

# Machine Learning with R

## Introduction

**mlr** offers a unified interface for the basic building blocks of machine learning: tasks, learners, hyperparameters, etc.

**Tasks** contain a description of a task (classification, regression, clustering, etc.) and a data set.

**Learners** specify a machine learning algorithm (GLM, SVM, xgboost, etc.) and its parameters.

**Hyperparameters** are learner settings that can be specified directly or tuned. A **parameter set** lists the possible hyperparameters for a given learner.

**Wrapped Models** are learners that have been trained on a task and can be used to make predictions.

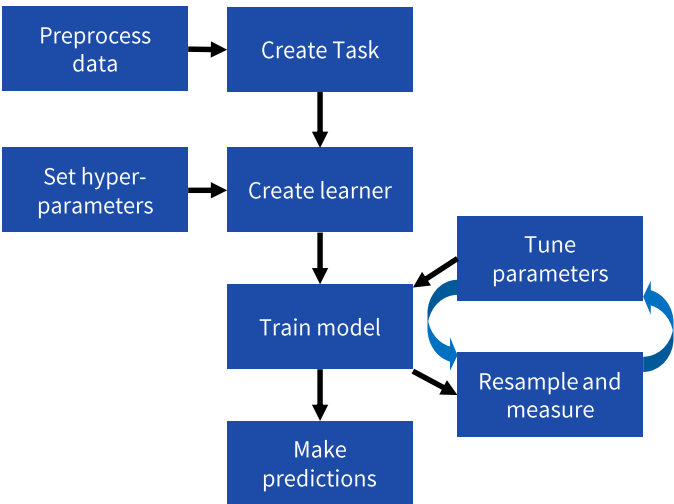
**Predictions** are the results of applying a model to either new data or the original training data.

**Measures** control how learner performance is evaluated, e.g. RMSE, LogLoss, AUC, etc.

**Resampling** estimates generalization performance by separating training data from test data. Common strategies include holdout and cross-validation.

Links: [Tutorial](#) | [CRAN](#) | [Github](#)

## mlr workflow



## Setup

### Preprocessing data

`createDummyFeatures(obj=, target=, method=, cols=)`  
Creates (0,1) flags for each non-numeric variable excluding `target`. Can be applied to entire dataset or only specific `cols`

`normalizeFeatures(obj=, target=, method=, cols=, range=, on.constant=)`  
Normalizes numerical features according to specified `method`:  
• "center" (subtract mean)  
• "scale" (divide by std. deviation)  
• "standardize" (center and scale)  
• "range" (linear scale to given range, default `range=c(0,1)`)

`mergeSmallFactorLevels(task=, cols=, min.perc=)`  
Combine infrequent factor levels into a single merged level

`summarizeColumns(obj=)` where `obj` is a data.frame or task.  
Provides type, NA, and distributional data about each column

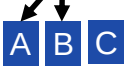
See also `capLargeValues` `dropFeatures` `removeConstantFeatures` `summarizeLevels`

### Creating a task



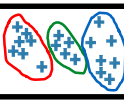
`makeClassifTask(data=, target=)`  
Classification of a target variable, with optional positive class `positive`

0 63 100

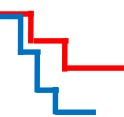


`makeRegrTask(data=, target=)`  
Regression on a target variable

`makeMultilabelTask(data=, target=)`  
Classification where the target can belong to more than one class per observation



`makeClusterTask(data=)`  
Unsupervised clustering on a data set



`makeSurvTask(data=, target=c("time", "event"))`  
Survival analysis with a survival time column and an event column



`makeCostSensTask(data=, costs=)`  
Cost-sensitive classification where each observation-cost pair has a specified cost

Other arguments that can be passed to a `task`:

- `weights`=Weighting vector to apply to observations
- `blocking`=Factor vector where each level indicates a block of observations that will not be split up in resampling

### Making a learner

`makeLearner(cl=predict.type=, ..., par.vals=)`  
Choose an algorithm class to perform the task and determine what that algorithm will predict

- `cl`=name of algorithm, e.g. "classif.xgboost" "regr.randomForest" "cluster.kmeans"
- `predict.type="response"` returns a prediction type that matches the source data; "prob" returns a predicted probability for classification problems only; "se" returns the a standard error of the prediction for regression problems only. Only certain learners can return "prob" and "se"
- `par.vals`=takes a list of hyperparameters and passes them to the learner; parameters can also be passed directly (...)

You can make multiple learners at once with `makeLearners()`

- mlr has integrated over 170 different learning algorithms
- Full list: `View(listLearners())` shows all learners
  - Available learners for a task: `View(listLearners(task))`
  - Filtered list: `View(listLearners("classif", properties=c("prob", "factors")))` shows all classification learners "classif" which can predict probabilities "prob" and handle factor inputs "factors"
  - See also `getLearnerProperties()`

## Training & Testing

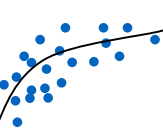
### Setting hyperparameters



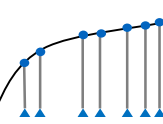
`setHyperPars(learner=, ...)`  
Set the hyperparameters (settings) for each learner, if you don't want to use the defaults. You can also specify hyperparameters in the `makeLearner()` call

`getParamSet(learner=)`  
Show the possible universe of parameters for your learner; can take a learner directly, or a text string such as "classif.qda"

### Train a model and predict



`train(learner=, task=)`  
Train a model (`WrappedModel`) by applying a learner to a task. By default, the model will train on all observations. The underlying model can be extracted with `getLearnerModel()`



`predict(object=, task=, newdata=)`  
Use a trained model to make predictions on a task or dataset. The resulting `pred` object can be viewed with `View(pred)` or accessed by `as.data.frame(pred)`

### Measuring performance

`performance(pred=, measures=)`  
Calculate performance of predictions according to one or more of several measures (use `listMeasures()` for full list):

- `classifacc auc bac ber brier[scaled] f1 fdr fn fnr fp fpr gmean multiclass[.au1u .aunp .aunu .brier] npv ppv qsr ssr tn tnr tp tpr wkappa`
- `regrarsq expvar kendalltau mae mape medae medse mse msle rae rmse rmsle rrse rsq sae spearmanrho sse`
- `clusterdb dunn G1 G2 silhouette`
- `multilabelmultilabel[.f1 .subset01 .tpr .ppv .acc .hamloss]`
- `costsens mcp meancosts`
- `surv cindex`
- `other featperc timeboth timepredict timetrain`

For detailed performance data on classification tasks, use:

- `calculateConfusionMatrix(pred=)`
- `calculateROCMeasures(pred=)`

### Resampling a learner

`makeResampleDesc(method=, ..., stratify=)`  
`method` must be one of the following:  
• "CV" (cross-validation, for number of folds use `iters=`)  
• "LOO" (leave-one-out cross-validation, for folds use `iters=`)  
• "RepCV" (repeated cross-validation, for number of repetitions use `reps=`, for folds use `folds=`)  
• "Subsample" (aka Monte-Carlo cross-validation, for iterations use `iters=`, for train % use `split=`)  
• "Bootstrap" (out-of-bag bootstrap, uses `iters=`)  
• "Holdout" (for train % use `split=`)  
`stratify` keeps target proportions consistent across samples.

`makeResampleInstance(desc=, task=)` can reduce noise by ensuring the resampling is done identically every time.

`resample(learner=, task=, resampling=, measures=)`  
Train and test model according to specified resampling strategy.

mlr includes several pre-specified resample descriptions: `cv2` (2-fold cross-validation), `cv3`, `cv5`, `cv10`, `hout` (holdout with split 2/3 for training, 1/3 for testing). Convenience functions also exist to `resample()` with a specific strategy: `crossval()`, `repcv()`, `holdout()`, `subsample()`, `bootstrapOoB()`, `bootstrapB632()`, `bootstrapB632plus()`

## Refining Performance

### Tuning hyperparameters

Set search space using `makeParamSet(make<type>Param())`

- `makeNumericParam(id=, lower=, upper=, trafo=)`
- `makeIntegerParam(id=, lower=, upper=, trafo=)`
- `makeIntegerVectorParam(id=, len=, lower=, upper=, trafo=)`
- `makeDiscreteParam(id=, values=c(...))` (can also be used to test discrete values of numeric or integer parameters)

`trafo` transforms the parameter output using a specified function, e.g. `lower=-2, upper=2, trafo=function(x) 10^x` would test values between 0.01 and 100, scaled exponentially. Other acceptable parameter types include `Logical` `LogicalVector` `CharacterVector` `DiscreteVector`

Set a search algorithm with `makeTuneControl<type>()`

- `Grid(resolution=10L)` Grid of all possible points
- `Random(maxit=100)` Randomly sample search space
- `MBO(budget=)` Use Bayesian model-based optimization
- `Trace(n.instances=)` Iterated racing process
- Other types: `CMAES`, `Design`, `GenSA`

Tune using `tuneParams(learner=, task=, resampling=, measures=, par.set=, control=)`

## Quickstart

### Prepare data for training and testing

```
library(mlbench)
data(Soybean)
soy = createDummyFeatures(Soybean, target="Class")
tsk = makeClassifTask(data=soy, target="Class")
ho = makeResampleInstance("Holdout", tsk)
tsk.train = subsetTask(tsk, ho$train.inds[[1]])
tsk.test = subsetTask(tsk, ho$test.inds[[1]])
```

Convert the factor inputs in the Soybean dataset into (0,1) dummy features which can be used by the XGboost algorithm. Create a task to predict the "Class" column. Create a train set with 2/3 of data and a test set with the remaining 1/3 (default).

### Create learner and evaluate performance

```
lrn = makeLearner("classif.xgboost", nrounds=10)
cv = makeResampleDesc("CV", iters=5)
res = resample(lrn, tsk.train, cv, acc)

Create an XGboost learner which will build 10 trees. Then test performance using 5-fold cross-validation. Accuracy should be between 0.90-0.92.
```

### Tune hyperparameters and retrain model

```
ps = makeParamSet(makeNumericParam("eta", 0, 1),
  makeNumericParam("lambda", 0, 200),
  makeIntegerParam("max_depth", 1, 20))
tc = makeTuneControlMBO(budget=100)
tr = tuneParams(lrn, tsk.train, cv5, acc, ps, tc)
lrn = setHyperPars(lrn, par.vals=tr$x)

Tune hyperparameters eta, lambda, and max_depth by defining a search space and using Model Based Optimization (MBO) to control the search. Then perform 100 rounds of 5-fold cross-validation, improving accuracy to ~0.93. Update the XGboost learner with the tuned hyperparameters.
```

```
mdl = train(lrn, tsk.train)
prd = predict(mdl, tsk.test)
calculateConfusionMatrix(prd)
mdl = train(lrn, tsk)

Train the model on the train set and make predictions on the test set. Show performance as a confusion matrix. Finally, re-train model on the full set to use on new data. You are now ready to go out into the real world and make 93% accurate predictions!
```

## Configuration

- mlr's default settings can be changed using `configureMlr()`:
- `show.info` Whether to show verbose output by default when training, tuning, resampling, etc. (`TRUE`)
  - `on.learner.error` How to handle a learner error. "stop" halts execution, "warn" returns NAs and displays a warning, "quiet" returns NAs with no warning ("stop")
  - `on.learner.warning` How to handle a learner warning. "warn" displays a warning, "quiet" suppresses it ("warn")
  - `on.par.without.desc` How to handle a parameter with no description. "stop", "warn", "quiet" ("stop")
  - `on.par.out.of.bounds` How to handle a parameter with an out-of-bounds value. "stop", "warn", "quiet" ("stop")
  - `on.measure.not.applicable` How to handle a measure not applicable to a learner. "stop", "warn", "quiet" ("stop")
  - `show.learner.output` Whether to show learner output to the console during training (`TRUE`)
  - `on.error.dump` Whether to create an error dump for crashed learners if `on.learner.error` is not set to "stop" (`TRUE`)

Use `getMlrOptions()` to see current settings

## Parallelization

mlr works with the `parallelMap` package to take advantage of multicore and cluster computing for faster operations. mlr automatically detects which operations are able to run in parallel.

To begin parallel operation use:

- `parallelStart(mode=, cpus=, level=)`
- `mode` determines how the parallelization is performed:
    - "local" no parallelization applied, simply uses `mapply`
    - "multicore" multicore execution on a single machine, uses `parallel::mclapply`. Not available in Windows.
    - "socket" multicore execution in socket mode
    - "mpi" Snow MPI cluster on one or multiple machines using `parallel::makeCluster` and `parallel::clusterMap`
    - "BatchJobs" Batch queuing HPC clusters using `BatchJobs::batchMap`
  - `cpus` determines how many logical cores will be used
  - `level` controls parallelization: "mlr.benchmark" "mlr.resample" "mlr.selectFeatures" "mlr.tuneParams" "mlr.ensemble"

To end parallelization, use `parallelStop()`

## Imputation

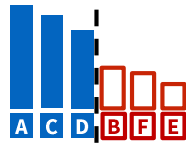
- `impute(obj=, target=, cols=, dummy.cols=, dummy.type=)`  
Applies specified logic to data frame or task containing NAs and returns an imputation description which can be used on new data
- `obj`=data frame or task on which to perform imputation
  - `target`=specify target variable which will not be imputed
  - `cols`=column names and logic for imputation\*
  - `dummy.cols`=column names to create a NA (T/F) column\*
  - `dummy.type`=set to "numeric" to use (0,1) instead of (T/F)
- \*Can also use `classes` and `dummy.classes` in place of `cols`

Imputation logic is passed to `cols` or `classes` via a list, e.g.:  
`cols=list(V1=imputeMean())` where `V1` is the column to which to apply the imputation, and `imputeMean()` is the imputation method. Available imputation methods include:  
`imputeConst(const=)` `imputeMedian()` `imputeMode()` `imputeMin(multiplier=)` `imputeMax(multiplier=)` `imputeNormal(mean=, sd=)` `imputeHist(breaks=, use.mids=)` `imputeLearner(learner=, features=)`  
`impute` returns a list containing the imputed dataset or task as well as an imputation description that can be used to reapply the same imputation to new data using `reimpute`

`reimpute(obj=, desc=)` Imputes missing values on a task or dataset (`obj`) using a description (`desc`) created by `impute`

## Feature Extraction

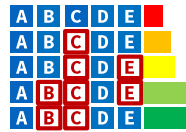
### Feature filtering



`filterFeatures(task=, method=, perc=, abs=, threshold=)`  
Uses a learner-agnostic feature evaluation method to rank feature importance, then includes only features in the top n percent (`perc=`), top n (`abs=`), or which meet a set performance threshold (`threshold=`).

Outputs a task with features that failed the test omitted. `method` defaults to "randomForestSRC.rfsrc", but can be set to: "anova.test" "carscore" "cforest.importance" "chi.squared" "gain.ratio" "information.gain" "kruskal.test" "linear.correlation" "mrmr" "oneR" "permutation.importance" "randomForest.importance" "randomForestSRC.rfsrc" "randomForestSRC.var.select" "rank.correlation" "relief" "symmetrical.uncertainty" "univariate.model.score" "variance"

### Feature selection



`selectFeatures(learner=, task=, resampling=, measures=, control=)`  
Uses a feature selection algorithm (`control`) to resample and build a model repeatedly using different feature sets each time in order to find the best set.

Available controls include:

- `makeFeatSelControlExhaustive(max.features=)` Try every combination of features up to optional `max.features`
- `makeFeatSelControlRandom(maxit=, prob=, max.features=)` Randomly sample features with probability `prob` (default 0.5) until `maxit` (default 100) iterations; return the best one found
- `makeFeatSelControlSequential(method=, maxit=, max.features=, alpha=, beta=)` Perform an iterative search using a `method` from the following: "sfs" forward search, "sbs" backward search, "sffs" floating forward search, "sfbs" floating backward search. `alpha` indicates minimum improvement required to add a feature; `beta` indicates minimum required to remove a feature
- `makeFeatSelControlGA(maxit=, max.features=, mu=, lambda=, crossover.rate=, mutation.rate=)` Genetic algorithm trains on random feature vectors, then uses crossover on the best performers to produce 'offspring', repeated over generations. `mu` is size of parent population, `lambda` is size of children population, `crossover.rate` is probability of choosing a bit from first parent, `mutation.rate` is probability of flipping a bit (on or off)

`selectFeatures` returns a `FeatSelResult` object which contains optimal features and an optimization path. To apply feature selection result (`fsr`) to your task (`tsk`), use:  
`tsk = subsetTask(tsk, features=fsr$x)`

## Benchmarking

`benchmark(learners=, tasks=, resamplings=, measures=)`  
Allows easy comparison of multiple learners on a single task, a single learner on multiple tasks, or multiple learners on multiple tasks. Returns a benchmark result object.

Benchmark results can be accessed with a variety of functions beginning with `getBMR<object>`: `AggrPerformance` `FeatSelResults` `FilteredFeatures` `LearnerIds` `LearnerShortNames` `Learners` `MeasureIds` `Measures` `Models` `Performances` `Predictions` `TaskDescs` `TaskIds` `TuneResults`

mlr contains several toy tasks which are useful for benchmarking: `agri.task` `bc.task` `bh.task` `costiris.task` `iris.task` `lung.task` `mtcars.task` `pid.task` `sonar.task` `wdbc.task` `yeast.task`

## Visualization

### Performance

`generateThreshVsPerfData(obj=, measures=)` Measure performance at different probability cutoffs to determine optimal decision threshold for binary classification problems

- `plotThreshVsPerf(obj)` Plot visual representation of threshold curve(s) from `ThreshVsPerfData`
- `plotROCCurves(obj)` Plot receiver operating characteristic (ROC) curve from `ThreshVsPerfData`. Must set `measures=list(fpr, tpr)`

### Residuals

- `plotResiduals(obj=)` Plots residuals for `Prediction` or `BenchmarkResult`

### Learning curve

`generateLearningCurveData(learners=, task=, resampling=, percs=, measures=)` Measure performance of learner(s) trained on different percentages of task data

- `plotLearningCurve(obj=)` Plot curve showing learner performance vs. proportion of data used, uses `LearningCurveData`

### Feature importance

`generateFilterValuesData(task=, method=)` Get feature importance rankings using specified filter method

- `plotFilterValues(obj=)` Plot bar chart of feature importance based on filter method using `FilterValuesData`

### Hyperparameter tuning

`generateHyperParsEffectData(tune.result=)` Get the impact of different hyperparameter settings on model performance

- `plotHyperParsEffect(hyperpars.effect.t.data=, x=, y=, z=)` Create a plot showing hyperparameter impact on performance using `HyperParsEffectData`

See also:

- `plotOptPath(op=)` Display details of optimization process. Takes `<obj>$opt.path`, where `<obj>` is an object of class `tuneResult` or `FeatSelResult`
- `plotTuneMultiCritResult(res=)` Show pareto front for results of tuning to multiple performance measures

### Partial dependence

`generatePartialDependenceData(obj=, input=)` Get partial dependence of model (`obj`) prediction over each feature of data (`input`)

- `plotPartialDependence(obj=)` Plots partial dependence of model using `PartialDependenceData`

### Benchmarking

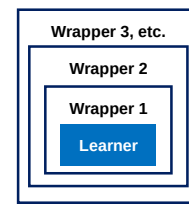
- `plotBMRBoxplots(bmr=)` Distribution of performances
- `plotBMRSummary(bmr=)` Scatterplot of avg. performances
- `plotBMRRanksAsBarChart(bmr=)` Rank learners in bar plot

### Other

- `generateCritDifferencesData(bmr=, measure=, p.value=, test=)` Perform critical-differences test using either the Bonferroni-Dunn ("bd") or "Nemenyi" test
- `plotCritDifferences(obj=)`

- `generateCalibrationData(obj=)` Evaluate calibration of probability predictions vs. true incidence
- `plotCalibration(obj=)`

## Wrappers



**Wrappers** fuse a learner with additional functionality. mlr treats a learner with wrappers as a single learner, and hyperparameters of wrappers can be tuned jointly with underlying model parameters. Models trained with wrappers will apply them to new data.

### Preprocessing and imputation

`makeDummyFeaturesWrapper(learner=)`  
`makeImputeWrapper(learner=, classes=, cols=)`  
`makePreprocWrapper(learner=, train=, predict=)`  
`makePreprocWrapperCaret(learner=, ...)`  
`makeRemoveConstantFeaturesWrapper(learner=)`

### Class imbalance

`makeOverBaggingWrapper(learner=)`  
`makeSMOTWrapper(learner=)`  
`makeUndersampleWrapper(learner=)`  
`makeWeightedClassesWrapper(learner=)`

### Cost-sensitive learning

`makeCostSensClassifWrapper(learner=)`  
`makeCostSensRegrWrapper(learner=)`  
`makeCostSensWeightedPairsWrapper(learner=)`

### Multilabel classification

`makeMultilabelBinaryRelevanceWrapper(learner=)`  
`makeMultilabelClassifierChainsWrapper(learner=)`  
`makeMultilabelDBRWrapper(learner=)`  
`makeMultilabelNestedStackingWrapper(learner=)`  
`makeMultilabelStackingWrapper(learner=)`

### Other

`makeBaggingWrapper(learner=)`  
`makeConstantClassWrapper(learner=)`  
`makeDownsampleWrapper(learner=, dw.perc=)`  
`makeFeatSelWrapper(learner=, resampling=, control=)`  
`makeFilterWrapper(learner=, fw.perc=, fw.abs=, fw.threshold=)`  
`makeMultiClassWrapper(learner=)`  
`makeTuneWrapper(learner=, resampling=, par.set=, control=)`

## Nested Resampling

mlr supports **nested resampling** for complex operations such as tuning and feature selection through wrappers. In order to get a good estimate of generalization performance and avoid data leakage, both an outer (for tuning/feature selection) and an inner (for the base model) resampling process are advised.

- Outer resampling can be specified in `resample` or `benchmark`
- Inner resampling can be specified in `makeTuneWrapper`, `makeFeatSelWrapper`, etc.

## Ensembles

`makeStackedLearner(base.learners=, super.learner=, method=)` Combines multiple learners to create an ensemble

- `base.learners`=learners to use for initial predictions
- `super.learner`=learner to use for final prediction
- `method`=how to combine base learner predictions:
  - "average" simple average of all base learners
  - "stack.nocv", "stack.cv" train super learner on results of base learners, with or without cross-validation
  - "hill.climb" search for optimal weighted average
  - "compress" with a neural network for faster performance