Advanced mlr Session 1

Bernd Bischl

2018-07-03 – 2018-07-03

MLR INTRO AND RECAP

MOTIVATION: MACHINE LEARNING IN R

The **good** news:

- CRAN serves hundreds of packages for machine learning
- Often compliant to the unwritten interface definition:

```r
model = fit(target ~ ., data = train.data, ...)
predictions = predict(model, newdata = test.data, ...)
```

The **bad** news:

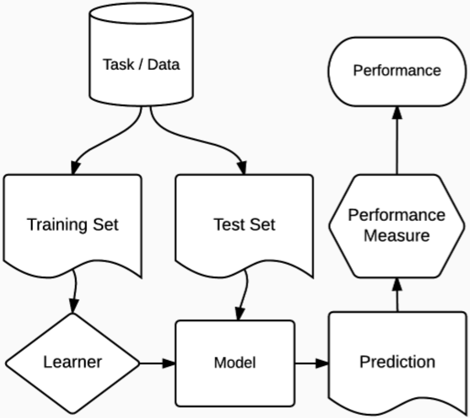
- Some packages' API is "just different"
- Functionality is always package or model-dependent, even though the procedure might be general
- No meta-information available or buried in docs

Our goal: A domain-specific language for ML concepts!

MOTIVATION: MLR

- 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 coming in 2017
- About 30K lines of code, 8K lines of unit tests

- Unified interface for the basic building blocks: tasks, learners, hyperparameters, ...



BASIC FEATURES OF MLR

- Tasks and Learners
- Train, Test, Resample
- Performance
- Benchmarking
- Hyperparameter Tuning
- Nested Resampling
- Parallelization

TRAIN / TEST / EVAL

```
task = sonar.task
n = getTaskSize(task)
lrn = makeLearner("classif.rpart", predict.type = "prob")
mod = train(lrn, task, subset = seq(1, n, 2))
pred = predict(mod, task = task, subset = seq(2, n, 2))
performance(pred, measures = list(mmce, mlr::auc))

##  mmce   auc
## 0.298 0.721
```

RESAMPLE

```r
rdesc = makeResampleDesc("CV", iters = 3L, stratify = TRUE)
r = resample(lrn, task, rdesc)
print(r$aggr)

## mmce.test.mean
##          0.279

print(r$measures.test)

##   iter  mmce
## 1    1 0.304
## 2    2 0.286
## 3    3 0.246

print(head(as.data.frame(r$pred), 3L))

##    id truth prob.M prob.R response iter  set
## 1  98     M  0.098  0.902        R    1 test
## 2 104     M  0.774  0.226        M    1 test
## 3 106     M  0.100  0.900        R    1 test
```

BENCHMARK

```r
lrns = list(
  makeLearner("classif.rpart"),
  makeLearner("classif.randomForest")
)
b = benchmark(lrns, task, cv2, measures = mmce)
print(b)

##          task.id           learner.id mmce.test.mean
## 1 Sonar-example        classif.rpart          0.284
## 2 Sonar-example classif.randomForest          0.163
```

BENCHMARK

```
print(getBMRAggrPerformances(b, as.df = TRUE))

##          task.id          learner.id mmce.test.mean
## 1 Sonar-example        classif.rpart          0.284
## 2 Sonar-example classif.randomForest          0.163

print(getBMRPerformances(b, as.df = TRUE))

##          task.id          learner.id iter  mmce
## 1 Sonar-example        classif.rpart    1 0.337
## 2 Sonar-example        classif.rpart    2 0.231
## 3 Sonar-example classif.randomForest    1 0.192
## 4 Sonar-example classif.randomForest    2 0.135

print(head(getBMRPredictions(b, as.df = TRUE), 3L))

##          task.id    learner.id id truth response iter  set
## 1 Sonar-example classif.rpart  1     R        M    1 test
## 2 Sonar-example classif.rpart  2     R        M    1 test
## 3 Sonar-example classif.rpart  3     R        M    1 test
```

TUNING

- Optimize parameters or decisions for ML algorithm w.r.t. the estimated prediction error
- Tuner proposes configuration, eval by resampling, tuner receives performance, iterate

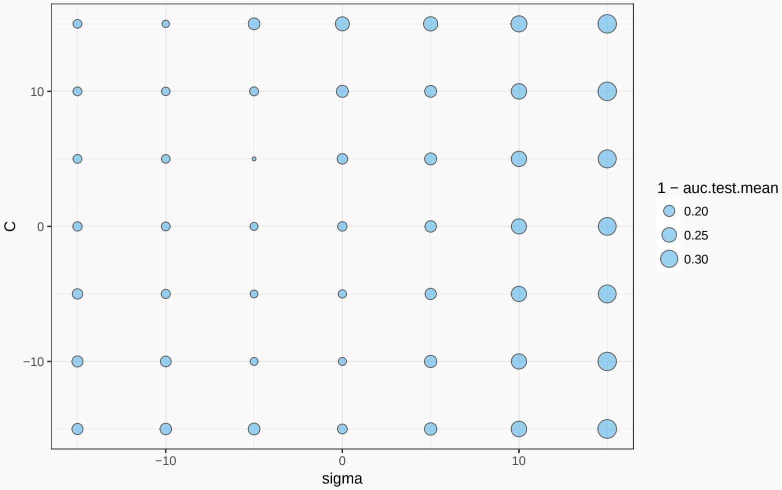
SOME GENERAL REMARKS ON TUNING

- Our optimization problem is derivative-free, we can only ask for the quality of selected points (black-box problem)
- Our optimization problem is stochastic in principle. We want to optimize expected performance and use resampling
- Evaluation of our target function will probably take quite some time; Parallelization is often mandatory
- Categorical and dependent parameters complicate the problem

GRID SEARCH

Try all combinations of finite grid

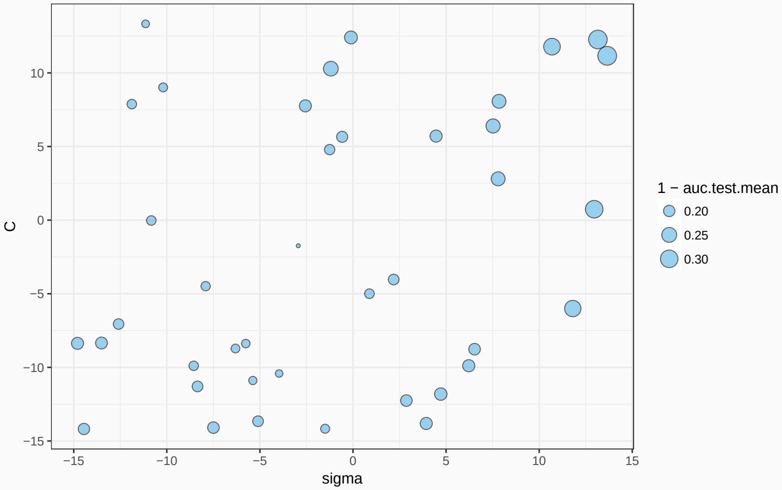
⇝ Inefficient, combinatorial explosion, searches irrelevant areas



RANDOM SEARCH

Uniformly randomly draw configurations

⇝ Scales better then grid search, easily extensible



ADVANCED TUNING TECHNIQUES

- Simulated Annealing
- Genetic Algorithm / CMAES
- Iterated F-Racing
- Model-based Optimization / Bayesian Optimization

HYPERPARAMETERS IN MLR

```
lrn = makeLearner("classif.rpart")
getParamSet(lrn)

##                    Type len  Def    Constr Req Tunable Trafo
## minsplit        integer   -   20 1 to Inf   -    TRUE     -
## minbucket       integer   -    - 1 to Inf   -    TRUE     -
## cp              numeric   - 0.01   0 to 1   -    TRUE     -
## maxcompete      integer   -    4 0 to Inf   -    TRUE     -
## maxsurrogate    integer   -    5 0 to Inf   -    TRUE     -
## usesurrogate   discrete   -    2    0,1,2   -    TRUE     -
## surrogatestyle discrete   -    0      0,1   -    TRUE     -
## maxdepth        integer   -   30  1 to 30   -    TRUE     -
## xval            integer   -   10 0 to Inf   -   FALSE     -
## parms           untyped   -    -        -   -    TRUE     -
```

HYPERPARAMETERS IN MLR

Either set them in constructor, or change them later:

```
lrn = makeLearner("classif.ksvm", C = 5, sigma = 3)
lrn = setHyperPars(lrn, C = 1, sigma = 2)
```

- Create a set of parameters
- Here we optimize an RBF SVM on logscale

```
lrn = makeLearner("classif.ksvm",
  predict.type = "prob")

# this is actually a bad way to encode the SVM space, see a few slides later
# how to do this properly
par.set = makeParamSet(
  makeNumericParam("C", lower = 0.001, upper = 100),
  makeNumericParam("sigma", lower = 0.001, upper = 100)
)
```

TUNING IN MLR

Optimize the hyperparameter of learner

```
tune.ctrl = makeTuneControlRandom(maxit = 50L)
tr = tuneParams(lrn, task = task, par.set = par.set,
  resampling = hout, control = tune.ctrl,
  measures = mlr::auc)
```

```
tr$x

## $C
## [1] 0.965
##
## $sigma
## [1] 9.92

tr$y

## auc.test.mean
##         0.754

head(as.data.frame(tr$opt.path), 3L)[, c(1,2,3,7)]

##      C sigma auc.test.mean exec.time
## 1 63.1  76.9         0.692     0.260
## 2 74.5  90.2         0.687     0.279
## 3 23.4  63.7         0.701     0.284
```

PARAMETER TYPES

```
makeNumericParam("x" ,lower = -1, upper = 1)
makeIntegerParam("x" ,lower = -1L, upper = 1L)
makeDiscreteParam("x" ,values = c("a", "b", "c"))
makeLogicalParam("x")
```

and vector-types exist for all param types

```
makeNumericVectorParam("x" , len = 3L, lower = -1, upper = 1)

##              Type len Def  Constr Req Tunable Trafo
## 1 numericvector   3   - -1 to 1   -    TRUE     -
```

DEPENDENT PARAMS AND TRAFOS

```
lrn = makeLearner("classif.ksvm")
ps = makeParamSet(
  makeDiscreteParam("kernel", values = c("polydot", "rbfdot")),
  makeNumericParam("C", lower = -15, upper = 15,
    trafo = function(x) 2^x),
  makeNumericParam("sigma", lower = -15, upper = 15,
    trafo = function(x) 2^x,
   requires = quote(kernel == "rbfdot")),
  makeIntegerParam("degree", lower = 1, upper = 5,
   requires = quote(kernel == "polydot"))
)
```

NESTED RESAMPLING

NESTED RESAMPLING

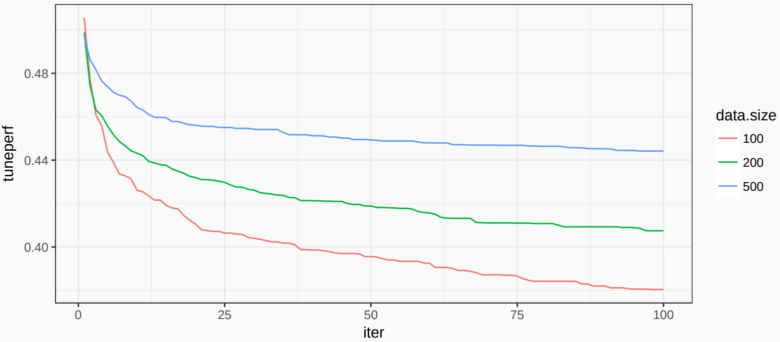
In model selection, we are interested in selecting the best model from a set of potential candidate models (e.g., different model classes, different hyperparameter settings, different feature sets).

- We cannot evaluate our finally selected inducer on the same resampling splits that we have used to perform model selection for it, e.g., to tune its hyperparameters
- By repeatedly evaluating the inducer on the same test set, or the same CV splits, information about the test can enter the algorithm
- Danger of overfitting to the resampling splits / overtuning
- The final performance estimate will be optimistically biased
- One could also see this as a problem similar to multiple testing

NESTED RESAMPLING - INSTRUCTIVE EXAMPLE

- Assume a binary classification problem with equal class sizes
- Assume an inducer $a(\mathcal{D}, \lambda)$, with hyperparameter λ
- a shall be a (nonsensical) feature-independent classifier, where λ has no effect a predicts random labels with equal probability
- Of course, a's generalization error is 50%
- A cross-validation of a (with any fixed λ) will easily show this (given that the partitioned data set for CV is not too small)
- Now lets "tune" a, by trying out 100 different λ values
- We repeat the experiment 50 times and average results

NESTED RESAMPLING - INSTRUCTIVE EXAMPLE



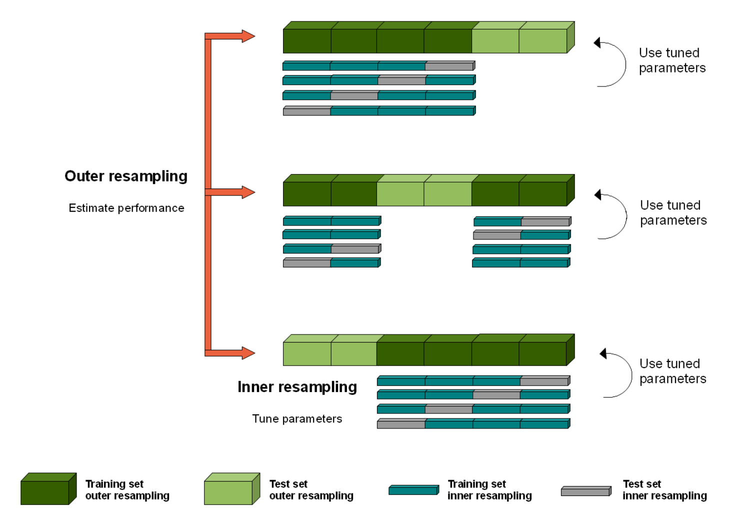
- Plotted is the best “tuning error” after k tuning iterations
- We have performed the experiment for different sizes of learning data that where cross-validated.
- Experiment was simulated with mlr’s `"classif.featureless"`

NESTED RESAMPLING

- Again, simply simulate what happens in model application.
- All parts of the model building (including model selection, preprocessing) should be embedded in the resampling, i.e., repeated for every pair of training/test data.
- For steps that themselves require resampling (e.g. hyperparameter tuning) this results in two nested resampling loops, i.e. a resampling strategy for both tuning and outer evaluation.
- Simplest form is a 3-way split into a training, optimization and test set. Inducers are trained on the training set, evaluated on the optimization set. After the final model is selected, we fit on joint training+optimization set and evaluate a final time on the test set. Note that we touch the test set only once, and have no way of "cheating".

NESTED RESAMPLING

Outer loop with 3-fold CV and inner loop with 4-fold CV



NESTED RESAMPLING: DEMO

```r
lrn = makeLearner("classif.xgboost")
ps = makeParamSet(
  makeIntegerParam("nrounds", lower = 50, upper = 300),
  makeNumericParam("eta", lower = -5, upper = -0.01,
    trafo = function(x) 2^x)
)
```

NESTED RESAMPLING: DEMO

```r
ctrl = makeTuneControlRandom(maxit = 20)

# this adds the tuning to the learner,
# we use holdout on inner resampling
inner = makeResampleDesc(method = "Holdout")
lrn2 = makeTuneWrapper(lrn, inner, par.set = ps,
  control = ctrl, measures = mmce)

# now run everything, we use CV with 2 folds
# on the outer loop
outer = makeResampleDesc(method = "CV", iters = 2)
r = resample(lrn2, sonar.task, outer,
  extract = getTuneResult)
```

NESTED RESAMPLING: DEMO

```
# lets look at some results from the outer iterations
r$extract[[1]]$x

## $nrounds
## [1] 79
##
## $eta
## [1] 0.184

r$extract[[1]]$y

## mmce.test.mean
##          0.171

r$extract[[1]]$opt.path

## Optimization path
##   Dimensions: x = 2/2, y = 1
##   Length: 20
##   Add x values transformed: FALSE
##   Error messages: TRUE. Errors: 0 / 20.
##   Exec times: TRUE. Range: 0.041 - 0.726. 0 NAs.
```

PARALLELIZATION

PARALLELIZATION

- Many tasks in statistics are embarrassingly parallel (independence assumptions, resampling, ...)
- `R` is mostly single-threaded (matrix operations may be parallel, depending on your installation)
- Multiple backends for explicit parallelization available:
  - Multicore (packages parallel/multicore)
  - Socket and MPI cluster (packages parallel/snow/Rmpi)
  - HPC-Clusters (package `batchtools`): SLURM, Torque/PBS, SGE, LSF, Docker, SSH makeshift clusters, ...

PARALLELIZATION

- We use `parallelMap` in `mlr` an abstraction for all backends
- Initialize with `parallelStart()`
- Parallelize function call with `parallelMap()`/`parallelLapply()`/...
- Stop with `parallelStop()`

```
parallelStartSocket(4)
parallelMap(function(x) x^2, 1:10)
parallelStop()
```

PARALLELIZATION

- The first loop which is marked as parallel executable will be automatically parallelized
- Which loop is suited best for parallelization depends on the number of iterations
- Levels allow fine grained control over the parallelization
  - `mlr.resample`: Each resampling iteration (a train / test step) is a parallel job.
  - `mlr.benchmark`: Each experiment "run this learner on this data set" is a parallel job.
  - `mlr.tuneParams`: Each evaluation in hyperparameter space "resample with these parameter settings" is a parallel job. How many of these can be run independently in parallel depends on the tuning algorithm.
  - `mlr.selectFeatures`: Each evaluation in feature space "resample with this feature subset" is a parallel job.

PARALLELIZATION

```r
lrns = list(makeLearner("classif.rpart"), makeLearner("classif.svm"))
rdesc = makeResampleDesc("Bootstrap", iters = 100)

parallelStartSocket(4)

## Starting parallelization in mode=socket with cpus=4.

bm = benchmark(learners = lrns, tasks = iris.task, resamplings = rdesc)

## Exporting objects to slaves for mode socket: .mlr.slave.options

## Mapping in parallel: mode = socket; cpus = 4; elements = 2.

parallelStop()

## Stopped parallelization. All cleaned up.
```

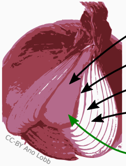
Parallelize the bootstrap instead:

```
parallelStartSocket(4, level = "mlr.resample")

## Starting parallelization in mode=socket with cpus=4.

bm = benchmark(learners = lrns, tasks = iris.task, resamplings = rdesc)

## Exporting objects to slaves for mode socket: .mlr.slave.options

## Mapping in parallel: mode = socket; cpus = 4; elements = 100.

## Exporting objects to slaves for mode socket: .mlr.slave.options

## Mapping in parallel: mode = socket; cpus = 4; elements = 100.

parallelStop()

## Stopped parallelization. All cleaned up.
```

MLR WRAPPERS

MLR LEARNER WRAPPERS

- Extend the functionality of learners by adding an **mlr** wrapper to them
- The wrapper hooks into the train and predict of the base learner and extends it
- This way, you can create a new **mlr** learner with extended functionality
- Hyperparameter definition spaces get joined!




```
lrn = makeLearner(...)
lrn = makeRemoveConstantFeaturesWrapper(lrn, ...)
lrn = makeDummyFeaturesWrapper(lrn, ...)
lrn = makeImputeWrapper(lrn ...)
train(lrn, tsk, ...)
```

AVAILABLE WRAPPERS

- *Preprocessing*: PCA, normalization, dummy encoding, ...
- *Parameter Tuning*: grid, optim, random search, genetic algorithms, CMAES, iRace, MBO
- *Filter*: correlation- and entropy-based, $\mathcal{X}^2$-test, mRMR, ...
- *Feature Selection*: (floating) sequential forward/backward, exhaustive search, genetic algorithms, ...
- *Impute*: dummy variables, imputations with mean, median, min, max, empirical distribution or other learners
- *Bagging* to fuse learners on bootstraped samples
- *Stacking* to combine models in heterogenous ensembles
- *Over- and Undersampling* for unbalanced classification

WRAPPER EXAMPLE I

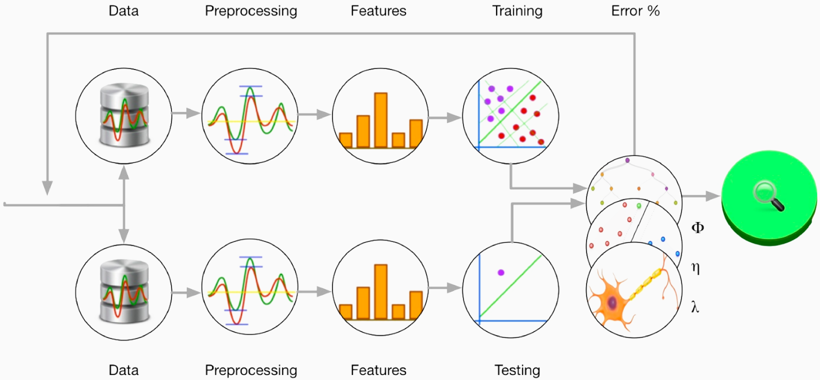
```
set.seed(1)
library(ggplot2); library(RColorBrewer)
lrn = makeLearner("classif.randomForest", ntree = 200)
lrn = makeRemoveConstantFeaturesWrapper(learner = lrn)
lrn = makeDownsampleWrapper(learner = lrn)
lrn = makeFilterWrapper(lrn, fw.method = "gain.ratio")
filterParams(getParamSet(lrn), tunable = TRUE, type = c("numeric", "integer"))

##                 Type len Def      Constr Req Tunable Trafo
## fw.perc      numeric   -   -      0 to 1   -    TRUE     -
## fw.abs       integer   -   -    0 to Inf   -    TRUE     -
## fw.threshold numeric   -   - -Inf to Inf   -    TRUE     -
## dw.perc      numeric   -   1      0 to 1   -    TRUE     -
## ntree        integer   - 500    1 to Inf   -    TRUE     -
## mtry         integer   -   -    1 to Inf   -    TRUE     -
## nodesize     integer   -   1    1 to Inf   -    TRUE     -
## maxnodes     integer   -   -    1 to Inf   -    TRUE     -
```

WRAPPER EXAMPLE II

```
ps = makeParamSet(
  makeNumericParam("fw.perc", lower = 0.1, upper = 1),
  makeNumericParam("dw.perc", lower = 0.1, upper = 1))
res = tuneParams(lrn, sonar.task, resampling = cv10, par.set = ps,
  control = makeTuneControlGrid(resolution = 7), show.info = FALSE)
res

## Tune result:
## Op. pars: fw.perc=1; dw.perc=0.85
## mmce.test.mean=0.169
```

CPO

MACHINE LEARNING PIPELINES



COMPOSABLE PREPROCESSING OPERATORS

- `mlrCPO`: **C**omposable **P**reprocessing **O**perators for mlr
- Google Summer of Code 2017 Project: Operator Based Machine Learning Pipeline Construction
- `dplyr`-like composition for mlr tasks and learners

```
task = iris.task
task = task %>>% cpoScale(scale = FALSE) %>>% cpoPca() %>>%  # pca
  cpoFilterChiSquared(abs = 3) %>>%  # filter
  cpoModelMatrix(~ 0 + .^2)  # interactions
head(getTaskData(task))

##     PC1    PC2     PC3 PC1:PC2 PC1:PC3  PC2:PC3 Species
## 1 -2.68 -0.319  0.0279   0.857 -0.0749 -0.00892  setosa
## 2 -2.71  0.177  0.2105  -0.480 -0.5712  0.03725  setosa
## 3 -2.89  0.145 -0.0179  -0.419  0.0517 -0.00259  setosa
## 4 -2.75  0.318 -0.0316  -0.874  0.0866 -0.01005  setosa
## 5 -2.73 -0.327 -0.0901   0.892  0.2458  0.02943  setosa
## 6 -2.28 -0.741 -0.1687   1.691  0.3847  0.12505  setosa
```

CPOS MODIFY DATA I

CPOs can be *independent* objects describing a preprocessing pipeline:

```
pipeline = cpoImputeMax() %>>% cpoDummyEncode() %>>% cpoFilterVariance()
getParamSet(pipeline)

##                                    Type len    Def      Constr Req Tunable Trafo
## impute.max.multiplier           numeric   -      1 -Inf to Inf   -    TRUE     -
## impute.max.impute.new.levels    logical   -   TRUE           -   -    TRUE     -
## impute.max.recode.factor.levels logical   -   TRUE           -   -    TRUE     -
## dummyencode.reference.cat       logical   -  FALSE           -   -    TRUE     -
## variance.perc                   numeric   - <NULL>      0 to 1   -    TRUE     -
## variance.abs                    integer   - <NULL>    0 to Inf   -    TRUE     -
## variance.threshold              numeric   - <NULL> -Inf to Inf   -    TRUE     -
```

CPOS MODIFY DATA II

```
str(getHyperPars(pipeline))

## List of 7
##  $ impute.max.multiplier            : num 1
##  $ impute.max.impute.new.levels     : logi TRUE
##  $ impute.max.recode.factor.levels: logi TRUE
##  $ dummyencode.reference.cat        : logi FALSE
##  $ variance.perc                    : NULL
##  $ variance.abs                     : NULL
##  $ variance.threshold               : NULL
```

CPOS MODIFY DATA III

```
pipeline = setHyperPars(pipeline, variance.perc = 0.5)
tsk = iris.task %>>% pipeline
tsk

## Supervised task: iris-example
## Type: classif
## Target: Species
## Observations: 150
## Features:
##    numerics     factors     ordered functionals
##           2           0           0           0
## Missings: FALSE
## Has weights: FALSE
## Has blocking: FALSE
## Has coordinates: FALSE
## Classes: 3
##     setosa versicolor  virginica
##         50         50         50
## Positive class: NA
```

CPOS ENHANCE LEARNERS

CPOs can be preprocessing pipelines for learners

```
lrn1 = makeLearner("classif.logreg")
getLearnerProperties(lrn1)

## [1] "twoclass" "numerics" "factors"  "prob"     "weights"

lrn2 = pipeline %>>% lrn1
getLearnerProperties(lrn2)

## [1] "missings" "numerics" "twoclass" "prob"
```

- mlrCPO takes care of consistent transformation of train and test data!

LISTING CPOS

Builtin CPOs can be listed with listCPO().

```
listCPO()[, c("name", "category", "subcategory")]
```

	name	category	subcategory
11	cpoDropConstants	data	cleanup
36	cpoFixFactors	data	cleanup
10	cpoCollapseFact	data	factor data preprocessing
4	cpoAsNumeric	data	feature conversion
15	cpoDummyEncode	data	feature conversion
13	cpoImpactEncodeClassif	data	feature conversion
14	cpoImpactEncodeRegr	data	feature conversion
12	cpoProbEncode	data	feature conversion
55	cpoQuantileBinNumerics	data	feature conversion
61	cpoSelect	data	feature selection
62	cpoSelectFreeProperties	data	feature selection
51	cpoAddCols	data	features
50	cpoMakeCols	data	features
1	cpoApplyFun	data	general data preprocessing
53	cpoModelMatrix	data	general

MLRCPO: OVERVIEW

- CPOs are a very powerful and versatile for preprocessing
- Create custom learners (comparable to wrappers)
- Apply same preprocessing to multiple tasks
- CPO-Pipelines with learners can be JOINTLY tuned - e.g. with Bayesian optimization with mlrMBO
- Detailed instructions and documentation: https://github.com/mlr-org/mlrCPO