1.3. Design principles

*

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

- Command-line before GUI. 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 a graphical user interfaces (GUIs) because their dependencies would make installation unnecessarily complex, especially on headless servers. For the same reason, visualizations of data and results are provided in the extra package mlr3viz, which avoids dependencies on ggplot2. mlr3shiny provides an interface for some basic machine learning tasks using the shiny package.
- Object-oriented programming (OOP). Embrace R6 for a clean, object-oriented design, object state-changes, and reference semantics.
- Tabular data. Embrace data.table for fast and convenient computations on tabular data.
- Unify container and result classes as much as possible and provide result data in data.tables. This considerably simplifies the API and allows easy selection and "split-apply-combine" (aggregation) operations. We combine data.table and R6 to place references to non-atomic and compound objects in tables and make heavy use of list columns.
- Defensive programming and type safety. All user input is checked with checkmate (Lang 2017). We document return types, and avoid mechanisms popular in base R which "simplify" the result unpredictably (e.g., sapply() or the drop argument for indexing data.frames).
- Light on dependencies. One of the main maintenance burdens for mlr was to keep up with changing learner interfaces and behavior of the many packages it depended on. We require far fewer packages in mlr3 to make installation and maintenance easier. We still provide the same functionality, but it is split into more packages that have fewer dependencies individually. As mentioned above, this is particularly the case for all visualization functionality, which is contained in a separate package to avoid unnecessary dependencies in all other packages.

²https://cran.r-project.org

1.4. Package ecosystem

mlr3 uses the following packages that not developed by core members of the mlr3 team:

- R6: Reference class objects.
- data.table: Extension of R's data.frame.
- digest: Hash digests.
- uuid: Unique string identifiers.
- lgr: Logging.
- mlbench: Collection of machine learning data sets.
- evaluate: For capturing output, warnings, and exceptions (Section 9.2).
- future / future.apply: For parallelization (Section 9.1).

These are core packages in the R ecosystem.

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.



A complete list with links to the repository for the respective package can be found on our package overview $page^a$.

 $^a {\it https://github.com/mlr-org/mlr3/wiki/Extension-Packages}$

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.

1.5. Quick R6 introduction for beginners

R6 is one of R's more recent paradigm for object-oriented programming (OOP). It addresses short-comings 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 <code>\$new()</code>. For example, <code>foo = Foo\$new(bar = 1)</code> creates a new object of class <code>Foo</code>, setting the <code>bar</code> 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

1. Introduction and Overview

\$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).



For more details on R6, have a look at the excellent R6 vignettes^a, especially the introduction^b. For comprehensive R6 information, we refer to the R6 chapter from Advanced R^c .

```
<sup>a</sup>https://r6.r-lib.org/
```

1.6. 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.tables 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.frames or tibble/dplyrs are just a single 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
```

```
х у
```

^bhttps://r6.r-lib.org/articles/Introduction.html

^chttps://adv-r.hadley.nz/r6.html

^{1: 1} a

^{2: 2} a

^{3: 3} b

^{4: 4} b

```
5: 5 c
6: 6 c
```

data.tables can be used much like data.frames, 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 RPubs post by Ronald Stalder³) that make it easy to combine multiple data.tables in different ways.



For an in-depth introduction, we refer the reader to the excellent data. table introduction $vignette^{a}$.

^ahttps://cran.r-project.org/web/packages/data.table/vignettes/datatable-intro.html

1.7. Essential mlr3 utilities

Sugar functions

Most objects in mlr3 can be created through special functions that are called *sugar functions*. They provide shortcuts for common code idioms, reducing the amount of code a user has to write. We heavily use sugar functions throughout this book and give the equivalent "full form" only for reference. In most cases, the sugar functions will achieve what you want to do, but in special cases you may have to use the full R6 code. For example lrn("regr.rpart") is the sugar version of LearnerRegrRpart\$new().

Dictionaries

mlr3 uses dictionaries for learners, tasks, and other objects that are often used in common machine learning 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. Often, values in dictionaries are accessed through sugar functions that automatically use the applicable dictionary without the user having to specify it; 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.

 $^{^3} https://rstudio-pubs-static.s3.amazonaws.com/52230_5ae0d25125b544caab32f75f0360e775.html$

1. Introduction and Overview

For example a learner can be retrieved directly from the mlr_learners Dictionary using the key "classif.featureless" (mlr_learners\$get("classif.featureless")).

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. In the background, mlr3viz uses ggplot2. mlr3viz extends fortify and autoplotfor use with common mlr3 outputs including Prediction, Learner, and Benchmark objects (these objects will be introduced and covered in the next chapter). The most common use of mlr3viz is the autoplot() function, where the type of the object passed determines the type of the plot. Plotting types can be found by running ?autoplot.X. For example, the documentation of plots for regression tasks can be found by running ?autoplot.TaskRegr.

2. Basics

TODO (150-200 WORDS)

In this chapter, we will introduce the essential building blocks of mlr3, along with the corresponding R6 classes and operations used for machine learning.

The data, which mlr3 encapsulates in tasks, is split into non-overlapping training and test sets. As we are interested in models that generalize beyond the training data rather than just memorizing it, separate test data allows to evaluate models in an unbiased way and assess to what extent they have learned the concepts that underlie the data. The training data is given to a learner, which builds a model based on it. Examples of such learners include classification tree learners (classif.rpart), regression support vector machine learners (regr.svm), and many others, see the complete list here¹. The model a learner constructs is saved in the learner object and can then used to produce predictions on the test data. These predictions can be compared to the ground truth values to assess the quality of the model with various performance measures. Usually, the value of a measure is a numeric score. This value is usually called the estimate of the generalization error – given new data that we have not seen before, how well do we estimate the learned model to perform?

Partitioning the entire data set into training and test sets is called resampling in mlr3. A single resampling may not provide the best estimate of the generalization performance because it is based on only a single data point. As data are usually partitioned randomly, a single split can produce training and test sets that are very different, hence creating the misleading impression that the particular type of model does not perform well. Repeating the procedure of partitioning, building a model on the training set, and evaluating it on the test set gives multiple such data points and in general provides a more robust estimate of the generalization performance.

2.1. Tasks

Tasks are objects that contain the (usually tabular) data and additional meta-data that defines a machine learning problem. The meta-data contains, 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.

¹https://mlr-org.com/learners.html

2.1.1. Built-in Tasks

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

To get a task from the dictionary, use the tsk() function and assign the return value to a new feature. 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 all mlr3-objects come with:

```
task_mtcars$help()
```



If you are familiar with R's help system (i.e. the help() and ? functions), this may seem confusing. task_mtcars is the feature that holds the penguins 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")
```



Thousands more data sets are readily available via Openml.org^a (Vanschoren et al. 2013) and mlr3oml. For example, to download the data set credit-g^b with data id 31 and automatically convert it to a classification task, all you need to do is:

```
1 library("mlr3oml")
2 tsk("oml", task_id = 31)

ahttps://openml.org
bhttps://www.openml.org/search?type=data&id=31
```

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.

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).

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() in data 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 function, 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")
task_mtcars

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

? Tip

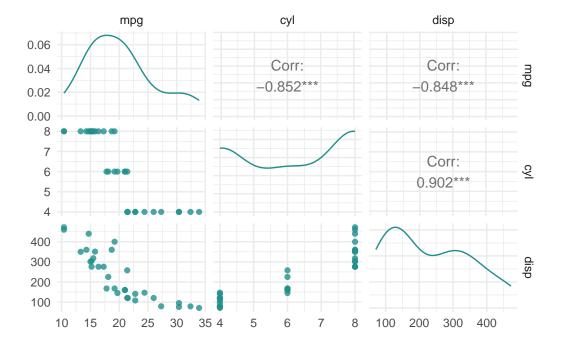
The task constructors as_task_regr() and as_task_classif() will check for non-ascii characters in the column names of your data. As many ML models do not work properly with arbitrary UTF8 names, mlr3 defaults to throw an error if any of the column names contains either a non-ascii character or does not comply to 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 due to variable naming).

data can be any rectangular data format, e.g. a data.frame(), data.table(), or tibble(). Internally, the data is converted and stored in an abstract 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 given provided, the departed name of x will be used (an R way of turning data into strings).

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")
```





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.

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:

```
task_mtcars$nrow
```

[1] 32

```
task_mtcars$ncol
```

[1] 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.

```
task_mtcars$feature_names
```

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

```
task_mtcars$target_names
```

[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 slot:

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

[1] 1 2 3 4 5 6

Row IDs are not used as features when predicting, they are meta-information 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:

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

```
cyl
                                     disp
     mpg
                       :4.000
Min.
       :10.40
               Min.
                                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.:326.0
3rd Qu.:22.80
                3rd Qu.:8.000
       :33.90
                        :8.000
Max.
                Max.
                                 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 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 cloned 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.



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
```

2.1.4. Roles (Rows and Columns)

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, see the documentation of Task. These 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
```

```
$feature
[1] "am" "carb"
$target
[1] "mpg"
$name
[1] "model"
$order
character(0)
```

```
$stratum
character(0)
$group
character(0)
$weight
character(0)
```

As you can see, there are additional column roles; the interested reader is referred to the documentation of Task for more detail. 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"
```

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
                            160 110 3.90 2.620 16.46
                                                                   4
                                                                            1
    Mazda RX4 Wag 21.0
                         6 160 110 3.90 2.875 17.02
                                                                   4
                                                                            2
       Datsun 710 22.8
                                                                            3
                         4 108 93 3.85 2.320 18.61
                                                                   1
   Hornet 4 Drive 21.4
                            258 110 3.08 3.215 19.44
                                                                   1
                                                                            4
                                                      1 0
                                                              3
                         6
Hornet Sportabout 18.7
                         8 360 175 3.15 3.440 17.02
                                                              3
                                                                   2
                                                                            5
                         6 225 105 2.76 3.460 20.22
                                                                            6
          Valiant 18.1
[...] (26 rows omitted)
```

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

- 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().

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. Because the underlying data are still

2. Basics

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
```

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

- 1. Role use: Rows that are generally available for training (although they may also be used for the test set). This role is the default role. The \$filter() call changes this role, in the same way that \$select() changes the "feature" column role.
- 2. Role validation: Rows that are not used for training. Rows that have missing values in the target column during task creation are automatically set to the validation role.

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

- 1. It is often good practice to validate the final model on an external validation set to identify overfitting.
- 2. Some observations may be unlabeled in the original data, e.g. in competitions like Kaggle².

These observations cannot be used for training a model, but can be used for getting predictions from a trained model.

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. Unlike tasks, only supported learners can be used in mlr3. The list of learners supported in the base package mlr3 is deliberately small; 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-information about the learners. The base class of each learner is Learner.

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

²https://www.kaggle.com/

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

Each learner provides the following meta-information:

- **\$feature_types**: the type of features the learner can deal with.
- \$packages: the packages required to train a model with this learner and make predictions.
- **\$properties**: additional properties and capabilities. For example, a learner has the property "missings" if it is able to handle missing feature values, and "importance" if it computes and allows to extract data on the relative importance of the features.
- \$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").

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

```
1 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: The 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**: The 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. 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). 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

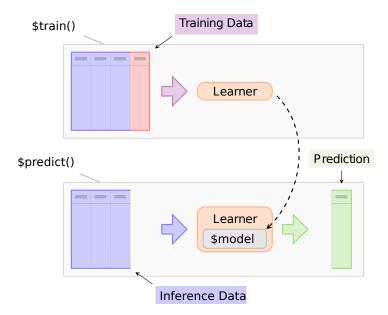


Figure 2.1.: Overview of the different stages of a learner.

(splits\$train). This operation adds the learned model to the existing Learner object. We can now access the stored model via the field \$model.

```
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

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 *</pre>
```

We see that the learner has identified features in the task that are predictive of the class (mpg) and uses them to partition observations in the tree. There are additional details on how the data

is partitioned across branches of the tree; the textual representation of the model depends on the type of learner. For more information on this particular type of model and its output, see rpart::print.rpart().

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 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	${\tt nlevels}$	default	value
1:	ср	${\tt ParamDbl}$	0	1	Inf	0.01	
2:	keep_model	${\tt ParamLgl}$	NA	NA	2	FALSE	
3:	maxcompete	${\tt ParamInt}$	0	Inf	Inf	4	
4:	maxdepth	${\tt ParamInt}$	1	30	30	30	
5:	maxsurrogate	${\tt ParamInt}$	0	Inf	Inf	5	
6:	minbucket	${\tt ParamInt}$	1	Inf	Inf	<nodefault[3]></nodefault[3]>	
7:	minsplit	${\tt ParamInt}$	1	Inf	Inf	20	
8:	surrogatestyle	${\tt ParamInt}$	0	1	2	0	
9:	usesurrogate	${\tt ParamInt}$	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, it is a named list:

```
learner_rpart$param_set$values
```

\$xval

[1] 0

```
learner_rpart$param_set$values$minsplit = 10
learner_rpart$param_set$values
```

\$xval

[1] 0

\$minsplit

[1] 10



It is possible to assign all hyperparameters in one go by assigning a named list to **\$values**: learner\$param_set\$values = list(minsplit = 10, ...). This operation removes all previously-set hyperparameters.

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

```
learner_rpart = lrn("regr.rpart", minsplit = 10)
2 learner_rpart$param_set$values
$xval
[1] 0
$minsplit
[1] 10
  learner_rpart$train(task_mtcars, splits$train)
  learner_rpart$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 at 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 now use it to make predictions. We can give the test partition to the **\$predict()** function:

```
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** that predicts a numeric quantity. The "truth" column contains the ground truth data that was not used to get a prediction. The "response" column contains the value predicted by the model, allowing for easy comparison with the ground truth data.

We can also use separate data to make predictions, which can either come from another source (and be part of a separate task), or simply be a separate data.frame:

```
mtcars_new = data.frame(cyl = c(5, 6),
                             disp = c(100, 120),
2
                             hp = c(100, 150),
3
                             drat = c(4, 3.9),
4
                             wt = c(3.8, 4.1),
                             qsec = c(18, 19.5),
                             vs = c(1, 0),
                             am = c(1, 1),
8
                             gear = c(6, 4),
                             carb = c(3, 5))
10
   mtcars_new
11
```

```
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-information 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
```

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

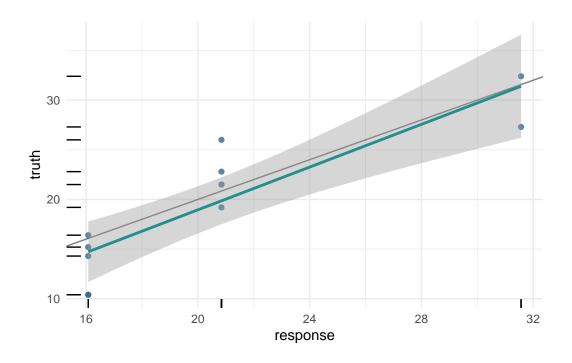
We can also access the predictions directly:

```
# directly access the predicted quantities
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)
```



2.2.3. Changing the Prediction Type

Regression learners default to predicting the 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 linear model instead, from the mlr3learners package:

```
Tip
```

Section 2.6.1 shows how to list learners that support the standard error prediction type.

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

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 support 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"
                                "regr.rmsle" "regr.rrse"
[11] "regr.rae"
                  "regr.rmse"
                                                           "regr.rse"
[16] "regr.rsq"
                                "regr.smape" "regr.srho"
                                                           "regr.sse"
                  "regr.sae"
```

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 the difference between the predicted value and the ground truth value (except for unsupervised tasks, which we will discuss later). 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
```

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 it 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 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	Tris 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
                                          Pima Indian Diabetes
 8:
               pima
                                                                  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 Animals
                                                                  classif 101
                7.00
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.

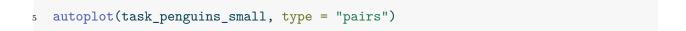
The target variable, species, is of type factor and has the following three classes or levels:

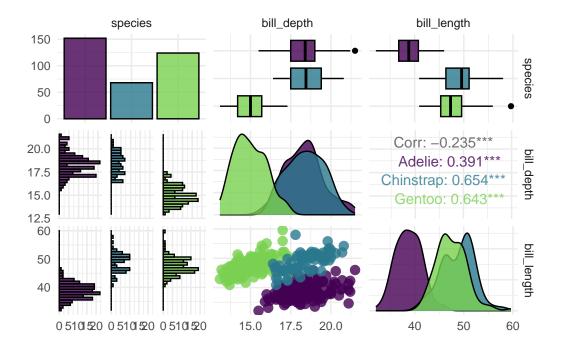
```
unique(task_penguins$data(cols = "species"))

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$clone()
task_penguins_small$select(head(task_penguins_small$feature_names, 2))
```





2.4.2. Classification Learners

Classification learners (LearnerClassif) are a different 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

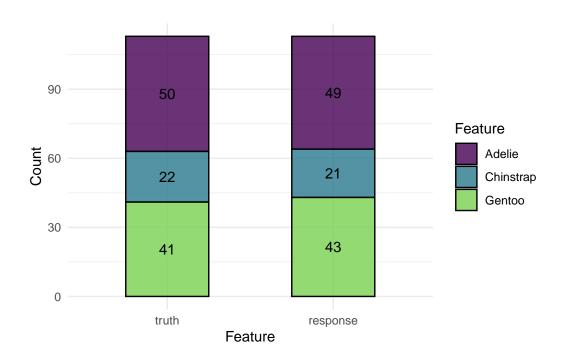
- * Model: -
- * Parameters: xval=0
- * Packages: mlr3, rpart
- * Predict Types: [response], prob
- * Feature Types: logical, integer, numeric, factor, ordered
- * Properties: importance, missings, multiclass, selected_features, twoclass, weights

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)
3 learner_rpart$model
n = 231
node), split, n, loss, yval, (yprob)
     * denotes terminal node
1) root 231 129 Adelie (0.44155844 0.19913420 0.35930736)
 2) flipper_length< 206 143  43 Adelie (0.69930070 0.30069930 0.00000000)
   3) flipper_length>=206 88
                        5 Gentoo (0.02272727 0.03409091 0.94318182) *
predictions = learner_rpart$predict(task_penguins, splits$test)
2 predictions
<PredictionClassif> for 113 observations:
   row_ids
           truth response
        1
            Adelie Adelie
           Adelie
                   Adelie
        3
            Adelie
                   Adelie
      337 Chinstrap
                    Gentoo
      339 Chinstrap Chinstrap
      342 Chinstrap Chinstrap
```

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

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



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:

r	cow_ids	truth	response	<pre>prob.Adelie</pre>	${\tt prob.Chinstrap}$	<pre>prob.Gentoo</pre>
	1	Adelie	Adelie	0.96969697	0.03030303	0.0000000
	2	Adelie	Adelie	0.96969697	0.03030303	0.0000000
	3	Adelie	Adelie	0.96969697	0.03030303	0.0000000
	337	${\tt Chinstrap}$	Gentoo	0.02272727	0.03409091	0.9431818
	339	${\tt Chinstrap}$	${\tt Chinstrap}$	0.09090909	0.90909091	0.0000000
	342	${\tt Chinstrap}$	${\tt Chinstrap}$	0.09090909	0.90909091	0.0000000



Tip

Section 2.6.1 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"
                            "classif.fpr"
[13] "classif.fp"
                                                   "classif.logloss"
                            "classif.mauc_au1u"
[16] "classif.mauc_au1p"
                                                   "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"
```

Some of these measures require that 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 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 0.9380531
```

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 on Wikipedia³.

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

```
predictions$confusion
```

t	truth		
response	Adelie	${\tt Chinstrap}$	Gentoo
Adelie	48	1	0
Chinstrap	2	18	1
Gentoo	0	3	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")

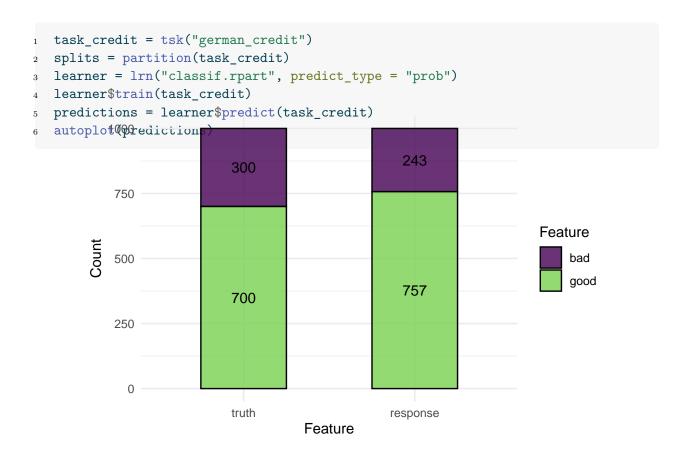
# switch positive class to level 'M'
task_sonar$positive = "M"
```

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.

³https://en.wikipedia.org/wiki/Confusion_matrix

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.



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)
autoplot(predictions)
```





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

Classification tasks with more than two predicted classes are very common and you may wish to change the thresholds for these tasks. The default threshold for each class is 0.5. Thresholds work slightly differently with multiple classes:

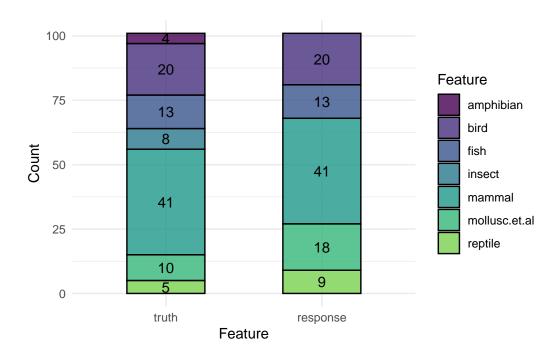
- The probability for a data point is divided by each class threshold resulting in n ratios (n classes).
- The highest ratio is selected (ties are random by default).

Lowering the threshold for a class will mean that it is more likely to be predicted and raising it will have the opposite effect. The zoo dataset illustrates this concept nicely.

When trained normally some classes are not ever predicted (see below).

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

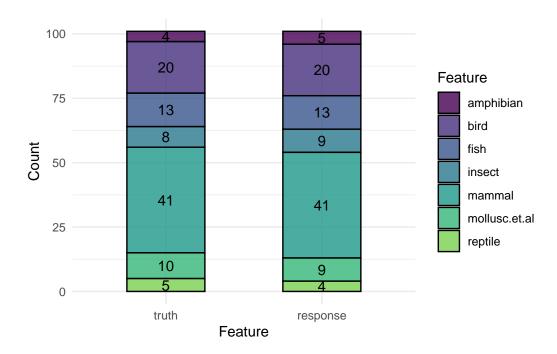
autoplot(preds)
```



The classes amphibian and insect are never predicted. The classes mollusc and reptile are predicted more than it appears in the truth data. We can try to correct for this by lowering the threshold for amphibian and insect; and/or raising it for mollusc and reptile. \$set_threshold() can be used with a named numeric vector of thresholds like so:

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

# new predictions
# | fig-cap: Stacked barplot of true (truth) and estimated (response) class labels.
autoplot(preds$set_threshold(new_thresh))
```



You can see that altering the thresholds can lead to better performance. Read more about how to automatically tuning thresholds in Chapter 6.

2.5. 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 8.

2.6. Additional Learners

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

The list of learners included in the mlr3 package is dliberately small to avoid large sets of dependencies for the 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.
- Rpart decision tree learner classif.rpart: Tree learner from rpart.
- Rpart 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.
- Naive 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 website⁴.

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.

2.6.1. Listing Learners

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

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

task_type	label	key	
classif	Adaptive Boosting	<pre>classif.AdaBoostM1</pre>	1:
classif	Tree-based Model	classif.C50	2:
classif	Nearest Neighbour	classif.IBk	3:
classif	Tree-based Model	classif.J48	4:
classif	Propositional Rule Learner.	classif.JRip	5:
surv	Priority Lasso	<pre>surv.priority_lasso</pre>	136:
surv	Random Forest	surv.ranger	137:
surv	Random Forest	surv.rfsrc	138:
surv	Support Vector Machine	surv.svm	139:
surv	Gradient Boosting	surv.xgboost	140:

⁴https://mlr-org.com/learners.html

4 variables not shown: [feature_types, packages, properties, predict_types]

The resulting data.table contains a lot of meta-information 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
                          Bayesian Additive Regression Trees
 4:
        regr.bart
                                                                   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:
                                                         <NA>
         regr.svm
                                                                   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:

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

```
key
                                                         label task_type
         regr.debug
                                 Debug Learner for Regression
1:
                                                                    regr
2:
         regr.earth Multivariate Adaptive Regression Splines
                                                                    regr
3: regr.featureless
                               Featureless Regression Learner
                                                                    regr
4:
                        Generalized Additive Regression Model
           regr.gam
                                                                    regr
5:
           regr.glm
                                Generalized Linear Regression
                                                                    regr
6:
            regr.km
                                                          <NA>
                                                                    regr
7:
                                                          <NA>
            regr.lm
                                                                    regr
8:
           regr.mob
                           Model-based Recursive Partitioning
                                                                    regr
                                                          <NA>
9:
        regr.ranger
                                                                    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) "missing
```

	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	${\tt Featureless\ Classification\ Learner}$	classif
7:	classif.gbm	Gradient Boosting	classif
8:	${\tt classif.imbalanced_rfsrc}$	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></na>	classif
4 v	ariables not shown: [featu	re_types, packages, properties, pre	edict_types]

TODO (150-200 WORDS)

Now that we are familiar with the basics of how to construct tasks and learners and how to fit models, we would like to know more about how to assess fitted models. In particular, we would like to assign one or several numeric *scores* to a given model. This allows us to select a *best* model out of several fitted models and furthermore, helps us assess whether the models we have fitted are useful for a task at hand. Scores in this context can reflect different qualities of a model, depending on the measures we employ. To provide an example, in some cases we might be interested in how well our model predicts (in terms of global accuracy). In others we might care about other perspectives, such as how many features it uses or how fast it predicts.

In supervised learning contexts, the core idea in performance evaluation is often that we would like to assess how good a fitted model will work on new, unseen data points. In order to mimic this, we often hold out some data during model development. Afterwards, we use predictions made on the held-out data and compare them to the true label to simulate how our model would have fared on new data points. This strategy is often called *train-test* split or *holdout*. Performance Measures now summarize such comparisons in a single number, e.g. a model's accuracy is the fraction of cases in which it predicted the true outcome correctly.

If datasets are small, the model might perform poorly on the test data by chance. We would therefore like to repeat this process multiple times to reduce the variance in our estimation. This gives rise to different resampling (Section 3.2) strategies encoding more complex strategies.

Performance Measures

Measures are a way to quantify model properties along relevant dimensions. We cover how to *construct Measure objects for different task_types, * use Measure objects to score models

An in-depth guide on how to construct custom measures can be found in the extending measures section.

Resampling

Resampling (Section 3.2) is a methodology to construct training and test splits. We cover how to

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

In this section, we will use Resampling objects that define the resampling strategy we want to employ. Calling resample on a Task, Learner and a Resampling object then will return a ResampleResult which contains the results of applying the strategy on a Task and Learner. Additional information on resampling can be found in the section about nested resampling and in the chapter on model optimization.

Benchmarking

Benchmarking is used to compare the performance of different models, for example models trained with different learners, on different tasks, or with different resampling methods. This is usually done to obtain an overview of how different methods perform across different tasks. We cover how to

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

3.0.0.1. Further Reading

- If data is pre-processed before feeding it to a model, the train-test split needs to be taken into account. Pipelines solve this problem by cleanly separating train and predict steps!
- Depending on the task at hand, more involved resampling strategies might be required, e.g. for spatiotemporal data.
- Bischl et al. (2012) provide an overview of resampling methods.

3.1. ROC Curve and Thresholds

As we have seen before, binary classification is unique because of the presence of a positive and negative class and a threshold probability to distinguish between the two. ROC Analysis, which stands for receiver operating characteristics, applies specifically to this case and allows a better picture of the trade-offs when choosing between the two classes.

We saw earlier that one can retrieve the confusion matrix of a Prediction by accessing the \$confusion field. In the following code chunk, we first retrieve the Sonar classification task and construct a classification tree learner. Next, we use the partition() helper function to randomly split the rows of the Sonar task into two disjunct sets: a training set and a test 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.

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

# split into training and test
splits = partition(task, ratio = 0.8)
print(str(splits))
```

```
List of 2
$ train: int [1:167] 3 4 5 6 7 8 9 12 13 14 ...
$ test : int [1:41] 1 2 10 11 24 28 35 37 46 48 ...

NULL

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

truth
response M R
M 19 5
```

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

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

R 3 14

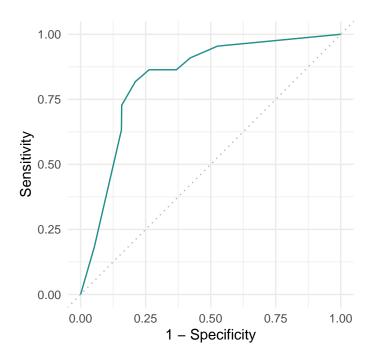
- True Positive Rate (TPR): How many of the true positives did we predict as positive?
- True Negative Rate (TNR): How many of the true negatives did we predict as negative?
- Positive Predictive Value PPV: If we predict positive how likely is it a true positive?
- Negative Predictive Value NPV: If we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

It is difficult to achieve a high TPR and low FPR simultaneously. We can characterize the behavior of a binary classifier for different thresholds by plotting the TPR and FPR values – this is the ROC curve. The best classifier lies on the top-left corner – the TPR is 1 and the FPR is 0. Classifiers on the diagonal produce labels randomly (possibly with different proportions). For example, if each positive x will be randomly classified with 25% as "positive", we get a TPR of 0.25. If we assign each negative x randomly to "positive" we get a FPR of 0.25. In practice, we should never obtain a classifier clearly below the diagonal – ROC curves for different labels are symmetric with respect to the diagonal, so a curve below the diagonal would indicate that the positive and negative class labels have been switched by the classifier.

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

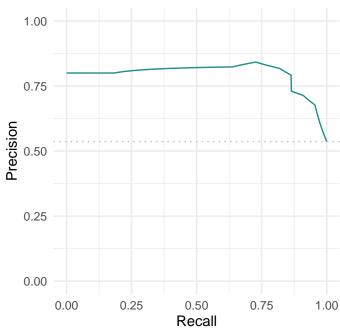
```
library("mlr3viz")

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



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

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



3.2. Resampling

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

Name	Identifier
cross validation	"CV"
leave-one-out cross validation	"loo"
repeated cross validation	"repeated_cv"
bootstrapping	"bootstrap"
subsampling	"subsampling"
holdout	"holdout"
in-sample resampling	"insample"
custom resampling	"custom"

Note

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

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

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

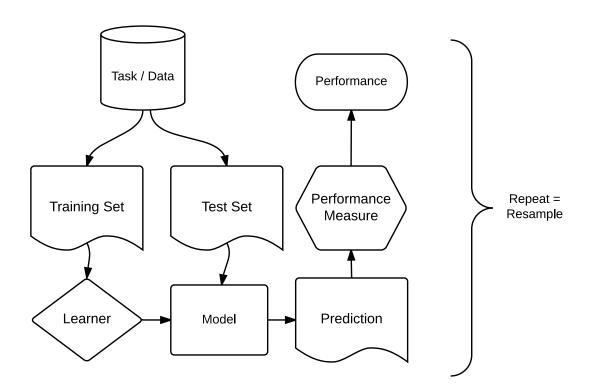
3.2.1. Settings

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

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

When performing resampling with a dataset, we first need to define which approach to use. mlr3 resampling strategies and their parameters can be queried by looking at the data.table output of the mlr_resamplings dictionary; this also lists the parameters that can be changed to affect the behavior of each strategy:

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



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



Additional resampling methods for special use cases are available via extension packages, such as mlr3spatiotempcv for spatial data.

What we showed in the train/predict/score part is the equivalent of holdout resampling, done manually, so let's consider this one first. We can retrieve elements from the dictionary mlr_resamplings with the convenience function rsmp():

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

<ResamplingHoldout>: Holdout

* Iterations: 1

* Instantiated: FALSE * Parameters: ratio=0.6667

Note

Note that the \$is_instantiated field is set to FALSE. This means we did not apply the strategy to a dataset yet.

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

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

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

2. Specifying the resampling parameters directly during construction:

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

<ResamplingHoldout>: Holdout

* Iterations: 1

* Instantiated: FALSE * Parameters: ratio=0.8

3.2.2. Instantiation

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

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

```
resampling$instantiate(task)
train = resampling$train_set(1)
test = resampling$test_set(1)
str(train)
```

int [1:275] 2 3 4 5 6 7 8 9 10 11 ...

```
str(test)
```

int [1:69] 1 13 14 15 17 20 28 32 42 44 ...

```
# are they disjunct?
intersect(train, test)
```

integer(0)

```
# are all row ids either in train or test?
setequal(c(train, test), task$row_ids)
```

[1] TRUE

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

3.2.3. Execution

With a Task, a Learner, and a Resampling object we can now perform a resampling: fit the learner on a subset of the task repeatedly and predict on the left-out observations. For this, we call the resample() function which returns a ResampleResult object. We can tell resample() to keep the fitted models (for example for later inspection) by setting the store_models option to TRUE.

Note

The butcher package allows you to remove components of a fitted model that may no longer be needed. However, from the point of view of the mlr3 framework, it is impossible to find a good compromise here. Of course, as much balast as possible should be removed. However, if too much is removed, some graphics, outputs or third-party extensions may not work anymore as intended. Our conservative policy here is that the training data should not be part of the stored model. If this is the case somewhere, please open an issue.

To manually apply butcher, just replace the stored model in the learner:

```
task = tsk("penguins")
learner = lrn("classif.rpart")
learner$train(task)

learner$model = butcher::butcher(learner$model)
```

Per default, mlr3 discards fitted models after the prediction step to keep the ResampleResult object small w.r.t. its memory consumption.

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

```
formula of the second sec
```

<ResampleResult> of 3 iterations

* Task: penguins

* Learner: classif.rpart * Warnings: 0 in 0 iterations * Errors: 0 in 0 iterations

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

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

```
1 rr$aggregate(msr("classif.ce"))
classif.ce
0.05819985
```

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

• Check for warnings or errors:

```
rr$warnings

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

rr$errors
```

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

• Extract and inspect the resampling splits; this allows us to see in detail which observations were used for what purpose when:

```
rr$resampling
<ResamplingCV>: Cross-Validation
* Iterations: 3
* Instantiated: TRUE
* Parameters: folds=3

rr$resampling$iters
```

[1] 3

```
str(rr$resampling$train_set(1))
   int [1:229] 9 12 16 17 19 24 25 29 30 34 ...
 str(rr$resampling$test_set(1))
   int [1:115] 1 5 8 10 15 18 21 22 23 26 ...
• Retrieve the model trained in a specific iteration and inspect it, for example, to investigate
  why the performance in this iteration was very different from the average:
 1 lrn = rr$learners[[1]]
 2 lrn$model
 n = 229
 node), split, n, loss, yval, (yprob)
        * denotes terminal node
  1) root 229 128 Adelie (0.44104803 0.22707424 0.33187773)
    2) bill_length< 42.35 94
                             1 Adelie (0.98936170 0.00000000 0.01063830) *
    3) bill_length>=42.35 135 60 Gentoo (0.05925926 0.38518519 0.55555556)
                                     6 Chinstrap (0.10344828 0.89655172 0.00000000) *
      6) island=Dream, Torgersen 58
                            2 Gentoo (0.02597403 0.00000000 0.97402597) *
      7) island=Biscoe 77
• Extract the individual predictions:
 rr$prediction() # all predictions merged into a single Prediction object
  <PredictionClassif> for 344 observations:
                 truth response prob.Adelie prob.Chinstrap prob.Gentoo
      row_ids
                                                  0.00000000 0.01063830
                           Adelie 0.98936170
            1
                 Adelie
            5
                 Adelie
                                                  0.00000000 0.01063830
                           Adelie 0.98936170
            8
                 Adelie
                          Adelie 0.98936170
                                                  0.0000000 0.01063830
                           Gentoo 0.01162791
                                                  0.01162791 0.97674419
          340 Chinstrap
          342 Chinstrap Chinstrap 0.04761905
                                                  0.92857143 0.02380952
          343 Chinstrap
                           Gentoo 0.01162791
                                                  0.01162791 0.97674419
 rr$predictions()[[1]] # predictions of first resampling iteration
  <PredictionClassif> for 115 observations:
                 truth response prob.Adelie prob.Chinstrap prob.Gentoo
      row_ids
                 Adelie
                           Adelie
                                    0.9893617
                                                   0.0000000
                                                               0.0106383
            1
            5
                 Adelie
                           Adelie
                                    0.9893617
                                                   0.0000000
                                                               0.0106383
            8
                 Adelie
                           Adelie
                                    0.9893617
                                                   0.0000000
                                                               0.0106383
```

```
      335 Chinstrap Chinstrap
      0.1034483
      0.8965517
      0.0000000

      336 Chinstrap Chinstrap
      0.1034483
      0.8965517
      0.0000000

      339 Chinstrap Chinstrap
      0.1034483
      0.8965517
      0.0000000
```

• Filter the result, keep only specified resampling iterations:

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

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

3.2.4. Custom resampling

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

[1] 1

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

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

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

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

3.2.5. Resampling with (predefined) groups

In some cases, it is desirable to keep observations together, i.e., either have all observations of the same group in the training set or have all observations of the same group in the test set. This can be defined through the column role "group" during Task creation, i.e., a column in the data specifies the groups (see also the help page on this column role).

A potential use case for this is spatial modeling, where one wants to account for groups of observations during resampling.



Besides the "group" column role, dedicated spatiotemporal resampling methods are available in mlr3spatiotempcv. More general information about this topic can be found in the spatiotemporal resampling section.

If you want the ensure that the observations in the training and test set are similarly distributed as in the complete task, one or multiple columns can be used for stratification via the column role stratum. See also the mlr3gallery post on this topic for a practical introduction.

```
i In the old mlr package, grouping was called "blocking".
```

3.2.6. Plotting Resample Results

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

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

```
task task id
                                                learner
                                                            learner id
                         pima <LearnerClassifRpart[38]> classif.rpart
 1: <TaskClassif[50]>
 2: <TaskClassif[50]>
                         pima <LearnerClassifRpart[38]> classif.rpart
 3: <TaskClassif[50]>
                         pima <LearnerClassifRpart[38]> classif.rpart
                         pima <LearnerClassifRpart[38]> classif.rpart
 4: <TaskClassif[50]>
                         pima <LearnerClassifRpart[38]> classif.rpart
 5: <TaskClassif[50]>
                         pima <LearnerClassifRpart[38]> classif.rpart
6: <TaskClassif[50]>
 7: <TaskClassif[50]>
                         pima <LearnerClassifRpart[38]> classif.rpart
                         pima <LearnerClassifRpart[38]> classif.rpart
8: <TaskClassif[50]>
                         pima <LearnerClassifRpart[38]> classif.rpart
 9: <TaskClassif[50]>
                         pima <LearnerClassifRpart[38]> classif.rpart
10: <TaskClassif[50]>
5 variables not shown: [resampling, resampling_id, iteration, prediction, classif.auc]
```

```
# boxplot of AUC values across the 10 folds
library("mlr3viz")
autoplot(rr, measure = msr("classif.auc"))

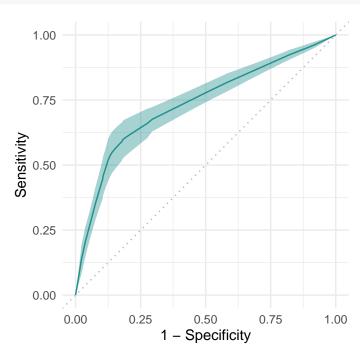
0.85

0.80

0.75

0.70
```

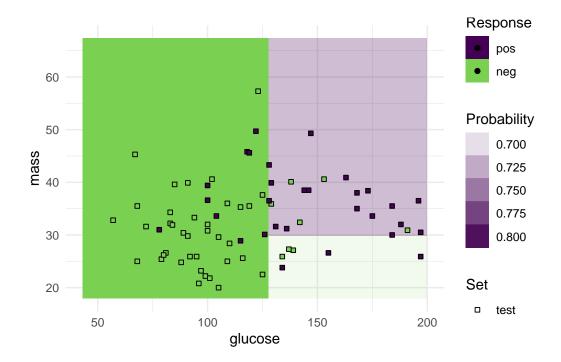
```
# ROC curve, averaged over 10 folds
autoplot(rr, type = "roc")
```



We can also plot the predictions of individual models:

```
# learner predictions for the first fold
rr1 = rr$clone()$filter(1)
autoplot(rr1, type = "prediction")
```

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



? Tip

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

3.3. Benchmarking

Comparing the performance of different learners on multiple tasks and/or different resampling schemes is a common task. This operation is usually referred to as "benchmarking" in the field of machine learning. The mlr3 package offers the benchmark() convenience function that takes care of most of the work of repeatedly training and evaluating models under the same conditions.

3.3.1. Design Creation

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

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

```
library("mlr3verse")
2
  design = benchmark_grid(
3
     tasks = tsks(c("spam", "german_credit", "sonar")),
     learners = lrns(c("classif.ranger", "classif.rpart", "classif.featureless"),
      predict_type = "prob", predict_sets = c("train", "test")),
     resamplings = rsmps("cv", folds = 3)
  print(design)
```

```
task
                                             learner
                                                             resampling
1: <TaskClassif[50]>
                          <LearnerClassifRanger[38]> <ResamplingCV[20]>
2: <TaskClassif[50]>
                           <LearnerClassifRpart[38]> <ResamplingCV[20]>
3: <TaskClassif[50]> <LearnerClassifFeatureless[38]> <ResamplingCV[20]>
4: <TaskClassif[50]>
                          <LearnerClassifRanger[38]> <ResamplingCV[20]>
5: <TaskClassif[50]>
                           <LearnerClassifRpart[38]> <ResamplingCV[20]>
6: <TaskClassif[50]> <LearnerClassifFeatureless[38]> <ResamplingCV[20]>
7: <TaskClassif[50]>
                          <LearnerClassifRanger[38]> <ResamplingCV[20]>
                           <LearnerClassifRpart[38]> <ResamplingCV[20]>
8: <TaskClassif[50]>
9: <TaskClassif[50]> <LearnerClassifFeatureless[38]> <ResamplingCV[20]>
```

The constructed design table can be passed to benchmark() to start the computation. It is also possible to construct a custom design manually, for example, to exclude certain task-learner combinations.



🌢 Danger

However, if you construct a custom design with data.table(), the train/test splits will be different for each row of the design if you do not manually instantiate the resampling before creating the design. See the help page on benchmark_grid() for an example.

3.3.2. Execution Aggregation of Results

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

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



Note that we did not have to instantiate the resampling manually. benchmark_grid() took care of it for us: each resampling strategy is instantiated once for each task during the construction of the exhaustive grid. This way, each learner operates on the same training and test sets which makes the results easier to compare.

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

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

tab = bmr$aggregate(measures)
print(tab[, .(task_id, learner_id, auc_train, auc_test)])
```

```
task_id
                          learner_id auc_train auc_test
1:
                      classif.ranger 0.9994318 0.9860031
            spam
2:
                       classif.rpart 0.9152765 0.9061875
            spam
            spam classif.featureless 0.5000000 0.5000000
3:
                      classif.ranger 0.9984109 0.8055811
4: german_credit
5: german_credit
                       classif.rpart 0.8073881 0.7110071
6: german_credit classif.featureless 0.5000000 0.5000000
                      classif.ranger 1.0000000 0.9142603
7:
           sonar
8:
                       classif.rpart 0.8953818 0.7588270
           sonar
           sonar classif.featureless 0.5000000 0.5000000
9:
```

We can aggregate the results even further. For example, we might be interested to know which learner performed best across all tasks. Simply aggregating the performances with the mean is usually not statistically sound. Instead, we calculate the rank statistic for each learner, grouped by task. Then the calculated ranks, grouped by the learner, are aggregated with the data.table package. As larger AUC scores are better, we multiply the values by -1 such that the best learner has a rank of 1.

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

```
task_id
                          learner_id rank_train rank_test
1:
            spam
                      classif.ranger
                                               2
2:
                       classif.rpart
                                                         2
            spam
3:
            spam classif.featureless
                                               3
                                                         3
                      classif.ranger
4: german_credit
                                               1
                                                         1
5: german_credit
                       classif.rpart
                                               2
                                                         2
6: german_credit classif.featureless
                                               3
                                                         3
7:
           sonar
                      classif.ranger
                                               1
                                                         1
                       classif.rpart
                                               2
                                                         2
8:
           sonar
           sonar classif.featureless
9:
                                               3
                                                         3
```

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

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

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

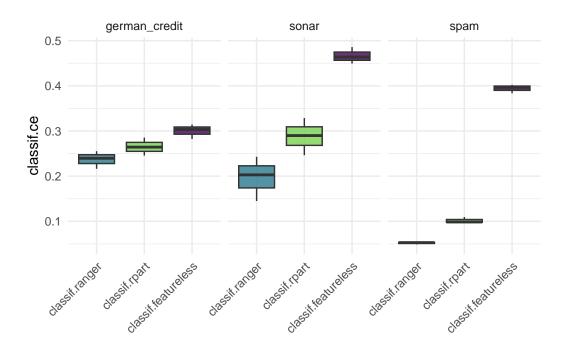
Unsurprisingly, the featureless learner is worse overall. The winner is the classification forest, which outperforms a single classification tree.

3.3.3. Plotting Benchmark Results

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

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

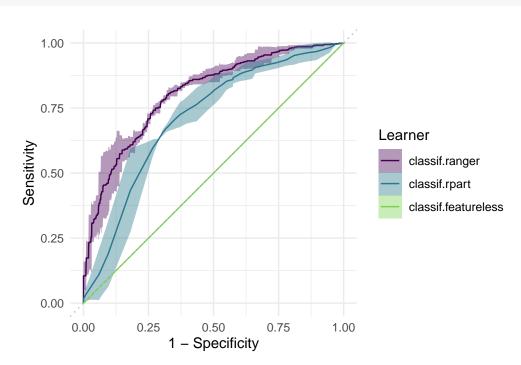
3. Performance Evaluation and Comparison



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

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

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



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

3.3.4. Statistical Tests

The package mlr3benchmark provides some infrastructure for applying statistical significance tests on BenchmarkResult. Currently, Friedman tests and pairwise Friedman-Nemenyi tests (Demšar 2006) are supported for benchmarks with at least two tasks and at least two learners. The results can be summarized in Critical Difference Plots.

```
library("mlr3benchmark")

bma = as.BenchmarkAggr(bmr, measures = msr("classif.auc"))
bma$friedman_posthoc()
```

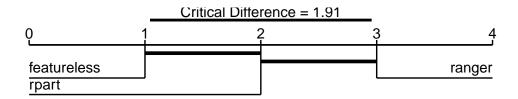
Pairwise comparisons using Nemenyi-Wilcoxon-Wilcox all-pairs test for a two-way balanced comparisons.

```
data: auc and learner_id and task_id
```

```
ranger rpart rpart 0.438 - featureless 0.038 0.438
```

P value adjustment method: single-step

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



3.3.5. Extracting ResampleResults

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

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

<ResampleResult> of 3 iterations

* Task: german_credit
* Learner: classif.ranger
* Warnings: 0 in 0 iterations
* Errors: 0 in 0 iterations

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

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

classif.auc 0.8055811

[1] 2 5 7 8 9 10

3.3.6. Converting and Merging

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

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

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

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

# Merge 2nd BMR into the first BMR
bmr = c(bmr1, bmr2)
print(bmr)
```

Most machine learning algorithms are configurated by a set of hyperparameters. The goal of hyperparameter optimization is to find the optimal hyperparameter configuration of a machine learning algorithm for a given task. This chapter presents an introduction to hyperparameter optimization in the mlr3 ecosystem. As a practical example, we optimize the cost and gamma hyperparameters of a support vector machine on the sonar task. We introduce the tuning instance class that describes the tuning problem and the tuner class that wraps an optimization algorithm. After running the optimization, we show how to analyze the results and fit a final model. We also show how to run a multi-objective optimization with multiple measures. Then we move on to more advanced topics like search space transformations, fallback learners and encapsulation. Finally, we show how to use nested resampling to get an unbiased estimate of the performance of an optimized model.

```
set.seed(4)
```

Machine learning algorithms usually include parameters and hyperparameters. Parameters are what we might think of as model coefficients or weights, when fitting a model we are essentially just running algorithms that fit parameters. In contrast, hyperparameters, are configured by the user and determine how the model will fit its parameters. Examples include setting the number of trees in a random forest, penalty variables in SVMs, or the learning rate in a neural network. Building a neural network is sometimes referred to as an 'art' as there are so many hyperparameters to configure that strongly influence model performance, this is also true for other machine learning algorithms. So in this chapter, we will demonstrate how to make this into more of a science.

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 numerical black-box optimization: an ML algorithm is configured with values chosen for one or more hyperparameters, this algorithm is then evaluated (optimally with a robust resampling method) and its performance measured, this is repeated with multiple configurations and the configuration with the best performance is selected. 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 the same underlying algorithm but with different hyperparameter configurations. For example, we could naively tune the number of trees in a random forest using basic mlr3 code:

```
bmr = benchmark(benchmark_grid(
tasks = tsk("penguins_simple"),
learners = list(
```

Hyperparamet

HPO: Hyperpa Optimization

```
lrn("classif.ranger", num.trees = 1, id = "1 tree"),
lrn("classif.ranger", num.trees = 10, id = "10 trees"),
lrn("classif.ranger", num.trees = 100, id = "100 trees")),
resamplings = rsmp("cv", folds = 3)
))
autoplot(bmr)
```

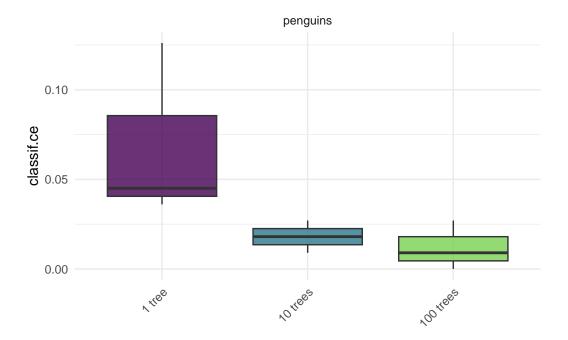


Figure 4.1.: In this code example we benchmark three random forest models with 1, 10, and 100 trees respectively, using 3-fold resampling, classification error loss, and tested on the simplified penguin dataset. The plot shows that the models with 10 and 100 trees are better performing across all three folds and 100 trees may be better than 10.

Human trial-and-error (which is essentially what we are doing above), 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 last few decades for robust and efficient HPO. Most HPO methods are iterative and propose different configurations until some termination criterion is met, at which point the optimal configuration is then returned (Figure 4.2). Popular, modern examples are given by algorithms based on evolutionary algorithms or Bayesian optimization methods. Recent HPO methods often also make use of evaluating a configuration at multiple so-called fidelity levels, e.g., a neural network can be trained for an increasing number of epochs, gradient boosting can be performed for an increasing number of boosting steps and training data can always be subsampled to only include a smaller fraction of all available data. The general idea of multi-fidelity HPO methods is that the performance of a model obtained by using computationally cheap lower fidelity evaluations (few numbers of epochs or boosting steps, only using a small sample of all available data for training) is predictive of the performance of the model obtained using computationally expensive higher fidelity evaluations 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). Another interesting direction of HPO is to optimize multiple metrics (Section 4.3) simultaneously, e.g., minimizing the generalization error along with the size of the model. This gives rise to multi-objective HPO. For more details on HPO in general, the reader is referred to Bischl et al. (2021) and Feurer and Hutter (2019).

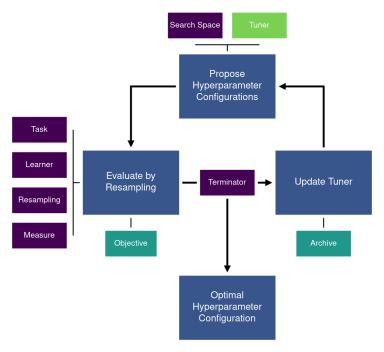


Figure 4.2.: Representation of the hyperparameter optimization loop in mlr3tuning. Blue - Hyperparameter optimization loop. Purple - Objects of the tuning instance supplied by the user. Blue-Green - Internally created objects of the tuning instance. Green - Optimization Algorithm.

4.1. Model Tuning

mlr3tuning is the hyperparameter optimization package of the mlr3 ecosystem. At the 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 a support vector machine (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 with a radial kernel and specify we want to tune this using "C-classification" (the alternative is "nu-classification", which

has the same underlying algorithm but with a nu parameter to tune over [0,1] instead of cost over $[0,\infty)$).

```
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)[, list(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
                                                   2
 6: decision.values ParamLgl
                                   NA
                                         NA
 7:
              degree ParamInt
                                    1
                                        Inf
                                                 Inf
 8:
             epsilon ParamDbl
                                    0
                                        Inf
                                                 Inf
 9:
              fitted ParamLgl
                                   NA
                                         NA
                                                   2
10:
               gamma ParamDbl
                                    0
                                        Inf
                                                 Inf
              kernel ParamFct
11:
                                  NA
                                         NA
                                                   4
                                -Inf
                                                 Inf
12:
                  nu ParamDbl
                                        Inf
13:
               scale ParamUty
                                   NA
                                         NA
                                                 Inf
14:
          shrinking ParamLgl
                                   NA
                                         NA
                                                   2
          tolerance ParamDbl
                                    0
15:
                                        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.

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-5, 1e5),
gamma = to_tune(1e-5, 1e5),
type = "C-classification",
kernel = "radial"
```

Search Space

```
6 )
7 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

The cost and gamma hyperparameters are usually tuned on the logarithmic scale. You can find out more in Section 4.2.2.

Search spaces are usually chosen by experience. In some cases these can be quite complex, Section 9.4 describes how to construct these. Section 4.2.3 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 known as specifying the tuning budget. mlr3tuning includes many methods to specify when to terminate an algorithm, which are known as Terminators. Available terminators are listed in Table 4.1.

Tuning Budget Terminators

Function call and default parameters
<pre>trm("evals", n_evals = 500)</pre>
<pre>trm("run_time", secs = 100)</pre>
<pre>trm("perf_reached", level = 0.1)</pre>
<pre>trm("stagnation", iters = 5, threshold = 1e-5)</pre>

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

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 ("run_time") or the number of evaluations ("evals"). Choosing a runtime is often based on practical considerations and intuition. Using a time limit can be important on clusters so that the tuning is finished before the account budget is exhausted. The "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 as if the level is set too optimistically, the tuning might never terminate. The "stagnation" terminator stops when no progress is made in a certain amount of iterations. Note, this could result in the optimization being terminated too early if the search space is too complex. We use "none" when tuners, such as Grid Search and Hyperband, control the termination themselves. Terminators can be freely combined with the "combo" terminator, this is explored in the exercises at the end of this chapter. A complete and always up-to-date list of terminators can be found on our website at https://mlr-org.com/terminators.html.

4.1.3. Tuning Instance with ti

uning Instance

A tuning instance can be constructed manually (Section 4.1.3) with the ti() function or automated (Section 4.1.6) 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. The ti function constructs a tuning instance which collects together the information required to optimise a model.

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 two measures to ti() would result in TuningInstanceMultiCrit (Section 4.3)). For this example we will use three-fold resampling and will optimise the classification error measure. Note that we use trm("none") as we are using an exhaustive grid search.

```
resampling = rsmp("cv", folds = 3)
measure = msr("classif.ce")
```

```
learner = lrn("classif.svm",
     cost = to_tune(1e-5, 1e5),
     gamma = to_tune(1e-5, 1e5),
     kernel = "radial",
     type = "C-classification"
   )
10
11
  instance = ti(
12
    task = tsk("sonar"),
13
     learner = learner,
14
     resampling = rsmp("cv", folds = 3),
     measures = msr("classif.ce"),
     terminator = trm("none")
   )
18
   instance
```

4.1.4. Tuner

After we created the tuning problem, we can look at *how* to tune. There are multiple **Tuners** in **Tuners** in **Tuners mlr3tuning**, which implement different HPO algorithms.

Tuner	Function call Method
Random Search	tnr("random_search'Samples configurations from a uniform distribution randomly
	(Bergstra and Bengio 2012).
Grid Search	tnr("grid_search") Discretizes the range of each configuration and exhaustively evaluates each combination.

Tuner	Function call	Method
Iterative Racing Bayesian Optimization	<pre>tnr("irace") tnr("mbo")</pre>	Races down a random set of configurations and uses the surviving ones to initialize a new set of configurations which focus on a promising region of the search space (López-Ibáñez et al. 2016). Iterative algorithms that make use of a continuously updated surrogate model built for the objective function. By optimizing a (comparably cheap to evaluate) acquisition function defined on the surrogate prediction, the next candidate is chosen for evaluation, resulting in good sample
Hyperband	tnr("hyperband")	efficiency. Multi-fidelity algorithm that speeds up a random search with adaptive resource allocation and early stopping (Li et al. 2017).
Covariance Matrix Adaptation Evolution Strategy	tnr("cmaes")	Evolution strategy algorithm with sampling from a multivariate Gaussian who is updated with the success of the previous population (Hansen and Auger 2011).
Generalized Simulated Annealing	tnr("gensa")	Auger 2011). Probabilistic algorithm for numeric search spaces (Xiang et al. 2013; Tsallis and Stariolo 1996).

Tuner	Function call	Method
Nonlinear Optimization	tnr("nloptr")	Several nonlinear optimization algorithms for numeric search spaces.

Table 4.2.: Tuning algorithms available in mlr3tuning, their function call and the methodology.

When selecting algorithms, grid search and random search are the most basic and are often selected first in initial experiments. They are 'naive' algorithms in that they try new configurations whilst ignoring performance from previous attempts. In contrast, more advanced algorithms such as Iterative Racing and CMA-ES learn from the previously evaluated configurations to find good configurations more quickly. Some advanced algorithms are included in extension packages, for example the package mlr3mbo implements Bayesian optimization (also called Model-Based Optimization), and mlr3hyperband implements algorithms of the hyperband family. A complete and up-to-date list of tuners can be found on the website.

For our SVM example, we will use a simple grid search with a resolution of 5, which is the distinct values to try *per hyperparameter*. For example for a search space $\{1, 2, 3, 4, 5, 6\}$ then a grid search with resolution 3 would pick three values evenly apart in this search space, i.e., $\{2, 4, 6\}$. The batch_size controls how many configurations are evaluated at the same time (see Section 9.1).

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

<TunerGridSearch>: Grid Search

- * Parameters: resolution=5, batch_size=5
- * Parameter classes: ParamLgl, ParamInt, ParamDbl, ParamFct
- * Properties: dependencies, single-crit, multi-crit
- * Packages: mlr3tuning

In our example we are tuning over two numeric parameters, 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 used to construct a model from the chosen learner, which is fit to the chosen task (Figure 4.2).

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.

Just like learners, tuners also have parameters, known as control parameters, which (as the name suggests) controls the behavior of the tuners. Unlike learners, default values for control parameters usually give good results and these rarely need to be changed. Control parameters are stored in the <code>param_set</code> field.

Control Param

```
tuner$param_set
```

<ParamSet>

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

4.1.5. Trigger the Tuning

Now 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 4.2).

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 and learner_param_vals optional constants (none in this example). See section Section 4.2.2 for more information.

4.1.6. Quick Tuning with tune

In the previous section, we looked at creating a tuning instance manually using ti(), which offers more control over the tuning process. However, you can also simplify this (albeit with slightly less control) using the tune() sugar function. Internally this creates a TuningInstanceSingleCrit, starts the tuning and returns the result with the instance.

```
learner = lrn("classif.svm",
     cost = to_tune(1e-5, 1e5),
     gamma = to_tune(1e-5, 1e5),
     kernel = "radial",
     type = "C-classification"
   )
6
   instance = tune(
     method = tnr("grid_search", resolution = 5, batch_size = 5),
9
     task = tsk("sonar"),
10
     learner = learner,
     resampling = rsmp("cv", folds = 3),
     measures = msr("classif.ce")
13
```

```
14 )
15
16 instance$result
```

4.1.7. Analyzing the Result

Whether you use ti or tune the output is the same and the 'archive' lists all evaluated hyperparameter configurations:

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

```
cost gamma classif.ce
1: 5.0e+04 5.0e+04 0.4661836
2: 5.0e+04 1.0e+05 0.4661836
3: 7.5e+04 7.5e+04 0.4661836
4: 1.0e+05 1.0e-05 0.2499655
5: 1.0e+05 7.5e+04 0.4661836
---
21: 1.0e-05 7.5e+04 0.4661836
22: 1.0e-05 1.0e+05 0.4661836
23: 2.5e+04 2.5e+04 0.4661836
24: 1.0e+05 2.5e+04 0.4661836
25: 1.0e+05 5.0e+04 0.4661836
```

Each row of the archive is a different evaluated configuration (there are 25 rows in total in the full data.table). The columns here show the tested configurations, the measure we optimize, the completed configuration time stamp, and the total train and predict times. If we only specify a single-objective criterium 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, how long did the model take to run? Were there any errors in running?

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

```
timestamp runtime_learners errors warnings
1: 2023-02-01 15:50:34
                                               0
                                   0.059
                                                        0
2: 2023-02-01 15:50:34
                                   0.065
                                               0
                                                        0
3: 2023-02-01 15:50:34
                                   0.071
                                               0
4: 2023-02-01 15:50:34
                                   0.052
                                               0
                                                        0
5: 2023-02-01 15:50:34
                                   0.068
                                                        0
```

```
21: 2023-02-01 15:50:36
                                    0.057
                                                0
                                                          0
22: 2023-02-01 15:50:36
                                    0.064
                                                0
                                                          0
23: 2023-02-01 15:50:36
                                    0.059
                                                0
                                                          0
24: 2023-02-01 15:50:36
                                    0.067
                                                0
                                                          0
25: 2023-02-01 15:50:36
                                                          0
                                    0.061
                                                0
```

Now we see not only was our optimal configuration the best performing with respect to classification error, but also it had the fastest runtime.

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

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

```
gamma classif.ce classif.fpr classif.fnr
 1: 5.0e+04 5.0e+04 0.4661836
                                 1.0000000
                                              0.000000
 2: 5.0e+04 1.0e+05 0.4661836
                                 1.0000000
                                              0.000000
 3: 7.5e+04 7.5e+04 0.4661836
                                 1.0000000
                                              0.000000
4: 1.0e+05 1.0e-05 0.2499655
                                              0.232703
                                 0.2628968
                                              0.000000
 5: 1.0e+05 7.5e+04 0.4661836
                                 1.0000000
21: 1.0e-05 7.5e+04 0.4661836
                                 1.0000000
                                              0.000000
22: 1.0e-05 1.0e+05 0.4661836
                                 1.0000000
                                              0.000000
23: 2.5e+04 2.5e+04 0.4661836
                                 1.0000000
                                              0.000000
24: 1.0e+05 2.5e+04 0.4661836
                                 1.0000000
                                              0.000000
25: 1.0e+05 5.0e+04 0.4661836
                                 1.0000000
                                              0.000000
```

Now we see our model is also the best performing with respect to FPR and FNR!

You can view 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).

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

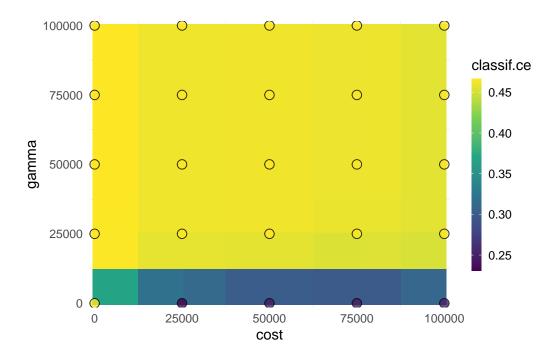


Figure 4.3.: Model performance with different configurations for cost and gamma. Bright yellow regions represent the model performing worse and dark blue performing better. We can see that high cost values and gamma values around exp(-5) achieve the best performance.

4.1.8. 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 with the optimal configurations:

```
svm_tuned = lrn("classif.svm", id = "SVM Tuned")
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$truth(), type = "C-classification",
   kernel = "radial", gamma = 1e-05, cost = 25000.0000075, probability = (self$predict_type == "prob"))
```

Parameters:

SVM-Type: C-classification

SVM-Kernel: radial 25000 cost:

Number of Support Vectors:



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 the model's performance. However, doing so would lead to bias and therefore nested resampling is required (Section 4.5). Therefore 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 4.4.

4.2. Advanced Tuning

4.2.1. Encapsulation and Fallback Learner

So far, we have only looked at the case where no issues occur. However, it often happens that learners with certain configurations do not converge, run out of memory, or terminate with an error. We can protect the tuning process from failing learners with encapsulation. The encapsulation separates the tuning from the training of the individual learner. The encapsulation method is set in the learner.

```
learner$encapsulate = c(train = "evaluate", predict = "evaluate")
```

The encapsulation can be set individually for training and predicting. There are currently two options for encapsulating a learner. The evaluate package and the callr package. The callr package comes with more overhead because the encapsulation spawns a separate R process. Both packages allow setting a timeout which is useful when a learner does not converge. We set a timeout of 30 seconds.

```
learner$timeout = 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 data or just the majority class. See ?@sec-fallback-learner for more detailed information.

The featureless learner predicts the most frequent label.

```
learner$fallback = lrn("classif.featureless")
```

Errors and warnings that occurred during tuning are stored in the archive.

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

	cost	gamma	classif.ce	errors	warnings
1:	5.0e+04	5.0e+04	0.4661836	0	0
2:	5.0e+04	1.0e+05	0.4661836	0	0
3:	7.5e+04	7.5e+04	0.4661836	0	0
4:	1.0e+05	1.0e-05	0.2499655	0	0
5:	1.0e+05	7.5e+04	0.4661836	0	0
21:	1.0e-05	7.5e+04	0.4661836	0	0
22:	1.0e-05	1.0e+05	0.4661836	0	0
23:	2.5e+04	2.5e+04	0.4661836	0	0
24:	1.0e+05	2.5e+04	0.4661836	0	0
25:	1.0e+05	5.0e+04	0.4661836	0	0

4.2.2. Advanced Search Spaces

rithm.

Usually, the cost and gamma hyperparameters are tuned on the logarithmic scale which means the optimization algorithm searches in [log(1e-5), log(1e5)] but transforms the selected configuration with exp() before passing 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.

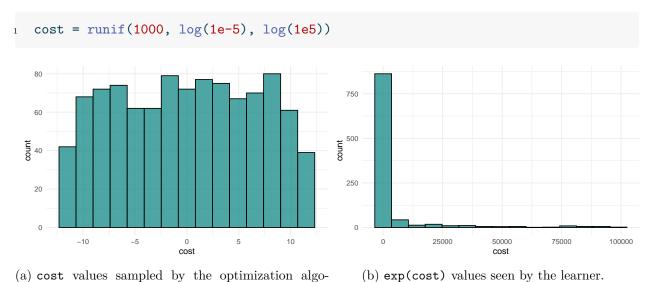


Figure 4.4.: Histogram of sampled cost values.

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"
   )
6
   instance = tune(
     method = tnr("grid_search", resolution = 5, batch_size = 5),
9
     task = tsk("sonar"),
10
     learner = learner,
11
     resampling = rsmp("cv", folds = 3),
     measures = msr("classif.ce")
13
   )
14
15
   instance$result
16
```

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
```

4.2.3. Search Spaces Collection

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
```

```
classif.glmnet.rbv2 Classification GLM with RandomBot classif.glmnet
 2:
 3:
      classif.kknn.default
                             Classification KKNN with Default
                                                                 classif.kknn
 4:
         classif.kknn.rbv2 Classification KKNN with RandomBot
                                                                 classif.kknn
 5: classif.ranger.default Classification Ranger with Default classif.ranger
20:
           regr.rpart.rbv2
                              Regression Rpart with RandomBot
                                                                   regr.rpart
21:
          regr.svm.default
                                  Regression SVM with Default
                                                                     regr.svm
22:
             regr.svm.rbv2
                                Regression SVM with RandomBot
                                                                     regr.svm
23:
      regr.xgboost.default
                              Regression XGBoost with Default
                                                                 regr.xgboost
24:
         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}.{publication}. The sugar function lts() is used to retrieve a TuningSpace.

TRUE

A tuning space can be passed to ti() as the search_space.

0.1

cp 1e-04

3:

```
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")
)
instance
```

```
<TuningInstanceSingleCrit>
* State: Not optimized
* Objective: <ObjectiveTuning:classif.rpart_on_sonar>
* Search Space:
                         lower upper nlevels
                class
1:
          cp ParamDbl -9.21034
                                    0
                                          Inf
   maxdepth ParamInt 1.00000
                                   30
                                           30
3: minbucket ParamInt
                       1.00000
                                  100
                                          100
4: minsplit ParamInt
                       1.00000
                                  100
                                          100
* Terminator: <TerminatorEvals>
```

Alternatively, we can explicitly set the search space of a learner with TuneTokens

```
vals = lts("classif.rpart.default")$values
2 vals
$minsplit
Tuning over:
range [2, 128] (log scale)
$minbucket
Tuning over:
range [1, 64] (log scale)
$cp
Tuning over:
range [1e-04, 0.1] (log scale)
learner = lrn("classif.rpart")
learner$param_set$set_values(.values = vals)
3 learner
<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
When passing a learner to lts(), the default search space from the Bischl et al. (2021) article is
applied.
1 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 **nrounds** hyperparameter in XGBoost:

```
1 lts("classif.xgboost.rbv2", nrounds = to_tune(1, 1024))
```

 $\verb| TuningSpace: classif.xgboost.rbv2>: Classification XGBoost with RandomBot| \\$

	id	lower	upper	levels	logscale	
1:	booster	NA	NA	<pre>gblinear,gbtree,dart</pre>	FALSE	
2:	nrounds	1e+00	1024		FALSE	
3:	eta	1e-04	1		TRUE	
4:	gamma	1e-05	7		TRUE	
5:	lambda	1e-04	1000		TRUE	
6:	alpha	1e-04	1000		TRUE	
7:	subsample	1e-01	1		FALSE	
8:	max_depth	1e+00	15		FALSE	
9:	min_child_weight	1e+00	100		TRUE	
10:	colsample_bytree	1e-02	1		FALSE	
11:	<pre>colsample_bylevel</pre>	1e-02	1		FALSE	
12:	rate_drop	0e+00	1		FALSE	
13:	skip_drop	0e+00	1		FALSE	
	Ship_diop		_		- 11202	

4.3. Multi-Objective Tuning

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. In a more complex example, consider the problem of deploying a classifier in a healthcare setting, there is clearly an ethical argument to tune the model to make the best possible predictions, however in machine learning this can often lead to models that are harder to interpret (think about deep neural networks!). In this case, we may be interested in minimizing both classification error (for example) and complexity.

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 are referred to as Pareto efficient and the set of all these configurations is referred to as the Pareto front (Figure 4.5).

Pareto Front

Multi-objective

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 Spam 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, logscale = TRUE),
    minsplit = to_tune(2, 128, logscale = TRUE),
    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.

```
instance = ti(
task = tsk("spam"),
learner = learner,
resampling = rsmp("cv", folds = 3),
measures = measures,
terminator = trm("evals", n_evals = 20),
store_models = TRUE # required to inspect selected_features
)
instance
```

```
<TuningInstanceMultiCrit>
* State: Not optimized
* Objective: <ObjectiveTuning:classif.rpart_on_spam>
* Search Space:
         id
               class
                          lower
                                    upper nlevels
         cp ParamDbl -9.2103404 -2.302585
1:
                                              Inf
2: minsplit ParamDbl 0.6931472 4.859812
                                              Inf
3: maxdepth ParamInt 1.0000000 30.000000
                                               30
* Terminator: <TerminatorEvals>
```

As before we will then select and run a tuning algorithm, here we use random search:

```
tuner = tnr("random_search", batch_size = 20)
tuner$optimize(instance)
```

Finally, we inspect the best-performing configurations, i.e., the Pareto set. And then inspect the estimated Pareto set and visualize the estimated Pareto front:

```
instance archive best()[, list(cp, minsplit, maxdepth, classif.ce, selected_features)]
```

	cp	minsplit	maxdepth	classif.ce	selected_features
4.	-	-	-		-
Ι:	-4.897655	3.338026	17	0.10302293	7.666667
2:	-7.051424	3.896918	12	0.10041637	12.333333
3:	-4.245193	1.541127	8	0.10780375	5.666667
4:	-2.774911	3.148014	30	0.16387611	2.333333
5:	-4.807622	1.987267	11	0.10432672	7.333333
6:	-4.853549	4.375677	19	0.10671726	6.000000
7:	-3.739909	1.772185	26	0.11193239	5.000000
8:	-4.937744	1.286328	29	0.10302293	7.666667
9:	-4.553218	2.750349	22	0.10563050	6.666667
10:	-3.714380	2.555973	25	0.11193239	5.000000
11:	-5.076189	2.735166	11	0.10085053	9.333333
12:	-4.519566	4.470367	30	0.10845564	5.333333
13:	-5.445322	1.663845	23	0.08715871	13.000000
14:	-6.465093	1.536021	16	0.07759311	29.000000
15:	-6.528747	2.155507	6	0.08585422	16.333333

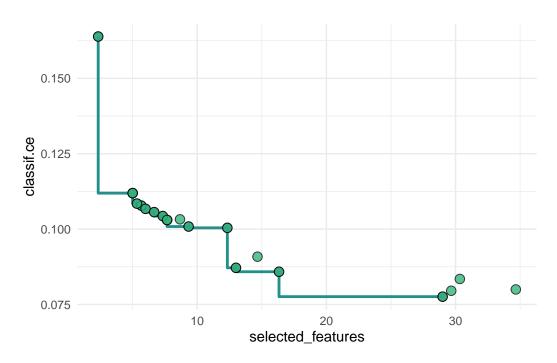


Figure 4.5.: Pareto front of selected features and classification error. Black dots represent tested configurations, each red dot individually represents a Pareto-optimal configuration and all red dots together represent the Pareto front.

4.4. Automated Tuning with AutoTuner

One of the most powerful 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 extract information on the best hyperparameter settings 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"
   )
6
   at = auto tuner(
     method = tnr("grid_search", resolution = 5, batch_size = 5),
     learner = learner,
10
     resampling = rsmp("cv", folds = 3),
11
     measure = msr("classif.ce")
12
   )
13
14
  at
15
```

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

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

classif.ce 0.2608696

We could also pass the AutoTuner to resample() and benchmark(), which would result in a nested resampling (Section 4.5), discussed next.

4.5. Nested Resampling

Hyperparameter optimization generally requires an additional layer or resampling to prevent bias in tuning. 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. This bias is represented in Figure 4.6 which shows an algorithm being tuned on data that has been split intro training and testing data, and then the same data is used to estimate the model performance after selecting the best configuration after HPO.

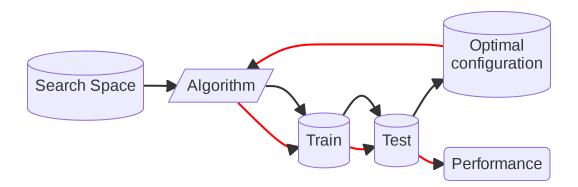


Figure 4.6.: Illustration of biased tuning. An algorithm is tuned by training on the Train dataset and then the optimal configuration is selected by evaluation on the Test data. The model's performance is then evaluated with the optimal configuration on the same data.

Nested resampling separates model optimization from the process of estimating the performance of the model by adding an additional layer of resampling, i.e., whilst model performance is estimated using a resampling method in the 'usual way', tuning is then performed by resampling the resampled data (Figure 4.7). 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

Nested Resam

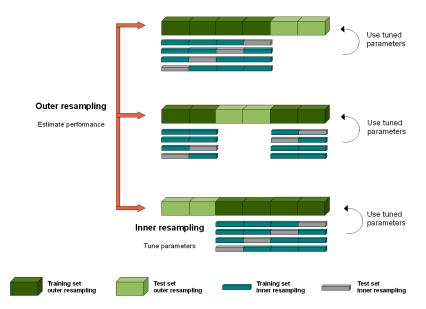


Figure 4.7.: An illustration of nested resampling. The green blocks represent 3-fold coss-validation for the outer resampling for model evaluation and the blue and gray blocks represent 4-fold cross-validation for the inner resampling for HPO.

produces many hyperparameter configurations which should not be used to construct a final model (Simon 2007).

In words this process runs as follows:

- 1. Outer resampling Instantiate 3-fold cross-validation to create different testing and training data sets.
- 2. Inner resampling Within the training data instantiate 4-fold cross-validation to create different inner testing and training data sets.
- 3. HPO Tune the hyperparameters using the inner data splits (blue and gray blocks).
- 4. Training Fit the learner on the outer training data set using the optimal hyperparameter configuration obtained from the inner resampling (dark green blocks).
- 5. Evaluation Evaluate the performance of the learner on the outer testing data (light green blocks).
- 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.

That is enough theory for now, let us take a look at how this works in mlr3.

4.5.1. Nested Resampling with AutoTuner

Nested resampling in mlr3 becomes quite simple with the AutoTuner (Section 4.4). 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"
   )
6
   at = auto_tuner(
     method = tnr("grid_search", resolution = 5, batch_size = 5),
     learner = learner,
10
     resampling = rsmp("cv", folds = 4),
11
     measure = msr("classif.ce"),
12
     term_evals = 20,
13
   )
14
15
   task = tsk("sonar")
   outer_resampling = rsmp("cv", folds = 3)
17
18
   rr = resample(task, at, outer resampling, store models = TRUE)
19
20
21
   rr
```

<ResampleResult> of 3 iterations

* Task: sonar * Learner: classif.svm.tuned

* Warnings: 0 in 0 iterations * Errors: 0 in 0 iterations

Note we set store_models = TRUE so that the AutoTuner models are stored 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 controlling the process (Section 9.1.4).

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

iteration cost gamma classif.ce

```
1: 1 11.512925 -5.756463  0.1663866

2: 2 5.756463 -5.756463  0.2090336

3: 3 5.756463 -5.756463  0.1941176

1 extract_inner_tuning_archives(rr)[,

2 list(iteration, cost, gamma, classif.ce)]
```

```
iteration
                     cost
                                gamma classif.ce
 1:
            1 -11.512925 -11.512925
                                       0.4567227
 2:
             1 -11.512925
                                       0.4567227
                             5.756463
 3:
               -5.756463 -11.512925
             1
                                       0.4567227
 4:
                 0.000000
             1
                             5.756463
                                       0.4567227
 5:
                 5.756463 -11.512925
                                       0.2747899
            3
                 0.000000 -11.512925
56:
                                       0.4678571
57:
            3
                 0.000000
                           -5.756463
                                       0.2151261
58:
            3
                 0.000000
                                       0.4678571
                            5.756463
59:
            3
                 5.756463
                            0.000000
                                       0.4678571
60:
               11.512925
                           -5.756463
                                       0.1941176
```

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 greatly between the resampling iterations. On the one hand, this could be due to the optimization algorithm used, for example, with simple algorithms like random search, we do not expect stability of hyperparameters. On the other hand, more advanced methods like irace converge to an optimal hyperparameter configuration. Another reason for instability in hyperparameters could be due to small data sets and/or a low number of resampling iterations (i.e., the usual small data high variance problem).

4.5.2. 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)[,
    list(iteration, cost, gamma, classif.ce)]
   iteration
                  cost
                            gamma classif.ce
1:
           1 11.512925 -5.756463
                                   0.1663866
              5.756463 -5.756463
2:
                                   0.2090336
3:
              5.756463 -5.756463
                                   0.1941176
  rr$score()[,
    list(iteration, classif.ce)]
```

```
iteration classif.ce
1: 1 0.1285714
2: 2 0.1014493
3: 3 0.1884058
```

Significantly lower predictive performances on the outer resampling indicate that the models with the optimized hyperparameters overfit the data.

It is therefore important to ensure that the performance of a tuned model is *always* reported as the aggregated performance of all outer resampling iterations, which is an unbiased estimate of future model performance. Note here we use the term *unbiased* to refer only to the statistical procedure of the performance estimation. The underlying prediction of the model could still be biased e.g. due to a bias in the data set.

```
rr$aggregate()
```

```
classif.ce 0.1394755
```

As a final note, nested resampling is computationally expensive, as a simple example using five outer folds and three inner folds with a grid search of resolution 5 used to tune 2 parameters, results in 5.35*5 = 375 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 9.1).

4.6. Conclusion

In this chapter, we learned how to optimize a model using tuning instances, about different tuners and terminators, 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 4.3 alongside their R6 classes. If you are interested in learning more about the underlying R6 classes to gain finer control of these methods, then take a look at the online API.

S3 function	R6 Class	Summary	
tnr()	Tuner	Determines an optimisation algorithm	
trm()	Terminator	Controls when to terminate the tuning algorithm	
ti()	TuningInstanceS StglesCtrining settings and save results		
	or		
	TuningInstanceMultiCrit		
<pre>paradox::to_tune()</pre>	paradox::TuneTolSetts which parameters in a learn		
		tune and over what search space	
<pre>auto_tuner()</pre>	AutoTuner	Automates the tuning process	

S3 function	R6 Class	Summary
<pre>extract_inner_tuning_results()</pre>	-	Extracts inner results from nested resampling
<pre>extract_inner_tuning_archives()</pre>	-	Extracts inner archives from nested resampling

Table 4.3.: 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 cheatsheet¹ summarizes the most important functions of mlr3tuning and the mlr3 gallery² features a collection of case studies and demonstrations about optimization, most notably learn how to:

- Apply advanced methods in the practical tuning series³.
- Optimize an rpart classification tree with only a few lines of code⁴.
- Tune an XGBoost model with early stopping⁵.
- Quickly load and tune over search spaces that have been published in literature with mlr3tuningspaces⁶.

4.7. 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.

¹https://cheatsheets.mlr-org.com/mlr3tuning.pdf

²https://mlr-org.com/gallery.html#category:tuning

³https://mlr-org.com/gallery.html#category:practical tuning series

⁴https://mlr-org.com/gallery/2022-11-10-hyperparameter-optimization-on-the-palmer-penguins/

⁵https://mlr-org.com/gallery/2022-11-04-early-stopping-with-xgboost/

⁶https://mlr-org.com/gallery/2021-07-06-introduction-to-mlr3tuningspaces/

5. Feature Selection

TODO (150-200 WORDS)

5.1. Introduction

Feature selection, also known as variable or descriptor selection, is the process of finding a subset of features to use with a given model. Using an optimal set of features can have several benefits:

- improved 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, and
- no need to collect potentially expensive features.

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 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).

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 the 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). In the Filters section, 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.

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

5. Feature Selection

feature subsets in a random search. Below, in the Wrapper Methods section, 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.

For this chapter, the reader should know the basic concepts of mlr3, i.e. know about tasks and learners. Basics about performance evaluation, i.e. resampling and benchmarking are helpful but not strictly necessary. In this section, we mostly focus on feature selection as a means of improving model performance.

5.1.1. Further Reading

- 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.
- Filters can be optimized with Hyperparameter Optimization.

5.2. Filters

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. We can distinguish between several types of filters, univariate and multivariate filters. While univariate filter methods only consider the relationship between each feature and the target variable, multivariate filters can take interactions with other features into account. A benefit of univariate filters is, that they are usually computationally cheaper than more complex filter or wrapper methods. 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. 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 values for each feature.

There is a list of all implemented filter methods in the Appendix.

5.2.1. 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, to calculate an information gain filter:

```
library("mlr3verse")
filter = flt("information_gain")

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
```

5.2.2. 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 is in the Appendix.

For some learners, the desired filter method needs to be set during learner creation. For example, learner classif.ranger comes with multiple integrated methods, c.f. the help page of

ranger::ranger(). To use the feature importance method "impurity", select it during learner
construction:

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

Now you 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
```

5.2.3. Embedded Methods

Another alternative are embedded methods. Many learners internally select a subset of the features which they find helpful for prediction. These subsets can usually be queried, as the following example demonstrates:

```
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
2: flipper_length
3:
      bill_length
       bill_depth
4:
                        0
5:
                        0
               sex
6:
                        0
              year
7:
        body_mass
                        0
```

Contrary to other filter methods, embedded methods just return value of 1 (selected features) and 0 (dropped feature).

5.2.4. 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, 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"
```

Note that 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.

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

5. Feature Selection

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 the Pipelines Chapter, 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 to the filter, i.e. tune the filter threshold to optimize the feature selection regarding prediction performance:

```
# combine filter with learner
graph = po("filter", filter = flt("information_gain")) %>>%
po("learner", lrn("classif.rpart"))
learner = as_learner(graph)

# tune how many feature to include
library("mlr3tuning")
```

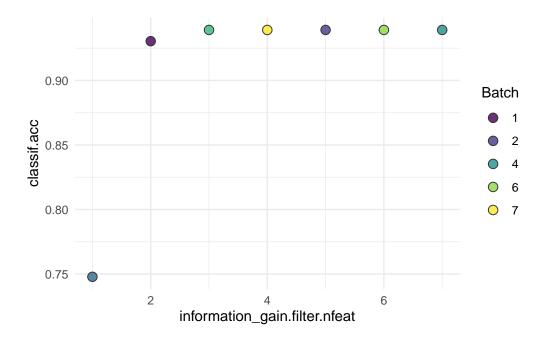
Loading required package: paradox

```
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

```
# plot tuning results
autoplot(instance)
```

5. Feature Selection



For more details, see the Pipelines and Hyperparameter Optimization chapters.

5.3. Wrapper Methods

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 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 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.

5.3.1. Simple Forward Selection Example

We start with the simple example from above and do sequential forward selection with the penguins data:

```
library("mlr3fselect")
2
```

```
instance = fselect(
method = "sequential",
task = tsk("penguins"),
learner = lrn("classif.rpart"),
resampling = rsmp("holdout"),
measure = msr("classif.acc")
)
```

To show all analyzed feature subsets and the corresponding performance, use:

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

```
bill_depth bill_length body_mass flipper_length island
1:
        TRUE
                   FALSE
                             FALSE
                                           FALSE FALSE FALSE
2:
       FALSE
                    TRUE
                             FALSE
                                           FALSE FALSE FALSE
3:
       FALSE
                   FALSE
                              TRUE
                                           FALSE FALSE FALSE
4:
       FALSE
                   FALSE
                             FALSE
                                            TRUE FALSE FALSE FALSE
5:
       FALSE
                   FALSE
                             FALSE
                                           FALSE
                                                   TRUE FALSE FALSE
6:
       FALSE
                   FALSE
                             FALSE
                                           FALSE FALSE TRUE FALSE
```

7 variables not shown: [classif.acc, runtime_learners, timestamp, batch_nr, warnings, errors,

And to only show the best feature set:

```
instance$result_feature_set
```

```
[1] "bill_length" "flipper_length" "island"
```

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.

5.3.2. 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. 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.

5.3.3. 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 (FSelectorRandomSearch)
- Exhaustive Search (FSelectorExhaustiveSearch)
- Sequential Search (FSelectorSequential)
- Recursive Feature Elimination (FSelectorRFE)
- Design Points (FSelectorDesignPoints)
- Genetic Search (FSelectorGeneticSearch)
- Shadow Variable Search (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")
```

5.3.4. 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)
```

```
bill_depth bill_length body_mass flipper_length island sex year

1: FALSE TRUE FALSE TRUE TRUE TRUE TRUE

2 variables not shown: [features, classif.acc]
```

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.
- 4. The **Terminator** is queried if the budget is exhausted. 1 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 (\$result_feature_set) and the corresponding measured performance (\$result_y) can be accessed from the instance:

```
instance$result_feature_set

[1] "bill_length" "flipper_length" "island" "sex"

[5] "year"

classif.acc
   0.9478261
```

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 and can be accessed by using as.data.table():

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

```
bill_depth bill_length body_mass flipper_length island
                                                             sex
                                                                  year
1:
        FALSE
                    FALSE
                              FALSE
                                               TRUE FALSE FALSE FALSE
2:
                     TRUE
                                               TRUE
                                                            TRUE
        FALSE
                              FALSE
                                                      TRUE
3:
         TRUE
                    FALSE
                              FALSE
                                              FALSE FALSE FALSE
4:
         TRUE
                     TRUE
                               TRUE
                                               TRUE FALSE FALSE
5:
        FALSE
                    FALSE
                              FALSE
                                               TRUE
                                                      TRUE FALSE FALSE
6:
        FALSE
                    FALSE
                              FALSE
                                               TRUE FALSE FALSE FALSE
```

7 variables not shown: [classif.acc, runtime_learners, timestamp, batch_nr, warnings, errors,

5. Feature Selection

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. Note that 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 is required.

5.3.5. 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 lowest classification error and lowest time to train the model. The full list of performance measures can be found here.

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 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 = 20)
)
```

The function fsi creates an instance of FSelectInstanceMultiCrit if more than one measure is selected. We now create an FSelector, e.g. using random search, 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.

```
fselector = fs("random_search")
fselector$optimize(instance)
```

```
bill_depth bill_length body_mass flipper_length island sex year

1: TRUE FALSE TRUE FALSE FALSE TRUE TRUE

2: TRUE TRUE TRUE TRUE TRUE FALSE TRUE FALSE

3 variables not shown: [features, classif.acc, time_train]
```

As above, the best feature subset (\$result_feature_set) and the corresponding measured performance (\$result_y) 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.

```
instance$result_feature_set
[[1]]
[1] "bill_depth" "body_mass"
                                              "year"
[[2]]
[1] "bill_depth"
                      "bill_length"
                                        "body_mass"
                                                           "flipper_length"
[5] "sex"
  instance$result_y
   classif.acc time train
1:
     0.7826087
                     0.027
2:
     0.9391304
                     0.027
```

More generally, 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.

5.3.6. 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. Analogously to the previous subsection, a new classification tree ("classif.rpart) learner is created. This classification tree learner is then wrapped in a random search feature selector, which automatically starts a feature selection on the given task using an inner resampling (holdout), 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().

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

at = auto_fselector(
   method = fs("random_search"),
   learner = learner,
   resampling = rsmp("holdout"),
   measure = msr("classif.acc"),
   terminator = trm("evals", n_evals = 10)
```

twoclass, weights

```
9 )
10 at

<AutoFSelector:classif.rpart.fselector>
  * Model: list
  * Packages: mlr3, mlr3fselect, rpart
  * Predict Type: response
  * Feature Types: logical, integer, numeric, factor, ordered
```

* Properties: importance, missings, multiclass, selected_features,

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 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 classification tree lrn("classif.rpart"). For that, we create a benchmark grid with the task, the learners and a 3-fold cross validation.

```
grid = benchmark_grid(
task = tsk("penguins"),
learner = list(at, lrn("classif.rpart")),
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)]</pre>
```

Because of the implicit feature selection in classification trees, we do not expect huge performance improvements by the the feature selection in this example.

6. Pipelines

TODO (150-200 WORDS)

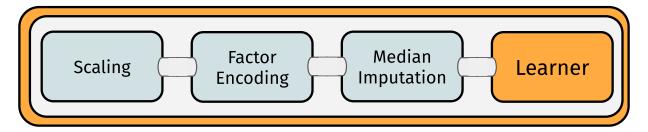
Note

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)

6. Pipelines

• Alternative path branching (data going one specific way, controlled by hyperparameters)

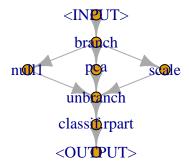
An extensive introduction to creating custom **PipeOps** (PO's) can be found in the technical introduction.

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)
```

```
label
               key
 1:
            boxcox
                            Box-Cox Transformation of Numeric Features
 2:
            branch
                                                         Path Branching
 3:
             chunk
                                      Chunk Input into Multiple Outputs
 4: classbalancing
                                                        Class Balancing
                                               Majority Vote Prediction
        classifavg
60:
         threshold Change the Threshold of a Classification Prediction
                     Tune the Threshold of a Classification Prediction
61:
    tunethreshold
62:
          unbranch
                                               Unbranch Different Paths
63:
                                        Interface to the vtreat Package
            vtreat
64:
        yeojohnson
                        Yeo-Johnson Transformation of Numeric Features
9 variables not shown: [packages, tags, feature_types, input.num, output.num, input.type.train
```

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 "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:

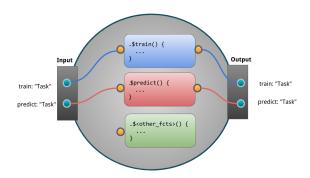
```
filter = po("filter",
filter = mlr3filters::flt("variance"),
param_vals = list(filter.frac = 0.5))
```

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. Pipelines



6.2. The Pipeline Operator: %>>%

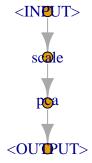
It is possible to create intricate **Graphs** with edges going all over the place (as long as no loops are introduced). Irrespective, there is usually a clear direction of flow between "layers" in the **Graph**. It is therefore convenient to build up a **Graph** from layers.

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 Graphs.

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.

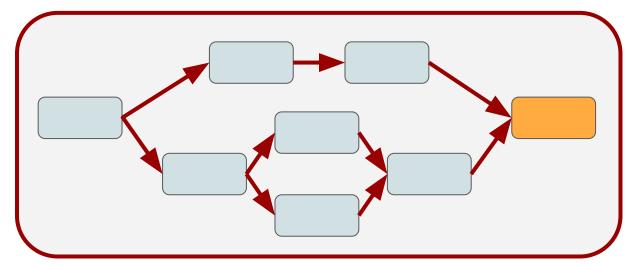
```
mutate = po("mutate")

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 Graphs.

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

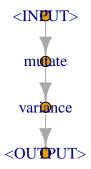
To illustrate how this sugar operator works under the surface we will include an example of the manual way (= hard way) to construct a **Graph**. This is done by creating an empty graph first. Then one fills the empty graph with POs, and connects edges between the POs. Conceptually, this may look like 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:

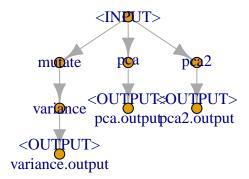
```
graph$plot()
```



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:

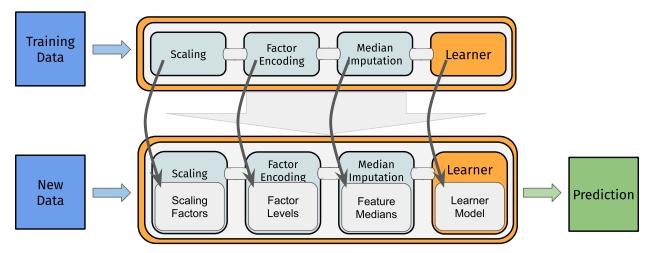
```
graph$plot()
graph$add_pipeop(po("pca", id = "pca2"))
graph$add_pipeop(po("pca"))
```



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

row_ids

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

```
$classif.rpart.output
```

 $\PredictionClassif> for 150 observations:$

truth response

```
1 setosa setosa
2 setosa setosa
3 setosa setosa
---
148 virginica virginica
149 virginica virginica
150 virginica virginica
```

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

```
glrn = as_learner(graph)
```

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

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

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

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

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$optimize(instance)
   classif.rpart.cp variance.filter.frac learner_param_vals x_domain
         0.04886918
                                0.6870548
                                                   <list[5]> <list[2]>
1 variable not shown: [classif.ce]
The tuning result can be found in the respective result slots.
instance$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.

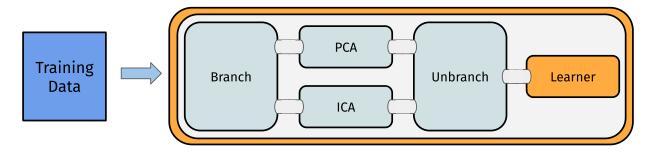
Possible types are:

- 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.

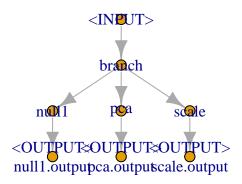
In the following we first create the branched graph and then show what happens if the "unbranching" is not applied:

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

Without "unbranching" one creates the following graph:

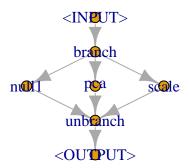
6. Pipelines

```
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:

```
State
      ID
                                              prdcssors
                              sccssors
 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

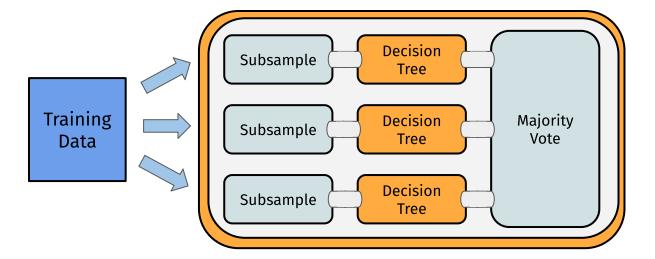
We first examine Bagging introduced by (Breiman 1996). The basic idea is to create multiple predictors and then aggregate those to a single, more powerful predictor.

"... 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:

6. Pipelines



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():

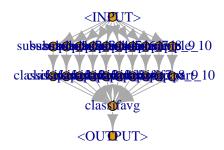
```
pred_set = ppl("greplicate", single_pred, 10L)
```

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:

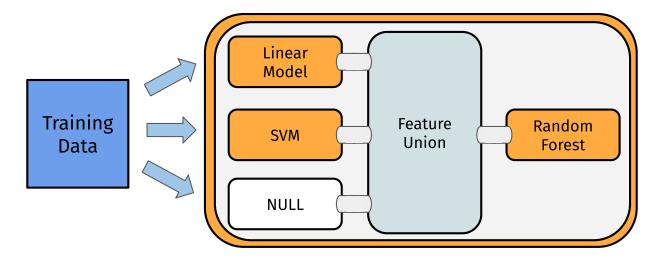
row_ids	truth	response	${\tt prob.setosa}$	${\tt prob.versicolor}$	<pre>prob.virginica</pre>
16	setosa	setosa	1	0.0	0.0
18	setosa	setosa	1	0.0	0.0
21	setosa	setosa	1	0.0	0.0
129	virginica	virginica	0	0.0	1.0
134	virginica	${\tt versicolor}$	0	0.5	0.5
149	virginica	virginica	0	0.0	1.0

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.

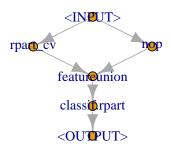
```
level_0 = gunion(list(lrn_0, po("nop")))
```

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	wirginica	virginica

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.

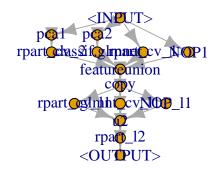
In this case, we train a glmnet and 2 different rpart models (some transform its inputs using PipeOpPCA) on our task in the "level 0" and concatenate them with the original features (via gunion). The result is then passed on to "level 1", where we copy the concatenated features 3 times and put this task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

6. Pipelines

"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
11
   lrn_2 = po("pca", id = "pca2") %>>% po("learner_cv", glmn)
13
   # Union them with a PipeOpNULL to keep original features
14
   level_0 = gunion(list(lrn_0, lrn_1, lrn_2, po("nop", id = "NOP1")))
15
16
   # Cbind the output 3 times, train 2 learners but also keep level
17
  # 0 predictions
18
   level_1 = level_0 %>>%
19
     po("featureunion", 4) %>>%
20
     po("copy", 3) %>>%
21
     gunion(list(
22
       po("learner_cv", rprt, id = "rpart_cv_l1"),
23
       po("learner_cv", glmn, id = "glmnt_cv_l1"),
24
       po("nop", id = "NOP_11")
25
     ))
26
   # Cbind predictions, train a final learner
   level_2 = level_1 %>>%
29
     po("featureunion", 3, id = "u2") %>>%
30
     po("learner", rprt, id = "rpart_12")
31
32
  # Plot the resulting graph
34 level_2$plot(html = FALSE)
```



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

```
1 lrn$
2 train(task, train.idx)$
3 predict(task, test.idx)$
4 score()
```

classif.ce 0.1

6.6. Adding new PipeOps

This section showcases how the mlr3pipelines package can be extended to include custom PipeOps. To run the following examples, we will need a Task; we are using the well-known "Iris" task:

```
library("mlr3")
task = tsk("iris")
task$data()
```

```
Species Petal.Length Petal.Width Sepal.Length Sepal.Width

1: setosa 1.4 0.2 5.1 3.5

2: setosa 1.4 0.2 4.9 3.0
```

6. Pipelines

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

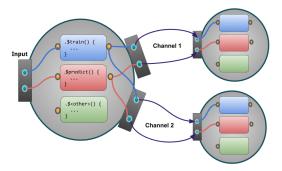
mlr3pipelines is fundamentally built around R6. When planning to create custom PipeOp objects, it can only help to familiarize yourself with it.

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 inand 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") {
```

```
},
6
      ),
7
      private == list(
8
         .train = function(inputs) {
9
10
        },
12
         .predict = function(inputs) {
13
14
15
      )
16
   )
17
```

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,
10
       output = output
11
     )
12
   }
13
```

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:

```
1 .train = function(inputs) {
2    self$state = list()
3    c(inputs, inputs)
4 }

1 .predict = function(inputs) {
2    c(inputs, inputs)
3 }
```

6.6.1.4. Putting it Together

The whole definition thus becomes

```
PipeOpCopyTwo = R6::R6Class("PipeOpCopyTwo",
     inherit = mlr3pipelines::PipeOp,
2
     public = list(
3
       initialize = function(id = "copy.two") {
4
         super$initialize(id,
5
           input = data.table::data.table(name = "input", train = "*", predict = "*"),
6
           output = data.table::data.table(name = c("output1", "output2"),
                                train = "*", predict = "*")
         )
       }
10
     ),
11
     private = list(
12
       .train = function(inputs) {
13
```

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

redict = function(inputs) {
c(inputs, 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:

```
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

copy.two <<UNTRAINED>>

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

When inheriting from PipeOpTaskPreproc, one must either implement the private methods .train_task() and .predict_task(), or the methods .train_dt(), .predict_dt(), depending on whether wants to operate on a Task object or on its data as data.tables. In the second

case, one can optionally also overload the .select_cols() method, which chooses which of the incoming Task's features are given to the .train_dt() / .predict_dt() functions.

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

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 <code>.predict_task()</code> 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, <code>mlr3</code> expects a <code>Learner</code> to always return just as many predictions as it was given input rows, so a <code>PipeOp</code> that removes <code>Task</code> rows during training can not be used inside a <code>GraphLearner</code>.

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,
2
     public = list(
3
       initialize = function(id = "drop.na") {
         super$initialize(id)
       }
     ),
     private = list(
a
       .train_task = function(task) {
10
         self$state = list()
11
         featuredata = task$data(cols = task$feature_names)
         exclude = apply(is.na(featuredata), 1, any)
13
         task$filter(task$row_ids[!exclude])
14
       },
15
16
       .predict_task = function(task) {
17
```

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
                        5.1
                                                  5.9
                                                               3.0
5: virginica
                                     1.8
```

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$initialize(id = id)
       }
     ),
     private = list(
9
        .select_cols = function(task) {
10
          task$feature_types[type == "numeric", id]
       },
12
13
        .train_dt = function(dt, levels, target) {
14
          sc = scale(as.matrix(dt))
15
          self$state = list(
16
            center = attr(sc, "scaled:center"),
17
            scale = attr(sc, "scaled:scale")
          )
19
         sc
20
       },
21
22
        .predict_dt = function(dt, levels) {
23
          t((t(dt) - self$state$center) / self$state$scale)
24
       }
25
     )
26
   )
```

(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:
                -1.3357516 -1.3110521 -1.13920048 -0.13153881
       setosa
  3:
                -1.3923993 -1.3110521 -1.38072709 0.32731751
       setosa
  4:
                -1.2791040 -1.3110521 -1.50149039 0.09788935
       setosa
  5:
                -1.3357516 -1.3110521 -1.01843718 1.24503015
       setosa
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 PipeOps that perform mostly the same operation during training and prediction. The point of Task preprocessing is often to modify the training data in mostly the same way as prediction data (but in a way that *may* depend on training data).

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

A simplification to make is therefore to have a 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,
2
     public = list(
       initialize = function(id = "drop.const") {
         super$initialize(id = id)
       }
     ),
     private = list(
9
       .get_state = function(task) {
10
         data = task$data(cols = task$feature_names)
         nonconst = sapply(data, function(column) length(unique(column)) > 1)
12
         list(cnames = colnames(data)[nonconst])
13
       },
14
15
       .transform = function(task) {
16
         task$select(self$state$cnames)
17
       }
     )
19
20
```

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
                                               4.9
2: setosa
                     1.4
                                 0.2
                                                            3.0
3: setosa
                     1.3
                                 0.2
                                               4.7
                                                            3.2
4: setosa
                     1.5
                                 0.2
                                               4.6
                                                            3.1
                     1.4
                                 0.2
                                               5.0
                                                            3.6
5: setosa
  gr = Graph$new()$add_pipeop(PipeOpDropConst$new())
  dropped_task = gr$train(irishead)[[1]]
4 dropped_task$data()
   Species Petal.Length Sepal.Length Sepal.Width
1: setosa
                     1.4
                                  5.1
                                               3.5
                     1.4
                                  4.9
                                               3.0
2: setosa
3: setosa
                     1.3
                                  4.7
                                               3.2
4: setosa
                     1.5
                                  4.6
                                               3.1
                                               3.6
5: setosa
                     1.4
                                  5.0
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
```

Empty data.table (0 rows and 4 cols): Species, Petal. Length, Sepal. Length, Sepal. Width

\$outtaskshell

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

```
Species Petal.Length Sepal.Length Sepal.Width
                          1.4
                                        5.1
                                                      3.5
  1:
        setosa
  2:
        setosa
                          1.4
                                        4.9
                                                      3.0
  3:
                          1.3
                                        4.7
                                                      3.2
        setosa
  4:
                          1.5
                                        4.6
                                                      3.1
        setosa
  5:
                          1.4
                                        5.0
                                                      3.6
        setosa
                          5.2
                                        6.7
146: virginica
                                                      3.0
                                                      2.5
147: virginica
                          5.0
                                        6.3
                                        6.5
148: virginica
                          5.2
                                                      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,
2
     public = list(
       initialize = function(id = "scale.always.simple") {
          super$initialize(id = id)
5
       }
     ),
7
     private = list(
9
        .select_cols = function(task) {
10
          task$feature_types[type == "numeric", id]
11
       },
12
13
        .get_state_dt = function(dt, levels, target) {
14
         list(
15
            center = sapply(dt, mean),
16
            scale = sapply(dt, sd)
17
          )
       },
19
```

```
20
21    .transform_dt = function(dt, levels) {
22         t((t(dt) - self$state$center) / self$state$scale)
23     }
24    )
25    )
```

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
                -1.3357516 -1.3110521 -0.89767388
  1:
       setosa
                                                    1.01560199
  2:
       setosa -1.3357516 -1.3110521 -1.13920048 -0.13153881
               -1.3923993 -1.3110521 -1.38072709 0.32731751
  3:
       setosa
  4:
       setosa
                -1.2791040 -1.3110521 -1.50149039 0.09788935
  5:
               -1.3357516 -1.3110521 -1.01843718
                                                   1.24503015
       setosa
 ___
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
                                         0.06843254 -0.13153881
150: virginica
                 0.7602115
                             0.7880307
```

6.6.4. Hyperparameters

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

Hyperparameters are declared during initialization, when calling the PipeOp's \$initialize() function, by giving a param_set argument. The param_set must be a ParamSet from the paradox package; see the tuning chapter or Section 9.4 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:

```
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
                                                        TRUE
2:
           scale ParamLgl
                             NA
                                   NA
                                                        TRUE
          robust ParamLgl
                             NA
                                   NA
                                            2 <NoDefault[3] > FALSE
4: affect_columns ParamUty
                             NA
                                   NΑ
                                          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

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 chind 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)

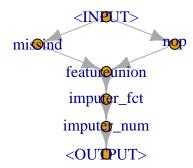
pou = po("featureunion")

# Impute categorical features by fitting a Learner ("classif.rpart") for each feature.

pof = po("imputelearner", lrn("classif.rpart"), id = "imputer_fct", affect_columns = selector.
```

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 0 0 0 0
missing_body_mass missing_flipper_length 0 0 0
year sex bill_depth 0 0
bill_length 0 0 0 0
bill_length 0 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
                    1.4
                                 0.2
                                              5.1
                                                           3.5
1: setosa
2: setosa
                    1.4
                                 0.2
                                              4.9
                                                           3.0
                    1.3
                                 0.2
                                              4.7
                                                           3.2
3: setosa
4: setosa
                    1.5
                                 0.2
                                              4.6
                                                           3.1
                    1.4
                                 0.2
                                              5.0
                                                           3.6
5: setosa
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
                    1.4
                                 0.2
                                               4.9
                                                           3.0
                                                                      7.9
2: setosa
3: setosa
                    1.3
                                 0.2
                                               4.7
                                                           3.2
                                                                      7.9
4: setosa
                    1.5
                                 0.2
                                               4.6
                                                           3.1
                                                                      7.7
   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.

6. Pipelines

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.

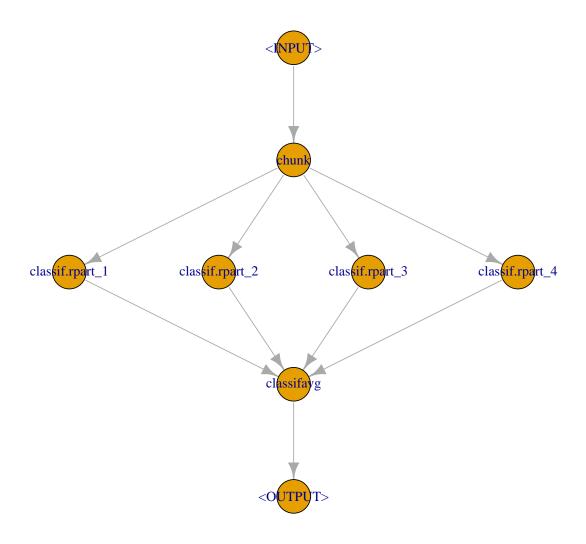
```
chks = po("chunk", 4)
lrns = ppl("greplicate", po("learner", lrn("classif.rpart")), 4)
```

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> of 1 iterations

* Task: iris

* Learner: classif.rpart
* Warnings: 0 in 0 iterations
* Errors: 0 in 0 iterations

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

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: PipeOps behave differently during a "training phase" and a "prediction phase". The training phase will typically generate a certain model of the data that is saved as internal state. The prediction phase will then operate on the input data depending on the trained model.

An example of this behavior is the *principal component analysis* operation ("PipeOpPCA"): During training, it will transform incoming data by rotating it in a way that leads to uncorrelated features ordered by their contribution to total variance. It will *also* save the rotation matrix to be 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()
```

```
PC3
                                                          PC4
       Species
                     PC1
                                 PC2
  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
```

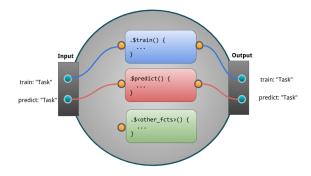
6. Pipelines

```
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()
               PC1
                          PC2
                                    PC3
                                                PC4
  Species
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 \times k) = (4 \times 4):
                                          PC3
                               PC2
                                                    PC4
                   PC1
Petal.Length 0.85667061 0.17337266 -0.07623608 0.4798390
Petal.Width
```

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:

Sepal.Length 0.36138659 -0.65658877 0.58202985 -0.3154872 Sepal.Width -0.08452251 -0.73016143 -0.59791083 0.3197231



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 PipeOps

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

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

	key	<pre>input.num</pre>	$\verb"output.num"$
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 PipeOps from the mlr_pipeops dictionary, it is also possible to give additional constructor arguments, such as an id or parameter values.

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

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

pofu = po("featureunion", innum = 2)

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

	Species	${\tt Petal.Length}$	${\tt Petal.Width}$	${\tt Sepal.Length}$	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:	${\tt virginica}$	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:

```
name train predict
1: input1 Task Task
```

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

2: input2 Task

Task

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

6.8.3.2. Output Channels

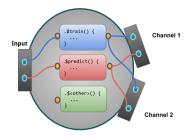
Unlike the typical notion of a function, PipeOps can also have multiple output channels. \$train() and \$predict() always return a list, so certain PipeOps may return lists with more than one element. Similar to input channels, the information about the number and type of outputs given by a PipeOp is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

```
po("chunk", outnum = 3)$output

name train predict
```

1: output1 Task Task 2: output2 Task Task 3: output3 Task Task

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



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:

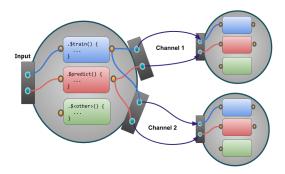
```
1 # po("backuplearner")
```

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)
```

```
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 PipeOps. The return value corresponds to the output of the PipeOp output channels that are not connected to other PipeOps.

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

```
Species Petal.Length
                             Petal.Width Sepal.Length Sepal.Width
1:
                                                        0.7861738
       setosa
               -1.27910398 -1.3110521482
                                         -1.01843718
2:
       setosa
               -1.22245633 -1.3110521482
                                          -1.25996379
                                                        0.7861738
3:
       setosa
              -1.50569459 -1.4422448248
                                          -1.86378030
                                                       -0.1315388
4:
              -1.16580868 -1.3110521482 -0.53538397
                                                        0.7861738
       setosa
```

```
5:
                -1.44904694 -1.3110521482
                                          -1.01843718
                                                         0.3273175
        setosa
 6:
        setosa
                -1.39239929 -1.3110521482 -1.74301699
                                                       -0.1315388
                                            0.67224905
 7: versicolor
                0.42032558 0.3944526477
                                                         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
                                                        -1.0492515
12: versicolor
                0.36367793 0.0008746178
                                          -0.41462067
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 introduced¹. 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 "feature-union".

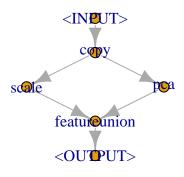
```
gr = Graph$new()$
add_pipeop(po("copy", outnum = 2))$
add_pipeop(po("scale"))$
add_pipeop(po("pca"))$
add_pipeop(po("featureunion", innum = 2))

gr$
```

¹It 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.

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

gr$plot(html = FALSE)
```



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

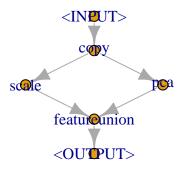
```
Species Petal.Length Petal.Width
                                            PC1
                                                         PC2
  1:
       setosa
               -1.3357516 -1.3110521 -2.561012 -0.006922191
       setosa -1.3357516 -1.3110521 -2.561012 -0.006922191
 2:
       setosa
  3:
              -1.3923993 -1.3110521 -2.653190 0.031849692
  4:
               -1.2791040 -1.3110521 -2.468834 -0.045694073
       setosa
               -1.3357516 -1.3110521 -2.561012 -0.006922191
  5:
       setosa
 ---
                 0.8168591
146: virginica
                             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 Graphs with edges going all over the place (as long as no loops are introduced), there is usually a clear direction of flow between "layers" in the Graph. It is therefore convenient to build up a Graph from layers, which can be done using the %>>% ("double-arrow") operator. It takes either a PipeOp or a Graph on each of its sides and connects all of the outputs of its left-hand side to one of the inputs each of its right-hand side—the number of inputs therefore must match the number of outputs. Together with the gunion() operation, which takes PipeOps or Graphs and arranges them next to each other akin to a (disjoint) graph union, the above network can more easily be constructed as follows:

```
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

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

```
po1 = po("scale")
po2 = po("scale")
```

```
3 po1 %>>% po2 # name clash
Error in gunion(list(g1, g2), in_place = c(TRUE, TRUE)): Assertion on 'ids of pipe operators of
po2$id = "scale2"
2 gr = po1 %>>% po2
3 gr
Graph with 2 PipeOps:
                State sccssors prdcssors
     ID
  scale <<UNTRAINED>> scale2
 scale2 <<UNTRAINED>>
                                  scale
# Alternative ways of getting new ids:
po("scale", id = "scale2")
PipeOp: <scale2> (not trained)
values: <robust=FALSE>
Input channels <name [train type, predict type]>:
  input [Task, Task]
Output channels <name [train type, predict type]>:
  output [Task, Task]
1 # sometimes names of PipeOps within a Graph need to be changed
gr2 = po("scale") %>>% po("pca")
3 gr %>>% gr2
Error in gunion(list(g1, g2), in_place = c(TRUE, TRUE)): Assertion on 'ids of pipe operators of
gr2$set_names("scale", "scale3")
2 gr %>>% gr2
Graph with 4 PipeOps:
                State sccssors prdcssors
  scale <<UNTRAINED>> scale2
 scale2 <<UNTRAINED>>
                        scale3
                                 scale
 scale3 <<UNTRAINED>>
                                 scale2
                           pca
    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> of 1 iterations

* Task: iris

* Learner: pca.classif.rpart
* Warnings: 0 in 0 iterations
* Errors: 0 in 0 iterations

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 Section 9.4.

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
           center ParamLgl
                                                2
                                                            TRUE
1:
                                      NA
2:
           scale. ParamLgl
                                NA
                                      NA
                                                2
                                                           FALSE
            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
```

6. Pipelines

\$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
             pca.center ParamLgl
                                           NA
                                                     2
                                                                 TRUE TRUE
1:
                                     NA
                                                     2
                                                                FALSE
2:
             pca.scale. ParamLgl
                                     NA
                                           NA
3:
              pca.rank. ParamInt
                                      1
                                          Inf
                                                   Inf
     pca.affect_columns ParamUty
                                                   Inf <Selector[1]>
4:
                                           NA
                                     NA
           scale.center ParamLgl
                                                     2
                                                                 TRUE
5:
                                     NA
                                           NA
                                                     2
6:
            scale.scale ParamLgl
                                     NA
                                           NA
                                                                 TRUE
7:
                                                     2 <NoDefault[3] > FALSE
           scale.robust ParamLgl
                                     NA
                                           NA
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:	ср	${\tt ParamDbl}$	0	1	Inf	0.01	
2:	keep_model	${\tt ParamLgl}$	NA	NA	2	FALSE	
3:	maxcompete	${\tt ParamInt}$	0	Inf	Inf	4	
4:	maxdepth	${\tt ParamInt}$	1	30	30	30	
5:	maxsurrogate	${\tt ParamInt}$	0	Inf	Inf	5	
6:	minbucket	${\tt ParamInt}$	1	Inf	Inf	<nodefault[3]></nodefault[3]>	
7:	minsplit	${\tt ParamInt}$	1	Inf	Inf	20	
8:	surrogatestyle	${\tt ParamInt}$	0	1	2	0	
9:	usesurrogate	${\tt ParamInt}$	0	2	3	2	
10:	xval	ParamInt	0	Inf	Inf	10	0

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

<ParamSetCollection>

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

1 variable not shown: [value]

glrn\$param_set

7. Preprocessing

TODO (150-200 WORDS)

TODO (150-200 WORDS)

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

Survival Analysis

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

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

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

Density Estimation

This sub-chapter explains how to conduct (unconditional) density estimation in mlr3. Density estimation is used to estimate the probability density function of a continuous variable. Unconditional density estimation is an unsupervised task, so there is no 'value' to predict. Instead, densities are estimated.

This sub-chapter explains how to estimate probability distributions for continuous variables using the mlr3proba extension package.

Spatiotemporal Analysis

Spatiotemporal analysis data observations entail reference information about spatial and temporal characteristics. One of the largest issues of spatiotemporal data analysis is the inevitable presence of auto-correlation in the data. Auto-correlation is especially severe in data with a marginal spatiotemporal variation. The sub-chapter on Spatiotemporal analysis provides instructions on how to account for spatiotemporal data.

Multilabel Classification

Multilabel classification deals with objects that can belong to more than one category at the same time. Numerous target labels are attributed to a single observation. Working with multilabel data requires one to use modified algorithms, to accommodate data specific characteristics. Two approaches to multilabel classification are prominently used:

- The problem transformation method
- The algorithm adaption method

Instructions on how to deal with multilabel classification inmlr3can be found in this sub-chapter.

Cost Sensitive Classification

This sub-chapter deals with the implementation of cost-sensitive classification. Regular classification aims to minimize the misclassification rate and thus all types of misclassification errors are deemed equally severe. Cost-sensitive classification is a setting where the costs caused by different kinds of errors are not assumed to be equal. The objective is to minimize the expected costs.

Analytical data for a big credit institution is used as a use case to illustrate the different features. Firstly, the sub-chapter provides guidance on how to implement a first model. Subsequently, the sub-chapter contains instructions on how to modify cost sensitivity measures, thresholding and threshold tuning.

Cluster Analysis

Cluster analysis aims to group data into clusters such that objects that are similar end up in the same cluster. Fundamentally, clustering and classification are similar. However, clustering is an unsupervised task because observations do not contain true labels while in classification, labels are needed in order to train a model.

This sub-chapter explains how to perform cluster analysis inmlr3with the help of mlr3cluster extension package.

Coming soon

In development we are also working on

- Multilabel classification mlr3multioutput
- Functional data analysis mlr3fda
- Ordinal analysis mlr3ordinal
- Time-series analysis / forecasting mlr3temporal

Send us a message if you want to help out on any of these!

8.1. Survival Analysis



Warning

Survival analysis with {mlr3proba} is currently in a fragile state after its removal from CRAN. Hence most code examples listed in this page will not work for the time being. We are actively working on a solution!

Survival analysis is a sub-field of supervised machine learning in which the aim is to predict the survival distribution of a given individual. Arguably the main feature of survival analysis is that, unlike classification and regression, learners are trained on two features:

1. the time until the event takes place

2. the event type: either censoring or death.

At a particular time-point, an individual is either: alive, dead, or censored. Censoring occurs if it is unknown if an individual is alive or dead. For example, say we are interested in patients in hospital and every day it is recorded if they are alive or dead, then after a patient leaves, it is unknown if they are alive or dead. Hence they are censored. If there was no censoring, then ordinary regression analysis could be used instead. Furthermore, survival data contains solely positive values and therefore needs to be transformed to avoid biases.

Note that survival analysis accounts for both censored and uncensored observations while adjusting respective model parameters.

The package mlr3proba (Sonabend et al. 2021) extends mlr3 with the following objects for survival analysis:

- TaskSurv to define (censored) survival tasks
- LearnerSurv as base class for survival learners
- PredictionSurv as specialized class for Prediction objects
- MeasureSurv as specialized class for performance measures

For a good introduction to survival analysis see *Modelling Survival Data in Medical Research* (Collett 2014).

8.1.1. TaskSurv

Unlike TaskClassif and TaskRegr which have a single 'target' argument, TaskSurv mimics the survival::Surv object and has three to four target arguments (dependent on censoring type). A TaskSurv can be constructed with the function as task_surv():

```
library("mlr3")
   library("mlr3proba")
   library("survival")
   as_task_surv(survival::bladder2[, -1L], id = "interval_censored",
     time = "start", event = "event", time2 = "stop", type = "interval")
   # type = "right" is default
   task = as_task_surv(survival::rats, id = "right_censored",
     time = "time", event = "status", type = "right")
10
11
   print(task)
12
13
   # the target column is a survival object:
14
   head(task$truth())
15
16
   # kaplan-meier plot
17
  # library("mlr3viz")
  autoplot(task)
```

8.1.2. Predict Types - crank, lp, and distr

Every PredictionSurv object can predict one or more of:

- 1p Linear predictor calculated as the fitted coefficients multiplied by the test data.
- distr Predicted survival distribution, either discrete or continuous. Implemented in distr6.
- crank Continuous risk ranking.
- response Predicted survival time.

lp and crank can be used with measures of discrimination such as the concordance index. Whilst lp is a specific mathematical prediction, crank is any continuous ranking that identifies who is more or less likely to experience the event. So far the only implemented learner that only returns a continuous ranking is surv.svm. If a PredictionSurv returns an lp then the crank is identical to this. Otherwise crank is calculated as the expectation of the predicted survival distribution. Note that for linear proportional hazards models, the ranking (but not necessarily the crank score itself) given by lp and the expectation of distr, is identical.

The example below uses the rats task shipped with mlr3proba.

```
task = tsk("rats")
learn = lrn("surv.coxph")

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

learn$train(task, row_ids = train_set)
prediction = learn$predict(task, row_ids = test_set)

print(prediction)
```

<PredictionSurv> for 60 observations:

8.1.3. Composition

Finally we take a look at the PipeOps implemented in mlr3proba, which are used for composition of predict types. For example, a predict linear predictor does not have a lot of meaning by itself, but it can be composed into a survival distribution. See mlr3pipelines for full tutorials and details on PipeOps.

```
library("mlr3pipelines")
2 library("mlr3learners")
  # PipeOpDistrCompositor - Train one model with a baseline distribution,
4 # (Kaplan-Meier or Nelson-Aalen), and another with a predicted linear predictor.
  task = tsk("rats")
  # remove the factor column for support with glmnet
7 task$select(c("litter", "rx"))
  learner_lp = lrn("surv.glmnet")
  learner_distr = lrn("surv.kaplan")
  prediction_lp = learner_lp$train(task)$predict(task)
   prediction_distr = learner_distr$train(task)$predict(task)
   prediction_lp$distr
13
   # Doesn't need training. Base = baseline distribution. ph = Proportional hazards.
14
15
   pod = po("compose_distr", form = "ph", overwrite = FALSE)
16
   prediction = pod$predict(list(base = prediction_distr, pred = prediction_lp))$output
17
   # Now we have a predicted distr!
20
   prediction$distr
21
22
   # This can all be simplified by using the distrcompose pipeline
23
24
   glm.distr = ppl("distrcompositor", learner = lrn("surv.glmnet"),
25
     estimator = "kaplan", form = "ph", overwrite = FALSE, graph_learner = TRUE)
   glm.distr$train(task)$predict(task)
```

8.1.4. Benchmark Experiment

Finally, we conduct a small benchmark study on the **rats** task using some of the integrated survival learners:

```
library("mlr3learners")

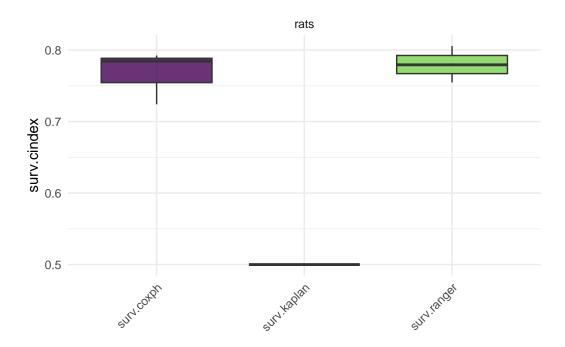
task = tsk("rats")

# some integrated learners
learners = lrns(c("surv.coxph", "surv.kaplan", "surv.ranger"))
print(learners)
```

```
$surv.coxph
<LearnerSurvCoxPH:surv.coxph>: Cox Proportional Hazards
* Model: -
* Parameters: list()
```

```
* Packages: mlr3, mlr3proba, survival, distr6
* Predict Types: crank, [distr], lp
* Feature Types: logical, integer, numeric, factor
* Properties: weights
$surv.kaplan
<LearnerSurvKaplan:surv.kaplan>: Kaplan-Meier Estimator
* Model: -
* Parameters: list()
* Packages: mlr3, mlr3proba, survival, distr6
* Predict Types: [crank], distr
* Feature Types: logical, integer, numeric, character, factor, ordered
* Properties: missings
$surv.ranger
<LearnerSurvRanger:surv.ranger>: Random Forest
* Model: -
* Parameters: num.threads=1
* Packages: mlr3, mlr3proba, mlr3extralearners, ranger
* Predict Types: crank, [distr]
* Feature Types: logical, integer, numeric, character, factor, ordered
* Properties: importance, oob_error, weights
# Harrell's C-Index for survival
measure = msr("surv.cindex")
g print(measure)
<MeasureSurvCindex:surv.cindex>
* Packages: mlr3, mlr3proba
* Range: [0, 1]
* Minimize: FALSE
* Average: macro
* Parameters: weight_meth=I, tiex=0.5, eps=0.001
* Properties: -
* Predict type: crank
* Return type: Score
set.seed(1)
bmr = benchmark(benchmark_grid(task, learners, rsmp("cv", folds = 3)))
3 bmr$aggregate(measure)
           resample_result task_id learner_id resampling_id iters surv.cindex
1: 1 <ResampleResult[21]> rats surv.coxph
                                                       cv
                                                              3 0.7671307
                                                               3 0.5000000
2: 2 <ResampleResult[21]> rats surv.kaplan
                                                        cv
3: 3 <ResampleResult[21]> rats surv.ranger
                                                        cv 3 0.7799153
```

autoplot(bmr, measure = measure)



The experiment indicates that both the Cox PH and the random forest have better discrimination than the Kaplan-Meier baseline estimator, but that the machine learning random forest is not consistently better than the interpretable Cox PH.

8.2. Density Estimation



Warning

Survival analysis with {mlr3proba} is currently in a fragile state after its removal from CRAN. Hence most code examples listed in this page will not work for the time being. We are actively working on a solution!

Density estimation is the learning task to find the unknown distribution from which an i.i.d. data set is generated. We interpret this broadly, with this distribution not necessarily being continuous (so may possess a mass not density). The conditional case, where a distribution is predicted conditional on covariates, is known as 'probabilistic supervised regression', and will be implemented in mlr3proba in the near-future. Unconditional density estimation is viewed as an unsupervised task. For a good overview to density estimation see Density estimation for statistics and data analysis (Silverman 1986).

The package mlr3proba extends mlr3 with the following objects for density estimation:

- TaskDens to define density tasks
- LearnerDens as base class for density estimators

- PredictionDens as specialized class for Prediction objects
- MeasureDens as specialized class for performance measures

In this example we demonstrate the basic functionality of the package on the faithful data from the datasets package. This task ships as pre-defined TaskDens with mlr3proba.

Unconditional density estimation is an unsupervised method. Hence, TaskDens is an unsupervised task which inherits directly from Task unlike TaskClassif and TaskRegr. However, TaskDens still has a target argument and a \$truth field defined by:

- target the name of the variable in the data for which to estimate density
- \$truth the values of the target column (which is *not* the true density, which is always unknown)

8.2.1. Train and Predict

Density learners have train and predict methods, though being unsupervised, 'prediction' is actually 'estimation'. In training, a distr6 object is created, see here for full tutorials on how to access the probability density function, pdf, cumulative distribution function, cdf, and other important fields and methods. The predict method is simply a wrapper around self\$model\$pdf and if available self\$model\$cdf, i.e. evaluates the pdf/cdf at given points. Note that in prediction the points to evaluate the pdf and cdf are determined by the target column in the TaskDens object used for testing.

```
# create task and learner

task_faithful = TaskDens$new(id = "eruptions", backend = datasets::faithful$eruptions)

learner = lrn("dens.hist")
```

```
6 # train/test split
7 train_set = sample(task_faithful$nrow, 0.8 * task_faithful$nrow)
8 test_set = setdiff(seq_len(task_faithful$nrow), train_set)
10 # fitting KDE and model inspection
learner$train(task_faithful, row_ids = train_set)
12 learner$model
$distr
Histogram()
$hist
$breaks
 [1] 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5
$counts
 [1] 39 30 5 4 25 62 49 3
$density
  \hbox{\tt [1]} \ \ 0.35944700 \ \ 0.27649770 \ \ 0.04608295 \ \ 0.03686636 \ \ 0.23041475 \ \ 0.57142857 \ \ 0.45161290 
 [8] 0.02764977
$mids
 [1] 1.75 2.25 2.75 3.25 3.75 4.25 4.75 5.25
$xname
 [1] "dat"
$equidist
 [1] TRUE
attr(,"class")
 [1] "histogram"
attr(,"class")
 [1] "dens.hist"
class(learner$model)
 [1] "dens.hist"
# make predictions for new data
prediction = learner$predict(task_faithful, row_ids = test_set)
```

Every PredictionDens object can estimate:

• pdf - probability density function

Some learners can estimate:

• cdf - cumulative distribution function

8.2.2. Benchmark Experiment

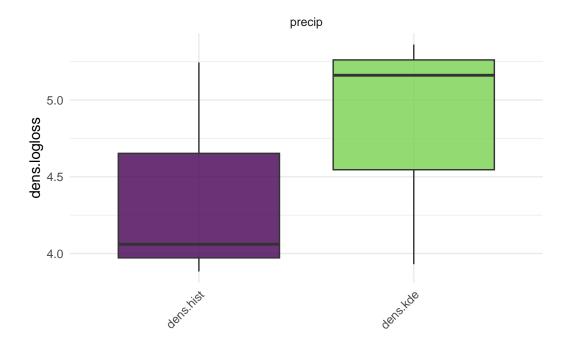
Finally, we conduct a small benchmark study on the **precip** task using some of the integrated survival learners:

```
1 # some integrated learners
learners = lrns(c("dens.hist", "dens.kde"))
g print(learners)
$dens.hist
<LearnerDensHistogram:dens.hist>: Histogram Density Estimator
* Model: -
* Parameters: list()
* Packages: mlr3, mlr3proba, distr6
* Predict Types: [pdf], cdf, distr
* Feature Types: integer, numeric
* Properties: -
$dens.kde
<LearnerDensKDE:dens.kde>: Kernel Density Estimator
* Model: -
* Parameters: kernel=Epan, bandwidth=silver
* Packages: mlr3, mlr3proba, distr6
* Predict Types: [pdf], distr
* Feature Types: integer, numeric
* Properties: missings
1 # Logloss for probabilistic predictions
measure = msr("dens.logloss")
3 print(measure)
<MeasureDensLogloss:dens.logloss>: Log Loss
* Packages: mlr3, mlr3proba
* Range: [0, Inf]
* Minimize: TRUE
* Average: macro
* Parameters: eps=1e-15
* Properties: -
* Predict type: pdf
```

```
set.seed(1)
bmr = benchmark(benchmark_grid(task, learners, rsmp("cv", folds = 3)))
bmr$aggregate(measure)
```

```
nr resample_result task_id learner_id resampling_id iters dens.logloss
1: 1 <ResampleResult[21]> precip dens.hist cv 3 4.396138
2: 2 <ResampleResult[21]> precip dens.kde cv 3 4.817715
```

```
autoplot(bmr, measure = measure)
```



The results of this experiment show that the sophisticated Penalized Density Estimator does not outperform the baseline histogram, but that the Kernel Density Estimator has at least consistently better (i.e. lower logloss) results.

8.3. Spatiotemporal Analysis

Data observations may entail reference information about spatial or temporal characteristics. Spatial information is stored as coordinates, usually named "x" and "y" or "lat"/"lon". Treating spatiotemporal data using non-spatial data methods can lead to over-optimistic performance estimates. Hence, methods specifically designed to account for the special nature of spatiotemporal data are needed.

In the mlr3 framework, the following packages relate to this field:

• mlr3spatiotempcv (biased-reduced performance estimation)

• mlr3spatial (spatial prediction support)

The following (sub-)sections introduce the potential pitfalls of spatiotemporal data in machine learning and how to account for it. Note that not all functionality will be covered, and that some of the used packages are still in early lifecycles. If you want to contribute to one of the packages mentioned above, please contact Patrick Schratz.

8.3.1. Creating a spatial Task

To make use of spatial resampling methods, a {mlr3} task that is aware of its spatial characteristic needs to be created. Two Task child classes exist in {mlr3spatiotempcv} for this purpose:

- TaskClassifST
- TaskRegrST

To create one of these, you have multiple options:

- 1. Use the constructor of the Task directly via \$new() this only works for data.table backends (!)
- 2. Use the as_task_* converters (e.g. if your data is stored in an sf object)

We recommend the latter, as the as_task_* converters aim to make task construction easier, e.g., by creating the DataBackend (which is required to create a Task in {mlr3}) automatically and setting the crs and coordinate_names fields. Let's assume your (point) data is stored in with an sf object, which is a common scenario for spatial analysis in R.

```
# create 'sf' object
data_sf = sf::st_as_sf(ecuador, coords = c("x", "y"), crs = 32717)

# create `TaskClassifST` from `sf` object
task = as_task_classif_st(data_sf, id = "ecuador_task", target = "slides", positive = "TRUE"
```

You can also use a plain data.frame. In this case, crs and coordinate_names need to be passed along explicitly as they cannot be inferred directly from the sf object:

```
task = as_task_classif_st(ecuador, id = "ecuador_task", target = "slides",
positive = "TRUE", coordinate_names = c("x", "y"), crs = 32717)
```

The *ST task family prints a subset of the coordinates by default:

```
TaskClassifST:ecuador_task> (751 x 11)
* Target: slides
* Properties: twoclass
* Features (10):
  - dbl (10): carea, cslope, dem, distdeforest, distroad, distslidespast, hcurv, log.carea, slope, vcurv
```

* Coordinates:

```
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
```

All *ST tasks can be treated as their super class equivalents TaskClassif or TaskRegr in subsequent {mlr3} modeling steps.

8.3.2. Autocorrelation

Data which includes spatial or temporal information requires special treatment in machine learning (similar to survival, ordinal and other task types listed in the special tasks chapter). In contrast to non-spatial/non-temporal data, observations inherit a natural grouping, either in space or time or in both space and time (Legendre 1993). This grouping causes observations to be autocorrelated, either in space (spatial autocorrelation (SAC)), time (temporal autocorrelation (TAC)) or both space and time (spatiotemporal autocorrelation (STAC)). For simplicity, the acronym STAC is used as a generic term in the following chapter for all the different characteristics introduced above.

What effects does STAC have in statistical/machine learning?

The overarching problem is that STAC violates the assumption that the observations in the train and test datasets are independent (Hastie, Friedman, and Tibshirani 2001). If this assumption is violated, the reliability of the resulting performance estimates, for example retrieved via cross-validation, is decreased. The magnitude of this decrease is linked to the magnitude of STAC in the dataset, which cannot be determined easily.

One approach to account for the existence of STAC is to use dedicated resampling methods. mlr3spatiotempcv provides access to the most frequently used spatiotemporal resampling methods. The following example showcases how a spatial dataset can be used to retrieve a bias-reduced performance estimate of a learner.

The following examples use the **ecuador** dataset created by Jannes Muenchow. It contains information on the occurrence of landslides (binary) in the Andes of Southern Ecuador. The landslides were mapped from aerial photos taken in 2000. The dataset is well suited to serve as an example because it it relatively small and of course due to the spatial nature of the observations. Please refer to Muenchow, Brenning, and Richter (2012) for a detailed description of the dataset.

To account for the spatial autocorrelation probably present in the landslide data, we will make use one of the most used spatial partitioning methods, a cluster-based k-means grouping (Brenning 2012), (spcv_coords in mlr3spatiotempcv). This method performs a clustering in 2D space which

contrasts with the commonly used random partitioning for non-spatial data. The grouping has the effect that train and test data are more separated in space as they would be by conducting a random partitioning, thereby reducing the effect of STAC.

By contrast, when using the classical random partitioning approach with spatial data, train and test observations would be located side-by-side across the full study area (a visual example is provided further below). This leads to a high similarity between train and test sets, resulting in "better" but biased performance estimates in every fold of a CV compared to the spatial CV approach. However, these low error rates are mainly caused due to the STAC in the observations and the lack of appropriate partitioning methods and not by the power of the fitted model.

8.3.3. Spatiotemporal Cross-Validation and Partitioning

One part of spatiotemporal machine learning is dealing with the spatiotemporal components of the data during performance estimation. Performance is commonly estimated via cross-validation and mlr3spatiotemporv provides specialized resamplings methods for spatiotemporal data. The following chapters showcases how these methods can be applied and how they differ compared to non-spatial resampling methods, e.g. random partitioning. In addition, examples which show how resamplings with spatial information can be visualized using mlr3spatiotempore.

Besides performance estimation, prediction on spatiotemporal data is another challenging task. See Section 8.3.6 for more information about how this topic is handled within the mlr3 framework.

8.3.3.1. Spatial CV vs. Non-Spatial CV

In the following a spatial and non-spatial CV will be applied to showcase the mentioned performance differences.

The performance of a simple classification tree ("classif.rpart") is evaluated on a random partitioning (repeated_cv) with four folds and two repetitions. The chosen evaluation measure is "classification error" ("classif.ce").

For the spatial example, repeated_spcv_coords is chosen whereas repeated_cv represents the non-spatial example.

Note

The selection of repeated_spcv_coords in this example is arbitrary. For your use case, you might want to use a different spatial partitioning method (but not necessarily!). Have a look at the "Getting Started" vignette of mlr3spatiotempcv to see all available methods and choose one which fits your data and its prediction purpose.

8.3.3.1.1. Non-Spatial CV

In this example the **ecuador** example task is taken to estimate the performance of an **rpart** learner with fixed parameters on it.

⚠ Warning

In practice you usually might want to tune the hyperparameters of the learner in this case and apply a nested CV in which the inner loop is used for hyperparameter tuning.

```
library("mlr3")
   library("mlr3spatiotempcv")
   set.seed(42)
   # be less verbose
   lgr::get_logger("bbotk")$set_threshold("warn")
   lgr::get_logger("mlr3")$set_threshold("warn")
   task = tsk("ecuador")
10
   learner = lrn("classif.rpart", maxdepth = 3, predict_type = "prob")
11
   resampling_nsp = rsmp("repeated_cv", folds = 4, repeats = 2)
   rr_nsp = resample(
13
     task = task, learner = learner,
14
     resampling = resampling_nsp)
15
16
   rr_nsp$aggregate(measures = msr("classif.ce"))
```

classif.ce 0.3388575

8.3.3.1.2. Spatial CV

```
task = tsk("ecuador")

learner = lrn("classif.rpart", maxdepth = 3, predict_type = "prob")

resampling_sp = rsmp("repeated_spcv_coords", folds = 4, repeats = 2)

rr_sp = resample(

task = task, learner = learner,

resampling = resampling_sp)

rr_sp$aggregate(measures = msr("classif.ce"))
```

classif.ce 0.412529

Here, the performance of the classification tree learner is around 7 percentage points worse when using Spatial Cross-Validation (SpCV) compared to Non-Spatial Cross-Validation (NSpCV). The resulting difference in performance is variable as it depends on the dataset, the magnitude of STAC

and the learner itself. For algorithms with a higher tendency of overfitting to the training set, the difference between the two methods will be larger.

Now, what does it mean that the performance in the spatial case is worse? Should you ditch SpCV and keep using non-spatial partitioning? The answer is **NO**. The reason why the spatial partitioning scenario results in a lower predictive performance is because throughout the CV the model has been trained on data that is less similar than the test data compared against the non-spatial scenario. Or in other words: in the non-spatial scenario, train and test data are almost identical (due to spatial autocorrelation).

This means that the result from the SpCV setting is more close to the true predictive power of the model - whereas the result from non-spatial CV is overoptimistic and biased.

Note

The result of the SpCV setting is by no means the absolute truth - it is also biased, but (most often) less compared to the non-spatial setting.

8.3.4. Visualization of Spatiotemporal Partitions

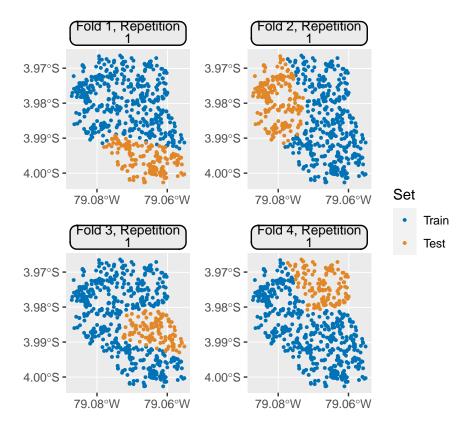
Every partitioning method in mlr3spatiotempcv comes with S3 methods for plot() and autoplot() to visualize the created groups. In a 2D space ggplot2 is used in the backgroudn while for spatiotemporal methods 3D visualizations via plotly are created.

The following examples shows how the resampling_sp object from the previous example can be visualized. In this case I want to look at the first four partitions of the first repetition. The point size is adjusted via argument size. After the plot creation, additional scale_* calls are used to adjust the coordinate labels on the x and y axes, respectively.

```
autoplot(resampling_sp, task, fold_id = c(1:4), size = 0.7) *

ggplot2::scale_y_continuous(breaks = seq(-3.97, -4, -0.01)) *

ggplot2::scale_x_continuous(breaks = seq(-79.06, -79.08, -0.02))
```



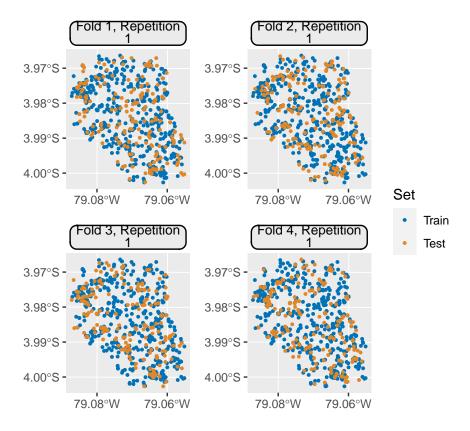
🛕 Warning

Note that setting the correct CRS for the given data is important which is done during task **creation**. Spatial offsets of up to multiple meters may occur if the wrong CRS is supplied initially.

This example used a built-in mlr3 task via tsk(). In practice however, one needs to create a spatiotemporal task via TaskClassifST/TaskRegrST and set the crs argument (unless a sf object is handed over).

mlr3spatiotempcv can also visualize non-spatial partitonings. This helps to visually compare differences. Let's use the objects from the previous example again, this time resampling_nsp.

```
autoplot(resampling_nsp, task, fold_id = c(1:4), size = 0.7) *
  ggplot2::scale_y_continuous(breaks = seq(-3.97, -4, -0.01)) *
  ggplot2::scale_x_continuous(breaks = seq(-79.06, -79.08, -0.02))
```



The visualization show very well how close train and test observations are located next to each other.

8.3.4.1. Spatial Block Visualization

This examples showcases another SpCV method: spcv_block. This method makes use of rectangular blocks to divide the study area into equally-sized parts. {mlr3spatiotempcv} has support for visualizing the created blocks and displaying their respective fold ID to get a better understanding how the final folds were composed out of the partitions. E.g. the "Fold 1 Repetition 1" plot shows that the test set is composed out of two "blocks" with the ID "1" in this case.

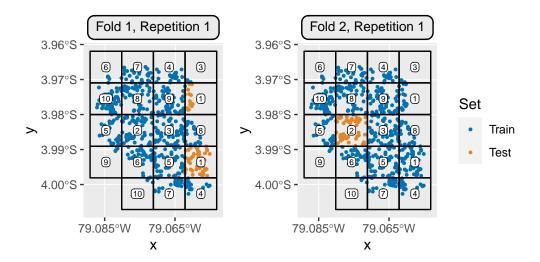
Note

The use of range = 1000L is arbitrary here and should not be copy-pasted into a real application.

```
task = tsk("ecuador")
resampling = rsmp("spcv_block", range = 1000L)

# Visualize train/test splits of multiple folds
autoplot(resampling, task, size = 0.7,
```

```
fold_id = c(1, 2), show_blocks = TRUE, show_labels = TRUE) *
ggplot2::scale_x_continuous(breaks = seq(-79.085, -79.055, 0.02))
```



8.3.4.2. Spatiotemporal Visualization

When going spatiotemporal, two dimensions are not enough anymore. To visualize space and time together, a 3D solution is needed. {mlr3spatiotempcv} makes use of plotly for this purpose.

The following examples uses a modified version of the cookfarm mlr3 task for showcasing reasons. By adjusting some levels, the individual partitions can be recognized very well.



Warning

In practice, you should not modify your data to achieve "good looking" visualizations as done in this example. This is only done for (visual) demonstration purposes.

In the following we use the sptcv cstf method after Meyer et al. (2018). Only the temporal variable is used in this first example, denoted by setting the column role "time" to variable "Date". This column role is then picked up by the resampling method. Last, autoplot() is called with an explicit definition of plot3D = TRUE. This is because sptcv_cstf can also be visualized in 2D (which only makes sense if the "space" column role is used for partitioning).

Last, sample fold n is used to take a stratified random sample from all partitions. The call to plotly::layout() only adjusts the default viewing angle of the plot - in interactive visualizations, this is not needed and the viewing angle can be adjusted with the mouse.

⚠ Warning

This is done to reduce the final plot size and keep things small for demos like this one. We

```
data = cookfarm mlr3
   set.seed(42)
   data$Date = sample(rep(c(
     "2020-01-01", "2020-02-01", "2020-03-01", "2020-04-01",
     "2020-05-01"), times = 1, each = 35768))
   task_spt = as_task_regr_st(data,
     id = "cookfarm", target = "PHIHOX",
     coordinate_names = c("x", "y"), coords_as_features = FALSE,
     crs = 26911)
   task_spt$set_col_roles("Date", roles = "time")
10
11
   rsmp_cstf_time = rsmp("sptcv_cstf", folds = 5)
12
13
   plot = autoplot(rsmp_cstf_time,
14
     fold_id = 5, task = task_spt, plot3D = TRUE,
15
     sample fold n = 3000L
16
   )
17
   plotly::layout(plot, scene = list(camera = list(eye = list(z = 0.58))))
```

If both space and time are used for partitioning in **sptcv_cstf**, the visualization becomes even more powerful as it allows to also show the observations which are omitted, i.e., not being used in either train and test sets for a specific fold.

```
task_spt$set_col_roles("SOURCEID", roles = "space")
task_spt$set_col_roles("Date", roles = "time")

rsmp_cstf_space_time = rsmp("sptcv_cstf", folds = 5)

plot = autoplot(rsmp_cstf_space_time,
    fold_id = 4, task = task_spt, plot3D = TRUE,
    show_omitted = TRUE, sample_fold_n = 3000L)

plotly::layout(plot, scene = list(camera = list(eye = list(z = 0.58, x = -1.4, y = 1.6))))
```

Combining multiple spatiotemporal plots with plotly::layout() is possible but somewhat cumbersome. First, a list of plots containing the individuals plots must be created. These plots can then be passed to plotly::subplot(). This return is then passed to plotly::layout().

```
pl = autoplot(rsmp_cstf_space_time, task = task_spt,
     fold_id = c(1, 2, 3, 4), point_size = 3,
     axis_label_fontsize = 10, plot3D = TRUE,
     sample_fold_n = 3000L, show_omitted = TRUE
   )
6
   # Warnings can be ignored
   pl_subplot = plotly::subplot(pl)
   plotly::layout(pl_subplot,
10
     title = "Individual Folds",
11
     scene = list(
12
       domain = list(x = c(0, 0.5), y = c(0.5, 1)),
13
       aspectmode = "cube",
14
       camera = list(eye = list(z = 0.20, x = -1.4, y = 1.6))
15
     ),
16
     scene2 = list(
17
       domain = list(x = c(0.5, 1), y = c(0.5, 1)),
       aspectmode = "cube",
19
       camera = list(eye = list(z = 0.1, x = -1.4, y = 1.6))
20
     ),
21
     scene3 = list(
22
       domain = list(x = c(0, 0.5), y = c(0, 0.5)),
23
       aspectmode = "cube",
24
       camera = list(eye = list(z = 0.1, x = -1.4, y = 1.6))
25
     ),
26
     scene4 = list(
27
       domain = list(x = c(0.5, 1), y = c(0, 0.5)),
       aspectmode = "cube",
29
       camera = list(eye = list(z = 0.58, x = -1.4, y = 1.6))
30
31
   )
32
```

Unfortunately, titles in subplots cannot be created dynamically. However, there is a manual workaround via annotations show in this RPubs post.

8.3.5. Choosing a Resampling Method

While the example in this section made use of the <code>spcv_coords</code> method, this should by no means infer that this method is the best or only method suitable for this task. Even though this method is quite popular, it was mainly chosen because of the clear visual grouping differences when being applied on the <code>ecuador</code> task when compared to random partitioning.

In fact, most often multiple spatial partitioning methods can be used for a dataset. It is recommended (required) that users familiarize themselves with each implemented method and decide which method to choose based on the specific characteristics of the dataset. For almost all methods implemented in mlr3spatiotempcv, there is a scientific publication describing the strengths and

weaknesses of the respective approach (either linked in the help file of mlr3spatiotempcv or its respective dependency packages).



In the example above, a cross-validation without hyperparameter tuning was shown. If a nested CV is desired, it is recommended to use the same spatial partitioning method for the inner loop (= tuning level). See Schratz et al. (2019) for more details and chapter 11 of Geocomputation with R (Lovelace, Nowosad, and Muenchow 2019).



Tip?

A list of all implemented methods in mlr3spatiotempcv can be found in the Getting Started vignette of the package.

If you want to learn even more about the field of spatial partitioning, STAC and the problems associated with it, the works of Prof. Hanna Meyer and Prof. Alexander Brenning are very much recommended for further reference.

8.3.6. Spatial Prediction

Support for (parallel) spatial prediction with terra, raster, stars and sf objects is available via mlr3spatial.

mlr3spatial has two main scopes:

- Provide DataBackends for spatial objects (vector and raster data)
- Simplify spatial prediction tasks

Overall it aims to reduce the roundtripping between spatial objects -> data.frame / data.table -> spatial object during modeling.

8.3.6.1. Spatial Data Backends

A common scenario is the existence of spatial information in (point) vector data. mlr3spatial provides as_task_* helpers to directly convert these into a spatiotemporal (ST) task:

```
library(mlr3verse)
library(mlr3spatial)
library(sf)
# load sample points
leipzig_vector = sf::read_sf(system.file("extdata",
  "leipzig_points.gpkg", package = "mlr3spatial"),
  stringsAsFactors = TRUE)
# create land cover task
```

```
12 tsk_leipzig
<TaskClassifST:leipzig_vector> (97 x 9)
* Target: land_cover
* Properties: multiclass
* Features (8):
  - dbl (8): b02, b03, b04, b06, b07, b08, b11, ndvi
* Coordinates:
           Х
 1: 732480.1 5693957
 2: 732217.4 5692769
 3: 732737.2 5692469
 4: 733169.3 5692777
 5: 732202.2 5692644
93: 733018.7 5692342
94: 732551.4 5692887
95: 732520.4 5692589
96: 732542.2 5692204
97: 732437.8 5692300
```

This saves users from stripping the coordinates from the vector data or transforming the sf object to a data.frame or data.table in the first place.

tsk_leipzig = as_task_classif_st(leipzig_vector, target = "land_cover")

mlr3spatial adds support for the following spatial data classes:

stars (from package stars)
SpatRaster (from package terra)
RasterLayer (from package raster)
RasterStack (from package raster)

8.3.6.2. Spatial prediction

The goal of spatial prediction is usually a raster image that covers a specific area. Most often this area is quite large and contains millions of pixels. The goal is to predict on each pixel and return a raster image containing these predictions.

To save users from having to go the extra mile of extracting the final model and using it with the respective predict() function of the spatial R package of their desired outcome class, mlr3spatial provides predict_spatial(). predict_spatial() let's users

- choose the output class of the resulting raster image
- optionally predict in parallel via the future package, using a parallel backend of their choice

To use a raster image for prediction, it must be wrapped into a TaskUnsupervised for internal mlr3 reasons. Next, the learner can be used to predict on the task.

```
leipzig_raster = terra::rast(system.file("extdata", "leipzig_raster.tif",
    package = "mlr3spatial"))

tsk_predict = as_task_unsupervised(leipzig_raster)

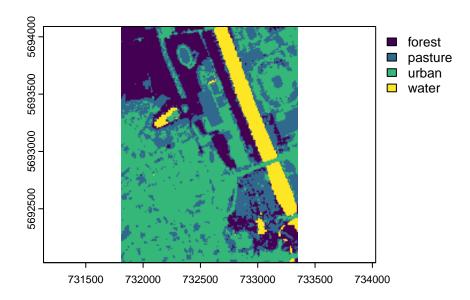
lrn = lrn("classif.ranger")
lrn$train(tsk_leipzig)

# plan("multisession") # optional parallelization
pred = predict_spatial(tsk_predict, lrn, format = "terra")
class(pred)
```

```
[1] "SpatRaster"
attr(,"package")
[1] "terra"
```

The resulting object can be visualized and treated as any other terra object. Thanks to the improved handling of factor variables in terra, the raster image contains the correct labeled classes from the prediction right away.

```
library(terra, exclude = "resample")
plot(pred, col = c("#440154FF", "#443A83FF", "#31688EFF",
"#21908CFF", "#35B779FF", "#8FD744FF", "#FDE725FF"))
```



If you are interested in the performance of parallel spatial prediction, have a look at the Spatial Prediction Benchmark vignette - the resulting plot is shown below.



⚠ The results of the shown benchmark results should be taken with care as something seems to go wrong when using all-core parallelization with terra::predict(). Also both relative and absolute numbers will vary on your local machine and results might change depending on eventual package updates in the future.

8.4. Cost-Sensitive Classification

In regular classification, the aim is to minimize the misclassification rate, and thus all types of misclassification errors are deemed equally severe. A more general setting is cost-sensitive classification. Cost-sensitive classification does not assume that the costs caused by different kinds of errors are equal. The objective of cost-sensitive classification is to minimize the expected costs.

Imagine you are an analyst for a big credit institution. Let's also assume that a correct bank decision would result in 35% of the profit at the end of a specific period. A correct decision means that the bank predicts that a customer will pay their bills (hence would obtain a loan), and the customer indeed has good credit. On the other hand, a wrong decision means that the bank predicts that the customer's credit is in good standing, but the opposite is true. This would result in a loss of 100% of the given loan.

	Good Customer (truth)	Bad Customer (truth)		
Good Customer (predicted)	+ 0.35	- 1.0		
Bad Customer (predicted)	0	0		

Expressed as costs (instead of profit), we can write down the cost-matrix as follows:

```
costs = matrix(c(-0.35, 0, 1, 0), nrow = 2)
dimnames(costs) = list(response = c("good", "bad"), truth = c("good", "bad"))
print(costs)
```

```
truth
response
          good bad
    good -0.35
          0.00
    bad
                  0
```

An exemplary data set for such a problem is the German Credit task:

```
library("mlr3verse")
task = tsk("german_credit")
table(task$truth())
```

```
bad
good
 700
     300
```

The data has 70% of customers who can pay back their credit and 30% of bad customers who default on their debt. A manager, who doesn't have any model, could decide to give either everybody credit or to give nobody credit. The resulting costs for the German credit data are:

```
# nobody:
2 (700 * costs[2, 1] + 300 * costs[2, 2]) / 1000

[1] 0
```

```
# everybody
(700 * costs[1, 1] + 300 * costs[1, 2]) / 1000
```

[1] 0.055

If the average loan is \$20,000, the credit institute will lose more than one million dollars if it would grant everybody a credit:

```
# average profit * average loan * number of customers
2 0.055 * 20000 * 1000
```

[1] 1100000

Our goal is to find a model that minimizes the costs (and maximizes the expected profit).

8.4.1. A First Model

For our first model, we choose an ordinary logistic regression (implemented in add-on package mlr3learners). We first create a classification task, then resample the model using 10-fold cross-validation and extract the resulting confusion matrix:

```
learner = lrn("classif.log_reg")
rr = resample(task, learner, rsmp("cv"))

confusion = rr$prediction()$confusion
print(confusion)
```

```
truth
response good bad
good 606 154
bad 94 146
```

To calculate the average costs like above, we can simply multiply the elements of the confusion matrix with the elements of the previously introduced cost matrix and sum the values of the resulting matrix:

```
avg_costs = sum(confusion * costs) / 1000
print(avg_costs)
```

[1] -0.0581

With an average loan of \$20,000, the logistic regression yields the following costs:

```
1 avg_costs * 20000 * 1000
```

[1] -1162000

Instead of losing over \$1,000,000, the credit institute now can expect a profit of more than \$1,000,000.

8.4.2. Cost-sensitive Measure

Our natural next step would be to further improve the modeling step in order to maximize the profit. For this purpose, we first create a cost-sensitive classification measure that calculates the costs based on our cost matrix. This allows us to conveniently quantify and compare modeling decisions. Fortunately, there already is a predefined measure Measure for this purpose: MeasureClassifCosts. The costs have to be provided as a numeric matrix whose columns and rows are named with class labels (just like the previously constructed costs matrix):

```
cost_measure = msr("classif.costs", costs = costs)
print(cost_measure)
```

<MeasureClassifCosts:classif.costs>: Cost-sensitive Classification

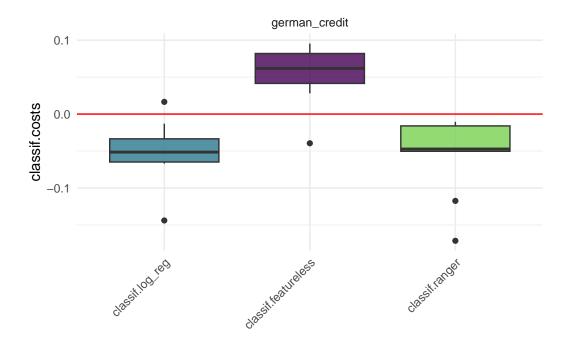
* Packages: mlr3
* Range: [-Inf, Inf]
* Minimize: TRUE
* Average: macro
* Parameters: normalize=TRUE
* Properties: -

* Predict type: response

If we now call **resample()** or **benchmark()**, the cost-sensitive measures will be evaluated. We compare the logistic regression to a simple featureless learner and to a random forest from package **ranger**:

```
learners = list(
lrn("classif.log_reg"),
lrn("classif.featureless"),
lrn("classif.ranger")
```

```
5 )
6 cv10 = rsmp("cv", folds = 10)
7 bmr = benchmark(benchmark_grid(task, learners, cv10))
8 autoplot(bmr, measure = cost_measure) + ggplot2::geom_hline(yintercept = 0, colour = "red")
```



As expected, the featureless learner is performing comparably badly. The logistic regression and the random forest both yield a profit on average.

8.4.3. Thresholding

Although we now correctly evaluate the models in a cost-sensitive fashion, the models themselves are unaware of the classification costs. They assume the same costs for both wrong classification decisions (false positives and false negatives). Some learners natively support cost-sensitive classification (e.g., XXX). However, we will concentrate on a more generic approach that works for all models which can predict probabilities for class labels: thresholding.

Most learners can calculate the probability p for the positive class. If p exceeds the threshold 0.5, they predict the positive class and the negative class otherwise.

For our binary classification case of the credit data, we primarily want to minimize the errors where the model predicts "good", but truth is "bad" (i.e., the number of false positives) as this is the more expensive error. If we now increase the threshold to values > 0.5, we reduce the number of false negatives. Note that we increase the number of false positives simultaneously, or, in other words, we are trading false positives for false negatives.

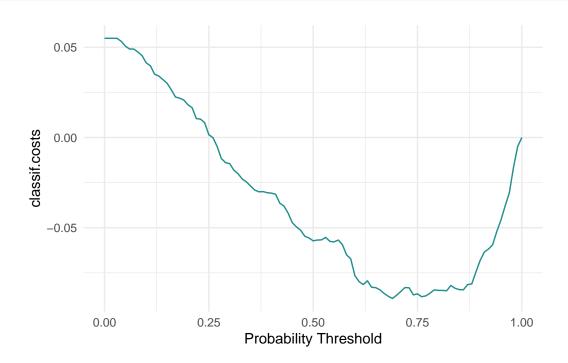
```
# fit models with probability prediction
learner = lrn("classif.log_reg", predict_type = "prob")
3 rr = resample(task, learner, rsmp("cv"))
4 p = rr$prediction()
5 print(p)
<PredictionClassif> for 1000 observations:
    row_ids truth response prob.good prob.bad
                     bad 0.40126720 0.5987328
              bad
         17 good
                     good 0.97406360 0.0259364
            good
         46
                  good 0.74870171 0.2512983
        973
             bad
                     bad 0.09917846 0.9008215
        976 good
                     good 0.78358973 0.2164103
        997 good
                     good 0.51492948 0.4850705
1 # helper function to try different threshold values interactively
with_threshold = function(p, th) {
    p$set_threshold(th)
   list(confusion = p$confusion, costs = p$score(measures = cost_measure))
 }
7 with_threshold(p, 0.5)
$confusion
        truth
response good bad
    good 598 152
    bad
         102 148
$costs
classif.costs
      -0.0573
with_threshold(p, 0.75)
$confusion
        truth
response good bad
    good 462 75
    bad
          238 225
$costs
classif.costs
      -0.0867
```

```
$\text{with_threshold(p, 1.0)}$
$\text{confusion} \text{truth}$
response good bad \text{good 0 0 0} \text{bad 700 300}$
$\text{costs}$
```

\$costs classif.costs 0

There is also an autoplot() method which systematically varies the threshold between 0 and 1 and calculates the corresponding scores:

```
autoplot(p, type = "threshold", measure = cost_measure)
```



Instead of manually or visually searching for good values, the base R function optimize() can do the job for us:

```
# simple wrapper function that takes a threshold and returns the resulting model performance
# this wrapper is passed to optimize() to find its minimum for thresholds in [0.5, 1]

# f = function(th) {
# with_threshold(p, th)$costs
# with_threshold(p, th)$costs
```

Note that the function "optimize()" is intended for unimodal functions and, therefore, may converge to a local optimum here. See the next section for better alternatives to tune the threshold.

8.4.4. Threshold Tuning

Currently mlr3pipelines offers two main strategies towards adjusting classification thresholds. We can either expose the thresholds as a hyperparameter of the Learner by using PipeOpThreshold. This allows us to tune the thresholds via an outside optimizer from mlr3tuning. Alternatively, we can also use PipeOpTuneThreshold which automatically tunes the threshold after each learner is fit. Both methods are described in the following subsections.

8.4.5. PipeOpThreshold

PipeOpThreshold can be put directly after a Learner.

A simple example would be:

```
gr = lrn("classif.rpart", predict_type = "prob") %>>% po("threshold")
l = as_learner(gr)
```

Note, that predict_type = "prob" is required for po("threshold") to have any effect.

The thresholds are now exposed as a hyperparameter of the GraphLearner we created:

```
1 l$param_set
```

<ParamSetCollection>

```
id
                                      class lower upper nlevels
                                                                          default
 1:
                 classif.rpart.cp ParamDbl
                                                 0
                                                       1
                                                              Inf
                                                                             0.01
 2:
        classif.rpart.keep_model ParamLgl
                                                NA
                                                      NA
                                                                2
                                                                            FALSE
 3:
        classif.rpart.maxcompete ParamInt
                                                 0
                                                     Inf
                                                              Inf
                                                                                4
          classif.rpart.maxdepth ParamInt
                                                      30
 4:
                                                 1
                                                               30
                                                                               30
 5:
      classif.rpart.maxsurrogate ParamInt
                                                 0
                                                     Inf
                                                              Inf
                                                                                5
 6:
         classif.rpart.minbucket ParamInt
                                                 1
                                                     Inf
                                                              Inf <NoDefault[3]>
          classif.rpart.minsplit ParamInt
 7:
                                                 1
                                                     Inf
                                                              Inf
                                                                               20
 8: classif.rpart.surrogatestyle ParamInt
                                                 0
                                                       1
                                                                2
                                                                                0
                                                       2
                                                                3
                                                                                2
 9:
      classif.rpart.usesurrogate ParamInt
                                                 0
               classif.rpart.xval ParamInt
                                                 0
10:
                                                     Inf
                                                              Inf
                                                                               10
            threshold.thresholds ParamUty
11:
                                                NA
                                                      NA
                                                              Inf <NoDefault[3]>
1 variable not shown: [value]
```

We can now tune those thresholds from the outside as follows:

Before tuning, we have to define which hyperparameters we want to tune over. In this example, we only tune over the thresholds parameter of the threshold pipe operator. You can easily imagine that we can also jointly tune over additional hyperparameters, i.e., rpart's cp parameter.

As the Task we aim to optimize for is a binary task, we can simply specify the threshold param:

```
library("paradox")
ps = ps(threshold.thresholds = p_dbl(lower = 0, upper = 1))
```

We now create a **AutoTuner** which automatically tunes the supplied learner over the **ParamSet** we supplied above.

```
at = AutoTuner$new(
learner = 1,
resampling = rsmp("cv", folds = 3L),
measure = msr("classif.ce"),
search_space = ps,
terminator = trm("evals", n_evals = 5L),
tuner = tnr("random_search")
)
at$train(tsk("german_credit"))
```

Inside the trafo, we simply collect all set params into a named vector via map_dbl and store it in the threshold.thresholds slot expected by the learner.

Again, we create a **AutoTuner**, which automatically tunes the supplied learner over the **ParamSet** we supplied above.

One drawback of this strategy is that this requires us to fit a new model for each new threshold setting. While setting a threshold and computing performance is relatively cheap, fitting the learner is often more computationally demanding. An often better strategy is, therefore, to optimize the thresholds separately after each model fit.

8.4.6. PipeOpTunethreshold

PipeOpTuneThreshold on the other hand works together with PipeOpLearnerCV. It directly optimizes the cross-validated predictions made by this PipeOp. This is necessary to avoid over-fitting the threshold tuning.

A simple example would be:

```
gr = po("learner_cv", lrn("classif.rpart", predict_type = "prob")) %>>% po("tunethreshold")
le l2 = as_learner(gr)
```

Note, that predict_type = "prob" is required for po("tunethreshold") to work. Additionally, note that this time no threshold parameter is exposed, and it is automatically tuned internally.

```
1 12$param_set
```

<ParamSetCollection>

	id	class	lower	upper	nlevels
1:	classif.rpart.resampling.method	${\tt ParamFct}$	NA	NA	2
2:	classif.rpart.resampling.folds	${\tt ParamInt}$	2	Inf	Inf
3:	<pre>classif.rpart.resampling.keep_response</pre>	ParamLgl	NA	NA	2
4:	classif.rpart.cp	${\tt ParamDbl}$	0	1	Inf
5:	<pre>classif.rpart.keep_model</pre>	ParamLgl	NA	NA	2
6:	classif.rpart.maxcompete	${\tt ParamInt}$	0	Inf	Inf
7:	classif.rpart.maxdepth	${\tt ParamInt}$	1	30	30
8:	classif.rpart.maxsurrogate	${\tt ParamInt}$	0	Inf	Inf
9:	classif.rpart.minbucket	ParamInt	1	Inf	Inf
10:	classif.rpart.minsplit	${\tt ParamInt}$	1	Inf	Inf
11:	classif.rpart.surrogatestyle	${\tt ParamInt}$	0	1	2
12:	classif.rpart.usesurrogate	${\tt ParamInt}$	0	2	3
13:	classif.rpart.xval	${\tt ParamInt}$	0	Inf	Inf
14:	<pre>classif.rpart.affect_columns</pre>	${\tt ParamUty}$	NA	NA	Inf
15:	tunethreshold.measure	${\tt ParamUty}$	NA	NA	Inf
16:	tunethreshold.optimizer	ParamUty	NA	NA	Inf
17:	tunethreshold.log_level	ParamUty	NA	NA	Inf
2 va	ariables not shown: [default, value]				

Note that we can set rsmp("intask") as a resampling strategy for "learner_cv" in order to evaluate predictions on the "training" data. This is generally not advised, as it might lead to over-fitting on the thresholds but can significantly reduce runtime.

For more information, see the post on Threshold Tuning on the mlr3 gallery.

8.5. Cluster Analysis

Cluster analysis is a type of unsupervised machine learning where the goal is to group data into clusters, where each cluster contains similar observations. The similarity is based on specified

metrics that are task and application-dependent. Cluster analysis is closely related to classification in the sense that each observation needs to be assigned to a cluster or a class. However, unlike classification problems where each observation is labeled, clustering works on data sets without true labels or class assignments.

The package mlr3cluster extends mlr3 with the following objects for cluster analysis:

- TaskClust to define clustering tasks
- LearnerClust as base class for clustering learners
- PredictionClust as specialized class for Prediction objects
- MeasureClust as specialized class for performance measures

Since clustering is a type of unsupervised learning, TaskClust is slightly different from TaskRegr and TaskClassif objects. More specifically:

- truth() function is missing because observations are not labeled.
- target field is empty and will return character(0) if accessed anyway.

Additionally, LearnerClust provides two extra fields that are absent from supervised learners:

- assignments returns cluster assignments for training data. It returns NULL if accessed before training.
- save_assignments is a boolean field that controls whether or not to store training set assignments in a learner.

Finally, PredictionClust contains two additional fields:

- partition stores cluster partitions.
- prob stores cluster probabilities for each observation.

8.5.1. Train and Predict

Clustering learners provide both train and predict methods. The analysis typically consists of building clusters using all available data. To be consistent with the rest of the library, we refer to this process as training.

Some learners can assign new observations to existing groups with predict. However, prediction does not always make sense, as is the case for hierarchical clustering. In hierarchical clustering, the goal is to build a hierarchy of nested clusters by either splitting large clusters into smaller ones or merging smaller clusters into bigger ones. The final result is a tree or dendrogram which can change if a new data point is added. For consistency with the rest of the ecosystem, mlr3cluster offers predict method for hierarchical clusterers but it simply assigns all points to a specified number of clusters by cutting the resulting tree at a corresponding level. Moreover, some learners estimate the probability of each observation belonging to a given cluster. predict_types field gives a list of prediction types for each learner.

After training, the model field stores a learner's model that looks different for each learner depending on the underlying library. predict returns a PredictionClust object that gives a simplified view of the learned model. If the data given to the predict method is the same as the one on which the learner was trained, predict simply returns cluster assignments for the "training" observations.

On the other hand, if the test set contains new data, predict will estimate cluster assignments for that data set. Some learners do not support estimating cluster partitions on new data and will instead return assignments for training data and print a warning message.

In the following example, a \$k\$-means learner is applied on the US arrest data set. The class labels are predicted and the contribution of the task features to assignment of the respective class are visualized.

```
library("mlr3")
library("mlr3cluster")
library("mlr3viz")
set.seed(1L)

# create an example task
task = tsk("usarrests")
print(task)
```

```
<TaskClust:usarrests> (50 x 4): US Arrests

* Target: -

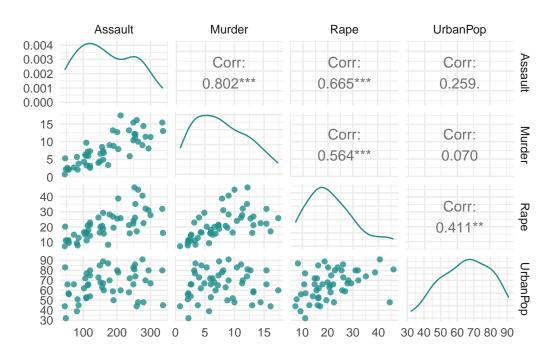
* Properties: -

* Features (4):

- int (2): Assault, UrbanPop

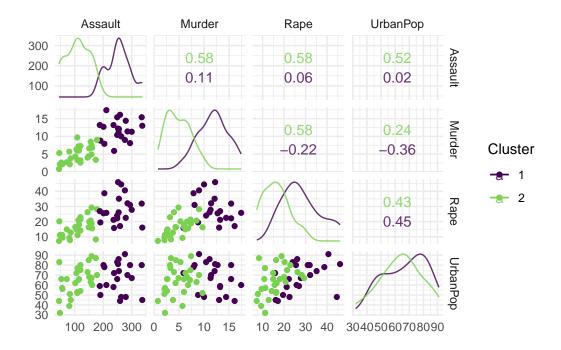
- dbl (2): Murder, Rape
```

autoplot(task)



```
# create a k-means learner
learner = lrn("clust.kmeans")
4 # assigning each observation to one of the two clusters (default in clust.kmeans)
5 learner$train(task)
6 learner$model
K-means clustering with 2 clusters of sizes 21, 29
Cluster means:
  Assault Murder
                    Rape UrbanPop
1 255.0000 11.857143 28.11429 67.61905
2 109.7586 4.841379 16.24828 64.03448
Clustering vector:
 [39] 2 1 2 1 1 2 2 2 2 2 2 2
Within cluster sum of squares by cluster:
[1] 41636.73 54762.30
 (between_SS / total_SS = 72.9 %)
Available components:
[1] "cluster"
                "centers"
                                                       "tot.withinss"
                             "totss"
                                         "withinss"
[6] "betweenss"
                "size"
                             "iter"
                                         "ifault"
```

```
# make "predictions" for the same data
prediction = learner$predict(task)
autoplot(prediction, task)
```



8.5.2. Measures

The difference between supervised and unsupervised learning is that there is no ground truth data in unsupervised learning. In a supervised setting, such as classification, we would need to compare our predictions to true labels. Since clustering is an example of unsupervised learning, there are no true labels to which we can compare. However, we can still measure the quality of cluster assignments by quantifying how closely objects within the same cluster are related (cluster cohesion) as well as how distinct different clusters are from each other (cluster separation).

There are a few built-in evaluation metrics available to assess the quality of clustering. One of them is within sum of squares (WSS) which calculates the sum of squared differences between observations and centroids. WSS is useful because it quantifies cluster cohesion. The range of this measure is $[0, \infty)$ where a smaller value means that clusters are more compact.

Another measure is silhouette quality index that quantifies how well each point belongs to its assigned cluster versus neighboring cluster. Silhouette values are in [-1, 1] range.

Points with silhouette closer to:

- 1 are well clustered
- 0 lie between two clusters
- -1 likely placed in the wrong cluster

The following is an example of conducting a benchmark experiment with various learners on **iris** data set without target variable and assessing the quality of each learner with both within sum of squares and silhouette measures.

```
design = benchmark_grid(
   tasks = TaskClust$new("iris", iris[-5]),
   learners = list(
        lrn("clust.kmeans", centers = 3L),
        lrn("clust.pam", k = 2L),
        lrn("clust.cmeans", centers = 3L)),
   resamplings = rsmp("insample"))
   print(design)

task learner resampling
1: <TaskClust[46]> <LearnerClustKMeans[38]> <ResamplingInsample[20]>
2: <TaskClust[46]> <LearnerClustPAM[38]> <ResamplingInsample[20]>
```

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

# define measure
measures = list(msr("clust.wss"), msr("clust.silhouette"))
bmr$aggregate(measures)
```

3: <TaskClust[46]> <LearnerClustCMeans[38]> <ResamplingInsample[20]>

```
resample_result task_id
                                     learner_id resampling_id iters clust.wss
  nr
   1 <ResampleResult[21]>
                              iris clust.kmeans
                                                     insample
                                                                  1 78.85144
1:
                                                                  1 153.32572
   2 <ResampleResult[21]>
                              iris
                                      clust.pam
                                                     insample
3: 3 <ResampleResult[21]>
                                                     insample
                              iris clust.cmeans
                                                                  1 79.02617
1 variable not shown: [clust.silhouette]
```

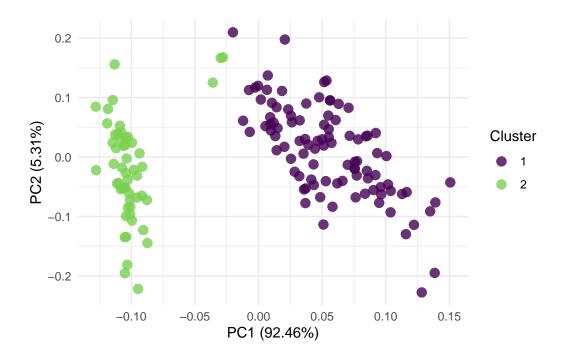
The experiment shows that using k-means algorithm with three centers produces a better within sum of squares score than any other learner considered. However, pam (partitioning around medoids) learner with two clusters performs the best when considering silhouette measure which takes into the account both cluster cohesion and separation.

8.5.3. Visualization

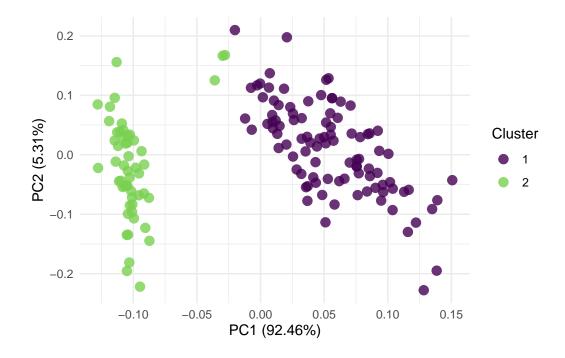
Cluster analysis in mlr3 is integrated with mlr3viz which provides a number of useful plots. Some of those plots are shown below.

```
task = TaskClust$new("iris", iris[-5])
learner = lrn("clust.kmeans")
learner$train(task)
prediction = learner$predict(task)

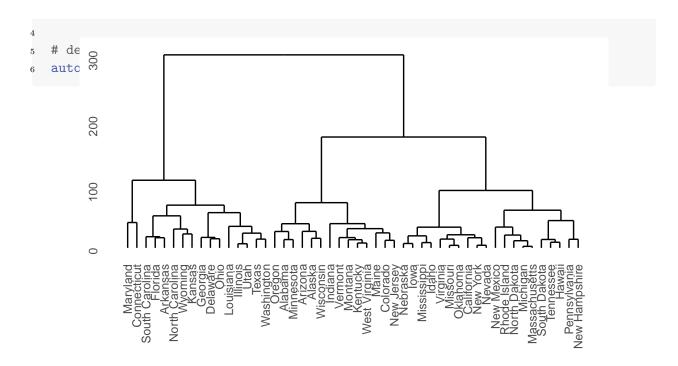
# performing PCA on task and showing assignments
autoplot(prediction, task, type = "pca")
```



same as above but with probability ellipse that assumes normal distribution
autoplot(prediction, task, type = "pca", frame = TRUE, frame.type = "norm")

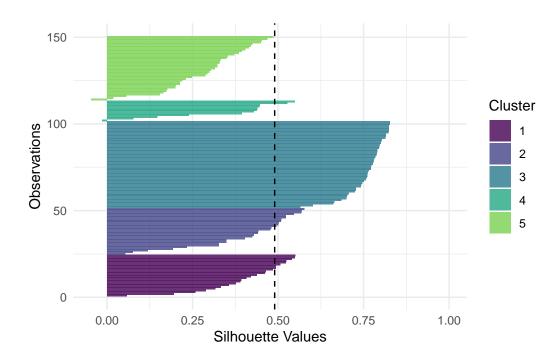


```
task = tsk("usarrests")
learner = lrn("clust.hclust")
learner$train(task)
```



Silhouette plots can help to visually assess the quality of the analysis and help choose a number of clusters for a given data set. The red dotted line shows the mean silhouette value and each bar represents a data point. If most points in each cluster have an index around or higher than mean silhouette, the number of clusters is chosen well.

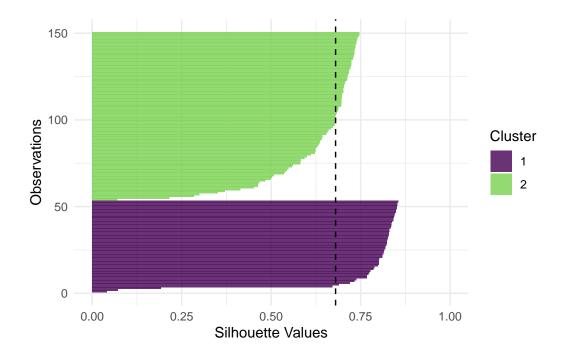
```
# silhouette plot allows to visually inspect the quality of clustering
task = TaskClust$new("iris", iris[-5])
learner = lrn("clust.kmeans")
learner$param_set$values = list(centers = 5L)
learner$train(task)
prediction = learner$predict(task)
autoplot(prediction, task, type = "sil")
```



The plot shows that all points in cluster 5 and almost all points in clusters 4, 2 and 1 are below average silhouette index. This means that a lot of observations lie either on the border of clusters or are likely assigned to the wrong cluster.

```
learner = lrn("clust.kmeans")
learner$param_set$values = list(centers = 2L)
learner$train(task)
prediction = learner$predict(task)
autoplot(prediction, task, type = "sil")
```

8. Special Tasks



Setting the number of centers to two improves both the average silhouette score as well as the overall quality of clustering because almost all points in cluster 1 are higher than, and a lot of points in cluster 2 are close to the mean silhouette. Hence, having two centers might be a better choice for the number of clusters.

9. Technical

TODO (150-200 WORDS)

This chapter provides an overview of the technical details of the mlr3 framework. This includes the following topics:

- Parallelization with the **future** framework (Section 9.1),
- how to handle errors and troubleshoot (Section 9.2),
- working with out-of-memory data, e.g., data stored in databases (Section 9.3),
- working with hyperparameters (Section 9.4), and
- adjust the logger to your needs (Section 9.5).

9.1. Parallelization

Parallelization refers to running multiple jobs in parallel, i.e., executing them simultaneously on multiple CPU cores, CPUs, or computational nodes. This process allows for significant savings in computing power.

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 or forks are ways to utilize all cores. If you have multiple machines, they need a way to communicate and exchange information, e.g. via protocols like network sockets or the Message Passing Interface (MPI). We don't want to delve too deep into such details here, but want to introduce some terminology:

- We call the parallelization platform together with its implementation for R the **parallelization backend**. As many parallelization backends have a different API, we are using the **future** package as an additional abstraction layer. mlr3 just interfaces **future** while the user can control how the code is executed.
- The R session or process which orchestrates the computational work is called **main**, and it starts computational **jobs**.
- The R sessions, processes, forks or machines which receive the jobs, do the calculation and then send back the result are called **workers**.

We distinguish between *implicit parallelism* and *explicit parallelism*. For the former, no special directives are required to enable the parallelization, everything works fully automatically. For the latter, parallelization has to be manually configured. On the one hand, this gives you full control over the execution, but on the other hand, this poses a greater obstacle for non-experts.

Note

We don't cover parallelization on GPUs here. mlr3 only distributes the fitting of multiple learners, e.g., during resampling, benchmarking, or tuning. On this rather abstract level, GPU parallelization doesn't work efficiently. 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., here for xgboost.

9.1.1. Implicit Parallelization

We talk about implicit parallelization in the context of mlr3, if mlr3 calls external code (i.e., code from foreign CRAN packages which implements a Learner) that itself runs in parallel. Note that this definition includes GPU acceleration.

Many machine learning algorithms can parallelize their model fit using threading, e.g., the random forest implementation in ranger or the boosting implemented in xgboost. During threading, the implementation instructs some sequential parts of the code to be executed independently of the other parts in the same process.

For example, while fitting a 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.

Warning

It does not make practical sense to actually execute in parallel every operation that can be parallelized. Starting and terminating workers (here: threads) 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 must be related to the runtime of the sequential execution. If the sequential execution is comparably fast, enabling parallelization often just introduces additional complexity and slows down the execution.

Unfortunately, threading conflicts with certain parallel backends used during explicit parallelization, causing the system to be overutilized in the best case and causing hangs or segfaults in the worst case. For this reason, we introduced the convention that implicit parallelization is turned off per default. Hyperparameters that control the number of threads are tagged with the label "threads". Currently, controlling the number of threads is possible for some learners and filters from the mlr3filters package:

```
library("mlr3learners") # for the ranger learner
learner = lrn("classif.ranger")
```

```
learner$param_set$ids(tags = "threads")
```

[1] "num.threads"

To enable the parallelization for this learner, we provide the helper function set threads() which

```
# 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

1  # auto-detect cores on the local machine
```

```
<LearnerClassifRanger:classif.ranger>
```

- * Model: -
- * Parameters: num.threads=2

2 set threads(learner)

- * 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

Danger

Automatic detection of the number of CPUs 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 comes with reasonable heuristics for many setups. See this blog post for some background information about the 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 energy-efficient CPU cores, for example, the "Icestorm" cores on a Mac M1 chip. These are comparably slower than the high-performance "Firestorm" cores and

not well suited for heavy computations.

• 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.

9.1.2. Explicit Parallelization

Here, we talk about explicit parallelization if mlr3 starts and controls the parallelization itself. For this purpose, an additional abstraction layer is used to be able to operate on a unified interface for a broad range of parallel backends: the future package. There are two operations where mlr3 calls the future package: while performing resampling via resample() and while benchmarking via benchmark(). During resampling, because all resampling iterations are independent of each other, all iterations can be executed in parallel. The same holds for benchmarking, where additionally to the independent model fits of a single resampling, all combinations in the provided design are also independent. These iterations are performed by future using the parallel backend configured with future::plan(). Extension packages like mlr3tuning internally call benchmark() during tuning and thus work in parallel, too.



When computational problems are so easy to parallelize, they are often referred to as "embar-rassingly parallel".

Whenever you loop over elements with a map-like function (e.g., lapply(), sapply(), mapply(), vapply() or a function from package purrr), you are facing an embarrassingly parallel problem. Such problems are straightforward to parallelize with R, e.g., with the furrr package providing map-like functions executed in parallel via the future framework. The same holds for for-loops with independent iterations, i.e., loops where the current iteration does not rely on the results of previous iterations.

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

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

# define objects to perform a resampling
task = tsk("spam")
learner = lrn("classif.rpart")
resampling = rsmp("cv", folds = 3)
```

```
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 this default have already been discussed for implicit parallelization). You should see a decrease in the reported elapsed time, but in practice, you cannot expect the runtime to fall linearly as the number of cores increases (Amdahl's law). 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. Therefore, it is advised to only consider parallelization for resamplings where each iteration runs at least several seconds.

Figure 9.1 illustrates the parallelization from the above example. From left to right:

- 1. The main process calls the resample() function.
- 2. The task is split into 3 folds.
- 3. The folds are passed to three 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.

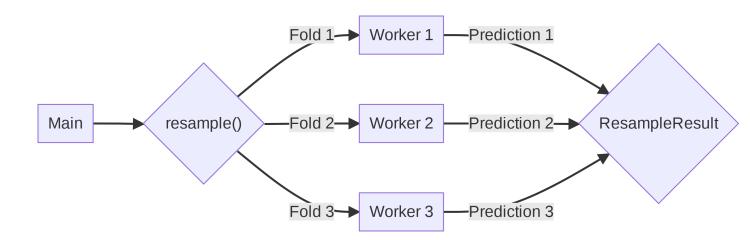


Figure 9.1.: Parallelization of a resampling using a 3-fold cross-validation

Note

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

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

9.1.3. Reproducibility

Usually reproducibility is a major concern during parallelization as special pseudorandom number generators (PRNGs) are required. Luckily, this problem is already solved for us by the excellent future package mlr3 calls under the hood. future ensures that all workers will receive the exactly same PRNG streams. Although this alone does not guarantee full reproducibility, it is one problem less to worry about.

You can find more details about the used pseudo RNG in this blog post.

9.1.4. Nested Resampling Parallelization

Nested resampling results in two nested resampling loops, and the user can choose which of them should be parallelized. 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 the nested resampling section):

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

Loading required package: paradox

```
learner = lrn("classif.rpart",
minsplit = to_tune(2, 128, logscale = TRUE)

tage at = auto_tuner(
method = tnr("random_search"),
learner = learner,
resampling = rsmp("cv", folds = 2), # inner CV
measure = msr("classif.ce"),
term_evals = 20,

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> of 5 iterations
* Task: penguins
* Learner: classif.rpart.tuned
* Warnings: 0 in 0 iterations
* Errors: 0 in 0 iterations
```

Here, we have two opportunities to tune: the inner cross-validation of the auto tuner with 2 folds, or the outer cross-validation of the resampling with 5 folds. Let's say 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, i.e., depending on the parallelization backend and its scheduling strategy,

- (a) until all four iterations have been finished, and the results have been collectively reported back to the main process, or
- (b) either one of the four cores has terminated the first core reporting back will get a new task as soon as possible.

Note

The former method usually comes with less synchronization overhead and is best suited for short jobs with homogeneous runtimes. The latter yields better runtimes 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. For many backends, you cannot control the scheduling. However, future allows you to first chunk jobs together which combines multiple tasks into blocks that run sequentially on a worker, avoiding the intermediate synchronization steps.

The resulting CPU utilization of the nested resampling example on 4 CPUs is visualized in two Figures:

• Figure 9.2 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 9.3 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.

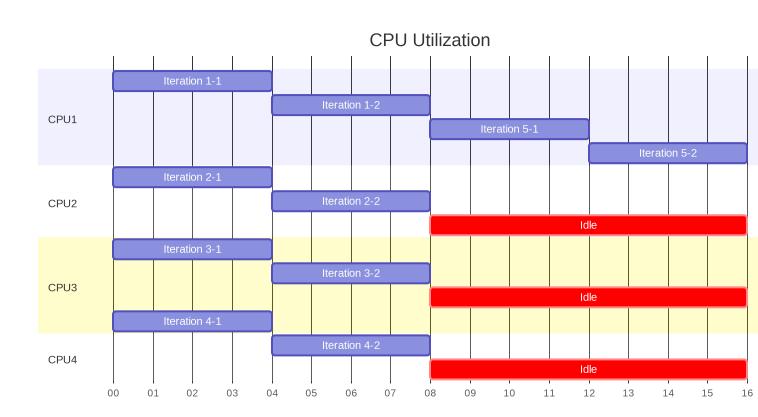


Figure 9.2.: CPU utilization while parallelizing the outer resampling of a 2-fold cross-validation nested inside a 5-fold cross-validation on 4 CPUs.

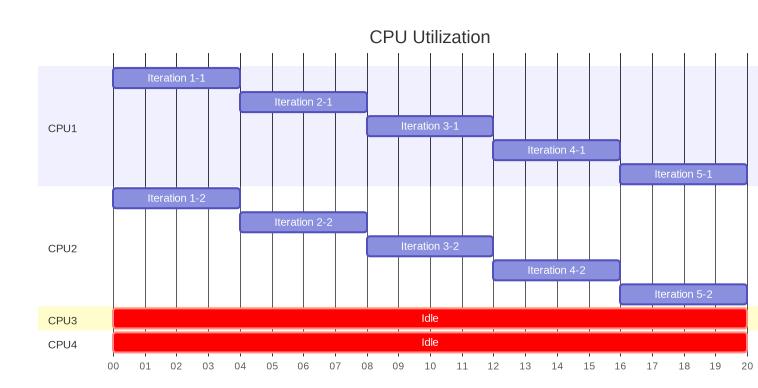


Figure 9.3.: CPU utilization while parallelizing the inner resampling of a 2-fold cross-validation nested inside a 5-fold cross-validation on 4 CPUs.

9. Technical

If possible, the number of iterations can be adapted to the available hardware. There is no law set in stone that you have to do, 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. 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).

Danger

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 performance, 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 Terminators work as intended, i.e., you cannot exceed the computational budget.

9.2. Error Handling

In ML, it is not uncommon for something to break. 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 can't 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 emerging 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 will concentrate on the debug learner for classification:

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

 $\verb|\classifDebug:classif.debug>: Debug Learner for Classification|\\$

* 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 hyperparameters 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	${\tt ParamDbl}$	0	1	Inf	0	
2:	error_train	${\tt ParamDbl}$	0	1	Inf	0	
3:	message_predict	${\tt ParamDbl}$	0	1	Inf	0	
4:	message_train	${\tt ParamDbl}$	0	1	Inf	0	
5:	<pre>predict_missing</pre>	${\tt ParamDbl}$	0	1	Inf	0	
6:	<pre>predict_missing_type</pre>	${\tt ParamFct}$	NA	NA	2	na	
7:	save_tasks	${\tt ParamLgl}$	NA	NA	2	FALSE	
8:	segfault_predict	${\tt ParamDbl}$	0	1	Inf	0	
9:	$segfault_train$	${\tt ParamDbl}$	0	1	Inf	0	
10:	sleep_train	${\tt ParamUty}$	NA	NA	Inf	<nodefault[3]></nodefault[3]>	
11:	sleep_predict	${\tt ParamUty}$	NA	NA	Inf	<nodefault[3]></nodefault[3]>	
12:	threads	${\tt ParamInt}$	1	Inf	Inf	<nodefault[3]></nodefault[3]>	
13:	warning_predict	${\tt ParamDbl}$	0	1	Inf	0	
14:	warning_train	${\tt ParamDbl}$	0	1	Inf	0	
15:	X	${\tt ParamDbl}$	0	1	Inf	<nodefault[3]></nodefault[3]>	
16:	iter	${\tt ParamInt}$	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("iris"))
```

Error in .__LearnerClassifDebug__.train(self = self, private = private, : Error from classif.de

If this has been a regular learner, we could now start debugging with traceback() (or create a Minimal Reproducible Example (MRE) to file a bug report upstream).

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().

9.2.1. 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 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), and
- 2. a statistically sound way to proceed while being able to aggregate over partial results (next Section 9.2.2).

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:

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

```
1 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 segfaults which otherwise would tear down the complete 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. Unfortunately, this 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.

9.2.2. Fallback learners

Fallback learners have the purpose of allowing scoring results in cases where a Learner failed to fit a model, refuses to provide predictions for all observations or predicts missing values.

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 missing scores. 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. If a score is missing, 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 we don't have to enable encapsulation 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 we don't have a model, 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("classif.rpart")), rsmp("classif.rpart")), rsmp("classif.rpart"))
```

```
5 aggr[, .(learner_id, warnings, errors, classif.ce)]
```

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
1:
           1 Error from classif.debug->train()
2:
           4 Error from classif.debug->train()
3:
           5 Error from classif.debug->train()
           6 Error from classif.debug->train()
4:
5:
           7 Error from classif.debug->train()
           8 Error from classif.debug->train()
6:
7:
           9 Error from classif.debug->train()
8:
          10 Error from classif.debug->train()
```

A similar problem 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.

Clearly, at its core, this is the same problem as outlined before. Algorithm B would easily outperform algorithm A, but you have not factored in that you can not generate predictions for all observations. Long story short, if a fallback learner is involved, missing predictions of the base learner will be automatically replaced with predictions from the fallback learner. This is illustrated in the following example:

```
task = tsk("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>
172 0 172 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.

9.2.3. Actionable Errors

All problems demonstrated so far are artificial and non-actionable. The usefulness of encapsulation and error logging usually only really becomes apparent in large benchmarks, especially in combination with parallelization. For a fair comparison, you need to distinguish between the following cases:

- (a) You have made a mistake, e.g., forgot a required preprocessing step in your pipeline. Action: Fix problems, restart computation.
- (b) Temporary problems related to the executing system, e.g., network hiccups. Action: Restart computation.
- (c) Intrinsic, deterministic and reproducible problem with the model fitting. Action: Impute with fallback learner.

The package mlr3batchmark provides functionality to map jobs of a benchmark to computational jobs for the package batchtools. This provides a convenient way get fine-grained control over the execution of each single resampling iteration and then combine the results afterwards to a BenchmarkResult again to proceed with the analysis.

9.3. Data Backends

In mlr3, Tasks 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. The default backend uses data.table via the DataBackendDataTable as a very fast and efficient in-memory database. For example, we can query some information of the mlr_tasks_penguins task:

```
task = tsk("penguins")
backend = task$backend
backend$nrow
```

[1] 344

```
1 backend$ncol
```

[1] 9

For bigger data, or when working with many tasks simultaneously in the same R session, 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, PostgreSQL or SQLite.
- 2. DataBackendDuckDB for the impressive DuckDB database connected via duckdb: a fast, zero-configuration alternative to SQLite.
- 3. DataBackendDuckDB, again, but for Parquet files. 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.

9.3.1. Databases with DataBackendDplyr

To demonstrate the <code>DataBackendDplyr</code> we use the NYC flights data set from the <code>nycflights13</code> package and move it into a SQLite database. Although <code>as_sqlite_backend()</code> 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
 $ month
              : int [1:336776] 1 1 1 1 1 1 1 1 1 1 ...
               : int [1:336776] 1 1 1 1 1 1 1 1 1 1 ...
$ day
 $ dep time : int [1:336776] 517 533 542 544 554 555 557 557 558 ...
 $ sched_dep_time: int [1:336776] 515 529 540 545 600 558 600 600 600 ...
 $ 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 ...
               : chr [1:336776] "UA" "UA" "AA" "B6" ...
 $ carrier
 $ flight
              : int [1:336776] 1545 1714 1141 725 461 1696 507 5708 79 301 ...
               : chr [1:336776] "N14228" "N24211" "N619AA" "N804JB" ...
 $ tailnum
              : chr [1:336776] "EWR" "LGA" "JFK" "JFK" ...
 $ origin
 $ dest
               : chr [1:336776] "IAH" "IAH" "MIA" "BQN" ...
              : num [1:336776] 227 227 160 183 116 150 158 53 140 138 ...
 $ air_time
$ distance : num [1:336776] 1400 1416 1089 1576 762 ...
 $ hour
               : num [1:336776] 5 5 5 5 6 5 6 6 6 6 ...
            : num [1:336776] 15 29 40 45 0 58 0 0 0 0 ...
 $ minute
 $ time_hour : POSIXct[1:336776], format: "2013-01-01 05:00:00" "2013-01-01 05:00:00" ...
1 # 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
```

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

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

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

Attaching package: 'dplyr'

11 rm(flights)

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")

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

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

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

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

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

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

```
tbl = mutate(tbl, carrier = case_when(
    carrier %in% c("00", "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:

```
ı b$nrow
```

[1] 163707

```
ı b$ncol
```

[1] 13

```
b$head()
```

```
row_id year month day hour minute dep_time arr_time carrier flight air_time
        2 2013
                               5
                                     29
                                              533
                                                                  UA
                                                                        1714
                                                                                   227
1:
                     1
                         1
                                                        850
                               5
2:
        4 2013
                         1
                                     45
                                              544
                                                       1004
                                                                  B6
                                                                         725
                                                                                   183
                               5
3:
        6 2013
                     1
                                     58
                                              554
                                                        740
                                                                  UA
                                                                        1696
                                                                                   150
4:
        8 2013
                         1
                               6
                                      0
                                              557
                                                        709
                                                                  ΕV
                                                                        5708
                     1
                                                                                    53
       10 2013
                               6
                                      0
                                              558
                                                        753
                                                                         301
5:
                     1
                         1
                                                                  AA
                                                                                   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 and create the following mlr3 objects:

- A regression task, based on the previously created mlr3db::DataBackendDplyr.
- A regression learner (regr.rpart).
- A resampling strategy: 3 times repeated subsampling using 2% of the observations for training ("subsampling")
- Measures "mse", "time_train" and "time_predict"

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

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

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

<ResampleResult> of 3 iterations
* Task: flights_sqlite
* Learner: regr.rpart
* Warnings: 0 in 0 iterations
* Errors: 0 in 0 iterations

1 rr$aggregate(measures)

regr.mse time_train time_predict
1246.0767508 0.2966667 2.1006667
```

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)
2 DBI::dbDisconnect(con)
```

9.3.2. Parquet Files with DataBackendDuckDB

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 you might only need a small proportion of the data, the complete data frame sits in memory and consumes memory. This is especially a problem if you work with many tasks simultaneously.
- 2. During parallelization, the complete data needs to be transferred to the workers which can cause significant overhead.

A very simple way to avoid this is given by just converting the DataBackendDataTable to a DataBackendDuckDB. As we have already demonstrated how to operate on a SQLite database, and DuckDB is not different in that regard. 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:

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.

9.4. Parameters (using paradox)

Note

We are currently revising the book. This section contains a lot of legacy code. You can find the new paradox syntax in the Defining a Tuning Spaces section.

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

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

The tools provided by paradox therefore relate to:

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

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

9.4.1. Reference Based Objects

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

```
library("paradox")
  ps = ParamSet$new()
  ps2 = ps
 ps3 = ps$clone(deep = TRUE)
  print(ps) # the same for ps2 and ps3
<ParamSet>
Empty.
 ps$add(ParamLgl$new("a"))
print(ps) # ps was changed
<ParamSet>
         class lower upper nlevels
                                         default value
1: a ParamLgl
                 NA
                       NA
                                 2 <NoDefault[3]>
 print(ps2) # contains the same reference as ps
<ParamSet>
                                         default value
         class lower upper nlevels
1: a ParamLgl
                 NA
                       NA
                                 2 <NoDefault[3]>
print(ps3) # is a "clone" of the old (empty) ps
<ParamSet>
```

Empty.

9.4.2. Defining a Parameter Space

9.4.2.1. Single Parameters

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

The Param class has various subclasses that represent different value types:

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

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

```
library("paradox")
parA = ParamLgl$new(id = "A")
parB = ParamInt$new(id = "B", lower = 0, upper = 10, tags = c("tag1", "tag2"))
parC = ParamDbl$new(id = "C", lower = 0, upper = 4, special_vals = list(NULL))
parD = ParamFct$new(id = "D", levels = c("x", "y", "z"), default = "y")
parE = ParamUty$new(id = "E", custom_check = function(x) checkmate::checkFunction(x))
```

Every parameter must have:

- id A name for the parameter within the parameter set
- default A default value
- special vals A list of values that are accepted even if they do not conform to the type
- tags Tags that can be used to organize parameters

The numeric (Int and Db1) parameters furthermore allow for specification of a **lower** and **upper** bound. Meanwhile, the Fct parameter must be given a vector of **levels** that define the possible states its parameter can take. The Uty parameter can also have a **custom_check** function that must return TRUE when a value is acceptable and may return a **character(1)** error description otherwise. The example above defines parE as a parameter that only accepts functions.

All values which are given to the constructor are then accessible from the object for inspection using \$. Although all these values can be changed for a parameter after construction, this can be a bad idea and should be avoided when possible.

Instead, a new parameter should be constructed. Besides the possible values that can be given to a constructor, there are also the \$class, \$nlevels, \$is_bounded, \$has_default, \$storage_type, \$is_number and \$is_categ slots that give information about a parameter.

A list of all slots can be found in ?Param.

```
1 parB$lower
```

9. Technical

[1] 0

```
parA$levels

[1] TRUE FALSE

parE$class
```

[1] "ParamUty"

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

```
as.data.table(parA)
```

```
id class lower upper levels nlevels is_bounded special_vals
1: A ParamLgl NA NA TRUE, FALSE 2 TRUE <list[0]>
3 variables not shown: [default, storage_type, tags]
```

9.4.2.1.1. Type / Range Checking

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

```
parA$test(FALSE)
```

[1] TRUE

```
parA$test("FALSE")
```

[1] FALSE

```
parA$check("FALSE")
```

[1] "Must be of type 'logical flag', not 'character'"

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

9.4.2.2. Parameter Sets

The ordered collection of parameters is handled in a ParamSet¹. It is initialized using the \$new() function and optionally takes a list of Params as argument. Parameters can also be added to the constructed ParamSet using the \$add() function. It is even possible to add whole ParamSets to other ParamSets.

```
ps = ParamSet$new(list(parA, parB))
ps$add(parC)
ps$add(ParamSet$new(list(parD, parE)))
print(ps)
```

<ParamSet>

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

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

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

```
psSmall = ps$clone()
psSmall$subset(c("A", "B", "C"))
print(psSmall)
```

<ParamSet>

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

Just as for Params, and much more useful, it is possible to get the ParamSet as a data.table using as.data.table(). This makes it easy to subset parameters on certain conditions and aggregate information about them, using the variety of methods provided by data.table.

```
as.data.table(ps)
```

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

```
levels nlevels is_bounded special_vals
   id
         class lower upper
1:
   A ParamLgl
                  NA
                        NA
                            TRUE, FALSE
                                             2
                                                     TRUE
                                                             t[0]>
   B ParamInt
                   0
                        10
                                            11
                                                     TRUE
                                                             t[0]>
2:
3:
   C ParamDbl
                   0
                         4
                                                             t[1]>
                                           Inf
                                                     TRUE
   D ParamFct
4:
                  NA
                        NA
                                             3
                                                     TRUE
                                                             t[0]>
                                 x,y,z
   E ParamUty
                                                             t[0]>
                  NA
                        NA
                                           Inf
                                                    FALSE
3 variables not shown: [default, storage type, tags]
```

9.4.2.2.1. Type / Range Checking

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

```
ps$check(list(A = TRUE, B = 0, E = identity))
```

[1] TRUE

```
ps$check(list(A = 1))
```

[1] "A: Must be of type 'logical flag', not 'double'"

```
ps$check(list(Z = 1))
```

[1] "Parameter 'Z' not available. Did you mean 'A' / 'B' / 'C'?"

9.4.2.2.2. Values in a ParamSet

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

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

```
$A
[1] TRUE
$B
[1] 1
```

The parameter constraints are automatically checked:

```
ps$values$B = 100
```

Error in self\$assert(xs): Assertion on 'xs' failed: B: Element 1 is not <= 10.

9.4.2.2.3. Dependencies

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

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

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

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

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

ps$check(list(A = FALSE, D = "x", B = 1))  # OK: all dependencies met
```

[1] TRUE

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

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

9. Technical

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

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

```
ps$check(list(A = FALSE, D = "z")) # OK: B is absent
```

[1] TRUE

```
ps$check(list(A = TRUE)) # OK: neither B nor D present
```

[1] TRUE

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

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

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

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

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

```
ı ps$deps
```

```
id on cond
1: D A <CondEqual[9]>
2: B D <CondAnyOf[9]>
```

9.4.2.3. Vector Parameters

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

```
ps2d = ParamDbl$new("x", lower = 0, upper = 1)$rep(2)
 print(ps2d)
<ParamSet>
              class lower upper nlevels
                                                default value
        id
1: x_rep_1 ParamDbl
                        0
                               1
                                     Inf <NoDefault[3]>
2: x_rep_2 ParamDbl
                        0
                               1
                                     Inf <NoDefault[3]>
  ps$add(ps2d)
  print(ps)
<ParamSet>
              class lower upper nlevels
        id
                                                default parents value
1:
                       NA
                              NA
                                       2 <NoDefault[3]>
                                                                  TRUE
         A ParamLgl
2:
         B ParamInt
                        0
                              10
                                      11 <NoDefault[3]>
                                                               D
                                                                     1
         C ParamDbl
3:
                              4
                                     Inf <NoDefault[3]>
4:
         D ParamFct
                       NΑ
                              NA
                                       3
                                                               Α
5:
         E ParamUty
                       NA
                                     Inf <NoDefault[3]>
                              NA
                                     Inf <NoDefault[3]>
6: x_rep_1 ParamDbl
                        0
                               1
```

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

Inf <NoDefault[3]>

7: x_rep_2 ParamDbl

0

1

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

```
repeated parameters.

ps$tags

$A
character(0)

$B
[1] "tag1" "tag2"

$C
character(0)

$D
character(0)
```

9. Technical

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

9.4.3. Parameter Sampling

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

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

9.4.3.1. Parameter Designs

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

9.4.3.2. Grid Design

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

```
design = generate_design_grid(psSmall, 2)
print(design)
```

```
7: FALSE 10 0
8: FALSE 10 4
```

```
generate_design_grid(psSmall, param_resolutions = c(B = 1, C = 2))
```

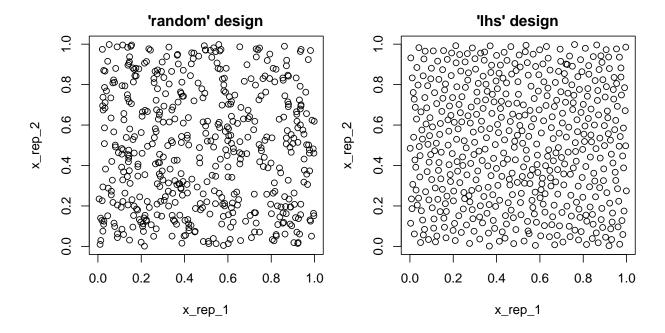
<Design> with 4 rows:

B C A
1: 0 0 TRUE
2: 0 0 FALSE
3: 0 4 TRUE
4: 0 4 FALSE

9.4.3.3. Random Sampling

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

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



9.4.3.4. Generalized Sampling: The Sampler Class

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

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

9.4.3.4.1. 1D-Samplers

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

9.4.3.4.2. Hierarchical Sampler

The SamplerHierarchical sampler is an auxiliary sampler that combines many 1D-Samplers to get a combined distribution. Its name "hierarchical" implies that it is able to respect parameter dependencies. This suggests that parameters only get sampled when their dependencies are met.

The following example shows how this works: The Int parameter B depends on the Lgl parameter A being TRUE. A is sampled to be TRUE in about half the cases, in which case B takes a value between 0 and 10. In the cases where A is FALSE, B is set to NA.

```
psSmall$add_dep("B", "A", CondEqual$new(TRUE))
sampH = SamplerHierarchical$new(psSmall,
list(Sampler1DCateg$new(parA),
Sampler1DUnif$new(parB),
Sampler1DUnif$new(parC))

sampled = sampH$sample(1000)
table(sampled$data[, c("A", "B")], useNA = "ifany")
```

```
В
            0
Α
                 1
                      2
                           3
                                    5
                                         6
                                              7
                                                   8
                                                            10 <NA>
  FALSE
            0
                 0
                      0
                           0
                                0
                                    0
                                         0
                                              0
                                                   0
                                                        0
                                                             0
                                                                 493
  TRUE
           51
               49
                    50
                         42
                              47
                                   47
                                        44
                                             52
                                                       49
                                                            41
                                                                   0
                                                  35
```

9.4.3.4.3. Joint Sampler

Another way of combining samplers is the SamplerJointIndep. SamplerJointIndep also makes it possible to combine Samplers that are not 1D. However, SamplerJointIndep currently can not handle ParamSets with dependencies.

```
sampJ = SamplerJointIndep$new(
list(Sampler1DUnif$new(ParamDbl$new("x", 0, 1)),
Sampler1DUnif$new(ParamDbl$new("y", 0, 1)))
sampJ$sample(5)
```

<Design> with 5 rows:

```
x y
1: 0.9980689 0.29325189
2: 0.5696531 0.14916164
3: 0.7999309 0.87583963
4: 0.9762024 0.06155586
5: 0.4334808 0.60938294
```

9.4.3.4.4. SamplerUnif

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

9.4.4. Parameter Transformation

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

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

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

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

```
psexp = ParamSet$new(list(ParamDbl$new("par", 0, 1)))
  psexp$trafo = function(x, param_set) {
    xpar = -log(xpar)
  }
  design = generate_design_random(psexp, 2)
 print(design)
<Design> with 2 rows:
          par
1: 0.07302144
2: 0.15870426
design$transpose() # trafo is TRUE
[[1]]
[[1]]$par
[1] 2.617002
[[2]]
[[2]]$par
[1] 1.840713
Compare this to $transpose() without transformation:
design$transpose(trafo = FALSE)
[[1]]
[[1]]$par
[1] 0.07302144
[[2]]
[[2]]$par
[1] 0.1587043
```

9.4.4.1. Transformation between Types

Usually the design created with one ParamSet is then used to configure other objects that themselves have a ParamSet which defines the values they take. The ParamSets which can be used for random sampling, however, are restricted in some ways: They must have finite bounds, and they may not contain "untyped" (ParamUty) parameters. \$trafo provides the glue for these situations. There

is relatively little constraint on the trafo function's return value, so it is possible to return values that have different bounds or even types than the original ParamSet. It is even possible to remove some parameters and add new ones.

Suppose, for example, that a certain method requires a function as a parameter. Let's say a function that summarizes its data in a certain way. The user can pass functions like median() or mean(), but could also pass quantiles or something completely different. This method would probably use the following ParamSet:

```
methodPS = ParamSet$new(
list(
ParamUty$new("fun",
custom_check = function(x) checkmate::checkFunction(x, nargs = 1))

print(methodPS)
```

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

```
samplingPS = ParamSet$new(
     list(
       ParamFct$new("fun", c("mean", "median", "min", "max"))
     )
   )
   samplingPS$trafo = function(x, param_set) {
     # x$fun is a `character(1)`,
     # in particular one of 'mean', 'median', 'min', 'max'.
     # We want to turn it into a function!
10
     x$fun = get(x$fun, mode = "function")
11
12
   }
13
   design = generate_design_random(samplingPS, 2)
  print(design)
```

```
<Design> with 2 rows:
   fun
1: max
2: min
```

9. Technical

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

```
xvals = design$transpose()
print(xvals[[1]])
```

\$fun

```
function (..., na.rm = FALSE) .Primitive("max")
```

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

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

[1] TRUE

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

[1] 10

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

```
design = generate_design_random(samplingPS2, 2)
print(design)
```

```
<Design> with 2 rows:
    quantile
1: 0.03543698
2: 0.65563727
```

The Design now contains the column "quantile" that will be used by the \$transpose function to create the fun parameter. We also check that it fits the requirement set by methodPS, and that it is a function.

```
1  xvals = design$transpose()
2  print(xvals[[1]])

$fun
function(input) quantile(input, x$quantile)
<environment: 0x55670e8f3358>

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

[1] TRUE

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

3.543698%
1.318933
```

9.4.5. Defining a Tuning Spaces

When running an optimization, it is important to inform the tuning algorithm about what hyperparameters are valid. Here the names, types, and valid ranges of each hyperparameter are important. All this information is communicated with objects of the class ParamSet, which is defined in paradox. While it is possible to create ParamSet-objects using its \$new-constructor, it is much shorter and readable to use the ps-shortcut, which will be presented here. For an in-depth description of paradox and its classes, see Section 9.4.

Note, that ParamSet objects exist in two contexts. First, ParamSet-objects are used to define the space of valid parameter settings for a learner (and other objects). Second, they are used to define a search space for tuning. We are mainly interested in the latter. For example we can consider the minsplit parameter of the classif.rpart Learner. The ParamSet associated with the learner has a lower but no upper bound. However, for tuning the value, a lower and upper bound must be given because tuning search spaces need to be bounded. For Learner or PipeOp objects, typically "unbounded" ParamSets are used. Here, however, we will mainly focus on creating "bounded" ParamSets that can be used for tuning. See Section 9.4 for more details on using ParamSets to define parameter ranges for use-cases besides tuning.

9.4.5.1. Creating ParamSets

An empty "ParamSet") - not yet very useful - can be constructed using just the "ps") call:

9. Technical

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

<ParamSet> Empty.

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

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

<ParamSet>

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

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

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

These domain constructors each take some of the following arguments:

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

- **special_vals**: Valid values besides the normally accepted values for a parameter. Not used for tuning search spaces.
- **custom_check**: Function that checks whether a value given to **p_uty** is valid. Not used for tuning search spaces.

The lower and upper parameters are always in the first and second position respectively, except for p_fct where levels is in the first position. It is preferred to omit the labels (ex: upper = 0.1 becomes just 0.1). This way of defining a ParamSet is more concise than the equivalent definition above. Preferred:

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

9.4.5.2. Transformations (trafo)

We can use the paradox function generate_design_grid to look at the values that would be evaluated by grid search. (We are using rbindlist() here because the result of \$transpose() is a list that is harder to read. If we didn't use \$transpose(), on the other hand, the transformations that we investigate here are not applied.) In generate_design_grid(search_space, 3), search_space is the ParamSet argument and 3 is the specified resolution in the parameter space. The resolution for categorical parameters is ignored; these parameters always produce a grid over all of their valid levels. For numerical parameters the endpoints of the params are always included in the grid, so if there were 3 levels for the kernel instead of 2 there would be 9 rows, or if the resolution was 4 in this example there would be 8 rows in the resulting table.

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

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

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

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

)
```

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

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

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

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

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

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

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

```
search_space = ps(
     class.weights = p_dbl(0.1, 0.9, trafo = function(x) c(spam = x, nonspam = 1 - x))
3
  generate_design_grid(search_space, 3)$transpose()
[[1]]
[[1]]$class.weights
   spam nonspam
    0.1
            0.9
[[2]]
[[2]]$class.weights
   spam nonspam
    0.5
            0.5
[[3]]
[[3]]$class.weights
   spam nonspam
    0.9
            0.1
```

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

9.4.6. Automatic Factor Level Transformation

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

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

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

9. Technical

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

This is equivalent to the following:

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

Note: Though the resolution is 3 here, in this case it doesn't matter because both **cost** and **kernel** are factors (the resolution for categorical variables is ignored, these parameters always produce a grid over all their valid levels).

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

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

typeof(search_space$params$cost$levels)
```

[1] "character"

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

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

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

[[1]]
[[1]]$class.weights
```

```
0.5 0.5

[[2]]

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

spam nonspam

9.4.6.1. Parameter Dependencies (depends)

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

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

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

```
cost kernel degree
1: 0.1 polynomial 1
2: 0.1 polynomial 2
3: 0.1 polynomial 3
```

9. Technical

```
4:
     0.1
             radial
                         NA
     1.0 polynomial
                          1
    1.0 polynomial
                          2
 7:
     1.0 polynomial
                          3
8:
    1.0
             radial
                         NA
9: 10.0 polynomial
                          1
10: 10.0 polynomial
                          2
11: 10.0 polynomial
                          3
12: 10.0
             radial
                         NA
```

9.4.6.2. Creating Tuning ParamSets from other ParamSets

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

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

```
learner = lrn("classif.svm")
  learner$param set$values$kernel = "polynomial" # for example
  learner$param_set$values$degree = to_tune(lower = 1, upper = 3)
  print(learner$param_set$search_space())
<ParamSet>
             class lower upper nlevels
                                               default value
1: degree ParamInt
                       1
                             3
                                     3 <NoDefault[3]>
  rbindlist(generate_design_grid(
    learner$param_set$search_space(), 3)$transpose()
  )
  degree
1:
        1
```

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

2: 3: 2

3

```
learner$param_set$values$shrinking = to_tune()
2
  print(learner$param_set$search_space())
<ParamSet>
           id
                 class lower upper nlevels
                                                    default value
      degree ParamInt
                                  3
                                          3 <NoDefault[3]>
1:
                            1
                                          2
2: shrinking ParamLgl
                          NA
                                 NA
                                                       TRUE
  rbindlist(generate_design_grid(
     learner$param_set$search_space(), 3)$transpose()
  )
   degree shrinking
        1
                TRUE
1:
2:
        1
               FALSE
3:
        2
                TRUE
        2
4:
               FALSE
5:
        3
                TRUE
6:
        3
               FALSE
```

"to_tune") can also be constructed with a <code>Domain</code> object, i.e. something constructed with a <code>p_****</code> call. This way it is possible to tune continuous parameters with discrete values, or to give trafos or dependencies. One could, for example, tune the <code>cost</code> as above on three given special values, and introduce a dependency of <code>shrinking</code> on it. Notice that a short form for <code>to_tune(<levels>)</code> is a short form of <code>to_tune(p_fct(<levels>))</code>.

Note

When introducing the dependency, we need to use the degree value from *before* the implicit trafo, which is the name or as.character() of the respective value, here "val2"!

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

<ParamSet>

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

```
rbindlist(generate_design_grid(learner$param_set$search_space(), 3)$transpose(), fill = TRUE
  degree cost shrinking
1:
        1 0.3
2:
        1 0.7
                    TRUE
3:
        1 0.7
                   FALSE
4:
        2 0.3
                      NA
        2 0.7
5:
                    TRUE
6:
        2 0.7
                   FALSE
7:
        3 0.3
                      NA
8:
        3 0.7
                    TRUE
        3 0.7
9:
                   FALSE
The "search_space() picks up dependencies from the underlying ParamSet automatically. So if
the kernel is tuned, then degree automatically gets the dependency on it, without us having to
specify that. (Here we reset cost and shrinking to NULL for the sake of clarity of the generated
output.)
  learner$param_set$values$cost = NULL
  learner$param_set$values$shrinking = NULL
  learner$param_set$values$kernel = to_tune(c("polynomial", "radial"))
  print(learner$param_set$search_space())
<ParamSet>
             class lower upper nlevels
                                               default parents value
1: degree ParamInt
                             3 3 <NoDefault[3] > kernel
                      1
2: kernel ParamFct
                            NA
                                2 <NoDefault[3]>
                      NA
```

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

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

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

```
learner$param_set$values$class.weights = to_tune(
    ps(spam = p_dbl(0.1, 0.9), nonspam = p_dbl(0.1, 0.9),
      .extra_trafo = function(x, param_set) list(c(spam = x$spam, nonspam = x$nonspam))
head(generate_design_grid(learner$param_set$search_space(), 3)$transpose(), 3)
[[1]]
[[1]]$kernel
[1] "polynomial"
[[1]]$degree
[1] 1
[[1]]$class.weights
   spam nonspam
   0.1
            0.1
[[2]]
[[2]]$kernel
[1] "polynomial"
[[2]]$degree
[1] 1
[[2]]$class.weights
  spam nonspam
   0.1
            0.5
[[3]]
[[3]]$kernel
[1] "polynomial"
[[3]]$degree
[1] 1
[[3]]$class.weights
   spam nonspam
   0.1
           0.9
```

9.5. Logging

We use the lgr package for logging and progress output.

9.5.1. Changing mlr3 logging levels

To change the setting for mlr3 for the current session, you need to retrieve the logger (which is a R6 object) from lgr, and then change the threshold of the like this:

```
requireNamespace("lgr")

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

The default log level is "info". All available levels can be listed as follows:

```
getOption("lgr.log_levels")
```

```
fatal error warn info debug trace 100 200 300 400 500 600
```

To increase verbosity, set the log level to a higher value, e.g. to "debug" with:

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

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

```
1 lgr::get_logger("mlr3")$set_threshold("warn")
```

lgr comes with a global option called "lgr.default_threshold" which can be set via options() to make your choice permanent across sessions.

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

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

9.5.2. Redirecting output

Redirecting output is already extensively covered in the documentation and vignette of lgr. Here is just a short example that adds an additional appender to log events into a temporary file in JSON format:

```
tf = tempfile("mlr3log_", fileext = ".json")

# get the logger as R6 object
logger = lgr::get_logger("mlr")
```

```
# add Json appender
logger$add_appender(lgr::AppenderJson$new(tf), name = "json")

# signal a warning
logger$warn("this is a warning from mlr3")

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

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

9.5.3. 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". This should only be done to investigate problems, however, and not for production use, because

- 1. this disables parallelization, and
- 2. this leads to different RNG behavior and therefore to results that are not reproducible when the debug mode is set.

10. Model Interpretation

The goal of this chapter is to present key methods that allow in-depth posthoc analysis of an already trained model. The methods presented are model-agnostic, i.e. they can be applied to models of different classes. When using predictive models in practice, it is often the case that high performance on a validation set is not enough. Users more and more often want to know which variables are important and how they influence the model's predictions. For the end user, such knowledge allows better utilisation of models in the decision-making process, e.g. by analysing different possible decision options. In addition, if the model's behaviour turns out to be in line with the domain knowledge or the user's intuition then the user's confidence in the prediction will increase. For the modeller, an in-depth analysis of the model allows undesirable model behaviour to be detected and corrected.

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 10.2, and
- DALEX presented in Section 10.3.

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.

10.1. 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)
```

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(),
```

probabil

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
00B 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 penguinsspecies. 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.

10.2. 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).

10.2.1. 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", "ye effect = FeatureEffects$new(model)

plot(effect, features = num_features)
```

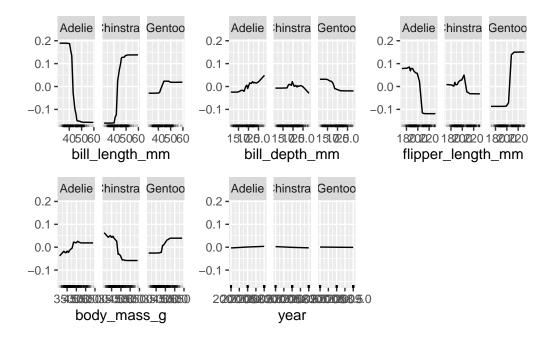


Figure 10.1.: Plot of the results from Feature Effects. Feature Effects computes and plots feature effects of prediction models

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.

10.2.2. 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)
```

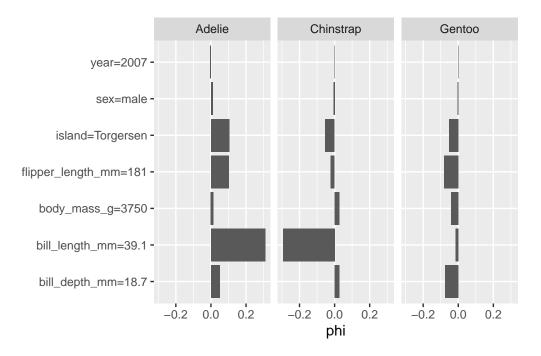


Figure 10.2.: Plot of the results from Shapley. ϕ gives the increase or decrease in probability given the values on the vertical axis

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.

10.2.3. FeatureImp

```
effect = FeatureImp$new(model, loss = "ce")
effect$plot(features = num_features)
```

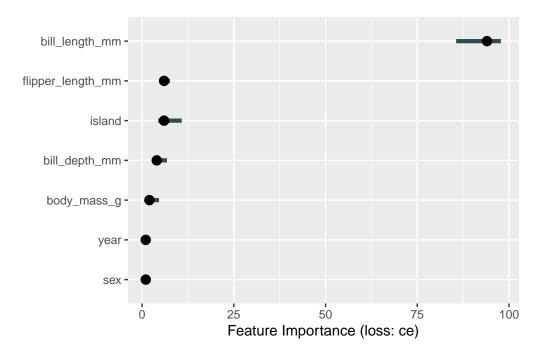


Figure 10.3.: Plot of the results from FeatureImp. FeatureImp visualizes the importance of features given the prediction model

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.

10.2.4. 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
```

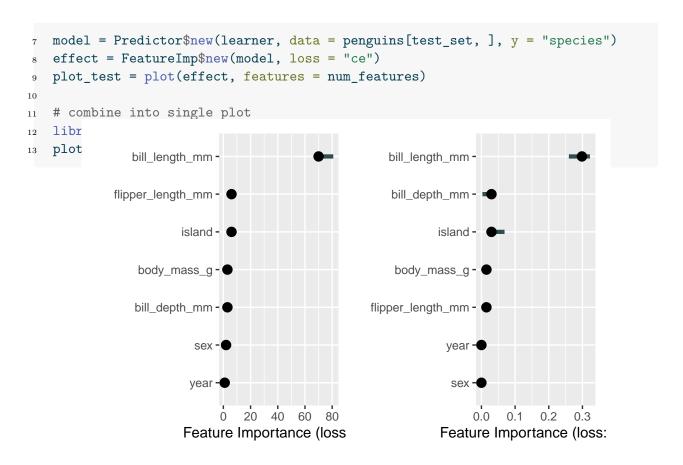


Figure 10.4.: FeatImp on train (left) and test (right)

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)
```

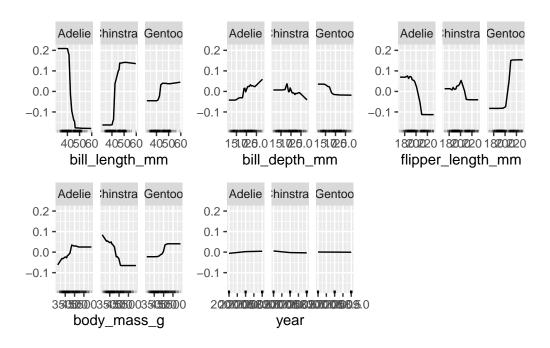


Figure 10.5.: FeatEffect train data set

```
model = Predictor$new(learner, data = penguins[test_set, ], y = "species")
effect = FeatureEffects$new(model)
plot(effect, features = num_features)
                   hinstra
            Adelie
                                                                    Adelie
                                                                           hinstra Gentoo
                          Gentoo
                                        Adelie
                                               hinstra
                                                      Gentoo
        0.2 -
                                    0.2 -
                                                               0.2 -
                                    0.1 -
        0.1 -
                                                               0.1 -
                                    0.0 -
        0.0 -
                                                                0.0
       -0.1
                                   -0.1 -
                                                               -0.1 -
       -0.2 -
                                   -0.2 -1 1 1 1
                                                               -0.2 - , , , , , , , ,
                                                                           1777771
                                       14620 14620 14620
             4050
                    4050
                           4050
                                                                  1890020089002008900200
              bill_length_mm
                                          bill_depth_mm
                                                                    flipper_length_mm
            Adelie
                   hinstra Gentoo
                                        Adelie
                                               hinstra Gentoo
        0.2 -
                                    0.2 -
        0.1 -
                                    0.1 -
        0.0
                                    0.0
       -0.1
                                   -0.1 -
       -0.2
                                   -0.2 -1 , 1 , 1 , 1 , 1 , 1 , 1 , 1
          30405050B0905050B09050000
                                     20000033900033900038.5.0
               body_mass_g
                                                year
```

Figure 10.6.: 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.

10.3. **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 10.1 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 interactive 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.

10.3.1. 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 10.7 for an overview of key functions that will be discussed.

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().

Explanatory Model Analysis DALEX::explain() Global Analysis Local Analysis Model Performance, AUC, RMSE Model Predict DALEX::predict() DALEX::model_performance() Shallow Feature Importance, VIP Feature Attributions, SHAP, BD DALEX::model_parts() DALEX::predict_parts() Deep Feature Profiles, PD, ALE Feature Profiles, Ceteris Paribus DALEX::model_profile() DALEX::predict_profile()

Figure 10.7.: 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.

```
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. 0.14.1 , 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.

10.3.2. 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_.

10.3.2.1. Model Performance

As shown in Figure Figure 10.7, 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.

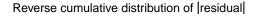
```
perf_penguin = model_performance(ranger_exp)
perf_penguin
```

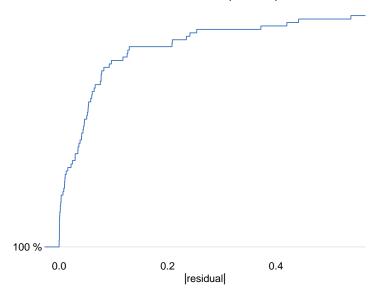
```
Measures for: multiclass micro_F1 : 1 macro_F1 : 1 w_macro_F1 : 1 accuracy : 1 w_macro_auc: 1 cross_entro: 6.034954
```

Residuals:

```
0% 10% 20% 30% 40% 50% 0.000000000 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")
location of theme = set_theme_dalex("ema")
location 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.

10.3.2.2. Permutational Variable Importance

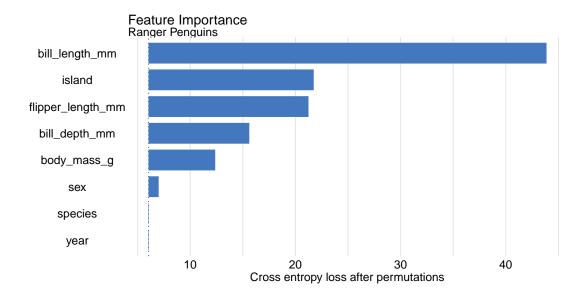
A popular technique for assessing variable importance in a model-agnostic manner is the permutation variable importance. It is based on the difference (or ratio) in the selected loss function after the selected variable or set of variables has been permuted. Read more about this technique in Variable-importance 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
   _full_model_
1
                          6.034954 Ranger Penguins
2
                          5.988560 Ranger Penguins
           year
3
                          6.034954 Ranger Penguins
        species
4
            sex
                          7.002289 Ranger Penguins
5
                         12.377824 Ranger Penguins
    body_mass_g
                         15.617252 Ranger Penguins
6 bill_depth_mm
```

```
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.

10.3.2.3. 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.

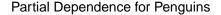
The DALEX::model_profile() function calculates the partial dependence profiles. The type argument of this function also allows *Marginal profiles* and *Accumulated Local profiles* to be calculated. Again, the result of the explanation can be model_profile with the generic function plot().

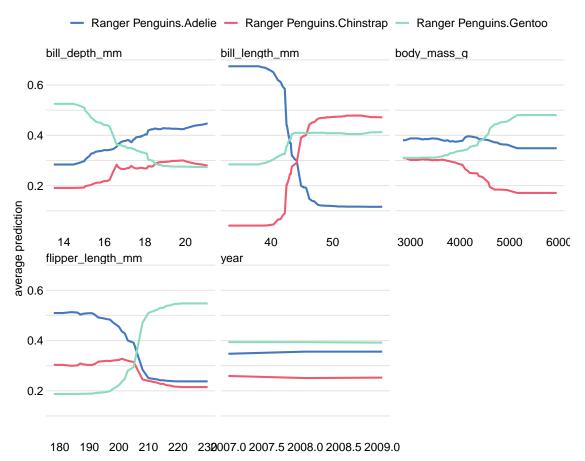
```
ranger_profiles = model_profile(ranger_exp)
ranger_profiles
```

```
Top profiles
                                                     _yhat_ _ids_
        vname
                                  label
                                             _x_
                   Ranger Penguins. Adelie 13.500 0.2839077
1 bill_depth_mm
2 bill_depth_mm Ranger Penguins.Chinstrap 13.500 0.1908264
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
                   Ranger Penguins.Gentoo 13.566 0.5252659
                                                                0
6 bill_depth_mm
```

```
plot(ranger_profiles) +
theme(legend.position = "top") +
ggtitle("Partial Dependence for Penguins","")
```

10. Model Interpretation





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.

10.3.3. 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_1~1 body_~2 sex
                                                                             year
  <fct>
          <fct>
                             <dbl>
                                           <dbl>
                                                        <int>
                                                                <int> <fct> <int>
1 Adelie Torgersen
                              39.1
                                             18.7
                                                          181
                                                                 3750 male
                                                                             2007
# ... with abbreviated variable names 1: flipper_length_mm, 2: body_mass_g
```

10.3.3.1. Model Prediction

As shown in Figure Figure 10.7, 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)
```

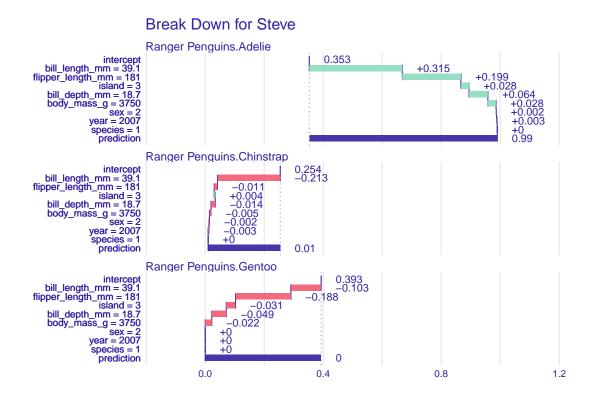
```
Adelie Chinstrap Gentoo [1,] 0.9900897 0.009910317 0
```

10.3.3.2. 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")
```



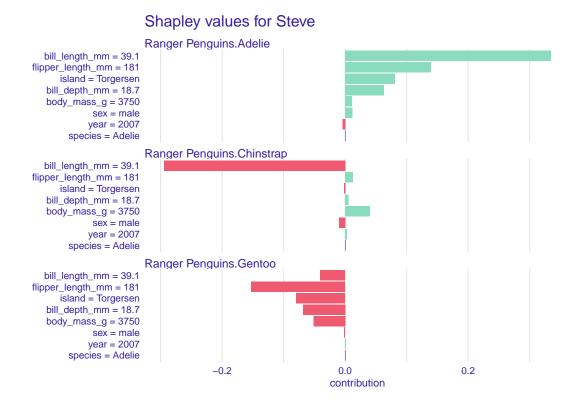
Looking at the plots above, we can read that the biggest contributors to the final prediction were for Steve the variables bill length and flipper.

10.3.3.3. 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.

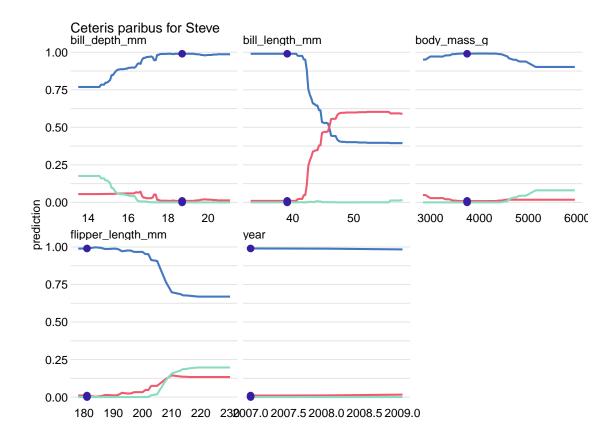
10.3.3.4. Ceteris Paribus

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", " ")
```

10. Model Interpretation



Blue dot stands for the prediction for Steve. Only a big change in bill length could convince the model of Steve's different species.

TODO (150-200 WORDS)

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

The approach is always the same:

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

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

11.1. Adding new Learners

Although many learners are already included in the mlr3 ecosystem, there might be a situation in which your algorithm of choice is not implemented. Here, we show how to create a custom mlr3learner step-by-step using mlr3extralearners::create_learner. If you intend to add a learner to mlr3extralearners, it is strongly recommended to first open a learner request issue to inform the mlr3 team about your idea. This allows to discuss the implementation details and potential peculiarities of the learner before putting actual work in.

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

Summary of steps for adding a new learner

- 1. Check that the learner does not already exist here.
- 2. Install mlr3extralearners and if you want to create a PR, also fork and clone it.
- 3. Run mlr3extralearners::create learner().
- 4. Add the learner's ParamSet.
- 5. Manually add .train and .predict private methods to the learner.
- 6. Implement supported optional extractors and hotstarting if applicable.
- 7. Fill out the missing parts in the learner's description. To add references you first need to create an entry in bibentries.R
- 8. Check that unit tests and parameter tests pass (these are automatically created).
- 9. Run cleaning functions.
- 10. Open a pull request with the "new learner" template.



Warning

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

11.1.1. Setting-up mlr3extralearners

In order to use mlr3extralearners::create_learner you must have mlr3extralearners installed. Note that mlr3extralearners is not on CRAN and has to be installed from GitHub. To ensure that you have the latest version, run remotes::install_github("mlr-org/mlr3extralearners") before proceeding.

If you want to create a pull request to mlr3extralearners you also need to

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

11.1.2. Calling create_learner

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

```
library("mlr3extralearners")
   create_learner(
     path = "path/to/a/folder",
     classname = "Rpart",
     type = "classif",
     key = "rpart",
     algorithm = "Decision Tree",
     package = "rpart",
     caller = "rpart",
     feature_types = c("logical", "integer", "numeric", "factor", "ordered"),
10
     predict_types = c("response", "prob"),
11
     properties = c("importance", "missings", "multiclass", "twoclass", "weights"),
12
     gh_name = "RaphaelS1",
13
     label = "Regression and Partition Tree",
14
     data_formats = "data.table"
15
   )
16
```

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

- 1. path = "path/to/a/folder" This determines where the templates are being generated. If the path is the root of an R package, the learner is created in the ./R directory and the test files in ./tests/testthat. Otherwise, all files are being created in the folder pointed to by path. Already existing files will not be modified.
- 2. classname = "Rpart" Set the R6 class name to LearnerClassifRpart (classif is below)

- 3. algorithm = "Decision Tree" Create the title as "Classification Decision Tree Learner", where "Classification" is determined automatically from type and "Learner" is added for all learners.
- 4. type = "classif" Setting the learner as a classification learner, automatically filling the title, class name, id ("classif.rpart") and task type.
- 5. key = "rpart" Used with type to create the unique ID of the learner, "classif.rpart".
- 6. package = "rpart" Setting the package from which the learner is implemented, this fills in things like the training function (along with caller) and the man field.
- 7. caller = "rpart" This tells the .train function, and the description which function is called to run the algorithm, with package this automatically fills rpart::rpart.
- 8. feature_types = c("logical", "integer", "numeric", "factor", "ordered") Sets the type of features that can be handled by the learner. See meta information.
- 9. predict_types = c("response", "prob"), Sets the available prediction types as response (pointwise prediction) and prob (probabilities). See meta information.
- 10. properties = c("importance", "missings", "multiclass", "twoclass", "weights") Sets the properties the learner supports. By including "importance" a public method called importance will be created that must be manually filled. See meta information and optional extractors.
- 11. gh_name = "RaphaelS1" Fills the '@author' tag with my GitHub handle, this is required as it identifies the maintainer of the learner.

The sections below show an exemplary execution of mlr3extralearners::create_learner.

11.1.3. learner_package_type_key.R

The first generated file which must be updated after running create_learner() is the one following the structure learner_<package>_<type>_<key>.R; in this example learner_rpart_classif_rpart.R.

For our example, the resulting script looks like this:

```
#' @title Classification Decision Tree Learner
#' @author RaphaelS1
#' @name mlr_learners_classif.rpart
#'
#'
#' @description
#' FIXME: BRIEF DESCRIPTION OF THE LEARNER.
#' Calls [rpart::rpart()] from FIXME: (CRAN VS NO CRAN): \CRANpkg{rpart} | 'rpart'.
#'
#' @section Custom mlr3 parameters:
#' #' @section Custom mlr3 parameters:
#' #' @section Custom mlr3 defaults:
#'
#' @section Custom mlr3 defaults:
#' #' @section Custom mlr3 defaults:
#' #' @section Installation:
#' FIXME: DEVIATIONS FROM UPSTREAM DEFAULTS. DELETE IF NOT APPLICABLE.
#'
#' @section Installation:
#' FIXME: CUSTOM INSTALLATION INSTRUCTIONS. DELETE IF NOT APPLICABLE.
#'
#'
```

```
#' @templateVar id classif.rpart
   #' @template learner
19
   # '
20
   #' @references
21
   #' FIXME: ADD REFERENCES
   # 1
23
   #' @template seealso_learner
   #' @template example
25
   #' @export
26
   LearnerClassifRpart = R6Class("LearnerClassifRpart",
27
     inherit = LearnerClassif,
28
     public = list(
29
       #' @description
30
       #' Creates a new instance of this [R6] [R6::R6Class] class.
31
       initialize = function() {
32
          # FIXME: MANUALLY ADD PARAMETERS BELOW AND THEN DELETE THIS LINE
33
         param_set = ps()
34
35
          # FIXME: MANUALLY UPDATE PARAM VALUES BELOW IF APPLICABLE THEN DELETE THIS LINE.
36
         param_set$values = list()
37
          super$initialize(
39
            id = "classif.rpart",
40
            packages = "rpart",
41
            feature_types = c("logical", "integer", "numeric", "factor", "ordered"),
42
            predict_types = c("response", "prob"),
43
            param_set = param_set,
44
            properties = c("importance", "missings", "multiclass", "twoclass", "weights"),
45
            man = "mlr3extralearners::mlr_learners_classif.rpart",
46
            label = "Regression and Partition Tree"
47
48
       },
49
       # FIXME: ADD IMPORTANCE METHOD IF APPLICABLE AND DELETE OTHERWISE
50
       # SEE mlr3extralearners::LearnerRegrRandomForest FOR AN EXAMPLE
51
       #' @description
52
       #' The importance scores are extracted from the slot FIXME:.
       #' @return Named `numeric()`.
       importance = function() {
55
         pars = self$param_set$get_values(tags = "importance")
56
          # FIXME: Implement importance
57
       }
58
     ),
59
     private = list(
60
        .train = function(task) {
61
          # get parameters for training
62
         pars = self$param_set$get_values(tags = "train")
63
64
```

```
# FIXME: IF LEARNER DOES NOT HAVE 'weights' PROPERTY THEN DELETE THESE LINES.
65
          if ("weights" %in% task$properties) {
66
            # Add weights to learner
67
          }
69
          # FIXME: CREATE OBJECTS FOR THE TRAIN CALL
          # AT LEAST "data" AND "formula" ARE REQUIRED
71
          formula = task$formula()
          data = task$data()
73
74
          # FIXME: HERE IS SPACE FOR SOME CUSTOM ADJUSTMENTS BEFORE PROCEEDING TO THE
75
          # TRAIN CALL. CHECK OTHER LEARNERS FOR WHAT CAN BE DONE HERE
76
          # USE THE mlr3misc::invoke FUNCTION (IT'S SIMILAR TO do.call())
          invoke(
79
            rpart::rpart,
80
            formula = formula,
81
            data = data,
82
             .args = pars
83
          )
        },
85
        .predict = function(task) {
86
          # get parameters with tag "predict"
87
          pars = self$param_set$get_values(tags = "predict")
88
          # get newdata and ensure same ordering in train and predict
90
          newdata = ordered_features(task, self)
91
92
          # Calculate predictions for the selected predict type.
          type = self$predict_type
94
95
          pred = invoke(predict, self$model, newdata = newdata, type = type, .args = pars)
96
97
          # FIXME: ADD PREDICTIONS TO LIST BELOW
98
          list()
99
        }
100
      )
101
102
103
    .extralrns_dict$add("classif.rpart", LearnerClassifRpart)
104
```

Now we have to do the following (the description will be addressed later):

- 1. Add the learner's parameters to the ParamSet.
- 2. Optionally change default values for the parameters.
- 3. Fill in the private .train method, which takes a (filtered) Task and returns a model.
- 4. Fill in the private .predict method, which operates on the model in self\$model (stored during \$train()) and a (differently subsetted) Task to return a named list of predictions.

5. As we included "importance" in properties, we have to add the public method importance() which returns a named numeric vectors with the decreasingly sorted importance scores (values) for the different features (names).

11.1.4. Meta-information

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

- id: The ID of the new learner. Usually consists of <type>.<key>, for example: "classif.rpart".
- packages: The upstream package name(s) of the implemented learner.
- param_set: A set of hyperparameters and their descriptions provided as a paradox::ParamSet. For each hyperparameter the appropriate class needs to be chosen. When using the paradox::ps shortcut, a short constructor of the form p_*** can be used:

```
- paradox::ParamLgl / paradox::p_lgl for scalar logical hyperparameters.
```

- paradox::ParamInt / paradox::p_int for scalar integer hyperparameters.
- paradox::ParamDbl / paradox::p_dbl for scalar numeric hyperparameters.
- paradox::ParamFct / paradox::p_fct for scalar factor hyperparameters (this includes characters).
- paradox::ParamUty / paradox::p_uty for everything else (e.g. vector parameters or list parameters).
- predict_types: Set of predict types the learner supports. These differ depending on the type of the learner. See mlr_reflections\$learner_predict_types for the full list of predict types supported by mlr3.

- LearnerClassif

- * response: Only predicts a class label for each observation.
- * prob: Also predicts the posterior probability for each class for each observation.

- LearnerRegr

- * response: Only predicts a numeric response for each observation.
- * se: Also predicts the standard error for each value of response.

- LearnerSurv

- * 1p Linear predictor calculated as the fitted coefficients multiplied by the test data.
- * distr Predicted survival distribution, either discrete or continuous. Implemented in distr6.
- * crank Continuous risk ranking.
- * response Predicted survival time.

- LearnerDens

- * pdf- Predicts the probability density function.
- * cdf Predicts the cumulative distribution function.
- * distr Predicts a distribution as implemented in distr6.

- LearnerClust

- * partition Assigns the observation to a partition.
- * prob Returns a probability for each partition.

- feature_types: Set of feature types the learner is able to handle. See mlr_reflections\$task_feature_type for feature types supported by mlr3.
- properties: Set of properties of the learner. See mlr_reflections\$learner_properties for the full list of learner properties supported by mlr3. The list of properties includes:
 - "twoclass": The learner works on binary classification problems.
 - "multiclass": The learner works on multi-class classification problems.
 - "missings": The learner can natively handle missing values.
 - "weights": The learner can work on tasks which have observation weights / case weights.
 - "importance": The learner supports extracting importance values for features.
 - "selected_features": The learner supports extracting the features which were selected by the model.
 - "oob_error": The learner supports extracting the out of bag error.
 - "loglik": The learner supports extracting the log-likelihood of the learner.
 - "hotstart_forward": The learner allows to continue training the model e.g. by adding more trees to a random forest.
 - "hotstart_backward": The learner allows to "undo" some of the training, e.g. by removing some trees from a model.
- man: The roxygen identifier of the learner. This is used within the \$help() method of the super class to open the help page of the learner.
- label: The label of the learner. This should briefly describe the learner (similar to the description's title) and is for example used for printing.

11.1.5. ParamSet

The ParamSet is the set of hyperparameters used in model training and predicting, this is given as a paradox::ParamSet. The set consists of a list of hyperparameters, where each has a specific class for the hyperparameter type (see above). In addition, each parameter has one or more tags, that determine in which method they are used.

Beyond that there are other tags that serve specific purposes:

- The tag "threads" should be used (if applicable) to tag the parameter that determines the number of threads used for the learner's internal parallelization. This parameter can be set using set threads.
- The tag "required" should be used to tag parameters that must be provided for the algorithm to be executable.
- In case optional extractors are available, the can (although this is rarely the case) also have parameters and can be tagged accordingly.
- If hotstarting is available, the fidelity parameter should be tagged with "hotstart".

For classif.rpart the param_set can be defined as follows

```
param_set = ps(
minsplit = p_int(lower = 1L, default = 20L, tags = "train"),
minbucket = p_int(lower = 1L, tags = "train"),
cp = p_dbl(lower = 0, upper = 1, default = 0.01, tags = "train"),
```

```
maxcompete = p_int(lower = OL, default = 4L, tags = "train"),
maxsurrogate = p_int(lower = OL, default = 5L, tags = "train"),
maxdepth = p_int(lower = 1L, upper = 30L, default = 30L, tags = "train"),
susesurrogate = p_int(lower = OL, upper = 2L, default = 2L, tags = "train"),
surrogatestyle = p_int(lower = OL, upper = 1L, default = OL, tags = "train"),
xval = p_int(lower = OL, default = OL, tags = "train"),
keep_model = p_lgl(default = FALSE, tags = "train")

param_set$values = list(xval = OL)
```

Within mlr3 packages we suggest to stick to the shorthand notation above for consistency, however the param_set can be written with the underlying R6 classes as shown here

```
param_set = ParamSet$new(list(
ParamInt$new(id = "minsplit", default = 20L, lower = 1L, tags = "train"),
ParamInt$new(id = "minbucket", lower = 1L, tags = "train"),
ParamDbl$new(id = "cp", default = 0.01, lower = 0, upper = 1, tags = "train"),
ParamInt$new(id = "maxcompete", default = 4L, lower = 0L, tags = "train"),
ParamInt$new(id = "maxsurrogate", default = 5L, lower = 0L, tags = "train"),
ParamInt$new(id = "maxdepth", default = 30L, lower = 1L, upper = 30L, tags = "train"),
ParamInt$new(id = "usesurrogate", default = 2L, lower = 0L, upper = 2L, tags = "train"),
ParamInt$new(id = "surrogatestyle", default = 0L, lower = 0L, upper = 1L, tags = "train"),
ParamInt$new(id = "xval", default = 0L, lower = 0L, tags = "train"),
ParamLgl$new(id = "keep_model", default = FALSE, tags = "train")

param_set$values = list(xval = 0L)
```

You should read though the learner documentation to find the full list of available parameters. Just looking at some of these in this example:

- "cp" is numeric, has a feasible range of [0,1] and defaults to 0.01. The parameter is used during "train".
- "xval" is integer has a lower bound of 0, a default of 0 and the parameter is used during "train".
- "keep model" is logical with a default of FALSE and is used during "train".

In some rare cases you may want to change the default parameter values. You can do this by changing the param_set\$values. You can see we have done this for "classif.rpart" where the default for xval is changed to 0. Note that the default in the ParamSet is recorded as our changed default (0), and not the original (10). It is strongly recommended to only change the defaults if absolutely required, when this is the case add the following to the learner documentation:

```
#' @section Custom mlr3 defaults:
#' - `<parameter>`:
#' - Actual default: <value>
#' - Adjusted default: <value>
#' - Reason for change: <text>
```

11.1.6. Train function

The train function takes a Task as input and must return a model. Let's say we want to translate the following call of rpart::rpart() into code that can be used inside the .train() method.

First, we write something down that works completely without mlr3:

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

We need to pass the formula notation Species ~ ., the data and the hyperparameters. To get the hyperparameters, we call self\$param_set\$get_values(tag = "train") and thereby query all parameters that are using during "train". Then, the dataset is extracted from the Task. Because the learner has the property "weights", we insert the weights of the task if there are any.

Last, we call the upstream function rpart::rpart() with the data and pass all hyperparameters via argument .args using the mlr3misc::invoke() function. The latter is simply an optimized version of do.call() that we use within the mlr3 ecosystem.

```
.train = function(task) {
     pars = self$param_set$get_values(tags = "train")
2
     if ("weights" %in% task$properties) {
3
       pars$weights = task$weights$weight
     formula = task$formula()
     data = task$data()
     invoke(
       rpart::rpart,
       formula = formula,
10
       data = data,
11
        .args = pars
12
     )
13
   }
14
```

11.1.7. Predict function

The internal predict method .predict() also operates on a Task as well as on the fitted model that has been created by the train() call previously and has been stored in self\$model.

The return value is a Prediction object. We proceed analogously to what we did in the previous section. We start with a version without any mlr3 objects and continue to replace objects until we have reached the desired interface:

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

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

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

Next, we transition from data to a Task again and construct a list with the return type requested by the user, this is stored in the \$predict_type slot of a learner class. Note that the Task is automatically passed to the prediction object, so all you need to do is return the predictions! Make sure the list names are identical to the task predict types.

The final .predict() method is below, we could omit the pars line as there are no parameters with the "predict" tag but we keep it here to be consistent:

```
.predict = function(task) {
     pars = self$param_set$get_values(tags = "predict")
     # get newdata and ensure same ordering in train and predict
     newdata = ordered_features(task, self)
     if (self$predict_type == "response") {
       response = invoke(
          predict,
          self$model,
          newdata = newdata,
          type = "class",
10
          .args = pars
11
       )
13
       return(list(response = response))
14
     } else {
15
       prob = invoke(
16
         predict,
17
          self$model,
18
         newdata = newdata,
          type = "prob",
          .args = pars
21
22
       return(list(prob = prob))
23
     }
24
25
```

⚠ Warning

You cannot rely on the column order of the data returned by task\$data() as the order of columns may be different from the order of the columns during \$.train. The newdata line ensures the ordering is the same by calling the same order as in train!

11.1.8. Optional Extractors

Specific learner implementations are free to implement additional getters to ease the access of certain parts of the model in the inherited subclasses. The blueprint for these methods is only included in the generated learner template, if the property is set when calling <code>create_learner()</code>. The comments in the templates will include references to other learners that have this property and can be used as guiding examples. To determine whether these methods are applicable, one has to determine whether the learner supports this method in principle <code>and</code> whether it is implemented in the upstream package.

For the following operations, extractors are standardized:

- importance(...) Returns the feature importance score as numeric vector. The higher the score, the more important the variable. The returned vector is named with feature names and sorted in decreasing order. Note that the model might omit features it has not used at all. The learner must be tagged with property "importance".
- selected_features(...) Returns a subset of selected features as character(). The learner must be tagged with property "selected_features".
- oob_error(...) Returns the out-of-bag error of the model as numeric(1). The learner must be tagged with property "oob_error".
- loglik(...) Extracts the log-likelihood (c.f. stats::logLik()). The learner must be tagged with property "loglik".

In this case, we only have to implement the importance() method.

```
importance = function() {
   if (is.null(self$model)) {
     stopf("No model stored")
   }

importance = sort(self$model$variable.importance, decreasing = TRUE)
   if (is.null(importance)) {
     importance = mlr3misc::set_names(numeric())
   }
   return(importance)
}
```

11.1.9. Hotstarting

Some learners support resuming or continuing from an already fitted model. We assume that hotstarting is only possible if a single hyperparameter (also called the fidelity parameter, usually controlling the complexity or expensiveness) is altered and all other hyperparameters are identical. The fidelity parameters should be tagged with "hotstart". Examples are:

• Random Forest: Starting from a model with n trees, a random forest with n + k trees can be obtained by adding k trees ("hotstart_forward") and a random forest with n - k trees can be obtained by removing k trees ("hotstart_backward").

• Gradient Boosting: When having fitted a model with n iterations, we only need k iterations to obtain a model with n + k iterations. ("hotstart_forward").

For more information see HotstartStack.

11.1.10. Control objects/functions of learners

Some learners rely on a "control" object/function such as glmnet::glmnet.control(). Accounting for such depends on how the underlying package works:

- If the package forwards the control parameters via ... and makes it possible to just pass control parameters as additional parameters directly to the train call, there is no need to distinguish both "train" and "control" parameters.
- If the control parameters need to be passed via a separate argument, one can e.g. use formalArgs(glmnet::glmnet.control) to get the names of the control parameters and then extract them from the pars like shown below.

```
pars = self$param_set$get_values(tags = "train")
ii = names(pars) %in% formalArgs(glmnet::glmnet.control)
pars_ctrl = pars[ii]
pars_train = pars[!ii]
invoke([...], .args = pars_train, control = pars_ctrl)
```

11.1.11. Adding the description

Once the learner is implemented - and is not only intended for personal use - it's description should be filled out. Most steps should be clear from the instructions given in the template. For the section '@references', the entry first has to be added to the file bibentries.R, essentially by converting the bibtex file to a R bibentry function call.

11.1.12. Testing the learner

Once your learner is created, you are ready to start testing if it works, there are three types of tests: manual, unit and parameter.

11.1.12.1. Train and Predict

For a bare-bone check you can just try to run a simple train() call locally.

```
task = tsk("iris") # assuming a Classif learner
lrn = lrn("classif.rpart")
lrn$train(task)
p = lrn$predict(task)
```

```
5 p$confusion
```

If it runs without erroring, that's a very good start!

11.1.12.2. Autotest

To ensure that your learner is able to handle all kinds of different properties and feature types, we have written an "autotest" that checks the learner for different combinations of such.

The "autotest" setup is generated automatically by <code>create_learner()</code>. It will have a name with the form <code>test_package_type_key.R</code>, in our case this will actually be <code>test_rpart_classif_rpart.R</code>. This will create the following script, for which no changes are required to pass (assuming the learner was correctly created):

```
test_that("autotest", {
   learner = lrn("classif.rpart")
   expect_learner(learner)
   # note that you can skip tests using the exclude argument
   result = run_autotest(learner)
   expect_true(result, info = result$error)
}
```

For some learners that have required parameters, it is needed to set some values for required parameters after construction so that the learner can be run in the first place.

You can also exclude some specific test arrangements within the "autotest" via the argument exclude in the run_autotest() function. Currently the run_autotest() function lives in inst/testthat of the mlr3 and still lacks documentation. This should change in the near future.

To finally run the test suite, call devtools::test() or hit CTRL + Shift + T if you are using RStudio.

11.1.12.3. Checking Parameters

Some learners have a high number of parameters and it is easy to miss out on some during the creation of a new learner. In addition, if the maintainer of the upstream package changes something with respect to the arguments of the algorithm, the learner is in danger to break. Also, new arguments could be added upstream and manually checking for new additions all the time is tedious.

Therefore we have written a "Parameter Check" that runs regularly for every learner. This "Parameter Check" compares the parameters of the mlr3 ParamSet against all arguments available in the upstream function that is called during \$train() and \$predict(). Again the file is automatically created by create_learner(). This will be named like test_paramtest_package_type_key.R, so in our example test_paramtest_rpart_classif_rpart.R. When the .train function calls multiple functions (e.g. a control function as described above), a list of functions can be passed to the parameter test.

The test comes with an exclude argument that should be used to exclude and explain why certain arguments of the upstream function are not within the ParamSet of the mlr3learner. This will likely be required for all learners as common arguments like x, target or data are handled by the mlr3 interface and are therefore not included within the ParamSet.

However, there might be more parameters that need to be excluded, for example:

- Type dependent parameters, i.e. parameters that only apply for classification or regression learners.
- Parameters that are actually deprecated by the upstream package and which were therefore not included in the mlr3 ParamSet.

All excluded parameters should have a comment justifying their exclusion.

In our example, the final paramtest script looks like:

```
test_that("classif.rpart train", {
     learner = lrn("classif.rpart")
2
     # this can also be a list of functions
     fun = rpart::rpart
     exclude = c(
       "formula", # handled internally
       "model", # handled internally
       "data", # handled internally
       "weights", # handled by task
       "subset", # handled by task
10
       "na.action", # handled internally
11
       "method", # handled internally
12
       "x", # handled internally
       "y", # handled internally
       "parms", # handled internally
15
       "control", # handled internally
16
       "cost" # handled internally
17
     )
18
19
     paramtest = run_paramtest(learner, fun, exclude, tag = "train")
20
     expect_paramtest(paramtest)
21
22
23
   test_that("classif.rpart predict", {
24
     learner = lrn("classif.rpart")
25
     fun = rpart:::predict.rpart
26
     exclude = c(
27
       "object", # handled internally
       "newdata", # handled internally
       "type", # handled internally
30
       "na.action" # handled internally
31
     )
32
33
```

```
paramtest = run_paramtest(learner, fun, exclude, tag = "predict")
spect_paramtest(paramtest)
spect_paramtest(paramtest)
```

11.1.13. Package Cleaning

Once all tests are passing, run the following functions to ensure that the package remains clean and tidy

```
1. devtools::document(roclets = c('rd', 'collate', 'namespace'))
2. If you haven't done this before run: remotes::install_github('mlr-org/styler.mlr')
3. styler::style_pkg(style = styler.mlr::mlr_style)
4. usethis::use_tidy_description()
5. lintr::lint_package()
```

Please fix any errors indicated by lintr before creating a pull request. Finally ensure that all FIXME are resolved and deleted in the generated files.

You are now ready to add your learner to the mlr3 ecosystem! Simply open a pull request to https://github.com/mlr-org/mlr3extralearners/pulls with the new learner template and complete the checklist in there. Creating this request will trigger an automated workflow that checks whether various conditions (such as rcmdcheck::rcmdcheck()) are satisfied. Once the pull request is approved and merged, your learner will automatically appear on the package website.

11.1.14. Thanks and Maintenance

Thank you for contributing to the mlr3 ecosystem!

When you created the learner you would have given your GitHub handle, meaning that you are now listed as the learner author and maintainer. This means that if the learner breaks it is your responsibility to fix the learner - you can view the status of your learner here.

11.1.15. Learner FAQ

Question 1

How to deal with Parameters which have no default?

Answer

If the learner does not work without providing a value, set a reasonable default in param_set\$values, add tag "required" to the parameter and document your default properly.

Question 2

Where to add the package of the upstream package in the DESCRIPTION file?

Add it to the "Suggests" section.

Question 3

How to handle arguments from external "control" functions such as glmnet::glmnet_control()?

Answer

See "Control objects/functions of learners".

Question 4

How to document if my learner uses a custom default value that differs to the default of the upstream package?

Answer

If you set a custom default for the mlr3learner that does not cope with the one of the upstream package (think twice if this is really needed!), add this information to the help page of the respective learner.

You can use the following skeleton for this:

```
#' @section Custom mlr3 defaults:
#' * ` <parameter > `:
#' * Actual default: <value>
#' * Adjusted default: <value>
#' * Reason for change: <text>
```

Question 5

When should the "required" tag be used when defining Params and what is its purpose?

Answer

The "required" tag should be used when the following conditions are met:

- The upstream function cannot be run without setting this parameter, i.e. it would throw an error.
- The parameter has no default in the upstream function.

In mlr3 we follow the principle that every learner should be constructable without setting custom parameters. Therefore, if a parameter has no default in the upstream function, a custom value is usually set for this parameter in the mlr3learner (remember to document such changes in the help page of the learner).

Even though this practice ensures that no parameter is unset in an mlr3learner and partially removes the usefulness of the "required" tag, the tag is still useful in the following scenario:

If a user sets custom parameters after construction of the learner

```
lrn = lrn("<id>")
lrn*param_set*values = list("<param>" = <value>)
```

Here, all parameters besides the ones set in the list would be unset. See paradox::ParamSet for more information. If a parameter is tagged as "required" in the ParamSet, the call above would error and prompt the user that required parameters are missing.

Question 6

What is this error when I run devtools::load_all()

```
1 > devtools::load_all(".")
2 Loading mlr3extralearners
3 Warning message:
4 .onUnload failed in unloadNamespace() for 'mlr3extralearners', details:
5 call: vapply(hooks, function(x) environment(x)$pkgname, NA_character_)
6 error: values must be length 1,
7 but FUN(X[[1]]) result is length 0
```

Answer

This is not an error but a warning and you can safely ignore it!

11.2. Adding new Measures

In this section we showcase how to implement a custom performance measure.

A good starting point is writing down the loss function independently of mlr3 (we also did this in the mlr3measures package). Here, we illustrate writing measure by implementing the root of the mean squared error for regression problems:

```
root_mse = function(truth, response) {
    mse = mean((truth - response)^2)
    sqrt(mse)
}
root_mse(c(0, 0.5, 1), c(0.5, 0.5, 0.5))
```

[1] 0.4082483

In the next step, we embed the root_mse() function into a new R6 class inheriting from base classes MeasureRegr/Measure. For classification measures, use MeasureClassif. We keep it simple here and only explain the most important parts of the Measure class:

```
# properties, see below
12
            properties = character(),
13
14
            # required predict type of the learner
15
            predict_type = "response",
16
            # feasible range of values
            range = c(0, Inf),
19
20
            # minimize during tuning?
21
            minimize = TRUE
22
          )
23
        }
24
      ),
25
26
      private = list(
27
        # custom scoring function operating on the prediction object
28
        .score = function(prediction, ...) {
29
          root_mse = function(truth, response) {
30
            mse = mean((truth - response)^2)
            sqrt(mse)
32
          }
34
          root_mse(prediction$truth, prediction$response)
35
        }
36
      )
37
   )
38
```

This class can be used as template for most performance measures. If something is missing, you might want to consider having a deeper dive into the following arguments:

- properties: If you tag you measure with the property "requires_task", the Task is automatically passed to your .score() function (don't forget to add the argument Task in the signature). The same is possible with "requires_learner" if you need to operate on the Learner and "requires_train_set" if you want to access the set of training indices in the score function.
- aggregator: This function (defaulting to mean()) controls how multiple performance scores, i.e. from different resampling iterations, are aggregated into a single numeric value if average is set to micro averaging. This is ignored for macro averaging.
- predict_sets: Prediction sets (subset of ("train", "test")) to operate on. Defaults to the "test" set.

Finally, if you want to use your custom measure just like any other measure shipped with mlr3 and access it via the mlr_measures dictionary, you can easily add it:

```
mlr3::mlr_measures$add("root_mse", MeasureRootMSE)
```

Typically it is a good idea to put the measure together with the call to mlr_measures\$add() in a new R file and just source it in your project.

```
## source("measure_root_mse.R")
msr("root_mse")
```

<MeasureRootMSE:root_mse>

* Packages: mlr3 * Range: [0, Inf] * Minimize: TRUE * Average: macro * Parameters: list() * Properties: -

* Predict type: response

11.3. Adding new PipeOps

This section showcases how the mlr3pipelines package can be extended to include custom PipeOps. To run the following examples, we will need a Task; we are using the well-known "Iris" task:

```
library("mlr3")
task = tsk("iris")
task$data()
```

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

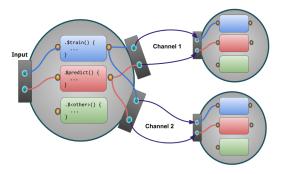
mlr3pipelines is fundamentally built around R6. When planning to create custom PipeOp objects, it can only help to familiarize yourself with it.

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

11.3.1. General Case Example: PipeOpCopy

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

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



11.3.1.1. First Steps: Inheriting from PipeOp

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

```
PipeOpCopyTwo = R6::R6Class("PipeOpCopyTwo",
     inherit = mlr3pipelines::PipeOp,
2
     public = list(
3
       initialize = function(id = "copy.two") {
       },
     ),
     private == list(
        .train = function(inputs) {
9
10
       },
11
12
        .predict = function(inputs) {
13
       }
15
     )
16
   )
17
```

Note, that **private** methods, e.g. .train and .predict etc are prefixed with a ...

11.3.1.2. Channel Definitions

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

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

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

```
initialize = function(id = "copy.two") {
     input = data.table::data.table(name = "input", train = "*", predict = "*")
     # the following will create two rows and automatically fill the `train`
     # and `predict` cols with "*"
     output = data.table::data.table(
       name = c("output1", "output2"),
       train = "*", predict = "*"
     super$initialize(id,
       input = input,
10
       output = output
11
     )
12
   }
13
```

11.3.1.3. Train and Predict

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

Two things to consider:

- The .train() function must always modify the self\$state variable to something that is not NULL or NO_OP. This is because the \$state slot is used as a signal that PipeOp has been trained on data, even if the state itself is not important to the PipeOp (as in our case). Therefore, our .train() will set self\$state = list().
- It is not necessary to "clone" our input or make deep copies, because we don't modify the data. However, if we were changing a reference-passed object, for example by changing data

in a Task, we would have to make a deep copy first. This is because a PipeOp may never modify its input object by reference.

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

```
1 .train = function(inputs) {
2    self$state = list()
3    c(inputs, inputs)
4  }

1 .predict = function(inputs) {
2    c(inputs, inputs)
3  }
```

11.3.1.4. Putting it Together

The whole definition thus becomes

```
PipeOpCopyTwo = R6::R6Class("PipeOpCopyTwo",
     inherit = mlr3pipelines::PipeOp,
     public = list(
       initialize = function(id = "copy.two") {
         super$initialize(id,
            input = data.table::data.table(name = "input", train = "*", predict = "*"),
           output = data.table::data.table(name = c("output1", "output2"),
                                 train = "*", predict = "*")
         )
       }
10
     ),
11
     private = list(
12
       .train = function(inputs) {
13
         self$state = list()
14
         c(inputs, inputs)
15
       },
16
17
        .predict = function(inputs) {
         c(inputs, inputs)
19
       }
     )
22
```

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()
```

11.3.2. Special Case: Preprocessing

gr = Graph\$new()

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

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

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

11.3.2.1. 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 <code>.predict_task()</code> 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, <code>mlr3</code> expects a <code>Learner</code> to always return just as many predictions as it was given input rows, so a <code>PipeOp</code> that removes <code>Task</code> rows during training can not be used inside a <code>GraphLearner</code>.

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,
2
     public = list(
       initialize = function(id = "drop.na") {
          super$initialize(id)
       }
     ),
     private = list(
9
        .train_task = function(task) {
10
          self$state = list()
11
          featuredata = task$data(cols = task$feature_names)
12
          exclude = apply(is.na(featuredata), 1, any)
13
          task$filter(task$row_ids[!exclude])
14
       },
15
16
        .predict_task = function(task) {
17
          # nothing to be done
          task
       }
20
21
     )
22
```

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

11.3.2.2. Example: PipeOpScaleAlways

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

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

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

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

```
PipeOpScaleAlways = R6::R6Class("PipeOpScaleAlways",
     inherit = mlr3pipelines::PipeOpTaskPreproc,
2
     public = list(
        initialize = function(id = "scale.always") {
          super$initialize(id = id)
       }
     ),
     private = list(
9
        .select_cols = function(task) {
10
          task$feature_types[type == "numeric", id]
11
       },
13
        .train_dt = function(dt, levels, target) {
14
          sc = scale(as.matrix(dt))
15
          self$state = list(
16
            center = attr(sc, "scaled:center"),
17
            scale = attr(sc, "scaled:scale")
          )
19
          SC
20
       },
21
22
        .predict_dt = function(dt, levels) {
23
          t((t(dt) - self$state$center) / self$state$scale)
24
       }
25
     )
26
   )
27
```

(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:
                                         -1.50149039 0.09788935
        setosa
                 -1.2791040 -1.3110521
  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
                                          0.79301235 -0.13153881
                  0.8168591
                              1.0504160
149: virginica
                  0.9301544
                              1.4439941
                                          0.43072244 0.78617383
150: virginica
                  0.7602115
                              0.7880307
                                          0.06843254 -0.13153881
```

11.3.3. Special Case: Preprocessing with Simple Train

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

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

A simplification to make is therefore to have a 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.

11.3.3.1. Example: PipeOpDropConst

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

The .get_state() function's result is saved to the \$state slot, so we want to return something that is useful for dropping features. We choose to save the names of all the columns that have nonzero variance. For brevity, we use length(unique(column)) > 1 to check whether more than

one distinct value is present; a more sophisticated version could have a tolerance parameter for numeric values that are very close to each other.

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

The full PipeOp could be written as follows:

```
PipeOpDropConst = R6::R6Class("PipeOpDropConst",
     inherit = mlr3pipelines::PipeOpTaskPreprocSimple,
2
     public = list(
       initialize = function(id = "drop.const") {
         super$initialize(id = id)
       }
     ),
     private = list(
9
       .get_state = function(task) {
10
         data = task$data(cols = task$feature_names)
11
         nonconst = sapply(data, function(column) length(unique(column)) > 1)
         list(cnames = colnames(data)[nonconst])
13
       },
14
15
       .transform = function(task) {
16
         task$select(self$state$cnames)
17
       }
     )
   )
```

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
   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

11.3.3.2. Example: PipeOpScaleAlwaysSimple

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

```
PipeOpScaleAlwaysSimple = R6::R6Class("PipeOpScaleAlwaysSimple",
     inherit = mlr3pipelines::PipeOpTaskPreprocSimple,
     public = list(
       initialize = function(id = "scale.always.simple") {
          super$initialize(id = id)
       }
6
     ),
     private = list(
        .select_cols = function(task) {
10
          task$feature_types[type == "numeric", id]
11
       },
13
        .get_state_dt = function(dt, levels, target) {
14
         list(
15
            center = sapply(dt, mean),
16
            scale = sapply(dt, sd)
17
          )
18
       },
19
        .transform_dt = function(dt, levels) {
21
          t((t(dt) - self$state$center) / self$state$scale)
22
       }
23
24
   )
25
```

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$data()
      Species Petal.Length Petal.Width Sepal.Length Sepal.Width
  1:
       setosa
                -1.3357516 -1.3110521 -0.89767388 1.01560199
  2:
                -1.3357516 -1.3110521 -1.13920048 -0.13153881
       setosa
  3:
       setosa
               -1.3923993 -1.3110521 -1.38072709 0.32731751
  4:
                -1.2791040 -1.3110521 -1.50149039 0.09788935
       setosa
  5:
                -1.3357516 -1.3110521 -1.01843718 1.24503015
       setosa
 ___
146: virginica
                 0.8168591
                             1.4439941
                                         1.03453895 -0.13153881
147: virginica
                 0.7035638
                             0.9192234
                                         0.55148575 -1.27867961
148: virginica
                 0.8168591
                             1.0504160
                                         0.79301235 -0.13153881
149: virginica
                 0.9301544
                             1.4439941
                                         0.43072244 0.78617383
150: virginica
                 0.7602115
                             0.7880307
                                         0.06843254 -0.13153881
 result_posa_simple$data()
      Species Petal.Length Petal.Width Sepal.Length Sepal.Width
                -1.3357516 -1.3110521 -0.89767388 1.01560199
  1:
       setosa
  2:
       setosa -1.3357516 -1.3110521 -1.13920048 -0.13153881
       setosa -1.3923993 -1.3110521 -1.38072709 0.32731751
  3:
  4:
       setosa
                -1.2791040 -1.3110521 -1.50149039 0.09788935
  5:
              -1.3357516 -1.3110521 -1.01843718 1.24503015
       setosa
146: virginica
                 0.8168591
                             1.4439941
                                         1.03453895 -0.13153881
147: virginica
                 0.7035638
                             0.9192234
                                         0.55148575 -1.27867961
148: virginica
                             1.0504160
                                         0.79301235 -0.13153881
                 0.8168591
149: virginica
                 0.9301544
                             1.4439941
                                         0.43072244 0.78617383
150: virginica
                 0.7602115
                             0.7880307
                                         0.06843254 -0.13153881
```

11.3.4. Hyperparameters

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

Hyperparameters are declared during initialization, when calling the PipeOp's \$initialize() function, by giving a param_set argument. The param_set must be a ParamSet from the paradox

package; see the tuning chapter or Section 9.4 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()".

11.3.4.1. Hyperparameter Example: PipeOpScale

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

```
PipeOpScale$public_methods$initialize

function (id = "scale", param_vals = list())
.__PipeOpScale__initialize(self = self, private = private, super = super,
    id = id, param_vals = param_vals)
<environment: namespace:mlr3pipelines>
```

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

```
pss = po("scale")
print(pss$param_set)
```

<ParamSet:scale>

```
class lower upper nlevels
                                                        default value
1:
           center ParamLgl
                               NA
                                     NA
                                               2
                                                           TRUE
2:
            scale ParamLgl
                               NA
                                     NΑ
                                               2
                                                           TRUF.
           robust ParamLgl
                               NA
                                     NA
                                               2 <NoDefault[3] > FALSE
4: affect_columns ParamUty
                               NΑ
                                     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

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()
```

11. Extending

	Species	Petal.Length	${\tt Petal.Width}$	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

11.4. Adding new Tuners

In this section, we show how to implement a custom tuner for mlr3tuning. The main task of a tuner is to iteratively propose new hyperparameter configurations that we want to evaluate for a given task, learner and validation strategy. The second task is to decide which configuration should be returned as a tuning result - usually it is the configuration that led to the best observed performance value. If you want to implement your own tuner, you have to implement an R6-Object that offers an .optimize method that implements the iterative proposal and you are free to implement .assign_result to differ from the before-mentioned default process of determining the result.

Before you start with the implementation make yourself familiar with the main R6-Objects in bbotk (Black-Box Optimization Toolkit). This package does not only provide basic black box optimization algorithms and but also the objects that represent the optimization problem (OptimInstance) and the log of all evaluated configurations (Archive).

There are two ways to implement a new tuner: a) If your new tuner can be applied to any kind of optimization problem it should be implemented as a <code>Optimizer</code>. Any <code>Optimizer</code> can be easily transformed to a <code>Tuner</code>. b) If the new custom tuner is only usable for hyperparameter tuning, for example because it needs to access the task, learner or resampling objects it should be directly implemented in <code>mlr3tuning</code> as a <code>Tuner</code>.

11.4.1. Adding a new Tuner

This is a summary of steps for adding a new tuner. The fifth step is only required if the new tuner is added via bbotk.

- 1. Check the tuner does not already exist as a [Optimizer](https://bbotk.mlr-org.com/reference/Optimizer.htm or [Tuner](https://mlr3tuning.mlr-org.com/reference/Tuner.html) in the GitHub reposito-
- 2. Use one of the existing optimizers / tuners as a template.
- 3. Overwrite the .optimize private method of the optimizer / tuner.
- 4. Optionally, overwrite the default .assign_result private method.
- 5. Use the mlr3tuning::TunerFromOptimizer class to transform the Optimizer to a Tuner.

- 6. Add unit tests for the tuner and optionally for the optimizer.
- 7. Open a new pull request for the [Tuner](https://mlr3tuning.mlr-org.com/reference/Tuner.html) and optionally a second one for the [Optimizer](https://bbotk.mlr-org.com/reference/Optimizer.html).

11.4.2. Template

If the new custom tuner is implemented via bbotk, use one of the existing optimizer as a template e.g. bbotk::OptimizerRandomSearch. There are currently only two tuners that are not based on a Optimizer: mlr3hyperband::TunerHyperband and mlr3tuning::TunerIrace. Both are rather complex but you can still use the documentation and class structure as a template. The following steps are identical for optimizers and tuners.

Rewrite the meta information in the documentation and create a new class name. Scientific sources can be added in R/bibentries.R which are added under @source in the documentation. The example and dictionary sections of the documentation are auto-generated based on the @templateVar id <tuner_id>. Change the parameter set of the optimizer / tuner and document them under @section Parameters. Do not forget to change mlr_optimizers\$add() / mlr_tuners\$add() in the last line which adds the optimizer / tuner to the dictionary.

11.4.3. Optimize method

The \$.optimize() private method is the main part of the tuner. It takes an instance, proposes new points and calls the \$eval_batch() method of the instance to evaluate them. Here you can go two ways: Implement the iterative process yourself or call an external optimization function that resides in another package.

11.4.3.1. Writing a custom iteration

Usually, the proposal and evaluation is done in a repeat-loop which you have to implement. Please consider the following points:

- You can evaluate one or multiple points per iteration
- You don't have to care about termination, as \$eval_batch() won't allow more evaluations
 then allowed by the bbotk::Terminator. This implies, that code after the repeat-loop will
 not be executed.
- You don't have to care about keeping track of the evaluations as every evaluation is automatically stored in inst\$archive.
- If you want to log additional information for each evaluation of the Objective in the Archive you can simply add columns to the data.table object that is passed to \$eval_batch().

11.4.3.2. Calling an external optimization function

Optimization functions from external packages usually take an objective function as an argument. In this case, you can pass inst\$objective_function which internally calls \$eval_batch(). Check out OptimizerGenSA for an example.

11.4.4. Assign result method

The default <code>\$.assign_result()</code> private method simply obtains the best performing result from the archive. The default method can be overwritten if the new tuner determines the result of the optimization in a different way. The new function must call the <code>\$assign_result()</code> method of the instance to write the final result to the instance. See <code>mlr3tuning::TunerIrace</code> for an implementation of <code>\$.assign_result()</code>.

11.4.5. Transform optimizer to tuner

This step is only needed if you implement via bbotk. The mlr3tuning::TunerFromOptimizer class transforms a Optimizer to a Tuner. Just add the Optimizer to the optimizer field. See mlr3tuning::TunerRandomSearch for an example.

11.4.6. Add unit tests

The new custom tuner should be thoroughly tested with unit tests. Tuners can be tested with the test_tuner() helper function. If you added the Tuner via a Optimizer, you should additionally test the Optimizer with the test_optimizer() helper function.

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.
- 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, 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." https://doi.org/10.48550/ARXIV.2107.05847.
- 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.
- 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.
- Brenning, Alexander. 2012. "Spatial Cross-Validation and Bootstrap for the Assessment of Prediction Rules in Remote Sensing: The R Package Sperrorest." In 2012 IEEE International Geoscience and Remote Sensing Symposium. IEEE. https://doi.org/10.1109/igarss.2012.6352393.
- Bücker, Michael, Gero Szepannek, Alicja Gosiewska, and Przemyslaw Biecek. 2022. "Transparency, 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.
- 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.
- 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.
- 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.
- 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.
- 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.
- Lang, Michel. 2017. "checkmate: Fast Argument Checks for Defensive R Programming." The R Journal 9 (1): 437–45. https://doi.org/10.32614/RJ-2017-028.
- Lang, Michel, Martin Binder, Jakob Richter, Patrick Schratz, Florian Pfisterer, Stefan Coors, Quay Au, Giuseppe Casalicchio, Lars Kotthoff, and Bernd Bischl. 2019. "mlr3: A Modern Object-Oriented Machine Learning Framework in R." *Journal of Open Source Software*, December. https://doi.org/10.21105/joss.01903.
- Legendre, Pierre. 1993. "Spatial Autocorrelation: Trouble or New Paradigm?" Ecology 74 (6): 1659–73. https://doi.org/10.2307/1939924.
- Li, Lisha, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2017. "Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization." *The Journal of Machine Learning Research* 18 (1): 6765–6816.
- 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.
- Lovelace, Robin, Jakub Nowosad, and Jannes Muenchow. 2019. Geocomputation with R. CRC Press.
- Meyer, Hanna, Christoph Reudenbach, Tomislav Hengl, Marwan Katurji, and Thomas Nauss. 2018. "Improving Performance of Spatio-Temporal Machine Learning Models Using Forward Feature Selection and Target-Oriented Validation." *Environmental Modelling & Software* 101 (March): 1–9. https://doi.org/10.1016/j.envsoft.2017.12.001.

- Muenchow, J., A. Brenning, and M. Richter. 2012. "Geomorphic Process Rates of Landslides Along a Humidity Gradient in the Tropical Andes." *Geomorphology* 139-140: 271-84. https://doi.org/https://doi.org/10.1016/j.geomorph.2011.10.029.
- 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.
- Schratz, Patrick, Jannes Muenchow, Eugenia Iturritxa, Jakob Richter, and Alexander Brenning. 2019. "Hyperparameter Tuning and Performance Assessment of Statistical and Machine-Learning Algorithms Using Spatial Data." *Ecological Modelling* 406 (August): 109–20. https://doi.org/10.1016/j.ecolmodel.2019.06.002.
- Silverman, Bernard W. 1986. Density Estimation for Statistics and Data Analysis. Vol. 26. CRC press.
- Simon, Richard. 2007. "Resampling Strategies for Model Assessment and Selection." In Fundamentals of Data Mining in Genomics and Proteomics, edited by Werner Dubitzky, Martin Granzow, and Daniel Berrar, 173–86. Boston, MA: Springer US. https://doi.org/10.1007/978-0-387-47509-7 8.
- Sonabend, Raphael, Franz J Király, Andreas Bender, Bernd Bischl, and Michel Lang. 2021. "mlr3proba: An R Package for Machine Learning in Survival Analysis." *Bioinformatics*, February. https://doi.org/10.1093/bioinformatics/btab039.
- Tsallis, Constantino, and Daniel A Stariolo. 1996. "Generalized Simulated Annealing." *Physica A: Statistical Mechanics and Its Applications* 233 (1-2): 395–406.
- Vanschoren, Joaquin, Jan N. van Rijn, Bernd Bischl, and Luis Torgo. 2013. "OpenML: Networked Science in Machine Learning." SIGKDD Explorations 15 (2): 49–60. https://doi.org/10.1145/2641190.2641198.
- 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.

A. 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)
```

```
<DictionaryTask> with 32 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, sonar, spam, titanic,
  unemployment, usarrests, whas, wine, zoo
```

```
n 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

n 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.db, clust.dunn, clust.silhouette, clust.wss, debug, dens.logloss, oob_error, regr.bias, regr.ktau, regr.mae, regr.mape, regr.maxae, regr.medae, regr.medse, regr.mse, regr.msle, regr.pbias, regr.rae, regr.rmse, regr.rmsle, regr.rrse, regr.rse, regr.rsq, regr.sae, regr.smape, regr.srho, regr.sse, selected_features, sim.jaccard, sim.phi, surv.brier, surv.calib_alpha, surv.calib_beta, surv.chambless_auc, surv.cindex, surv.dcalib, surv.graf, surv.hung_auc, surv.intlogloss, surv.logloss, surv.mae, surv.mse, surv.nagelk_r2, surv.oquigley_r2, surv.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, tunethreshold, unbranch, vtreat, yeojohnson

n mlr_graphs

```
<DictionaryGraph> with 13 stored values
Keys: bagging, branch, crankcompositor, distrcompositor, greplicate,
  ovr, probregrcompositor, robustify, stacking, survaverager,
  survbagging, survtoregr, targettrafo
n mlr_tuners
<DictionaryTuner> with 10 stored values
Keys: cmaes, design_points, gensa, grid_search, hyperband, irace, mbo,
 nloptr, random_search, successive_halving
n mlr_terminators
<DictionaryTerminator> with 8 stored values
Keys: clock_time, combo, evals, none, perf_reached, run_time,
  stagnation, stagnation_batch
n mlr_tuning_spaces
<DictionaryTuningSpaces> with 24 stored values
Keys: classif.glmnet.default, classif.glmnet.rbv2,
  classif.kknn.default, classif.kknn.rbv2, classif.ranger.default,
  classif.ranger.rbv2, classif.rpart.default, classif.rpart.rbv2,
  classif.svm.default, classif.svm.rbv2, classif.xgboost.default,
  classif.xgboost.rbv2, regr.glmnet.default, regr.glmnet.rbv2,
  regr.kknn.default, regr.kknn.rbv2, regr.ranger.default,
 regr.ranger.rbv2, regr.rpart.default, regr.rpart.rbv2,
  regr.svm.default, regr.svm.rbv2, regr.xgboost.default,
 regr.xgboost.rbv2
nlr_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
n mlr_fselectors
<DictionaryFSelector> with 7 stored values
Keys: design_points, exhaustive_search, genetic_search, random_search,
 rfe, sequential, shadow_variable_search
```

B. Solutions to exercises

B.1. Solutions to Chapter 2

B.2. Solutions to Chapter 3

B.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",
                 = to_tune(0, 1),
     mtry.ratio
     sample.fraction = to_tune(1e-1, 1),
                 = to_tune(1, 2000)
     num.trees
   )
   instance = ti(
     task = tsk("mtcars"),
     learner = learner,
10
     resampling = rsmp("cv", folds = 3),
11
     measures = msr("regr.rmse"),
     terminator = trm("evals", n_evals = 50)
   )
14
15
   tuner = tnr("random_search", batch_size = 10)
16
17
   tuner$optimize(instance)
```

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)
  )
8 at = auto_tuner(
  method = tnr("random_search", batch_size = 10),
    learner = learner,
   resampling = rsmp("holdout"),
11
    measures = msr("regr.rmse"),
    terminator = trm("evals", n_evals = 50)
14 )
15
task = tsk("mtcars")
 outer_resampling = rsmp("cv", folds = 3)
 rr = resample(task, at, outer_resampling, store_models = TRUE)
20 rr$aggregate()
```

regr.mse 12.16805

- **B.4. Solutions to Chapter 5**
- **B.5. Solutions to Chapter 6**
- **B.6. Solutions to Chapter 8**
- **B.7. Solutions to Chapter 9**
- B.8. Solutions to Chapter 10
- **B.9. Solutions to Chapter 11**

C. 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 4.2.2 (2022-10-31) Platform: x86_64-pc-linux-gnu (64-bit) Running under: Ubuntu 22.04.1 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 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 LC_ADDRESS=C [7] LC_PAPER=C.UTF-8 LC_NAME=C [10] LC_TELEPHONE=C LC_MEASUREMENT=C.UTF-8 LC_IDENTIFICATION=C attached base packages: [1] stats graphics grDevices utils datasets methods base loaded via a namespace (and not attached): [1] digest_0.6.30 lifecycle_1.0.3 jsonlite_1.8.4 magrittr_2.0.3 [5] evaluate_0.18 cli_3.4.1 rlang_1.0.6 stringi_1.7.8 [9] vctrs_0.5.1 rmarkdown_2.18 tools_4.2.2 stringr_1.5.0 compiler_4.2.2 [13] glue_1.6.2 $xfun_0.35$ fastmap_1.1.0 [17] htmltools_0.5.3 knitr_1.41

Index

```
Control Parameters, 79
Hyperband, 79
Hyperparameter Optimization, 71
Hyperparameters, 71
Inner Resampling, 94
MBO, 79
mlr3tuning, 73
Multi-objective Optimization, 89
Nested Resampling, 93
Optimism Of The Training Error, 93
Outer Resampling, 94
Parameters, 71
Pareto Front, 89
Pareto Optimality, 89
Pareto-dominate, 89
Search Space, 74
Support Vector Machine, 73
Terminators, 75
Tuners, 77
Tuning, 71
Tuning Budget, 75
Tuning Instance, 76
Tuning Space, 74
```