Package 'mlr'

July 21, 2014

Title mlr: Machine Learning in R.

Description Interface to a large number of classification and regression techniques, including machine-readable parameter descriptions. There is also an experimental extension for survival analysis, clustering and general, example-specific cost-sensitive learning. Generic resampling,including cross-validation, bootstrapping and subsampling. Hyperparameter tuning with modern optimization techniques, for single- and multi-objective problems. Filter and wrapper methods for feature selection. Extension of basic learners with additional operations common in machine learning, also allowing for easy nested resampling. Most operations can be parallelized.

URL https://github.com/berndbischl/mlr

BugReports https://github.com/berndbischl/mlr/issues

License BSD_3_clause + file LICENSE

Depends R (>= 3.0.0), ParamHelpers (>= 1.2), BBmisc (>= 1.7), stats

Imports parallelMap (>= 1.1), codetools, survival, checkmate (>= 1.1)

Suggests testthat, ada, adabag, caret, class, cluster, clusterSim,clValid, cmaes, CoxBoost, crs, DiceKriging, DiceOptim,DiscriMiner, e1071, earth, emoa, FNN, FSelector, gbm, GenSA,ggplot2, glmnet, Hmisc, irace, kernlab, kknn, klaR, LiblineaR,mboost, mco, mda, mlbench, mRMRe, nnet, party, penalized, pls,pROC, randomForest, randomForestSRC, reshape2, rrlda,robustbase, rpart, rsm, RWeka, ROCR, stepPlr

LazyData yes

ByteCompile yes

Version 2.1

Author Bernd Bischl [aut, cre],Michel Lang [aut],Jakob Bossek [aut],Leonard Judt [aut],Jakob Richter [aut],Tobias Kuehn [aut],Erich Studeru

Maintainer Bernd Bischl <bernd_bischl@gmx.net>

NeedsCompilation yes

Repository CRAN

Date/Publication 2014-07-21 20:07:56

R topics documented:

`Aggregation` *An aggregation method reduces the performance values of the test (and possibly the training sets) to a single value. To see all possible, implemented aggregations look at* `aggregations`.

Description

The aggregation can access all relevant information of the result after resampling and combine them into a single value. Though usually something very simple like taking the mean of the test set performances is done.

Details

Object members:

id [`character(1)`] Name of aggregation method.

fun [`function(task, perf.test, perf.train, measure, group, pred)`] Aggregation function.

See Also

`makeAggregation`

`aggregations` *Aggregation methods.*

Description

- **test.mean**
  Mean of performance values on test sets.
- **test.sd**
  Standard deviation of performance values on test sets.
- **test.median**
  Median of performance values on test sets.
- **test.min**
  Minimum of performance values on test sets.
- **test.max**
  Maximum of performance values on test sets.
- **test.sum**
  Sum of performance values on test sets.
- **train.mean**
  Mean of performance values on training sets.
- **train.sd**
  Standard deviation of performance values on training sets.
- **train.median**
  Median of performance values on training sets.
- **train.min**
  Minimum of performance values on training sets.
- **train.max**
  Maximum of performance values on training sets.
- **train.sum**
  Sum of performance values on training sets.
- **b632**
  Aggregation for B632 bootstrap.
- **b632plus**
  Aggregation for B632+ bootstrap.
- **testgroup.mean**
  Performance values on test sets are grouped according to resampling method. The mean for very group is calculated, then the mean of those means. Mainly used for repeated CV.

Usage

```
test.mean

test.sd
```

test.median

test.min

test.max

test.sum

test.range

test.sqrt.of.mean

train.mean

train.sd

train.median

train.min

train.max

train.sum

train.range

train.sqrt.of.mean

b632

b632plus

testgroup.mean

Format

None

See Also

Aggregation

---

analyzeFeatSelResult *Show and visualize the steps of feature selection.*

Description

This function prints the steps `selectFeatures` took to find its optimal set of features and the reason why it stopped. It can also print information about all calculations done in each intermediate step.

Currently only implemented for sequential feature selection.

Usage

```
analyzeFeatSelResult(res, reduce = TRUE)
```

Arguments

`res`	[`FeatSelResult`] The result of of `selectFeatures`.
`reduce`	[`logical(1)`] Per iteration: Print only the selected feature (or all features that were evaluated)? Default is `TRUE`.

Value

`invisible(NULL)` .

See Also

Other featsel: `FeatSelControl`, `FeatSelControlExhaustive`, `FeatSelControlGA`, `FeatSelControlRandom`, `FeatSelControlSequential`, `makeFeatSelControlExhaustive`, `makeFeatSelControlGA`, `makeFeatSelControlRandom`, `makeFeatSelControlSequential`; `getFeatSelResult`; `makeFeatSelWrapper`; `selectFeatures`

---

asROCRPrediction *Converts predictions to a format package ROCR can handle.*

Description

Converts predictions to a format package ROCR can handle.

Usage

```
asROCRPrediction(pred)
```

Arguments

`pred`	[`Prediction`] Prediction object.

benchmark *Benchmark experiment for multiple learners and tasks.*

Description

Complete benchmark experiment to compare different learning algorithms across one or more tasks w.r.t. a given resampling strategy. Experiments are paired, meaning always the same training / test sets are used for the different learners. Furhtermore, your learners can be automatically tuned using `makeTuneWrapper`.

Usage

```
benchmark(learners, tasks, resamplings, measures,
  show.info = getMlrOption("show.info"))
```

Arguments

`learners`	[(list of) `Learner`] Learning algorithms which should be compared.
`tasks`	[(list of) `Task`] Tasks that learners should be run on.
`resamplings`	[(list of) `ResampleDesc`	`ResampleInstance`] Resampling strategy for each tasks. If only one is provided, it will be replicated to match the number of tasks. If missing, a 10-fold cross validation is used.
`measures`	[(list of) `Measure`] Performance measures for all tasks. If missing, the default measure of the first task is used.
`show.info`	[`logical(1)`] Print verbose output on console? Default is set via `configureMlr`.

Value

`BenchmarkResult` .

See Also

Other benchmark: `BenchmarkResult`; `FilterResult`, `getFilterResult`; `getAggrPerformances`; `getFeatSelResult`; `getPerformances`; `getPredictions`; `getTuneResult`

BenchmarkResult *Result of a benchmark run.*

Description

Container for results of benchmarked experiments using benchmark. The structure of the object itself is rather complicated, it is recommended to retrive required information via link{getAggrPerformances}, link{getPredictions}, getPerformances, getFeatSelResult, getTuneResult or getFilterResult. Alternatively, you can convert the object using as.data.frame

See Also

Other benchmark: FilterResult, getFilterResult; benchmark; getAggrPerformances; getFeatSelResult; getPerformances; getPredictions; getTuneResult

configureMlr *Configures the behavior of the package.*

Description

Configuration is done by setting custom options.

If you do not set an option here, its current value will be kept.

If you call this function with an empty argument list, everything is set to its defaults.

Usage

```
configureMlr(show.info, on.learner.error, on.learner.warning,
  on.par.without.desc, show.learner.output)
```

Arguments

show.info	[logical(1)] Some methods of mlr support a show.info argument to enable verbose output on the console. This option sets the default value for these arguments. Setting the argument manually in one of these functions will overwrite the default value for that specific function call. Default is TRUE.
on.learner.error	[character(1)] What should happen if an error in an underlying learning algorithm is caught: "stop": R exception is generated. "warn": A FailureModel will be created, which predicts only NAs and a warning will be generated. "quiet": Same as "warn" but without the warning. Default is "stop".

on.learner.warning
: [character(1)]
What should happen if a warning in an underlying learning algorithm is generated:
"warn": The warning is generated as usual.
"quiet": The warning is suppressed.
Default is "warn".

on.par.without.desc
: [character(1)]
What should happen if a parameter of a learner is set to a value, but no parameter description object exists, indicating a possibly wrong name:
"stop": R exception is generated.
"warn": Warning, but parameter is still passed along to learner.
"quiet": Same as "warn" but without the warning.
Default is "stop".

show.learner.output
: [logical(1)]
Should the output of the learning algorithm during training and prediction be shown or captured and suppressed? Default is TRUE.

Value

invisible(NULL) .

See Also

Other configure: getMlrOptions

---

createDummyFeatures *Generate dummy variables for factor features.*

Description

Replace all factor features with their dummy variables. Internally model.matrix is used. Non factor features will be left untouched and passed to the result.

Usage

```
createDummyFeatures(obj, target = character(0L), method = "1-of-n",
  exclude = character(0L))
```

Arguments

obj
: [data.frame | Task]
Input data.

target
: [character()]
Name of the column(s) specifying the response. Only used when obj is a data.frame, otherwise ignored.

`method` [character(1)]
Available are:
"1-of-n": For n factor levels there will be n dummy variables.
"reference": There will be n-1 dummy variables leaving out the first factor level of each variable.

`exclude` [character]
Names of the columns to exclude. The target does not have to be included here. Default is none.

Value

`data.frame` | `Task` . Same type as `obj`.

See Also

`model.matrix`

`crossover` *crossover*

Description

Takes two bit strings and creates a new one of the same size by selecting the items from the first string or the second, based on a given rate (the probability of choosing an element from the first string).

Arguments

`x` [logical]
First parent string.

`y` [logical]
Second parent string.

`rate` [numeric(1)]
A number representing the probability of selecting an element of the first string. Default is `0.5`.

Value

`crossover` .

crossval *Fit models according to a resampling strategy.*

Description

resample: Given a resampling strategy, which defines sets of training and test indices, fits the selected learner using the training sets and performs predictions for the training/test sets. This depends on what you selected in the resampling strategy, see parameter predict in makeResampleDesc.

Then performance measures are calculated on all respective data sets and aggregated.

You are able to return all fitted models (parameter models) or extract specific parts of the models (parameter extract) as returning all of them completely might be memory intensive.

For construction of the resampling strategies use the factory methods makeResampleDesc and makeResampleInstance.

The remaining functions on this page are convenience wrappers for the various existing resampling strategies.

Usage

```
crossval(learner, task, iters = 10L, stratify = FALSE, measures,
  models = FALSE, ..., show.info = TRUE)

repcv(learner, task, folds = 10L, reps = 10L, stratify = FALSE, measures,
  models = FALSE, ..., show.info = TRUE)

holdout(learner, task, split = 2/3, stratify = FALSE, measures,
  models = FALSE, ..., show.info = TRUE)

subsample(learner, task, iters = 30, split = 2/3, stratify = FALSE,
  measures, models = FALSE, ..., show.info = TRUE)

bootstrapOOB(learner, task, iters = 30, stratify = FALSE, measures,
  models = FALSE, ..., show.info = TRUE)

bootstrapB632(learner, task, iters = 30, stratify = FALSE, measures,
  models = FALSE, ..., show.info = TRUE)

bootstrapB632plus(learner, task, iters = 30, stratify = FALSE, measures,
  models = FALSE, ..., show.info = TRUE)

resample(learner, task, resampling, measures, weights = NULL,
  models = FALSE, extract, show.info = getMlrOption("show.info"))
```

Arguments

learner [Learner | character(1)]
The learner. If you pass a string the learner will be created via makeLearner.

task [Task]
The task.

resampling [ResampleDesc or ResampleInstance]
Resampling strategy. If a description is passed, it is instantiated automatically.

iters [integer(1)]
See ResampleDesc.

folds [integer(1)]
See ResampleDesc.

reps [integer(1)]
See ResampleDesc.

split [numeric(1)]
See ResampleDesc.

stratify [logical(1)]
See ResampleDesc.

measures [Measure | list of Measure]
Performance measure(s) to evaluate.

weights [numeric]
Optional, non-negative case weight vector to be used during fitting. If given, must be of same length as observations in task and in corresponding order. Overwrites weights specified in the task. By default NULL which means no weights are used unless specified in the task.

models [logical(1)]
Should all fitted models be returned? Default is FALSE.

extract [function]
Function used to extract information from a fitted model during resampling. Is applied to every WrappedModel resulting from calls to train during resampling. Default is to extract nothing.

... [any]
Further hyperparameters passed to learner.

show.info [logical(1)]
Print verbose output on console? Default is set via configureMlr.

Value

List of:

measures.test [data.frame]
Gives you access to performance measurements on the individual test sets. Rows correspond to sets in resampling iterations, columns to performance measures.

measures.train [data.frame]
Gives you access to performance measurements on the individual training sets. Rows correspond to sets in resampling iterations, columns to performance measures. Usually not available, only if specifically requested, see general description above.

aggr [numeric] Named vector of aggregated performance values. Names are coded like this <measure>.<aggregation>.

`err.msgs [data.frame]`
: Number of rows equals resampling iterations and columns are: "iter", "train", "predict". Stores error messages generated during train or predict, if these were caught via `configureMlr`.

`pred [ResamplePrediction]`
: Container for all predictions during resampling.

`models [list of WrappedModel]`
: List of fitted models or `NULL`.

`extract [list]`
: List of extracted parts from fitted models or `NULL`.

See Also

Other resample: `ResampleDesc`, `makeResampleDesc`; `ResampleInstance`, `makeResampleInstance`; `ResamplePrediction`

Examples

```
task = makeClassifTask(data = iris, target = "Species")
rdesc = makeResampleDesc("CV", iters = 2)
r = resample(makeLearner("classif.qda"), task, rdesc)
print(r$aggr)
print(r$measures.test)
print(r$pred)
```

---

downsample *Downsample (subsample) a task or a data.frame.*

Description

Decrease the observations in a `task` or a `ResampleInstance` to a given percentage of observations.

Usage

```
downsample(obj, perc = 1, stratify = FALSE)
```

Arguments

`obj`
: [`Task` | `ResampleInstance`]
Input data or a `ResampleInstance`.

`perc`
: [`numeric(1)`]
Percentage from [0, 1]. Default is 1.

`stratify`
: [`logical(1)`]
Only for classification: Should the downsampled data be stratified according to the target classes? Default is `FALSE`.

Value

`data.frame` | `Task` | `ResampleInstance` . Same type as `obj`.

See Also

makeResampleInstance

Other downsample: makeDownsampleWrapper

---

dropFeatures *Drop some features of task.*

Description

Drop some features of task.

Usage

```
dropFeatures(task, features)
```

Arguments

task	[Task] The task.
features	[character] Features to drop.

Value

Task .

---

estimateResidualVariance *Estimate the residual variance*

Description

Estimate the residual variance of a regression model on a given task. If a regression learner is provided instead of a model, the model is trained (see train) first.

Usage

```
estimateResidualVariance(x, task, data, target)
```

Arguments

x	[Learner or WrappedModel] Learner or wrapped model.
task	[RegrTask] Regression task. If missing, data and target must be supplied.
data	[data.frame] A data frame containing the features and target variable. If missing, task must be supplied.
target	[character(1)] Name of the target variable. If missing, task must be supplied.

FailureModel *Failure model.*

Description

A subclass of WrappedModel. It is created - if you set the respective option in configureMlr - when a model internally crashed during training. The model always predicts NAs.

Its encapsulated learner.model is simply a string: The error message that was generated when the model crashed. The following code shows how to access the message.

Examples

```
configureMlr(on.learner.error = "warn")
data = iris
data$newfeat = 1 # will make LDA crash
task = makeClassifTask(data = data, target = "Species")
m = train("classif.lda", task) # LDA crashed, but mlr catches this
print(m)
print(m$learner.model) # the error message
p = predict(m, task) # this will predict NAs
print(p)
print(performance(p))
configureMlr(on.learner.error = "stop")
```

FeatSelResult *Result of feature selection.*

Description

Container for results of feature selection. Contains the obtained features, their performance values and the optimization path which lead there.
You can visualize it using analyzeFeatSelResult.

Details

Object members:

learner [Learner] Learner that was optimized.

control [FeatSelControl] Control object from feature selection.

x [character] Vector of feature names identified as optimal.

y [numeric] Performance values for optimal x.

opt.path [OptPath] Optimization path which lead to x.

---

filterFeatures *Filter features by thresholding filter values.*

Description

First, calls getFilterValues. Features are then selected via select and val.

Usage

```
filterFeatures(task, method = "random.forest.importance", select = "perc",
  val, ...)
```

Arguments

task	[Task] The task.
method	[character(1)] See getFilterValues. Default is "random.forest.importance".
select	[character(1)] How to select top-scoring features. "perc" = select top-scoring percentage, "abs" = select absolute number of top-scoring features, "threshold" = select all features whose criterion value is >= val. Default is "perc".
val	[numeric(1)] Depends on select: Either a percentage from [0, 1], a number of features or a threshold value for the criterion.
...	[any] Passed down to selected method.

Value

Task .

See Also

Other filter: FilterResult, getFilterResult; FilterValues; getFilterValues; getFilteredFeatures; listFilterMethods; makeFilterWrapper

FilterValues *Result of* getFilterValues.

Description

- task.desc [TaskDesc]Task description.
- method [character]Filter method.
- data [data.frame]Has columns: name = Names of features; val = Feature importance values; type = Feature column type.

See Also

Other filter: FilterResult, getFilterResult; filterFeatures; getFilterValues; getFilteredFeatures; listFilterMethods; makeFilterWrapper

getAggrPerformances *Extract the aggregated measures of a benchmark result.*

Description

Extract the aggregated measures of a benchmark result.

Usage

```
getAggrPerformances(object)
```

Arguments

object	[BenchmarkResult] Benchmark result.

Value

data.frame .

See Also

Other benchmark: BenchmarkResult; FilterResult, getFilterResult; benchmark; getFeatSelResult; getPerformances; getPredictions; getTuneResult

getBaggingModels *Returns the list of models fitted in bagging.*

Description

Returns the list of models fitted in bagging.

Usage

```
getBaggingModels(model, learner.models = FALSE)
```

Arguments

model	[WrappedModel] Model produced by training a bagging learner.
learner.models	[logical(1)] Return underlying R models (e.g., rpart models) or wrapped mlr models (WrappedModel). Default is FALSE.

Value

list .

getConfMatrix *Confusion matrix.*

Description

Calculates confusion matrix for (possibly resampled) prediction. Rows indicate true classes, columns predicted classes.

Usage

```
getConfMatrix(pred, relative = FALSE)
```

Arguments

pred	[Prediction] Prediction object.
relative	[logical(1)] If TRUE rows are normalized to show relative frequencies. Default is FALSE.

Details

Code inspired by errormatrix.

Value

matrix . A confusion matrix.

See Also

predict.WrappedModel

Examples

```
## create classification task and use linear discriminant analysis for classification
task = makeClassifTask(data = iris, target = "Species")
lrn = makeLearner("classif.lda")

## set up training and test data
n = nrow(iris)
mixed.set = sample(1:n)
training.set = mixed.set[1:(n/2)]
test.set = mixed.set[(n/2 + 1):n]

## train model
mod = train(lrn, task, subset = training.set)

## get predictions and show calculate confusion matrix
pred = predict(mod, newdata = iris[test.set, ])
print(getConfMatrix(pred))
print(getConfMatrix(pred, relative = TRUE))
```

---

getCostSensClassifModel

Returns the underlying classification model.

Description

Returns the underlying classification model.

Usage

```
getCostSensClassifModel(model, learner.model = TRUE)
```

Arguments

model	[WrappedModel] Model produced by training a cost-sensitive classification learner.
learner.model	[logical(1)] Return underlying R model or wrapped mlr model (WrappedModel). Default is FALSE.

Value

list .

getCostSensRegrModels *Returns the list of fitted models.*

Description

Returns the list of fitted models.

Usage

```
getCostSensRegrModels(model, learner.models = FALSE)
```

Arguments

model	[WrappedModel] Model produced by training a cost-sensitive regression learner.
learner.models	[logical(1)] Return underlying R models or wrapped mlr models (WrappedModel). Default is FALSE.

Value

list .

getCostSensWeightedPairsModels *Returns the list of fitted models.*

Description

Returns the list of fitted models.

Usage

```
getCostSensWeightedPairsModels(model, learner.models = FALSE)
```

Arguments

model	[WrappedModel] Model produced by training a cost-sensitive regression learner.
learner.models	[logical(1)] Return underlying R models or wrapped mlr models (WrappedModel). Default is FALSE.

Value

list .

getFailureModelMsg *Return error message of FailureModel.*

Description

Such a model is created when one sets the corresponding option in `configureMlr`. If no failure occurred, `NA` is returned.

For complex wrappers this getter returns the first error essage encountered in ANY model that failed.

Usage

```
getFailureModelMsg(model)
```

Arguments

`model`	[`WrappedModel`] The model.

Value

`character(1)` .

getFeatSelResult *Returns the selected feature set and optimization path after training or benchmarking.*

Description

Returns the selected feature set and optimization path after training or benchmarking.

Usage

```
getFeatSelResult(object)
```

Arguments

`object`	[`WrappedModel`	`BenchmarkResult`] Trained Model created with `makeFeatSelWrapper` or benchmark result created with `benchmark`.

Value

`FeatSelResult` or list of `FeatSelResult`s . `NULL`, if no feature selection was performed.

See Also

Other benchmark: BenchmarkResult; FilterResult, getFilterResult; benchmark; getAggrPerformances; getPerformances; getPredictions; getTuneResult

Other featsel: FeatSelControl, FeatSelControlExhaustive, FeatSelControlGA, FeatSelControlRandom, FeatSelControlSequential, makeFeatSelControlExhaustive, makeFeatSelControlGA, makeFeatSelControlRandom, makeFeatSelControlSequential; analyzeFeatSelResult; makeFeatSelWrapper; selectFeatures

---

getFilteredFeatures *Returns the filtered features.*

Description

Returns the filtered features.

Usage

```
getFilteredFeatures(model)
```

Arguments

model	[WrappedModel] Trained Model created with makeFilterWrapper.

Value

character .

See Also

Other filter: FilterResult, getFilterResult; FilterValues; filterFeatures; getFilterValues; listFilterMethods; makeFilterWrapper

---

getFilterResult *Returns a filter result after training or benchmarking.*

Description

Returns a filter result after training or benchmarking.

Usage

```
getFilterResult(object)
```

Arguments

object	[WrappedModel	BenchmarkResult] Trained Model created with makeFilterWrapper or benchmark result created with benchmark.

Value

FilterResult or list of FilterResults .

See Also

Other benchmark: BenchmarkResult; benchmark; getAggrPerformances; getFeatSelResult; getPerformances; getPredictions; getTuneResult

Other filter: FilterValues; filterFeatures; getFilterValues; getFilteredFeatures; listFilterMethods; makeFilterWrapper

---

getFilterValues *Calculates feature filter values.*

Description

Calculates numerical filter values for all features. Look at package FSelector for details on the filter algorithms.

Currently only supports classification (C) and regression (R). Allowed feature types are abbreviated in table as numerics (N) and factors (F).

Available methods are:

Method	Tasks	Feats	Description
linear.correlation	R	N	Pearson's correlation between feature and target
rank.correlation	R	N	Spearman's correlation between feature and target
information.gain	C,R	N,F	Entropy-based information gain between feature and target
gain.ratio	C,R	N,F	Entropy-based gain ratio between feature and target
symmetrical.uncertainty	C,R	N,F	Entropy-based symmetrical uncertainty between feature and target
chi.squared	C,R	N,F	Chi-squared statistic of independence between feature and target
random.forest.importance	C,R	N,F	See importance
relief	C,R	N,F	RELIEF algorithm
oneR	C,R	N,F	OneR assocation rule
mRMR.classic	R	N	MRMR algorithm, see mRMR.classic from mRMRe package

Usage

```
getFilterValues(task, method = "random.forest.importance", ...)
```

Arguments

task	[Task] The task.
method	[character(1)] Filter method, see above. Default is "random.forest.importance".
...	[any] Passed down to selected method.

Value

FilterValues .

See Also

Other filter: FilterResult, getFilterResult; FilterValues; filterFeatures; getFilteredFeatures; listFilterMethods; makeFilterWrapper

---

getHyperPars *Get current parameter settings for a learner.*

Description

Get current parameter settings for a learner.

Usage

```
getHyperPars(learner, for.fun = c("train", "predict", "both"))
```

Arguments

learner	[Learner | character(1)] The learner. If you pass a string the learner will be created via makeLearner.
for.fun	[character(1)] Restrict the returned settings to hyperparameters corresponding to when the are used (see LearnerParam). Must be a subset of: "train", "predict" or "both". Default is c("train", "predict", "both").

Value

list . A named list of values.

See Also

Other learner: getParamSet; removeHyperPars; setHyperPars; setId; setPredictType; showHyperPars

getLearnerModel *Get underlying R model of learner integrated into mlr.*

Description

Get underlying R model of learner integrated into mlr.

Usage

```
getLearnerModel(model)
```

Arguments

model	[WrappedModel] The model, returned by e.g., train.

Value

any . A fitted model, depending the learner / wrapped package. E.g., a model of class rpart for learner "classif.rpart".

getMlrOptions *Returns a list of mlr's options*

Description

Returns a list of mlr's options

Usage

```
getMlrOptions()
```

Value

list .

See Also

Other configure: configureMlr

getParamSet *Get a description of all possible parameter settings for a learner.*

Description

Get a description of all possible parameter settings for a learner.

Usage

```
getParamSet(learner)
```

Arguments

learner	[Learner	character(1)] The learner. If you pass a string the learner will be created via makeLearner.

Value

ParamSet .

See Also

Other learner: getHyperPars; removeHyperPars; setHyperPars; setId; setPredictType; showHyperPars

getPerformances *Extract performance measures of bechmark result.*

Description

Extract performance measures of bechmark result.

Usage

```
getPerformances(object)
```

Arguments

object	[BenchmarkResult] Benchmark result.

Value

data.frame .

See Also

Other benchmark: BenchmarkResult; FilterResult, getFilterResult; benchmark; getAggrPerformances; getFeatSelResult; getPredictions; getTuneResult

getPredictions *Extract the predictions from a benchmark result.*

Description

Extract the predictions from a benchmark result.

Usage

```
getPredictions(object)
```

Arguments

object	[BenchmarkResult] Benchmark result.

Value

data.frame .

See Also

Other benchmark: BenchmarkResult; FilterResult, getFilterResult; benchmark; getAggrPerformances; getFeatSelResult; getPerformances; getTuneResult

getProbabilities *Get probabilities for some classes.*

Description

Get probabilities for some classes.

Usage

```
getProbabilities(pred, cl)
```

Arguments

pred	[Prediction] Prediction object.
cl	[character] Names of classes. Default is either all classes for multi-class problems or the positive class for binary classification.

Value

data.frame with numerical columns or a numerical vector if length of cl is 1. Order of columns is defined by cl.

See Also

Other predict: predict.WrappedModel; setPredictType

Examples

```
task = makeClassifTask(data = iris, target = "Species")
lrn = makeLearner("classif.lda", predict.type = "prob")
mod = train(lrn, task)
# predict probabilities
pred = predict(mod, newdata = iris)

# Get probabilities for all classes
head(getProbabilities(pred))

# Get probabilities for a subset of classes
head(getProbabilities(pred, c("setosa", "virginica")))
```

---

getTaskCosts *Extract costs in task.*

Description

Retuns "NULL" if the task is not of type "costsens".

Usage

```
getTaskCosts(task, subset)
```

Arguments

task	[CostSensTask] The task.
subset	[integer] Selected cases. Default is all cases.

Value

matrix | NULL .

See Also

Other task: getTaskData; getTaskFeatureNames; getTaskFormula, getTaskFormulaAsString; getTaskNFeats; getTaskTargets; subsetTask

getTaskData *Extract data in task.*

Description

Useful in trainLearner when you add a learning machine to the package.

Usage

```
getTaskData(task, subset, features, target.extra = FALSE,
  recode.target = "no")
```

Arguments

task	[Task] The task.
subset	[integer] Selected cases. Default is all cases.
features	[character] Selected features. Default is all.
target.extra	[logical(1)] Should target vector be returned separately? If not, a single data.frame including the target is returned, otherwise a list with the input data.frame and an extra vector for the targets. Default is FALSE.
recode.target	[character(1)] Should target classes be recoded? Only for binary classification. Possible are "no" (do nothing), "01", and "-1+1". In the two latter cases the target vector is converted into a numeric vector. The positive class is coded as +1 and the negative class either as 0 or -1. Default is "no".

Value

Either a data.frame or a list with data.frame data and vector target.

See Also

Other task: getTaskCosts; getTaskFeatureNames; getTaskFormula, getTaskFormulaAsString; getTaskNFeats; getTaskTargets; subsetTask

Examples

```
library("mlbench")
data(BreastCancer)

df = BreastCancer
df$Id = NULL
task = makeClassifTask(id = "BreastCancer", data = df, target = "Class", positive = "malignant")
```

```
head(getTaskData)
head(getTaskData(task, features = c("Cell.size", "Cell.shape"), recode.target = "-1+1"))
head(getTaskData(task, subset = 1:100, recode.target = "01"))
```

`getTaskFeatureNames` *Get feature names of task.*

Description

Target column name is not included.

Usage

```
getTaskFeatureNames(task)
```

Arguments

`task`	[Task] The task.

Value

`character .`

See Also

Other task: `getTaskCosts`; `getTaskData`; `getTaskFormula`, `getTaskFormulaAsString`; `getTaskNFeats`; `getTaskTargets`; `subsetTask`

`getTaskFormulaAsString`
Get formula of a task.

Description

This is simply "<target> ~ .".

Usage

```
getTaskFormulaAsString(x, target = getTargetNames(x))

getTaskFormula(x, target = getTargetNames(x), env = NULL)
```

Arguments

x	[Task	TaskDesc] Task or its description object.
target	[character(1)] Left hand side of formula. Default is defined by task x.
env	[environment] Environment of the formula. Set this to parent.frame() for the default behaviour. Default is NULL which deletes the environment.

Value

formula | character(1) .

See Also

Other task: getTaskCosts; getTaskData; getTaskFeatureNames; getTaskNFeats; getTaskTargets; subsetTask

---

getTaskNFeats *Get number of feature in task.*

Description

Get number of feature in task.

Usage

```
getTaskNFeats(task)
```

Arguments

task	[Task] The task.

Value

integer(1) .

See Also

Other task: getTaskCosts; getTaskData; getTaskFeatureNames; getTaskFormula, getTaskFormulaAsString; getTaskTargets; subsetTask

getTaskTargets *Get target column of task.*

Description

Get target column of task.

Usage

```
getTaskTargets(task, subset, recode.target = "no")
```

Arguments

task	[Task] The task.
subset	[integer] Selected cases. Default is all cases.
recode.target	[character(1)] Should target classes be recoded? Only for binary classification. Possible are "no" (do nothing), "01", and "-1+1". In the two latter cases the target vector is converted into a numeric vector. The positive class is coded as +1 and the negative class either as 0 or -1. Default is "no".

Value

A factor for classification or a numeric for regression.

See Also

Other task: getTaskCosts; getTaskData; getTaskFeatureNames; getTaskFormula, getTaskFormulaAsString; getTaskNFeats; subsetTask

Examples

```
task = makeClassifTask(data = iris, target = "Species")
getTaskTargets(task)
getTaskTargets(task, subset = 1:50)
```

getTuneResult *Returns the optimal hyperparameters and optimization path after training or benchmarking.*

Description

Returns the optimal hyperparameters and optimization path after training or benchmarking.

Usage

```
getTuneResult(object)
```

Arguments

object	[WrappedModel | BenchmarkResult] Trained Model created with makeTuneWrapper or benchmark result created with benchmark.

Value

TuneResult or list of TuneResults . NULL, if no tuning was performed.

See Also

Other benchmark: BenchmarkResult; FilterResult, getFilterResult; benchmark; getAggrPerformances; getFeatSelResult; getPerformances; getPredictions

Other tune: ModelMultiplexer, makeModelMultiplexer; TuneControl, TuneControlCMAES, TuneControlGenSA, TuneControlGrid, TuneControlIrace, TuneControlRandom, makeTuneControlCMAES, makeTuneControlGenSA, makeTuneControlGrid, makeTuneControlIrace, makeTuneControlRandom; makeModelMultiplexerParamSet; makeTuneWrapper; tuneParams; tuneThreshold

`imputations` *Built in imputation methods The built-ins are:*

- `imputeConstant(const)` *for imputation using a constant value,*
- `imputeMedian()` *for imputation using the median,*
- `imputeMode()` *for imputation using the mode,*
- `imputeMin(multiplier)` *for imputation using the minimum,*
- `imputeMax(multiplier)` *for imputation using the maximum,*
- `imputeNormal(mean, sd)` *for imputation using normally distributed random values. Mean and standard deviation will be calculated from the data if not provided.*
- `imputeHist(breaks, use.mids)` *for imputation using random values with probabilities calculated using* `table` *or* `hist`.
- `imputeLearner(learner, preimpute)` *for imputations using the response of a classification or regression learner.*

Description

Built in imputation methods The built-ins are:

- `imputeConstant(const)` for imputation using a constant value,
- `imputeMedian()` for imputation using the median,
- `imputeMode()` for imputation using the mode,
- `imputeMin(multiplier)` for imputation using the minimum,
- `imputeMax(multiplier)` for imputation using the maximum,
- `imputeNormal(mean, sd)` for imputation using normally distributed random values. Mean and standard deviation will be calculated from the data if not provided.
- `imputeHist(breaks, use.mids)` for imputation using random values with probabilities calculated using `table` or `hist`.
- `imputeLearner(learner, preimpute)` for imputations using the response of a classification or regression learner.

Usage

```
imputeConstant(const)

imputeMedian()

imputeMean()

imputeMode()

imputeMin(multiplier = 1)
```

```
imputeMax(multiplier = 1)

imputeUniform(min = NA_real_, max = NA_real_)

imputeNormal(mu = NA_real_, sd = NA_real_)

imputeHist(breaks, use.mids = TRUE)

imputeLearner(learner, features = NULL)
```

Arguments

const	[any] Constant valued use for imputation.
multiplier	[numeric(1)] Value that stored minimum or maximum is multiplied with when imputation is done.
min	[numeric(1)] Lower bound for uniform distribution. If NA (default), it will be estimated from the data.
max	[numeric(1)] Upper bound for uniform distribution. If NA (default), it will be estimated from the data.
mu	[numeric(1)] Mean of normal distribution. If missing it will be estimated from the data.
sd	[numeric(1)] Standard deviation of normal distribution. If missing it will be estimated from the data.
breaks	[numeric(1)] Number of breaks to use in hist. If missing, defaults to auto-detection via "Sturges".
use.mids	[logical(1)] If x is numeric and a histogram is used, impute with bin mids (default) or instead draw uniformly distributed samples within bin range.
learner	[Learner] Supervised learner. Its predictions will be used for imputations. Note that the target column is not available for this operation.
features	[character] Features to use in learner for prediction. Default is NULL which uses all available features except the target column of the original task.

See Also

Other impute: impute; makeImputeMethod; makeImputeWrapper; reimpute

impute *Impute and re-impute data*

Description

Allows imputation of missing feature values through various techniques. Note that you have the possibility to re-impute a data set in the same way as the imputation was performed during training. This especially comes in handy during resampling when one wants to perform the same imputation on the test set as on the training set.

The function `impute` performs the imputation on a data set and returns, alongside with the imputed data set, an "ImputationDesc" object which can contain "learned" coefficients and helpful data. It can then be passed together with a new data set to `reimpute`.

The imputation techniques can be specified for certain features or for feature classes, see function arguments.

You can either provide an arbitrary object, use a built-in imputation method listed under `imputations` or create one yourself using `makeImputeMethod`.

Usage

```
impute(data, target, classes = list(), cols = list(),
  dummy.cols = character(0L), dummy.type = "factor",
  impute.new.levels = TRUE, recode.factor.levels = TRUE)
```

Arguments

`data` [`data.frame`]
Input data.

`target` [`character`]
Name of the column(s) specifying the response.

`classes` [`named list`]
Named list containing imputation techniques for classes of columns. E.g. `list(numeric = imputeMedia`

`cols` [`named list`]
Named list containing names of imputation methods to impute missing values in the data column referenced by the list element's name. Overwrites imputation set via `classes`.

`dummy.cols` [`character`]
Column names to create dummy columns (containing binary missing indicator) for. Default is `character(0)`.

`dummy.type` [`character(1)`]
How dummy columns are encoded. Either as 0/1 with type "numeric" or as "factor". Default is "factor".

`impute.new.levels`
[`logical(1)`]
If new, unencountered factor level occur during reimputation, should these be handled as NAs and then be imputed the same way? Default is `TRUE`.

recode.factor.levels
: [logical(1)]
Recode factor levels after reimputation, so they match the respective element of lvls (in the description object) and therefore match the levels of the feature factor in the training data after imputation?. Default is TRUE.

Details

The description object contains these slots

target [character] See argument.

features [character] Feature names, these are the column names of data, excluding target.

lvls [named list] Mapping of column names of factor features to their levels, including newly created ones during imputation.

impute [named list] Mapping of column names to imputation functions.

dummies [named list] Mapping of column names to imputation functions.

impute.new.levels [logical(1)] See argument.

recode.factor.levels [logical(1)] See argument.

Value

data [data.frame]
list
: Imputed data.

desc [ImputationDesc]
: Description object.

See Also

Other impute: imputations, imputeConstant, imputeHist, imputeLearner, imputeMax, imputeMean, imputeMedian, imputeMin, imputeMode, imputeNormal, imputeUniform; makeImputeMethod; makeImputeWrapper; reimpute

Examples

```
df = data.frame(x = c(1, 1, NA), y = factor(c("a", "a", "b")), z = 1:3)
imputed = impute(df, target = character(0), cols = list(x = 99, y = imputeMode()))
print(imputed$data)
reimpute(data.frame(x = NA), imputed$desc)
```

isFailureModel *Is the model a FailureModel?*

Description

Such a model is created when one sets the corresponding option in `configureMlr`.

For complex wrappers this getter returns `TRUE` if ANY model contained in it failed.

Usage

```
isFailureModel(model)
```

Arguments

`model`	[`WrappedModel`] The model.

Value

`logical(1)` .

learnerArgsToControl *Convert arguments to control structure.*

Description

Find all elements in `...` which are not missing and call `control` on them.

Usage

```
learnerArgsToControl(control, ...)
```

Arguments

`control`	[`function`] Function that creates control structure.
`...`	[any] Arguments for control structure function.

Value

Control structure for learner.

LearnerProperties *Set, add, remove or query properties of learners*

Description

Set, add, remove or query properties of learners

Usage

```
setProperties(learner, props)

addProperties(learner, props)

removeProperties(learner, props)

hasProperties(learner, props)
```

Arguments

learner	[Learner	character(1)] The learner. If you pass a string the learner will be created via makeLearner.
props	[character] Vector of properties to set, add, remove or query.

Value

setProperties, addProperties and removeProperties return an updated Learner. hasProperties returns a logical vector of the same length of props.

learners *List of supported learning algorithms.*

Description

- **classif.ada**
  Boosting from ada package: ada
- **classif.boosting**
  Boosting from adabag package: boosting
  Note that xval has been set to 0 by default for speed.
- **classif.blackboost**
  Gradient boosting with regression trees from mboost package: blackboost
- **classif.cforest**
  Random forest based on conditional inference trees from party package: cforest

- **classif.ctree**
  Conditional Inference Trees from party package: `ctree`
- **classif.fnn**
  Fast k-Nearest Neighbor from FNN package: `knn`
- **classif.gbm**
  Gradient boosting machine from gbm package: `gbm`
- **classif.glmnet**
  GLM with lasso or elasticnet regularization from glmnet package: `glmnet`
- **classif.geoDA**
  Geometric Predictive Discriminant Analysis from DiscriMiner package: `geoDA`
- **classif.glmboost**
  Boosting for GLMs from mboost package: `glmboost`
  Note that `family` has been set to `Binomial()` by default.
- **classif.IBk**
  K-nearest neighbours from RWeka package: `IBk`
- **classif.J48**
  J48 Decision Trees from RWeka package: `J48` Note that NAs are directly passed to WEKA with `na.action = na.pass`.
- **classif.JRip**
  Propositional Rule Learner from RWeka package: `JRip` Note that NAs are directly passed to WEKA with `na.action = na.pass`.
- **classif.kknn**
  k-Nearest Neighbor from kknn package: `kknn`
- **classif.knn**
  k-Nearest Neighbor from class package: `knn`
- **classif.ksvm**
  Support Vector Machines from kernlab package: `ksvm`
  Note that kernel parameters have to be passed directly and not by using the kpar list in ksvm.
  Note that `fit` has been set to `FALSE` by default for speed.
- **classif.lda**
  Linear Discriminant Analysis from MASS package: `lda`
- **classif.LiblineaRBinary**
  Regularized Binary Linear Predictive Models Estimation from LiblineaR package: `LiblineaR`
  Note that this model subsumes the types 1,2,3,5
- **classif.LiblineaRLogReg**
  Regularized Logistic Regression from LiblineaR package: `LiblineaR` Note that this model subsumes type 0,6,7.
- **classif.LiblineaRMultiClass**
  Multi-class Support Vector Classification by Crammer and Singer from LiblineaR package: `LiblineaR` Note that this model is type 4.
- **classif.linDA**
  Linear Discriminant Analysis from DiscriMiner package: `linDA`
- **classif.logreg**
  Logistic Regression from stats package: `glm`

- **classif.lssvm**
  Least Squares Support Vector Machine from kernlab package: lssvm
  Note that fitted has been set to FALSE by default for speed.
- **classif.lvq1**
  Learning Vector Quantization from class package: lvq1
- **classif.mda**
  Mixture Discriminant Analysis from mda package: mda
  Note that keep.fitted has been set to FALSE by default for speed.
- **classif.multinom**
  Multinomial Regression from nnet package: multinom
- **classif.naiveBayes**
  Naive Bayes from e1071 package: naiveBayes
- **classif.nnet**
  Neural Network from nnet package: nnet
  Note that size has been set to 3 by default.
- **classif.OneR**
  1-R classifier from RWeka package: OneR
  Note that NAs are directly passed to WEKA with na.action = na.passi.
- **classif.PART**
  PART decision lists from RWeka package: PART
  Note that NAs are directly passed to WEKA with na.action = na.pass.
- **classif.plr**
  Logistic regression with a L2 penalty from stepPlr package: plr
  Note that AIC and BIC penalty types can be selected via the new parameter cp.type.
- **classif.plsDA**
  Partial Least Squares (PLS) Discriminant Analysis from DiscriMiner package: plsDA
- **classif.plsdaCaret**
  Partial Least Squares (PLS) Discriminant Analysis from caret package: plsda
- **classif.qda**
  Quadratic Discriminant Analysis from MASS package: qda
- **classif.quaDA**
  Quadratic Discriminant Analysis from DiscriMiner package: quaDA
- **classif.randomForest**
  Random Forest from randomForest package: randomForest.
  The argument fix.factors restores the factor levels seen in the training data before prediction to circumvent randomForest's internal sanity checks. Default is FALSE.
- **classif.rda**
  Regularized Discriminant Analysis from klaR package: rda
  Note that estimate.error has been set to FALSE by default for speed.
- **classif.rpart**
  Decision Tree from rpart package: rpart
  Note that xval has been set to 0 by default for speed.
- **classif.svm**
  Support Vector Machines (libsvm) from e1071 package: svm

- **regr.blackboost**
  Gradient boosting with regression trees from mboost package: `blackboost`
- **regr.cforest**
  Random forest based on conditional inference trees from party package: `cforest`
- **regr.crs**
  Regression Splines from crs package: `crs`
- **regr.earth**
  Multivariate Adaptive Regression Splines from earth package: `earth`
- **regr.fnn**
  Fast k-Nearest Neighbor from FNN package: `knn`
- **regr.gbm**
  Gradient boosting machine from gbm package: `gbm`
  Note that `distribution` has been set to "gaussian" by default.
- **regr.glmnet**
  GLM with lasso or elasticnet regularization from glmnet package: `glmnet`
- **regr.IBk**
  K-nearest neighbours from RWeka package: `IBk`
- **regr.kknn**
  K-Nearest-Neighbor regression from kknn package: `kknn`
- **regr.km**
  Kriging from DiceKriging package: `km`
- **regr.ksvm**
  Support Vector Machines from kernlab package: `ksvm`
  Note that kernel parameters have to be passed directly and not by using the kpar list in ksvm.
  Note that `fit` has been set to `FALSE` by default for speed.
- **regr.penalized.lasso**
  Lasso regression from penalized package: `penalized`
- **regr.lm**
  Simple linear regression from stats package: `lm`
- **regr.mars**
  Multivariate Adaptive Regression Splines from mda package: `mars`
- **regr.mob**
  Model-based recursive partitioning yielding a tree with fitted models associated with each terminal node from party package: `mob`
- **regr.nnet**
  Neural Network from nnet package: `nnet`
  Note that `size` has been set to 3 by default.
- **regr.pcr**
  Principal component regression from pls package: `pcr`
  Note that `model` has been set to `FALSE` by default for speed.
- **regr.randomForest**
  Random Forest from randomForest package: `randomForest`. The argument `fix.factors` restores the factor levels seen in the training data before prediction to circumvent randomForest's internal sanity checks. Default is `FALSE`.

- **regr.penalized.ridge**
  Ridge regression from penalized package: penalized
- **regr.rpart**
  Decision Tree from rpart package: rpart
  Note that xval has been set to 0 by default for speed.
- **regr.rsm**
  Response surface regression from rsm package: rsm
  Note that you select the order of the regression by using modelfun = "FO" (first order), "TWI" (two-way interactions, this is with 1st oder terms!) and "SO" (full second order).
- **regr.rvm**
  Relevance Vector Machine from package kernlab: rvm
  Note that kernel parameters have to be passed directly and not by using the kpar list in rvm.
  Note that fit has been set to FALSE by default for speed.
- **regr.svm**
  Support Vector Machines (libsvm) from e1071 package: svm
- **surv.cforest**
  Random forest based on conditional inference trees from party package: cforest
- **surv.CoxBoost**
  Cox proportional hazards model with componentwise likelhood based boosting from CoxBoost package: CoxBoost
- **surv.coxph**
  Cox proportional hazard model from survival package: coxph
- **surv.glmnet**
  GLM with regularization from glmnet package: glmnet
- **surv.randomForestSRC**
  Random Forests for Survival from randomForestSRC package: randomForestSRC
- **cluster.SimpleKMeans**
  k-means clustering from RWeka package: SimpleKMeans
- **cluster.EM**
  Expectation-maximization clustering from RWeka package: make_Weka_clusterer
- **cluster.XMeans**
  XMeans (k-means with automatic determination of k) clustering from RWeka package: XMeans
  Note that you might have to install the Weka package: WPM("install-package", "XMeans")

`listFilterMethods` *Returns all available feature filter methods.*

Description

Returns all available feature filter methods.

Usage

```
listFilterMethods()
```

Value

character(1) .

See Also

Other filter: FilterResult, getFilterResult; FilterValues; filterFeatures; getFilterValues; getFilteredFeatures; makeFilterWrapper

---

listLearners *Find matching learning algorithms.*

Description

Returns the class names of learning algorithms which have specific characteristics, e.g. whether they supports missing values, case weights, etc.

Note that the packages of all learners are loaded during the search.

Note that for general cost-sensitive learning, mlr currently supports mainly "wrapper" approaches like CostSensWeightedPairsWrapper, which are not listed, as they are not basic R learning algorithms.

Usage

```
listLearners(obj = NA_character_, properties = character(0L),
  quiet = TRUE, warn.missing.packages = TRUE, create = FALSE)

## Default S3 method:
listLearners(obj, properties = character(0L),
  quiet = TRUE, warn.missing.packages = TRUE, create = FALSE)

## S3 method for class 'character'
listLearners(obj, properties = character(0L),
  quiet = TRUE, warn.missing.packages = TRUE, create = FALSE)

## S3 method for class 'Task'
listLearners(obj, properties = character(0L), quiet = TRUE,
  warn.missing.packages = TRUE, create = FALSE)
```

Arguments

obj	[character(1) | Task] Either a task or the type of the task, in the latter case one of: "classif", "regr", "surv", "costsens", "cluster". Default is NA, matching all types.
properties	[character] Set of required properties to filter for. Default is character(0).
quiet	[logical(1)] Construct learners quietly to check their properties, shows no package startup messages. Turn off if you suspect errors. Default is TRUE.
warn.missing.packages	[logical(1)] If some learner cannot be constructed because its package is missing, should a warning be shown? Default is TRUE.
create	[logical(1)] Instantiate objects (or return strings)? Default is FALSE.

Value

character | list of Learner . Class names of matching learners or instantiated objects.

---

listMeasures *Find matching measures.*

Description

Returns the matching measures which have specific characteristics, e.g. whether they supports classification or regression.

Usage

```
listMeasures(obj, properties = character(0L), create = FALSE)

## Default S3 method:
listMeasures(obj, properties = character(0L),
  create = FALSE)

## S3 method for class 'character'
listMeasures(obj, properties = character(0L),
  create = FALSE)

## S3 method for class 'Task'
listMeasures(obj, properties = character(0L), create = FALSE)
```

Arguments

obj	[character(1) | Task] Either a task or the type of the task, in the latter case one of: "classif", "regr", "surv", "costsens", "cluster". Default is NA, matching all types.
properties	[character] Set of required properties to filter for. See Measure for some standardized properties. Default is character(0).
create	[logical(1)] Instantiate objects (or return strings)? Default is FALSE.

Value

character | list of Measure . Class names of matching measures or instantiated objects.

---

makeAggregation *Specifiy your own aggregation of measures*

Description

This is an adavanced feature of mlr. It gives access to some inner workings so the result might not be compatible with everything!

Usage

```
makeAggregation(id, fun)
```

Arguments

id	[character(1)] Name of the aggregation method. (Preferably the same name as the generated function)
fun	[function] A function with following signature: function(task, perf.test, perf.train, measure, group, pr • **task**: task (Task) object • **perf.test**: numerical vector of performance results on the test data set • **perf.train**. numerical vector of performance results on the train data set • **measure**: Measure object. • **group**: grouping vector • **pred**: Prediction object

Value

Aggregation object

See Also

aggregations, setAggregation

Examples

```
# computes the interquartile range on all performance values
test.iqr = makeAggregation(id = "test.iqr",
  fun = function (task, perf.test, perf.train, measure, group, pred) IQR(perf.test))
```

---

makeBaggingWrapper *Fuse learner with the bagging technique.*

Description

Fuses a learner with the bagging method (i.e., similar to what a randomForest does). Creates a learner object, which can be used like any other learner object. Models can easily be accessed via getBaggingModels.

Bagging is implemented as follows: For each iteration a random data subset is sampled (with or without replacement) and potentially the number of features is also restricted to a random subset. Note that this is usually handled in a slightly different way in the random forest where features are sampled at each tree split).

Prediction works as follows: For classification we do majority voting to create a discrete label and probabilities are predicted by considering the proportions of all predicted labels. For regression the mean value and the standard deviations across predictions is computed.

Note that the passed base learner must always have predict.type = 'response', while the BaggingWrapper can estimate probabilities and standard errors, so it can be set, e.g., to predict.type = 'prob'. For this reason, when you call setPredictType, the type is only set for the BaggingWrapper, not passed down to the inner learner.

Usage

```
makeBaggingWrapper(learner, bw.iters = 10L, bw.replace = TRUE, bw.size,
  bw.feats = 1)
```

Arguments

learner	[Learner	character(1)] The learner. If you pass a string the learner will be created via makeLearner.
bw.iters	[integer(1)] Iterations = number of fitted models in bagging. Default is 10.
bw.replace	[logical(1)] Sample bags with replacement (bootstrapping)? Default is TRUE.
bw.size	[numeric(1)] Percentage size of sampled bags. Default is 1 for bootstrapping and 0.632 for subsampling.
bw.feats	[numeric(1)] Percentage size of randomly selected features in bags. Default is 1.

Value

Learner .

---

makeClassifTask *Create a classification, regression, survival, cluster, or cost-sensitive classification task.*

Description

The task encapsulates the data and specifies - through its subclasses - the type of the task. It also contains a description object detailing further aspects of the data.

Useful operators are: getTaskFormula, getTaskFormulaAsString, getTaskFeatureNames, getTaskData, getTaskTargets, subsetTask.

Object members:

env [environment] Environment where data for the task are stored. Use getTaskData in order to access it.

weights [numeric] See argument above. NULL if not present.

blocking [factor] See argument above. NULL if not present.

task.desc [TaskDesc] Encapsulates further information about the task.

Usage

```
makeClassifTask(id, data, target, weights = NULL, blocking = NULL, positive,
  fixup.data = "warn", check.data = TRUE)

makeClusterTask(id, data, weights = NULL, blocking = NULL,
  fixup.data = "warn", check.data = TRUE)

makeCostSensTask(id, data, costs, blocking = NULL, fixup.data = "warn",
  check.data = TRUE)

makeRegrTask(id, data, target, weights = NULL, blocking = NULL,
  fixup.data = "warn", check.data = TRUE)

makeSurvTask(id, data, target, surv.type = "right", weights = NULL,
  blocking = NULL, fixup.data = "warn", check.data = TRUE)
```

Arguments

surv.type	[character(1)] Survival type. Allowed are "right" (default), "left" and "interval2". See Surv for details.
id	[character(1)] Id string for object. Default is the name of R variable passed to data.

data [data.frame]
A data frame containing the features and target variable(s).

target [character(1) | character(2)]
Name of the target variable. For survival analysis these are the names of the survival time and event columns, so it has length 2.

costs [data.frame]
A numeric matrix or data frame containing the costs of misclassification. We assume the general case of observation specific costs. This means we have n rows, corresponding to the observations, in the same order as data. The columns correspond to classes and their names are the class labels (if unnamed we use y1 to yk as labels). Each entry (i,j) of the matrix specifies the cost of predicting class j for observation i.

weights [numeric]
Optional, non-negative case weight vector to be used during fitting. Cannot be set for cost-sensitive learning. Default is NULL which means no (= equal) weights.

blocking [factor]
An optional factor of the same length as the number of observations. Observations with the same blocking level "belong together". Specifically, they are either put all in the training or the test set during a resampling iteration. Default is NULL which means no blocking.

positive [character(1)]
Positive class for binary classification. Default is the first factor level of the target attribute.

fixup.data [character(1)]
Should some basic cleaning up of data be performed? Currently this means removing empty factor levels for the columns. Possible coices are: "no" = Don't do it. "warn" = Do it but warn about it. "quiet" = Do it but keep silent. Default is "warn".

check.data [logical(1)]
Should sanity of data be checked initially at task creation? You should have good reasons to turn this off (one might be speed). Default is TRUE

Value

Task .

See Also

Other costsens: CostSensClassifModel, CostSensClassifWrapper, makeCostSensClassifWrapper; CostSensRegrModel, CostSensRegrWrapper, makeCostSensRegrWrapper; CostSensWeightedPairsModel, CostSensWeightedPairsWrapper, makeCostSensWeightedPairsWrapper

Examples

```
library(mlbench)
data(BostonHousing)
```

```
data(Ionosphere)

makeClassifTask(data = iris, target = "Species")
makeRegrTask(data = BostonHousing, target = "medv")
# an example of a classification task with more than those standard arguments:
blocking = factor(c(rep(1, 51), rep(2, 300)))
makeClassifTask(id = "myIonosphere", data = Ionosphere, target = "Class",
  positive = "good", blocking = blocking)
makeClusterTask(data = iris[, -5L])
```

`makeCostMeasure` *Creates a measure for non-standard misclassification costs.*

Description

Creates a measure for non-standard misclassification costs.

Usage

```
makeCostMeasure(id = "costs", minimize = TRUE, costs, task,
  combine = mean, best = NULL, worst = NULL)
```

Arguments

`id`	[`character(1)`] Name of measure. Default is "costs".
`minimize`	[`logical(1)`] Should the measure be minimized? Otherwise you are effectively specifying a benefits matrix. Default is `TRUE`.
`costs`	[`matrix`] Matrix of misclassification costs. Rows and columns have to be named with class labels, order does not matter. Rows indicate true classes, columns predicted classes.
`task`	[`ClassifTask`] Classification task. Has to be passed, so validity of matrix names can be checked.
`combine`	[`function`] How to combine costs over all cases for a SINGLE test set? Note this is not the same as the `aggregate` argument in `makeMeasure` You can set this as well via `setAggregation`, as for any measure. Default is `mean`.
`best`	[`numeric(1)`] Best obtainable value for measure. Default is `-Inf` or `Inf`, depending on `minimize`.
`worst`	[`numeric(1)`] Worst obtainable value for measure. Default is `Inf` or `-Inf`, depending on `minimize`.

Value

Measure .

See Also

Other performance: G1, G2, acc, auc, bac, ber, cindex, db, dunn, f1, fdr, featperc, fn, fnr, fp, fpr, gmean, gpr, mae, mcc, mcp, meancosts, measures, medae, medse, mmce, mse, multiclass.auc, npv, ppv, rmse, sae, silhouette, sse, timeboth, timepredict, timetrain, tn, tnr, tp, tpr; Measure, makeMeasure; makeCustomResampledMeasure; performance

---

makeCostSensClassifWrapper

Wraps a classification learner for use in cost-sensitive learning.

Description

Creates a wrapper, which can be used like any other learner object. The classification model can easily be accessed via getCostSensClassifModel.

This is a very naive learner, where the costs are transformed into classification labels - the label for each case is the name of class with minimal costs. (If ties occur, the label which is better on average w.r.t. costs over all training data is preferred.) Then the classifier is fitted to that data and subsequently used for prediction.

Usage

```
makeCostSensClassifWrapper(learner)
```

Arguments

learner	[Learner	character(1)] The classification learner. If you pass a string the learner will be created via makeLearner.

Value

Learner .

See Also

Other costsens: ClassifTask, ClusterTask, CostSensTask, RegrTask, SurvTask, Task, makeClassifTask, makeClusterTask, makeCostSensTask, makeRegrTask, makeSurvTask; CostSensRegrModel, CostSensRegrWrapper, makeCostSensRegrWrapper; CostSensWeightedPairsModel, CostSensWeightedPairsWrapper, makeCostSensWeightedPairsWrapper

makeCostSensRegrWrapper

Wraps a regression learner for use in cost-sensitive learning.

Description

Creates a wrapper, which can be used like any other learner object. Models can easily be accessed via getCostSensRegrModels.

For each class in the task, an individual regression model is fitted for the costs of that class. During prediction, the class with the lowest predicted costs is selected.

Usage

```
makeCostSensRegrWrapper(learner)
```

Arguments

learner	[Learner	character(1)] The regression learner. If you pass a string the learner will be created via makeLearner.

Value

Learner .

See Also

Other costsens: ClassifTask, ClusterTask, CostSensTask, RegrTask, SurvTask, Task, makeClassifTask, makeClusterTask, makeCostSensTask, makeRegrTask, makeSurvTask; CostSensClassifModel, CostSensClassifWrapper, makeCostSensClassifWrapper; CostSensWeightedPairsModel, CostSensWeightedPairsW
makeCostSensWeightedPairsWrapper

makeCostSensWeightedPairsWrapper

Wraps a classifier for cost-sensitive learning to produce a weighted pairs model.

Description

Creates a wrapper, which can be used like any other learner object. Models can easily be accessed via getCostSensWeightedPairsModels.

For each pair of labels, we fit a binary classifier. For each observation we define the label to be the element of the pair with minimal costs. During fitting, we also weight the observation with the absolute difference in costs. Prediction is performed by simple voting.

This approach is sometimes called cost-sensitive one-vs-one (CS-OVO), because it is obviously very similar to the one-vs-one approach where one reduces a normal multi-class problem to multiple binary ones and aggregates by voting.

Usage

```
makeCostSensWeightedPairsWrapper(learner)
```

Arguments

learner	[Learner	character(1)] The classification learner. If you pass a string the learner will be created via makeLearner.

Value

Learner .

See Also

Other costsens: ClassifTask, ClusterTask, CostSensTask, RegrTask, SurvTask, Task, makeClassifTask, makeClusterTask, makeCostSensTask, makeRegrTask, makeSurvTask; CostSensClassifModel, CostSensClassifWrapper, makeCostSensClassifWrapper; CostSensRegrModel, CostSensRegrWrapper, makeCostSensRegrWrapper

---

makeCustomResampledMeasure

Construct your own resampled performance measure.

Description

Construct your own performance measure, used after resampling. Note that individual training / test set performance values will be set to NA, you only calculate an aggregated value. If you can define a function that makes sense for every single training / test set, implement your own Measure.

Usage

```
makeCustomResampledMeasure(id, minimize = TRUE, properties = character(0L),
  allowed.pred.types = character(0L), fun, extra.args = list(),
  best = NULL, worst = NULL)
```

Arguments

id	[character(1)] Name of aggregated measure.
fun	[function(task, pred, group, pred, extra.args)] Calculates performance value from ResamplePrediction object. For rare case you can also use the task, the grouping or the extra arguments extra.args.
extra.args	[list] List of extra arguments which will always be passed to fun. Default is empty list.

`minimize` [logical(1)]
Should the measure be minimized? Default is in TRUE.

`properties` [character]
Set of measure properties. Some standard property names include:

classif Is the measure applicable for classification?

classif.multi Is the measure applicable for multi-class classification?

regr Is the measure applicable for regression?

surv Is the measure applicable for survival?

costsens Is the measure applicable for cost-sensitve learning?

Default is character(0).

`allowed.pred.types` [character]
Which prediction types are allowed for this measure? Subset of "response","prob", "se". Default is character(0).

`best` [numeric(1)]
Best obtainable value for measure. Default is -Inf or Inf, depending on minimize.

`worst` [numeric(1)]
Worst obtainable value for measure. Default is Inf or -Inf, depending on minimize.

Value

Measure .

See Also

Other performance: G1, G2, acc, auc, bac, ber, cindex, db, dunn, f1, fdr, featperc, fn, fnr, fp, fpr, gmean, gpr, mae, mcc, mcp, meancosts, measures, medae, medse, mmce, mse, multiclass.auc, npv, ppv, rmse, sae, silhouette, sse, timeboth, timepredict, timetrain, tn, tnr, tp, tpr; Measure, makeMeasure; makeCostMeasure; performance

---

makeDownsampleWrapper *Fuse learner with simple downsampling (subsampling).*

Description

Creates a learner object, which can be used like any other learner object. It will only be trained on a subset of the original data to save computational time.

Usage

```
makeDownsampleWrapper(learner, dw.perc = 1, dw.stratify = FALSE)
```

Arguments

learner	[Learner	character(1)] The learner. If you pass a string the learner will be created via makeLearner.
dw.perc	[numeric(1)] See downsample. Default is 1.
dw.stratify	[logical(1)] See downsample. Default is FALSE.

Value

Learner .

See Also

Other downsample: downsample

---

makeFeatSelControlExhaustive

Create control structures for feature selection.

Description

Feature selection method used by selectFeatures. The following methods are available:

FeatSelControlExhaustive Exhaustive search. All feature sets (up to a certain number of features max.features) are searched.

FeatSelControlRandom Random search. Features vectors are randomly drawn, up to a certain number of features max.features. A feature is included in the current set with probability prob. So we are basically drawing (0,1)-membership-vectors, where each element is Bernoulli(prob) distributed.

FeatSelControlSequential Deterministic forward or backward search. That means extending (forward) or shrinking (backward) a feature set. Depending on the given method different approaches are taken.
sfs Sequential Forward Search: Starting from an empty model, in each step the feature increasing the performance measure the most is added to the model.
sbs Sequential Backward Search: Starting from a model with all features, in each step the feature decreasing the performance measure the least is removed from the model.
sffs Sequential Floating Forward Search: Starting from an empty model, in each step the algorithm chooses the best model from all models with one additional feature and from all models with one feature less.
sfbs Sequential Floating Backward Search: Similar to sffs but starting with a full model.

FeatSelControlGA Search via genetic algorithm. The GA is a simple (mu, lambda) or (mu + lambda) algorithm, depending on the comma setting. A comma strategy selects a new population of size mu out of the lambda > mu offspring. A plus strategy uses the joint pool of mu parents and lambda offspring for selecting mu new candidates. Out of those mu features, the

new lambda features are generated by randomly choosing pairs of parents. These are crossed over and crossover.rate represents the probability of choosing a feature from the first parent instead of the second parent. The resulting offspring is mutated, i.e., its bits are flipped with probability mutation.rate. If max.features is set, offspring are repeatedly generated until the setting is satisfied.

Usage

```
makeFeatSelControlExhaustive(same.resampling.instance = TRUE,
  maxit = NA_integer_, max.features = NA_integer_)

makeFeatSelControlGA(same.resampling.instance = TRUE, impute.val = NULL,
  maxit = NA_integer_, max.features = NA_integer_, comma = FALSE,
  mu = 10L, lambda, crossover.rate = 0.5, mutation.rate = 0.05)

makeFeatSelControlRandom(same.resampling.instance = TRUE, maxit = 100L,
  max.features = NA_integer_, prob = 0.5)

makeFeatSelControlSequential(same.resampling.instance = TRUE,
  impute.val = NULL, method, alpha = 0.01, beta = -0.001,
  maxit = NA_integer_, max.features = NA_integer_)
```

Arguments

same.resampling.instance
: [logical(1)]
Should the same resampling instance be used for all evaluations to reduce variance? Default is TRUE.

impute.val
: [numeric]
If something goes wrong during optimization (e.g, the learner crashes), this value is fed back to the tuner, so the tuning algorithm does not abort. It is not stored in the optimization path, an NA and a corresponding error message are logged instead. Note that this value is later multiplied by -1 for maximization measures internally, so you need to enter a larger positive value for maximization here as well. Default is the worst obtainable value of the performance measure you optimize for when you aggregate by mean value, or Inf instead. For multi-criteria optimization pass a vector of imputation values, one for each of your measures, in the same order as your measures.

maxit
: [integer(1)]
Maximal number of iterations. Note, that this is usually not equal to the number of function evaluations.

max.features
: [integer(1)]
Maximal number of features.

prob
: [numeric(1)]
Parameter of the random feature selection. Probability of choosing a feature.

method
: [character(1)]
Parameter of the sequential feature selection. A character representing the method.

Possible values are `sfs` (forward search), `sbs` (backward search), `sffs` (floating forward search) and `sfbs` (floating backward search).

`alpha` [`numeric(1)`]
Parameter of the sequential feature selection. Minimal required value of improvement difference for a forward / adding step. Default is 0.01.

`beta` [`numeric(1)`]
Parameter of the sequential feature selection. Minimal required value of improvement difference for a backward / removing step. Negative values imply that you allow a slight decrease for the removal of a feature. Default is -0.001.

`mu` [`integer(1)`]
Parameter of the GA feature selection. Size of the parent population.

`lambda` [`integer(1)`]
Parameter of the GA feature selection. Size of the children population (should be smaller or equal to `mu`).

`crossover.rate` [`numeric(1)`]
Parameter of the GA feature selection. Probability of choosing a bit from the first parent within the crossover mutation.

`mutation.rate` [`numeric(1)`]
Parameter of the GA feature selection. Probability of flipping a feature bit, i.e. switch between selecting / deselecting a feature.

`comma` [`logical(1)`]
Parameter of the GA feature selection, indicating whether to use a (`mu`, `lambda`) or (`mu` + `lambda`) GA. The default is `FALSE`.

Value

`FeatSelControl` . The specific subclass is one of `FeatSelControlExhaustive`, `FeatSelControlRandom`, `FeatSelControlSequential`, `FeatSelControlGA`.

See Also

Other featsel: `analyzeFeatSelResult`; `getFeatSelResult`; `makeFeatSelWrapper`; `selectFeatures`

`makeFeatSelWrapper` *Fuse learner with feature selection.*

Description

Fuses a base learner with a search strategy to select variables. Creates a learner object, which can be used like any other learner object, but which internally uses `selectFeatures`. If the train function is called on it, the search strategy and resampling are invoked to select an optimal set of variables. Finally, a model is fitted on the complete training data with these variables and returned. See `selectFeatures` for more details.

After training, the optimal features (and other related information) can be retrieved with `getFeatSelResult`.

Usage

```
makeFeatSelWrapper(learner, resampling, measures, bit.names, bits.to.features,
  control, show.info = getMlrOption("show.info"))
```

Arguments

learner	[Learner | character(1)] The learner. If you pass a string the learner will be created via makeLearner.
resampling	[ResampleInstance | ResampleDesc] Resampling strategy for feature selection. If you pass a description, it is instantiated once at the beginning by default, so all points are evaluated on the same training/test sets. If you want to change that behaviour, look at FeatSelControl.
measures	[list of Measure | Measure] Performance measures to evaluate. The first measure, aggregated by the first aggregation function is optimized, others are simply evaluated.
bit.names	[character] Names of bits encoding the solutions. Also defines the total number of bits in the encoding. Per default these are the feature names of the task.
bits.to.features	[function(x, task)] Function which transforms an integer-0-1 vector into a character vector of selected features. Per default a value of 1 in the ith bit selects the ith feature to be in the candidate solution.
control	[see FeatSelControl] Control object for search method. Also selects the optimization algorithm for feature selection.
show.info	[logical(1)] Print verbose output on console? Default is set via configureMlr.

Value

Learner .

See Also

Other featsel: FeatSelControl, FeatSelControlExhaustive, FeatSelControlGA, FeatSelControlRandom, FeatSelControlSequential, makeFeatSelControlExhaustive, makeFeatSelControlGA, makeFeatSelControlRandom, makeFeatSelControlSequential; analyzeFeatSelResult; getFeatSelResult; selectFeatures

Examples

```
# nested resampling with feature selection (with a pretty stupid algorithm for selection)
outer = makeResampleDesc("CV", iters = 2L)
inner = makeResampleDesc("Holdout")
ctrl = makeFeatSelControlRandom(maxit = 3)
lrn = makeFeatSelWrapper("classif.ksvm", resampling = inner, control = ctrl)
# we also extract the selected features for all iteration here
r = resample(lrn, iris.task, outer, extract = getFeatSelResult)
```

makeFilterWrapper *Fuse learner with a feature filter method.*

Description

Fuses a base learner with a filter method. Creates a learner object, which can be used like any other learner object. Internally uses `filterFeatures` before every model fit.

After training, the selected features can be retrieved with `getFilteredFeatures`.

Note that observation weights do not influence the filtering and are simply passed down to the next learner.

Usage

```
makeFilterWrapper(learner, fw.method = "random.forest.importance",
  fw.select = "perc", fw.val = 1)
```

Arguments

learner	[Learner | character(1)] The learner. If you pass a string the learner will be created via `makeLearner`.
fw.method	[character(1)] See `getFilterValues`. Default is "random.forest.importance".
fw.select	[character(1)] See `filterFeatures`. Default is "perc".
fw.val	[numeric(1)] See `filterFeatures`. Default is 1.

Value

`Learner` .

See Also

Other filter: `FilterResult`, `getFilterResult`; `FilterValues`; `filterFeatures`; `getFilterValues`; `getFilteredFeatures`; `listFilterMethods`

Examples

```
task = makeClassifTask(data = iris, target = "Species")
lrn = makeLearner("classif.lda")
inner = makeResampleDesc("Holdout")
outer = makeResampleDesc("CV", iters = 2)
lrn = makeFilterWrapper(lrn, fw.val = 0.5)
mod = train(lrn, task)
print(getFilteredFeatures(mod))
# now nested resampling, where we extract the features that the filter method selected
r = resample(lrn, task, outer, extract = function(model) {
```

```
  getFilteredFeatures(model)
})
print(r$extract)
```

`makeFixedHoldoutInstance` *Generate a fixed holdout instance for resampling.*

Description

Generate a fixed holdout instance for resampling.

Usage

```
makeFixedHoldoutInstance(train.inds, test.inds, size)
```

Arguments

`train.inds`	[integer] Indices for training set.
`test.inds`	[integer] Indices for test set.
`size`	[integer(1)] Size of the data set to resample.

Value

ResampleInstance .

`makeImputeMethod` *Create a custom imputation method.*

Description

This is a constructor to create your own imputation methods.

Usage

```
makeImputeMethod(learn, impute, args = list())
```

Arguments

learn	[function(data, target, col, ...)] Function to learn and extract information on column col out of data frame data. Argument target specifies the target column of the learning task. The function has to return a named list of values.
impute	[function(data, target, col, ...)] Function to impute missing values in col using information returned by learn on the same column. All list elements of the return values o learn are passed to this function into ....
args	[list] Named list of arguments to pass to learn via ....

See Also

Other impute: imputations, imputeConstant, imputeHist, imputeLearner, imputeMax, imputeMean, imputeMedian, imputeMin, imputeMode, imputeNormal, imputeUniform; impute; makeImputeWrapper; reimpute

---

makeImputeWrapper *Fuse learner with an imputation method.*

Description

Fuses a base learner with an imputation method. Creates a learner object, which can be used like any other learner object. Internally uses impute before training the learner and reimpute before predicting.

Usage

```
makeImputeWrapper(learner, classes = list(), cols = list(),
  dummy.cols = character(0L), dummy.type = "factor",
  impute.new.levels = TRUE, recode.factor.levels = TRUE)
```

Arguments

learner	[Learner	character(1)] The learner. If you pass a string the learner will be created via makeLearner.
classes	[named list] Named list containing imputation techniques for classes of columns. E.g. list(numeric = imputeMedia
cols	[named list] Named list containing names of imputation methods to impute missing values in the data column referenced by the list element's name. Overwrites imputation set via classes.
dummy.cols	[character] Column names to create dummy columns (containing binary missing indicator) for. Default is character(0).

dummy.type [character(1)]
How dummy columns are encoded. Either as 0/1 with type "numeric" or as "factor". Default is "factor".

impute.new.levels
[logical(1)]
If new, unencountered factor level occur during reimputation, should these be handled as NAs and then be imputed the same way? Default is TRUE.

recode.factor.levels
[logical(1)]
Recode factor levels after reimputation, so they match the respective element of lvls (in the description object) and therefore match the levels of the feature factor in the training data after imputation?. Default is TRUE.

Value

Learner .

See Also

Other impute: imputations, imputeConstant, imputeHist, imputeLearner, imputeMax, imputeMean, imputeMedian, imputeMin, imputeMode, imputeNormal, imputeUniform; impute; makeImputeMethod; reimpute

---

makeLearner *Create learner object.*

Description

For a classification learner the predict.type can be set to "prob" to predict probabilities and the maximum value selects the label. The threshold used to assign the label can later be changed using the setThreshold function.

Usage

```
makeLearner(cl, id = cl, predict.type = "response", fix.factors = FALSE,
  ..., par.vals = list())
```

Arguments

cl [character(1)]
Class of learner. By convention, all classification learners start with "classif.", all regression learners with "regr." and all survival learners start with "surv.". A list of all learners is available on the learners help page.

id [character(1)]
Id string for object. Used to display object. Default is cl.

predict.type [character(1)]
Classification: "response" (= labels) or "prob" (= probabilities and labels by selecting the ones with maximal probability). Regression: "response" (= mean response) or "se" (= standard errors and mean response). Survival: "response" (= some sort of orderable risk) or "prob" (= time dependent probabilities). Clustering: "response" (= cluster IDs). Default is "response".

fix.factors [logical(1)]
In some cases, problems occur in underlying learners for factor features during prediction. If the new features have LESS factor levels than during training (a strict subset), the learner might produce an error like "type of predictors in new data do not match that of the training data". In this case one can repair this problem by setting this option to TRUE. We will simply add the missing factor levels missing from the test feature (but present in training) to that feature. Default is FALSE.

... [any]
Optional named (hyper)parameters. Alternatively these can be given using the par.vals argument.

par.vals [list]
Optional list of named (hyper)parameters. The arguments in ... take precedence over values in this list. We strongly encourage you to use one or the other to pass (hyper)parameters to the learner but not both.

Value

Learner .

See Also

[resample], [predict.WrappedModel]

Examples

```
makeLearner("classif.rpart")
makeLearner("classif.lda", predict.type = "prob")
lrn = makeLearner("classif.lda", method = "t", nu = 10)
print(lrn$par.vals)
```

---

makeMeasure *Construct performance measure.*

Description

A measure object encapsulates a function to evaluate the performance of a prediction. Information about already implemented measures can be obtained here: measures.

A learner is trained on a a training set d1, results in a model m, predicts another set d2 (which may be a different one or the training set), resulting in the prediction. The performance measure can now be defined using all of the information of the original task, the fitted model and the prediction.

Object slots:

id [character(1)] See argument.

minimize [logical(1)] See argument.

properties [character] See argument.

allowed.pred.types [character] See argument.

req.pred [logical(1)] Is prediction object required in calculation?

req.task [logical(1)] Is task object required in calculation?.

req.model [logical(1)] Is model object required in calculation?

fun [function] See argument.

extra.args [list] See argument.

aggr [Aggregation] See argument..

best [numeric(1)] See argument.

worst [numeric(1)] See argument.

Usage

```
makeMeasure(id, minimize, properties = character(0L),
  allowed.pred.types = character(0L), fun, extra.args = list(),
  aggr = test.mean, best = NULL, worst = NULL)
```

Arguments

id	[character(1)] Name of measure.
minimize	[logical(1)] Should the measure be minimized? Default is in TRUE.
properties	[character] Set of measure properties. Some standard property names include: **classif** Is the measure applicable for classification? **classif.multi** Is the measure applicable for multi-class classification? **regr** Is the measure applicable for regression? **surv** Is the measure applicable for survival? **costsens** Is the measure applicable for cost-sensitve learning? Default is character(0).
allowed.pred.types	[character] Which prediction types are allowed for this measure? Subset of "response","prob", "se". Default is character(0).
fun	[function(task, model, pred, extra.args)] Calculates performance value.
extra.args	[list] List of extra arguments which will always be passed to fun. Default is empty list.

aggr [Aggregation]
Aggregation funtion, which is used to aggregate the values measured on test / training sets of the measure to a single value. Default is test.mean.

best [numeric(1)]
Best obtainable value for measure. Default is -Inf or Inf, depending on minimize.

worst [numeric(1)]
Worst obtainable value for measure. Default is Inf or -Inf, depending on minimize.

Value

Measure .

See Also

Other performance: G1, G2, acc, auc, bac, ber, cindex, db, dunn, f1, fdr, featperc, fn, fnr, fp, fpr, gmean, gpr, mae, mcc, mcp, meancosts, measures, medae, medse, mmce, mse, multiclass.auc, npv, ppv, rmse, sae, silhouette, sse, timeboth, timepredict, timetrain, tn, tnr, tp, tpr; makeCostMeasure; makeCustomResampledMeasure; performance

Examples

```
f = function(task, model, pred, extra.args)
  sum((pred$data$response - pred$data$truth)^2)
makeMeasure(id = "my.sse", minimize = TRUE, properties = c("regr", "response"), fun = f)
```

---

makeModelMultiplexer *Create model multiplexer for model selection to tune over multiple possible models.*

Description

Combines multiple base learners by dispatching on the hyperparameter "selected.learner" to a specific model class. This allows to tune not only the model class (SVM, random forest, etc) but also their hyperparameters in one go. Combine this with tuneParams and makeTuneControlIrace for a very powerful approach, see example below.

The parameter set is the union of all (unique) base learners. In order to avoid name clashes all parameter names are prefixed with the base learner id, i.e. "[learner.id].[parameter.name]".

Usage

```
makeModelMultiplexer(base.learners)
```

Arguments

base.learners [list of Learner]
List of Learners with unique IDs.

Value

ModelMultiplexer . A Learner specialized as ModelMultiplexer.

Note

Note that logging output during tuning is somewhat shortened to make it more readable. I.e., the artificial prefix before parameter names is suppressed.

See Also

Other multiplexer: makeModelMultiplexerParamSet

Other tune: TuneControl, TuneControlCMAES, TuneControlGenSA, TuneControlGrid, TuneControlIrace, TuneControlRandom, makeTuneControlCMAES, makeTuneControlGenSA, makeTuneControlGrid, makeTuneControlIrace, makeTuneControlRandom; getTuneResult; makeModelMultiplexerParamSet; makeTuneWrapper; tuneParams; tuneThreshold

Examples

```
bls = list(
  makeLearner("classif.ksvm"),
  makeLearner("classif.randomForest")
)
lrn = makeModelMultiplexer(bls)
# simple way to contruct param set for tuning
# parameter names are prefixed automatically and the 'requires'
# element is set, too, to make all paramaters subordinate to 'selected.learner'
ps = makeModelMultiplexerParamSet(lrn,
  makeNumericParam("sigma", lower = -10, upper = 10, trafo = function(x) 2^x),
  makeIntegerParam("ntree", lower = 1L, upper = 500L)
)
print(ps)
rdesc = makeResampleDesc("CV", iters = 2L)
# to save some time we use random search. but you probably want something like this:
# ctrl = makeTuneControlIrace(maxExperiments = 500L)
ctrl = makeTuneControlRandom(maxit = 10L)
res = tuneParams(lrn, iris.task, rdesc, par.set = ps, control = ctrl)
print(res)
print(head(as.data.frame(res$opt.path)))

# more unique and reliable way to construct the param set
ps = makeModelMultiplexerParamSet(lrn,
  classif.ksvm = makeParamSet(
    makeNumericParam("sigma", lower = -10, upper = 10, trafo = function(x) 2^x)
  ),
  classif.randomForest = makeParamSet(
    makeIntegerParam("ntree", lower = 1L, upper = 500L)
  )
)

# this is how you would construct the param set manually, works too
ps = makeParamSet(
```

```
  makeDiscreteParam("selected.learner", values = extractSubList(bls, "id")),
  makeNumericParam("classif.ksvm.sigma", lower = -10, upper = 10, trafo = function(x) 2^x,
    requires = quote(selected.learner == "classif.ksvm")),
  makeIntegerParam("classif.randomForest.ntree", lower = 1L, upper = 500L,
    requires = quote(selected.learner == "classif.randomForst"))
)

# all three ps-objects are exactly the same internally.
```

`makeModelMultiplexerParamSet` *Creates a parameter set for model multiplexer tuning.*

Description

Handy way to create the param set with too much typing.

The following is done automatically:

- The `selected.learner` param is created
- Parameter names are prefixed.
- The `requires` field of each param is set. This makes all parameters subordinate to `selected.learner`

Usage

```
makeModelMultiplexerParamSet(multiplexer, ..., .check = TRUE)
```

Arguments

`multiplexer`	[ModelMultiplexer] The muliplexer learner.
`...`	[ParamSet	Param] (a) First option: Named param sets. Names must correspond to base learners. You only need to enter the parameters you want to tune without reference to the `selected.learner` field in any way. (b) Second option. Just the params you would enter in the param sets. Even shorterto create. Only works when it can be uniquely identified to which learner each of your passed parameters belongs.
`.check`	[logical] Check that for each param in `...` one param in found in the base learners. Default is `TRUE`

Value

ParamSet .

See Also

Other multiplexer: ModelMultiplexer, makeModelMultiplexer

Other tune: ModelMultiplexer, makeModelMultiplexer; TuneControl, TuneControlCMAES, TuneControlGenSA, TuneControlGrid, TuneControlIrace, TuneControlRandom, makeTuneControlCMAES, makeTuneControlGenSA, makeTuneControlGrid, makeTuneControlIrace, makeTuneControlRandom; getTuneResult; makeTuneWrapper; tuneParams; tuneThreshold

Examples

```
# See makeModelMultiplexer
```

---

makeOverBaggingWrapper

Fuse learner with the bagging technique and oversampling for imbalancy correction.

Description

Fuses a classification learner for binary classification with an over-bagging method for imbalancy correction when we have strongly unequal class sizes. Creates a learner object, which can be used like any other learner object. Models can easily be accessed via getBaggingModels.

OverBagging is implemented as follows: For each iteration a random data subset is sampled. Minority class examples are oversampled with replacement with a given rate. Majority class examples are either simply copied into each bag, or bootstrapped with replacement until we have as many majority class examples as in the original training data. Features are currently not changed or sampled.

Prediction works as follows: For classification we do majority voting to create a discrete label and probabilities are predicted by considering the proportions of all predicted labels.

Usage

```
makeOverBaggingWrapper(learner, obw.iters = 10L, obw.rate = 1,
  obw.maxcl = "boot")
```

Arguments

learner	[Learner | character(1)] The learner. If you pass a string the learner will be created via makeLearner.
obw.iters	[integer(1)] Number of fitted models in bagging. Default is 10.
obw.rate	[numeric(1)] Factor to upsample the smaller class in each bag. Must be between 1 and Inf, where 1 means no oversampling and 2 would mean doubling the class size. Default is 1.

obw.maxcl [character(1)]
character value that controls how to sample majority class. "all" means every instance of the majority class gets in each bag, "boot" means the majority class instances are bootstrapped in each iteration. Default is "boot".

Value

Learner .

See Also

Other imbalancy: makeOversampleWrapper, makeUndersampleWrapper; oversample, undersample; smote

---

makePreprocWrapper *Fuse learner with preprocessing.*

Description

Fuses a base learner with a preprocessing method. Creates a learner object, which can be used like any other learner object, but which internally preprocesses the data as requested. If the train or predict function is called on data / a task, the preprocessing is always performed automatically.

Usage

```
makePreprocWrapper(learner, train, predict, par.set = makeParamSet(),
  par.vals = list())
```

Arguments

learner [Learner | character(1)]
The learner. If you pass a string the learner will be created via makeLearner.

train [function(data, target, args)]
Function to preprocess the data before training. target is a string and denotes the target variable in data. args is a list of further arguments and parameters to influence the preprocessing. Must return a list(data, control), where data is the preprocessed data and control stores all information necessary to do the preprocessing before predictions.

predict [function(data, target, args, control)]
Function to preprocess the data before prediction. target is a string and denotes the target variable in data. args are the args that were passed to train. control is the object you returned in train.

par.set [ParamSet]
Parameter set of LearnerParam objects to describe the parameters in args. Default is empty set.

par.vals [list]
Named list of default values for params in args repectively par.set. Default is empty list.

Value

Learner .

---

makeResampleDesc *Create a description object for a resampling strategy.*

Description

A description of a resampling algorithm contains all necessary information to create a ResampleInstance, when given the size of the data set.

Usage

```
makeResampleDesc(method, predict = "test", ..., stratify = FALSE)
```

Arguments

method	[character(1)] "CV" for cross-validation, "LOO" for leave-one-out, "RepCV" for repeated cross-validation, "Bootstrap" for out-of-bag bootstrap, "Subsample" for subsampling, "Holdout" for holdout.
predict	[character(1)] What to predict during resampling: "train", "test" or "both" sets. Default is "test".
...	[any] Further parameters for strategies. **iters** [integer(1)] Number of iterations, for "CV", "Subsample" and "Bootstrap" **split** [numeric(1)] Proportion of training cases for "Holdout" and "Subsample" between 0 and 1. Default is 2/3. **reps [integer(1)**] Repeats for "RepCV". Here iters = folds * reps. Default is 10. **folds [integer(1)**] Folds in the repeated CV for RepCV. Here iters = folds * reps. Default is 10.
stratify	[logical(1)] Should stratification be done for the classes in classification tasks? This means that the resampling strategy is applied to all classes individually and the resulting index sets are joined to make sure that the proportion of observations in each training set is as in the original data set. Useful for imbalanced class sizes.

Details

Some notes on some special strategies:

Repeated cross-validation Use "RepCV". Then you have to set the aggregation function for your preferred performance measure to "testgroup.mean" via `setAggregation`.

B632 bootstrap Use "Bootstrap" for bootstrap and set predict to "both". Then you have to set the aggregation function for your preferred performance measure to "b632" via `setAggregation`.

B632+ bootstrap Use "Bootstrap" for bootstrap and set predict to "both". Then you have to set the aggregation function for your preferred performance measure to "b632plus" via `setAggregation`.

Fixed Holdout set Use `makeFixedHoldoutInstance`.

Object slots:

id [`character(1)`] Name of resampling strategy.

iters [`integer(1)`] Number of iterations. Note that this is always the complete number of generated train/test sets, so for a 10-times repeated 5fold cross-validation it would be 50.

predict [`character(1)`] See argument.

stratify [`logical(1)`] See argument.

All parameters passed in ... under the respective argument name See arguments.

Value

`ResampleDesc` .

See Also

Other resample: `ResampleInstance`, `makeResampleInstance`; `ResamplePrediction`; `bootstrapB632`, `bootstrapB632plus`, `bootstrapOOB`, `crossval`, `holdout`, `repcv`, `resample`, `subsample`

Examples

```
# Bootstraping
makeResampleDesc("Bootstrap", iters = 10)
makeResampleDesc("Bootstrap", iters = 10, predict = "both")

# Subsampling
makeResampleDesc("Subsample", iters = 10, split = 3/4)
makeResampleDesc("Subsample", iters = 10)

# Holdout a.k.a. test sample estimation
makeResampleDesc("Holdout")
```

makeResampleInstance *Instantiates a resampling strategy object.*

Description

This class encapsulates training and test sets generated from the data set for a number of iterations. It mainly stores a set of integer vectors indicating the training and test examples for each iteration.

Usage

```
makeResampleInstance(desc, task, size, ...)
```

Arguments

desc	[ResampleDesc	character(1)] Resampling description object or name of resampling strategy. In the latter case makeResampleDesc will be called internally on the string.
task	[Task] Data of task to resample from. Prefer to pass this instead of size.
size	[integer] Size of the data set to resample. Can be used instead of task.
...	[any] Passed down to makeResampleDesc in case you passed a string in desc. Otherwise ignored.

Details

Object slots:

desc [ResampleDesc **]** See argument.

size [integer(1)] See argument.

train.inds [list of integer **]** List of of training indices for all iterations.

test.inds [list of integer **]** List of of test indices for all iterations.

group [factor **]** Optional grouping of resampling iterations. This encodes whether specfic iterations 'belong together' (e.g. repeated CV), and it can later be used to aggregate performance values accordingly. Default is 'factor()'.

Value

ResampleInstance .

See Also

Other resample: ResampleDesc, makeResampleDesc; ResamplePrediction; bootstrapB632, bootstrapB632plus, bootstrapOOB, crossval, holdout, repcv, resample, subsample

Examples

```
rdesc = makeResampleDesc("Bootstrap", iters = 10)
rin = makeResampleInstance(rdesc, task = iris.task)

rdesc = makeResampleDesc("CV", iters = 50)
rin = makeResampleInstance(rdesc, size = nrow(iris))

rin = makeResampleInstance("CV", iters = 10, task = iris.task)
```

`makeSMOTEWrapper` *Fuse learner with SMOTE oversampling for imbalancy correction in binary classification.*

Description

Creates a learner object, which can be used like any other learner object. Internally uses `smote` before every model fit.

Note that observation weights do not influence the sampling and are simply passed down to the next learner.

Usage

```
makeSMOTEWrapper(learner, sw.rate = 1, sw.nn = 5L)
```

Arguments

`learner`	[`Learner`	`character(1)`] The learner. If you pass a string the learner will be created via `makeLearner`.
`sw.rate`	[`numeric(1)`] Factor to oversample the smaller class. Must be between 1 and `Inf`, where 1 means no oversampling and 2 would mean doubling the class size. Default is 1.
`sw.nn`	[`integer(1)`] Number of nearest neighbors to consider. Default is 5.

Value

`Learner` .

makeTuneControlCMAES *Create control structures for tuning.*

Description

The following tuners are available:

makeTuneControlGrid Grid search. All kinds of parameter types can be handled. You can either use their correct param type and `resolution`, or discretize them yourself by always using `makeDiscreteParam` in the `par.set` passed to `tuneParams`.

makeTuneControlRandom Random search. All kinds of parameter types can be handled.

makeTuneControlCMAES CMA Evolution Strategy with method `cma_es`. Can handle numeric(vector) and integer(vector) hyperparameters, but no dependencies. For integers the internally proposed numeric values are automatically rounded. The sigma variance parameter is initialized to 1/4 of the span of box-constraints per parameter dimension.

makeTuneControlGenSA Generalized simulated annealing with method `GenSA`. Can handle numeric(vector) and integer(vector) hyperparameters, but no dependencies. For integers the internally proposed numeric values are automatically rounded.

makeTuneControlIrace Tuning with iterated F-Racing with method `irace`. All kinds of parameter types can be handled. We return the best of the final elite candidates found by irace in the last race. Its estimated performance is the mean of all evaluations ever done for that candidate.

Some notes on irace: For resampling you have to pass a `ResampleDesc`, not a `ResampleInstance`. The resampling strategy is randomly instantiated `n.instances` times and these are the instances in the sense of irace (`instances` element of `tunerConfig` in `irace`). Also note that irace will always store its tuning results in a file on disk, see the package documentation for details on this and how to change the file path.

Usage

```
makeTuneControlCMAES(same.resampling.instance = TRUE, impute.val = NULL,
  start = NULL, ...)

makeTuneControlGenSA(same.resampling.instance = TRUE, impute.val = NULL,
  start = NULL, ...)

makeTuneControlGrid(same.resampling.instance = TRUE, impute.val = NULL,
  resolution = 10L)

makeTuneControlIrace(impute.val = NULL, n.instances = 100L,
  show.irace.output = FALSE, ...)

makeTuneControlRandom(same.resampling.instance = TRUE, maxit = 100L)
```

Arguments

`resolution` [integer]
Resolution of the grid for each numeric/integer parameter in `par.set`. For vector parameters, it is the resolution per dimension. Either pass one resolution for all parameters, or a named vector. See `generateGridDesign`. Default is 10.

`n.instances` [integer(1)]
Number of random resampling instances for irace, see details. Default is 100.

`show.irace.output` [logical(1)]
Show console output of irace while tuning? Default is `FALSE`.

`same.resampling.instance` [logical(1)]
Should the same resampling instance be used for all evaluations to reduce variance? Default is `TRUE`.

`impute.val` [numeric]
If something goes wrong during optimization (e.g, the learner crashes), this value is fed back to the tuner, so the tuning algorithm does not abort. It is not stored in the optimization path, an NA and a corresponding error message are logged instead. Note that this value is later multiplied by -1 for maximization measures internally, so you need to enter a larger positive value for maximization here as well. Default is the worst obtainable value of the performance measure you optimize for when you aggregate by mean value, or `Inf` instead. For multi-criteria optimization pass a vector of imputation values, one for each of your measures, in the same order as your measures.

`start` [numeric]
Named list of initial parameter values.

`...` [any]
Further control parameters passed to the `control` argument of `cma_es` and `tunerConfig` argument of `irace`.

`maxit` [integer(1)]
Number of iterations for random search. Default is 100.

Value

`TuneControl` . The specific subclass is one of `TuneControlGrid`, `TuneControlRandom`, `TuneControlCMAES`, `TuneControlGenSA`, `TuneControlIrace`.

See Also

Other tune: `ModelMultiplexer`, `makeModelMultiplexer`; `getTuneResult`; `makeModelMultiplexerParamSet`; `makeTuneWrapper`; `tuneParams`; `tuneThreshold`

makeTuneMultiCritControlGrid
Create control structures for multi-criteria tuning.

Description

The following tuners are available:

makeTuneControlGrid Grid search. All kinds of parameter types can be handled. You can either use their correct param type and `resolution`, or discretize them yourself by always using `makeDiscreteParam` in the `par.set` passed to `tuneParams`.

makeTuneMultiCritControlRandom Random search. All kinds of parameter types can be handled.

makeTuneMultiCritControlNSGA2 Evolutionary method `nsga2`. Can handle numeric(vector) and integer(vector) hyperparameters, but no dependencies. For integers the internally proposed numeric values are automatically rounded.

Usage

```
makeTuneMultiCritControlGrid(same.resampling.instance = TRUE,
  resolution = 10L)

makeTuneMultiCritControlNSGA2(same.resampling.instance = TRUE,
  impute.val = NULL, ...)

makeTuneMultiCritControlRandom(same.resampling.instance = TRUE,
  maxit = 100L)
```

Arguments

`maxit` [integer(1)]
Number of iterations for random search. Default is 100.

`same.resampling.instance` [logical(1)]
Should the same resampling instance be used for all evaluations to reduce variance? Default is `TRUE`.

`resolution` [integer]
Resolution of the grid for each numeric/integer parameter in `par.set`. For vector parameters, it is the resolution per dimension. Either pass one resolution for all parameters, or a named vector. See `generateGridDesign`. Default is 10.

`impute.val` [numeric]
If something goes wrong during optimization (e.g, the learner crashes), this value is fed back to the tuner, so the tuning algorithm does not abort. It is not stored in the optimization path, an NA and a corresponding error message are logged instead. Note that this value is later multiplied by -1 for maximization measures internally, so you need to enter a larger positive value for maximization

here as well. Default is the worst obtainable value of the performance measure you optimize for when you aggregate by mean value, or `Inf` instead. For multi-criteria optimization pass a vector of imputation values, one for each of your measures, in the same order as your measures.

`...` [any]
Further control parameters passed to the `control` argument of `cma_es` and `tunerConfig` argument of `irace`.

Value

`TuneMultiCritControl` . The specific subclass is one of `TuneMultiCritControlGrid`, `TuneMultiCritControlRandom`, `TuneMultiCritControlNSGA2`.

See Also

Other tune_multicrit: `plotTuneMultiCritResult`; `tuneParamsMultiCrit`

`makeTuneWrapper` *Fuse learner with tuning.*

Description

Fuses a base learner with a search strategy to select its hyperparameters. Creates a learner object, which can be used like any other learner object, but which internally uses `tuneParams`. If the train function is called on it, the search strategy and resampling are invoked to select an optimal set of hyperparameter values. Finally, a model is fitted on the complete training data with these optimal hyperparameters and returned. See `tuneParams` for more details.

After training, the optimal hyperparameters (and other related information) can be retrieved with `getTuneResult`.

Usage

```
makeTuneWrapper(learner, resampling, measures, par.set, control,
  show.info = getMlrOption("show.info"))
```

Arguments

`learner` [`Learner` | `character(1)`]
The learner. If you pass a string the learner will be created via `makeLearner`.

`resampling` [`ResampleInstance` | `ResampleDesc`]
Resampling strategy to evaluate points in hyperparameter space. If you pass a description, it is instantiated once at the beginning by default, so all points are evaluated on the same training/test sets. If you want to change that behavior, look at `TuneControl`.

`measures` [list of `Measure` | `Measure`]
Performance measures to evaluate. The first measure, aggregated by the first aggregation function is optimized, others are simply evaluated.

par.set	[ParamSet] Collection of parameters and their constraints for optimization.
control	[TuneControl] Control object for search method. Also selects the optimization algorithm for tuning.
show.info	[logical(1)] Print verbose output on console? Default is set via configureMlr.

Value

Learner .

See Also

Other tune: ModelMultiplexer, makeModelMultiplexer; TuneControl, TuneControlCMAES, TuneControlGenSA, TuneControlGrid, TuneControlIrace, TuneControlRandom, makeTuneControlCMAES, makeTuneControlGenSA, makeTuneControlGrid, makeTuneControlIrace, makeTuneControlRandom; getTuneResult; makeModelMultiplexerPar tuneParams; tuneThreshold

Examples

```
task = makeClassifTask(data = iris, target = "Species")
lrn = makeLearner("classif.ksvm")
# stupid mini grid
ps = makeParamSet(
  makeDiscreteParam("C", values = 1:2),
  makeDiscreteParam("sigma", values = 1:2)
)
ctrl = makeTuneControlGrid()
inner = makeResampleDesc("Holdout")
outer = makeResampleDesc("CV", iters = 2)
lrn = makeTuneWrapper(lrn, resampling = inner, par.set = ps, control = ctrl)
mod = train(lrn, task)
print(getTuneResult(mod))
# nested resampling for evaluation
# we also extract tuned hyper pars in each iteration
r = resample(lrn, task, outer, extract = getTuneResult)
print(r$extract)
```

---

makeUndersampleWrapper

Fuse learner with simple ove/underrsampling for imbalancy correction in binary classification.

Description

Creates a learner object, which can be used like any other learner object. Internally uses oversample or undersample before every model fit.

Note that observation weights do not influence the sampling and are simply passed down to the next learner.

Usage

```
makeUndersampleWrapper(learner, usw.rate = 1)

makeOversampleWrapper(learner, osw.rate = 1)
```

Arguments

learner	[Learner	character(1)] The learner. If you pass a string the learner will be created via makeLearner.
usw.rate	[numeric(1)] Factor to downsample the bigger class. Must be between 0 and 1, where 1 means no downsampling, 0.5 implies reduction to 50 percent and 0 would imply reduction to 0 observations. Default is 1.
osw.rate	[numeric(1)] Factor to oversample the smaller class. Must be between 1 and Inf, where 1 means no oversampling and 2 would mean doubling the class size. Default is 1.

Value

Learner .

See Also

Other imbalancy: makeOverBaggingWrapper; oversample, undersample; smote

---

makeWeightedClassesWrapper

Wraps a classifier for weighted fitting where each class receives a weight.

Description

Creates a wrapper, which can be used like any other learner object.

Fitting is performed in a weighted fashion where each observation receives a weight, depending on the class it belongs to, see wcw.weight. This might help to mitigate problems caused by imbalanced class distributions.

This weighted fitting can be achieved in two ways:

a) The learner already has a parameter for class weighting, so one weight can directly be defined per class. Example: "classif.ksvm" and parameter class.weights. In this case we don't really do anything fancy. We convert wcw.weight a bit, but basically simply bind its value to the class weighting param. The wrapper in this case simply offers a convenient, consistent fashion for class weighting - and tuning! See example below.

b) The learner does not have a direct parameter to support class weighting, but supports observation weights, so hasProperties(learner, 'weights') is TRUE. This means that an individual, arbitrary weight can be set per observation during training. We set this weight depending on the class internally in the wrapper. Basically we introduce something like a new "class.weights" parameter for the learner via observation weights.

Usage

```
makeWeightedClassesWrapper(learner, wcw.param = NULL, wcw.weight = 1)
```

Arguments

learner	[Learner	character(1)] The classification learner. If you pass a string the learner will be created via makeLearner.
wcw.param	[character(1)] Name of already existing learner param which allows class weighting. Or NULL if such a param does not exist. Must during training accept a named vector of class weights, where length equals the number of classes. Default is NULL.
wcw.weight	[numeric] Weight for each class. Must be a vector of the same number of elements as classes are in task, and must also be in the same order as the class levels are in task$task.desc$class.levels. For convenience, one must pass a single number in case of binary classification, which is then taken as the weight of the positive class, while the negative class receives a weight of 1. Default is 1.

Value

Learner .

Examples

```
# using the direct parameter of the SVM
lrn = makeWeightedClassesWrapper("classif.ksvm", wcw.param = "class.weights", wcw.weight = 0.01)
res = holdout(lrn, sonar.task)
print(getConfMatrix(res$pred))

# using the observation weights of logreg
lrn = makeWeightedClassesWrapper("classif.logreg", wcw.weight = 0.01)
res = holdout(lrn, sonar.task)
print(getConfMatrix(res$pred))

# tuning the imbalancy param and the SVM param in one go
lrn = makeWeightedClassesWrapper("classif.ksvm", wcw.param = "class.weights")
ps = makeParamSet(
  makeNumericParam("wcw.weight", lower = 1, upper = 10),
  makeNumericParam("C", lower = -12, upper = 12, trafo = function(x) 2^x),
  makeNumericParam("sigma", lower = -12, upper = 12, trafo = function(x) 2^x)
)
ctrl = makeTuneControlRandom(maxit = 3L)
rdesc = makeResampleDesc("CV", iters = 2L, stratify = TRUE)
res = tuneParams(lrn, sonar.task, rdesc, par.set = ps, control = ctrl)
print(res)
print(res$opt.path)
```

makeWrappedModel *Induced model of learner.*

Description

Result from train.

It internally stores the underlying fitted model, the subset used for training, features used for training, levels of factors in the data set and computation time that was spent for training.

Object members: See arguments.

The constructor makeWrappedModel is mainly for internal use.

Usage

```
makeWrappedModel(learner, learner.model, task.desc, subset, features,
  factor.levels, time)
```

Arguments

learner	[Learner	character(1)] The learner. If you pass a string the learner will be created via makeLearner.
learner.model	[any] Underlying model.
task.desc	[TaskDesc] Task description object.
subset	[integer] Subset used for training.
features	[character] Features used for training.
factor.levels	[named list of character] Levels of factor variables (features and potentially target) in training data. Named by variable name, non-factors do not occur in the list.
time	[numeric(1)] Computation time for model fit in seconds.

Value

WrappedModel .

`measures` *Performance measures.*

Description

A performance measure is evaluated after a single train/predict step and returns a single number to assess the quality of the prediction (or maybe only the model, think AIC). The measure itself knows whether it wants to be minimized or maximized and for what tasks it is applicable. See below for a list of already implemented measures. If you want a measure for a misclassification cost matrix, look at `makeCostMeasure`. If you want to implement your own measure, look at `makeMeasure`.

Classification:

- **mmce**
  Mean misclassification error.
- **acc**
  Accuracy.
- **bac**
  Balanced accuracy. Mean of true positive rate and true negative rate
- **ber**
  Balanced error rate. Mean of misclassification error rates on all individual classes.
- **tp**
  True positives.
- **tpr**
  True positive rate, also called hit rate or recall.
- **fp**
  False positives, also called false alarms.
- **fpr**
  False positive rate, also called false alarm rate or fall-out.
- **tn**
  True negatives, also called correct rejections.
- **tnr**
  True negative rate. Also called specificity.
- **fn**
  False negatives, also called misses.
- **fnr**
  False negative rate.
- **ppv**
  Positive predictive value, also called precision.
- **npv**
  Negative predictive value.
- **fdr**
  False discovery rate.

- **f1**
  F1 measure.
- **mcc**
  Matthews correlation coefficient.
- **gmean**
  G-mean, geometric mean of recall and specificity.
- **gpr**
  Geometric mean of precision and recall.
- **auc**
  Area under the curve.
- **multiclass.auc**
  Area under the curve for multiclass problems. Calls `pROC::multiclass.roc`.

Only `mmce`, `acc`, `multiclass.auc` and `ber` can be used for multiclass problems.

Regression:

- **sse**
  Sum of squared errors
- **mse**
  Mean of squared errors
- **medse**
  Median of squared errors
- **sae**
  Sum of absolute errors
- **mae**
  Mean of absolute errors
- **medae**
  Median of absolute errors
- **rmse**
  Root mean square error

Survival:

- **cindex**
  Concordance index

Cost-sensitive:

- **meancosts**
  Mean costs of the predicted choices.
- **mcp**
  Misclassification penalty, i.e. average difference between costs of oracle and model prediction.

Clustering:

- **db**
  Davies-Bouldin cluster separation measure, see `index.DB`

- **dunn**
  Dunn index, see dunn
- **G1**
  Calinski-Harabasz pseudo F statistic, see index.G1
- **G2**
  Baker and Hubert adaptation of Goodman-Kruskal's gamma statistic, see index.G2
- **silhouette**
  Rousseeuw's silhouette internal cluster quality index, see index.S

General:

- **timetrain**
  Time of fitting the model
- **timepredict**
  Time of predicting test set
- **timeboth**
  timetrain + trainpredict
- **featperc**
  Percentage of original features used for model, useful for feature selection.

Usage

none

none

none

none

none

none

none

none

none

none

none

none

none

none

none

none

none

none

none

none

none

none

none

none

none

none

none

none

none

none

none

none

none

none

none

none

meancosts

mcp

none

none

none

none

none

Format

none

See Also

Other performance: Measure, makeMeasure; makeCostMeasure; makeCustomResampledMeasure; performance

---

`mlr.bh` *Boston Housing regression task*

Description

Contains the task (`bh.task`) and a matching resample instance (`bh.rin`).

References

See BostonHousing.

`mlr.iris` *Iris classification task*

Description

Contains the task (`iris.task`) and a resample instance (`iris.rin`).

References

See iris.

mlr.sonar *Sonar classification task*

Description

Contains the task (sonar.task) and a matching resample instance (sonar.rin).

Contains the task (sonar.task) and a matching resample instance (sonar.rin).

References

See Sonar.

See Sonar.

normalizeFeatures *Normalize features*

Description

Normalize features by different methods. Internally normalize is used. Non numerical features will be left untouched and passed to the result.

Usage

```
normalizeFeatures(obj, target = character(0L), method = "standardize",
  exclude = character(0L), range = c(0, 1))
```

Arguments

obj	[data.frame | Task] Input data.
target	[character(1)] Name of the column(s) specifying the response. Only used when obj is a data.frame, otherwise ignored.
method	[character(1)] Normalizing method. Available are: "center": centering of each feature "scale": scaling of each feature "standardize": centering and scaling "range": Scale the data to a given range.
exclude	[character] Names of the columns to exclude. The target does not have to be included here. Default is none.
range	[numeric(2)] Range the features should be scaled to. Default is c(0,1).

Value

data.frame | Task . Same type as obj.

See Also

normalize

---

oversample *Over- or undersample binary classification task to handle class imbalancy.*

Description

Oversampling: From the smaller class, observations are randomly drawn with repetitions.

Undersampling: From the larger class, observations are randomly drawn without repetitions.

Usage

```
oversample(task, rate)

undersample(task, rate)
```

Arguments

task	[Task] The task.
rate	[numeric(1)] Factor to upsample the smaller or downsample the bigger class. For undersampling: Must be between 0 and 1, where 1 means no downsampling, 0.5 implies reduction to 50 percent and 0 would imply reduction to 0 observations. For oversampling: Must be between 1 and Inf, where 1 means no oversampling and 2 would mean doubling the class size.

Value

Task .

See Also

Other imbalancy: makeOverBaggingWrapper; makeOversampleWrapper, makeUndersampleWrapper; smote

`performance` *Measure performance of prediction.*

Description

Measures the quality of a prediction w.r.t. some performance measure.

Usage

```
performance(pred, measures, task, model)
```

Arguments

`pred`	[`Prediction`] Prediction object.
`measures`	[`Measure`	list of `Measure`] Performance measure(s) to evaluate.
`task`	[`Task`] Learning task, might be requested by performance measure, usually not needed except for clustering.
`model`	[`WrappedModel`] Model built on training data, might be requested by performance measure, usually not needed.

Value

named `numeric` . Performance value(s), named by measure(s).

See Also

Other performance: `G1`, `G2`, `acc`, `auc`, `bac`, `ber`, `cindex`, `db`, `dunn`, `f1`, `fdr`, `featperc`, `fn`, `fnr`, `fp`, `fpr`, `gmean`, `gpr`, `mae`, `mcc`, `mcp`, `meancosts`, `measures`, `medae`, `medse`, `mmce`, `mse`, `multiclass.auc`, `npv`, `ppv`, `rmse`, `sae`, `silhouette`, `sse`, `timeboth`, `timepredict`, `timetrain`, `tn`, `tnr`, `tp`, `tpr`; `Measure`, `makeMeasure`; `makeCostMeasure`; `makeCustomResampledMeasure`

Examples

```
training.set = seq(1, nrow(iris), by = 2)
test.set = seq(2, nrow(iris), by = 2)

task = makeClassifTask(data = iris, target = "Species")
lrn = makeLearner("classif.lda")
mod = train(lrn, task, subset = training.set)
pred = predict(mod, newdata = iris[test.set, ])
performance(pred, measures = mmce)

# Compute multiple performance measures at once
ms = list("mmce" = mmce, "acc" = acc, "timetrain" = timetrain)
performance(pred, measures = ms, task, mod)
```

plotFilterValues *Plot filter values.*

Description

Plot filter values.

Usage

```
plotFilterValues(fvalues, sort = "dec", n.show = 20L,
  feat.type.cols = c("darkgreen", "darkblue"))
```

Arguments

fvalues	[FilterValues] Filtr values.
sort	[character(1)] Sort features like this. "dec" = decreasing, "inc" = increasing, "none" = no sorting. Default is decreasing.
n.show	[integer(1)] Number of features (maximal) to show. Default is 20.
feat.type.cols	[character(2)*] Colors for factor and numeric features. NULL means no colors. Default is dark-green and darkblue.

Value

ggplot2 plot object.

Examples

```
fv = getFilterValues(iris.task, method = "chi.squared")
plotFilterValues(fv)
```

plotLearnerPrediction *Visualizes a learning algorithm on a 1D or 2D data set.*

Description

Trains the model for 1 or 2 selected features, then displays it via ggplot. Good for teaching or exploring models.

For classification, only 2D plots are supported. The data points, the classification and potentially through color alpha blending the posterior probabilities are shown.

For regression, 1D and 2D plots are supported. 1D shows the data, the estimated mean and potentially the estimated standard error. 2D does not show estimated standard error, but only the estimated mean via background color.

The plot title displays the model id, its parameters, the test training performance and the cross-validation performance.

Usage

```
plotLearnerPrediction(learner, task, features = NULL, measures, cv = 10L,
  ..., gridsize, pointsize = 2, prob.alpha = TRUE, se.band = TRUE,
  err.mark = "train", bg.cols = c("darkblue", "green", "darkred"),
  err.col = "orange")
```

Arguments

learner	[Learner | character(1)] The learner. If you pass a string the learner will be created via makeLearner.
task	[Task] The task.
features	[character] Selected features for model. By default the first 2 features are used.
measures	[Measure | list of Measure] Performance measure(s) to evaluate.
cv	[integer(1)] Do cross-validation and display in plot title? Number of folds. 0 means no CV. Default is 10.
...	[any] Parameters for learner.
gridsize	[integer(1)] Grid resolution per axis for background predictions. Default is 500 for 1D and 100 for 2D.
pointsize	[numeric(1)] Pointsize for ggplot2 geom_point for data points. Default is 2.
prob.alpha	[logical(1)] For classification: Set alpha value of background to probability for predicted class? Allows visualization of "confidence" for prediction. If not, only a constant color is displayed in the background for the predicted label. Default is TRUE.
se.band	[logical(1)] For regression in 1D: Show band for standard error estimation? Default is TRUE.

err.mark [character(1)]: For classification: Either mark error of the model on the training data ("train") or during cross-validation ("cv") or not at all with "none". Default is "train".

bg.cols [character(3)]
Background colors for classification and regression. Sorted from low, medium to high. Default is TRUE.

err.col [character(1)]
For classification: Color of misclassified data points. Default is "orange"

Value

The ggplot2 object.

---

plotThreshVsPerf *Plot threshold vs. performance(s) for 2-class classification.*

Description

Plot threshold vs. performance(s) for 2-class classification.

Usage

```
plotThreshVsPerf(pred, measures, mark.th = NA_real_, gridsize = 100L,
  linesize = 1.5)
```

Arguments

pred [Prediction]
Prediction object.

measures [Measure | list of Measure]
Performance measure(s) to evaluate.

mark.th [numeric(1)]
Mark given threshold with vertical line? Default is NA which means not to do it.

gridsize [integer(1)]
Grid resolution for x-axis (threshold). Default is 100.

linesize [numeric(1)]
Linesize for ggplot2 geom_line for performance graphs. Default is 1.5.

Value

ggplot2 plot object.

plotTuneMultiCritResult *Plots multi-criteria results after tuning.*

Description

Visualizes the pareto front and possibly the dominated points.

Usage

```
plotTuneMultiCritResult(res, path = TRUE, col = NULL, shape = NULL,
  pointsize = 2)
```

Arguments

res	[TuneMultiCritResult] Result of tuneParamsMultiCrit.
path	[logical(1)] Visualize all evaluated points (or only the non-dominated pareto front)? For the full path, the size of the points on the front is slightly increased. Default is TRUE.
col	[character(1)] Which column of res$opt.path should be mapped to ggplot2 color? Default is NULL, which means none.
shape	[character(1)] Which column of res$opt.path should be mapped to ggplot2 shape? Default is NULL, which means none.
pointsize	[numeric(1)] Point size for ggplot2 geom_point for data points. Default is 2.

Value

ggplot2 plot object.

See Also

Other tune_multicrit: TuneMultiCritControl, TuneMultiCritControlGrid, TuneMultiCritControlNSGA2, TuneMultiCritControlRandom, makeTuneMultiCritControlGrid, makeTuneMultiCritControlNSGA2, makeTuneMultiCritControlRandom; tuneParamsMultiCrit

Examples

```
# see tuneParamsMultiCrit
```

`predict.WrappedModel` *Predict new data.*

Description

Predict the target variable of new data using a fitted model. What is stored exactly in the [Prediction] object depends on the `predict.type` setting of the Learner. If `predict.type` was set to "prob" probability thresholding can be done calling the `setThreshold` function on the prediction object.

Usage

```
## S3 method for class 'WrappedModel'
predict(object, task, newdata, subset, ...)
```

Arguments

`object`	[WrappedModel] Wrapped model, result of `train`.
`task`	[Task] The task. If this is passed, data from this task is predicted.
`newdata`	[`data.frame`] New observations which should be predicted. Pass this alternatively instead of `task`.
`subset`	[`integer`] Index vector to subset `task` or `newdata`. Default is all data.
`...`	[any] Currently ignored.

Value

`Prediction` .

See Also

Other predict: `getProbabilities`; `setPredictType`

Examples

```
# train and predict
train.set = seq(1, 150, 2)
test.set = seq(2, 150, 2)
model = train("classif.lda", iris.task, subset = train.set)
p = predict(model, newdata = iris, subset = test.set)
print(p)
predict(model, task = iris.task, subset = test.set)

# predict now probabiliies instead of class labels
```

```
lrn = makeLearner("classif.lda", predict.type = "prob")
model = train(lrn, iris.task, subset = train.set)
p = predict(model, task = iris.task, subset = test.set)
print(p)
getProbabilities(p)
```

`Prediction` *Prediction object.*

Description

Result from `predict.WrappedModel`. Use `as.data.frame` to access all information in a convenient format. The function `getProbabilities` is useful to access predicted probabilities.

The `data` member of the object contains always the following columns: `id`, index numbers of predicted cases from the task, `response` either a numeric or a factor, the predicted response values, `truth`, either a numeric or a factor, the true target values. If probabilities were predicted, as many numeric columns as there were classes named `prob.classname`. If standard errors were predicted, a numeric column named `se`.

Object members:

predict.type [`character(1)`] Type set in `setPredictType`.

data [`data.frame`] See details.

threshold [`numeric(1)`] Threshold set in predict function.

task.desc [`TaskDesc`] Task description object.

time [`numeric(1)`] Time learner needed to generate predictions.

`predictLearner` *Predict new data with an R learner.*

Description

Mainly for internal use. Predict new data with a fitted model. You have to implement this method if you want to add another learner to this package.

Usage

```
predictLearner(.learner, .model, .newdata, ...)
```

Arguments

.learner	[RLearner] Wrapped learner.
.model	[WrappedModel] Model produced by training.
.newdata	[data.frame] New data to predict. Does not include target column.
...	[any] Additional parameters, which need to be passed to the underlying predict function.

Details

You implementation must adhere to the following: The model must be fitted on the subset of .task given by .subset. All parameters in ... must be passed to the underlying training function.

Value

For classification: Either a factor for type "response" or a matrix for type "prob". In the latter case the columns must be named with the class labels. For regressions: Either a numeric for type "response" or a matrix with two columns for type "se". In the latter case first column is the estimated response (mean value) and the second column the estimated standard errors.

---

reimpute *Re-impute a data set*

Description

This function accepts a data frame and a imputation description as returned by impute to perform the following actions:

1. Restore dropped columns, setting them to NA
2. Add dummy variables for columns as specified in impute
3. Optionally check factors for new levels to treat them as NAs
4. Reorder factor levels to ensure identical integer representation as before
5. Impute missing values using previously collected data

Usage

```
reimpute(x, desc)
```

Arguments

x	[data.frame] Object to reimpute. Currently only data frames are supported.
desc	[ImputationDesc] Imputation description as returned by impute.

Value

Imputated x.

See Also

Other impute: imputations, imputeConstant, imputeHist, imputeLearner, imputeMax, imputeMean, imputeMedian, imputeMin, imputeMode, imputeNormal, imputeUniform; impute; makeImputeMethod; makeImputeWrapper

---

removeConstantFeatures

Remove constant features from a data set.

Description

Constant features can lead to errors in some models and obviously provide no information in the training set that can be learned from. With the argument "perc", there is a possibility to also remove features for which less than "perc" percent of the observations differ from the mode value.

Usage

```
removeConstantFeatures(x, target, perc = 0, dont.rm = character(0L),
  na.ignore = FALSE, tol = .Machine$double.eps^0.5,
  show.info = getMlrOption("show.info"))
```

Arguments

x	[data.frame | Task] The data set or task.
target	[character] Name of the column(s) specifying the response if you passed a data.frame. User input is ignored if you pass a task and target is automatically set. Never removed.
perc	[numeric(1)] The percentage of a feature values in [0, 1) that must differ from the mode value. Default is 0, which means only constant features with exactly one observed level are removed.
dont.rm	[character] Names of the columns which must not be deleted. Default is no columns.
na.ignore	[logical(1)] Should NAs be ignored in the percentage calculation? (Or should they be treated as a single, extra level in the percentage calculation?) Default is FALSE.
tol	[numeric(1)] Numerical tolerance to treat two numbers as equal. Variables stored as double will get rounded accordingly before computing the mode. Default is sqrt(.Maschine$double.eps).
show.info	[logical(1)] Print verbose output on console? Default is set via configureMlr.

Value

data.frame | Task .

---

removeHyperPars *Remove hyperparameters settings of a learner.*

Description

Remove settings (previously set through mlr) for some parameters. Which means that the default behavior for that param will now be used.

Usage

```
removeHyperPars(learner, ids = character(0L))
```

Arguments

learner	[Learner	character(1)] The learner. If you pass a string the learner will be created via makeLearner.
ids	[character] Parameter names to remove settings for. Default is character(0L).

Value

Learner .

See Also

Other learner: getHyperPars; getParamSet; setHyperPars; setId; setPredictType; showHyperPars

---

ResamplePrediction *Prediction from resampling.*

Description

Contains predictions from resampling, returned (among other stuff) by function resample. Can basically be used in the same way as Prediction, its super class. The main differences are: (a) The internal data.frame (member data) contains an additional column id, specifying the iteration of the resampling strategy, and and additional columns set, specifying whether the prediction was from an observation in the "train" or "test" set. (b) The prediction time is a numeric vector, its length equals the number of iterations.

See Also

Other resample: ResampleDesc, makeResampleDesc; ResampleInstance, makeResampleInstance; bootstrapB632, bootstrapB632plus, bootstrapOOB, crossval, holdout, repcv, resample, subsample

RLearner *Internal construction / wrapping of learner object.*

Description

Wraps an already implemented learning method from R to make it accessible to mlr. Call this method in your constructor. You have to pass an id (name), the required package(s), a description object for all changeable parameters (you dont have to do this for the learner to work, but it is strongly recommended), and use property tags to define features of the learner.

Usage

```
makeRLearner()

makeRLearnerClassif(cl, package, par.set, par.vals = list(),
  properties = character(0L))

makeRLearnerRegr(cl, package, par.set, par.vals = list(),
  properties = character(0L))

makeRLearnerSurv(cl, package, par.set, par.vals = list(),
  properties = character(0L))

makeRLearnerCluster(cl, package, par.set, par.vals = list(),
  properties = character(0L))
```

Arguments

`cl` [character(1)]
Class name for learner to create. By convention, all classification learners start with "classif.", all regression learners with "regr." and all survival learners start with "surv.".

`package` [character]
Package(s) to load for the implementation of the learner.

`properties` [character(1)]
Set of learner properties. Some standard property names include:

numerics Can numeric features be handled?

factors Can factor features be handled?

missings Can missing features be handled?

oneclas,twoclass,multiclass Can one-class, two-class or multi-class classification problems be handled?

prob Can probabilites be predicted?

se Can standard errors be predicted?

Default is `character(0)`.

par.set [ParamSet]
Parameter set of (hyper)parameters and their constraints.

par.vals [list]
Always set hyperparameters to these values when the object is constructed. Useful when default values are missing in the underlying function. The values can later be overwritten when the user sets hyperparameters. Default is empty list.

Value

RLearnerClassif, RLearnerRegr or RLearnerSurv .

---

selectFeatures *Feature selection by wrapper approach.*

Description

Optimizes the features for a classification or regression problem by choosing a variable selection wrapper approach. Allows for different optimization methods, such as forward search or a genetic algorithm. You can select such an algorithm (and its settings) by passing a corresponding control object. For a complete list of implemented algorithms look at the subclasses of [FeatSelControl].

All algorithms operate on a 0-1-bit encoding of candidate solutions. Per default a single bit corresponds to a single feature, but you are able to change this by using the arguments bit.names and bits.to.features. Thus allowing you to switch on whole groups of features with a single bit.

Usage

```
selectFeatures(learner, task, resampling, measures, bit.names, bits.to.features,
  control, show.info = getMlrOption("show.info"))
```

Arguments

learner [Learner | character(1)]
The learner. If you pass a string the learner will be created via makeLearner.

task [Task]
The task.

resampling [ResampleInstance | ResampleDesc]
Resampling strategy for feature selection. If you pass a description, it is instantiated once at the beginning by default, so all points are evaluated on the same training/test sets. If you want to change that behaviour, look at FeatSelControl.

measures [list of Measure | Measure]
Performance measures to evaluate. The first measure, aggregated by the first aggregation function is optimized, others are simply evaluated.

bit.names [character]
Names of bits encoding the solutions. Also defines the total number of bits in the encoding. Per default these are the feature names of the task.

bits.to.features
: [function(x, task)]
Function which transforms an integer-0-1 vector into a character vector of selected features. Per default a value of 1 in the ith bit selects the ith feature to be in the candidate solution.

control
: [see FeatSelControl] Control object for search method. Also selects the optimization algorithm for feature selection.

show.info
: [logical(1)]
Print verbose output on console? Default is set via configureMlr.

Value

FeatSelResult .

See Also

Other featsel: FeatSelControl, FeatSelControlExhaustive, FeatSelControlGA, FeatSelControlRandom, FeatSelControlSequential, makeFeatSelControlExhaustive, makeFeatSelControlGA, makeFeatSelControlRandom, makeFeatSelControlSequential; analyzeFeatSelResult; getFeatSelResult; makeFeatSelWrapper

Examples

```
rdesc = makeResampleDesc("Holdout")
ctrl = makeFeatSelControlSequential(method = "sfs", maxit = NA)
res = selectFeatures("classif.rpart", iris.task, rdesc, control = ctrl)
analyzeFeatSelResult(res)
```

---

setAggregation *Set aggregation function of measure.*

Description

Set how this measure will be aggregated after resampling. To see possible aggregation functions: aggregations.

Usage

```
setAggregation(measure, aggr)
```

Arguments

measure
: [Measure]
Performance measure.

aggr
: [Aggregation]
Aggregation function.

Value

Measure with changed aggregation behaviour.

setHyperPars *Set the hyperparameters of a learner object.*

Description

Set the hyperparameters of a learner object.

Usage

```
setHyperPars(learner, ..., par.vals)
```

Arguments

learner	[Learner	character(1)] The learner. If you pass a string the learner will be created via makeLearner.
...	[any] Named (hyper)parameters with new setting. Alternatively these can be passed using the par.vals argument.
par.vals	[list] Optional list of named (hyper)parameter settings. The arguments in ... take precedence over values in this list.

Value

Learner .

See Also

Other learner: getHyperPars; getParamSet; removeHyperPars; setId; setPredictType; showHyperPars

Examples

```
cl1 = makeLearner("classif.ksvm", sigma = 1)
cl2 = setHyperPars(cl1, sigma = 10, par.vals = list(C = 2))
print(cl1)
# note the now set and altered hyperparameters:
print(cl2)
```

setHyperPars2 *Only exported for internal use.*

Description

Only exported for internal use.

Usage

```
setHyperPars2(learner, par.vals)
```

Arguments

learner	[Learner] The learner.
par.vals	[list] List of named (hyper)parameter settings.

setId *Set the id of a learner object.*

Description

Set the id of a learner object.

Usage

```
setId(learner, id)
```

Arguments

learner	[Learner	character(1)] The learner. If you pass a string the learner will be created via makeLearner.
id	[character(1)] New id for learner.

Value

Learner .

See Also

Other learner: getHyperPars; getParamSet; removeHyperPars; setHyperPars; setPredictType; showHyperPars

setPredictType *Set the type of predictions the learner should return.*

Description

Possible prediction types are: Classification: Labels or class probabilities (including labels). Regression: Numeric or response or standard errors (including numeric response). Survival: Linear predictor or survival probability.

For complex wrappers the predict type is usually also passed down the encapsulated learner in a recursive fashion.

Usage

```
setPredictType(learner, predict.type)
```

Arguments

learner	[Learner	character(1)] The learner. If you pass a string the learner will be created via makeLearner.
predict.type	[character(1)] Classification: "response" or "prob". Regression: "response" or "se". Survival: "response" (linear predictor) or "prob". Default is "response".

Value

Learner .

See Also

Other learner: getHyperPars; getParamSet; removeHyperPars; setHyperPars; setId; showHyperPars

Other predict: getProbabilities; predict.WrappedModel

setThreshold *Set threshold of prediction object.*

Description

Set threshold of prediction object for classification. Creates corresponding discrete class response for the newly set threshold. For binary classification: The positive class is predicted if the probability value exceeds the threshold. For multiclass: Probabilities are divided by corresponding thresholds and the class with maximum resulting value is selected. The result of both are equivalent if in the multi-threshold case the values are greater than 0 and sum to 1.

Usage

```
setThreshold(pred, threshold)
```

Arguments

pred	[Prediction] Prediction object.
threshold	[numeric] Threshold to produce class labels. Has to be a named vector, where names correspond to class labels. Only if pred is a prediction object resulting from binary classification it can be a single numerical threshold for the positive class.

Value

Prediction with changed threshold and corresponding response.

See Also

predict.WrappedModel

Examples

```
## create task and train learner (LDA)
task = makeClassifTask(data = iris, target = "Species")
lrn = makeLearner("classif.lda", predict.type = "prob")
mod = train(lrn, task)

## predict probabilities and compute performance
pred = predict(mod, newdata = iris)
performance(pred, measures = mmce)
head(as.data.frame(pred))
## adjust threshold and predict probabilities again
threshold = c(setosa = 0.4, versicolor = 0.3, virginica = 0.3)
pred = setThreshold(pred, threshold = threshold)
performance(pred, measures = mmce)
head(as.data.frame(pred))
```

---

showHyperPars *Display all possible hyperparameter settings for a learner that mlr knows.*

Description

Useful for a quick overview, also does not foce you to create the learner.

Usage

```
showHyperPars(learner)
```

Arguments

learner	[Learner	character(1)] The learner. If you pass a string the learner will be created via makeLearner.

Value

invisible(NULL) .

See Also

Other learner: getHyperPars; getParamSet; removeHyperPars; setHyperPars; setId; setPredictType

---

smote *Synthetic Minority Oversampling Technique to handle class imbalancy in binary classification.*

Description

In each iteration, samples one minority class element x1, then one of x1's nearest neighbors: x2. Both points are now interpolated / convex-combined, resulting in a new virtual data point x3 for the minority class.

The method handles factor features, too. The gower distance is used for nearest neighbor calculation, see daisy. For interpolation, the new factor level for x3 is sampled from the two given levels of x1 and x2 per feature.

Usage

```
smote(task, rate, nn = 5L)
```

Arguments

task	[Task] The task.
rate	[numeric(1)] Factor to upsample the smaller class. Must be between 1 and Inf, where 1 means no oversampling and 2 would mean doubling the class size.
nn	[integer(1)] Number of nearest neighbors to consider. Default is 5.

Value

Task .

See Also

Other imbalancy: makeOverBaggingWrapper; makeOversampleWrapper, makeUndersampleWrapper; oversample, undersample

subsetTask *Subset data in task.*

Description

Subset data in task.

Usage

```
subsetTask(task, subset, features)
```

Arguments

task	[Task] The task.
subset	[integer] Selected cases. Default is all cases.
features	[character] Selected inputs. Note that target feature is always included in the resulting task, you should not pass it here. Default is all features.

Value

Task . Task with subsetted data.

See Also

Other task: getTaskCosts; getTaskData; getTaskFeatureNames; getTaskFormula, getTaskFormulaAsString; getTaskNFeats; getTaskTargets

Examples

```
task = makeClassifTask(data = iris, target = "Species")
subsetTask(task, subset = 1:100)
```

TaskDesc *Description object for task.*

Description

Description object for task, encapsulates basic properties of the task without having to store the complete data set.

Details

Object members:

id [character(1)] Id string of task.

type [character(1)] Type of task, "classif" for classification, "regr" for regression, "surv" for survival, "costsens" for cost-sensitive classification.

target [character(0) | character(1) | character(2)] Name of target variable. For "surv" these are the names of the survival time and event columns, so it has length 2. For "costsens" ist has length 0, as there is no target column, but a cost matrix instead.

size [integer(1)] Number of cases in data set.

n.feat [integer(2)] Number of features, named vector with entries: "numerics", "factors".

has.missings [logical(1)] Are missing values present?

has.weights [logical(1)] Are weights specified for each observation?

has.blocking [logical(1)] Is a blocking factor for cases available in the task?

class.levels [character] All possible classes. Only present for "classif" and "costsens".

positive [character(1)] Positive class label for binary classification. Only present for "classif", NA for multiclass.

negative [character(1)] Negative class label for binary classification. Only present for "classif", NA for multiclass.

---

train *Train a learning algorithm.*

Description

Given a Task, creates a model for the learning machine which can be used for predictions on new data.

Usage

```
train(learner, task, subset, weights = NULL)
```

Arguments

learner	[Learner	character(1)] The learner. If you pass a string the learner will be created via makeLearner.
task	[Task] The task.
subset	[integer] An index vector specifying the training cases to be used for fitting. By default the complete data set is used.
weights	[numeric] Optional, non-negative case weight vector to be used during fitting. If given, must be of same length as subset and in corresponding order. By default NULL which means no weights are used unless specified in the task ([Task]). Weights from the task will be overwritten.

Value

WrappedModel .

See Also

predict.WrappedModel

Examples

```
training.set = sample(1:nrow(iris), nrow(iris) / 2)

## use linear discriminant analysis to classify iris data
task = makeClassifTask(data = iris, target = "Species")
learner = makeLearner("classif.lda", method = "mle")
mod = train(learner, task, subset = training.set)
print(mod)

## use random forest to classify iris data
task = makeClassifTask(data = iris, target = "Species")
learner = makeLearner("classif.rpart", minsplit = 7, predict.type = "prob")
mod = train(learner, task, subset = training.set)
print(mod)
```

---

trainLearner *Train an R learner.*

Description

Mainly for internal use. Trains a wrapped learner on a given training set. You have to implement this method if you want to add another learner to this package.

Usage

```
trainLearner(.learner, .task, .subset, .weights = NULL, ...)
```

Arguments

.learner	[RLearner] Wrapped learner.
.task	[Task] Task to train learner on.
.subset	[integer] Subset of cases for training set, index the task with this. You probably want to use getTaskData for this purpose.
.weights	[numeric] Weights for each observation.
...	[any] Additional (hyper)parameters, which need to be passed to the underlying train function.

Details

Your implementation must adhere to the following: The model must be fitted on the subset of `.task` given by `.subset`. All parameters must in `...` must be passed to the underlying training function.

Value

any . Model of the underlying learner.

TuneMultiCritResult *Result of multi-criteria tuning.*

Description

Container for results of hyperparameter tuning. Contains the obtained pareto set and front and the optimization path which lead there.

Object members:

learner [`Learner`] Learner that was optimized.

control [`TuneControl`] Control object from tuning.

x [`list`] List of lists of non-dominated hyperparameter settings in pareto set. Note that when you have trafos on some of your params, `x` will always be on the TRANSFORMED scale so you directly use it.

y [`matrix`] Pareto front for `x`.

opt.path [`OptPath`] Optimization path which lead to `x`. Note that when you have trafos on some of your params, the opt.path always contains the UNTRANSFORMED values on the original scale. You can simply call `trafoOptPath(opt.path)` to transform them, or, `as.data.frame{trafoOptPath(opt.pat`

---

tuneParams *Hyperparameter tuning.*

Description

Optimizes the hyperparameters of a learner. Allows for different optimization methods, such as grid search, evolutionary strategies, iterated F-race, etc. You can select such an algorithm (and its settings) by passing a corresponding control object. For a complete list of implemented algorithms look at `TuneControl`.

Multi-criteria tuning can be done with `tuneParamsMultiCrit`.

Usage

```
tuneParams(learner, task, resampling, measures, par.set, control,
  show.info = getMlrOption("show.info"))
```

Arguments

learner	[Learner	character(1)] The learner. If you pass a string the learner will be created via makeLearner.
task	[Task] The task.
resampling	[ResampleInstance	ResampleDesc] Resampling strategy to evaluate points in hyperparameter space. If you pass a description, it is instantiated once at the beginning by default, so all points are evaluated on the same training/test sets. If you want to change that behavior, look at TuneControl.
measures	[list of Measure	Measure] Performance measures to evaluate. The first measure, aggregated by the first aggregation function is optimized, others are simply evaluated.
par.set	[ParamSet] Collection of parameters and their constraints for optimization.
control	[TuneControl] Control object for search method. Also selects the optimization algorithm for tuning.
show.info	[logical(1)] Print verbose output on console? Default is set via configureMlr.

Value

TuneResult .

See Also

Other tune: ModelMultiplexer, makeModelMultiplexer; TuneControl, TuneControlCMAES, TuneControlGenSA, TuneControlGrid, TuneControlIrace, TuneControlRandom, makeTuneControlCMAES, makeTuneControlGenSA, makeTuneControlGrid, makeTuneControlIrace, makeTuneControlRandom; getTuneResult; makeModelMultiplexerPar makeTuneWrapper; tuneThreshold

Examples

```
# a grid search for an SVM (with a tiny number of points...)
# note how easily we can optimize on a log-scale
ps = makeParamSet(
  makeNumericParam("C", lower = -12, upper = 12, trafo = function(x) 2^x),
  makeNumericParam("sigma", lower = -12, upper = 12, trafo = function(x) 2^x)
)
ctrl = makeTuneControlGrid(resolution = 2L)
rdesc = makeResampleDesc("CV", iters = 2L)
res = tuneParams("classif.ksvm", iris.task, rdesc, par.set = ps, control = ctrl)
print(res)
print(as.data.frame(res$opt.path))
print(as.data.frame(trafoOptPath(res$opt.path)))

## Not run:
```

```
# we optimize the SVM over 3 kernels simultanously
# note how we use dependent params (requires = ...) and iterated F-racing here
ps = makeParamSet(
  makeNumericParam("C", lower = -12, upper = 12, trafo = function(x) 2^x),
  makeDiscreteParam("kernel", values = c("vanilladot", "polydot", "rbfdot")),
  makeNumericParam("sigma", lower = -12, upper = 12, trafo = function(x) 2^x,
    requires = quote(kernel == "rbfdot")),
  makeIntegerParam("degree", lower = 2L, upper = 5L,
    requires = quote(kernel == "polydot"))
)
print(ps)
ctrl = makeTuneControlIrace(maxExperiments = 200L)
rdesc = makeResampleDesc("Holdout")
res = tuneParams("classif.ksvm", iris.task, rdesc, par.set = ps, control = ctrl)
print(res)
print(head(as.data.frame(res$opt.path)))

## End(Not run)
```

`tuneParamsMultiCrit` *Hyperparameter tuning for multiple measures at once.*

Description

Optimizes the hyperparameters of a learner in a multi-criteria fasion. Allows for different optimization methods, such as grid search, evolutionary strategies, etc. You can select such an algorithm (and its settings) by passing a corresponding control object. For a complete list of implemented algorithms look at `TuneMultiCritControl`.

Usage

```
tuneParamsMultiCrit(learner, task, resampling, measures, par.set, control,
  show.info = getMlrOption("show.info"))
```

Arguments

`learner`	[`Learner`	`character(1)`] The learner. If you pass a string the learner will be created via `makeLearner`.
`task`	[`Task`] The task.
`resampling`	[`ResampleInstance`	`ResampleDesc`] Resampling strategy to evaluate points in hyperparameter space. If you pass a description, it is instantiated once at the beginning by default, so all points are evaluated on the same training/test sets. If you want to change that behavior, look at `TuneMultiCritControl`.
`par.set`	[`ParamSet`] Collection of parameters and their constraints for optimization.

measures [list of Measure]
Performance measures to optimize simultaneously.

control [TuneMultiCritControl]
Control object for search method. Also selects the optimization algorithm for tuning.

show.info [logical(1)]
Print verbose output on console? Default is set via configureMlr.

Value

TuneMultiCritResult .

See Also

Other tune_multicrit: TuneMultiCritControl, TuneMultiCritControlGrid, TuneMultiCritControlNSGA2, TuneMultiCritControlRandom, makeTuneMultiCritControlGrid, makeTuneMultiCritControlNSGA2, makeTuneMultiCritControlRandom; plotTuneMultiCritResult

Examples

```
# multi-criteria optimization of (tpr, fpr) with NGSA-II
lrn =  makeLearner("classif.ksvm")
rdesc = makeResampleDesc("Holdout")
ps = makeParamSet(
  makeNumericParam("C", lower = -12, upper = 12, trafo = function(x) 2^x),
  makeNumericParam("sigma", lower = -12, upper = 12, trafo = function(x) 2^x)
)
ctrl = makeTuneMultiCritControlNSGA2(popsize = 4L, generations = 5L)
res = tuneParamsMultiCrit(lrn, sonar.task, rdesc, par.set = ps,
  measures = list(tpr, fpr), control = ctrl)
plotTuneMultiCritResult(res, path = TRUE)
```

TuneResult *Result of tuning.*

Description

Container for results of hyperparameter tuning. Contains the obtained point in search space, its performance values and the optimization path which lead there.

Object members:

learner [Learner] Learner that was optimized.

control [TuneControl] Control object from tuning.

x [list] Named list of hyperparameter values identified as optimal. Note that when you have trafos on some of your params, x will always be on the TRANSFORMED scale so you directly use it.

y [numeric] Performance values for optimal x.

opt.path [OptPath] Optimization path which lead to x. Note that when you have trafos on some of your params, the opt.path always contains the UNTRANSFORMED values on the original scale. You can simply call trafoOptPath(opt.path) to transform them, or, as.data.frame{trafoOptPath(opt.pat

---

tuneThreshold *Tune prediction threshold.*

Description

Optimizes the threshold of prediction based on probabilities. Uses optimizeSubInts for 2class problems and cma_es for multiclass problems.

Usage

```
tuneThreshold(pred, measure, task, model, nsub = 20L, control = list())
```

Arguments

pred	[Prediction] Prediction object.
measure	[Measure] Performance measure to optimize.
task	[Task] Learning task. Rarely neeeded, only when required for the performance measure.
model	[WrappedModel] Fitted model. Rarely neeeded, only when required for the performance measure.
nsub	[integer(1)] Passed to optimizeSubInts for 2class problems. Default is 20.
control	[list] Control object for cma_es when used. Default is empty list.

Value

list . A named list with with the following components: th is the optimal threshold, perf the performance value.

See Also

Other tune: ModelMultiplexer, makeModelMultiplexer; TuneControl, TuneControlCMAES, TuneControlGenSA, TuneControlGrid, TuneControlIrace, TuneControlRandom, makeTuneControlCMAES, makeTuneControlGenSA, makeTuneControlGrid, makeTuneControlIrace, makeTuneControlRandom; getTuneResult; makeModelMultiplexerPar makeTuneWrapper; tuneParams

Index