mlr3: Towards an object-oriented, configurable ML pipeline system in R6

Bernd Bischl
Computational Statistics, LMU

Section 1

MLR2

Current state of mlr

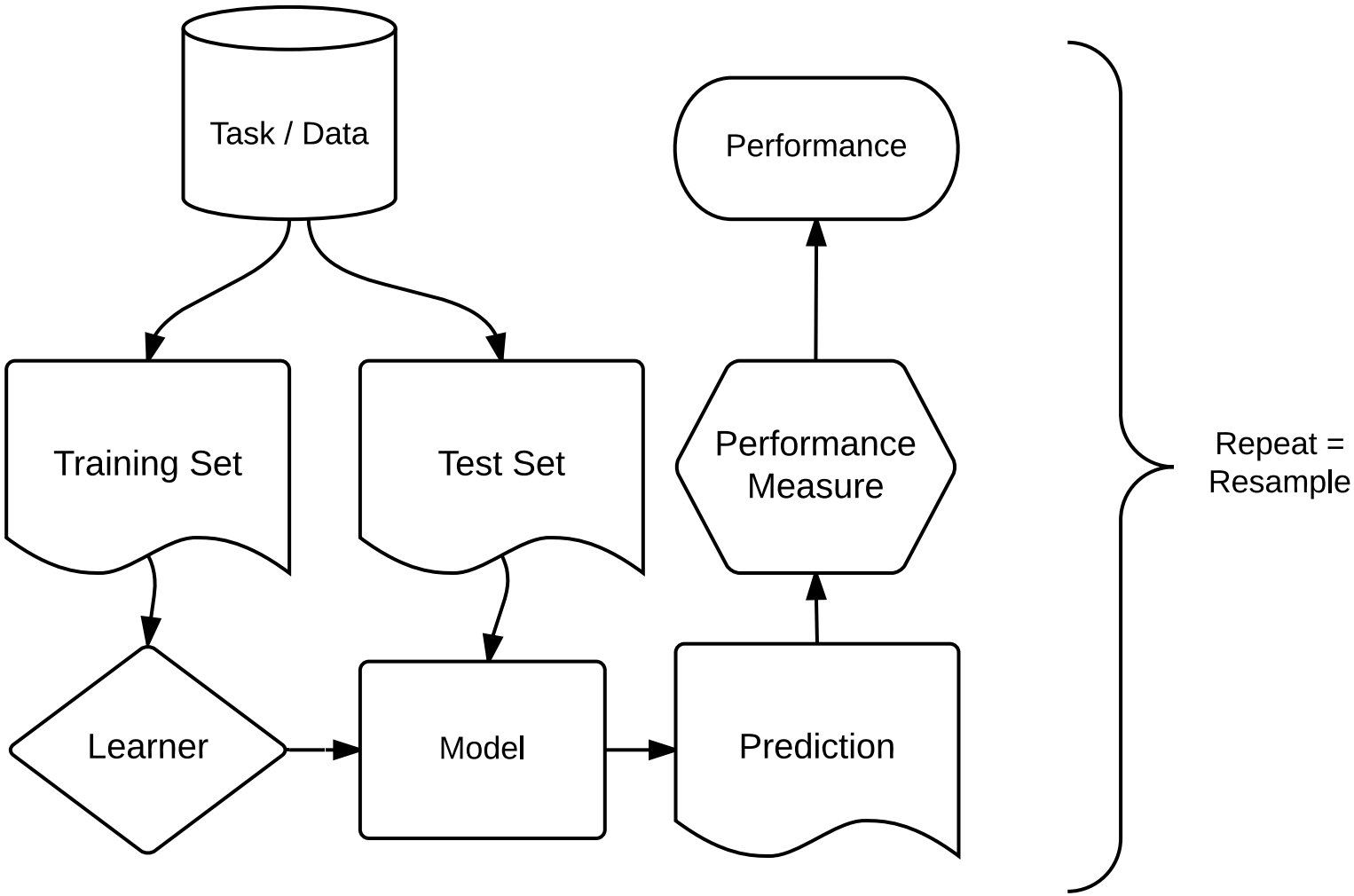
- Popular DSL for ML experiments
- Connected to hundreds ML algorithms in R
- Project home page:

`https://github.com/mlr-org/mlr`

  - Cheatsheet for an quick overview.
  - Tutorial for mlr documentation with many code examples.
  - Ask questions in the GitHub issue tracker.

- 8-10 main developers, quite a few contributors, 4 GSOC projects in 2015/16 and one in 2017.
- About 30K lines of code, 8K lines of unit tests.

Building Blocks



- `mlr` objects: tasks, learners, measures, resampling instances.

Motivation: mlr

- Clean and extensible via S3.
- Reflections: nearly all objects are queryable (i.e. you can ask them for their properties and program on them).
- The OO-structure allows many generic algorithms.

Main features:

- Support for classification, regression, survival and clustering
- Multilabel classification
- Resampling, Spatial and temporal resampling, Nested resampling
- Tuning and feature selection
- Wrapping / Pipelining
- Handling of functional data
- Visualization
- ...

The mlr ecosystem

Next to `mlr` a larger number of surrounding packages exist

- `ParamHelpers` - Description language for parameters in machine learning and optimization.
- `mlrMBO` - Toolbox for Bayesian optimization. Useful for tuning hyperparameters in machine learning.
- `mlrCPO` - Operator Based Machine Learning Pipeline Construction. Allows creation of preprocessing pipelines as DAGs.
- `mlr-extralearners` - Contains additional (possibly unstable) learning algorithms that are not part of the `mlr` package itself.
- `parallelMap` - Unified parallelization framework for multiple back-end. Used for parallelization in `mlr`.
- `batchtools` - Large-scale R experiments on batch systems / clusters.
- `iml` - Interpretable Machine Learning.

MLR IN ACTION I

Let's build a machine learning pipeline that does

- Impute missings (mean for nums, mode for factors)
- does different types of factor encodings
- Does different types of feature filtering

```
library(mlr)
library(mlrCPO)
library(mlrMBO)
library(parallelMap)

lrn = makeLearner("classif.xgboost")

cl = list(numeric = imputeMean(), factor = imputeMode())
pipeline = cpoImputeAll(classes = cl)
pipeline = pipeline %>>% cpoMultiplex(id = "factenc",
  cpos = list(cpoDummyEncode(), cpoImpactEncodeClassif()))
pipeline = pipeline %>>% cpoFilterFeatures()
pipeline = pipeline %>>% lrn
```

MLR IN ACTION II

```
ps = makeParamSet(
  makeDiscreteParam("factenc.selected.cpo",
    values = c("dummyencode", "impact.encode.classif")),
  makeDiscreteParam("filterFeatures.method",
    values = c("anova.test", "auc")),
  makeNumericParam("filterFeatures.perc",
    lower = 0.1, upper = 1),
  makeNumericParam("alpha", lower = -10, upper = 10,
    trafo = function(x) 2^x),
  makeIntegerParam("nrounds", lower = 1, upper = 100)
)

# we want Bayesian optimization for efficient configuration
ctrl = makeTuneControlMBO(budget = 2)
```

MLR IN ACTION III

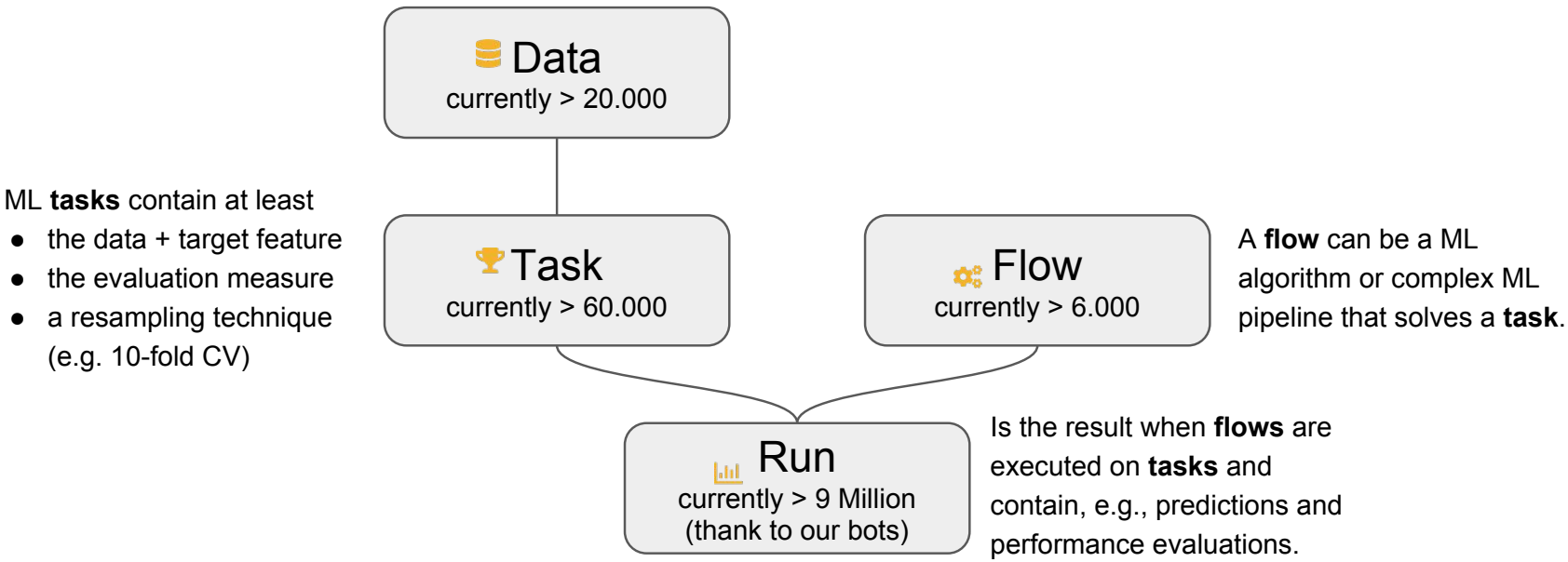
```
# attach autotuning to pipeline-xgboost
autoxgb = makeTuneWrapper(pipeline, cv3, par.set = ps,
  control = ctrl)

# Nested crossvalidation in parallel
parallelStartSocket()
r = resample(autoxgb, task, cv3)
parallelStop()
```

Section 2

OPENML

OpenML Project

OpenML.org is an online platform for sharing and organizing data, ML tasks, algorithms and experiments.
OpenML is based on 4 **basic elements**, i.e., Data, Task, Flow, and Run:

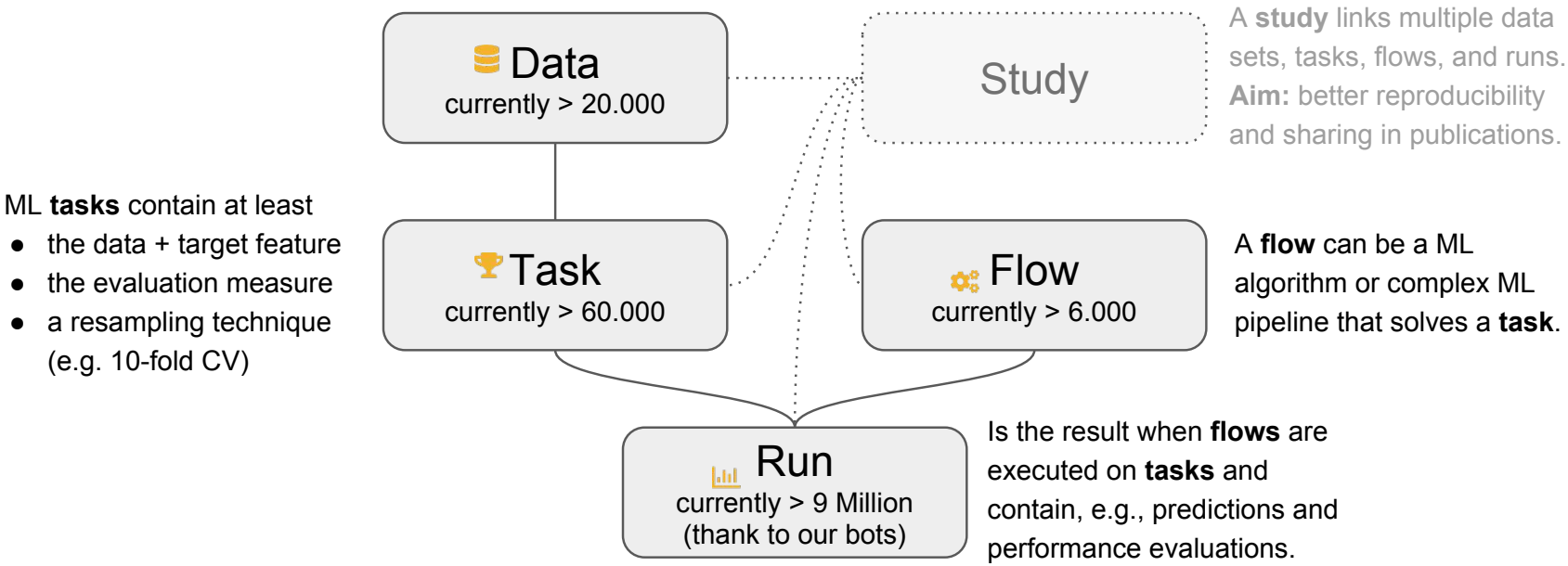


We have a REST API and client interfaces for `R` (with `mlr`), `Python` (with `sklearn`), `Java` (with `Weka` or `MOA`), and `.NET` (based on `C#`)...

Tutorials: `https://docs.openml.org/APIs`

OpenML Project

OpenML.org is an online platform for sharing and organizing data, ML tasks, algorithms and experiments.
OpenML is based on 4 **basic elements**, i.e., Data, Task, Flow, and Run:



We have a REST API and client interfaces for `R` (with `mlr`), `Python` (with `sklearn`), `Java` (with `Weka` or `MOA`), and `.NET` (based on `C#`)...

Tutorials: `https://docs.openml.org/APIs`

OpenML R Package

The `OpenML` package is nicely connected to `mlr` and contains functions to communicate with the OpenML-Server directly from your R session:

- `listOML*` functions: Explore and query existing **basic elements**.
- `getOML*` functions: Download available **basic elements**.
- `uploadOML*` functions: Upload your own **basic elements**.
- ... see Cheatsheet for an quick overview.

Example: Get available UCI tasks, run CART on them and upload results.

```
library(OpenML)
setOMLConfig(apikey = "MY_API_KEY")           # is required for uploading
lrn = makeLearner("classif.rpart")            # create a mlr CART learner
tasks = listOMLTasks(data.tag = "uci",         # list all UCI tasks with:
  evaluation.measures = "predictive_accuracy",    # accuracy as measure,
  estimation.procedure = "10-fold Crossvalidation")    # and 10-fold CV!
for (tid in tasks$task.id) {                       # iterate over task ids
  task = getOMLTask(tid)                            # download task by id
  run = runTaskMlr(task, learner = lrn)              # train CART on task
  upload = uploadOMLRun(run, tags = "MY_TAG")          # upload + tag run
}
res = listOMLRunEvaluations(tag = "MY_TAG")         # list results by tag
```

Section 3

MLR3

Lessons learned – Package Ecosystem

- CRAN has a time limit for R CMD check so that we had to disable most tests here
- This effectivly results in no reverse package checks if one of the suggested packages is uploaded to CRAN. As a result, `mlr` is permanently broken
- Continouus integration is very hacky, we now have multiple stages to pre-install and cache packages

Dependencies matter

Lessons learned – Data structures

- For storage, we often used lists of lists in `mlr`, e.g. retrieve a model from a benchmark:

```
model = result[["task-id"]][["learner-id"]]$
 models[[resampling_iteration]]$learner.model
```

- This is the most efficient storage (memory-wise), but working with it is tideous and error-prone.
- To overcome these issues, we wrote S3 getters:

```
fn = system.file("NAMESPACE", package = "mlr")
length(readLines(fn)) # `base` currently has ~1222
```

```
## [1] 1205
```

- Codebase is very hard to maintain

We need object orientation

MLR3

- Rewrite of `mlr`
- Be light on dependencies, but do not re-invent the wheel
- Embrace `R6` for OO and reference semantics
- Embrace `data.table` for interal storage / data transformation

https://github.com/mlr-org/mlr3

Current state of mlr3

Base functionality implemented:

- Objects `Task`, `Learner`, `Resampling`, `Measure`, `Experiment`
- `DataBackend` for tasks to work transparently with different data storage engines (Sparse, SQL, ...)
- `resample()`, `benchmark()`
- Parallelization via package `future`

Combining R6 and data.table

- Instead of lists of lists, we store most information in 2d data.tables using list columns.
- Benchmark Example: Each row describes one experiment and holds all required information:
- Columns which store R6 objects (Task, Learner, Resampling) just store a pointer (32/64 bit)
- Extracting information, subsetting objects, growing results or converting between objects is now embarassingly easy

Task	Learner	Resampling	Iter	Model
<TaskClassif>	<LearnerClassifRpart>	<ResamplingCV>	1	<rpart>
<TaskClassif>	<LearnerClassifRpart>	<ResamplingCV>	2	<rpart>

Stepwise modeling

```
task = mlr_tasks$get("iris")
learner = mlr_learners$get("classif.rpart")
train = 1:120; test = 121:150

e = Experiment$new(task, learner)
e$train(train)$predict(test)$score()
print(e)

## Experiment [scored (complete)]:
##  + Task: iris
##  + Learner: classif.rpart
##  + Model: [rpart]
##  + Predictions: [PredictionClassif]
##  + Performance: mmce=0.166667
##
## Public: clone, data, has_errors, hash, learner, logs, model,
##   performance, predict, prediction, score, state, task, test_set,
##   timings, train, train_set, validation_set
```

Experiment Object

```
e$performance

##   mmce
## 0.1667

class(e$model)

## [1] "rpart"

head(e$test_set, 3)

## [1] 121 122 123

head(as.data.table(e$prediction), 3)

##    row_id   response     truth
## 1:    121  virginica virginica
## 2:    122 versicolor virginica
## 3:    123  virginica virginica
```

Resampling

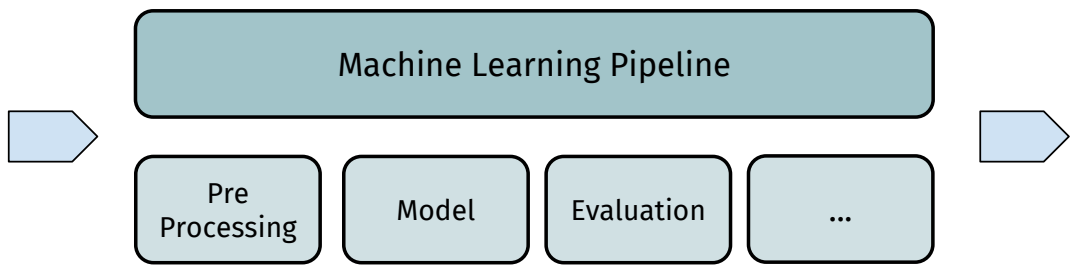
```
resampling = mlr_resamplings$get("cv")
rr = resample(task, learner, resampling)
head(as.data.table(rr), 2)

##                hash          task task_id                 learner
## 1: 889252311a01ca8b <TaskClassif>    iris <LearnerClassifRpart>
## 2: 889252311a01ca8b <TaskClassif>    iris <LearnerClassifRpart>
##       learner_id     resampling resampling_id iteration    mmce
## 1: classif.rpart <ResamplingCV>            cv         1 0.06667
## 2: classif.rpart <ResamplingCV>            cv         2 0.13333

e = rr$experiment(1)
```

Section 4

MLR PIPELINES

mlr Pipelines

- Many Machine Learning Workflows consist of multiple steps, such as preprocessing, computing features, or imputing missing data
- This is often a long winded and complicated process, and properly separating train and test data is very difficult

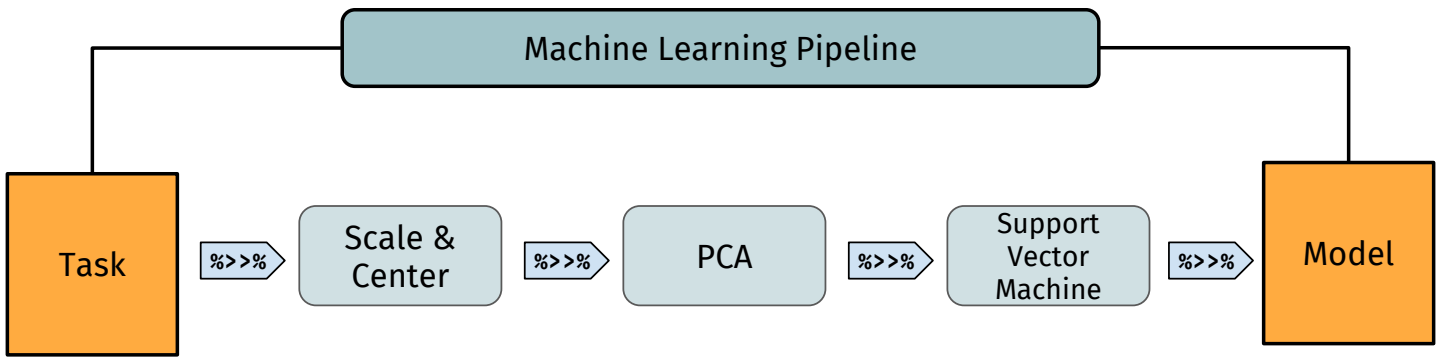


- Pipelines allow us to specify many difficult steps that are often undertaken in a few, concise lines
- By integrating pipelines with mlr3 and mlr3 tuning we can jointly tune over all hyperparameters the pipeline exposes

mlr Pipelines

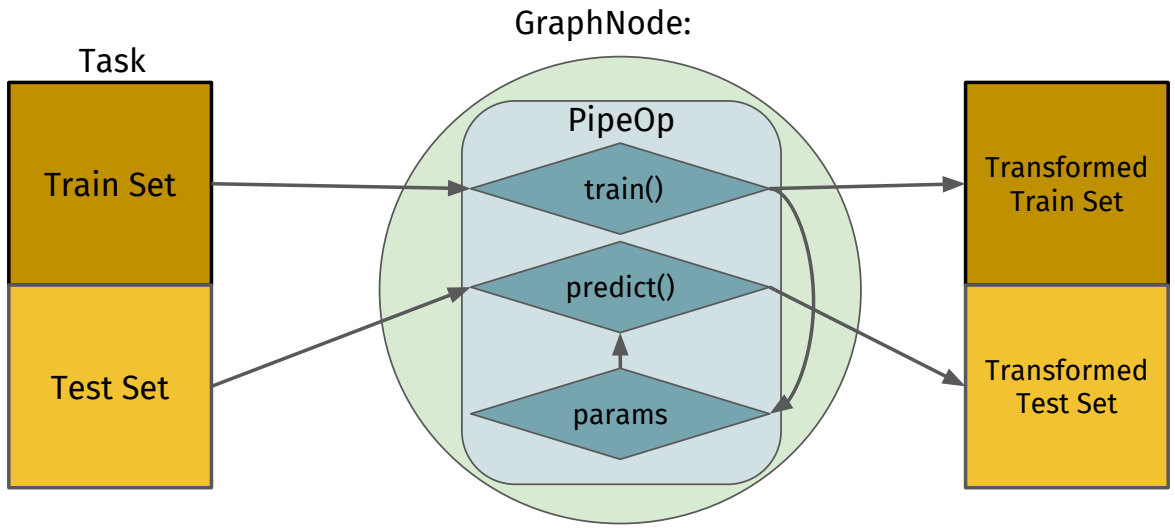
Pipelines provide:

- Multiple widely used operations (Scaling, PCA, Variable Selection, Imputation, Stacking and many others)
- A clean, extendible interface for custom pipeline operators
- A simple operator connection operator: %»%
- An abstraction for parallelization




```
# Pseudo Code:
> pipeOpScale() %>>% pipeOpPCA() %>>% pipeOpLearner("svm")
```

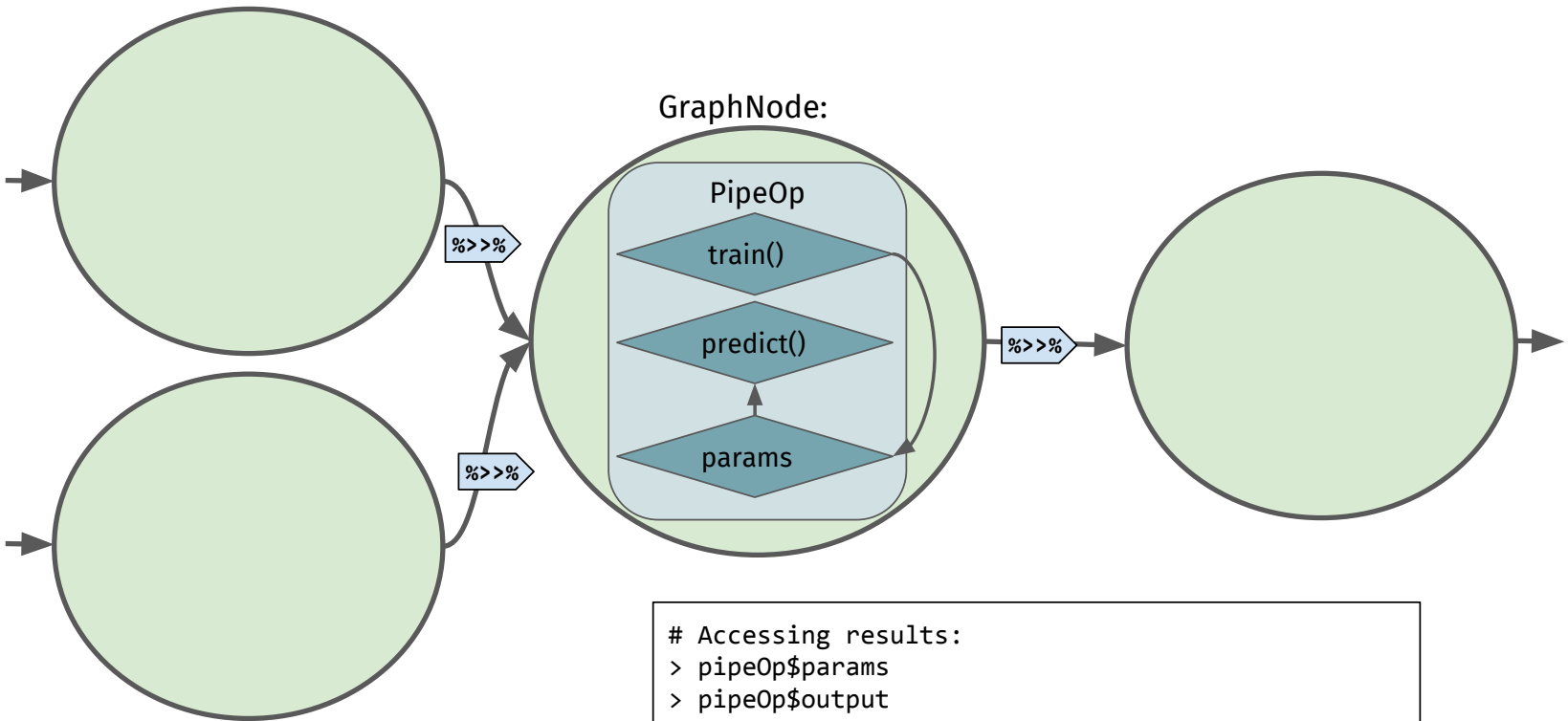
mlr Pipelines



- `train()` saves transformation params and outputs transformed training data
- `predict()` uses params and outputs transformed test data

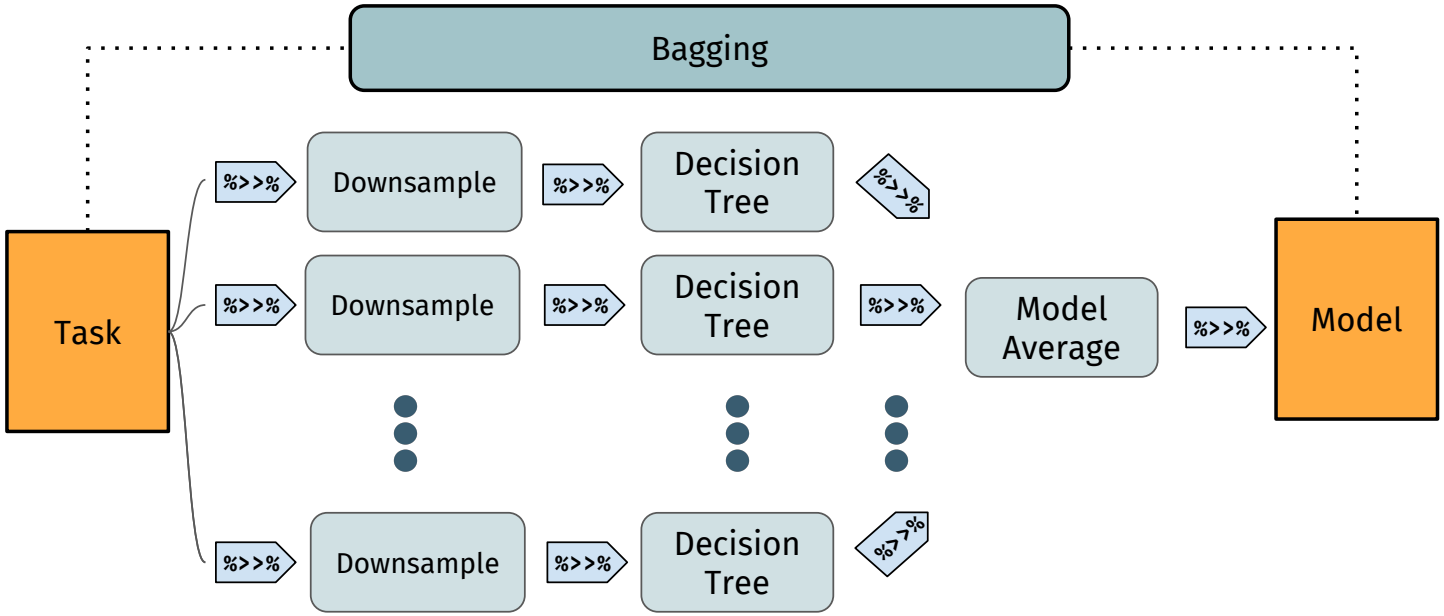
mlr Pipelines

Mulitple `GraphNodes`'s can be connected with %»%




```
# Accessing results:
> pipeOp$params
> pipeOp$output
```

mlr Pipelines




```
# Pseudo Code:
> rep(100, pipeOpDownsample() %>>% pipeOpLearner("rpart")) %>>% pipeOpModelAverage()
```

mlr Pipelines

- Such a graph is now an mlr3 learners and can be trained, prediced, resampled and tuned as any other learner
- The graph can easily be trained by walking in a similar manner like topological sorting, mainting and active front
- Individual trained parameters and objects can easily be accessed by indexing by PipeOp names

Thanks! Questions? Comments? Comment on Github?