mlr3book

Marc Becker, Martin Binder, Bernd Bischl, Natalie Foss, Lars Kotthoff, Michel Lang,

Table of contents

Quickstart

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

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

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

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

```
n= 120

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

1) root 120 70 setosa (0.41666667 0.41666667 0.16666667)
  2) Petal.Length< 2.45 50  0 setosa (1.00000000 0.00000000 0.00000000) *
  3) Petal.Length>=2.45 70 20 versicolor (0.00000000 0.71428571 0.28571429)
    6) Petal.Length< 4.95 49  1 versicolor (0.00000000 0.97959184 0.02040816) *
    7) Petal.Length>=4.95 21  2 virginica (0.00000000 0.09523810 0.90476190) *
```

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

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

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

```
classif.acc
  0.8333333
```

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

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

Introduction and Overview

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

Target Audience

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

mlr3 provides a domain-specific language for machine learning in R. We target both **practitioners** who want to quickly apply machine learning algorithms and **researchers** who want to implement, benchmark, and compare their new methods in a structured environment. The package is a complete rewrite of an earlier version of mlr that leverages many years of experience to provide a state-of-the-art system that is easy to use and extend.

Why a Rewrite?

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

Design Principles

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

- Backend over frontend. Most packages of the mlr3 ecosystem focus on processing and transforming data, applying machine learning algorithms, and computing results. We do not provide graphical user interfaces (GUIs); visualizations of data and results are provided in extra packages like mlr3viz.
- Embrace R6 for a clean, object-oriented design, object state-changes, and reference semantics.

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

Package Ecosystem

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

- R6: Reference class objects.
- data.table: Extension of R's `data.frame`.
- digest: Hash digests.
- uuid: Unique string identifiers.
- lgr: Logging facility.
- mlbench: A collection of machine learning data sets.

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

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

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

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

A complete list with links to the repositories for the respective packages can be found on our package overview.

Part I

Basics

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

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

The process of splitting up data into training and test sets, building a model, and evaluating it can be repeated several times, resampling different training and test sets from the original data each time. Multiple resampling iterations allow us to get a better and less biased generalizable performance estimate for a particular type of model. As data are usually partitioned randomly into training and test sets, a single split can for example produce training and test sets that are very different, hence creating to the misleading impression that the particular type of model does not perform well.

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

This chapter covers the following topics:

Tasks

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

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

Learners

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

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

How to extend learners and implement your own is covered in a supplemental advanced technical section.

Train and predict

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

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

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

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

1 Quick R6 Intro for Beginners

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

- Objects are created by calling the constructor of an `"R6::R6Class()"` object, specifically the initialization method `$new()`. For example, `foo = Foo$new(bar = 1)` creates a new object of class `Foo`, setting the `bar` argument of the constructor to the value `1`. Most objects in mlr3 are created through special functions (e.g. `lrn("regr.rpart")`) that encapsulate this and are also referred to as *sugar functions*.
- Objects have mutable state that is encapsulated in their fields, which can be accessed through the dollar operator. We can access the `bar` value in the `foo` variable from above through `foo$bar` and set its value by assigning the field, e.g. `foo$bar = 2`.
- In addition to fields, objects expose methods that allow to inspect the object's state, retrieve information, or perform an action that changes the internal state of the object. For example, the `$train` method of a learner changes the internal state of the learner by building and storing a trained model, which can then be used to make predictions, given data.
- Objects can have public and private fields and methods. The public fields and methods define the API to interact with the object. Private methods are only relevant for you if you want to extend mlr3, e.g. with new learners.
- R6 objects are internally environments, and as such have reference semantics. For example, `foo2 = foo` does not create a copy of `foo` in `foo2`, but another reference to the same actual object. Setting `foo$bar = 3` will also change `foo2$bar` to `3` and vice versa.
- To copy an object, use the `$clone()` method and the `deep = TRUE` argument for nested objects, for example, `foo2 = foo$clone(deep = TRUE)`.

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

1.1 Tasks

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

1.1.1 Task Types

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

- **Classification Task**: The target is a label (stored as `character()orfactor()`) with only relatively few distinct values → `"TaskClassif")`.
- **Regression Task**: The target is a numeric quantity (stored as `integer()` or `double()`) → `"TaskRegr")`.
- **Survival Task**: The target is the (right-censored) time to an event. More censoring types are currently in development → `"mlr3proba::TaskSurv")` in add-on package mlr3proba.
- **Density Task**: An unsupervised task to estimate the density → `"mlr3proba::TaskDens")` in add-on package mlr3proba.
- **Cluster Task**: An unsupervised task type; there is no target and the aim is to identify similar groups within the feature space → `"mlr3cluster::TaskClust")` in add-on package mlr3cluster.
- **Spatial Task**: Observations in the task have spatio-temporal information (e.g. coordinates) → `"mlr3spatiotempcv::TaskRegrST")` or `"mlr3spatiotempcv::TaskClassifST")` in add-on package mlr3spatiotempcv.
- **Ordinal Regression Task**: The target is ordinal → `TaskOrdinal` in add-on package mlr3ordinal (still in development).

1.1.2 Task Creation

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

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

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

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

Next, we create a regression task, i.e. we construct a new instance of the R6 class `"TaskRegr")`. Formally, the intended way to initialize an R6 object is to call the constructor `TaskRegr$new()`. Instead, we are calling the converter `"as_task_regr()"` to convert our `data.frame()` stored in the variable `data` to a task and provide the following information:

1. `x`: Object to convert. Works for rectangular data formats such as `data.frame()`, `data.table()`, or `tibble()`. Internally, the data is converted and stored in an abstract `"DataBackend")`. This allows connecting to out-of-memory storage systems like SQL servers via the extension package mlr3db.

2. `target`: The name of the prediction target column for the regression problem, here miles per gallon (`"mpg"`).
3. `id` (optional): An arbitrary identifier for the task, used in plots and summaries. If not provided, the deparsed name of `x` will be used.

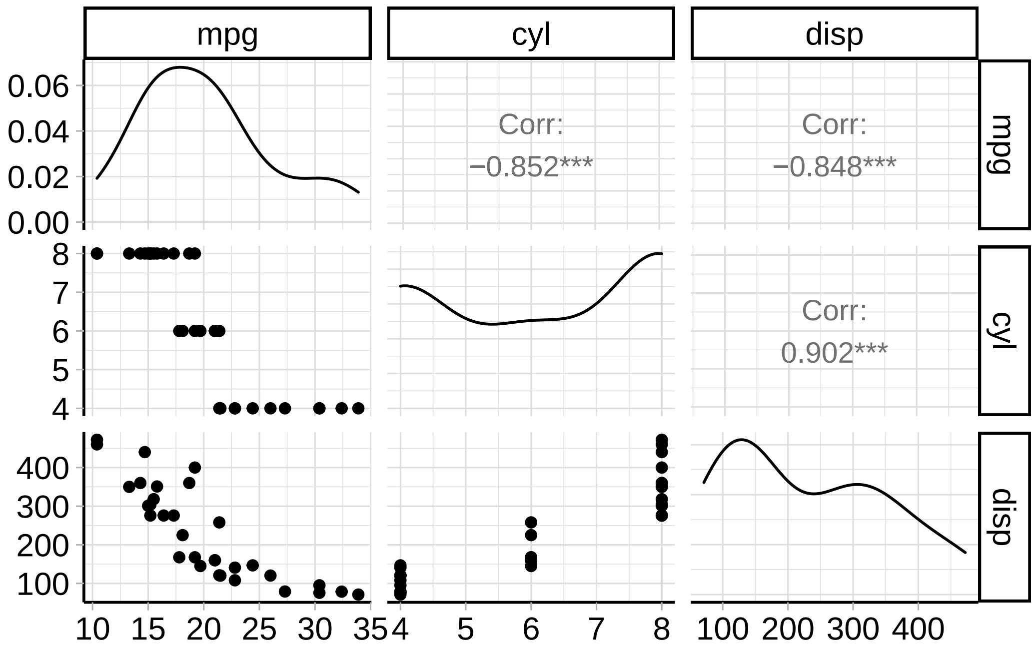
```
library("mlr3")

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

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

The `print()` method gives a short summary of the task: It has 32 observations and 3 columns, of which 2 are features stored in double-precision floating point format.

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

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



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

1.1.3 Predefined tasks

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

```
mlr_tasks
```

```
<DictionaryTask> with 11 stored values
Keys: boston_housing, breast_cancer, german_credit, iris, mtcars,
  penguins, pima, sonar, spam, wine, zoo
```

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

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

```
               key                    label task_type nrow ncol properties lgl
 1: boston_housing   Boston Housing Prices      regr  506   19              0
 2:  breast_cancer Wisconsin Breast Cancer   classif  683   10   twoclass   0
 3:  german_credit           German Credit   classif 1000   21   twoclass   0
 4:           iris            Iris Flowers   classif  150    5 multiclass   0
 5:         mtcars            Motor Trends      regr   32   11              0
 6:       penguins         Palmer Penguins   classif  344    8 multiclass   0
 7:           pima    Pima Indian Diabetes   classif  768    9   twoclass   0
 8:          sonar  Sonar: Mines vs. Rocks   classif  208   61   twoclass   0
 9:           spam       HP Spam Detection   classif 4601   58   twoclass   0
10:           wine            Wine Regions   classif  178   14 multiclass   0
11:            zoo             Zoo Animals   classif  101   17 multiclass  15
    int dbl chr fct ord pxc
 1:   3  13   0   2   0   0
 2:   0   0   0   0   9   0
 3:   3   0   0  14   3   0
 4:   0   4   0   0   0   0
 5:   0  10   0   0   0   0
 6:   3   2   0   2   0   0
 7:   0   8   0   0   0   0
 8:   0  60   0   0   0   0
 9:   0  57   0   0   0   0
10:   2  11   0   0   0   0
11:   1   0   0   0   0   0
```

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

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

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

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

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

```
                key                                 label task_type  nrow
 1:    bike_sharing                   Bike Sharing Demand      regr 17379
 2:  boston_housing                Boston Housing Prices      regr   506
 3:   breast_cancer              Wisconsin Breast Cancer   classif   683
 4:   german_credit                         German Credit   classif  1000
 5:            ilpd              Indian Liver Patient Data   classif   583
 6:            iris                          Iris Flowers   classif   150
 7:      kc_housing              King County House Sales      regr 21613
 8:       moneyball       Major League Baseball Statistics      regr  1232
 9:          mtcars                          Motor Trends      regr    32
10:       optdigits Optical Recognition of Handwritten Digits   classif  5620
11:        penguins                       Palmer Penguins   classif   344
12: penguins_simple            Simplified Palmer Penguins   classif   333
13:            pima                  Pima Indian Diabetes   classif   768
14:           sonar                  Sonar: Mines vs. Rocks   classif   208
15:            spam                     HP Spam Detection   classif  4601
16:         titanic                               Titanic   classif  1309
17:       usarrests                            US Arrests     clust    50
18:            wine                          Wine Regions   classif   178
19:             zoo                           Zoo Animals   classif   101
```

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

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

As an alternative, all mlr3 objects come with a `help()` method which opens the corresponding help page. To open the help page of the previously-constructed palmer penguins task, you can also call:

```r
task_penguins$help()
```

1.1.4 Task API

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

1.1.4.1 Retrieving Data

The data stored in a task can be retrieved directly from fields, for example for `task_mtcars` that we defined above we can get the number of rows and columns:

```r
task_mtcars
```

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

```r
task_mtcars$nrow
```

```
[1] 32
```

```r
task_mtcars$ncol
```

```
[1] 3
```

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

```r
task_mtcars$data()
```

```
     mpg cyl  disp
 1: 21.0   6 160.0
 2: 21.0   6 160.0
 3: 22.8   4 108.0
 4: 21.4   6 258.0
 5: 18.7   8 360.0
 6: 18.1   6 225.0
 7: 14.3   8 360.0
 8: 24.4   4 146.7
 9: 22.8   4 140.8
10: 19.2   6 167.6
11: 17.8   6 167.6
12: 16.4   8 275.8
13: 17.3   8 275.8
14: 15.2   8 275.8
15: 10.4   8 472.0
16: 10.4   8 460.0
17: 14.7   8 440.0
18: 32.4   4  78.7
19: 30.4   4  75.7
20: 33.9   4  71.1
21: 21.5   4 120.1
22: 15.5   8 318.0
23: 15.2   8 304.0
24: 13.3   8 350.0
25: 19.2   8 400.0
26: 27.3   4  79.0
27: 26.0   4 120.3
28: 30.4   4  95.1
29: 15.8   8 351.0
30: 19.7   6 145.0
31: 15.0   8 301.0
32: 21.4   4 121.0
     mpg cyl  disp
```

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

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

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

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

```
    mpg cyl  disp
1: 21.0   6 160.0
```

```
2: 18.7   8 360.0
3: 19.2   6 167.6
```

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

Similarly to row IDs, target and feature columns also have unique identifiers, i.e. names (stored as `character()`). Their names can be accessed via the public slots `$feature_names` and `$target_names`. Here, "target" refers to the variable we want to predict and "feature" to the predictors for the task. The target will usually be only a single name.

```r
task_mtcars$feature_names
```

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

```r
task_mtcars$target_names
```

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

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

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

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

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

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

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

1.1.4.2 Binary classification

Classification problems with a target variable with only two classes are called binary classification tasks. They are special in the sense that one of these classes is denoted *positive* and the other one *negative*. You can specify the *positive class* within the `"TaskClassif", text = "classification task")` object during task creation. If not explicitly set during construction, the positive class defaults to the first level of the target variable.

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

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

1.1.4.3 Roles (Rows and Columns)

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

For example, the previously-constructed `task_mtcars` task has the following column roles:

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

```
$feature
[1] "cyl"  "disp"

$target
[1] "mpg"

$name
character(0)

$order
character(0)

$stratum
character(0)

$group
character(0)

$weight
character(0)
```

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

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

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

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

The row names are now a feature whose values are stored in the column `"rn"`. We include this column here for educational purposes only. Generally speaking, there is no point in having a feature that uniquely identifies each row. Furthermore, the character data type will cause problems with many types of machine learning algorithms.

The identifier may be useful to label points in plots, for example to identify outliers. To achieve this, we will change the role of the `rn` column by removing it from the list of features and assign the new role `"name"`. There are two ways to do this:

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

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

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

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

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

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

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

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

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

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

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

Rows can have two different roles:

1. Role `use`: Rows that are generally available for model fitting (although they may also be used as test set in resampling). This role is the default role.
2. Role `validation`: Rows that are not used for training. Rows that have missing values in the target column during task creation are automatically set to the validation role.

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

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

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

1.1.4.4 Task Mutators

As shown above, modifying `$col_roles` or `$row_roles` (either via `set_col_roles()`/`set_row_roles()` or directly by modifying the named list) changes the view on the data. The additional convenience method `$filter()` subsets the current view based on row IDs and `$select()` subsets the view based on feature names.

```
task_penguins = tsk("penguins")
task_penguins$select(c("body_mass", "flipper_length")) # keep only these features
task_penguins$filter(1:3) # keep only these rows
task_penguins$head()
```

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

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

```
task_penguins$cbind(data.frame(letters = letters[1:3])) # add column letters
task_penguins$head()
```

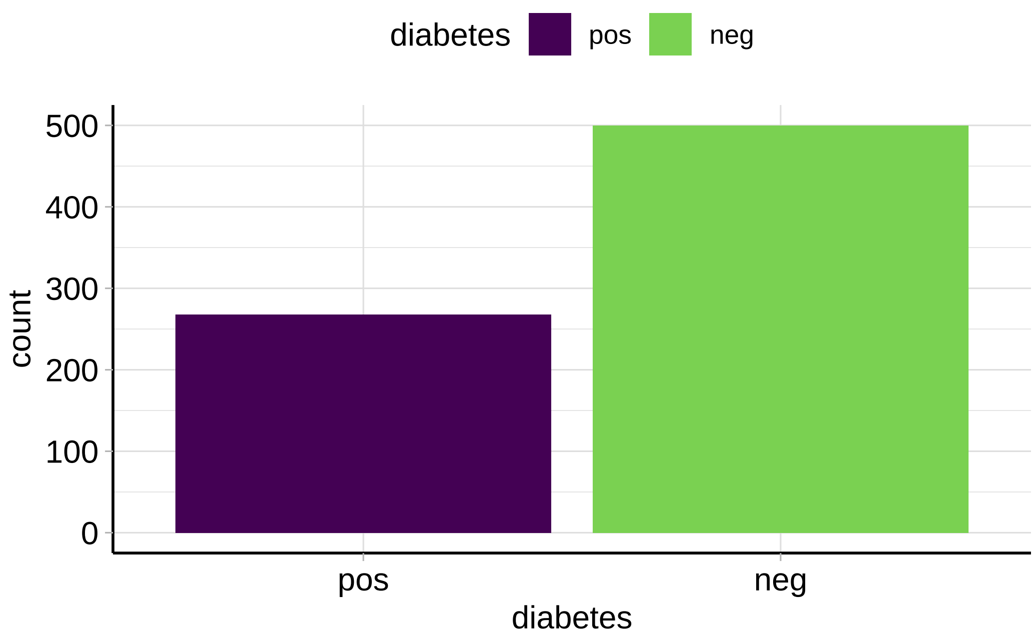
```
   species body_mass flipper_length letters
1:  Adelie      3750            181       a
2:  Adelie      3800            186       b
3:  Adelie      3250            195       c
```

1.1.5 Plotting Tasks

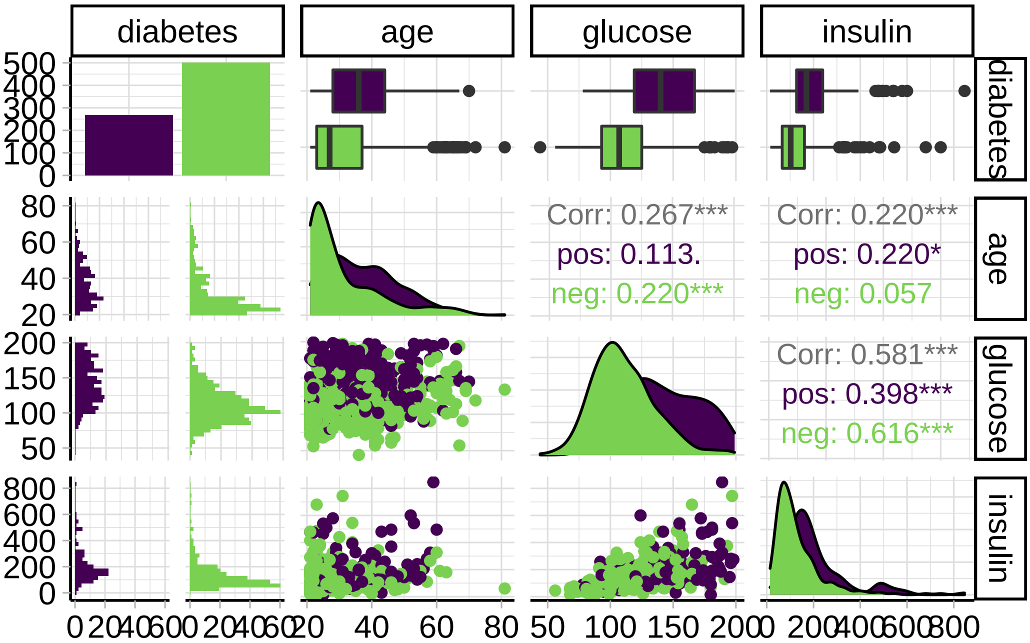
The mlr3viz package provides plotting facilities for many classes implemented in mlr3. The available plot types depend on the class, but all plots are returned as ggplot2 objects which can be easily customized.

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

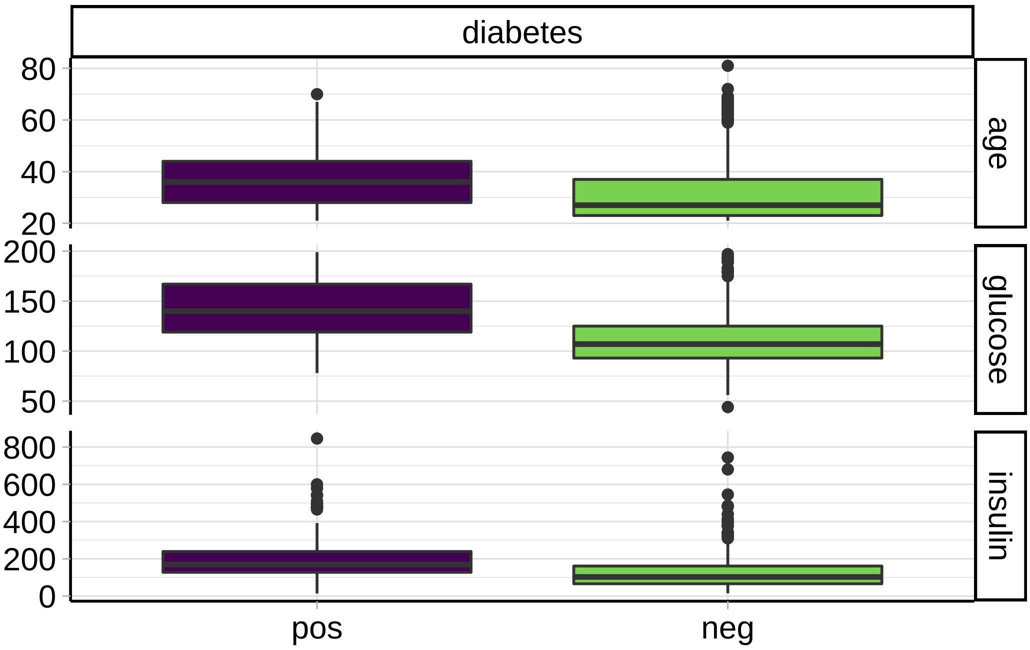
```
library("mlr3viz")

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

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

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




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




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



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

```
library("mlr3viz")

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

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

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


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