Package ‘mlr’

June 12, 2024

Title Machine Learning in R

Version 2.19.2

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.

License BSD_2_clause + file LICENSE


BugReports https://github.com/mlr-org/mlr/issues

Depends ParamHelpers (>= 1.10), R (>= 3.0.2)

Imports backports (>= 1.1.0), BBmisc (>= 1.11), checkmate (>= 1.8.2),
data.table (>= 1.12.4), ggplot2, methods, parallelMap (>= 1.3),
stats, stringi, survival, utils, XML

Suggests ada, adabag, batchtools, bit64, brnn, bst, C50, care, caret
(>= 6.0-57), class, clue, cluster, ClusterR, clusterSim (>=
0.44-5), cmaes, cowplot, crs, Cubist, deepnet, DiceKriging,
e1071, earth, elasticnet, emoa, evtree, fda.usc, FDbost, FNN,
forecast (>= 8.3), fpc, frbs, FSelector, FSelectorRcpp (>=
0.3.5), gbm, GenSA, ggpuru, glmnet, GPfit, h2o (>= 3.6.0.8),
Hmisc, irace (>= 2.0), kernlab, kknn, klaR, knitr, laGP,
LbilineaR, lintr (>= 1.0.0.9001), MASS, mboost, mco, mda,
memoiise, mlbench, mlr, mlrMBO, modeltools, mRMRe, neuralnet,
nnet, numDeriv, pamr, pander, party, pec, penalized (>=
0.9-47), pls, PMCMRplus, praznik (>= 5.0.0), randomForest,
ranger (>= 0.8.0), rappdirs, refund, rex, rFerns, rgenoud,
rdmarked, Rmpi, ROCR, rotationForest, rpart, RRF, rsm, RSNNS,
rucrdtw, RWeka, sda, sf, smoof, sparseLDA, stepPlr, survAUC,
svglite, testthat, tgp, TH.data, tidyr, tsfeatures, vdiffr, wavelets, xgboost (>= 0.7)

VignetteBuilder knitr
ByteCompile yes
Config/testthat/edition 3
Config/testthat/parallel true
Config/testthat/start-first featsel_plotFilterValues, base_plotResiduals, base_generateHyperParsEffect, tune_tuneIrace, featsel_filters, learners_all*, regr_h2ogbm

Encoding UTF-8
LazyData yes
RoxygenNote 7.3.1

SystemRequirements gdal (optional), geos (optional), proj (optional), udunits (optional), gsl (optional), gmp (optional), glu (optional), jags (optional), mpfr (optional), openmpi (optional)

NeedsCompilation yes

Author Bernd Bischl [aut] (<https://orcid.org/0000-0001-6002-6980>), Michel Lang [aut] (<https://orcid.org/0000-0001-9754-0393>), Lars Kotthoff [aut], Patrick Schratz [aut] (<https://orcid.org/0000-0003-0748-6624>), Julia Schiffner [aut], Jakob Richter [aut], Zachary Jones [aut], Giuseppe Casalicchio [aut] (<https://orcid.org/0000-0001-5324-5966>), Mason Gallo [aut], Jakob Bossek [ctb] (<https://orcid.org/0000-0002-4121-4668>), Erich Studerus [ctb] (<https://orcid.org/0000-0003-4233-0182>), Leonard Judt [ctb], Tobias Kuehn [ctb], Pascal Kerschke [ctb] (<https://orcid.org/0000-0003-2862-1418>), Florian Fendt [ctb], Philipp Probst [ctb] (<https://orcid.org/0000-0001-8402-6790>), Xudong Sun [ctb] (<https://orcid.org/0000-0003-3269-2307>), Janek Thomas [ctb] (<https://orcid.org/0000-0003-4511-6245>), Bruno Vieira [ctb], Laura Beggel [ctb] (<https://orcid.org/0000-0002-8872-8535>), Quay Au [ctb] (<https://orcid.org/0000-0002-5252-8902>), Martin Binder [aut, cre], Florian Pfisterer [ctb], Stefan Coors [ctb], Steve Bronder [ctb], Alexander Engelhardt [ctb], Christoph Molnar [ctb], Annette Spooner [ctb]
Contents

Maintainer  Martin Binder  <mlr.developer@mb706.com>
Repository  CRAN
Date/Publication  2024-06-12 10:50:02 UTC

mlr-package ................................. 8
addRRMeasure ................................ 10
Aggregation ................................ 11
aggregations ............................... 11
agri.task .................................. 12
analyzeFeatSelResult ...................... 12
asROCRPrediction ............................ 13
batchmark .................................. 13
bc.task .................................... 15
benchmark .................................. 15
BenchmarkResult ......................... 17
bh.task ..................................... 18
cache_helpers ............................... 18
calculateConfusionMatrix .................. 19
calculateROCMeasures ...................... 20
capLargeValues ............................. 22
configureMlr ............................... 23
ConfusionMatrix ............................ 25
convertBMRToRankMatrix ................... 26
convertMLBenchObjToTask .................. 27
costiris.task .............................. 27
createDummyFeatures ....................... 28
createSpatialResamplingPlots ............. 29
crossover .................................. 32
downsample ................................ 32
dropFeatures ............................... 33
estimateRelativeOverfitting ............. 34
estimateResidualVariance ................. 35
extractFDABsignal .......................... 36
extractFDADTWKernel ...................... 36
extractFDAFeatures ....................... 37
extractFDAFourier ......................... 39
extractFDAFPCA ............................ 39
extractFDAMultiResFeatures .............. 40
extractFDATsfeatures ...................... 41
extractFDAWavelets ....................... 42
FailureModel .............................. 43
FeatSelControl ............................. 44
FeatSelResult ............................. 47
filterFeatures ............................. 48
friedmanPostHocTestBMR ................... 50
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>friedmanTestBMR</td>
<td>51</td>
</tr>
<tr>
<td>fuelsubset.task</td>
<td>52</td>
</tr>
<tr>
<td>generateCalibrationData</td>
<td>52</td>
</tr>
<tr>
<td>generateCritDifferencesData</td>
<td>54</td>
</tr>
<tr>
<td>generateFeatureImportanceData</td>
<td>56</td>
</tr>
<tr>
<td>generateFilterValuesData</td>
<td>58</td>
</tr>
<tr>
<td>generateHyperParsEffectData</td>
<td>60</td>
</tr>
<tr>
<td>generateLearningCurveData</td>
<td>61</td>
</tr>
<tr>
<td>generatePartialDependenceData</td>
<td>63</td>
</tr>
<tr>
<td>generateThreshVsPerfData</td>
<td>66</td>
</tr>
<tr>
<td>getBMRAggrPerformances</td>
<td>67</td>
</tr>
<tr>
<td>getBMRFeatSelResults</td>
<td>68</td>
</tr>
<tr>
<td>getBMRFilteredFeatures</td>
<td>70</td>
</tr>
<tr>
<td>getBMRLearnerIds</td>
<td>71</td>
</tr>
<tr>
<td>getBMRLearners</td>
<td>72</td>
</tr>
<tr>
<td>getBMRLearnerShortNames</td>
<td>72</td>
</tr>
<tr>
<td>getBMRLearnerShortNames</td>
<td>72</td>
</tr>
<tr>
<td>getBMRLearnerIds</td>
<td>73</td>
</tr>
<tr>
<td>getBMLMarks</td>
<td>74</td>
</tr>
<tr>
<td>getBMRLearners</td>
<td>74</td>
</tr>
<tr>
<td>getBMRLearnerShortNames</td>
<td>74</td>
</tr>
<tr>
<td>getBMRLearnerShortNames</td>
<td>74</td>
</tr>
<tr>
<td>getBMRLearnerIds</td>
<td>75</td>
</tr>
<tr>
<td>getBMRLearnerShortNames</td>
<td>75</td>
</tr>
<tr>
<td>getBMRLearnerIds</td>
<td>76</td>
</tr>
<tr>
<td>getBMRLearnerShortNames</td>
<td>76</td>
</tr>
<tr>
<td>getBMRLearnerIds</td>
<td>78</td>
</tr>
<tr>
<td>getBMRLearnerShortNames</td>
<td>78</td>
</tr>
<tr>
<td>getBMRLearnerIds</td>
<td>78</td>
</tr>
<tr>
<td>getBMRLearnerShortNames</td>
<td>79</td>
</tr>
<tr>
<td>getBMRLearnerIds</td>
<td>79</td>
</tr>
<tr>
<td>getBMRLearnerIds</td>
<td>80</td>
</tr>
<tr>
<td>getBMRLearnerShortNames</td>
<td>80</td>
</tr>
<tr>
<td>getBMRLearnerIds</td>
<td>81</td>
</tr>
<tr>
<td>getBMRLearnerShortNames</td>
<td>81</td>
</tr>
<tr>
<td>getBMRLearnerIds</td>
<td>82</td>
</tr>
<tr>
<td>getBMRLearnerShortNames</td>
<td>82</td>
</tr>
<tr>
<td>getBMRLearnerIds</td>
<td>83</td>
</tr>
<tr>
<td>getBMRLearnerShortNames</td>
<td>83</td>
</tr>
<tr>
<td>getBMRLearnerIds</td>
<td>84</td>
</tr>
<tr>
<td>getBMRLearnerShortNames</td>
<td>84</td>
</tr>
<tr>
<td>getBMRLearnerIds</td>
<td>84</td>
</tr>
<tr>
<td>getBMRLearnerShortNames</td>
<td>85</td>
</tr>
<tr>
<td>getBMRLearnerIds</td>
<td>85</td>
</tr>
<tr>
<td>getBMRLearnerShortNames</td>
<td>85</td>
</tr>
<tr>
<td>getBMRLearnerIds</td>
<td>86</td>
</tr>
<tr>
<td>getBMRLearnerShortNames</td>
<td>86</td>
</tr>
<tr>
<td>getBMRLearnerIds</td>
<td>87</td>
</tr>
<tr>
<td>getBMRLearnerShortNames</td>
<td>87</td>
</tr>
<tr>
<td>getBMRLearnerIds</td>
<td>88</td>
</tr>
<tr>
<td>getBMRLearnerShortNames</td>
<td>88</td>
</tr>
<tr>
<td>getBMRLearnerIds</td>
<td>89</td>
</tr>
<tr>
<td>getBMRLearnerShortNames</td>
<td>89</td>
</tr>
<tr>
<td>getBMRLearnerIds</td>
<td>89</td>
</tr>
<tr>
<td>getBMRLearnerShortNames</td>
<td>90</td>
</tr>
<tr>
<td>getBMRLearnerIds</td>
<td>91</td>
</tr>
<tr>
<td>getBMRLearnerShortNames</td>
<td>91</td>
</tr>
<tr>
<td>getBMRLearnerIds</td>
<td>92</td>
</tr>
<tr>
<td>getBMRLearnerShortNames</td>
<td>92</td>
</tr>
<tr>
<td>getBMRLearnerIds</td>
<td>92</td>
</tr>
<tr>
<td>getBMRLearnerShortNames</td>
<td>93</td>
</tr>
<tr>
<td>getBMRLearnerIds</td>
<td>93</td>
</tr>
<tr>
<td>getBMRLearnerShortNames</td>
<td>94</td>
</tr>
<tr>
<td>getBMRLearnerIds</td>
<td>94</td>
</tr>
<tr>
<td>getBMRLearnerShortNames</td>
<td>94</td>
</tr>
<tr>
<td>getBMRLearnerIds</td>
<td>95</td>
</tr>
<tr>
<td>getBMRLearnerShortNames</td>
<td>95</td>
</tr>
<tr>
<td>getBMRLearnerIds</td>
<td>96</td>
</tr>
<tr>
<td>getBMMLearnerIds</td>
<td>96</td>
</tr>
</tbody>
</table>
Contents

getNestedTuneResultsOptPathDf ........................................ 97
getNestedTuneResultsX ................................................. 98
getOOBPreds ............................................................... 98
getParamSet ............................................................... 99
getPredictionDump ......................................................... 100
getPredictionProbabilities ................................................ 100
getPredictionResponse ..................................................... 101
getPredictionTaskDesc .................................................... 102
getProbabilities ............................................................ 103
getResamplingIndices ..................................................... 103
getRRDump ................................................................. 104
getRRPredictionList ....................................................... 105
getRRPredictions ......................................................... 105
getRRTaskDesc ............................................................. 106
getRRTaskDescription .................................................... 107
getStackedBaseLearnerPredictions ........................................ 107
getTaskClassLevels ....................................................... 108
getTaskCosts ................................................................. 108
getTaskData ................................................................. 109
getTaskDesc ................................................................. 110
getTaskDescription ....................................................... 111
getTaskFeatureNames ...................................................... 111
getTaskFormula ............................................................. 112
getTaskId ..................................................................... 113
getTaskNFeats ............................................................... 113
getTaskSize ................................................................. 114
getTaskTargetNames ....................................................... 114
getTaskTargets ............................................................. 115
getTaskType ................................................................. 116
getTuneResult .............................................................. 116
getTuneResultOptPath ..................................................... 117
gunpoint.task .............................................................. 117
hasFunctionalFeatures ..................................................... 118
hasProperties ............................................................... 118
helpLearner ................................................................. 119
helpLearnerParam .......................................................... 119
imputations ................................................................. 120
impute ........................................................................ 122
iris.task ................................................................... 124
isFailureModel ............................................................. 124
joinClassLevels ............................................................ 125
learnerArgsToControl ...................................................... 125
LearnerProperties .......................................................... 126
learners ...................................................................... 127
listFilterEnsembleMethods ................................................ 127
listFilterMethods ........................................................ 128
listLearnerProperties ..................................................... 129
listLearners ............................................................... 129
Contents

makeRLearner.classif.fdausc.kernel .............................................. 189
makeRLearner.classif.fdausc.np .................................................... 190
makeSMOTEWrapper ................................................................. 190
makeStackedLearner ................................................................. 191
makeSurvTask ........................................................................... 194
makeTuneControlCMAES ............................................................... 195
makeTuneControlDesign ............................................................. 197
makeTuneControlGenSA ................................................................. 198
makeTuneControlIRace ................................................................. 202
makeTuneControlMBO ................................................................. 204
makeTuneControlRandom .............................................................. 206
makeTuneWrapper ..................................................................... 207
makeUndersampleWrapper ............................................................ 209
makeWeightedClassesWrapper ....................................................... 210
makeWrappedModel ................................................................. 212
MeasureProperties ........................................................................ 213
measures ..................................................................................... 214
mergeBenchmarkResults ............................................................... 217
mergeSmallFactorLevels ............................................................... 218
mlrFamilies ................................................................................. 219
mtcars.task ................................................................................. 220
normalizeFeatures ....................................................................... 221
oversample .................................................................................. 222
parallelization ............................................................................ 223
performance ............................................................................... 224
phoneme.task .............................................................................. 225
pid.task ...................................................................................... 225
plotBMRBoxplots ....................................................................... 226
plotBMRRanksAsBarChart ............................................................. 227
plotBMRSummary ....................................................................... 228
plotCalibration ........................................................................... 230
plotCritDifferences ..................................................................... 231
plotFilterValues ......................................................................... 232
plotHyperParsEffect ................................................................... 233
plotLearnerPrediction .................................................................. 236
plotLearningCurve ....................................................................... 238
plotPartialDependence ............................................................... 239
plotResiduals ............................................................................. 240
plotROCCurves .......................................................................... 241
plotThresholdVsPerf ................................................................. 242
plotTuneMultiCritResult ............................................................. 244
predict.WrappedModel ............................................................... 245
predictLearner ........................................................................... 246
reduceBatchmarkResults ............................................................. 247
reextractFDAFeatures ................................................................. 248
reimpute ..................................................................................... 249
removeConstantFeatures ............................................................. 250
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>removeHyperPars</td>
<td>251</td>
</tr>
<tr>
<td>resample</td>
<td>252</td>
</tr>
<tr>
<td>ResamplePrediction</td>
<td>256</td>
</tr>
<tr>
<td>ResampleResult</td>
<td>257</td>
</tr>
<tr>
<td>R Learner</td>
<td>258</td>
</tr>
<tr>
<td>selectFeatures</td>
<td>260</td>
</tr>
<tr>
<td>setAggregation</td>
<td>262</td>
</tr>
<tr>
<td>setHyperPars</td>
<td>263</td>
</tr>
<tr>
<td>setHyperPars2</td>
<td>264</td>
</tr>
<tr>
<td>setId</td>
<td>264</td>
</tr>
<tr>
<td>setLearnerId</td>
<td>265</td>
</tr>
<tr>
<td>setMeasurePars</td>
<td>266</td>
</tr>
<tr>
<td>setPredictThreshold</td>
<td>266</td>
</tr>
<tr>
<td>setPredictType</td>
<td>267</td>
</tr>
<tr>
<td>setThreshold</td>
<td>268</td>
</tr>
<tr>
<td>simplifyMeasureNames</td>
<td>269</td>
</tr>
<tr>
<td>smote</td>
<td>270</td>
</tr>
<tr>
<td>sonar.task</td>
<td>271</td>
</tr>
<tr>
<td>spam.task</td>
<td>271</td>
</tr>
<tr>
<td>spatial.task</td>
<td>271</td>
</tr>
<tr>
<td>subsetTask</td>
<td>272</td>
</tr>
<tr>
<td>summarizeColumns</td>
<td>273</td>
</tr>
<tr>
<td>summarizeLevels</td>
<td>274</td>
</tr>
<tr>
<td>Task</td>
<td>274</td>
</tr>
<tr>
<td>TaskDesc</td>
<td>276</td>
</tr>
<tr>
<td>train</td>
<td>277</td>
</tr>
<tr>
<td>trainLearner</td>
<td>278</td>
</tr>
<tr>
<td>TuneControl</td>
<td>279</td>
</tr>
<tr>
<td>TuneMultiCritControl</td>
<td>280</td>
</tr>
<tr>
<td>TuneMultiCritResult</td>
<td>284</td>
</tr>
<tr>
<td>tuneParams</td>
<td>284</td>
</tr>
<tr>
<td>tuneParamsMultiCrit</td>
<td>287</td>
</tr>
<tr>
<td>TuneResult</td>
<td>288</td>
</tr>
<tr>
<td>tuneThreshold</td>
<td>289</td>
</tr>
<tr>
<td>wpbc.task</td>
<td>290</td>
</tr>
<tr>
<td>yeast.task</td>
<td>290</td>
</tr>
</tbody>
</table>

**Index**

---

**mlr-package**

*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.

Author(s)

Maintainer: Martin Binder <mlr.developer@mb706.com>
Authors:

• Bernd Bischl <bernd_bischl@gmx.net> (ORCID)
• Michel Lang <michellang@gmail.com> (ORCID)
• Lars Kotthoff <larsko@uwyo.edu>
• Patrick Schratz <patrick.schratz@gmail.com> (ORCID)
• Julia Schiffner <schiffner@math.uni-duesseldorf.de>
• Jakob Richter <code@jakob-r.de>
• Zachary Jones <zmj@zmjones.com>
• Giuseppe Casalicchio <giuseppe.casalicchio@stat.uni-muenchen.de> (ORCID)
• Mason Gallo <masonagallo@gmail.com>

Other contributors:

• Jakob Bossek <jakob.bossek@tu-dortmund.de> (ORCID) [contributor]
• Erich Studerus <erich.studerus@upkbs.ch> (ORCID) [contributor]
• Leonard Judt <leonard.judt@tu-dortmund.de> [contributor]
• Tobias Kuehn <tobi.kuehn@gmx.de> [contributor]
• Pascal Kerschke <kerschke@uni-muenster.de> (ORCID) [contributor]
• Florian Fendt <flo_fendt@gmx.de> [contributor]
• Philipp Probst <philipp.probst@gmx.de> (ORCID) [contributor]
• Xudong Sun <xudong.sun@stat.uni-muenchen.de> (ORCID) [contributor]
• Janek Thomas <janek.thomas@stat.uni-muenchen.de> (ORCID) [contributor]
• Bruno Vieira <bruno.hebling.vieira@usp.br> [contributor]
• Laura Beggel <laura.beggel@web.de> (ORCID) [contributor]
• Quay Au <quay.au@stat.uni-muenchen.de> (ORCID) [contributor]
• Florian Pfisterer <pfistererf@googlemail.com> [contributor]
• Stefan Coors <stefan.coors@gmx.net> [contributor]
• Steve Bronder <sab2287@columbia.edu> [contributor]
• Alexander Engelhardt <alexander.w.engelhardt@gmail.com> [contributor]
• Christoph Molnar <christoph.molnar@stat.uni-muenchen.de> [contributor]
• Annette Spooner <a.spooner@unsw.edu.au> [contributor]
See Also

Useful links:

- https://mlr.mlr-org.com
- https://github.com/mlr-org/mlr
- Report bugs at https://github.com/mlr-org/mlr/issues

---

addRRMeasure  
*Compute new measures for existing ResampleResult*

### Description

Adds new measures to an existing ResampleResult.

### Usage

```r
addRRMeasure(res, measures)
```

### Arguments

- `res` *(ResampleResult)*  
The result of `resample` run with `keep.pred = TRUE`.

- `measures` *(Measure | list of Measure)*  
Performance measure(s) to evaluate. Default is the default measure for the task, see here `getDefaultMeasure`.

### Value

*(ResampleResult)*.

### See Also

Other resample: `ResamplePrediction`, `ResampleResult`, `getRRPredictionList()`, `getRRPredictions()`, `getRRTaskDesc()`, `getRRTaskDescription()`, `makeResampleDesc()`, `makeResampleInstance()`, `resample()`
Aggregation

Description

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.

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.

Object members:

id (character(1)) Name of the aggregation method.
name (character(1)) Long name of the aggregation method.
properties (character) Properties of the aggregation.
fun (‘function(task, perf.test, perf.train, measure, group, pred)’) Aggregation function.

See Also

makeAggregation

aggregations

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.
analyzeFeatSelResult

testgroup.mean  Performance values on test sets are grouped according to resampling method. The mean for every group is calculated, then the mean of those means. Mainly used for repeated CV.

testgroup.sd  Similar to testgroup.mean - after the mean for every group is calculated, the standard deviation of those means is obtained. Mainly used for repeated CV.

test.join  Performance measure on joined test sets. This is especially useful for small sample sizes where unbalanced group sizes have a significant impact on the aggregation, especially for cross-validation. test.join might make sense now. For the repeated CV, the performance is calculated on each repetition and then aggregated with the arithmetic mean.

See Also

Aggregation

agri.task  European Union Agricultural Workforces clustering task.

Description

Contains the task (agri.task).

References

See cluster::agriculture.

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.
asROCRPrediction

Value

(invisible(NULL)).

See Also

Other featsel: FeatSelControl, 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.

See Also

Other roc: calculateROCMeasures()
Other predict: getPredictionProbabilities(), getPredictionResponse(), getPredictionTaskDesc(), predict.WrappedModel(), setPredictThreshold(), setPredictType()

batchmark

Run machine learning benchmarks as distributed experiments.

Description

This function is a very parallel version of benchmark using batchtools. Experiments are created in the provided registry for each combination of learners, tasks and resamplings. The experiments are then stored in a registry and the runs can be started via batchtools::submitJobs. A job is one train/test split of the outer resampling. In case of nested resampling (e.g. with makeTuneWrapper), each job is a full run of inner resampling, which can be parallelized in a second step with ParallelMap.

For details on the usage and support backends have a look at the batchtools tutorial page: https://github.com/mllg/batchtools.

The general workflow with batchmark looks like this:

1. Create an ExperimentRegistry using batchtools::makeExperimentRegistry.
2. Call `batchmark(...)` which defines jobs for all learners and tasks in a `base::expand.grid` fashion.
3. Submit jobs using `batchtools::submitJobs`.
4. Babysit the computation, wait for all jobs to finish using `batchtools::waitForJobs`.
5. Call `reduceBatchmarkResult()` to reduce results into a `BenchmarkResult`.

If you want to use this with OpenML datasets you can generate tasks from a vector of dataset IDs easily with `tasks = lapply(data.ids, function(x) convertOMLDataSetToMlr(getOMLDataSet(x)))`.

**Usage**

```r
batchmark(
  learners,
  tasks,
  resamplings,
  measures,
  keep.pred = TRUE,
  keep.extract = FALSE,
  models = FALSE,
  reg = batchtools::getDefaultRegistry()
)
```

**Arguments**

- **learners** (list of Learner | character)
  Learning algorithms which should be compared, can also be a single learner. If you pass strings the learners will be created via `makeLearner`.

- **tasks** list of Task
  Tasks that learners should be run on.

- **resamplings** ([list of) ResampleDesc)
  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.

- **keep.pred** (logical(1))
  Keep the prediction data in the `pred` slot of the result object. If you do many experiments (on larger data sets) these objects might unnecessarily increase object size / mem usage, if you do not really need them. The default is set to TRUE.

- **keep.extract** (logical(1))
  Keep the `extract` slot of the result object. When creating a lot of benchmark results with extensive tuning, the resulting R objects can become very large in size. That is why the tuning results stored in the `extract` slot are removed by default (`keep.extract = FALSE`). Note that when `keep.extract = FALSE` you will not be able to conduct analysis in the tuning results.

- **models** (logical(1))
  Should all fitted models be stored in the `ResampleResult`? Default is FALSE.
**bc.task**

*Wisconsin Breast Cancer classification task.*

**Description**

Contains the task (bc.task).

**References**

See `mlbench::BreastCancer`. The column "Id" and all incomplete cases have been removed from the task.

**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. Furthermore, you can of course pass "enhanced" learners via wrappers, e.g., a learner can be automatically tuned using `makeTuneWrapper`.  

**reg**

(batchtools::Registry)

Registry, created by `batchtools::makeExperimentRegistry`. If not explicitly passed, uses the last created registry.

**Value**

(data.table). Generated job ids are stored in the column “job.id”.

**See Also**

Other benchmark: `BenchmarkResult`, `benchmark()`, `friedmanPostHocTestBMR()`, `friedmanTestBMR()`, `generateCritDifferencesData()`, `getBMRAggrPerformances()`, `getBMRFeatSelResults()`, `getBMRFilteredFeatures()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`, `getBMRFeatureIds()`, `getBMRFeatureShortNames()`
Usage

benchmark(
  learners,
  tasks,
  resamplings,
  measures,
  keep.pred = TRUE,
  keep.extract = FALSE,
  models = FALSE,
  show.info = getMlrOption("show.info")
)

Arguments

learners (list of Learner | character)
Learning algorithms which should be compared, can also be a single learner. If
you pass strings the learners will be created via makeLearner.

tasks list of Task
Tasks that learners should be run on.

resamplings (list of ResampleDesc | ResampleInstance)
Resampling strategy for each task. 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.

keep.pred (logical(1))
Keep the prediction data in the pred slot of the result object. If you do many ex-
periments (on larger data sets) these objects might unnecessarily increase object
size / mem usage, if you do not really need them. The default is set to TRUE.

keep.extract (logical(1))
Keep the extract slot of the result object. When creating a lot of benchmark
results with extensive tuning, the resulting R objects can become very large in
size. That is why the tuning results stored in the extract slot are removed by
default (keep.extract = FALSE). Note that when keep.extract = FALSE you
will not be able to conduct analysis in the tuning results.

models (logical(1))
Should all fitted models be stored in the ResampleResult? Default is FALSE.

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

Value

BenchmarkResult.
BenchmarkResult

See Also

Other benchmark: BenchmarkResult.batchmark(), convertBMRToRankMatrix(), friedmanPostHocTestBMR(), friedmanTestBMR(), generateCritDifferencesData(), getBMRAggrPerformances(), getBMRFeatSelResults(), getBMRFilteredFeatures(), getBMRLearnerIds(), getBMRLearnerShortNames(), getBMRLearners(), getBMRMeasureIds(), getBMRMeasures(), getBMRModels(), getBMRPerformances(), getBMRpredictions(), getBMRTaskDescs(), getBMRTaskIds(), getBMRTuneResults(), plotBMRboxplots(), plotBMRRanksAsBarChart(), plotBMRSummary(), plotCritDifferences(), reduceBatchmarkResults()

Examples

```r
lrns = list(makeLearner("classif.lda"), makeLearner("classif.rpart"))
tasks = list(iris.task, sonar.task)
rdesc = makeResampleDesc("CV", iters = 2L)
meas = list(acc, ber)
bmr = benchmark(lrns, tasks, rdesc, measures = meas)
rmat = convertBMRToRankMatrix(bmr)
print(rmat)
plotBMRSUMmary(bmr)
plotBMRBoxplots(bmr, ber, style = "violin")
plotBMRRanksAsBarChart(bmr, pos = "stack")
friedmanTestBMR(bmr)
friedmanPostHocTestBMR(bmr, p.value = 0.05)
```

BenchmarkResult

**BenchmarkResult object.**

Description

Result of a benchmark experiment conducted by benchmark with the following members:

- **results** (list of ResampleResult): A nested list of resample results, first ordered by task id, then by learner id.
- **measures** (list of Measure): The performance measures used in the benchmark experiment.
- **learners** (list of Learner): The learning algorithms compared in the benchmark experiment.

The print method of this object shows aggregated performance values for all tasks and learners.

It is recommended to retrieve required information via the getBMR* getter functions. You can also convert the object using as.data.frame.
bh.task  

*Boston Housing* regression task.

**Description**

Contains the task (bh.task).

**References**

See `mlbench::BostonHousing`.

---

**cache_helpers**  

*Get or delete mlr cache directory*

**Description**

Helper functions to deal with mlr caching.

**Usage**

```r
getCacheDir()
deleteCacheDir()
```

**Details**

`getCacheDir()` returns the default mlr cache directory
`deleteCacheDir()` clears the default mlr cache directory. Custom cache directories must be deleted by hand.
**calculateConfusionMatrix**

*Confusion matrix.*

**Description**

Calculates the confusion matrix for a (possibly resampled) prediction. Rows indicate true classes, columns predicted classes. The marginal elements count the number of classification errors for the respective row or column, i.e., the number of errors when you condition on the corresponding true (rows) or predicted (columns) class. The last bottom right element displays the total amount of errors.

A list is returned that contains multiple matrices. If `relative = TRUE` we compute three matrices, one with absolute values and two with relative. The relative confusion matrices are normalized based on rows and columns respectively, if `FALSE` we only compute the absolute value matrix.

The print function returns the relative matrices in a compact way so that both row and column marginals can be seen in one matrix. For details see `ConfusionMatrix`.

Note that for resampling no further aggregation is currently performed. All predictions on all test sets are joined to a vector `yhat`, as are all labels joined to a vector `y`. Then `yhat` is simply tabulated vs. `y`, as if both were computed on a single test set. This probably mainly makes sense when cross-validation is used for resampling.

**Usage**

```r
calculateConfusionMatrix(pred, relative = FALSE, sums = FALSE, set = "both")
```

### S3 method for class 'ConfusionMatrix'

```r
print(x, both = TRUE, digits = 2, ...)
```

**Arguments**

- `pred` *(Prediction)*
  Prediction object.

- `relative` *(logical(1))*
  If `TRUE` two additional matrices are calculated. One is normalized by rows and one by columns.

- `sums` *(logical(1))*
  If `TRUE` add absolute number of observations in each group.

- `set` *(character(1))*
  Specifies which part(s) of the data are used for the calculation. If `set` equals `train` or `test`, the `pred` object must be the result of a resampling, otherwise an error is thrown. Defaults to “both”. Possible values are “train”, “test”, or “both”.

- `x` *(ConfusionMatrix)*
  Object to print.

- `both` *(logical(1))*
  If `TRUE` both the absolute and relative confusion matrices are printed.
Calculate receiver operator measures.

Calculate the absolute number of correct/incorrect classifications and the following evaluation measures:

- **tpr** True positive rate (Sensitivity, Recall)
- **fpr** False positive rate (Fall-out)
- **fnr** False negative rate (Miss rate)
- **tnr** True negative rate (Specificity)
calculateROCMeasures

- ppv Positive predictive value (Precision)
- for False omission rate
- lrp Positive likelihood ratio (LR+)
- fdr False discovery rate
- npv Negative predictive value
- acc Accuracy
- lrn Negative likelihood ratio (LR-)
- dor Diagnostic odds ratio

For details on the used measures see measures and also https://en.wikipedia.org/wiki/Receiver_operating_characteristic.

The element for the false omission rate in the resulting object is not called for but fomr since for should never be used as a variable name in an object.

Usage

calculateROCMeasures(pred)

## S3 method for class 'ROCMeasures'
print(x, abbreviations = TRUE, digits = 2, ...)

Arguments

- pred (Prediction)
  Prediction object.
- x (ROCMeasures)
  Created by calculateROCMeasures.
- abbreviations (logical(1))
  If TRUE a short paragraph with explanations of the used measures is printed additionally.
- digits (integer(1))
  Number of digits the measures are rounded to.
- ... (any)
  Currently not used.

Value

(ROCMeasures). A list containing two elements confusion.matrix which is the 2 times 2 confusion matrix of absolute frequencies and measures, a list of the above mentioned measures.

Functions

- print(ROCMeasures):
See Also

Other roc: asROCRPrediction()
Other performance: ConfusionMatrix, calculateConfusionMatrix(), estimateRelativeOverfitting(), makeCostMeasure(), makeCustomResampledMeasure(), makeMeasure(), measures.performance(), setAggregation(), setMeasurePars()

Examples

lrn = makeLearner("classif.rpart", predict.type = "prob")
fit = train(lrn, sonar.task)
pred = predict(fit, task = sonar.task)
calculateROCMeasures(pred)

---

capLargeValues

Convert large/infinite numeric values in a data.frame or task.

Description

Convert numeric entries which large/infinite (absolute) values in a data.frame or task. Only numeric/integer columns are affected.

Usage

capLargeValues(
  obj,
  target = character(0L),
  cols = NULL,
  threshold = Inf,
  impute = threshold,
  what = "abs"
)

Arguments

obj (data.frame | Task)
Input data.

target (character)
Name of the column(s) specifying the response. Target columns will not be capped. Default is character(0).

cols (character)
Which columns to convert. Default is all numeric columns.

threshold (numeric(1))
Threshold for capping. Every entry whose absolute value is equal or larger is converted. Default is Inf.
configureMlr

configureMlr

impute (numeric(1))
Replacement value for large entries. Large negative entries are converted to
-impute. Default is threshold.

what (character(1))
What kind of entries are affected? “abs” means abs(x) > threshold, “pos”
means abs(x) > threshold && x > 0, “neg” means abs(x) > threshold && x < 0. Default is “abs”.

Value

(data.frame)

See Also

Other eda_and_preprocess: createDummyFeatures(), dropFeatures().mergeSmallFactorLevels(),
normalizeFeatures(), removeConstantFeatures().summarizeColumns().summarizeLevels()

Examples

capLargeValues(iris, threshold = 5, impute = 5)

configureMlr

configureMlr

Configure 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,
on.par.out.of.bounds,
on.measure.not.applicable,
show.learner.output,
on.error.dump
)

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”.

on.par.out.of.bounds (character(1))
What should happen if a parameter of a learner is set to an out of bounds value.
“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”.

on.measure.not.applicable (logical(1))
What should happen if a measure is not applicable to a learner.
“stop”: R exception is generated.
“warn”: Warning, but value of the measure will be NA.
“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.
on.error.dump  (logical(1))  
Specify whether FailureModel models and failed predictions should contain an error dump that can be used with debugger to inspect an error. This option is only effective if \texttt{on.learner.error} is “warn” or “quiet”. If it is \texttt{TRUE}, the dump can be accessed using \texttt{getFailureModelDump} on the FailureModel, \texttt{getPredictionDump} on the failed prediction, and \texttt{getRRDump} on resample predictions. Default is \texttt{FALSE}.

Value

\texttt{(invisible(NULL))}.

See Also

Other configure: \texttt{getMlrOptions()}
Convert BenchmarkResult to a rank-matrix.

Description

Computes a matrix of all the ranks of different algorithms over different datasets (tasks). Ranks are computed from aggregated measures. Smaller ranks imply better methods, so for measures that are minimized, small ranks imply small scores. For measures that are maximized, small ranks imply large scores.

Usage

```r
convertBMRToRankMatrix(
  bmr,
  measure = NULL,
  ties.method = "average",
  aggregation = "default"
)
```

Arguments

- `bmr` (BenchmarkResult): Benchmark result.
- `measure` (Measure): Performance measure. Default is the first measure used in the benchmark experiment.
- `ties.method` (character(1)): See `base::rank` for details.
- `aggregation` (character(1)): “mean” or “default”. See `getBMRAggrPerformances` for details on “default”.

Value

(matrix) with measure ranks as entries. The matrix has one row for each learner, and one column for each task.

See Also

Other benchmark: `BenchmarkResult`, `batchmark()`, `benchmark()`, `friedmanPostHocTestBMR()`, `friedmanTestBMR()`, `generateCritDifferencesData()`, `getBMRAggrPerformances()`, `getBMRFeatSelResults()`, `getBMRFilteredFeatures()`, `getBMRLearnerIds()`, `getBMRLearnerShortNames()`, `getBMRLearners()`, `getBMRCritDifferences()`, `getBMRCritDifferencesData()`, `getBMRFixedPerformances()`, `getBMRFixedPredictions()`, `getBMRFmemo()`, `getBMRTaskDescs()`, `getBMRTaskIds()`, `getBMRTuneResults()`, `plotBMBoxplots()`, `plotBMRRanksAsBarChart()`, `plotBMRSummary()`, `plotCritDifferences()`, `reduceBatchmarkResults()`
Examples

# see benchmark

```r
print(convertMLBenchObjToTask("Ionosphere"))
print(convertMLBenchObjToTask("mlbench.spirals", n = 100, sd = 0.1))
```

Description

We auto-set the target column, drop any column which is called “Id” and convert logicals to factors.

Usage

```r
convertMLBenchObjToTask(x, n = 100L, ...)
```

Arguments

- `x` (character(1))
  Name of an mlbench function or dataset.
- `n` (integer(1))
  Number of observations for data simul functions. Note that for a few mlbench function this setting is not exactly respected by mlbench. Default is 100.
- `...` (any)
  Passed on to data simul functions.

Examples

```r
print(convertMLBenchObjToTask("Ionosphere"))
print(convertMLBenchObjToTask("mlbench.spirals", n = 100, sd = 0.1))
```

(costiris.task) Iris cost-sensitive classification task.

Description

Contains the task (costiris.task).

References

See `datasets::iris`. The cost matrix was generated artificially following

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

Arguments
- **obj**  
  (data.frame | Task)  
  Input data.
- **target**  
  (character(1) | character(2) | character(n.classes))  
  Name(s) of the target variable(s). Only used when `obj` is a data.frame, otherwise ignored. If survival analysis is applicable, these are the names of the survival time and event columns, so it has length 2. For multilabel classification these are the names of logical columns that indicate whether a class label is present and the number of target variables corresponds to the number of classes.
- **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.
  Default is "1-of-n".
- **cols**  
  (character)  
  Columns to create dummy features for. Default is to use all columns.

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

See Also
Other eda_and_preprocess: `capLargeValues()`, `dropFeatures()`, `mergeSmallFactorLevels()`, `normalizeFeatures()`, `removeConstantFeatures()`, `summarizeColumns()`, `summarizeLevels()`
createSpatialResamplingPlots

Create (spatial) resampling plot objects.

Description

Visualize partitioning of resample objects with spatial information.

Usage

createSpatialResamplingPlots(
  task = NULL,
  resample = NULL,
  crs = NULL,
  datum = 4326,
  repetitions = 1,
  color.train = "#0072B5",
  color.test = "#E18727",
  point.size = 0.5,
  axis.text.size = 14,
  x.axis.breaks = waiver(),
  y.axis.breaks = waiver()
)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td>Task</td>
</tr>
<tr>
<td>resample</td>
<td>ResampleResult or named list with (multiple) ResampleResult</td>
</tr>
<tr>
<td>crs</td>
<td>integer</td>
</tr>
<tr>
<td>datum</td>
<td>integer</td>
</tr>
<tr>
<td>repetitions</td>
<td>integer</td>
</tr>
<tr>
<td>color.train</td>
<td>character</td>
</tr>
<tr>
<td>color.test</td>
<td>character</td>
</tr>
<tr>
<td>point.size</td>
<td>integer</td>
</tr>
<tr>
<td>axis.text.size</td>
<td>integer</td>
</tr>
</tbody>
</table>
createSpatialResamplingPlots

x.axis.breaks numeric
Custom x axis breaks

y.axis.breaks numeric
Custom y axis breaks

Details

If a named list is given to resample, names will appear in the title of each fold. If multiple inputs are given to resample, these must be named.

This function makes a hard cut at five columns of the resulting gridded plot. This means if the resample object consists of folds > 5, these folds will be put into the new row.

For file saving, we recommend to use cowplot::save_plot.

When viewing the resulting plot in RStudio, margins may appear to be different than they really are. Make sure to save the file to disk and inspect the image.

When modifying axis breaks, negative values need to be used if the area is located in either the western or southern hemisphere. Use positive values for the northern and eastern hemisphere.

Value

(list of 2L containing (1) multiple ‘gg’ objects and (2) their corresponding labels.

CRS

The crs has to be suitable for the coordinates stored in the Task. For example, if the coordinates are UTM, crs should be set to a UTM projection. Due to a limited axis space in the resulting grid (especially on the x-axis), the data will by default projected into a lat/lon projection, specifically EPSG 4326. If other projections are desired for the resulting map, please set argument datum accordingly. This argument will be passed onto ggplot2::coord_sf.

Author(s)

Patrick Schratz

See Also

Other plot: plotBMRBoxplots(), plotBMRanksAsBarChart(), plotBMRSummary(), plotCalibration(), plotCritDifferences(), plotLearningCurve(), plotPartialDependence(), plotROCCurves(), plotResiduals(), plotThreshVsPerf()

Examples

rdesc = makeResampleDesc("SpRepCV", folds = 5, reps = 4)
r = resample(makeLearner("classif.qda"), spatial.task, rdesc)

## -------------------------------------------------------------
## single unnamed resample input with 5 folds and 2 repetitions
```r
## createSpatialResamplingPlots

plots = createSpatialResamplingPlots(spatial.task, r, crs = 32717,
  repetitions = 2, x.axis.breaks = c(-79.065, -79.085),
  y.axis.breaks = c(-3.970, -4))
cowplot::plot_grid(plotlist = plots["Plots"], ncol = 5, nrow = 2,
  labels = plots["Labels"])

## single named resample input with 5 folds and 1 repetition and 32717 datum

plots = createSpatialResamplingPlots(spatial.task, list("Resamp" = r),
  crs = 32717, datum = 32717, repetitions = 1)
cowplot::plot_grid(plotlist = plots["Plots"], ncol = 5, nrow = 1,
  labels = plots["Labels"])

## multiple named resample inputs with 5 folds and 1 repetition

rdesc1 = makeResampleDesc("SpRepCV", folds = 5, reps = 4)
r1 = resample(makeLearner("classif.qda"), spatial.task, rdesc1)
rdesc2 = makeResampleDesc("RepCV", folds = 5, reps = 4)
r2 = resample(makeLearner("classif.qda"), spatial.task, rdesc2)

plots = createSpatialResamplingPlots(spatial.task,
  list("SpRepCV" = r1, "RepCV" = r2), crs = 32717, repetitions = 1,
  x.axis.breaks = c(-79.055, -79.085), y.axis.breaks = c(-3.975, -4))
cowplot::plot_grid(plotlist = plots["Plots"], ncol = 5, nrow = 2,
  labels = plots["Labels"])

## Complex arrangements of multiple named resample inputs with 5 folds and 1 repetition

p1 = cowplot::plot_grid(plots["Plots"][[1]], plots["Plots"][[2]],
  plots["Plots"][[3]], ncol = 3, nrow = 1, labels = plots["Labels"][1:3],
  label.size = 18)
p12 = cowplot::plot_grid(plots["Plots"][[4]], plots["Plots"][[5]],
  ncol = 2, nrow = 1, labels = plots["Labels"][4:5], label.size = 18)
p2 = cowplot::plot_grid(plots["Plots"][[6]], plots["Plots"][[7]],
  plots["Plots"][[8]], ncol = 3, nrow = 1, labels = plots["Labels"][6:8],
  label.size = 18)
p22 = cowplot::plot_grid(plots["Plots"][[9]], plots["Plots"][[10]],
  ncol = 2, nrow = 1, labels = plots["Labels"][9:10], label.size = 18)
cowplot::plot_grid(p1, p12, p2, p22, ncol = 1)
```
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).

downsample

Description

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

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.
dropFeatures

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.

See Also

Other eda_and_preprocess: capLargeValues(), createDummyFeatures(), mergeSmallFactorLevels(), normalizeFeatures(), removeConstantFeatures(), summarizeColumns(), summarizeLevels()
estimateRelativeOverfitting

Estimate relative overfitting.

Description
Estimates the relative overfitting of a model as the ratio of the difference in test and train performance to the difference of test performance in the no-information case and train performance. In the no-information case the features carry no information with respect to the prediction. This is simulated by permuting features and predictions.

Usage

```
estimateRelativeOverfitting(
  predish, measures, task, learner = NULL, pred.train = NULL, iter = 1
)
```

Arguments

- **predish** `(ResampleDesc | ResamplePrediction | Prediction)`
  Resampling strategy or resampling prediction or test predictions.

- **measures** `(Measure | list of Measure)`
  Performance measure(s) to evaluate. Default is the default measure for the task, see here getDefaultMeasure.

- **task** `(Task)`
  The task.

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

- **pred.train** `(Prediction)`
  Training predictions. Only needed if test predictions are passed.

- **iter** `(integer)`
  Iteration number. Default 1, usually you don’t need to specify this. Only needed if test predictions are passed.

Details
Currently only support for classification and regression tasks is implemented.

Value

`(data.frame)`. Relative overfitting estimate(s), named by measure(s), for each resampling iteration.
**estimateResidualVariance**

**References**


**See Also**

Other performance: `ConfusionMatrix`, `calculateConfusionMatrix()`, `calculateROCMeasures()`, `makeCostMeasure()`, `makeCustomResampledMeasure()`, `makeMeasure()`, `measures`, `performance()`, `setAggregation()`, `setMeasurePars()`

**Examples**

```r
task = makeClassifTask(data = iris, target = "Species")
rdesc = makeResampleDesc("CV", iters = 2)
estimateRelativeOverfitting(rdesc, acc, task, makeLearner("classif.knn"))
estimateRelativeOverfitting(rdesc, acc, task, makeLearner("classif.lda"))
rpred = resample("classif.knn", task, rdesc)$pred
estimateRelativeOverfitting(rpred, acc, 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**

```r
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.
extractFDABsignal  

**Bspline mlq features**

**Description**

The function extracts features from functional data based on the Bspline fit. For more details refer to `FDboost::bsignal()`.

**Usage**

```r
extractFDABsignal(bsignal.knots = 10L, bsignal.df = 3)
```

**Arguments**

- `bsignal.knots` (integer(1))
  The number of knots for bspline.
- `bsignal.df` (numeric(1))
  The effective degree of freedom of penalized bspline.

**Value**

(data.frame).

**See Also**

Other fda_featextractor: `extractFDADTWKernel()`, `extractFDAFPCA()`, `extractFDAFourier()`, `extractFDAMultiResFeatures()`, `extractFDATsfeatures()`, `extractFDAWavelets()`

---

extractFDADTWKernel  

**DTW kernel features**

**Description**

The function extracts features from functional data based on the DTW distance with a reference dataframe.

**Usage**

```r
extractFDADTWKernel(
    ref.method = "random",
    n.refs = 0.05,
    refs = NULL,
    dtwwindow = 0.05
)
```
extractFDAFeatures

Arguments

ref.method (character(1))
How should the reference curves be obtained? Method random draws n.refs random reference curves, while all uses all curves as references. In order to use user-provided reference curves, this parameter is set to fixed.

n.refs (numeric(1))
Number of reference curves to be drawn (as a fraction of the number of observations in the training data).

refs (matrix|integer(n))
Integer vector of training set row indices or a matrix of reference curves with the same length as the functionals in the training data. Overwrites ref.method and n.refs.

dtwwindow (numeric(1))
Size of the warping window size (as a proportion of query length).

Value

(data.frame).

See Also

Other fda_featextractor: extractFDABsignal(), extractFDAFPCA(), extractFDAFourier(), extractFDAMultiResFeatures(), extractFDATsfeatures(), extractFDAWavelets()

extractFDAFeatures  Extract features from functional data.

Description

Extract non-functional features from functional features using various methods.

The function extractFDAFeatures performs the extraction for all functional features via the methods specified in feat.methods and transforms all mentioned functional (matrix) features into regular data.frame columns. Additionally, a “extractFDAFeatDesc” object which contains learned coefficients and other helpful data for re-extraction during the predict-phase is returned. This can be used with reextractFDAFeatures in order to extract features during the prediction phase.

Usage

extractFDAFeatures(obj, target = character(0L), feat.methods = list(), ...)

Arguments

obj (Task | data.frame)
Task or data.frame to extract functional features from. Must contain functional features as matrix columns.

target (character(1))
Task target column. Only necessary for data.frames Default is character(0).

feat.methods (named list)
List of functional features along with the desired methods for each functional feature. “all” applies the extractFDAFeatures method to each functional feature. Names of feat.methods must match column names of functional features. Available feature extraction methods are available under family fda_featextractor. Specifying a functional feature multiple times with different extraction methods allows for the extraction of different features from the same functional. Default is list() which does nothing.

...
(any)
Further hyperparameters passed on to the feat.methods specified above.

Details

The description object contains these slots:

- target (character): See argument.
- coln (character): Column names of data.
- fd.cols (character): Functional feature names.
- extractFDAFeat (list): Contains feature.methods and relevant parameters for reextraction.

Value

(list)

- data | task (data.frame | Task): Extracted features, same type as obj.
- desc (extractFDAFeatDesc): Description object. See description for details.

See Also

Other fda: makeExtractFDAFeatMethod(), makeExtractFDAFeatsWrapper()

Examples

df = data.frame(x = matrix(rnorm(24), ncol = 8), y = factor(c("a", "a", "b")))
fdf = makeFunctionalData(df, fd.features = list(x1 = 1:4, x2 = 5:8), exclude.cols = "y")
task = makeClassifTask(data = fdf, target = "y")
extracted = extractFDAFeatures(task,
      feat.methods = list("x1" = extractFDAFourier(), "x2" = extractFDAWavelets(filter = "haar")))
print(extracted$task)
reextractFDAFeatures(task, extracted$desc)
extractFDAFourier

**Fast Fourier transform features.**

**Description**

The function extracts features from functional data based on the fast fourier transform. For more details refer to stats:fft.

**Usage**

```R
extractFDAFourier(trafo.coeff = "phase")
```

**Arguments**

- `trafo.coeff` (character(1))
  Specifies which transformation of the complex frequency domain representation should be calculated as a feature representation. Must be one of “amplitude” or “phase”. Default is “phase”. The phase shift is returned in Rad, i.e. values lie in [-180, 180].

**Value**

(data.frame).

**See Also**

Other fda_featextractor: extractFDAbsignal(), extractFDADTWKernel(), extractFDAFPCA(), extractFDAMultiResFeatures(), extractFDAStsfeatures(), extractFDAWavelets()

---

extractFDAFPCA

**Extract functional principal component analysis features.**

**Description**

The function extracts the functional principal components from a data.frame containing functional features. Uses stats::prcomp.

**Usage**

```R
extractFDAFPCA(rank. = NULL, center = TRUE, scale. = FALSE)
```
extractFDAMultiResFeatures

Arguments

- **rank.** (integer(1))
  Number of principal components to extract. Default is NULL
- **center** (logical(1))
  Should data be centered before applying PCA?
- **scale.** (logical(1))
  Should data be scaled before applying PCA?

Value

(data.frame).

See Also

Other fda_featextractor: extractFDABsignal(), extractFDADTWKernel(), extractFDAFourier(), extractFDAMultiResFeatures(), extractFDATsfeatures(), extractFDAWavelets()

extractFDAMultiResFeatures

Multiresolution feature extraction.

Description

The function extracts currently the mean of multiple segments of each curve and stacks them as features. The segments length are set in a hierarchy way so the features cover different resolution levels.

Usage

extractFDAMultiResFeatures(res.level = 3L, shift = 0.5, seg.lens = NULL)

Arguments

- **res.level** (integer(1))
  The number of resolution hierarchy, each length is divided by a factor of 2.
- **shift** (numeric(1))
  The overlapping proportion when slide the window for one step.
- **seg.lens** (integer(1))
  Curve subsequence lengths. Needs to sum up to the length of the functional.

Value

(data.frame).

See Also

Other fda_featextractor: extractFDABsignal(), extractFDADTWKernel(), extractFDAFourier(), extractFDAMultiResFeatures(), extractFDATsfeatures(), extractFDAWavelets()
extractFDATsfeatures  Time-Series Feature Heuristics

Description
The function extracts features from functional data based on known Heuristics. For more details refer to `tsfeatures::tsfeatures()`. Under the hood this function uses the package `tsfeatures::tsfeatures()`. For more information see Hyndman, Wang and Laptev, Large-Scale Unusual Time Series Detection, ICDM 2015.

Note: Currently computes the following features:

Usage

```r
extractFDATsfeatures(
  scale = TRUE,
  trim = FALSE,
  trim_amount = 0.1,
  parallel = FALSE,
  na.action = na.pass,
  feats = NULL,
  ...
)
```

Arguments

- **scale** (logical(1))
  If TRUE, time series are scaled to mean 0 and sd 1 before features are computed.

- **trim** (logical(1))
  If TRUE, time series are trimmed by trim_amount before features are computed. Values larger than trim_amount in absolute value are set to NA.

- **trim_amount** (numeric(1))
  Default level of trimming if trim==TRUE.

- **parallel** (logical(1))
  If TRUE, multiple cores (or multiple sessions) will be used. This only speeds things up when there are a large number of time series.

- **na.action** (logical(1))
  A function to handle missing values. Use `na.interp` to estimate missing values.

- **feats** (character)
  A character vector of function names to apply to each time-series in order to extract features.
  Default:
extractFDAWavelets

Discrete Wavelet transform features.

Description

The function extracts discrete wavelet transform coefficients from the raw functional data. See wavelets::dwt for more information.

Usage

extractFDAWavelets(filter = "la8", boundary = "periodic")

Arguments

filter (character(1))
Specifies which filter should be used. Must be one of dl|la|bl|c followed by an even number for the level of the filter. The level of the filter needs to be smaller or equal than the time-series length. For more information and acceptable filters see help(wt.filter). Defaults to la8.

boundary (character(1))
Boundary to be used. "periodic" assumes circular time series, for "reflection" the series is extended to twice its length. Default is "periodic".

Value

(data.frame).
**FailureModel**

**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.

The if mlr option `on.error.dump` is `TRUE`, the `FailureModel` contains the debug trace of the error. It can be accessed with `getFailureModelDump` and inspected with debugger.

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.

**See Also**

Other debug: `ResampleResult, getPredictionDump(), getRRDump()`

**Examples**

```r
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")
```
FeatSelControl

Create control structures for feature selection.

Description

Feature selection method used by selectFeatures. The methods used here follow a wrapper approach, described in Kohavi and John (1997) (see references).

The following optimization algorithms 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

```r
makeFeatSelControlExhaustive(
  same.resampling.instance = TRUE,
  maxit = NA_integer_,
  max.features = NA_integer_,
  tune.threshold = FALSE,
  tune.threshold.args = list(),
  log.fun = "default"
```

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,
  tune.threshold = FALSE,
  tune.threshold.args = list(),
  log.fun = "default"
)

makeFeatSelControlRandom(
  same.resampling.instance = TRUE,
  maxit = 100L,
  max.features = NA_integer_,
  prob = 0.5,
  tune.threshold = FALSE,
  tune.threshold.args = list(),
  log.fun = "default"
)

makeFeatSelControlSequential(
  same.resampling.instance = TRUE,
  impute.val = NULL,
  method,
  alpha = 0.01,
  beta = -0.001,
  maxit = NA_integer_,
  max.features = NA_integer_,
  tune.threshold = FALSE,
  tune.threshold.args = list(),
  log.fun = "default"
)

Arguments

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

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.

tune.threshold (logical(1))
Should the threshold be tuned for the measure at hand, after each feature set evaluation, via tuneThreshold? Only works for classification if the predict type is "prob". Default is FALSE.

tune.threshold.args (list)
Further arguments for threshold tuning that are passed down to tuneThreshold. Default is none.

log.fun (function | character(1))
Function used for logging. If set to "default" (the default), the evaluated design points, the resulting performances, and the runtime will be reported. If set to "memory" the memory usage for each evaluation will also be displayed, with character(1) small increase in run time. Otherwise character(1) function with arguments learner, resampling, measures, par.set, control, opt.path, dob, x, y, remove.nas, stage and prev.stage is expected. The default displays the performance measures, the time needed for evaluating, the currently used memory and the max memory ever used before (the latter two both taken from gc). See the implementation for details.

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. Imputation is only active if on.learner.error is configured not to stop in configureMlr. 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.

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

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.
**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.

---

**prob** *(numeric(1))*


**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.

**Value**

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

**References**


**See Also**

Other featsel: `analyzeFeatSelResult()`, `getFeatSelResult()`, `makeFeatSelWrapper()`, `selectFeatures()`
threshold (numeric) Vector of finally found and used thresholds if tune.threshold was enabled in FeatSelControl, otherwise not present and hence NULL.

opt.path (ParamHelpers::OptPath) Optimization path which lead to x.

---

filterFeatures (Task) Filter features by thresholding filter values.

Description

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

Usage

filterFeatures(
  task,
  method = "FSelectorRcpp_information.gain",
  fval = NULL,
  perc = NULL,
  abs = NULL,
  threshold = NULL,
  fun = NULL,
  fun.args = NULL,
  mandatory.feat = NULL,
  select.method = NULL,
  base.methods = NULL,
  cache = FALSE,
  ...
)

Arguments

task (Task) The task.

method (character(1)) See listFilterMethods. Default is “FSelectorRcpp_information.gain”.

fval (FilterValues) Result of generateFilterValuesData. If you pass this, the filter values in the object are used for feature filtering. method and ... are ignored then. Default is NULL and not used.

perc (numeric(1)) If set, select perc*100 top scoring features. perc = 1 means to select all features. Mutually exclusive with arguments abs, threshold and fun.

abs (numeric(1)) If set, select abs top scoring features. Mutually exclusive with arguments perc, threshold and fun.
threshold (numeric(1))
If set, select features whose score exceeds threshold. Mutually exclusive with arguments perc, abs and fun.

fun (function)
If set, select features via a custom thresholding function, which must return the number of top scoring features to select. Mutually exclusive with arguments perc, abs and threshold.

fun.args (any)
Arguments passed to the custom thresholding function.

mandatory.feat (character)
Mandatory features which are always included regardless of their scores

select.method
If multiple methods are supplied in argument method, specify the method that is used for the final subsetting.

base.methods
If method is an ensemble filter, specify the base filter methods which the ensemble method will use.

cache (character(1) | logical)
Whether to use caching during filter value creation. See details.

... (any)
Passed down to selected filter method.

Value

Task.

Caching

If cache = TRUE, the default mlr cache directory is used to cache filter values. The directory is operating system dependent and can be checked with getCacheDir().
The default cache can be cleared with deleteCacheDir(). Alternatively, a custom directory can be passed to store the cache.

Note that caching is not thread safe. It will work for parallel computation on many systems, but there is no guarantee.

Simple and ensemble filters

Besides passing (multiple) simple filter methods you can also pass an ensemble filter method (in a list). The ensemble method will use the simple methods to calculate its ranking. See listFilterEnsembleMethods() for available ensemble methods.

See Also

Other filter: generateFilterValuesData(), getFilteredFeatures(), listFilterEnsembleMethods(), listFilterMethods(), makeFilter(), makeFilterEnsemble(), makeFilterWrapper(), plotFilterValues()
Examples

```r
# simple filter
filterFeatures(iris.task, method = "FSelectorRcpp_gain.ratio", abs = 2)
# ensemble filter
filterFeatures(iris.task, method = "E-min",
               base.methods = c("FSelectorRcpp_gain.ratio",
                               "FSelectorRcpp_information.gain"), abs = 2)
```

---

**friedmanPostHocTestBMR**

*Perform a posthoc Friedman-Nemenyi test.*

---

**Description**

Performs a PMCMRplus::frdAllPairsNemenyiTest for a BenchmarkResult and a selected measure. This means *all pairwise comparisons* of learners are performed. The null hypothesis of the post hoc test is that each pair of learners is equal. If the null hypothesis of the included ad hoc stats::friedman.test can be rejected an object of class pairwise.htest is returned. If not, the function returns the corresponding friedman.test.

Note that benchmark results for at least two learners on at least two tasks are required.

**Usage**

```r
friedmanPostHocTestBMR(
  bmr,  # BenchmarkResult
  measure = NULL,  # Measure
  p.value = 0.05,  # numeric(1)
  aggregation = "default"  # character(1)
)
```

**Arguments**

- **bmr** *(BenchmarkResult)*
  Benchmark result.

- **measure** *(Measure)*
  Performance measure. Default is the first measure used in the benchmark experiment.

- **p.value** *(numeric(1))*
  p-value for the tests. Default: 0.05

- **aggregation** *(character(1))*
  "mean" or "default". See getBMRAggrPerformances for details on "default".
friedmanTestBMR

Value

(pairwise.htest): See PMCMRplus::frdAllPairsNemenyiTest for details. Additionally two components are added to the list:

- f.rejnull (logical(1)): Whether the according friedman.test rejects the Null hypothesis at the selected p.value
- crit.difference (list(2)): Minimal difference the mean ranks of two learners need to have in order to be significantly different

See Also

Other benchmark: BenchmarkResult, batchmark(), benchmark(), convertBMRToRankMatrix(), friedmanTestBMR(), generateCritDifferencesData(), getBMRAggrPerformances(), getBMRFeatSelResults(), getBMRFiltedFeatures(), getBMRLearnerIds(), getBMRLearnerShortNames(), getBMRLearners(), getBMRMeasureIds(), getBMRMeasures(), getBMRModels(), getBMRPerformances(), getBMRPredictions(), getBMRTaskDescs(), getBMRTaskIds(), getBMRTuneResults(), plotBMRRankMatrix(), plotBMRRanksAsBarChart(), plotBMRSummary(), plotCritDifferences(), reduceBatchmarkResults()

Examples

# see benchmark

| friedmanTestBMR | Perform overall Friedman test for a BenchmarkResult. |

Description

Performs a stats::friedman.test for a selected measure. The null hypothesis is that apart from an effect of the different (Task), the location parameter (aggregated performance measure) is the same for each Learner. Note that benchmark results for at least two learners on at least two tasks are required.

Usage

friedmanTestBMR(bmr, measure = NULL, aggregation = "default")

Arguments

bmr (BenchmarkResult) Benchmark result.
measure (Measure) Performance measure. Default is the first measure used in the benchmark experiment.
aggregation (character(1)) “mean” or “default”. See getBMRAggrPerformances for details on “default".
generateCalibrationData

Value

(\texttt{htest}): \underline{See \texttt{stats::friedman.test}} for details.

See Also

Other benchmark: \texttt{BenchmarkResult\_batchmark()}, \texttt{benchmark()}, \texttt{convertBMRTToRankMatrix()}, \texttt{friedmanPostHocTestBMR()}, \texttt{generateCritDifferencesData()}, \texttt{getBMRAggrPerformances()}, \texttt{getBMRFeatSelResults()}, \texttt{getBMRFilteredFeatures()}, \texttt{getBMRLearnerIds()}, \texttt{getBMRLearnerShortNames()}, \texttt{getBMRLearners()}, \texttt{getBMRMetabolids()}, \texttt{getBMRMetametabolids()}, \texttt{getBMRMetamodels()}, \texttt{getBMRMetaperformances()}, \texttt{getBMRPredictions()}, \texttt{getBMRTaskDescs()}, \texttt{getBMRTaskIds()}, \texttt{getBMRTuneResults()}, \texttt{plotBMRRBoxplots()}, \texttt{plotBMRRanksAsBarChart()}, \texttt{plotBMRRsumary()}, \texttt{plotCritDifferences()}, \texttt{reduceBatchmarkResults()}

Examples

# see benchmark

\begin{verbatim}

fuelsubset.task                          \textit{FuelSubset functional data regression task.}

\end{verbatim}

Description

Contains the task (fuelsubset.task). 2 functional covariates and 1 scalar covariate. You have
to predict the heat value of some fuel based on the ultraviolet radiation spectrum and infrared ray
radiation and one scalar column called h2o.

Details

The features and grids are scaled in the same way as in \texttt{FDboost::FDboost}.

References

Statistical Modelling, 15(3), 279–300.

\begin{verbatim}

generateCalibrationData                 \textit{Generate classifier calibration data.}

\end{verbatim}

Description

A calibrated classifier is one where the predicted probability of a class closely matches the rate at
which that class occurs, e.g. for data points which are assigned a predicted probability of class
A of .8, approximately 80 percent of such points should belong to class A if the classifier is well
calibrated. This is estimated empirically by grouping data points with similar predicted probabilities
for each class, and plotting the rate of each class within each bin against the predicted probability
bins.

\begin{verbatim}

# see benchmark

fuelsubset.task

\end{verbatim}
**Usage**

```r
generateCalibrationData(obj, breaks = "Sturges", groups = NULL, task.id = NULL)
```

**Arguments**

- **obj** (list of `Prediction` | list of `ResampleResult` | `BenchmarkResult`)  
  Single prediction object, list of them, single resample result, list of them, or a benchmark result. In case of a list probably produced by different learners you want to compare, then name the list with the names you want to see in the plots, probably learner shortnames or ids.

- **breaks** (character(1) | numeric)  
  If character(1), the algorithm to use in generating probability bins. See `hist` for details. If numeric, the cut points for the bins. Default is “Sturges”.

- **groups** (integer(1))  
  The number of bins to construct. If specified, `breaks` is ignored. Default is `NULL`.

- **task.id** (character(1))  
  Selected task in `BenchmarkResult` to do plots for, ignored otherwise. Default is first task.

**Value**

- **CalibrationData.** A list containing:
  - **proportion** `data.frame` with columns:
    - Learner Name of learner.
    - bin Bins calculated according to the breaks or groups argument.
    - Class Class labels (for binary classification only the positive class).
    - Proportion Proportion of observations from class `Class` among all observations with posterior probabilities of class `Class` within the interval given in `bin`.
  - **data** `data.frame` with columns:
    - Learner Name of learner.
    - truth True class label.
    - Class Class labels (for binary classification only the positive class).
    - Probability Predicted posterior probability of `Class`.
    - bin Bin corresponding to `Probability`.
  - **task** (`TaskDesc`)  
    Task description.

**References**

generateCritDifferencesData

Generate data for critical-differences plot.

Description

Generates data that can be used to plot a critical differences plot. Computes the critical differences according to either the "Bonferroni-Dunn" test or the "Nemenyi" test. "Bonferroni-Dunn" usually yields higher power as it does not compare all algorithms to each other, but all algorithms to a baseline instead. Learners are drawn on the y-axis according to their average rank. For test = "nemenyi" a bar is drawn, connecting all groups of not significantly different learners. For test = "bd" an interval is drawn around the algorithm selected as a baseline. All learners within this interval are not significantly different from the baseline. Calculation:

\[ CD = q_\alpha \sqrt{\left( \frac{k(k+1)}{6N} \right)} \]

Where \( q_\alpha \) is based on the studentized range statistic. See references for details.

Usage

generateCritDifferencesData(
  bmr,
  measure = NULL,
  p.value = 0.05,
  baseline = NULL,
  test = "bd"
)

Arguments

- **bmr**: (BenchmarkResult) Benchmark result.
- **measure**: (Measure) Performance measure. Default is the first measure used in the benchmark experiment.
- **p.value**: (numeric(1)) P-value for the critical difference. Default: 0.05
baseline (character(1)): (learner.id)
Select a learner.id as baseline for the test = "bd" ("Bonferroni-Dunn") critical differences diagram. The critical difference interval will then be positioned around this learner. Defaults to best performing algorithm.
For test = "nemenyi", no baseline is needed as it performs all pairwise comparisons.

test (character(1))
Test for which the critical differences are computed.
"bd" for the Bonferroni-Dunn Test, which is comparing all classifiers to a baseline, thus performing a comparison of one classifier to all others.
Algorithms not connected by a single line are statistically different from the baseline.
"nemenyi" for the PMCMRplus::frdAllPairsNemenyiTest which is comparing all classifiers to each other. The null hypothesis that there is a difference between the classifiers can not be rejected for all classifiers that have a single grey bar connecting them.

Value

(critDifferencesData). List containing:

data (data.frame) containing the info for the descriptive part of the plot
friedman.nemenyi.test (list) of class pairwise.htest contains the calculated PMCMRplus::frdAllPairsNemenyiTest
cd.info (list) containing info on the critical difference and its positioning
baseline baseline chosen for plotting
p.value p.value used for the PMCMRplus::frdAllPairsNemenyiTest and for computation of the critical difference

See Also

Other generate_plot_data: generateCalibrationData(), generateFeatureImportanceData(), generateFilterValuesData(), generateLearningCurveData(), generatePartialDependenceData(), generateThreshVsPerfData()

Other benchmark: BenchmarkResult, batchmark(), benchmark(), convertBMRToRankMatrix(), friedmanPostHocTestBMR(), friedmanTestBMR(), getBMRAggrPerformances(), getBMRFeatSelResults(), getBMRFilteredFeatures(), getBMRLearnerIds(), getBMR LearnerShortNames(), getBMR Learners(), getBMRMeasureIds(), getBMRMeasures(), getBMRModels(), getBMRPerformances(), getBMRPredictions(), getBMRTaskDescs(), getBMRTaskIds(), getBMRTuneResults(),plotBMRBoxplots(), plotBMRanksAsBarChart(), plotBMRSummary(), plotCritDifferences(), reduceBatchmarkResults()
generateFeatureImportanceData

Generate feature importance.

Description

Estimate how important individual features or groups of features are by contrasting prediction performances. For method “permutation.importance” compute the change in performance from permuting the values of a feature (or a group of features) and compare that to the predictions made on the unpermuted data.

Usage

generateFeatureImportanceData(
  task,
  method = "permutation.importance",
  learner,
  features = getTaskFeatureNames(task),
  interaction = FALSE,
  measure,
  contrast = function(x, y) x - y,
  aggregation = mean,
  nmc = 50L,
  replace = TRUE,
  local = FALSE,
  show.info = FALSE
)

Arguments

task: (Task)
The task.

method: (character(1))
The method used to compute the feature importance. The only method available is “permutation.importance”. Default is “permutation.importance”.

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

features: (character)
The features to compute the importance of. The default is all of the features contained in the Task.

interaction: (logical(1))
Whether to compute the importance of the features argument jointly. For method = "permutation.importance" this entails permuting the values of all features together and then contrasting the performance with that of the performance without the features being permuted. The default is FALSE.
generateFeatureImportanceData

measure (Measure)
Performance measure. Default is the first measure used in the benchmark experiment.

contrast (function)
A difference function that takes a numeric vector and returns a numeric vector of the same length. The default is element-wise difference between the vectors.

aggregation (function)
A function which aggregates the differences. This function must take a numeric vector and return a numeric vector of length 1. The default is mean.

nmc (integer(1))
The number of Monte-Carlo iterations to use in computing the feature importance. If nmc == -1 and method = "permutation.importance" then all permutations of the features are used. The default is 50.

replace (logical(1))
Whether or not to sample the feature values with or without replacement. The default is TRUE.

local (logical(1))
Whether to compute the per-observation importance. The default is FALSE.

show.info (logical(1))
Whether progress output (feature name, time elapsed) should be displayed.

Value

(FeatureImportance). A named list which contains the computed feature importance and the input arguments.

Object members:

res (data.frame)
Has columns for each feature or combination of features (colon separated) for which the importance is computed. A row coresponds to importance of the feature specified in the column for the target.

interaction (logical(1))
Whether or not the importance of the features was computed jointly rather than individually.

measure (Measure)
The measure used to compute performance.

contrast (function)
The function used to compare the performance of predictions.

aggregation (function)
The function which is used to aggregate the contrast between the performance of predictions across Monte-Carlo iterations.

replace (logical(1))
Whether or not, when method = "permutation.importance", the feature values are sampled with replacement.
generateFilterValuesData

nmc (integer(1))
The number of Monte-Carlo iterations used to compute the feature importance.
When nmc == -1 and method = "permutation.importance" all permutations are used.

local (logical(1))
Whether observation-specific importance is computed for the features.

References

See Also
Other generate_plot_data: generateCalibrationData(), generateCritDifferencesData(), generateFilterValuesData(), generateLearningCurveData(), generatePartialDependenceData(), generateThreshVsPerfData(), plotFilterValues()

Examples

```r
lrn = makeLearner("classif.rpart", predict.type = "prob")
fit = train(lrn, iris.task)
imp = generateFeatureImportanceData(iris.task, "permutation.importance",
    lrn, "Petal.Width", nmc = 10L, local = TRUE)
```

---

generateFilterValuesData

*Calculates feature filter values.*

Description
Calculates numerical filter values for features. For a list of features, use listFilterMethods.

Usage

generateFilterValuesData(
task,
    method = "FSelectorRcpp_information.gain",
    nselect = getTaskNFeats(task),
    ...
    more.args = list()
)
generateFilterValuesData

Arguments

- task (Task)
The task.
- method (character | list)
  Filter method(s). In case of ensemble filters the list notation needs to be used. See the examples for more information. Default is “FSselectorRcpp_information.gain”.
- nselect (integer(1))
  Number of scores to request. Scores are getting calculated for all features per default.
- ... (any)
  Passed down to selected method. Can only be use if method contains one element.
- more.args (named list)
  Extra args passed down to filter methods. List elements are named with the filter method name the args should be passed down to. A more general and flexible option than .... Default is empty list.

Value

(FilterValues). A list containing:

- task.desc [TaskDesc]
  Task description.
- data (data.frame) with columns:
  - name(character)
    Name of feature.
  - type(character)
    Feature column type.
  - method(numeric)
    One column for each method with the feature importance values.

Simple and ensemble filters

Besides passing (multiple) simple filter methods you can also pass an ensemble filter method (in a list). The ensemble method will use the simple methods to calculate its ranking. See listFilterEnsembleMethods() for available ensemble methods.

See Also

Other generate_plot_data: generateCalibrationData(), generateCritDifferencesData(), generateFeatureImportanceData(), generateLearningCurveData(), generatePartialDependenceData(), generateThreshVsPerfData(), plotFilterValues()

Other filter: filterFeatures(), getFilteredFeatures(), listFilterEnsembleMethods(), listFilterMethods(), makeFilter(), makeFilterEnsemble(), makeFilterWrapper(), plotFilterValues()
Examples

```r
# two simple filter methods
fval = generateFilterValuesData(iris.task,
    method = c("FSelectorRcpp_gain.ratio", "FSelectorRcpp_information.gain"))
# using ensemble method "E-mean"
fval = generateFilterValuesData(iris.task,
    method = list("E-mean", c("FSelectorRcpp_gain.ratio",
    "FSelectorRcpp_information.gain")))
```

---

generateHyperParsEffectData

*Generate hyperparameter effect data.*

Description

Generate cleaned hyperparameter effect data from a tuning result or from a nested cross-validation tuning result. The object returned can be used for custom visualization or passed downstream to an out of the box mlr method, `plotHyperParsEffect`.

Usage

```r
generateHyperParsEffectData(
    tune.result,
    include.diagnostics = FALSE,
    trafo = FALSE,
    partial.dep = FALSE
)
```

Arguments

- **tune.result** *(TuneResult | ResampleResult)*
  - Result of `tuneParams` (or `resample` ONLY when used for nested cross-validation).
  - The tuning result (or results if the output is from nested cross-validation), also containing the optimizer results. If nested CV output is passed, each element in the list will be considered a separate run, and the data from each run will be included in the dataframe within the returned `HyperParsEffectData`.

- **include.diagnostics** *(logical(1))
  - Should diagnostic info (eol and error msg) be included? Default is `FALSE`.

- **trafo** *(logical(1))
  - Should the units of the hyperparameter path be converted to the transformed scale? This is only useful when `trafo` was used to create the path. Default is `FALSE`.
partial.dep (logical(1))
Should partial dependence be requested based on converting to reg task? This sets a flag so that we know to use partial dependence downstream. This should most likely be set to TRUE if 2 or more hyperparameters were tuned simultaneously. Partial dependence should always be requested when more than 2 hyperparameters were tuned simultaneously. Setting to TRUE will cause plotHyperParsEffect to automatically plot partial dependence when called downstream. Default is FALSE.

Value

(HyperParsEffectData) Object containing the hyperparameter effects dataframe, the tuning performance measures used, the hyperparameters used, a flag for including diagnostic info, a flag for whether nested cv was used, a flag for whether partial dependence should be generated, and the optimization algorithm used.

Examples

```r
## Not run:
# 3-fold cross validation
ps = makeParamSet(makeDiscreteParam("C", values = 2^(-4:4)))
ctrl = makeTuneControlGrid()
rdesc = makeResampleDesc("CV", iters = 3L)
res = tuneParams("classif.ksvm", task = pid.task, resampling = rdesc,
                 par.set = ps, control = ctrl)
data = generateHyperParsEffectData(res)
plt = plotHyperParsEffect(data, x = "C", y = "mmce.test.mean")
plt + ylab("Misclassification Error")

# nested cross validation
ps = makeParamSet(makeDiscreteParam("C", values = 2^(-4:4)))
ctrl = makeTuneControlGrid()
rdesc = makeResampleDesc("CV", iters = 3L)
lrn = makeTuneWrapper("classif.ksvm", control = ctrl,
                     resampling = rdesc, par.set = ps)
res = resample(lrn, task = pid.task, resampling = cv2,
               extract = getTuneResult)
data = generateHyperParsEffectData(res)
plotHyperParsEffect(data, x = "C", y = "mmce.test.mean", plot.type = "line")

## End(Not run)
```

**generateLearningCurveData**

*Generates a learning curve.*

**Description**

Observe how the performance changes with an increasing number of observations.
Usage

generateLearningCurveData(
  learners,
  task,
  resampling = NULL,
  percs = seq(0.1, 1, by = 0.1),
  measures,
  stratify = FALSE,
  show.info = getMlrOption("show.info")
)

Arguments

learners [(list of) Learner]
  Learning algorithms which should be compared.

task (Task)
  The task.

resampling (ResampleDesc | ResampleInstance)
  Resampling strategy to evaluate the performance measure. If no strategy is given
  a default "Holdout" will be performed.

percs (numeric)
  Vector of percentages to be drawn from the training split. These values represent
  the x-axis. Internally makeDownsampleWrapper is used in combination with
  benchmark. Thus for each percentage a different set of observations is drawn
  resulting in noisy performance measures as the quality of the sample can differ.

measures [(list of) Measure]
  Performance measures to generate learning curves for, representing the y-axis.

stratify (logical(1))
  Only for classification: Should the downsampled data be stratified according to
  the target classes?

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

Value

(LearningCurveData). A list containing:

- The Task
- List of Measure)
  Performance measures
- data (data.frame) with columns:
  - learner Names of learners.
  - percentage Percentages drawn from the training split.
  - One column for each Measure passed to generateLearningCurveData.
generatePartialDependenceData

See Also

Other generate_plot_data: generateCalibrationData(), generateCritDifferencesData(), generateFeatureImportanceData(), generateFilterValuesData(), generatePartialDependenceData(), generateThreshVsPerfData(), plotFilterValues()

Other learning_curve: plotLearningCurve()

Examples

r = generateLearningCurveData(list("classif.rpart", "classif.knn"),
  task = sonar.task, percs = seq(0.2, 1, by = 0.2),
  measures = list(tp, fp, tn, fn),
  resampling = makeResampleDesc(method = "Subsample", iters = 5),
  show.info = FALSE)
plotLearningCurve(r)

generatePartialDependenceData

Generate partial dependence.

Description

Estimate how the learned prediction function is affected by one or more features. For a learned
function f(x) where x is partitioned into x_s and x_c, the partial dependence of f on x_s can be
summarized by averaging over x_c and setting x_s to a range of values of interest, estimating
E_{x_c}(f(x_s, x_c)). The conditional expectation of f at observation i is estimated similarly. Addi-
tionally, partial derivatives of the marginalized function w.r.t. the features can be computed.

This function requires the mmpf package to be installed. It is currently not on CRAN, but can be
installed through GitHub using devtools::install_github(’zmjones/mmpf/pkg’).

Usage

generatePartialDependenceData(
  obj,
  input,
  features = NULL,
  interaction = FALSE,
  derivative = FALSE,
  individual = FALSE,
  fun = mean,
  bounds = c(qnorm(0.025), qnorm(0.975)),
  uniform = TRUE,
  n = c(10, NA),
  ...
)

Arguments

obj (WrappedModel)
Result of \texttt{train}.

input (data.frame | Task)
Input data.

features character
A vector of feature names contained in the training data. If not specified all features in the input will be used.

interaction (logical(1))
Whether the features should be interacted or not. If \texttt{TRUE} then the Cartesian product of the prediction grid for each feature is taken, and the partial dependence at each unique combination of values of the features is estimated. Note that if the length of features is greater than two, \texttt{plotPartialDependence} cannot be used. If \texttt{FALSE} each feature is considered separately. In this case \texttt{features} can be much longer than two. Default is \texttt{FALSE}.

derivative (logical(1))
Whether or not the partial derivative of the learned function with respect to the features should be estimated. If \texttt{TRUE} interaction must be \texttt{FALSE}. The partial derivative of individual observations may be estimated. Note that computation time increases as the learned prediction function is evaluated at \texttt{gridsize} points * the number of points required to estimate the partial derivative. Additional arguments may be passed to \texttt{numDeriv::grad} (for regression or survival tasks) or \texttt{numDeriv::jacobian} (for classification tasks). Note that functions which are not smooth may result in estimated derivatives of 0 (for points where the function does not change within +/- epsilon) or estimates trending towards +/- infinity (at discontinuities). Default is \texttt{FALSE}.

individual (logical(1))
Whether to plot the individual conditional expectation curves rather than the aggregated curve, i.e., rather than aggregating (using \texttt{fun}) the partial dependences of features, plot the partial dependences of all observations in \texttt{data} across all values of the features. The algorithm is developed in Goldstein, Kapelner, Bleich, and Pitkin (2015). Default is \texttt{FALSE}.

fun function
A function which operates on the output on the predictions made on the input data. For regression this means a numeric vector, and, e.g., for a multiclass classification problem, this migh instead be probabilities which are returned as a numeric matrix. This argument can return vectors of arbitrary length, however, if their length is greater than one, they must by named, e.g., \texttt{fun = mean} or \texttt{fun = function(x) c("mean" = mean(x), "variance" = var(x))}. The default is the mean, unless \texttt{obj} is classification with \texttt{predict.type = "response"} in which case the default is the proportion of observations predicted to be in each class.

bounds (numeric(2))
The value (lower, upper) the estimated standard error is multiplied by to estimate the bound on a confidence region for a partial dependence. Ignored if
predict.type != "se" for the learner. Default is the 2.5 and 97.5 quantiles (-1.96, 1.96) of the Gaussian distribution.

`uniform` (logical(1))
Whether or not the prediction grid for the features is a uniform grid of size n[1] or sampled with replacement from the input. Default is TRUE.

`n` (integer21)
The first element of n gives the size of the prediction grid created for each feature. The second element of n gives the size of the sample to be drawn without replacement from the input data. Setting n[2] less than the number of rows in the input will decrease computation time. The default for n[1] is 10, and the default for n[2] is the number of rows in the input.

... additional arguments to be passed to mmpf’s marginalPrediction.

Value

PartialDependenceData. A named list, which contains the partial dependence, input data, target, features, task description, and other arguments controlling the type of partial dependences made.

Object members:

- `data` data.frame
  Has columns for the prediction: one column for regression and survival analysis, and a column for class and the predicted probability for classification as well as a column for each element of features. If individual = TRUE then there is an additional column idx which gives the index of the data that each prediction corresponds to.

- `task.desc` TaskDesc
  Task description.

- `target` Target feature for regression, target feature levels for classification, survival and event indicator for survival.

- `features` character
  Features argument input.

- `interaction` (logical(1))
  Whether or not the features were interacted (i.e. conditioning).

- `derivative` (logical(1))
  Whether or not the partial derivative was estimated.

- `individual` (logical(1))
  Whether the partial dependences were aggregated or the individual curves are retained.

References


generateThreshVsPerfData

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

Description
Generates data on threshold vs. performance(s) for 2-class classification that can be used for plotting.

Usage
`generateThreshVsPerfData(obj, measures, gridsize = 100L, aggregate = TRUE, task.id = NULL)
``

Arguments
- **obj** (list of Prediction | list of ResampleResult | BenchmarkResult)
  Single prediction object, list of them, single resample result, list of them, or a benchmark result. In case of a list probably produced by different learners you want to compare, then name the list with the names you want to see in the plots, probably learner shortnames or ids.
getBMRAggrPerformances

Extract the aggregated performance values from a benchmark result.

Description

Either a list of lists of “aggr” numeric vectors, as returned by resample, or these objects are rbind-ed with extra columns “task.id” and “learner.id”.

Usage

getBMRAggrPerformances(
  bmr,
  task.ids = NULL,
  learner.ids = NULL,
  as.df = FALSE,
  drop = FALSE
)
getBMRFeatSelResults

Arguments

- **bmr**: (BenchmarkResult) Benchmark result.
- **task.ids**: (character(1)) Restrict result to certain tasks. Default is all.
- **learner.ids**: (character(1)) Restrict result to certain learners. Default is all.
- **as.df**: (character(1)) Return one data.frame as result - or a list of lists of objects?. Default is FALSE.
- **drop**: (logical(1)) If drop is FALSE (the default), a nested list with the following structure is returned: res[task.ids][learner.ids]. If drop is set to TRUE it is checked if the list structure can be simplified. If only one learner was passed, a list with entries for each task is returned. If only one task was passed, the entries are named after the corresponding learner. For an experiment with both one task and learner, the whole list structure is removed. Note that the name of the task/learner will be dropped from the return object.

Value

(list | data.frame). See above.

See Also

Other benchmark: BenchmarkResult, batchmark(), benchmark(), convertBMRToRankMatrix(), friedmanPostHocTestBMR(), friedmanTestBMR(), generateCritDifferencesData(), getBMRFeatSelResults(), getBMRFilteredFeatures(), getBMRLearnerIds(), getBMRLearnerShortNames(), getBMRLearners(), getBMRMeasureIds(), getBMRMeasures(), getBMRModels(), getBMRPerformances(), getBMRPredictions(), getBMRTaskDescs(), getBMRTaskIds(), getBRTuneResults(), plotBMRBoxplots(), plotBMRsAsBarChart(), plotBMRSummary(), plotCritDifferences(), reduceBatchmarkResults()

---

**getBMRFeatSelResults**: Extract the feature selection results from a benchmark result.

Description

Returns a nested list of FeatSelResults. The first level of nesting is by data set, the second by learner, the third for the benchmark resampling iterations. If as.df is TRUE, a data frame with “task.id”, “learner.id”, the resample iteration and the selected features is returned.

Note that if more than one feature is selected and a data frame is requested, there will be multiple rows for the same dataset-learner-iteration; one for each selected feature.
getBMRFeatSelResults

Usage

getBMRFeatSelResults(
  bmr,
  task.ids = NULL,
  learner.ids = NULL,
  as.df = FALSE,
  drop = FALSE
)

Arguments

bmr (BenchmarkResult)
Benchmark result.

task.ids (character(1))
Restrict result to certain tasks. Default is all.

learner.ids (character(1))
Restrict result to certain learners. Default is all.

as.df (character(1))
Return one data.frame as result - or a list of lists of objects?. Default is FALSE.

drop (logical(1))
If drop is FALSE (the default), a nested list with the following structure is returned:
res[task.ids][learner.ids].
If drop is set to TRUE it is checked if the list structure can be simplified.
If only one learner was passed, a list with entries for each task is returned.
If only one task was passed, the entries are named after the corresponding learner.
For an experiment with both one task and learner, the whole list structure is removed.
Note that the name of the task/learner will be dropped from the return object.

Value

(list | data.frame). See above.

See Also

Other benchmark: BenchmarkResult, batchmark(), benchmark(), convertBMRToRankMatrix(),
friedmanPostHocTestBMR(), friedmanTestBMR(), generateCritDifferencesData(), getBMRAggrPerformances(),
getBMRFilteredFeatures(), getBMR LearnerIds(), getBMR Learner ShortNames(), getBMR Learners(),
getBMR MeasureIds(), getBMR Measures(), getBMR Models(), getBMR Performances(), getBMRPredictions(),
getBMR TaskDescs(), getBMRTaskIds(), getBMRTuneResults(), plotBMRRBoxplots(), plotBMRRanksAsBarChart(),
plotBMR Summary(), plotCritDifferences(), reduceBatchmarkResults()
getBMRFilteredFeatures

Extract the feature selection results from a benchmark result.

Description

Returns a nested list of characters The first level of nesting is by data set, the second by learner, the third for the benchmark resampling iterations. The list at the lowest level is the list of selected features. If as.df is TRUE, a data frame with “task.id”, “learner.id”, the resample iteration and the selected features is returned.

Note that if more than one feature is selected and a data frame is requested, there will be multiple rows for the same dataset-learner-iteration; one for each selected feature.

Usage

getBMRFilteredFeatures(
  bmr,
  task.ids = NULL,
  learner.ids = NULL,
  as.df = FALSE,
  drop = FALSE
)

Arguments

bmr (BenchmarkResult) Benchmark result.
task.ids (character(1)) Restrict result to certain tasks. Default is all.
learner.ids (character(1)) Restrict result to certain learners. Default is all.
as.df (character(1)) Return one data.frame as result - or a list of lists of objects?. Default is FALSE.
drop (logical(1)) If drop is FALSE (the default), a nested list with the following structure is returned:

res[[task.ids]][[learner.ids]].
If drop is set to TRUE it is checked if the list structure can be simplified.
If only one learner was passed, a list with entries for each task is returned.
If only one task was passed, the entries are named after the corresponding learner.
For an experiment with both one task and learner, the whole list structure is removed.
Note that the name of the task/learner will be dropped from the return object.
getBMRLearnerIds

Value

(list | data.frame). See above.

See Also

Other benchmark: BenchmarkResult, batchmark(), benchmark(), convertBMRToRankMatrix(),
friedmanPostHocTestBMR(), friedmanTestBMR(), generateCritDifferencesData(),
getBMRAggrPerformances(),
getBMRFeatSelResults(), getBMRLearnerIds(), getBMRLearnerShortNames(), getBMRLearners(),
getBMRMeasureIds(), getBMRMeasures(), getBMRModels(), getBMRPerformances(),
generateCritDifferencesData().

Description

Gets the IDs of the learners used in a benchmark experiment.

Usage

getBMRLearnerIds(bmr)

Arguments

bmr (BenchmarkResult)
Benchmark result.

Value

(character).

See Also

Other benchmark: BenchmarkResult, batchmark(), benchmark(), convertBMRToRankMatrix(),
friedmanPostHocTestBMR(), friedmanTestBMR(), generateCritDifferencesData(),
getBMRAggrPerformances(),
getBMRFeatSelResults(), getBMRLearnerIds(), getBMRLearnerShortNames(), getBMRLearners(),
getBMRMeasureIds(), getBMRMeasures(), getBMRModels(), getBMRPerformances(),
generateCritDifferencesData().
getBMRLearnShortNames

Return learner short.names used in benchmark.

Description

Gets the learner short.names of the learners used in a benchmark experiment.

Usage

getBMRLearnShortNames(bmr)

Arguments

bmr (BenchmarkResult)
Benchmark result.

Value

(list).

See Also

Other benchmark: BenchmarkResult, batchmark(), benchmark(), convertBMRToRankMatrix(),
friedmanPostHocTestBMR(), friedmanTestBMR(), generateCritDifferencesData(), getBMRAggrPerformances(),
getBMRFeatSelResults(), getBMRFilteredFeatures(), getBMRLearnIds(), getBMRLearnShortNames(),
getBMRMeasureIds(), getBMRMeasures(), getBMRModels(), getBMRPerformances(), getBMRPredictions(),
getBMRTaskDescs(), getBMRTaskIds(), getBRTuneResults(), plotBMRBoxplots(), plotBMRanksAsBarChart(),
plotBMRSummary(), plotCritDifferences(), reduceBatchmarkResults()
Value

(character).

See Also

Other benchmark: BenchmarkResult, batchmark(), benchmark(), convertBMRToRankMatrix(),
friedmanPostHocTestBMR(), friedmanTestBMR(), generateCritDifferencesData(), getBMRAggrPerformances(),
getBMRFeatSelResults(), getBMRFilteredFeatures(), getBMRLearnerIds(), getBMRLearners(),
getBMRMeasureIds(), getBMRMeasures(), getBMRModels(), getBMRPerformances(), getBMRPredictions(),
getBMRTaskDescs(), getBMRTaskIds(), getBMRTuneResults(), plotBMROboxplots(), plotBMRRanksAsBarChart(),
plotBMRSummary(), plotCritDifferences(), reduceBatchmarkResults()
### getBMRMeasures

Return measures used in benchmark.

**Description**

Gets the measures used in a benchmark experiment.

**Usage**

```r
getBMRMeasures(bmr)
```

**Arguments**

- `bmr` *(BenchmarkResult)*
  
  Benchmark result.

**Value**

*(list)*. See above.

**See Also**

Other benchmark: `BenchmarkResult`, `batchmark()`, `convertBMRToRankMatrix()`, `friedmanPostHocTestBMR()`, `friedmanTestBMR()`, `generateCritDifferencesData()`, `getBMRAggrPerformances()`, `getBMRFeatSelResults()`, `getBMRFilteredFeatures()`, `getBMR LearnerIds()`, `getBMR Learner ShortNames()`, `getBMR Learners()`, `getBMR MeasureIds()`, `getBMR Models()`, `getBMR Performances()`, `getBMR Predictions()`, `getBMRTaskDescs()`, `getBMRTaskIds()`, `getBMRTuneResults()`, `plotBMRBoxplots()`, `plotBMRRanksAsBarChart()`, `plotBMRSummary()`, `plotCritDifferences()`, `reduceBatchmarkResults()`

---

### getBMRModels

Extract all models from benchmark result.

**Description**

A list of lists containing all WrappedModels trained in the benchmark experiment.

If models is FALSE in the call to `benchmark`, the function will return NULL.

**Usage**

```r
getBMRModels(bmr, task.ids = NULL, learner.ids = NULL, drop = FALSE)
```
**getBMRPerformances**

*Extract the test performance values from a benchmark result.*

**Arguments**

- **bmr** *(BenchmarkResult)*
  Benchmark result.

- **task.ids** *(character(1))*
  Restrict result to certain tasks. Default is all.

- **learner.ids** *(character(1))*
  Restrict result to certain learners. Default is all.

- **drop** *(logical(1))*
  If drop is FALSE (the default), a nested list with the following structure is returned:
  res[task.ids][learner.ids].
  If drop is set to TRUE it is checked if the list structure can be simplified.
  If only one learner was passed, a list with entries for each task is returned.
  If only one task was passed, the entries are named after the corresponding learner.
  For an experiment with both one task and learner, the whole list structure is removed.
  Note that the name of the task/learner will be dropped from the return object.

**Value**

*(list).*

**See Also**

Other benchmark: `BenchmarkResult`, `batchmark()`, `benchmark()`, `convertBMRToRankMatrix()`, `friedmanPostHocTestBMR()`, `friedmanTestBMR()`, `generateCritDifferencesData()`, `getBMRAggrPerformances()`, `getBMRFeatSelResults()`, `getBMRFilteredFeatures()`, `getBMRLearnerIds()`, `getBMRLearnerShortNames()`, `getBMRLearners()`, `getBMRMeasureIds()`, `getBMRMeasures()`, `getBMRPerformances()`, `getBMRPredictions()`, `getBMRTaskDescs()`, `getBMRTaskIds()`, `getBMRTuneResults()`, `plotBMRBoxplots()`, `plotBMRRanksAsBarChart()`, `plotBMRSummary()`, `plotCritDifferences()`, `reduceBatchmarkResults()`

**Description**

Either a list of lists of “measure.test” data.frames, as returned by resample, or these objects are rbind-ed with extra columns “task.id” and “learner.id”.

**Usage**

```r
getBMRPerformances(
  bmr,
  task.ids = NULL,
  learner.ids = NULL,
)```
getBMRPredictions

Extract the predictions from a benchmark result.

Arguments

- `bmr` (BenchmarkResult): Benchmark result.
- `task.ids` (character(1)): Restrict result to certain tasks. Default is all.
- `learner.ids` (character(1)): Restrict result to certain learners. Default is all.
- `as.df` (character(1)): Return one data.frame as result - or a list of lists of objects?. Default is FALSE.
- `drop` (logical(1)): If drop is FALSE (the default), a nested list with the following structure is returned:
  `res[task.ids][learner.ids]`. If drop is set to TRUE it is checked if the list structure can be simplified.
  If only one learner was passed, a list with entries for each task is returned.
  If only one task was passed, the entries are named after the corresponding learner.
  For an experiment with both one task and learner, the whole list structure is removed.
  Note that the name of the task/learner will be dropped from the return object.

Value

(list | data.frame). See above.

See Also

Other benchmark: BenchmarkResult, batchmark(), benchmark(), convertBMRToRankMatrix(),
friedmanPostHocTestBMR(), friedmanTestBMR(), generateCritDifferencesData(), getBMRAggrPerformances(),
getBMRFeatSelResults(), getBMRFilteredFeatures(), getBMRLearnerIds(), getBMRLearnerShortNames(),
getBMRLearners(), getBMRMergeIds(), getBMRMergeMeasures(), getBMRMergeModels(),
getBMRTaskDescs(), getBMRTaskIds(), getBMRTuneResults(), plotBMRBoxplots(), plotBMRRanksAsBarChart(),
plotBMRSummary(), plotCritDifferences(), reduceBatchmarkResults()
getBMRPredictions

Usage

getBMRPredictions(
  bmr,
  task.ids = NULL,
  learner.ids = NULL,
  as.df = FALSE,
  drop = FALSE
)

Arguments

bmr (BenchmarkResult)
Benchmark result.

task.ids (character(1))
Restrict result to certain tasks. Default is all.

learner.ids (character(1))
Restrict result to certain learners. Default is all.

as.df (character(1))
Return one data.frame as result - or a list of lists of objects?. Default is FALSE.

drop (logical(1))
If drop is FALSE (the default), a nested list with the following structure is returned:
res[task.ids][learner.ids].
If drop is set to TRUE it is checked if the list structure can be simplified.
If only one learner was passed, a list with entries for each task is returned.
If only one task was passed, the entries are named after the corresponding learner.
For an experiment with both one task and learner, the whole list structure is removed.
Note that the name of the task/learner will be dropped from the return object.

Value

(list | data.frame). See above.

See Also

Other benchmark: BenchmarkResult, batchmark(), benchmark(), convertBMRToRankMatrix(),
friedmanPostHocTestBMR(), friedmanTestBMR(), generateCritDifferencesData(), getBMRAggrPerformances(),
getBMRFeatSelResults(), getBMRFilteredFeatures(), getBMR LearnerIds(), getBMR LearnerShortNames(),
getBMLearners(), getBMRMeasureIds(), getBMR Measures(), getBMR Models(), getBMR Performances(),
getBMR TaskDescs(), getBMR TaskIds(), getBRTuneResults(), plotBMR Boxplots(), plotBMR RanksAsBarChart(),
plotBMR Summary(), plotCritDifferences(), reduceBatchmarkResults()
getBMRTaskDescriptions

Extract all task descriptions from benchmark result (DEPRECATED).

**Description**

A list containing all TaskDescs for each task contained in the benchmark experiment.

**Usage**

getBMRTaskDescriptions(bmr)

**Arguments**

- **bmr** (BenchmarkResult)
  Benchmark result.

**Value**

(list).

---

getBMRTaskDescs

Extract all task descriptions from benchmark result.

**Description**

A list containing all TaskDescs for each task contained in the benchmark experiment.

**Usage**

getBMRTaskDescs(bmr)

**Arguments**

- **bmr** (BenchmarkResult)
  Benchmark result.

**Value**

(list).
getBMRTaskIds

See Also

Other benchmark: BenchmarkResult, benchmark(), benchmark(), convertBMRToRankMatrix(), friedmanPostHocTestBMR(), friedmanTestBMR(), generateCritDifferencesData(), getBMRAggrPerformances(), getBMRFeatSelResults(), getBMRFilteredFeatures(), getBMRFeatSelResults(), getBMRLearnerIds(), getBMRLearnerShortNames(), getBMRLearners(), getBMRLearnerIds(), getBMRMeasureIds(), getBMRLearners(), getBMRModels(), getBMRPerformances(), getBMRPredictions(), getBMRTaskIds(), getBMRTaskDescs(), getBMRTuneResults(), plotBMRRanksAsBarChart(), plotBMRSummary(), plotCritDifferences(), reduceBatchmarkResults()

---

getBMRTaskIds  Return task ids used in benchmark.

Description

Gets the task IDs used in a benchmark experiment.

Usage

getBMRTaskIds(bmr)

Arguments

bmr  (BenchmarkResult)
    Benchmark result.

Value

(character).

See Also

Other benchmark: BenchmarkResult, benchmark(), benchmark(), convertBMRToRankMatrix(), friedmanPostHocTestBMR(), friedmanTestBMR(), generateCritDifferencesData(), getBMRAggrPerformances(), getBMRFeatSelResults(), getBMRFiteredFeatures(), getBMRFeatSelResults(), getBMRLearnerIds(), getBMRLearnerShortNames(), getBMRLearners(), getBMRMeasureIds(), getBMRLearners(), getBMRModels(), getBMRPerformances(), getBMRPredictions(), getBMRTaskDescs(), getBMRTuneResults(), plotBMRRanksAsBarChart(), plotBMRSummary(), plotCritDifferences(), reduceBatchmarkResults()
getBMRTuneResults  

Extract the tuning results from a benchmark result.

Description

Returns a nested list of TuneResults. The first level of nesting is by data set, the second by learner, the third for the benchmark resampling iterations. If \texttt{as.df} is \texttt{TRUE}, a data frame with the “task.id”, “learner.id”, the resample iteration, the parameter values and the performances is returned.

Usage

\begin{verbatim}
getBMRTuneResults(
  bmr,  
  task.ids = NULL,  
  learner.ids = NULL,  
  as.df = FALSE,  
  drop = FALSE
)
\end{verbatim}

Arguments

\begin{description}
\item[bmr] \texttt{(BenchmarkResult)} Benchmark result.
\item[task.ids] \texttt{(character(1))} Restrict result to certain tasks. Default is all.
\item[learner.ids] \texttt{(character(1))} Restrict result to certain learners. Default is all.
\item[as.df] \texttt{(logical(1))} Return one \texttt{data.frame} as result - or a list of lists of objects?. Default is \texttt{FALSE}.
\item[drop] \texttt{(logical(1))} If drop is \texttt{FALSE} (the default), a nested list with the following structure is returned: 
\texttt{res[[task.ids]][[learner.ids]].}
If drop is set to \texttt{TRUE} it is checked if the list structure can be simplified. If only one learner was passed, a list with entries for each task is returned. If only one task was passed, the entries are named after the corresponding learner. For an experiment with both one task and learner, the whole list structure is removed. Note that the name of the task/learner will be dropped from the return object.
\end{description}

Value

\texttt{(list | data.frame)}. See above.
**getCaretParamSet**

*Get tuning parameters from a learner of the caret R-package.*

**Description**

Constructs a grid of tuning parameters from a learner of the caret R-package. These values are then converted into a list of non-tunable parameters (par.vals) and a tunable ParamHelpers::ParamSet (par.set), which can be used by tuneParams for tuning the learner. Numerical parameters will either be specified by their lower and upper bounds or they will be discretized into specific values.

**Usage**

```r
getCaretParamSet(learner, length = 3L, task, discretize = TRUE)
```

**Arguments**

- **learner** (character(1))
  The name of the learner from caret (cf. 
  [https://topepo.github.io/caret/available-models.html](https://topepo.github.io/caret/available-models.html)).
  Note that the names in caret often differ from the ones in mlr.

- **length** (integer(1))
  A length / precision parameter which is used by caret for generating the grid of tuning parameters. caret generates either as many values per tuning parameter / dimension as defined by length or only a single value (in case of non-tunable par.vals).

- **task** (Task)
  Learning task, which might be requested for creating the tuning grid.

- **discretize** (logical(1))
  Should the numerical parameters be discretized? Alternatively, they will be defined by their lower and upper bounds. The default is TRUE.

**Value**

(list(2)). A list of parameters:

- par.vals contains a list of all constant tuning parameters
- par.set is a ParamHelpers::ParamSet, containing all the configurable tuning parameters

**See Also**

Other benchmark: BenchmarkResult, batchmark(), benchmark(), convertBMRToRankMatrix(), friedmanPostHocTestBMR(), friedmanTestBMR(), generateCritDifferencesData(), getBMRAggrPerformances(), getBMRFeatSelResults(), getBMRFilteredFeatures(), getBMR LearnerIds(), getBMR Learner ShortNames(), getBMR Learners(), getBMR Meas ureIds(), getBMR Measures(), getBMR Models(), getBMR Performances(), getBMR Predictions(), getBMR TaskDescs(), getBMR TaskIds(), plotBMRBoxplots(), plotBMR Ranks AsBarChart(), plotBMR Summary(), plotCrit Differences(), reduceBatchmarkResults()
**Examples**

```r
if (requireNamespace("caret") && requireNamespace("mlbench")) {
  library(caret)
  classifTask = makeClassifTask(data = iris, target = "Species")
  # (1) classification (random forest) with discretized parameters
  getCaretParamSet("rf", length = 9L, task = classifTask, discretize = TRUE)

  # (2) regression (gradient boosting machine) without discretized parameters
  library(mlbench)
  data(BostonHousing)
  regrTask = makeRegrTask(data = BostonHousing, target = "medv")
  getCaretParamSet("gbm", length = 9L, task = regrTask, discretize = FALSE)
}
```

---

**(getClassWeightParam)**

*Get the class weight parameter of a learner.*

---

**Description**

Gets the class weight parameter of a learner.

**Usage**

```r
getClassWeightParam(learner, lrn.id = NULL)
```

**Arguments**

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

- **lrn.id**
  ```r
  (character)
  ```
  Only used for `BaseEnsembles`. It is possible that multiple learners in a base ensemble have a class weight param. Specify the learner from which the class weight should be extracted.

**Value**

- **numeric LearnerParam**: A numeric parameter object, containing the class weight parameter of the given learner.

**See Also**

Other learner: `LearnerProperties`, `getHyperPars()`, `getLearnerId()`, `getLearnerNote()`, `getLearnerPackages()`, `getLearnerParVals()`, `getLearnerParamSet()`, `getLearnerPredictType()`, `getLearnerShortName()`, `getLearnerType()`, `getParamSet()`, `helpLearner()`, `helpLearnerParam()`, `makeLearner()`, `makeLearners()`, `removeHyperPars()`, `setHyperPars()`, `setId()`, `setLearnerId()`, `setPredictThreshold()`, `setPredictType()`
getConfMatrix

Confusion matrix.

Description

getConfMatrix is deprecated. Please use calculateConfusionMatrix.

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

The marginal elements count the number of classification errors for the respective row or column, i.e., the number of errors when you condition on the corresponding true (rows) or predicted (columns) class. The last element in the margin diagonal displays the total amount of errors.

Note that for resampling no further aggregation is currently performed. All predictions on all test sets are joined to a vector yhat, as are all labels joined to a vector y. Then yhat is simply tabulated vs y, as if both were computed on a single test set. This probably mainly makes sense when cross-validation is used for resampling.

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.

Value

(matrix). A confusion matrix.

See Also

predict.WrappedModel
getDefaultMeasure  
*Get default measure.*

**Description**

Get the default measure for a task type, task, task description or a learner. Currently these are:

- classif: mmce
- regr: mse
- cluster: db
- surv: cindex
- costsen: mcp
- multilabel: multilabel.hamloss

**Usage**

```r
defaultMeasure(x)
```

**Arguments**

- **x**  
  (character(1) | Task | TaskDesc | Learner)
  Task type, task, task description, learner name, a learner, or a type of learner (e.g. "classif").

**Value**

(Measure).

getFailureModelDump  
*Return the error dump of FailureModel.*

**Description**

Returns the error dump that can be used with debugger() to evaluate errors. If configureMlr configuration on.error.dump is FALSE, this returns NULL.

**Usage**

```r
defaultMeasureDump(model)
```

**Arguments**

- **model**  
  (WrappedModel)
  The model.

**Value**

(last.dump).
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 message 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.

Description

Returns the selected feature set and optimization path after training.

Usage

getFeatSelResult(object)

Arguments

object (WrappedModel)
Trained Model created with makeFeatSelWrapper.

Value

(FeatSelResult).

See Also

Other featsel: FeatSelControl, analyzeFeatSelResult(), makeFeatSelWrapper(), selectFeatures()
getFeatureImportance  Calculates feature importance values for trained models.

Description

For some learners it is possible to calculate a feature importance measure. getFeatureImportance extracts those values from trained models. See below for a list of supported learners.

Usage

getFeatureImportance(object, ...)

Arguments

object  (WrappedModel)
Wrapped model, result of train().

...  (any)
Additional parameters, which are passed to the underlying importance value generating function.

Details

- boosting
  Measure which accounts the gain of Gini index given by a feature in a tree and the weight of that tree.

- cforest
  Permutation principle of the 'mean decrease in accuracy' principle in randomForest. If auc=TRUE (only for binary classification), area under the curve is used as measure. The algorithm used for the survival learner is 'extremely slow and experimental; use at your own risk'. See party::varimp() for details and further parameters.

- gbm
  Estimation of relative influence for each feature. See gbm::relative.influence() for details and further parameters.

- h2o
  Relative feature importances as returned by h2o::h2o.varimp().

- randomForest
  For type = 2 (the default) the 'MeanDecreaseGini' is measured, which is based on the Gini impurity index used for the calculation of the nodes. Alternatively, you can set type to 1, then the measure is the mean decrease in accuracy calculated on OOB data. Note, that in this case the learner's parameter importance needs to be set to be able to compute feature importance values. See randomForest::importance() for details.

- RRF
  This is identical to randomForest.
• ranger
  Supports both measures mentioned above for the randomForest learner. Note, that you need to specifically set the learners parameter importance, to be able to compute feature importance measures. See \texttt{ranger::importance()} and \texttt{ranger::ranger()} for details.

• rpart
  Sum of decrease in impurity for each of the surrogate variables at each node

• xgboost
  The value implies the relative contribution of the corresponding feature to the model calculated by taking each feature’s contribution for each tree in the model. The exact computation of the importance in xgboost is undocumented.

Value

(FeatureImportance) An object containing a \texttt{data.frame} of the variable importances and further information.

getFilteredFeatures

\texttt{getFilteredFeatures} \hspace{1em} \textit{Returns the filtered features.}

Description

Returns the filtered features.

Usage

getFilteredFeatures(model)

Arguments

model \hspace{1em} \texttt{(WrappedModel)}
  Trained Model created with \texttt{makeFilterWrapper}.

Value

(character).

See Also

Other filter: \texttt{filterFeatures()}, \texttt{generateFilterValuesData()}, \texttt{listFilterEnsembleMethods()}, \texttt{listFilterMethods()}, \texttt{makeFilter()}, \texttt{makeFilterEnsemble()}, \texttt{makeFilterWrapper()}, \texttt{plotFilterValues()}
**getFunctionalFeatures**  
*Get only functional features from a task or a data.frame.*

**Description**

The parameters “subset”, “features”, and “recode.target” are ignored for the data.frame method.

**Usage**

```r
getFunctionalFeatures(object, subset = NULL, features, recode.target = "no")
## S3 method for class 'Task'
getFunctionalFeatures(object, subset = NULL, features, recode.target = "no")
## S3 method for class 'data.frame'
getFunctionalFeatures(object, subset = NULL, features, recode.target = "no")
```

**Arguments**

- `object` *(Task/data.frame)*  
  Object to check on.

- `subset` *(integer | logical | NULL)*  
  Selected cases. Either a logical or an index vector. By default NULL if all observations are used.

- `features` *(character | integer | logical)*  
  Vector of selected inputs. You can either pass a character vector with the feature names, a vector of indices, or a logical vector. In case of an index vector each element denotes the position of the feature name returned by `getTaskFeatureNames`. Note that the target feature is always included in the resulting task, you should not pass it here. Default is to use all features.

- `recode.target` *(character(1))*  
  Should target classes be recoded? Supported are binary and multilabel classification and survival. Possible values for binary classification are “01”, “-1+1” and “drop.levels”. 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”. “drop.levels” will remove empty factor levels in the target column. In the multilabel case the logical targets can be converted to factors with “multilabel.factor”. For survival, you may choose to recode the survival times to “left”, “right” or “interval2” censored times using “lcens”, “rcens” or “icens”, respectively. See `survival::Surv` for the format specification. Default for both binary classification and survival is “no” (do nothing).

**Value**

Returns a data.frame containing only the functional features.
**getHomogeneousEnsembleModels**

*Deprecated, use getLearnerModel instead.*

**Description**

Deprecated, use getLearnerModel instead.

**Usage**

```r
getHomogeneousEnsembleModels(model, learner.models = FALSE)
```

**Arguments**

- `model` (Deprecated).
- `learner.models` (Deprecated).

---

**getHyperPars**

*Get current parameter settings for a learner.*

**Description**

Retrieves the current hyperparameter settings of a learner.

**Usage**

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

**Arguments**

- `learner` *(Learner)*
  The learner.
- `for.fun` *(character(1))*
  Restrict the returned settings to hyperparameters corresponding to when the are used (see `ParamHelpers::LearnerParam`). Must be a subset of: “train”, “predict” or “both”. Default is `c("train", "predict", "both")`.

**Details**

This function only shows hyperparameters that differ from the learner default (because mlr changed the default) or if the user set hyperparameters manually during learner creation. If you want to have an overview of all available hyperparameters use `getParamSet()`.

**Value**

*(list). A named list of values.*
getLearnerId

**See Also**

Other learner: LearnerProperties, getLearnerId(), getClassWeightParam(), getLearnerNote(), getLearnerPackages(), getLearnerParVals(), getLearnerParamSet(), getLearnerPredictType(), getLearnerShortName(), getLearnerType(), getParamSet(), helpLearner(), helpLearnerParam(), makeLearner(), makeLearners(), removeHyperPars(), setHyperPars(), setId(), setLearnerId(), setPredictThreshold(), setPredictType()

**Examples**

getHyperPars(makeLearner("classif.ranger"))

## set learner hyperparameter `mtry` manually
getHyperPars(makeLearner("classif.ranger", mtry = 100))

---

**getLearnerId**  
*Get the ID of the learner.*

**Description**

Get the ID of the learner.

**Usage**

getLearnerId(learner)

**Arguments**

learner

*(Learner | character(1))*

The learner. If you pass a string the learner will be created via `makeLearner`.

**Value**

(character(1)).

**See Also**

Other learner: LearnerProperties, getLearnerId(), getClassWeightParam(), getHyperPars(), getLearnerNote(), getLearnerPackages(), getLearnerParVals(), getLearnerParamSet(), getLearnerPredictType(), getLearnerShortName(), getLearnerType(), getParamSet(), helpLearner(), helpLearnerParam(), makeLearner(), makeLearners(), removeHyperPars(), setHyperPars(), setId(), setLearnerId(), setPredictThreshold(), setPredictType()
getLearnerModel

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

**Description**

Get underlying R model of learner integrated into mlr.

**Usage**

```r
getLearnerModel(model, more.unwrap = FALSE)
```

**Arguments**

- `model` *(WrappedModel)*
  
  The model, returned by e.g., `train`.

- `more.unwrap` *(logical(1))*
  
  Some learners are not basic learners from R, but implemented in mlr as meta-techniques. Examples are everything that inherits from `HomogeneousEnsemble`. In these cases, the `learner.model` is often a list of mlr `WrappedModels`. This option allows to strip them further to basic R models. The option is simply ignored for basic learner models. Default is `FALSE`.

**Value**

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

getLearnerNote

*Get the note for the learner.*

**Description**

Get the note for the learner.

**Usage**

```r
getLearnerNote(learner)
```

**Arguments**

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

**Value**

*(character).*
getLearnerParamSet

See Also

Other learner: LearnerProperties, getClassWeightParam(), getHyperPars(), getLearnerId(), getLearnerPackages(), getLearnerParVals(), getLearnerParamSet(), getLearnerPredictType(), getLearnerShortName(), getLearnerType(), getParamSet(), helpLearner(), helpLearnerParam(), makeLearner(), makeLearners(), removeHyperPars(), setHyperPars(), setId(), setLearnerId(), setPredictThreshold(), setPredictType()

getLearnerPackages

Get the required R packages of the learner.

Description

Get the R packages the learner requires.

Usage

getLearnerPackages(learner)

Arguments

learner (Learner | character(1))

The learner. If you pass a string the learner will be created via makeLearner.

Value

(character).

See Also

Other learner: LearnerProperties, getClassWeightParam(), getHyperPars(), getLearnerId(), getLearnerNote(), getLearnerParVals(), getLearnerParamSet(), getLearnerPredictType(), getLearnerShortName(), getLearnerType(), getParamSet(), helpLearner(), helpLearnerParam(), makeLearner(), makeLearners(), removeHyperPars(), setHyperPars(), setId(), setLearnerId(), setPredictThreshold(), setPredictType()

getLearnerParamSet

Get the parameter set of the learner.

Description

Alias for getParamSet.

Usage

getLearnerParamSet(learner)
getLearnerParVals

Arguments

learner (Learner \text{\mid} \text{character}(1))
The learner. If you pass a string the learner will be created via makeLearner.

Value

ParamSet.

See Also

Other learner: LearnerProperties, getClassWeightParam(), getHyperPars(), getLearnerId(),
getLearnerNote(), getLearnerPackages(), getLearnerParVals(), getLearnerPredictType(),
getLearnerShortName(), getLearnerType(), getParamSet(), helpLearner(), helpLearnerParam(),
makeLearner(), makeLearners(), removeHyperPars(), setHyperPars(), setId(), setLearnerId(),
setPredictThreshold(), setPredictType()

getLearnerParVals \hspace{1cm} \textit{Get the parameter values of the learner.}

Description

Alias for getHyperPars.

Usage

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

Arguments

learner (Learner \text{\mid} \text{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 they are used (see ParamHelpers::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: LearnerProperties, getClassWeightParam(), getHyperPars(), getLearnerId(),
getLearnerNote(), getLearnerPackages(), getLearnerParamSet(), getLearnerPredictType(),
getLearnerShortName(), getLearnerType(), getParamSet(), helpLearner(), helpLearnerParam(),
makeLearner(), makeLearners(), removeHyperPars(), setHyperPars(), setId(), setLearnerId(),
setPredictThreshold(), setPredictType()
getLearnerPredictType  
*Get the predict type of the learner.*

**Description**

Get the predict type of the learner.

**Usage**

```r
getLearnerPredictType(learner)
```

**Arguments**

- **learner**  
  
  *(Learner | character(1))*

  The learner. If you pass a string the learner will be created via `makeLearner`.

**Value**

*(character(1)).*

**See Also**

Other learner: `LearnerProperties`, `getClassWeightParam()`, `getHyperPars()`, `getLearnerId()`, `getLearnerNote()`, `getLearnerPackages()`, `getLearnerParVals()`, `getLearnerParamSet()`, `getLearnerShortName()`, `getLearnerType()`, `getParamSet()`, `helpLearner()`, `helpLearnerParam()`, `makeLearner()`, `makeLearners()`, `removeHyperPars()`, `setHyperPars()`, `setLearnerId()`, `setPredictThreshold()`, `setPredictType()`

getLearnerShortName  
*Get the short name of the learner.*

**Description**

For an ordinary learner simply its short name is returned. For wrapped learners, the wrapper id is successively attached to the short name of the base learner. E.g: “rf.bagged.imputed”

**Usage**

```r
getLearnerShortName(learner)
```

**Arguments**

- **learner**  
  
  *(Learner | character(1))*

  The learner. If you pass a string the learner will be created via `makeLearner`. 
**getLearnerType**

Get the type of the learner.

### Description

Get the type of the learner.

### Usage

```r
getLearnerType(learner)
```

### Arguments

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

### Value

(character(1)).

### See Also

Other learner: `LearnerProperties`, `getClassWeightParam()`, `getHyperPars()`, `getLearnerId()`, `getLearnerNote()`, `getLearnerPackages()`, `getLearnerParVals()`, `getLearnerParamSet()`, `getLearnerPredictType()`, `getLearnerShortName()`, `getParamSet()`, `helpLearner()`, `helpLearnerParam()`, `makeLearner()`, `makeLearners()`, `removeHyperPars()`, `setHyperPars()`, `setId()`, `setLearnerId()`, `setPredictThreshold()`, `setPredictType()`
getMlrOptions

*Returns a list of mlr’s options.*

**Description**

Gets the options for mlr.

**Usage**

```r
getMlrOptions()
```

**Value**

*(list).*

**See Also**

Other configure: `configureMlr()`

getMultilabelBinaryPerformances

*Retrieve binary classification measures for multilabel classification predictions.*

**Description**

Measures the quality of each binary label prediction w.r.t. some binary classification performance measure.

**Usage**

```r
getMultilabelBinaryPerformances(pred, measures)
```

**Arguments**

- `pred` *(Prediction)*
  Multilabel Prediction object.

- `measures` *(Measure | list of Measure)*
  Performance measure(s) to evaluate, must be applicable to binary classification performance. Default is `mmce`.

**Value**

*(named matrix).* Performance value(s), column names are measure(s), row names are labels.
getNestedTuneResultsOptPathDf

See Also

Other multilabel: makeMultilabelBinaryRelevanceWrapper(), makeMultilabelClassifierChainsWrapper(), makeMultilabelDBRWrapper(), makeMultilabelNestedStackingWrapper(), makeMultilabelStackingWrapper()

Examples

# see makeMultilabelBinaryRelevanceWrapper

getNestedTuneResultsOptPathDf

Get the opt.path.s from each tuning step from the outer resampling.

Description

After you resampled a tuning wrapper (see makeTuneWrapper) with resample(..., extract = getTuneResult) this helper returns a data.frame with with all opt.path.s combined by rbind. An additional column iter indicates to what resampling iteration the row belongs.

Usage

getNestedTuneResultsOptPathDf(r, trafo = FALSE)

Arguments

r

(ResampleResult)
The result of resampling of a tuning wrapper.

trafo

(logical(1))
Should the units of the hyperparameter path be converted to the transformed scale? This is only necessary when trafo was used to create the opt.path.s. Note that opt.path.s are always stored on the untransformed scale. Default is FALSE.

Value

(data.frame). See above.

See Also

Other tune: TuneControl, getNestedTuneResultsX(), getResamplingIndices(), getTuneResult(), makeModelMultiplexer(), makeModelMultiplexerParamSet(), makeTuneControlCMAES(), makeTuneControlDesign(), makeTuneControlGenSA(), makeTuneControlGrid(), makeTuneControlIrace(), makeTuneControlMBO(), makeTuneControlRandom(), makeTuneWrapper(), tuneParams(), tuneThreshold()

Examples

# see example of makeTuneWrapper
getNestedTuneResultsX  Get the tuned hyperparameter settings from a nested tuning.

Description
After you resampled a tuning wrapper (see makeTuneWrapper) with resample(..., extract = getTuneResult) this helper returns a data.frame with the best found hyperparameter settings for each resampling iteration.

Usage
getNestedTuneResultsX(r)

Arguments
r  (ResampleResult)
The result of resampling of a tuning wrapper.

Value
(data.frame). One column for each tuned hyperparameter and one row for each outer resampling iteration.

See Also
Other tune: TuneControl, getNestedTuneResultsOptPathDf(), getResamplingIndices(), getTuneResult(), makeModelMultiplexer(), makeModelMultiplexerParamSet(), makeTuneControlCMAES(), makeTuneControlDesign(), makeTuneControlGenSA(), makeTuneControlGrid(), makeTuneControlIrace(), makeTuneControlMBO(), makeTuneControlRandom(), makeTuneWrapper(), tuneParams(), tuneThreshold()

Examples
# see example of makeTuneWrapper

getOOBPreds  Extracts out-of-bag predictions from trained models.

Description
Learners like randomForest produce out-of-bag predictions. getOOBPreds extracts this information from trained models and builds a prediction object as provided by predict (with prediction time set to NA). In the classification case: What is stored exactly in the (Prediction) object depends on the predict.type setting of the Learner.

You can call listLearners(properties = "oobpreds") to get a list of learners which provide this.
Usage

getOOBPreds(model, task)

Arguments

model (WrappedModel)
The model.

task (Task)
The task.

Value

(Prediction).

Examples

training.set = sample(1:150, 50)
lrn = makeLearner("classif.ranger", predict.type = "prob", predict.threshold = 0.6)
mod = train(lrn, sonar.task, subset = training.set)
oob = getOOBPreds(mod, sonar.task)
oob
performance(oob, measures = list(auc, mmce))

getParamSet

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

Description

Returns the ParamHelpers::ParamSet from a Learner.

Value

ParamSet.

See Also

Other learner: LearnerProperties, getClassWeightParam(), getHyperPars(), getLearnerId(),
getLearnerNote(), getLearnerPackages(), getLearnerParVals(), getLearnerParamSet(),
getLearnerPredictType(), getLearnerShortName(), getLearnerType(), helpLearner(), helpLearnerParam(),
makeLearner(), makeLearners(), removeHyperPars(), setHyperPars(), setId(), setLearnerId(),
setPredictThreshold(), setPredictType()
getPredictionDump

Return the error dump of a failed Prediction.

Description

Returns the error dump that can be used with debugger() to evaluate errors. If configureMlr configuration on.error.dump is FALSE or if the prediction did not fail, this returns NULL.

Usage

gетод PredictionDump(pred)

Arguments

pred (Prediction)
Prediction object.

Value

(last.dump).

See Also

Other debug: FailureModel, ResampleResult, getRRDump()

getPredictionProbabilities

Get probabilities for some classes.

Description

Get probabilities for some classes.

Usage

gетод PredictionProbabilities(pred, cl)

Arguments

pred (Prediction)
Prediction object.

cl (character)
Names of classes. Default is either all classes for multi-class / multilabel 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: asROCRPrediction(), getPredictionResponse(), getPredictionTaskDesc(), predict.WrappedModel(), setPredictThreshold(), 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(getPredictionProbabilities(pred))

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

Arguments

pred (Prediction)
Prediction object.

Value
See above.

See Also
Other predict: asROCRPrediction(), getPredictionProbabilities(), getPredictionTaskDesc(), predict.WrappedModel(), setPredictThreshold(), setPredictType()

getPredictionTaskDesc Get summarizing task description from prediction.

Description
See title.

Usage
getPredictionTaskDesc(pred)

Arguments

pred (Prediction)
Prediction object.

Value
ret_taskdesc

See Also
Other predict: asROCRPrediction(), getPredictionProbabilities(), getPredictionResponse(), predict.WrappedModel(), setPredictThreshold(), setPredictType()
getProbabilities

Deprecated, use getPredictionProbabilities instead.

description

getProbabilities(pred, cl)

Arguments

pred  Deprecated.
cl    Deprecated.

getResamplingIndices

Get the resampling indices from a tuning or feature selection wrapper.

description

After you resampled a tuning or feature selection wrapper (see makeTuneWrapper) with resample(..., extract = getTuneResult) or resample(..., extract = getFeatSelResult) this helper returns a list with the resampling indices used for the respective method.

Usage

getResamplingIndices(object, inner = FALSE)

Arguments

object    (ResampleResult)
          The result of resampling of a tuning or feature selection wrapper.
inner     (logical)
          If TRUE, returns the inner indices of a nested resampling setting.

Value

(list). One list for each outer resampling fold.

See Also

Other tune: TuneControl, getNestedTuneResultsOptPathDf(), getNestedTuneResultsX(), getTuneResult(), makeModelMultiplexer(), makeModelMultiplexerParamSet(), makeTuneControlCMAES(), makeTuneControlDesign(), makeTuneControlGenSA(), makeTuneControlGrid(), makeTuneControlIrace(), makeTuneControlMBO(), makeTuneControlRandom(), makeWrapper(), tuneParams(), tuneThreshold()
getRRDump

Return the error dump of ResampleResult.

Description

Returns the error dumps generated during resampling, which can be used with debugger() to debug errors. These dumps are saved if configureMlr configuration on.error.dump, or the corresponding learner config, is TRUE.

The returned object is a list with as many entries as the resampling being used has folds. Each of these entries can have a subset of the following slots, depending on which step in the resampling iteration failed: “train” (error during training step), “predict.train” (prediction on training subset), “predict.test” (prediction on test subset).

Usage

getRRDump(res)

Arguments

res (ResampleResult)
The result of resample.

Value

list.

See Also

Other debug: FailureModel, ResampleResult, getPredictionDump()
**getRRPredictionList**

Get list of predictions for train and test set of each single resample iteration.

### Description
This function creates a list with two slots `train` and `test` where each slot is again a list of `Prediction` objects for each single resample iteration. In case that `predict = "train"` was used for the resample description (see `makeResampleDesc`), the slot `test` will be NULL and in case that `predict = "test"` was used, the slot `train` will be NULL.

### Usage
```r
getRRPredictionList(res, ...)  
```

### Arguments
- **res** *(ResampleResult)*
  The result of `resample` run with `keep.pred = TRUE`.

- **...** *(any)*
  Further options passed to `makePrediction`.

### Value
`list`.

### See Also
Other resample: `ResamplePrediction`, `ResampleResult`, `addRRMeasure()`, `getRRPredictions()`, `getRRTaskDesc()`, `getRRTaskDescription()`, `makeResampleDesc()`, `makeResampleInstance()`, `resample()`

---

**getRRPredictions**

Get predictions from resample results.

### Description
Very simple getter.

### Usage
```r
getRRPredictions(res)
```

### Arguments
- **res** *(ResampleResult)*
  The result of `resample` run with `keep.pred = TRUE`. 
getRRTaskDesc

Value

(ResamplePrediction).

See Also

Other resample: ResamplePrediction, ResampleResult, addRRMeasure(), getRRPredictionList(), getRRTaskDesc(), getRRTaskDescription(), makeResampleDesc(), makeResampleInstance(), resample()

getRRTaskDesc Get task description from resample results (DEPRECATED).

Description

Get a summarizing task description.

Usage

getRRTaskDesc(res)

Arguments

res (ResampleResult)
The result of resample.

Value

(TaskDesc).

See Also

Other resample: ResamplePrediction, ResampleResult, addRRMeasure(), getRRPredictionList(), getRRPredictions(), getRRTaskDescription(), makeResampleDesc(), makeResampleInstance(), resample()
getRRTaskDescription  Get task description from resample results (DEPRECATED).

Description

Get a summarizing task description.

Usage

getRRTaskDescription(res)

Arguments

res  (ResampleResult)

The result of resample.

Value

(TaskDesc).

See Also

Other resample: ResamplePrediction, ResampleResult, addRRMeasure(), getRRPredictionList(), getRRPredictions(), getRRTaskDesc(), makeResampleDesc(), makeResampleInstance(), resample()

getStackedBaseLearnerPredictions

Returns the predictions for each base learner.

Description

Returns the predictions for each base learner.

Usage

getStackedBaseLearnerPredictions(model, newdata = NULL)

Arguments

model  (WrappedModel)

Wrapped model, result of train.

newdata  (data.frame)

New observations, for which the predictions using the specified base learners should be returned. Default is NULL and extracts the base learner predictions that were made during the training.
getTaskClassLevels

Get the class levels for classification and multilabel tasks.

Description

NB: For multilabel, getTaskTargetNames and getTaskClassLevels actually return the same thing.

Usage

getTaskClassLevels(x)

Arguments

x (Task | TaskDesc)
Task or its description object.

Value

(character).

See Also

Other task: getTaskCosts(), getTaskData(), getTaskDesc(), getTaskFeatureNames(), getTaskFormula(), getTaskId(), getTaskNFeats(), getTaskSize(), getTaskTargetNames(), getTaskTargets(), getTaskType(), subsetTask()

getTaskCosts

Extract costs in task.

Description

Returns “NULL” if the task is not of type “costsens”.

Usage

getTaskCosts(task, subset = NULL)

Arguments

task (CostSensTask)
The task.

subset (integer | logical | NULL)
Selected cases. Either a logical or an index vector. By default NULL if all observations are used.
getTaskData

Value
(matrix | NULL).

See Also
Other task: getTaskClassLevels(), getTaskData(), getTaskDesc(), getTaskFeatureNames(), getTaskFormula(), getTaskId(), getTaskNFeats(), getTaskSize(), getTaskTargetNames(), getTaskTargets(), getTaskType(), subsetTask()

getTaskData

Extract data in task.

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

Usage

getTaskData(
  task,
  subset = NULL,
  features,
  target.extra = FALSE,
  recode.target = "no",
  functionals.as = "dфcols"
)

Arguments

task
(Task)
The task.

subset
(integer | logical | NULL)
Selected cases. Either a logical or an index vector. By default NULL if all observations are used.

features
(character | integer | logical)
Vector of selected inputs. You can either pass a character vector with the feature names, a vector of indices, or a logical vector. In case of an index vector each element denotes the position of the feature name returned by getTaskFeatureNames. Note that the target feature is always included in the resulting task, you should not pass it here. Default is to use all features.

target.extra
(logical(1))
Should target vector be returned separately? If not, a single data.frame including the target columns is returned, otherwise a list with the input data.frame and an extra vector or data.frame for the targets. Default is FALSE.
recode.target (character(1))

Should target classes be recoded? Supported are binary and multilabel classification and survival. Possible values for binary classification are “01”, “-1+1” and “drop.levels”. 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”. “drop.levels” will remove empty factor levels in the target column. In the multilabel case the logical targets can be converted to factors with “multilabel.factor”. For survival, you may choose to recode the survival times to “left”, “right” or “interval2” censored times using “lcens”, “rcens” or “icens”, respectively. See survival::Surv for the format specification. Default for both binary classification and survival is “no” (do nothing).

functionals.as (character(1))

How to represents functional features? Option “matrix”: Keep them as matrix columns in the data.frame. Option “dfcols”: Convert them to individual numeric data.frame columns. Default is “dfcols”.

Value

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

See Also

Other task: getTaskClassLevels(), getTaskCosts(), getTaskDesc(), getTaskFeatureNames(), getTaskFormula(), getTaskId(), getTaskNFeats(), getTaskSize(), getTaskTargetNames(), getTaskTargets(), getTaskType(), 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"))

getTaskDesc Get a summarizing task description.

Description

See title.

Usage

getTaskDesc(x)
**getTaskDescription**

**Arguments**

x (Task | TaskDesc)
Task or its description object.

**Value**

ret_taskdesc

**See Also**

Other task: `getTaskClassLevels()`, `getTaskCosts()`, `getTaskData()`, `getTaskFeatureNames()`, `getTaskFormula()`, `getTaskId()`, `getTaskNFeats()`, `getTaskSize()`, `getTaskTargetNames()`, `getTaskTargets()`, `getTaskType()`, `subsetTask()`

---

**getTaskDescription**  
 Deprecated, use `getTaskDesc` instead.

---

**Description**

Deprecated, use `getTaskDesc` instead.

**Usage**

`getTaskDescription(x)`

**Arguments**

x (Task | TaskDesc)
Task or its description object.

---

**getTaskFeatureNames**  
 Get feature names of task.

---

**Description**

Target column name is not included.

**Usage**

`getTaskFeatureNames(task)`

**Arguments**

task (Task)
The task.
getTaskFormula

Value

(character).

See Also

Other task: getTaskClassLevels(), getTaskCosts(), getTaskData(), getTaskDesc(), getTaskFormula(), getTaskId(), getTaskNFeats(), getTaskSize(), getTaskTargetNames(), getTaskTargets(), getTaskType(), subsetTask()

getTaskFormula  Get formula of a task.

Description

This is usually simply <target> ~ . For multilabel it is <target_1> + ... + <target_k> ~.

Usage

getTaskFormula(
  x,
  target = getTaskTargetNames(x),
  explicit.features = FALSE,
  env = parent.frame()
)

Arguments

x  (Task | TaskDesc)
Task or its description object.

target  (character(1))
Left hand side of the formula. Default is defined by task x.

explicit.features  (logical(1))
Should the features (right hand side of the formula) be explicitly listed? Default is FALSE, i.e., they will be represented as "".

env  (environment)
Environment of the formula. Default is parent.frame().

Value

(formula).

See Also

Other task: getTaskClassLevels(), getTaskCosts(), getTaskData(), getTaskDesc(), getTaskFeatureNames(), getTaskId(), getTaskNFeats(), getTaskSize(), getTaskTargetNames(), getTaskTargets(), getTaskType(), subsetTask()
getTaskId

Get the id of the task.

Description
See title.

Usage
getTaskId(x)

Arguments
x (Task | TaskDesc)
Task or its description object.

Value
(character(1)).

See Also
Other task: getTaskClassLevels(), getTaskCosts(), getTaskData(), getTaskDesc(), getTaskFeatureNames(), getTaskFormula(), getTaskNFeats(), getTaskSize(), getTaskTargetNames(), getTaskTargets(), getTaskType(), subsetTask()

getTaskNFeats

Get number of features in task.

Description
See title.

Usage
getTaskNFeats(x)

Arguments
x (Task | TaskDesc)
Task or its description object.

Value
(integer(1)).
getTaskSize

Get number of observations in task.

Description
See title.

Usage
getTaskSize(x)

Arguments
x (Task | TaskDesc)
Task or its description object.

Value
(integer(1)).

See Also
Other task: getTaskClassLevels(), getTaskCosts(), getTaskData(), getTaskDesc(), getTaskFeatureNames(), getTaskFormula(), getTaskId(), getTaskSize(), getTaskTargetNames(), getTaskTargets(), getTaskType(), subsetTask()

getTaskTargetNames

Get the name(s) of the target column(s).

Description
NB: For multilabel, getTaskTargetNames and getTaskClassLevels actually return the same thing.

Usage
getTaskTargetNames(x)

Arguments
x (Task | TaskDesc)
Task or its description object.
getTaskTargets

Value

(character).

See Also

Other task: getTaskClassLevels(), getTaskCosts(), getTaskData(), getTaskDesc(), getTaskFeatureNames(), getTaskFormula(), getTaskId(), getTaskNFeats(), getTaskSize(), getTaskTargets(), getTaskType(), subsetTask()

getTaskTargets Get target data of task.

Description

Get target data of task.

Usage

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

Arguments

task (Task)
The task.

recode.target (character(1))
Should target classes be recoded? Supported are binary and multilabel classification and survival. Possible values for binary classification are “01”, “-1+1” and “drop.levels”. 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”. “drop.levels” will remove empty factor levels in the target column. In the multilabel case the logical targets can be converted to factors with “multilabel.factor”. For survival, you may choose to recode the survival times to “left”, “right” or “interval2” censored times using “lcens”, “rcens” or “icens”, respectively. See survival::Surv for the format specification. Default for both binary classification and survival is “no” (do nothing).

Value

A factor for classification or a numeric for regression, a data.frame of logical columns for multilabel.

See Also

Other task: getTaskClassLevels(), getTaskCosts(), getTaskData(), getTaskDesc(), getTaskFeatureNames(), getTaskFormula(), getTaskId(), getTaskNFeats(), getTaskSize(), getTaskTargetNames(), getTaskType(), subsetTask()
Examples

```r
task = makeClassifTask(data = iris, target = "Species")
getTaskTargets(task)
```

getTaskType

*Get the type of the task.*

Description

See title.

Usage

```r
getTaskType(x)
```

Arguments

- `x` 
  
  *(Task | TaskDesc)*  
  
  Task or its description object.

Value

(character(1)).

See Also

Other task: `getTaskClassLevels()`, `getTaskCosts()`, `getTaskData()`, `getTaskDesc()`, `getTaskFeatureNames()`, `getTaskFormula()`, `getTaskId()`, `getTaskNFeats()`, `getTaskSize()`, `getTaskTargetNames()`, `getTaskTargets()`, `subsetTask()`

getTuneResult

*Returns the optimal hyperparameters and optimization path after training.*

Description

Returns the optimal hyperparameters and optimization path after training.

Usage

```r
getTuneResult(object)
```

Arguments

- `object` 
  
  *(WrappedModel)*  
  
  Trained Model created with `makeTuneWrapper`. 
**getTuneResultOptPath**

*Value*

(TuneResult).

*See Also*

Other tune: TuneControl, getNestedTuneResultsOptPathDf(), getNestedTuneResultsX(), getResamplingIndices(), makeModelMultiplexer(), makeModelMultiplexerParamSet(), makeTuneControlCMAES(), makeTuneControlDesign(), makeTuneControlGenSA(), makeTuneControlGrid(), makeTuneControlIrace(), makeTuneControlMBO(), makeTuneControlRandom(), makeTuneWrapper(), tuneParams(), tuneThreshold()

---

**getTuneResultOptPath**  
Get the optimization path of a tuning result.

---

**Description**

Returns the opt.path from a (TuneResult) object.

**Usage**

getTuneResultOptPath(tune.result, as.df = TRUE)

**Arguments**

- **tune.result**  
  (TuneResult)  
  A tuning result of the (tuneParams) function.

- **as.df**  
  (logical(1))  
  Should the optimization path be returned as a data frame? Default is TRUE.

**Value**

(ParamHelpers::OptPath) or (data.frame).

---

**gunpoint.task**  
Gunpoint functional data classification task.

---

**Description**

Contains the task (gunpoint.task). You have to classify whether a person raises up a gun or just an empty hand.

**References**

hasFunctionalFeatures  
*Check whether the object contains functional features.*

**Description**
See title.

**Usage**

```r
hasFunctionalFeatures(obj)
```

**Arguments**

- **obj** *(Task | TaskDesc | data.frame)*  
  Object to check.

**Value**

```r
(logical(1))
```

---

hasProperties  
*Deprecated, use hasLearnerProperties instead.*

**Description**

Deprecated, use hasLearnerProperties instead.

**Usage**

```r
hasProperties(learner, props)
```

**Arguments**

- **learner**  
  Deprecated.

- **props**  
  Deprecated.
helpLearner

Access help page of learner functions.

Description

Interactive function that gives the user quick access to the help pages associated with various functions involved in the given learner.

Usage

helpLearner(learner)

Arguments

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

See Also

Other learner: LearnerProperties, getClassWeightParam(), getHyperPars(), getLearnerId(), getLearnerNote(), getLearnerPackages(), getLearnerParVals(), getLearnerParamSet(), getLearnerPredictType(), getLearnerShortName(), getLearnerType(), getParamSet(), helpLearnerParam(), makeLearner(), makeLearners(), removeHyperPars(), setHyperPars(), setId(), setLearnerId(), setPredictThreshold(), setPredictType()

Other help: helpLearnerParam()

helpLearnerParam

Get specific help for a learner’s parameters.

Description

Print the description of parameters of a given learner. The description is automatically extracted from the help pages of the learner, so it may be incomplete.

Usage

helpLearnerParam(learner, param = NULL)

Arguments

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

param (character | NULL)
Parameter(s) to describe. Defaults to NULL, which prints information on the documentation status of all parameters.
### imputations

**Built-in imputation methods.**

#### Description

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 imputing constant values shifted below the minimum using `min(x) - multiplier * diff(range(x))`,
- `imputeMax(multiplier)` for imputing constant values shifted above the maximum using `max(x) + multiplier * diff(range(x))`,
- `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, features = NULL)` for imputations using the response of a classification or regression learner.

#### Usage

```r
imputeConstant(const)
```

```r
imputeMedian()
```

```r
imputeMean()
```

```r
imputeMode()
```

```r
imputeMin(multiplier = 1)
```

```r
imputeMax(multiplier = 1)
```

```r
imputeUniform(min = NA_real_, max = NA_real_)
```
imputeNormal(mu = NA_real_, sd = NA_real_)

imputeHist(breaks, use.mids = TRUE)

imputeLearner(learner, features = NULL)

Arguments

cost: 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 graphics::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 | character(1)
Supervised learner. Its predictions will be used for imputations. If you pass a string the learner will be created via makeLearner. 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()
**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**

```r
impute(
  obj,
  target = character(0L),
  classes = list(),
  cols = list(),
  dummy.classes = character(0L),
  dummy.cols = character(0L),
  dummy.type = "factor",
  force.dummies = FALSE,
  impute.new.levels = TRUE,
  recode.factor.levels = TRUE
)
```

**Arguments**

- **obj** *(data.frame | Task)*
  Input data.

- **target** *(character)*
  Name of the column(s) specifying the response. Default is `character(0)`.

- **classes** *(named list)*
  Named list containing imputation techniques for classes of columns. E.g. `list(numeric = imputeMedian())`.

- **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. Overrules imputation set via `classes`.
**Details**

The description object contains these slots

- target (character): See argument
- features (character): Feature names (column names of data)
- classes (character): Feature classes (storage type of data)
- 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**

(list)

- data (data.frame): Imputed data.
- desc (ImputationDesc): Description object.

**See Also**

Other impute: `imputations`, `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$data)
reimpute(data.frame(x = NA_real_), imputed$desc)

iris.task

Iris classification task.

Description

Contains the task (iris.task).

References

See datasets::iris.

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)).
**joinClassLevels**

Join some class existing levels to new, larger class levels for classification problems.

**Usage**

```r
joinClassLevels(task, new.levels)
```

**Arguments**

- **task** *(Task)*
  The task.

- **new.levels** *(list of character)*
  Element names specify the new class levels to create, while the corresponding element character vector specifies the existing class levels which will be joined to the new one.

**Value**

*Task.*

**Examples**

```r
joinClassLevels(iris.task, new.levels = list(foo = c("setosa", "virginica")))
```

---

**learnerArgsToControl**

Convert arguments to control structure.

**Description**

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

**Usage**

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

**Arguments**

- **control** *(function)*
  Function that creates control structure.

- **...** *(any)*
  Arguments for control structure function.
### LearnerProperties

*Query properties of learners.*

### Description

Properties can be accessed with `getLearnerProperties(learner)`, which returns a character vector.

The learner properties are defined as follows:

- **numerics, factors, ordered** Can numeric, factor or ordered factor features be handled?
- **functionals** Can an arbitrary number of functional features be handled?
- **single.functional** Can exactly one functional feature be handled?
- **missings** Can missing values in features be handled?
- **weights** Can observations be weighted during fitting?
- **oneclas, twoclass, multiclass** Only for classif: Can one-class, two-class or multi-class classification problems be handled?
- **class.weights** Only for classif: Can class weights be handled?
- **rcens, lcens, icens** Only for surv: Can right, left, or interval censored data be handled?
- **prob** For classif, cluster, multilabel, surv: Can probabilites be predicted?
- **se** Only for regr: Can standard errors be predicted?
- **oobpreds** Only for classif, regr and surv: Can out of bag predictions be extracted from the trained model?
- **featimp** For classif, regr, surv: Does the model support extracting information on feature importance?

### Usage

- `getLearnerProperties(learner)`
- `hasLearnerProperties(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 query.
**Value**

getLearnerProperties returns a character vector with learner properties. hasLearnerProperties returns a logical vector of the same length as props.

**See Also**

Other learner: getClassWeightParam(), getHyperPars(), getLearnerId(), getLearnerNote(), getLearnerPackages(), getLearnerParVals(), getLearnerParamSet(), getLearnerPredictType(), getLearnerShortName(), getLearnerType().getParamSet().helpLearner().helpLearnerParam().makeLearner().makeLearners().removeHyperPars().setHyperPars().setId().setLearnerId().setPredictThreshold().setPredictType()

---

**learners**  
List of supported learning algorithms.

**Description**

All supported learners can be found by listLearners or as a table in the tutorial appendix: https://mlr.mlr-org.com/articles/tutorial/integrated_learners.html.

**listFilterEnsembleMethods**  
List ensemble filter methods.

**Description**

Returns a subset-able dataframe with filter information.

**Usage**

listFilterEnsembleMethods(desc = TRUE)

**Arguments**

- desc (logical(1))  
  Provide more detailed information about filters. Default is TRUE.

**Value**

(data.frame).

**See Also**

Other filter: filterFeatures(), generateFilterValuesData(), getFilteredFeatures(), listFilterMethods(), makeFilter(), makeFilterEnsemble(), makeFilterWrapper(), plotFilterValues()
listFilterMethods  List filter methods.

Description

Returns a subset-able dataframe with filter information.

Usage

```r
listFilterMethods(
  desc = TRUE,
  tasks = FALSE,
  features = FALSE,
  include.deprecated = FALSE
)
```

Arguments

- `desc` (logical(1))
  Provide more detailed information about filters. Default is TRUE.
- `tasks` (logical(1))
  Provide information on supported tasks. Default is FALSE.
- `features` (logical(1))
  Provide information on supported features. Default is FALSE.
- `include.deprecated` (logical(1))
  Should deprecated filter methods be included in the list. Default is FALSE.

Value

(data.frame).

See Also

Other filter: `filterFeatures()`, `generateFilterValuesData()`, `getFilteredFeatures()`, `listFilterEnsembleMethods()`, `makeFilter()`, `makeFilterEnsemble()`, `makeFilterWrapper()`, `plotFilterValues()`
listLearnerProperties

List the supported learner properties

Description

This is useful for determining which learner properties are available.

Usage

listLearnerProperties(type = "any")

Arguments

type

(character(1))

Only return properties for a specified task type. Default is “any”.

Value

(character).

listLearners

Find matching learning algorithms.

Description

Returns learning algorithms which have specific characteristics, e.g. whether they support missing values, case weights, etc.

Note that the packages of all learners are loaded during the search if you create them. This can be a lot. If you do not create them we only inspect properties of the S3 classes. This will be a lot faster.

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. The same applies for many multilabel methods, see, e.g., makeMultilabelBinaryRelevanceWrapper.

Usage

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

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

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

### Arguments

**obj**
(character(1) | Task)
Either character(1) task or the type of the task, in the latter case one of: “classif” “regr” “surv” “costsens” “cluster” “multilabel”. 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.

**check.packages**
(logical(1))
Check if required packages are installed. Calls find.package(). If create is TRUE, this is done implicitly and the value of this parameter is ignored. If
create is FALSE and check.packages is TRUE the returned table only contains learners whose dependencies are installed. If check.packages set to FALSE, learners that cannot actually be constructed because of missing packages may be returned. Default is FALSE.

create (logical(1))
Instantiate objects (or return info table)? Packages are loaded if and only if this option is TRUE. Default is FALSE.

Value

([data.frame | list] of Learner). Either a descriptive data.frame that allows access to all properties of the learners or a list of created learner objects (named by ids of listed learners).

Examples

## Not run:
listLearners("classif", properties = c("multiclass", "prob"))
data = iris
task = makeClassifTask(data = data, target = "Species")
listLearners(task)
## End(Not run)
listMeasures  

Find matching measures.

**Description**

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

**Usage**

```r
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 character(1) task or the type of the task, in the latter case one of: “classif” “regr” “surv” “costsens” “cluster” “multilabel”. 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.

---

listTaskTypes  

List the supported task types in mlr

**Description**

Returns a character vector with each of the supported task types in mlr.
l lung.task

Usage

taskTypes()

Value

(character).

---

lung.task  NCCTG Lung Cancer survival task.

Description

Contains the task (lung.task).

References

See survival::lung. Incomplete cases have been removed from the task.

---

makeAggregation  Specify your own aggregation of measures.

Description

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

Usage

makeAggregation(id, name = id, properties, fun)

Arguments

id  (character(1))
Name of the aggregation method (preferably the same name as the generated function).

name  (character(1))
Long name of the aggregation method. Default is id.

properties  (character)
Set of aggregation properties.

req.train  Are prediction or train sets required to calculate the aggregation?

req.test  Are prediction or test sets required to calculate the aggregation?

fun  (function(task, perf.test, perf.train, measure, group, pred))
Calculates the aggregated performance. In most cases you will only need the performances perf.test and optionally perf.train on the test and training data sets.
**makeBaggingWrapper**

Fuses a learner with the bagging technique. Creates a learner object, which can be used like any other learner object. Models can easily be accessed via `getLearnerModel`.

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.

**Value**

(Aggregation).

**See Also**

aggregations, `setAggregation`  

**Examples**

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

---

**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 `getLearnerModel`.

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

```r
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. At least one feature will always be selected.

Value

  Learner.

See Also

Other wrapper: `makeClassificationViaRegressionWrapper()`, `makeConstantClassWrapper()`, `makeCostSensClassifWrapper()`, `makeCostSensRegrWrapper()`, `makeDownsampleWrapper()`, `makeDummyFeaturesWrapper()`, `makeExtractFDAFeatsWrapper()`, `makeFeatSelWrapper()`, `makeFilterWrapper()`, `makeImputeWrapper()`, `makeMulticlassWrapper()`, `makeMultilabelBinaryRelevanceWrapper()`, `makeMultilabelClassifierChainsWrapper()`, `makeMultilabelDBRWrapper()`, `makeMultilabelNestedStackingWrapper()`, `makeMultilabelStackingWrapper()`, `makeOverBaggingWrapper()`, `makePreprocWrapper()`, `makePreprocWrapperCare()`

```r
makeRemoveConstantFeaturesWrapper(), makeSMOTEWrapper(), makeTuneWrapper(), makeUndersampleWrapper(), makeWeightedClassesWrapper()
```
**Description**

Builds regression models that predict for the positive class whether a particular example belongs to it (1) or not (-1).

Probabilities are generated by transforming the predictions with a softmax.

Inspired by WEKA's ClassificationViaRegression (http://weka.sourceforge.net/doc.dev/weka/classifiers/meta/ClassificationViaRegression.html).

**Usage**

```r
makeClassificationViaRegressionWrapper(learner, predict.type = "response")
```

**Arguments**

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

- **predict.type** 
  (character(1))
  “response” (= labels) or “prob” (= probabilities and labels by selecting the one with maximal probability).

**Value**

Learner.

**See Also**


**Examples**

```r
lrn = makeLearner("regr.rpart")
lrn = makeClassificationViaRegressionWrapper(lrn)
mod = train(lrn, sonar.task, subset = 1:140)
predictions = predict(mod, newdata = getTaskData(sonar.task)[141:208, 1:60])
```
makeClassifTask

Create a classification task.

Description

Create a classification task.

Usage

makeClassifTask(
  id = deparse(substitute(data)),
  data,
  target,
  weights = NULL,
  blocking = NULL,
  coordinates = NULL,
  positive = NA_character_,
  fixup.data = "warn",
  check.data = TRUE
)

Arguments

id (character(1))
Id string for object. Default is the name of the R variable passed to data.
data (data.frame)
A data frame containing the features and target variable(s).
target (character(1) | character(2) | character(n.classes))
Name(s) of the target variable(s). For survival analysis these are the names of
the survival time and event columns, so it has length 2. For multilabel classification
it contains the names of the logical columns that encode whether a label
is present or not and its length corresponds to the number of classes.
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.
coordinates (data.frame)
Coordinates of a spatial data set that will be used for spatial partitioning of the
data in a spatial cross-validation resampling setting. Coordinates have to be
numeric values. Provided data.frame needs to have the same number of rows as
data and consist of at least two dimensions.
makeClusterTask

Description

Create a cluster task.

Usage

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

Arguments

id            (character(1))
Id string for object. Default is the name of the R variable passed to data.

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

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.

positive      (character(1))
Positive class for binary classification (otherwise ignored and set to NA). 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 choices 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.

See Also

Task CostSensTask ClusterTask MultilabelTask RegrTask SurvTask

makeClusterTask     Create a cluster task.
makeConstantClassWrapper

Wraps a classification learner to support problems where the class label is (almost) constant.

Description

If the training data contains only a single class (or almost only a single class), this wrapper creates a model that always predicts the constant class in the training data. In all other cases, the underlying learner is trained and the resulting model used for predictions.

Probabilities can be predicted and will be 1 or 0 depending on whether the label matches the majority class or not.

Usage

makeConstantClassWrapper(learner, frac = 0)

Arguments

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

frac numeric(1)
The fraction of labels in [0, 1) that can be different from the majority label. Default is 0, which means that constant labels are only predicted if there is exactly one label in the data.
makeCostMeasure

Value

Learner.

See Also

Other wrapper: makeBaggingWrapper(), makeClassificationViaRegressionWrapper(), makeCostSensClassificationWrapper(), makeCostSensRegressionWrapper(), makeDownsampleWrapper(), makeDummyFeaturesWrapper(), makeExtractFDAFeatsWrapper(), makeFeatSelWrapper(), makeFilterWrapper(), makeImputeWrapper(), makeMulticlassWrapper(), makeMultilabelBinaryRelevanceWrapper(), makeMultilabelClassifierChainsWrapper(), makeMultilabelDBRWrapper(), makeMultilabelNestedStackingWrapper(), makeMultilabelStackingWrapper(), makeOverBaggingWrapper(), makePreprocWrapper(), makePreprocWrapperCaret(), makeRemoveConstantFeaturesWrapper(), makeSMOTEWrapper(), makeTuneWrapper(), makeUndersampleWrapper(), makeWeightedClassesWrapper()

makeCostMeasure Creates a measure for non-standard misclassification costs.

Description

Creates a cost measure for non-standard classification error costs.

Usage

makeCostMeasure(
  id = "costs",
  minimize = TRUE,
  costs,
  combine = mean,
  best = NULL,
  worst = NULL,
  name = id,
  note = ""
)

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.
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`.

**name**
- character

Name of the measure. Default is `id`.

**note**
- character

Description and additional notes for the measure. Default is `""`.

**Value**

Measure.

**See Also**

Other performance: `ConfusionMatrix`, `calculateConfusionMatrix()`, `calculateROCMeasures()`, `estimateRelativeOverfitting()`, `makeCustomResampledMeasure()`, `makeMeasure()`, `measures`, `performance()`, `setAggregation()`, `setMeasurePars()`
Value

Learner.

See Also

Other costsens: `makeCostSensClassifWrapper()`, `makeCostSensTask()`, `makeCostSensWeightedPairsWrapper()`  
Other wrapper: `makeBaggingWrapper()`, `makeClassificationViaRegressionWrapper()`, `makeConstantClassWrapper()`  
`makeCostSensRegrWrapper()`, `makeDownsampleWrapper()`, `makeDummyFeaturesWrapper()`, `makeExtractFDAFeatsWrapper()`, `makeFeatSelWrapper()`, `makeFilterWrapper()`, `makeImputeWrapper()`, `makeMulticlassWrapper()`, `makeMultilabelBinaryRelevanceWrapper()`, `makeMultilabelClassifierChainsWrapper()`, `makeMultilabelDBRWrapper()`, `makeMultilabelNestedStackingWrapper()`, `makeMultilabelStackingWrapper()`, `makeOverBaggingWrapper()`, `makePreprocWrapper()`, `makePreprocWrapperCaret()`, `makeRemoveConstantFeaturesWrapper()`, `makeSMOTEWrapper()`, `makeTuneWrapper()`, `makeUndersampleWrapper()`, `makeWeightedClassesWrapper()`
**makeCostSensTask**

Create a cost-sensitive classification task.

Description

Create a cost-sensitive classification task.

Usage

```r
makeCostSensTask(
  id = deparse(substitute(data)),
  data,
  costs,
  blocking = NULL,
  coordinates = NULL,
  fixup.data = "warn",
  check.data = TRUE
)
```

Arguments

- **id** (character(1))
  Id string for object. Default is the name of the R variable passed to `data`.

- **data** (data.frame)
  A data frame containing the features and target variable(s).

- **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`.

- **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.

- **coordinates** (data.frame)
  Coordinates of a spatial data set that will be used for spatial partitioning of the data in a spatial cross-validation resampling setting. Coordinates have to be numeric values. Provided `data.frame` needs to have the same number of rows as `data` and consist of at least two dimensions.
fixup.data  (character(1))
Should some basic cleaning up of data be performed? Currently this means removing empty factor levels for the columns. Possible choices 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.

See Also

Task ClassifTask ClusterTask MultilabelTask RegrTask SurvTask
Other costsens: makeCostSensClassifWrapper(), makeCostSensRegrWrapper(), 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 getLearnerModel.

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).
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(
  measure.id,
  aggregation.id,
  minimize = TRUE,
  properties = character(0L),
  fun,
  extra.args = list(),
  best = NULL,
  worst = NULL,
  measure.name = measure.id,
  aggregation.name = aggregation.id,
  note = ""
)

Arguments

measure.id (character(1))
Short name of measure.

aggregation.id (character(1))
Short name of aggregation.

minimize (logical(1))
Should the measure be minimized? Default is TRUE.

properties (character)
Set of measure properties. For a list of values see Measure. Default is character(0).

References


See Also

Other cost sens: makeCostSensClassifWrapper(), makeCostSensRegrWrapper(), makeCostSensTask()
fun (function(task, group, pred, extra.args))
Calculates performance value from ResamplePrediction object. For rare cases you can also use the task, the grouping or the extra arguments extra.args. - task (Task)
The task. - group (factor)
Grouping of resampling iterations. This encodes whether specific iterations 'belong together' (e.g. repeated CV). - pred (Prediction)
Prediction object. - extra.args (list)
See below.
exTRA.args (list)
List of extra arguments which will always be passed to fun. Default is empty list.
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.
measure.name (character(1))
Long name of measure. Default is measure.id.
aggregation.name (character(1))
Long name of the aggregation. Default is aggregation.id.
note (character)
Description and additional notes for the measure. Default is "":

Value
Measure.

See Also
Other performance: ConfusionMatrix, calculateConfusionMatrix(), calculateROCMasures(), estimateRelativeOverfitting(), makeCostMeasure(), makeMeasure(), measures, performance(), setAggregation(), setMeasurePars()

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()
Other wrapper: makeBaggingWrapper(), makeClassificationViaRegressionWrapper(), makeConstantClassWrapper()
               makeCostSensClassifWrapper(), makeCostSensRegrWrapper(), makeDummyFeaturesWrapper(),
               makeExtractFDAFeatsWrapper(), makeFeatSelWrapper(), makeImputeWrapper(),
               makeMulticlassWrapper(), makeMultilabelBinaryRelevanceWrapper(), makeMultilabelClassifierChainsWrapper()
               makeMultilabelDBRWrapper(), makeMultilabelNestedStackingWrapper(), makeMultilabelStackingWrapper(),
               makeOverBaggingWrapper(), makePreprocWrapper(), makePreprocWrapperCaret(), makeRemoveConstantFeaturesWrapper()
               makeSMOTEWrapper(), makeTuneWrapper(), makeUndersampleWrapper(), makeWeightedClassesWrapper()

Description

Fuses a base learner with the dummy feature creator (see createDummyFeatures). Returns a learner which can be used like any other learner.

Usage

makeDummyFeaturesWrapper(learner, method = "1-of-n", cols = NULL)

Arguments

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

Constructor for FDA feature extraction methods.

Description

This can be used to implement custom FDA feature extraction. Takes a learn and a reextract function along with some optional parameters to those as argument.

Usage

makeExtractFDAFeatMethod(learn, reextract, args = list(), par.set = NULL)

Arguments

learn (function(data, target, col, ...))
Function to learn and extract information on functional column col. Arguments are:

• data data.frame
  Data frame containing matrices with one row per observation of a single functional or time series and one column per measure time point. All entries need to be numeric.
• target (character(1))
  Name of the target variable. Default: “NULL”. The variable is only set to be consistent with the API.
• col (character(1) | numeric(1))
  Column names or indices, the extraction should be performed on. The function has to return a named list of values.

Value

Learner.

See Also

Other wrapper: makeBaggingWrapper(), makeClassificationViaRegressionWrapper(), makeConstantClassWrapper(), makeCostSensClassifWrapper(), makeCostSensRegrWrapper(), makeDownsampleWrapper(), makeExtractFDAFeatsWrapper(), makeFeatSelWrapper(), makeFilterWrapper(), makeImputeWrapper(), makeMulticlassWrapper(), makeMultilabelBinaryRelevanceWrapper(), makeMultilabelClassifierChainsWrapper(), makeMultilabelDBRWrapper(), makeMultilabelNestedStackingWrapper(), makeMultilabelStackingWrapper(), makeOverBaggingWrapper(), makePreprocWrapper(), makePreprocWrapperCaret(), makeRemoveConstantFeaturesWrapper(), makeSMOTEWrapper(), makeTuneWrapper(), makeUndersampleWrapper(), makeWeightedClassesWrapper()
**makeExtractFDAFeatsWrapper**

Fuse learner with an extractFDAFeatures method.

**Description**

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

**Usage**

```r
makeExtractFDAFeatsWrapper(learner, feat.methods = list())
```

**Arguments**

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

- **feat.methods** *(named list)*
  List of functional features along with the desired methods for each functional feature. “all” applies the `extractFDAFeatures` method to each functional feature. Names of `feat.methods` must match column names of functional features. Available feature extraction methods are available under family `fda_featextractor`. Specifying a functional feature multiple times with different extraction methods allows for the extraction of different features from the same functional. Default is `list()` which does nothing.

**Value**

Learner.

**See Also**

Other fda: `extractFDAFeatures()`, `makeExtractFDAFeatsWrapper()`
**makeFeatSelWrapper**

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**

```r
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 behavior, look at `FeatSelControl`.

---

**See Also**

Other fda: `extractFDAFeatures()`, `makeExtractFDAFeatMethod()`

Other wrapper: `makeBaggingWrapper()`, `makeClassificationViaRegressionWrapper()`, `makeConstantClassWrapper()`, `makeCostSensClassifWrapper()`, `makeCostSensRegrWrapper()`, `makeDownsampleWrapper()`, `makeDummyFeaturesWrapper()`, `makeFeatSelWrapper()`, `makeFilterWrapper()`, `makeImputeWrapper()`, `makeMulticlassWrapper()`, `makeMultilabelBinaryRelevanceWrapper()`, `makeMultilabelClassifierChainsWrapper()`, `makeMultilabelDBRWrapper()`, `makeMultilabelNestedStackingWrapper()`, `makeMultilabelStackingWrapper()`, `makeOverBaggingWrapper()`, `makePreprocWrapper()`, `makePreprocWrapperCaret()`, `makeRemoveConstantFeaturesWrapper()`, `makeSMOTEWrapper()`, `makeTuneWrapper()`, `makeUndersampleWrapper()`, `makeWeightedClassesWrapper()`
makeFeatSelWrapper

measures
(list of Measure | Measure)
Performance measures to evaluate. The first measure, aggregated by the first
aggregation function is optimized, others are simply evaluated. Default is the
default measure for the task, see here getDefaultMeasure.

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. Has to be used
together with bits.to.features.

bits.to.features
(function(x, task))
Function which transforms an integer-0-1 vector into a character vector of se-
lected features. Per default a value of 1 in the ith bit selects the ith feature to be
in the candidate solution. The vector x will correspond to the bit.names and
has to be of the same length.

control
[see FeatSelControl] Control object for search method. Also selects the opti-
mization 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, analyzeFeatSelResult(), getFeatSelResult(), selectFeatures()

Other wrapper: makeBaggingWrapper(), makeClassificationViaRegressionWrapper(), makeConstantClassWrapper()
makeCostSensClassifWrapper(), makeCostSensRegrWrapper(), makeDownsampleWrapper(),
makeDummyFeaturesWrapper(), makeExtractFDAFeatsWrapper(), makeFilterWrapper(), makeImputeWrapper(),
makeMulticlassWrapper(), makeMultilabelBinaryRelevanceWrapper(), makeMultilabelClassifierChainsWrapper(),
makeMultilabelDBRWrapper(), makeMultilabelNestedStackingWrapper(), makeMultilabelStackingWrapper(),
makeOverBaggingWrapper(), makePreprocWrapper(), makePreprocWrapperCaret(), makeRemoveConstantFeaturesWrapper(),
makeSMOTEWrapper(), makeTuneWrapper(), makeUndersampleWrapper(), makeWeightedClassesWrapper()

Examples
# nested resampling with feature selection (with a nonsense algorithm for selection)
outer = makeResampleDesc("CV", iters = 2L)
inner = makeResampleDesc("Holdout")
ctrl = makeFeatSelControlRandom(maxit = 1)
lnr = makeFeatSelWrapper("classif.ksvm", resampling = inner, control = ctrl)
# we also extract the selected features for all iteration here
r = resample(lnr, iris.task, outer, extract = getFeatSelResult)
Create a feature filter.

Description

Creates and registers custom feature filters. Implemented filters can be listed with `listFilterMethods`. Additional documentation for the `fun` parameter specific to each filter can be found in the description.

Usage

```r
makeFilter(name, desc, pkg, supported.tasks, supported.features, fun)
```

Arguments

- `name` (character(1))
  - Identifier for the filter.
- `desc` (character(1))
  - Short description of the filter.
- `pkg` (character(1))
  - Source package where the filter is implemented.
- `supported.tasks` (character)
  - Task types supported.
- `supported.features` (character)
  - Feature types supported.
- `fun` (function(task, nselect, ...)
  - Function which takes a task and returns a named numeric vector of scores, one score for each feature of task. Higher scores mean higher importance of the feature. At least `nselect` features must be calculated, the remaining may be set to `NA` or omitted, and thus will not be selected. The original order will be restored if necessary.

Value

Object of class “Filter”.

References


**makeFilterEnsemble**  Create an ensemble feature filter.

**Description**

Creates and registers custom ensemble feature filters. Implemented ensemble filters can be listed with `listFilterEnsembleMethods`. Additional documentation for the `fun` parameter specific to each filter can be found in the description.

**Usage**

```r
call: makeFilterEnsemble(name, base.methods, desc, fun)
```

**Arguments**

- `name` (character(1))
  Identifier for the filter.

- `base.methods` (character(1))
  the base filter methods which the ensemble method will use.

- `desc` (character(1))
  Short description of the filter.

- `fun` (function(task, nselect, ...)
  Function which takes a task and returns a named numeric vector of scores, one score for each feature of `task`. Higher scores mean higher importance of the feature. At least `nselect` features must be calculated, the remaining may be set to NA or omitted, and thus will not be selected. the original order will be restored if necessary.

**Value**

Object of class “FilterEnsemble”.

**See Also**

Other filter: `filterFeatures()`, `generateFilterValuesData()`, `getFilteredFeatures()`, `listFilterEnsembleMethods()`, `listFilterMethods()`,
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.

**Usage**

```r
makeFilterWrapper(
  learner,
  fw.method = "FSelectorRcpp_information.gain",
  fw.base.methods = NULL,
  fw.perc = NULL,
  fw.abs = NULL,
  fw.threshold = NULL,
  fw.fun = NULL,
  fw.fun.args = NULL,
  fw.mandatory.feat = NULL,
  cache = FALSE,
  ...
)
```

**Arguments**

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

- **fw.method**  
  `(character(1))`
  Filter method. See `listFilterMethods`. Default is “FSelectorRcpp_information.gain”.

- **fw.base.methods**  
  `(character(1))`
  Simple Filter methods for ensemble filters. See `listFilterMethods`. Can only be used in combination with ensemble filters. See `listFilterEnsembleMethods`.

- **fw.perc**  
  `(numeric(1))`
  If set, select fw.perc*100 top scoring features. Mutually exclusive with arguments `fw.abs`, `fw.threshold` and `fw.fun`.

- **fw.abs**  
  `(numeric(1))`
  If set, select fw.abs top scoring features. Mutually exclusive with arguments `fw.perc`, `fw.threshold` and `fw.fun`.

- **fw.threshold**  
  `(numeric(1))`
  If set, select features whose score exceeds fw.threshold. Mutually exclusive with arguments `fw.perc`, `fw.abs` and `fw.fun`.

- **fw.fun**  
  `(function)`
  If set, select features via a custom thresholding function, which must return the
makeFilterWrapper

number of top scoring features to select. Mutually exclusive with arguments
fw.perc, fw.abs and fw.threshold.

fw.fun.args  (any)
Arguments passed to the custom thresholding function

fw.mandatory.feat  (character)
Mandatory features which are always included regardless of their scores

cache  (character(1) | logical)
Whether to use caching during filter value creation. See details.

...  (any)
Additional parameters passed down to the filter. If you are using more than one
filter method, you need to pass the arguments in a named list via more.args. For
example more.args = list("FSelectorRcpp_information.gain" = list(equal
= TRUE)).

Details

If ensemble = TRUE, ensemble feature selection using all methods specified in fw.method is per-
formed. At least two methods need to be selected.

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.

Value

Learner.

Caching

If cache = TRUE, the default mlr cache directory is used to cache filter values. The directory is
operating system dependent and can be checked with getCacheDir(). Alternatively a custom
directory can be passed to store the cache. The cache can be cleared with deleteCacheDir().
Caching is disabled by default. Care should be taken when operating on large clusters due to
possible write conflicts to disk if multiple workers try to write the same cache at the same time.

See Also

Other filter: filterFeatures(), generateFilterValuesData(), getFilteredFeatures(), listFilterEnsembleMethods(),
listFilterMethods(), makeFilter(), makeFilterEnsemble(), plotFilterValues()

Other wrapper: makeBaggingWrapper(), makeClassificationViaRegressionWrapper(), makeConstantClassWrapper(),
makeCostSensClassifierWrapper(), makeCostSensRegrWrapper(), makeDownsampleWrapper(),
makeDummyFeaturesWrapper(), makeExtractFDAOFeaturesWrapper(), makeFeatSelWrapper(), makeImputeWrapper(),
makeMulticlassWrapper(), makeMultilabelBinaryRelevanceWrapper(), makeMultilabelClassifierChainsWrapper(),
makeMultilabelDBRWrapper(), makeMultilabelNestedStackingWrapper(), makeMultilabelStackingWrapper(),
makeOverBaggingWrapper(), makePreprocWrapper(), makePreprocWrapperCaret(), makeRemoveConstantFeaturesWrapper(),
makeSMOTEWrapper(), makeTuneWrapper(), makeUndersampleWrapper(), makeWeightedClassesWrapper()
Examples

```r
# Examples of using the makeFilterWrapper function

# First example using the function

task = makeClassifTask(data = iris, target = "Species")
lrn = makeLearner("classif.lda")
inner = makeResampleDesc("Holdout")
outer = makeResampleDesc("CV", iters = 2)
lrn = makeFilterWrapper(lrn, fw.perc = 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)

# usage of an ensemble filter

lrn = makeLearner("classif.lda")
lrn = makeFilterWrapper(lrn, fw.method = "E-Borda",
  fw.base.methods = c("FSelectorRcpp_gain.ratio", "FSelectorRcpp_information.gain"),
  fw.perc = 0.5)
r = resample(lrn, task, outer, extract = function(model) {
  getFilteredFeatures(model)
})
print(r$extract)

# usage of a custom thresholding function

biggest_gap = function(values, diff) {
  gap_size = 0
  gap_location = 0
  for (i in (diff + 1):length(values)) {
    gap = values[[i - diff]] - values[[i]]
    if (gap > gap_size) {
      gap_size = gap
      gap_location = i - 1
    }
  }
  return(gap_location)
}

lrn = makeLearner("classif.lda")
lrn = makeFilterWrapper(lrn, fw.method = "FSelectorRcpp_information.gain",
  fw.fun = biggest_gap, fw.fun.args = list("diff" = 1))
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. The function needs to know the largest possible index of the whole data set.

Value

(ResampleInstance).

makeFunctionalData

Create a data.frame containing functional features from a normal data.frame.

Description

To work with functional features, those features need to be stored as a matrix column in the data.frame, so mlr can automatically recognize them as functional features. This function allows for an easy conversion from a data.frame with numeric columns to the required format. If the data already contains matrix columns, they are left as-is if not specified otherwise in fd.features. See Examples for the structure of the generated output.

Usage

makeFunctionalData(data, fd.features = NULL, exclude.cols = NULL)
Arguments

- **data** (*data.frame*)
  A data.frame that contains the functional features as numeric columns.

- **fd.features** (*list*)
  Named list containing integer column indices or character column names. Each element defines a functional feature, in the given order of the indices or column names. The name of the list element defines the name of the functional feature. All selected columns have to correspond to numeric data.frame entries. The default is NULL, which means all numeric features are considered to be a single functional “fd1”.

- **exclude.cols** (*character | integer*)
  Column names or indices to exclude from conversion to functionals, even if they are in included in fd.features. Default is not to exclude anything.

Value

- (*data.frame*).

Examples

```r
# data.frame where columns 1:6 and 8:10 belong to a functional feature
d1 = data.frame(matrix(rnorm(100), nrow = 10), "target" = seq_len(10))
# Transform to functional data
d2 = makeFunctionalData(d1, fd.features = list("fd1" = 1:6, "fd2" = 8:10))
# Create a regression task
makeRegrTask(data = d2, target = "target")
```

makeImputeMethod  
*Create a custom imputation method.*

Description

This is a constructor to create your own imputation methods.

Usage

```r
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 of learn are passed to this function into ...
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

```r
makeImputeWrapper(
  learner,
  classes = list(),
  cols = list(),
  dummy.classes = character(0L),
  dummy.cols = character(0L),
  dummy.type = "factor",
  force.dummies = FALSE,
  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 = imputeMedian())`.

- **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. Overrules imputation set via classes.

- **dummy.classes** *(character)*
  Classes of columns to create dummy columns for. Default is character(0).

- **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”.

force.dummies (logical(1))
Force dummy creation even if the respective data column does not contain any NAs. Note that (a) most learners will complain about constant columns created this way but (b) your feature set might be stochastic if you turn this off. Default is FALSE.

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, impute(), makeImputeMethod(), reimpute()

Other wrapper: makeBaggingWrapper(), makeClassificationViaRegressionWrapper(), makeConstantClassWrapper(), makeCostSensClassifierWrapper(), makeCostSensRegrWrapper(), makeDownsampleWrapper(), makeDummyFeaturesWrapper(), makeExtractFDAFeatsWrapper(), makeFeatSelWrapper(), makeFilterWrapper(), makeMulticlassWrapper(), makeMultilabelBinaryRelevanceWrapper(), makeMultilabelClassifierChainsWrapper(), makeMultilabelDBRWrapper(), makeMultilabelNestedStackingWrapper(), makeMultilabelStackingWrapper(), makeOverBaggingWrapper(), makePreprocWrapper(), makePreprocWrapperCaret(), makeRemoveConstantFeaturesWrapper(), makeSMOTEWrapper(), makeTuneWrapper(), makeUndersampleWrapper(), makeWeightedClassesWrapper()
Usage

makeLearner(
  cl,
  id = cl,
  predict.type = "response",
  predict.threshold = NULL,
  fix.factors.prediction = FALSE,
  ...,
  par.vals = list(),
  config = list()
)

Arguments

cl (character(1))
Class of learner. By convention, all classification learners start with “classif.”
all regression learners with “regr.” all survival learners start with “surv.” all
clustering learners with “cluster.” and all multilabel classification learners start
with “multilabel.”. A list of all integrated 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). Clus-
tering: “response” (= cluster IDS) or “prob” (= fuzzy cluster membership prob-
abilities), Multilabel: “response” (= logical matrix indicating the predicted class
labels) or “prob” (= probabilities and corresponding logical matrix indicating
class labels). Default is “response”.

predict.threshold (numeric)
Threshold to produce class labels. Has to be a named vector, where names corre-
spond to class labels. Only for binary classification it can be a single numerical
threshold for the positive class. See setThreshold for details on how it is applied.
Default is NULL which means 0.5 / an equal threshold for each class.

fix.factors.prediction (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. If you want to set specific hyperparameters for a learner during model creation, these should go here. You can get a list of available hyperparameters using \texttt{getParamSet(<learner>)}. Alternatively hyperparameters can be given using the \texttt{par.vals} argument but ... should be preferred!

\textbf{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 ... for passing hyperparameters.

\textbf{config} (named list) Named list of config option to overwrite global settings set via \texttt{configureMlr} for this specific learner.

\textbf{Value} (Learner).

\textbf{par.vals vs. ...}

The former aims at specifying default hyperparameter settings from \texttt{mlr} which differ from the actual defaults in the underlying learner. For example, \texttt{respect.unordered.factors} is set to \texttt{order} in \texttt{mlr} while the default in \texttt{ranger::ranger} depends on the argument \texttt{splitrule}. \texttt{getHyperPars(<learner>)} can be used to query hyperparameter defaults that differ from the underlying learner. This function also shows all hyperparameters set by the user during learner creation (if these differ from the learner defaults).

\textbf{regr.randomForest}

For this learner we added additional uncertainty estimation functionality (\texttt{predict.type = "se"}) for the randomForest, which is not provided by the underlying package.

Currently implemented methods are:

- If \texttt{se.method = "jackknife"} the standard error of a prediction is estimated by computing the jackknife-after-bootstrap, the mean-squared difference between the prediction made by only using trees which did not contain said observation and the ensemble prediction.

- If \texttt{se.method = "bootstrap"} the standard error of a prediction is estimated by bootstrapping the random forest, where the number of bootstrap replicates and the number of trees in the ensemble are controlled by \texttt{se.boot} and \texttt{se.ntree} respectively, and then taking the standard deviation of the bootstrap predictions. The "brute force" bootstrap is executed when \texttt{ntree = se.ntree}, the latter of which controls the number of trees in the individual random forests which are bootstrapped. The "noisy bootstrap" is executed when \texttt{se.ntree < ntree} which is less computationally expensive. A Monte-Carlo bias correction may make the latter option preferable in many cases. Defaults are \texttt{se.boot = 50} and \texttt{se.ntree = 100}.

- If \texttt{se.method = "sd"}, the default, the standard deviation of the predictions across trees is returned as the variance estimate. This can be computed quickly but is also a very naive estimator.
For both “jackknife” and “bootstrap”, a Monte-Carlo bias correction is applied and, in the case that this results in a negative variance estimate, the values are truncated at 0.

Note that when using the “jackknife” procedure for se estimation, using a small number of trees can lead to training data observations that are never out-of-bag. The current implementation ignores these observations, but in the original definition, the resulting se estimation would be undefined.

Please note that all of the mentioned se.method variants do not affect the computation of the posterior mean “response” value. This is always the same as from the underlying randomForest.

regr.featureless

A very basic baseline method which is useful for model comparisons (if you don’t beat this, you very likely have a problem). Does not consider any features of the task and only uses the target feature of the training data to make predictions. Using observation weights is currently not supported.

Methods “mean” and “median” always predict a constant value for each new observation which corresponds to the observed mean or median of the target feature in training data, respectively.

The default method is “mean” which corresponds to the ZeroR algorithm from WEKA.

classif.featureless

Method “majority” predicts always the majority class for each new observation. In the case of ties, one randomly sampled, constant class is predicted for all observations in the test set. This method is used as the default. It is very similar to the ZeroR classifier from WEKA. The only difference is that ZeroR always predicts the first class of the tied class values instead of sampling them randomly.

Method “sample-prior” always samples a random class for each individual test observation according to the prior probabilities observed in the training data.

If you opt to predict probabilities, the class probabilities always correspond to the prior probabilities observed in the training data.

See Also

Other learner: LearnerProperties, getClassWeightParam(), getHyperPars(), getLearnerId(), getLearnerNote(), getLearnerPackages(), getLearnerParVals(), getLearnerParamSet(), getLearnerPredictType(), getLearnerShortName(), getLearnerType(),getParamSet(), helpLearner(), helpLearnerParam(), makeLearners(), removeHyperPars(), setHyperPars(), setId(), setLearnerId(), setPredictThreshold(), setPredictType()

Examples

makeLearner("classif.rpart")
makeLearner("classif.lda", predict.type = "prob")
lrn = makeLearner("classif.lda", method = "t", nu = 10)
getHyperPars(lrn)
makeLearners

Create multiple learners at once.

Description
Small helper function that can save some typing when creating multiple learner objects. Calls makeLearner multiple times internally.

Usage
makeLearners(cls, ids = NULL, type = NULL, ...)

Arguments
\begin{itemize}
\item \textbf{cls} \hspace{1cm} \texttt{(character)}
Classes of learners.
\item \textbf{ids} \hspace{1cm} \texttt{(character)}
Id strings. Must be unique. Default is cls.
\item \textbf{type} \hspace{1cm} \texttt{(character(1))}
Shortcut to prepend type string to cls so one can set cls = "rpart". Default is NULL, i.e., this is not used.
\item \textbf{...} \hspace{1cm} \texttt{(any)}
Optional named (hyper)parameters. If you want to set specific hyperparameters for a learner during model creation, these should go here. You can get a list of available hyperparameters using \emph{getParamSet(<learner>)}. Alternatively hyperparameters can be given using the \emph{par.vals} argument but \textbf{...} should be preferred!
\end{itemize}

Value

(named list of \texttt{Learner}). Named by ids.

See Also
Other learner: \texttt{LearnerProperties}, \texttt{getClassWeightParam()}, \texttt{getHyperPars()}, \texttt{getLearnerId()}, \texttt{getLearnerNote()}, \texttt{getLearnerPackages()}, \texttt{getLearnerParVals()}, \texttt{getLearnerParamSet()}, \texttt{getLearnerPredictType()}, \texttt{getLearnerShortName()}, \texttt{getLearnerType()}, \texttt{getParamSet()}, \texttt{helpLearner()}, \texttt{helpLearnerParam()}, \texttt{makeLearner()}, \texttt{removeHyperPars()}, \texttt{setHyperPars()}, \texttt{setId()}, \texttt{setLearnerId()}, \texttt{setPredictThreshold()}, \texttt{setPredictType()}

Examples

\begin{verbatim}
makeLearners(c("rpart", "lda"), type = "classif", predict.type = "prob")
\end{verbatim}
makeMeasure

**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 training set d1, results in a model m and 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.

**Usage**

```r
makeMeasure(
  id,
  minimize,
  properties = character(0L),
  fun,
  extra.args = list(),
  aggr = test.mean,
  best = NULL,
  worst = NULL,
  name = id,
  note = ""
)
```

**Arguments**

- **id** (character(1))
  Name of measure.

- **minimize** (logical(1))
  Should the measure be minimized? Default is TRUE.

- **properties** (character)
  Default is character(0).
fun (function(task, model, pred, feats, extra.args))
Calculates the performance value. Usually you will only need the prediction
object pred. - task (Task)
The task. - model (WrappedModel)
The fitted model. - pred (Prediction)
Prediction object. - feats (data.frame)
The features. - extra.args (list)
See below.

extra.args (list)
List of extra arguments which will always be passed to fun. Can be changed
after construction via setMeasurePars(). Default is empty list.

aggr (Aggregation)
Aggregation function, 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.

name (character)
Name of the measure. Default is id.

note (character)
Description and additional notes for the measure. Default is "".

Value
Measure.

See Also
Other performance: ConfusionMatrix, calculateConfusionMatrix(), calculateROCMeasures(),
estimateRelativeOverfitting(), makeCostMeasure(), makeCustomResampledMeasure(), measures,
performance(), setAggregation(), setMeasurePars()

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. learnerId.parameterName.

The predict.type of the Multiplexer is inherited from the predict.type of the base learners.

The getter getLearnerProperties returns the properties of the selected base learner.

**Usage**

```r
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, getNestedTuneResultsOptPathDf(), getNestedTuneResultsX(), getResamplingIndices(), getTuneResult(), makeModelMultiplexerParamSet(), makeTuneControlCMAES(), makeTuneControlDesign(), makeTuneControlGenSA(), makeTuneControlGrid(), makeTuneControlIrace(), makeTuneControlMBO(), makeTuneControlRandom(), makeTuneWrapper(), tuneParams(), tuneThreshold()
Examples

```r
set.seed(123)

library(BBmisc)
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)

df = as.data.frame(res$opt.path)
print(head(df[, -ncol(df)]))

# 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.randomForest"))
)

# 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 less 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 multiplexer learner.
...
(ParamHelpers::ParamSet | ParamHelpers::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 shorter to 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: makeModelMultiplexer()
Other tune: TuneControl, getNestedTuneResultsOptPathDf(), getNestedTuneResultsX(), getResamplingIndices(), getTuneResult(), makeModelMultiplexer(), makeTuneControlCMAES(), makeTuneControlDesign(), makeTuneControlGenSA(), makeTuneControlGrid(), makeTuneControlIrace(), makeTuneControlMBO(), makeTuneControlRandom(), makeTuneWrapper().tuneParams().tuneThreshold()

Examples

# See makeModelMultiplexer
**makeMulticlassWrapper**  
Fuses a base learner with a multi-class method. Creates a learner object, which can be used like any other learner object. This way learners which can only handle binary classification will be able to handle multi-class problems, too.

We use a multiclass-to-binary reduction principle, where multiple binary problems are created from the multiclass task. How these binary problems are generated is defined by an error-correcting-output-code (ECOC) code book. This also allows the simple and well-known one-vs-one and one-vs-rest approaches. Decoding is currently done via Hamming decoding, see e.g. here [https://jmlr.org/papers/volume11/escalera10a/escalera10a.pdf](https://jmlr.org/papers/volume11/escalera10a/escalera10a.pdf).

Currently, the approach always operates on the discrete predicted labels of the binary base models (instead of their probabilities) and the created wrapper cannot predict posterior probabilities.

**Usage**

```r
makeMulticlassWrapper(learner, mcw.method = "onevsrest")
```

**Arguments**

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

- **mcw.method**  
  (character(1) | function)  
  "onevsone" or "onevsrest". You can also pass a function, with signature `function(task)` and which returns a ECOC codematrix with entries +1,-1,0. Columns define new binary problems, rows correspond to classes (rows must be named). 0 means class is not included in binary problem. Default is “onevsrest”.

**Value**

Learner.

**See Also**

Other wrapper: `makeBaggingWrapper()`, `makeClassificationViaRegressionWrapper()`, `makeConstantClassWrapper()`, `makeCostSensClassifWrapper()`, `makeCostSensRegrWrapper()`, `makeDownsampleWrapper()`, `makeDummyFeaturesWrapper()`, `makeExtractFDafeatsWrapper()`, `makeFeatSelWrapper()`, `makeFilterWrapper()`, `makeImputeWrapper()`, `makeMultilabelBinaryRelevanceWrapper()`, `makeMultilabelClassifierChainsWrapper()`, `makeMultilabelDBRWrapper()`, `makeMultilabelNestedStackingWrapper()`, `makeMultilabelStackingWrapper()`, `makeOverBaggingWrapper()`, `makePreprocWrapper()`, `makePreprocWrapperCaret()`, `makeRemoveConstantFeaturesWrapper()`, `makeSMOTEWrapper()`, `makeTuneWrapper()`, `makeUndersampleWrapper()`, `makeWeightedClassesWrapper()`
Use binary relevance method to create a multilabel learner.

**Description**

Every learner which is implemented in mlr and which supports binary classification can be converted to a wrapped binary relevance multilabel learner. The multilabel classification problem is converted into simple binary classifications for each label/target on which the binary learner is applied.

Models can easily be accessed via `getLearnerModel`.

Note that it does not make sense to set a threshold in the used base learner when you predict probabilities. On the other hand, it can make a lot of sense, to call `setThreshold` on the `MultilabelBinaryRelevanceWrapper` for each label individually; Or to tune these thresholds with `tuneThreshold`; especially when you face very unbalanced class distributions for each binary label.

**Usage**

```r
makeMultilabelBinaryRelevanceWrapper(learner)
```

**Arguments**

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

**Value**

*Learner.*

**References**


**See Also**

Other wrapper: `makeBaggingWrapper()`, `makeClassificationViaRegressionWrapper()`, `makeConstantClassWrapper()`, `makeCostSensClassifWrapper()`, `makeCostSensRegrWrapper()`, `makeDownsampleWrapper()`, `makeDummyFeaturesWrapper()`, `makeExtractFDAFeatsWrapper()`, `makeFeatSelWrapper()`, `makeFilterWrapper()`, `makeImputeWrapper()`, `makeMulticlassWrapper()`, `makeMultilabelClassifierChainsWrapper()`, `makeMultilabelDBRWrapper()`, `makeMultilabelNestedStackingWrapper()`, `makeMultilabelStackingWrapper()`, `makeOverBaggingWrapper()`, `makePreprocWrapper()`, `makePreprocWrapperCaret()`, `makeRemoveConstantFeaturesWrapper()`, `makeSMOTEWrapper()`, `makeTuneWrapper()`, `makeUndersampleWrapper()`, `makeWeightedClassesWrapper()`

Other multilabel: `getMultilabelBinaryPerformances()`, `makeMultilabelClassifierChainsWrapper()`, `makeMultilabelDBRWrapper()`, `makeMultilabelNestedStackingWrapper()`, `makeMultilabelStackingWrapper()`
Examples

```r
if (requireNamespace("rpart")) {
  d = getTaskData(yeast.task)
  # drop some labels so example runs faster
  d = d[seq(1, nrow(d), by = 20), c(1:2, 15:17)]
  task = makeMultilabelTask(data = d, target = c("label1", "label2"))
  lrn = makeLearner("classif.rpart")
  lrn = makeMultilabelBinaryRelevanceWrapper(lrn)
  lrn = setPredictType(lrn, "prob")
  # train, predict and evaluate
  mod = train(lrn, task)
  pred = predict(mod, task)
  performance(pred, measure = list(multilabel.hamloss, multilabel.subset01, multilabel.f1))
  # the next call basically has the same structure for any multilabel meta wrapper
  getMultilabelBinaryPerformances(pred, measures = list(mmce, auc))
  # above works also with predictions from resample!
}
```

makeMultilabelClassifierChainsWrapper

*Use classifier chains method (CC) to create a multilabel learner.*

Description

Every learner which is implemented in mlr and which supports binary classification can be converted to a wrapped classifier chains multilabel learner. CC trains a binary classifier for each label following a given order. In training phase, the feature space of each classifier is extended with true label information of all previous labels in the chain. During the prediction phase, when true labels are not available, they are replaced by predicted labels.

Models can easily be accessed via `getLearnerModel`.

Usage

```r
makeMultilabelClassifierChainsWrapper(learner, order = NULL)
```

Arguments

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

- **order** *(character)*
  Specifies the chain order using the names of the target labels. E.g. for m target labels, this must be a character vector of length m that contains a permutation of the target label names. Default is `NULL` which uses a random ordering of the target label names.

Value

`Learner`.
makeMultilabelDBRWrapper

Use dependent binary relevance method (DBR) to create a multilabel learner.

Description

Every learner which is implemented in mlr and which supports binary classification can be converted to a wrapped DBR multilabel learner. The multilabel classification problem is converted into simple binary classifications for each label/target on which the binary learner is applied. For each target, actual information of all binary labels (except the target variable) is used as additional features. During prediction these labels need are obtained by the binary relevance method using the same binary learner.

Models can easily be accessed via getLearnerModel.
**Usage**

```r
makeMultilabelDBRWrapper(learner)
```

**Arguments**

- **learner**  
  
  (Learner \ character(1))  
  The learner. If you pass a string the learner will be created via `makeLearner`.

**Value**

Learner.

**References**

Montanes, E. et al. (2013) *Dependent binary relevance models for multi-label classification* Artificial Intelligence Center, University of Oviedo at Gijon, Spain.

**See Also**

Other wrapper: `makeBaggingWrapper()`, `makeClassificationViaRegressionWrapper()`, `makeCostSensClassifWrapper()`, `makeCostSensRegrWrapper()`, `makeDownsampleWrapper()`, `makeDummyFeaturesWrapper()`, `makeExtractFDAFeatsWrapper()`, `makeFeatSelWrapper()`, `makeFilterWrapper()`, `makeImputeWrapper()`, `makeMulticlassWrapper()`, `makeMultilabelBinaryRelevanceWrapper()`, `makeMultilabelClassifierChainsWrapper()`, `makeMultilabelNestedStackingWrapper()`, `makeMultilabelStackingWrapper()`, `makeOverBaggingWrapper()`, `makePreprocWrapper()`, `makePreprocWrapperCaret()`, `makeRemoveConstantFeaturesWrapper()`, `makeSMOTEWrapper()`, `makeTuneWrapper()`, `makeUndersampleWrapper()`, `makeWeightedClassesWrapper()

Other multilabel: `getMultilabelBinaryPerformances()`, `makeMultilabelBinaryRelevanceWrapper()`, `makeMultilabelClassifierChainsWrapper()`, `makeMultilabelNestedStackingWrapper()`, `makeMultilabelStackingWrapper()`, `makeMultilabelDBRWrapper`

**Examples**

```r
if (requireNamespace("rpart")) {
  d = getTaskData(yeast.task)  
  # drop some labels so example runs faster  
  d = d[seq(1, nrow(d), by = 20), c(1:2, 15:17)]  
  task = makeMultilabelTask(data = d, target = c("label1", "label2"))  
  lrn = makeLearner("classif.rpart")  
  lrn = makeMultilabelBinaryRelevanceWrapper(lrn)  
  lrn = setPredictType(lrn, "prob")  
  # train, predict and evaluate  
  mod = train(lrn, task)  
  pred = predict(mod, task)  
  performance(pred, measure = list(multilabel.hamloss, multilabel.subset01, multilabel.f1))  
  # the next call basically has the same structure for any multilabel meta wrapper  
  getMultilabelBinaryPerformances(pred, measures = list(mmce, auc))  
  # above works also with predictions from resample!
}
```
makeMultilabelNestedStackingWrapper

Use nested stacking method to create a multilabel learner.

Description

Every learner which is implemented in mlr and which supports binary classification can be converted to a wrapped nested stacking multilabel learner. Nested stacking trains a binary classifier for each label following a given order. In training phase, the feature space of each classifier is extended with predicted label information (by cross validation) of all previous labels in the chain. During the prediction phase, predicted labels are obtained by the classifiers, which have been learned on all training data.

Models can easily be accessed via `getLearnerModel`.

Usage

```r
makeMultilabelNestedStackingWrapper(learner, order = NULL, cv.folds = 2)
```

Arguments

- `learner` (Learner | character(1))
  The learner. If you pass a string the learner will be created via `makeLearner`.
- `order` (character)
  Specifies the chain order using the names of the target labels. E.g. for `m` target labels, this must be a character vector of length `m` that contains a permutation of the target label names. Default is `NULL` which uses a random ordering of the target label names.
- `cv.folds` (integer(1))
  The number of folds for the inner cross validation method to predict labels for the augmented feature space. Default is 2.

Value

Learner.

References

Montanes, E. et al. (2013), Dependent binary relevance models for multi-label classification Artificial Intelligence Center, University of Oviedo at Gijon, Spain.

See Also

makeMultilabelStackingWrapper

Use stacking method (stacked generalization) to create a multilabel learner.

Description

Every learner which is implemented in mlr and which supports binary classification can be converted to a wrapped stacking multilabel learner. Stacking trains a binary classifier for each label using predicted label information of all labels (including the target label) as additional features (by cross validation). During prediction these labels need are obtained by the binary relevance method using the same binary learner.

Models can easily be accessed via `getLearnerModel`.

Usage

```r
makeMultilabelStackingWrapper(learner, cv.folds = 2)
```

Arguments

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

- **cv.folds** *(integer(1))*
  The number of folds for the inner cross validation method to predict labels for the augmented feature space. Default is 2.

Examples

```r
if (requireNamespace("rpart")) {
  d = getTaskData(yeast.task)
  # drop some labels so example runs faster
  d = d[seq(1, nrow(d), by = 20), c(1:2, 15:17)]
  task = makeMultilabelTask(data = d, target = c("label1", "label2"))
  lrn = makeLearner("classif.rpart")
  lrn = makeMultilabelBinaryRelevanceWrapper(lrn)
  lrn = setPredictType(lrn, "prob")
  # train, predict and evaluate
  mod = train(lrn, task)
  pred = predict(mod, task)
  performance(pred, measure = list(multilabel.hamloss, multilabel.subset01, multilabel.f1))
  # the next call basically has the same structure for any multilabel meta wrapper
  getMultilabelBinaryPerformances(pred, measures = list(mmce, auc))
  # above works also with predictions from resample!
}
```
makeMultilabelTask

Create a multilabel task.

Description

Create a multilabel task.

Value

Learner.

References

Montanes, E. et al. (2013) Dependent binary relevance models for multi-label classification Artificial Intelligence Center, University of Oviedo at Gijon, Spain.

See Also

Other wrapper: makeBaggingWrapper(), makeClassificationViaRegressionWrapper(), makeConstantClassWrapper(), makeCostSensClassifWrapper(), makeCostSensRegrWrapper(), makeDownsampleWrapper(), makeDummyFeaturesWrapper(), makeExtractFDAFeatsWrapper(), makeFeatSelWrapper(), makeFilterWrapper(), makeImputeWrapper(), makeMulticlassWrapper(), makeMultilabelBinaryRelevanceWrapper(), makeMultilabelClassifierChainsWrapper(), makeMultilabelDBRWrapper(), makeMultilabelNestedStackingWrapper(), makeOverBaggingWrapper(), makePreprocWrapper(), makePreprocWrapperCaret(), makeRemoveConstantFeaturesWrapper(), makeSMOTEWrapper(), makeTuneWrapper(), makeUndersampleWrapper(), makeWeightedClassesWrapper()

Other multilabel: getMultilabelBinaryPerformances(), makeMultilabelBinaryRelevanceWrapper(), makeMultilabelClassifierChainsWrapper(), makeMultilabelDBRWrapper(), makeMultilabelNestedStackingWrapper()

Examples

```r
if (requireNamespace("rpart")) {
  d = getTaskData(yeast.task)
  # drop some labels so example runs faster
  d = d[seq(1, nrow(d), by = 20), c(1:2, 15:17)]
  task = makeMultilabelTask(data = d, target = c("label1", "label2"))
  lrn = makeLearner("classif.rpart")
  lrn = makeMultilabelBinaryRelevanceWrapper(lrn)
  lrn = setPredictType(lrn, "prob")
  # train, predict and evaluate
  mod = train(lrn, task)
  pred = predict(mod, task)
  performance(pred, measure = list(multilabel.hamloss, multilabel.subset01, multilabel.f1))
  # the next call basically has the same structure for any multilabel meta wrapper
  getMultilabelBinaryPerformances(pred, measures = list(mmce, auc))
  # above works also with predictions from resample!
}
```
Usage

makeMultilabelTask(
    id = deparse(substitute(data)),
    data,
    target,
    weights = NULL,
    blocking = NULL,
    coordinates = NULL,
    fixup.data = "warn",
    check.data = TRUE
)

Arguments

id (character(1))
Id string for object. Default is the name of the R variable passed to data.

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

target (character(1) | character(2) | character(n.classes))
Name(s) of the target variable(s). For survival analysis these are the names of the survival time and event columns, so it has length 2. For multilabel classification it contains the names of the logical columns that encode whether a label is present or not and its length corresponds to the number of classes.

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.

coordinates (data.frame)
Coordinates of a spatial data set that will be used for spatial partitioning of the data in a spatial cross-validation resampling setting. Coordinates have to be numeric values. Provided data.frame needs to have the same number of rows as data and consist of at least two dimensions.

fixup.data (character(1))
Should some basic cleaning up of data be performed? Currently this means removing empty factor levels for the columns. Possible choices 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.
**Details**

For multilabel classification we assume that the presence of labels is encoded via logical columns in data. The name of the column specifies the name of the label. `target` is then a char vector that points to these columns.

**Note**

For multilabel classification we assume that the presence of labels is encoded via logical columns in data. The name of the column specifies the name of the label. `target` is then a char vector that points to these columns.

**See Also**

Task ClassifTask ClusterTask CostSensTask RegrTask SurvTask

---

**Description**

Fuses a classification learner for binary classification with an over-bagging method for imbalanc

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 `getLearnerModel`.

OverBagging is implemented as follows: For each iteration a random data subset is sampled. Class examples are oversampled with replacement with a given rate. Members of the other class 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**

```r
makeOverBaggingWrapper(
  learner,
  obw.iters = 10L,
  obw.rate = 1,
  obw.maxcl = "boot",
  obw.cl = NULL
)
```
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 a 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))
How should other class (usually larger class) be handled? “all” means every instance of the class gets in each bag, “boot” means the class instances are bootstrapped in each iteration. Default is “boot”.

obw.cl (character(1))
Which class should be over- or undersampled. If NULL, makeOverBaggingWrapper will take the smaller class.

Value

Learner.

See Also

Other imbalancy: makeUndersampleWrapper(), oversample(), smote()
Other wrapper: makeBaggingWrapper(), makeClassificationViaRegressionWrapper(), makeConstantClassWrapper(), makeCostSensClassifWrapper(), makeCostSensRegrWrapper(), makeDownsampleWrapper(), makeDummyFeaturesWrapper(), makeExtractFDAFeatsWrapper(), makeFeatSelWrapper(), makeFilterWrapper(), makeImputeWrapper(), makeMulticlassWrapper(), makeMultilabelBinaryRelevanceWrapper(), makeMultilabelClassifierChainsWrapper(), makeMultilabelDBRWrapper(), makeMultilabelNestedStackingWrapper(), makeMultilabelStackingWrapper(), makePreprocWrapper(), makePreprocWrapperCaret(), makeRemoveConstantFeaturesWrapper(), makeSMOTEWrapper(), makeTuneWrapper(), makeUndersampleWrapper(), makeWeightedClassesWrapper()
**makePreprocWrapper**

Usage

```r
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 arguments that were passed to `train`. `control` is the object you returned in `train`. Must return the processed data.

- **par.set** *(ParamHelpers::ParamSet)*
  Parameter set of ParamHelpers::LearnerParam objects to describe the parameters in `args`. Default is empty set.

- **par.vals** *(list)*
  Named list of default values for params in `args` respectively `par.set`. Default is empty list.

Value

*(Learner)*.

See Also

Other wrapper: `makeBaggingWrapper()`, `makeClassificationViaRegressionWrapper()`, `makeCostSensClassifWrapper()`, `makeCostSensRegrWrapper()`, `makeDownsampleWrapper()`,
`makeDummyFeaturesWrapper()`, `makeExtractFDAFeatsWrapper()`, `makeFeatSelWrapper()`, `makeFilterWrapper()`,
`makeImputeWrapper()`, `makeMulticlassWrapper()`, `makeMultilabelBinaryRelevanceWrapper()`,
`makeMultilabelClassifierChainsWrapper()`, `makeMultilabelDBRWrapper()`, `makeMultilabelNestedStackingWrapper()`,
`makeMultilabelStackingWrapper()`, `makeOverBaggingWrapper()`, `makePreprocWrapperCaret()`,
`makeRemoveConstantFeaturesWrapper()`, `makeSMOTEWrapper()`, `makeTuneWrapper()`, `makeUndersampleWrapper()`,
`makeWeightedClassesWrapper()`
Fuse learner with preprocessing.

**Description**

Fuses a learner with preprocessing methods provided by `caret::preProcess`. Before training the preprocessing will be performed and the preprocessing model will be stored. Before prediction the preprocessing model will transform the test data according to the trained model.

After being wrapped the learner will support missing values although this will only be the case if `ppc.knnImpute`, `ppc.bagImpute` or `ppc.medianImpute` is set to `TRUE`.

**Usage**

```r
makePreprocWrapperCaret(learner, ...)
```

**Arguments**

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

- `...` *(any)*
  See `caret::preProcess` for parameters not listed above. If you use them you might want to define them in the `add.par.set` so that they can be tuned.

**Value**

- Learner.

**See Also**

Other wrapper: `makeBaggingWrapper()`, `makeClassificationViaRegressionWrapper()`, `makeConstantClassWrapper()`, `makeCostSensClassifWrapper()`, `makeCostSensRegrWrapper()`, `makeDownsampleWrapper()`, `makeDummyFeaturesWrapper()`, `makeExtractFDAFeatsWrapper()`, `makeFeatSelWrapper()`, `makeFilterWrapper()`, `makeImputeWrapper()`, `makeMulticlassWrapper()`, `makeMultilabelBinaryRelevanceWrapper()`, `makeMultilabelClassifierChainsWrapper()`, `makeMultilabelDBRWrapper()`, `makeMultilabelNestedStackingWrapper()`, `makeMultilabelStackingWrapper()`, `makeOverBaggingWrapper()`, `makePreprocWrapper()`, `makeRemoveConstantFeatureWrapper()`, `makeSMOTEWrapper()`, `makeTuneWrapper()`, `makeUndersampleWrapper()`, `makeWeightedClassesWrapper()`
**makeRegrTask**

Create a regression task.

**Description**

Create a regression task.

**Usage**

```r
makeRegrTask(
  id = deparse(substitute(data)),
  data,
  target,
  weights = NULL,
  blocking = NULL,
  coordinates = NULL,
  fixup.data = "warn",
  check.data = TRUE
)
```

**Arguments**

- **id** (character(1))
  Id string for object. Default is the name of the R variable passed to `data`.

- **data** (data.frame)
  A data frame containing the features and target variable(s).

- **target** (character(1) | character(2) | character(n.classes))
  Name(s) of the target variable(s). For survival analysis these are the names of the survival time and event columns, so it has length 2. For multilabel classification it contains the names of the logical columns that encode whether a label is present or not and its length corresponds to the number of classes.

- **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.

- **coordinates** (data.frame)
  Coordinates of a spatial data set that will be used for spatial partitioning of the data in a spatial cross-validation resampling setting. Coordinates have to be numeric values. Provided `data.frame` needs to have the same number of rows as data and consist of at least two dimensions.
Should some basic cleaning up of data be performed? Currently this means removing empty factor levels for the columns. Possible choices are: “no” = Don’t do it. “warn” = Do it but warn about it. “quiet” = Do it but keep silent. Default is “warn”.

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.

See Also

Task ClassifTask CostSensTask ClusterTask MultilabelTask SurvTask

Description

Fuses a base learner with the preprocessing implemented in removeConstantFeatures.

Usage

makeRemoveConstantFeaturesWrapper(  learner,  perc = 0,  dont.rm = character(0L),  na.ignore = FALSE,  wrap.tol = .Machine$double.eps^0.5)

Arguments

learner (Learner | character(1)) The learner. If you pass a string the learner will be created via makeLearner.
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?) Note that if the feature has only missing values, it is always removed. Default is FALSE.
wrap.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{\text{.Machine$double.eps}}}.  

Fuse learner with removal of constant features preprocessing.
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,
  stratify.cols = NULL,
  fixed = FALSE,
  blocking.cv = FALSE
)

Arguments

method (character(1))

predict (character(1))
  What to predict during resampling: "train", "test" or "both" sets. Default is "test".

... (any)
  Further parameters for strategies.
**makeResampleDesc**

**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.

**horizon** (numeric(1)) Number of observations in the forecast test set for “GrowingWindowCV” and “FixedWindowCV”. When horizon > 1 this will be treated as the number of observations to forecast, else it will be a fraction of the initial window. IE, for 100 observations, initial window of .5, and horizon of .2, the test set will have 10 observations. Default is 1.

**initial.window** (numeric(1)) Fraction of observations to start with in the training set for “GrowingWindowCV” and “FixedWindowCV”. When initial.window > 1 this will be treated as the number of observations in the initial window, else it will be treated as the fraction of observations to have in the initial window. Default is 0.5.

**skip** (numeric(1)) How many resamples to skip to thin the total amount for “GrowingWindowCV” and “FixedWindowCV”. This is passed through as the “by” argument in seq(). When skip > 1 this will be treated as the increment of the sequence of resampling indices, else it will be a fraction of the total training indices. IE for 100 training sets and a value of .2, the increment of the resampling indices will be 20. Default is “horizon” which gives mutually exclusive chunks of test indices.

**stratify** (logical(1)) Should stratification be done for the target variable? For 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. For survival tasks stratification is done on the events, resulting in training sets with comparable censoring rates.

**stratify.cols** (character) Stratify on specific columns referenced by name. All columns have to be factor or integer. Note that you have to ensure yourself that stratification is possible, i.e. that each strata contains enough observations. This argument and stratify are mutually exclusive.

**fixed** (logical(1)) Whether indices supplied via argument ‘blocking’ in the task should be used as fully pre-defined indices. Default is FALSE which means they will be used following the ‘blocking’ approach. fixed only works with ResampleDesc CV and the supplied indices must match the number of observations. When fixed = TRUE, the iters argument will be ignored and is internally set to the number of supplied factor levels in blocking.

**blocking.cv** (logical(1)) Should ‘blocking’ be used in CV? Default to FALSE. This is different to fixed
= TRUE and cannot be combined. Please check the mlr online tutorial for more details.

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)`.

Standard ResampleDesc objects

For common resampling strategies you can save some typing by using the following description objects:

- `hout` holdout a.k.a. test sample estimation (two-thirds training set, one-third testing set)
- `cv2` 2-fold cross-validation
- `cv3` 3-fold cross-validation
- `cv5` 5-fold cross-validation
- `cv10` 10-fold cross-validation

See Also

Other resample: `ResamplePrediction`, `ResampleResult`, `addRRMeasure()`, `getRRPredictionList()`, `getRRPredictions()`, `getRRTaskDesc()`, `getRRTaskDescription()`, `makeResampleInstance()`, `resample()`
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 specific iterations 'belong together' (e.g. repeated CV), and it can later be used to aggregate performance values accordingly. Default is 'factor()'.

makeRLearner.classif.fdausc.glm

Value

(ResampleInstance).

See Also

Other resample: ResamplePrediction, ResampleResult, addRRMeasure(), getRRPredictionList(), getRRPredictions(), getRRTaskDesc(), getRRTaskDescription(), makeResampleDesc(), resample()

Examples

```r
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)
```

makeRLearner.classif.fdausc.kernel

Learner for kernel classification for functional data.

Description

Learner for kernel Classification.

Usage

```r
## S3 method for class 'classif.fdausc.kernel'
makeRLearner()
```

---

makeRLearner.classif.fdausc.glm

Classification of functional data by Generalized Linear Models.

Description

Learner for classification using Generalized Linear Models.

Usage

```r
## S3 method for class 'classif.fdausc.glm'
makeRLearner()
```
makeR Learner.classif.fdausc.np

Learner for nonparametric classification for functional data.

Description

Learner for Nonparametric Supervised Classification.

Usage

## S3 method for class 'classif.fdausc.np'
makeR Learner()

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,
  sw.standardize = TRUE,
  sw.alt.logic = FALSE
)

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.
Description

A stacked learner uses predictions of several base learners and fits a super learner using these predictions as features in order to predict the outcome. The following stacking methods are available:

- **average**
  Averaging of base learner predictions without weights.

- **stack.nocv**
  Fits the super learner, where in-sample predictions of the base learners are used.

- **stack.cv**
  Fits the super learner, where the base learner predictions are computed by cross-validated predictions (the resampling strategy can be set via the `resampling` argument).

- **hill.climb**
  Select a subset of base learner predictions by hill climbing algorithm.

- **compress**
  Train a neural network to compress the model from a collection of base learners.

Value

Learner.

See Also

Other wrapper: `makeBaggingWrapper()`, `makeClassificationViaRegressionWrapper()`, `makeCostSensClassifWrapper()`, `makeCostSensRegrWrapper()`, `makeDownsampleWrapper()`, `makeDummyFeaturesWrapper()`, `makeExtractFDAFeatsWrapper()`, `makeFeatSelWrapper()`, `makeFilterWrapper()`, `makeImputeWrapper()`, `makeMulticlassWrapper()`, `makeMultilabelBinaryRelevanceWrapper()`, `makeMultilabelClassifierChainsWrapper()`, `makeMultilabelDBRWrapper()`, `makeMultilabelNestedStackingWrapper()`, `makeMultilabelStackingWrapper()`, `makeOverBaggingWrapper()`, `makePreprocWrapper()`, `makePreprocWrapperCaret()`, `makeRemoveConstantFeaturesWrapper()`, `makeTuneWrapper()`, `makeUndersampleWrapper()`, `makeWeightedClassesWrapper()`
Usage

makeStackedLearner(
  base.learners,
  super.learner = NULL,
  predict.type = NULL,
  method = "stack.nocv",
  use.feat = FALSE,
  resampling = NULL,
  parset = list()
)

Arguments

base.learners  
(list of Learner)
A list of learners created with makeLearner.

super.learner  (Learner | character(1))
The super learner that makes the final prediction based on the base learners. If you pass a string, the super learner will be created via makeLearner. Not used for method = 'average'. Default is NULL.

predict.type  (character(1))
Sets the type of the final prediction for method = 'average'. For other methods, the predict type should be set within super.learner. If the type of the base learner prediction, which is set up within base.learners, is

  • "prob" then predict.type = 'prob' will use the average of all base learner predictions and predict.type = 'response' will use the class with highest probability as final prediction.

  • "response" then, for classification tasks with predict.type = 'prob', the final prediction will be the relative frequency based on the predicted base learner classes and classification tasks with predict.type = 'response' will use majority vote of the base learner predictions to determine the final prediction. For regression tasks, the final prediction will be the average of the base learner predictions.

method  (character(1))
"average" for averaging the predictions of the base learners, "stack.nocv" for building a super learner using the predictions of the base learners, "stack.cv" for building a super learner using cross-validated predictions of the base learners. "hill.climb" for averaging the predictions of the base learners, with the weights learned from hill climbing algorithm and "compress" for compressing the model to mimic the predictions of a collection of base learners while speeding up the predictions and reducing the size of the model. Default is "stack.nocv".

use.feat  (logical(1))
Whether the original features should also be passed to the super learner. Not used for method = 'average'. Default is FALSE.
Resampling strategy for method = 'stack.cv'. Currently only CV is allowed for resampling. The default NULL uses 5-fold CV.

The parameters for hill.climb method, including:
- replace
  Whether a base learner can be selected more than once.
- init
  Number of best models being included before the selection algorithm.
- bagprob
  The proportion of models being considered in one round of selection.
- bagtime
  The number of rounds of the bagging selection.
- metric
  The result evaluation metric function taking two parameters pred and true, the smaller the score the better.

The parameters for compress method, including:
- k
  the size multiplier of the generated data
- prob
  the probability to exchange values
- s
  the standard deviation of each numerical feature

Examples

```r
# Classification
data(iris)
tsk = makeClassifTask(data = iris, target = "Species")
base = c("classif.rpart", "classif.lda", "classif.svm")
lrns = lapply(base, makeLearner)
lrns = lapply(lrns, setPredictType, "prob")
m = makeStackedLearner(base.learners = lrns,
  predict.type = "prob", method = "hill.climb")
tmp = train(m, tsk)
res = predict(tmp, tsk)

# Regression
data(BostonHousing, package = "mlbench")
tsk = makeRegrTask(data = BostonHousing, target = "medv")
base = c("regr.rpart", "regr.svm")
lrns = lapply(base, makeLearner)
m = makeStackedLearner(base.learners = lrns,
  predict.type = "response", method = "compress")
tmp = train(m, tsk)
res = predict(tmp, tsk)
```
makeSurvTask  

Create a survival task.

Description

Create a survival task.

Usage

```r
makeSurvTask(
  id = deparse(substitute(data)),
  data,
  target,
  weights = NULL,
  blocking = NULL,
  coordinates = NULL,
  fixup.data = "warn",
  check.data = TRUE
)
```

Arguments

- **id**
  - (character(1))
  - Id string for object. Default is the name of the R variable passed to `data`.
- **data**
  - (data.frame)
  - A data frame containing the features and target variable(s).
- **target**
  - (character(1) | character(2) | character(n.classes))
  - Name(s) of the target variable(s). For survival analysis these are the names of the survival time and event columns, so it has length 2. For multilabel classification it contains the names of the logical columns that encode whether a label is present or not and its length corresponds to the number of classes.
- **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.
coordinates (data.frame)
Coordinates of a spatial data set that will be used for spatial partitioning of the data in a spatial cross-validation resampling setting. Coordinates have to be numeric values. Provided data.frame needs to have the same number of rows as data and consist of at least two dimensions.

fixup.data (character(1))
Should some basic cleaning up of data be performed? Currently this means removing empty factor levels for the columns. Possible choices 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.

See Also
Task ClassifTask ClusterTask CostSensTask MultilabelTask RegrTask
makeTuneControlCMAES

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. Imputation is only active if on.learner.error is configured not to stop in configureMlr. 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  (list)
Named list of initial parameter values.

tune.threshold  (logical(1))
Should the threshold be tuned for the measure at hand, after each hyperparameter evaluation, via tuneThreshold? Only works for classification if the predict type is “prob”. Default is FALSE.

tune.threshold.args  (list)
Further arguments for threshold tuning that are passed down to tuneThreshold. Default is none.

log.fun  (function | character(1))
Function used for logging. If set to “default” (the default), the evaluated design points, the resulting performances, and the runtime will be reported. If set to “memory” the memory usage for each evaluation will also be displayed, with character(1) small increase in run time. Otherwise character(1) function with arguments learner, resampling, measures, par.set, control, opt.path, dob, x, y, remove.nas, stage and prev.stage is expected. The default displays the performance measures, the time needed for evaluating, the currently used memory and the max memory ever used before (the latter two both taken from gc). See the implementation for details.

final.dw.perc  (boolean)
If a Learner wrapped by a makeDownsampleWrapper is used, you can define the value of dw.perc which is used to train the Learner with the final parameter setting found by the tuning. Default is NULL which will not change anything.

budget  (integer(1))
Maximum budget for tuning. This value restricts the number of function evaluations. The budget corresponds to the product of the number of generations (maxit) and the number of offsprings per generation (lambda).

...  (any)
Further control parameters passed to the control arguments of cmaes::cma_es or GenSA::GenSA, as well as towards the tunerConfig argument of irace::irace.

Value

(TuneControlCMAES)
**makeTuneControlDesign**  
Create control object for hyperparameter tuning with predefined design.

**Description**
Completely pre-specify a data.frame of design points to be evaluated during tuning. All kinds of parameter types can be handled.

**Usage**
```r
makeTuneControlDesign(
  same.resampling.instance = TRUE,
  impute.val = NULL,
  design = NULL,
  tune.threshold = FALSE,
  tune.threshold.args = list(),
  log.fun = "default"
)
```

**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. Imputation is only active if on.learner.error is configured not to stop in configureMlr. 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.

- `design`  
  (data.frame)  
  data.frame containing the different parameter settings to be evaluated. The columns have to be named according to the ParamSet which will be used in

**See Also**
Other tune: TuneControl, getNestedTuneResultsOptPathDf(), getNestedTuneResultsX(), getResamplingIndices(), getTuneResult(), makeModelMultiplexer(), makeModelMultiplexerParamSet(), makeTuneControlDesign(), makeTuneControlGenSA(), makeTuneControlGrid(), makeTuneControlIrace(), makeTuneControlMBO(), makeTuneControlRandom(), makeTuneWrapper(), tuneParams(), tuneThreshold()
tune(). Proper designs can be created with `ParamHelpers::generateDesign` for instance.

**tune.threshold** (logical(1))
Should the threshold be tuned for the measure at hand, after each hyperparameter evaluation, via `tuneThreshold`? Only works for classification if the predict type is “prob”. Default is FALSE.

**tune.threshold.args** (list)
Further arguments for threshold tuning that are passed down to `tuneThreshold`. Default is none.

**log.fun** (function | character(1))
Function used for logging. If set to “default” (the default), the evaluated design points, the resulting performances, and the runtime will be reported. If set to “memory” the memory usage for each evaluation will also be displayed, with character(1) small increase in run time. Otherwise character(1) function with arguments learner, resampling, measures, par.set, control, opt.path, dob, x, y, remove.nas, stage and prev.stage is expected. The default displays the performance measures, the time needed for evaluating, the currently used memory and the max memory ever used before (the latter two both taken from gc). See the implementation for details.

**Value**

(TuneControlDesign)

**See Also**

Other `tune`: `TuneControl`, `getNestedTuneResultsOptPathDf()`, `getNestedTuneResultsX()`, `getResamplingIndices()`, `getTuneResult()`, `makeModelMultiplexer()`, `makeModelMultiplexerParamSet()`, `makeTuneControlCMAES()`, `makeTuneControlGenSA()`, `makeTuneControlGrid()`, `makeTuneControlIrace()`, `makeTuneControlMBO()`, `makeTuneControlRandom()`, `makeTuneWrapper()`, `tuneParams()`, `tuneThreshold()`

---

**makeTuneControlGenSA**  Create control object for hyperparameter tuning with GenSA.

**Description**

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

**Usage**

```r
makeTuneControlGenSA(
  same.resampling.instance = TRUE,
  impute.val = NULL,
  start = NULL,
)```
tune.threshold = FALSE,
tune.threshold.args = list(),
log.fun = "default",
final.dw.perc = NULL,
budget = NULL,
...)
)

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. Imputation is only active if on.learner.error is configured not to stop in configureMlr. 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
(list)
Named list of initial parameter values.

tune.threshold
(logical(1))
Should the threshold be tuned for the measure at hand, after each hyperparameter evaluation, via tuneThreshold? Only works for classification if the predict type is "prob". Default is FALSE.

tune.threshold.args
(list)
Further arguments for threshold tuning that are passed down to tuneThreshold. Default is none.

log.fun
(function | character(1))
Function used for logging. If set to "default" (the default), the evaluated design points, the resulting performances, and the runtime will be reported. If set to "memory" the memory usage for each evaluation will also be displayed, with character(1) small increase in run time. Otherwise character(1) function with arguments learner, resampling, measures, par.set, control, opt.path, dob, x, y, remove.nas, stage and prev.stage is expected. The default displays the performance measures, the time needed for evaluating, the currently used memory and the max memory ever used before (the latter two both taken from gc). See the implementation for details.

final.dw.perc
(boolean)
If a Learner wrapped by a makeDownsampleWrapper is used, you can define
makeTuneControlGrid

Create control object for hyperparameter tuning with grid search.

Description

A basic grid search can handle all kinds of parameter types. You can either use their correct param type and resolution, or discretize them yourself by always using ParamHelpers::makeDiscreteParam in the par.set passed to tuneParams.

Usage

makeTuneControlGrid(
  same.resampling.instance = TRUE,
  impute.val = NULL,
  resolution = 10L,
  tune.threshold = FALSE,
  tune.threshold.args = list(),
  log.fun = "default",
  final.dw.perc = NULL,
  budget = NULL
)
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. Imputation is only active if on.learner.error is configured not to stop in configureMlr. 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.

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 ParamHelpers::generateGridDesign. Default is 10.

tune.threshold (logical(1))
Should the threshold be tuned for the measure at hand, after each hyperparameter evaluation, via tuneThreshold? Only works for classification if the predict type is “prob”. Default is FALSE.

tune.threshold.args (list)
Further arguments for threshold tuning that are passed down to tuneThreshold. Default is none.

log.fun (function | character(1))
Function used for logging. If set to “default” (the default), the evaluated design points, the resulting performances, and the runtime will be reported. If set to “memory” the memory usage for each evaluation will also be displayed, with character(1) small increase in run time. Otherwise character(1) function with arguments learner, resampling, measures, par.set, control, opt.path, dob, x, y, remove.nas, stage and prev.stage is expected. The default displays the performance measures, the time needed for evaluating, the currently used memory and the max memory ever used before (the latter two both taken from gc). See the implementation for details.

final.dw.perc (boolean)
If a Learner wrapped by a makeDownsampleWrapper is used, you can define the value of dw.perc which is used to train the Learner with the final parameter setting found by the tuning. Default is NULL which will not change anything.

budget (integer(1))
Maximum budget for tuning. This value restricts the number of function evaluations. If set, must equal the size of the grid.
makeTuneControlIrace

Create control object for hyperparameter tuning with Irace.

Description

Tuning with iterated F-Racing with method `irace::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. More information on irace can be found in package vignette: vignette("irace-package", package = "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::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

```r
makeTuneControlIrace(
  impute.val = NULL,
  n.instances = 100L,
  show.irace.output = FALSE,
  tune.threshold = FALSE,
  tune.threshold.args = list(),
  log.fun = "default",
  final.dw.perc = NULL,
  budget = NULL,
  ...
)
```

Arguments

- `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. Imputation is only active if on.learner.error is configured not to stop in `configureMlr`. It is not stored in the optimization path, an NA and a corresponding error message

Value

(TuneControlGrid)

See Also

Other tune: `TuneControl`, `getNestedTuneResultsOptPathDf()`, `getNestedTuneResultsX()`, `getResamplingIndices()`, `getTuneResult()`, `makeModelMultiplexer()`, `makeModelMultiplexerParamSet()`, `makeTuneControlCMAES()`, `makeTuneControlDesign()`, `makeTuneControlGenSA()`, `makeTuneControlIrace()`, `makeTuneControlMBO()`, `makeTuneControlRandom()`, `makeTuneWrapper()`, `tuneParams()`, `tuneThreshold()`
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.

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.

tune.threshold  (logical(1))
Should the threshold be tuned for the measure at hand, after each hyperparam-
eter evaluation, via tuneThreshold? Only works for classification if the predict
type is “prob”. Default is FALSE.

tune.threshold.args  (list)
Further arguments for threshold tuning that are passed down to tuneThreshold.
Default is none.

log.fun  (function | character(1))
Function used for logging. If set to “default” (the default), the evaluated design
points, the resulting performances, and the runtime will be reported. If set to
“memory” the memory usage for each evaluation will also be displayed, with
character(1) small increase in run time. Otherwise character(1) function
with arguments learner, resampling, measures, par.set, control, opt.path,
dob, x, y, remove.nas, stage and prev.stage is expected. The default dis-
plays the performance measures, the time needed for evaluating, the currently
used memory and the max memory ever used before (the latter two both taken
from gc). See the implementation for details.

final.dw.perc  (boolean)
If a Learner wrapped by a makeDownsampleWrapper is used, you can define
the value of dw.perc which is used to train the Learner with the final parameter
setting found by the tuning. Default is NULL which will not change anything.

budget  (integer(1))
Maximum budget for tuning. This value restricts the number of function evalu-
ations. It is passed to maxExperiments.

...  (any)
Further control parameters passed to the control arguments of cmaes::cma_es
or GenSA::GenSA, as well as towards the tunerConfig argument of irace::irace.

Value

(TuneControlIrace)

See Also

Other tune: TuneControl, getNestedTuneResultsOptPathDf(), getNestedTuneResultsX(),
getResamplingIndices(), getTuneResult(), makeModelMultiplexer(), makeModelMultiplexerParamSet(),
makeTuneControlMBO() \hspace{1cm} Create control object for hyperparameter tuning with MBO.

**Description**

Model-based / Bayesian optimization with the function `mlrMBO::mbo` from the `mlrMBO` package. Please refer to https://github.com/mlr-org/mlrMBO for further info.

**Usage**

```r
makeTuneControlMBO(
  same.resampling.instance = TRUE,
  impute.val = NULL,
  learner = NULL,
  mbo.control = NULL,
  tune.threshold = FALSE,
  tune.threshold.args = list(),
  continue = FALSE,
  log.fun = "default",
  final.dw.perc = NULL,
  budget = NULL,
  mbo.design = NULL
)
```

**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. Imputation is only active if `on.learner.error` is configured not to stop in `configureMlr`. 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.

- `learner` (Learner | NULL)
  - The surrogate learner: A regression learner to model performance landscape. For the default, NULL, `mlrMBO` will automatically create a suitable learner based on the rules described in `mlrMBO::makeMBOLearner`. 
makeTuneControlMBO

mbo.control (mlrMBO::MBOControl | NULL)
Control object for model-based optimization tuning. For the default, NULL, the control object will be created with all the defaults as described in mlrMBO::makeMBOControl.

tune.threshold (logical(1))
Should the threshold be tuned for the measure at hand, after each hyperparameter evaluation, via tuneThreshold? Only works for classification if the predict type is “prob”. Default is FALSE.

tune.threshold.args (list)
Further arguments for threshold tuning that are passed down to tuneThreshold. Default is none.

continue (logical(1))
Resume calculation from previous run using mlrMBO::mboContinue? Requires “save.file.path” to be set. Note that the ParamHelpers::OptPath in the mlrMBO::OptResult will only include the evaluations after the continuation. The complete OptPath will be found in the slot $mbo.result$opt.path.

log.fun (function | character(1))
Function used for logging. If set to “default” (the default), the evaluated design points, the resulting performances, and the runtime will be reported. If set to “memory” the memory usage for each evaluation will also be displayed, with character(1) small increase in run time. Otherwise character(1) function with arguments learner, resampling, measures, par.set, control, opt.path, dob, x, y, remove.nas, stage and prev.stage is expected. The default displays the performance measures, the time needed for evaluating, the currently used memory and the max memory ever used before (the latter two both taken from gc). See the implementation for details.

final.dw.perc (boolean)
If a Learner wrapped by a makeDownsampleWrapper is used, you can define the value of dw.perc which is used to train the Learner with the final parameter setting found by the tuning. Default is NULL which will not change anything.

budget (integer(1))
Maximum budget for tuning. This value restricts the number of function evaluations.

mbo.design (data.frame | NULL)
Initial design as data frame. If the parameters have corresponding trafo functions, the design must not be transformed before it is passed! For the default, NULL, a default design is created like described in mlrMBO::mbo.

Value

(TuneControlMBO)

References

makeTuneControlRandom

Create control object for hyperparameter tuning with random search.

Description

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

Usage

makeTuneControlRandom(
  same.resampling.instance = TRUE,
  maxit = NULL,
  tune.threshold = FALSE,
  tune.threshold.args = list(),
  log.fun = "default",
  final.dw.perc = NULL,
  budget = NULL
)

Arguments

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

maxit
  (integer(1) | NULL)
  Number of iterations for random search. Default is 100.

tune.threshold
  (logical(1))
  Should the threshold be tuned for the measure at hand, after each hyperparameter evaluation, via tuneThreshold? Only works for classification if the predict type is "prob". Default is FALSE.

tune.threshold.args
  (list)
  Further arguments for threshold tuning that are passed down to tuneThreshold. Default is none.

log.fun
  (function | character(1))
  Function used for logging. If set to "default" (the default), the evaluated design points, the resulting performances, and the runtime will be reported. If set to "memory" the memory usage for each evaluation will also be displayed, with character(1) small increase in run time. Otherwise character(1) function

See Also

TuneControl, getNestedTuneResultsOptPathDf(), getNestedTuneResultsX(), getResamplingIndices(), getTuneResult(), makeModelMultiplexer(), makeModelMultiplexerParamSet(), makeTuneControlCMAES(), makeTuneControlDesign(), makeTuneControlGenSA(), makeTuneControlGrid(), makeTuneControlIrace(), makeTuneControlRandom(), makeTuneWrapper(), tuneParams(), tuneThreshold()
**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

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

with arguments `learner`, `resampling`, `measures`, `par.set`, `control`, `opt.path`, `dob`, `x`, `y`, `remove.nas`, `stage` and `prev.stage` is expected. The default displays the performance measures, the time needed for evaluating, the currently used memory and the max memory ever used before (the latter two both taken from `gc`). See the implementation for details.

**Value**

- `final.dw.perc` (boolean)
  - If a Learner wrapped by a `makeDownsampleWrapper` is used, you can define the value of `dw.perc` which is used to train the Learner with the final parameter setting found by the tuning. Default is NULL which will not change anything.

- `budget` (integer(1))
  - Maximum budget for tuning. This value restricts the number of function evaluations. The budget equals the number of iterations (`maxit`) performed by the random search algorithm.

### See Also

Other tune: TuneControl, `getNestedTuneResultsOptPathDf()`, `getNestedTuneResultsX()`, `getResamplingIndices()`, `getTuneResult()`, `makeModelMultiplexer()`, `makeModelMultiplexerParamSet()`, `makeTuneControlCMAES()`, `makeTuneControlDesign()`, `makeTuneControlGenSA()`, `makeTuneControlGrid()`, `makeTuneControlIrace()`, `makeTuneControlMBO()`, `makeTuneWrapper()`, `tuneParams()`, `tuneThreshold()`
Arguments

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

resampling (ResamplingInstance | 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. Default is the
default measure for the task, see here `getDefaultMeasure`.

par.set (ParamHelpers::ParamSet)
Collection of parameters and their constraints for optimization. Dependent pa-
rameters with a requires field must use quote and not expression to define it.

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: `TuneControl`, `getNestedTuneResultsOptPathDf()`, `getNestedTuneResultsX()`,
`getResamplingIndices()`, `getTuneResult()`, `makeModelMultiplexer()`, `makeModelMultiplexerParamSet()`,
`makeTuneControlCMAES()`, `makeTuneControlDesign()`, `makeTuneControlGenSA()`, `makeTuneControlGrid()`,
`makeTuneControlIrace()`, `makeTuneControlLBO()`, `makeTuneControlRandom()`, `tuneParams()`,
`tuneThreshold()`

Other wrapper: `makeBaggingWrapper()`, `makeClassificationViaRegressionWrapper()`, `makeConstantClassWrapper()`,
`makeCostSensClassifWrapper()`, `makeCostSensRegrWrapper()`, `makeDownsampleWrapper()`,
`makeDummyFeaturesWrapper()`, `makeExtractFDAFeatsWrapper()`, `makeFeatSelWrapper()`, `makeFilterWrapper()`,
`makeImputeWrapper()`, `makeMulticlassWrapper()`, `makeMultilabelBinaryRelevanceWrapper()`,
`makeMultilabelClassifierChainsWrapper()`, `makeMultilabelDBRWrapper()`, `makeMultilabelNestedStackingWrapper()`,
`makeMultilabelStackingWrapper()`, `makeOverBaggingWrapper()`, `makePreprocWrapper()`, `makePreprocWrapperCaret()`,
`makeRemoveConstantFeaturesWrapper()`, `makeSMOTEWrapper()`, `makeUndersampleWrapper()`,
`makeWeightedClassesWrapper()`

Examples

```
task = makeClassifTask(data = iris, target = "Species")
learner = makeLearner("classif.rpart")
```
# stupid mini grid
ps = makeParamSet(
    makeDiscreteParam("cp", values = c(0.05, 0.1)),
    makeDiscreteParam("minsplit", values = c(10, 20))
)
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)
getNestedTuneResultsOptPathDf(r)
getNestedTuneResultsX(r)

---

makeUndersampleWrapper

Fuse learner with simple over/undersampling 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, usw.cl = NULL)
makeOversampleWrapper(learner, osw.rate = 1, osw.cl = NULL)

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 a 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.
makeWeightedClassesWrapper

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 hasLearnerProperties(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.

usw.cl (character(1))
Class that should be undersampled. Default is NULL, which means the larger one.

osw.rate (numeric(1))
Factor to oversample a class. Must be between 1 and Inf, where 1 means no oversampling and 2 would mean doubling the class size. Default is 1.

osw.cl (character(1))
Class that should be oversampled. Default is NULL, which means the smaller one.

Value

Learner.

See Also

Other imbalancy: makeOverBaggingWrapper(), oversample(), smote()

Other wrapper: makeBaggingWrapper(), makeClassificationViaRegressionWrapper(), makeConstantClassWrapper(), makeCostSensClassifWrapper(), makeCostSensRegrWrapper(), makeDownsampleWrapper(), makeDummyFeaturesWrapper(), makeExtractFDAFeatsWrapper(), makeFeatSelWrapper(), makeFilterWrapper(), makeImputeWrapper(), makeMulticlassWrapper(), makeMultilabelBinaryRelevanceWrapper(),
makeMultilabelClassifierChainsWrapper(), makeMultilabelDBRWrapper(), makeMultilabelNestedStackingWrapper(), makeMultilabelStackingWrapper(), makeOverBaggingWrapper(), makePreprocWrapper(), makePreprocWrapperCaret(), makeRemoveConstantFeaturesWrapper(), makeSMOTEWrapper(), makeTuneWrapper(), makeWeightedClassesWrapper()
makeWeightedClassesWrapper

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 parameter, which allows class weighting. The default (wcw.param = NULL) will use the parameter defined in the learner (class.weights.param). During training, the parameter must accept a named vector of class weights, where length equals the number of classes.

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 getClassDesc(task)$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.

See Also

Other wrapper: makeBaggingWrapper(), makeClassificationViaRegressionWrapper(), makeConstantClassWrapper(), makeCostSensClassifWrapper(), makeCostSensRegrWrapper(), makeDownsampleWrapper(), makeDummyFeaturesWrapper(), makeExtractFDAFeatsWrapper(), makeFeatSelWrapper(), makeFilterWrapper(), makeImputeWrapper(), makeMulticlassWrapper(), makeMulticlassBinaryRelevanceWrapper(), makeMulticlassClassifierChainsWrapper(), makeMultilabelDBRWrapper(), makeMultilabelNestedStackingWrapper(), makeMultilabelStackingWrapper(), makeOverBaggingWrapper(), makePreprocWrapper(), makePreprocWrapperCaret(), makeRemoveConstantFeaturesWrapper(), makeSMOTEWrapper(), makeTuneWrapper(), makeUndersampleWrapper()

Examples

set.seed(123)
# using the direct parameter of the SVM (which is already defined in the learner)
lrn = makeWeightedClassesWrapper("classif.ksvm", wcw.weight = 0.01)
res = holdout(lrn, sonar.task)
print(calculateConfusionMatrix(res$pred))

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

# tuning the imbalance 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**

```r
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.
MeasureProperties

subset (integer | logical | NULL)
Selected cases. Either a logical or an index vector. By default NULL if all observations are used.

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.

MeasureProperties

Query properties of measures.

Description

Properties can be accessed with getMeasureProperties(measure), which returns a character vector.

The measure properties are defined in Measure.

Usage

getMeasureProperties(measure)

hasMeasureProperties(measure, props)

Arguments

measure (Measure)
Performance measure. Default is the first measure used in the benchmark experiment.

props (character)
Vector of properties to query.

Value

getMeasureProperties returns a character vector with measure properties. hasMeasureProperties returns a logical vector of the same length as props.
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.

All supported measures can be found by listMeasures or as a table in the tutorial appendix: https://mlr.mlr-org.com/articles/tutorial/measures.html.

If you want a measure for a misclassification cost matrix, look at makeCostMeasure. If you want to implement your own measure, look at makeMeasure.

Most measures can directly be accessed via the function named after the scheme measureX (e.g. measureSSE).

For clustering measures, we compact the predicted cluster IDs such that they form a continuous series starting with 1. If this is not the case, some of the measures will generate warnings.

Some measures have parameters. Their defaults are set in the constructor makeMeasure and can be overwritten using setMeasurePars.

Usage

measureSSE(truth, response)
measureMSE(truth, response)
measureRMSE(truth, response)
measureMEDSE(truth, response)
measureSAE(truth, response)
measureMAE(truth, response)
measureMEDAE(truth, response)
measureRSQ(truth, response)
measureEXPVAR(truth, response)
measureRRSE(truth, response)
measureRAE(truth, response)
measureMAPE(truth, response)
measures

    measureMSLE(truth, response)
    measureRMSLE(truth, response)
    measureKendallTau(truth, response)
    measureSpearmanRho(truth, response)
    measureMMCE(truth, response)
    measureACC(truth, response)
    measureBER(truth, response)
    measureAUNU(probabilities, truth)
    measureAUNP(probabilities, truth)
    measureAU1U(probabilities, truth)
    measureAU1P(probabilities, truth)
    measureMulticlassBrier(probabilities, truth)
    measureLogloss(probabilities, truth)
    measureSSR(probabilities, truth)
    measureQSR(probabilities, truth)
    measureLSR(probabilities, truth)
    measureKAPPA(truth, response)
    measureWKAPPA(truth, response)
    measureAUC(probabilities, truth, negative, positive)
    measureBrier(probabilities, truth, negative, positive)
    measureBrierScaled(probabilities, truth, negative, positive)
    measureBAC(truth, response)
    measureTP(truth, response, positive)
    measureTN(truth, response, negative)
measureFP(truth, response, positive)
measureFN(truth, response, negative)
measureTPR(truth, response, positive)
measureTNR(truth, response, negative)
measureFPR(truth, response, negative, positive)
measureFNR(truth, response, negative, positive)
measurePPV(truth, response, positive, probabilities = NULL)
measureNPV(truth, response, negative)
measureFDR(truth, response, negative)
measureMCC(truth, response, negative, positive)
measureF1(truth, response, positive)
measureGMEAN(truth, response, negative, positive)
measureGPR(truth, response, positive)
measureMultilabelHamloss(truth, response)
measureMultilabelSubset01(truth, response)
measureMultilabelF1(truth, response)
measureMultilabelACC(truth, response)
measureMultilabelPPV(truth, response)
measureMultilabelTPR(truth, response)

Arguments

**truth** (factor)
Vector of the true class.

**response** (factor)
Vector of the predicted class.

**probabilities** (numeric | matrix)
  a) For purely binary classification measures: The predicted probabilities for the positive class as a numeric vector. b) For multiclass classification measures: The predicted probabilities for all classes, always as a numeric matrix, where
mergeBenchmarkResults

columns are named with class labels.

negative (character(1))
The name of the negative class.

positive (character(1))
The name of the positive class.

References


See Also

Other performance: `ConfusionMatrix`, `calculateConfusionMatrix()`, `calculateROCMeasures()`, `estimateRelativeOverfitting()`, `makeCostMeasure()`, `makeCustomResampledMeasure()`, `makeMeasure()`, `performance()`, `setAggregation()`, `setMeasurePars()`

mergeBenchmarkResults  Merge different BenchmarkResult objects.

Description

The function automatically combines a list of `BenchmarkResult` objects into a single `BenchmarkResult` object as long as the full crossproduct of all task-learner combinations are available.

Usage

`mergeBenchmarkResults(bmr)`

Arguments

`bmr` (list of `BenchmarkResult`)
BenchmarkResult objects that should be merged.

Details

Note that if you want to merge several `BenchmarkResult` objects, you must ensure that all possible learner and task combinations will be contained in the returned object. Otherwise, the user will be notified which task-learner combinations are missing or duplicated.

When merging `BenchmarkResult` objects with different measures, all missing measures will automatically be recomputed.
mergeSmallFactorLevels

*Merges small levels of factors into new level.*

**Description**

Merges factor levels that occur only infrequently into combined levels with a higher frequency.

**Usage**

```r
mergeSmallFactorLevels(
  task,
  cols = NULL,
  min.perc = 0.01,
  new.level = ".merged"
)
```

**Arguments**

- **task** *(Task)*
  The task.

- **cols** *(character)*
  Which columns to convert. Default is all factor and character columns.

- **min.perc** *(numeric(1))*
  The smallest levels of a factor are merged until their combined proportion w.r.t. the length of the factor exceeds `min.perc`. Must be between 0 and 1. Default is 0.01.

- **new.level** *(character(1))*
  New name of merged level. Default is ".merged"

**Value**

Task, where merged levels are combined into a new level of name `new.level`.

**See Also**

Other `eda_and_preprocess`: `capLargeValues()`, `createDummyFeatures()`, `dropFeatures()`, `normalizeFeatures()`, `removeConstantFeatures()`, `summarizeColumns()`, `summarizeLevels()`
### Description

List of all mlr documentation families with members.

### Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>benchmark</td>
<td>batchmark, reduceBatchmarkResults, benchmark, benchmarkParallel, getBMRTaskIds, getBMR Learners, getBMR Learner Ids, getBMR Learner Short Names, getBMR Measures, getBMR Measure Ids, getBMR Predictions, getBMR Performances, getBMR Aggr Performances, getBMR Tune Results, getBMR Filtered Features, getBMR Models, getBMR Task Descs, convertBMRToRankMatrix, friedmanPostHocTestBMR, friedmanTestBMR, plotBMR Boxplots, plotBMR Ranks As Bar Chart, generateCritDifferencesData, plotCritDifferences</td>
</tr>
<tr>
<td>calibration</td>
<td>generateCalibrationData, plotCalibration</td>
</tr>
<tr>
<td>configure</td>
<td>configureMlr, getMlrOptions</td>
</tr>
<tr>
<td>costsens</td>
<td>makeCostSensTask, makeCostSensWeightedPairsWrapper</td>
</tr>
<tr>
<td>debug</td>
<td>predictFailureModel, getPredictionDump, getRRDump, print.ResampleResult</td>
</tr>
<tr>
<td>downsample</td>
<td>downsample</td>
</tr>
<tr>
<td>eda_and_preprocess</td>
<td>capLargeValues, createDummyFeatures, dropFeatures, mergeSmallFactorLevels, normalizeFeatures, removeConstantFeatures, summarizeColumns, summarizeLevels</td>
</tr>
<tr>
<td>extractFDAFeatures</td>
<td>reextractFDAFeatures</td>
</tr>
<tr>
<td>fda_featextractor</td>
<td>extractFDAFourier, extractFDAWavelets, extractFDA FPCA, extractFDA MultiResFeatures</td>
</tr>
<tr>
<td>fda</td>
<td>makeExtractFDAFeatMethod, extractFDAFeatures</td>
</tr>
<tr>
<td>featsel</td>
<td>analyzeFeatSelResult, makeFeatSelControl, getFeatSelResult, selectFeatures</td>
</tr>
<tr>
<td>filter</td>
<td>filterFeatures, makeFilter, listFilterMethods, getFilteredFeatures, generateFilterValuesData, getFilterValues</td>
</tr>
<tr>
<td>generate_plot_data</td>
<td>generateFeatureImportanceData, plotFilterValues, generatePartialDependenceData</td>
</tr>
<tr>
<td>help</td>
<td>helpLearner, helpLearnerParam</td>
</tr>
<tr>
<td>imbalancy</td>
<td>oversample, smote</td>
</tr>
<tr>
<td>impute</td>
<td>makeImputeMethod, imputeConstant, impute, reimpute</td>
</tr>
</tbody>
</table>
mtcars.task

Motor Trend Car Road Tests clustering task.

Description
Contains the task (mtcars.task).
normalizeFeatures

References
See datasets::mtcars.

normalizeFeatures  Normalize features.

Description
Normalize features by different methods. Internally BBmisc::normalize is used for every feature column. Non numerical features will be left untouched and passed to the result. For constant features most methods fail, special behaviour for this case is implemented.

Usage
normalizeFeatures(
  obj,
  target = character(0L),
  method = "standardize",
  cols = NULL,
  range = c(0, 1),
  on.constant = "quiet"
)

Arguments
obj (data.frame | Task) Input data.
target (character(1) | character(2) | character(n.classes)) Name(s) of the target variable(s). Only used when obj is a data.frame, otherwise ignored. If survival analysis is applicable, these are the names of the survival time and event columns, so it has length 2. For multilabel classification these are the names of logical columns that indicate whether a class label is present and the number of target variables corresponds to the number of classes.
method (character(1)) Normalizing method. Available are: "center": Subtract mean.
"scale": Divide by standard deviation.
"standardize": Center and scale.
"range": Scale to a given range.
cols (character) Columns to normalize. Default is to use all numeric columns.
range (numeric(2)) Range for method "range". Default is c(0,1).
How should constant vectors be treated? Only used, of “method != center”, since this method does not fail for constant vectors. Possible actions are: “quiet”: Depending on the method, treat them quietly: “scale”: No division by standard deviation is done, input values will be returned untouched. “standardize”: Only the mean is subtracted, no division is done. “range”: All values are mapped to the mean of the given range. “warn”: Same behaviour as “quiet”, but print a warning message. “stop”: Stop with an error.

Value

data.frame | Task. Same type as obj.

See Also

BBmisc::normalize
Other eda_and_preprocess: capLargeValues(), createDummyFeatures(), dropFeatures(), mergeSmallFactorLevels(), removeConstantFeatures(), summarizeColumns(), summarizeLevels()

oversample(task, rate, cl = NULL)

Usage

oversample(task, rate, cl = NULL)
undersample(task, rate, cl = NULL)

Arguments

task (Task)
The task.
rate (numeric(1))
Factor to upsample or downsample a 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
parallelization

be between 1 and Inf, where 1 means no oversampling and 2 would mean doubling the class size.

c1 (character(1))
Which class should be over- or undersampled. If NULL, oversample will select the smaller and undersample the larger class.

Value
Task.

See Also
Other imbalancy: makeOverBaggingWrapper(), makeUndersampleWrapper(), smote()

---

parallelization Supported parallelization methods

Description

mlr supports different methods to activate parallel computing capabilities through the integration of the parallelMap::parallelMap package, which supports all major parallelization backends for R. You can start parallelization with parallelStart*, where * should be replaced with the chosen backend. parallelMap::parallelStop is used to stop all parallelization backends.

Parallelization is divided into different levels and will automatically be carried out for the first level that occurs, e.g. if you call resample() after parallelMap::parallelStart, each resampling iteration is a parallel job and possible underlying calls like parameter tuning won’t be parallelized further.

The supported levels of parallelization are:

"mlr.resample" Each resampling iteration (a train/test step) is a parallel job.
"mlr.benchmark" Each experiment "run this learner on this data set" is a parallel job.
"mlr.tuneParams" Each evaluation in hyperparameter space "resample with these parameter settings" is a parallel job. How many of these can be run independently in parallel depends on the tuning algorithm. For grid search or random search there is no limit, but for other tuners it depends on how many points to evaluate are produced in each iteration of the optimization. If a tuner works in a purely sequential fashion, we cannot work magic and the hyperparameter evaluation will also run sequentially. But note that you can still parallelize the underlying resampling.
"mlr.selectFeatures" Each evaluation in feature space "resample with this feature subset" is a parallel job. The same comments as for "mlr.tuneParams" apply here.
"mlr.ensemble" For all ensemble methods, the training and prediction of each individual learner is a parallel job. Supported ensemble methods are the makeBaggingWrapper, makeCostSensRegrWrapper, makeMulticlassWrapper, makeMultilabelBinaryRelevanceWrapper and the makeOverBaggingWrapper.
Measure performance of prediction.

Description

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

Usage

performance(
  pred,  # Prediction object.
  measures,  # Performance measure(s) to evaluate. Default is the default measure for the task, see here getDefaultMeasure.
  task = NULL,  # Learning task, might be requested by performance measure, usually not needed except for clustering or survival.
  model = NULL,  # Model built on training data, might be requested by performance measure, usually not needed except for survival.
  feats = NULL,  # Features of predicted data, usually not needed except for clustering. If the prediction was generated from a task, you can also pass this instead and the features are extracted from it.
  simpleaggr = FALSE  # If TRUE, aggregation of ResamplePrediction objects is skipped. This is used internally for threshold tuning. Default is FALSE.
)

Arguments

- **pred**: (Prediction) Prediction object.
- **measures**: (Measure | list of Measure) Performance measure(s) to evaluate. Default is the default measure for the task, see here getDefaultMeasure.
- **task**: (Task) Learning task, might be requested by performance measure, usually not needed except for clustering or survival.
- **model**: (WrappedModel) Model built on training data, might be requested by performance measure, usually not needed except for survival.
- **feats**: (data.frame) Features of predicted data, usually not needed except for clustering. If the prediction was generated from a task, you can also pass this instead and the features are extracted from it.
- **simpleaggr**: (logical) If TRUE, aggregation of ResamplePrediction objects is skipped. This is used internally for threshold tuning. Default is FALSE.

Value

(named numeric). Performance value(s), named by measure(s).
phoneme.task

See Also

Other performance: ConfusionMatrix, calculateConfusionMatrix(), calculateROCMeasures(), estimateRelativeOverfitting(), makeCostMeasure(), makeCustomResampledMeasure(), makeMeasure(), measures, setAggregation(), setMeasurePars()

Examples

```r
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)
```

---

**phoneme.task**  
Phoneme functional data multilabel classification task.

**Description**

Contains the task (phoneme.task). The task contains a single functional covariate and 5 equally big classes (aa, ao, dcl, iy, sh). The aim is to predict the class of the phoneme in the functional. The dataset is contained in the package fda.usc.

**References**


---

**pid.task**  
PimaIndiansDiabetes classification task.

**Description**

Contains the task (pid.task).

**References**

See mlbench::PimaIndiansDiabetes. Note that this is the uncorrected version from mlbench.
plotBMRBoxplots

Create box or violin plots for a BenchmarkResult.

Description

Plots box or violin plots for a selected measure across all iterations of the resampling strategy, faceted by the task.id.

Usage

plotBMRBoxplots(
  bmr,
  measure = NULL,
  style = "box",
  order.lrns = NULL,
  order.tsks = NULL,
  pretty.names = TRUE,
  facet.wrap.nrow = NULL,
  facet.wrap.ncol = NULL
)

Arguments

bmr  (BenchmarkResult) Benchmark result.
measure  (Measure) Performance measure. Default is the first measure used in the benchmark experiment.
style  (character(1)) Type of plot, can be “box” for a boxplot or “violin” for a violin plot. Default is “box”.
order.lrns  (character(n.learners)) Character vector with learner.ids in new order.
order.tsks  (character(n.tasks)) Character vector with task.ids in new order.
pretty.names  (logical(1)) Whether to use the Measure name and the Learner short name instead of the id. Default is TRUE.
facet.wrap.nrow, facet.wrap.ncol  (integer) Number of rows and columns for facetting. Default for both is NULL. In this case ggplot’s facet_wrap will choose the layout itself.

Value

ggplot2 plot object.
See Also

Other plot: createSpatialResamplingPlots(), plotBMRRanksAsBarChart(), plotBMRSummary(), plotCalibration(), plotCritDifferences(), plotLearningCurve(), plotPartialDependence(), plotROCCurves(), plotResiduals(), plotThreshVsPerf()

Other benchmark: BenchmarkResult, batchmark(), benchmark(), convertBMRToRankMatrix(), friedmanPostHocTestBMR(), friedmanTestBMR(), generateCritDifferencesData(), getBMRAggrPerformances(), getBMRFeatSelResults(), getBMRFilteredFeatures(), getBMRLearnerIds(), getBMRLearnerShortNames(), getBMRLearners(), getBMRMetricIds(), getBMRMetricNames(), getBMRModels(), getBMRPerformances(), getBMRPredictions(), getBMRTaskDescs(), getBMRTaskIds(), getBMRTuneResults(), plotBMRRanksAsBarChart(), plotBMRSummary(), plotCritDifferences(), reduceBatchmarkResults()

Examples

# see benchmark

plotBMRRanksAsBarChart

Create a bar chart for ranks in a BenchmarkResult.

Description

Plots a bar chart from the ranks of algorithms. Alternatively, tiles can be plotted for every rank-task combination, see pos for details. In all plot variants the ranks of the learning algorithms are displayed on the x-axis. Areas are always colored according to the learner.id.

Usage

plotBMRRanksAsBarChart(bmr, measure = NULL, ties.method = "average", aggregation = "default", pos = "stack", order.lrns = NULL, order.tsks = NULL, pretty.names = TRUE)

Arguments

bmr  (BenchmarkResult)
Benchmark result.

measure  (Measure)
Performance measure. Default is the first measure used in the benchmark experiment.
plotBMRSummary

**ties.method** (character(1))
See `rank` for details.

**aggregation** (character(1))
“mean” or “default”. See `getBMRAggrPerformances` for details on “default”.

**pos** (character(1))
Optionally set how the bars are positioned in ggplot2. Ranks are plotted on the x-axis. “tile” plots a heat map with task as the y-axis. Allows identification of the performance in a special task. “stack” plots a stacked bar plot. Allows for comparison of learners within and across ranks. “dodge” plots a bar plot with bars next to each other instead of stacked bars.

**order.lrns** (character(n.learners))
Character vector with learner.ids in new order.

**order.tsks** (character(n.tasks))
Character vector with task.ids in new order.

**pretty.names** (logical(1))
Whether to use the short name of the learner instead of its ID in labels. Defaults to TRUE.

**Value**

ggplot2 plot object.

**See Also**

Other plot: `createSpatialResamplingPlots()`, `plotBMRBoxplots()`, `plotBMRSummary()`, `plotCalibration()`, `plotCritDifferences()`, `plotLearningCurve()`, `plotPartialDependence()`, `plotROCCurves()`, `plotResiduals()`, `plotThreshVsPerf()`

Other benchmark: `BenchmarkResult()`, `batchmark()`, `convertBMRToRankMatrix()`, `friedmanPostHocTestBMR()`, `friedmanTestBMR()`, `generateCritDifferencesData()`, `getBMRAggrPerformances()`, `getBMRFeatSelResults()`, `getBMRFilteredFeatures()`, `getBMRLearnerIds()`, `getBMRLearnerShortNames()`, `getBMRlearners()`, `getBMRMeasureIds()`, `getBMRMeasures()`, `getBMRModels()`, `getBMRPerformances()`, `getBMRpredictions()`, `getBMRTaskDescs()`, `getBMRTaskIds()`, `getBMRTuneResults()`, `plotBMRBoxplots()`, `plotBMRSummary()`, `plotCritDifferences()`, `reduceBatchmarkResults()`

**Examples**

# see benchmark

---

**Description**

Creates a scatter plot, where each line refers to a task. On that line the aggregated scores for all learners are plotted, for that task. Optionally, you can apply a rank transformation or just use one of ggplot2’s transformations like `ggplot2::scale_x_log10`. 
plotBMRSummary

Usage

plotBMRSummary(
  bmr,
  measure = NULL,
  trafo = "none",
  order.tsks = NULL,
  pointsize = 4L,
  jitter = 0.05,
  pretty.names = TRUE
)

Arguments

bmr (BenchmarkResult)
Benchmark result.
measure (Measure)
Performance measure. Default is the first measure used in the benchmark experiment.
trafo (character(1))
Currently either “none” or “rank”, the latter performing a rank transformation (with average handling of ties) of the scores per task. NB: You can add always add ggplot2::scale_x_log10 to the result to put scores on a log scale. Default is “none”.
order.tsks (character(n.tasks))
Character vector with task.ids in new order.
pointsize (numeric(1))
Point size for ggplot2 ggplot2::geom_point for data points. Default is 4.
jitter (numeric(1))
Small vertical jitter to deal with overplotting in case of equal scores. Default is 0.05.
pretty.names (logical(1))
Whether to use the short name of the learner instead of its ID in labels. Defaults to TRUE.

Value

ggplot2 plot object.

See Also

Other benchmark: BenchmarkResult, batchmark(), benchmark(), convertBMRToRankMatrix(),
friedmanPostHocTestBMR(), friedmanTestBMR(), generateCritDifferencesData(), getBMRAggrPerformances(),
getBMRFeatSelResults(), getBMRFilteredFeatures(), getBMR LearnerIds(), getBMR LearnerShortNames(),
getBMR Learners(), getBMRMeasureIds(), getBMR Measures(), getBM RModes(), getBMR Performances(),
getBMR Predictions(), getBMRTaskDescs(), getBMRTaskIds(), getBMRTuneResults(), plotBM RBoxplots(),
plotBMRRanksAsBarChart(), plotCritDifferences(), reduceBatchmarkResults()
plotCalibration

Other plot: createSpatialResamplingPlots(), plotBMRBoxplots(), plotBMRRanksAsBarChart(), plotCalibration(), plotCritDifferences(), plotLearningCurve(), plotPartialDependence(), plotROCcurves(), plotResiduals(), plotThreshVsPerf()

Examples

# see benchmark

plotCalibration  

Plot calibration data using ggplot2.

Description

Plots calibration data from generateCalibrationData.

Usage

plotCalibration(
  obj,  
  smooth = FALSE,  
  reference = TRUE,  
  rag = TRUE,  
  facet.wrap.nrow = NULL,  
  facet.wrap.ncol = NULL
)

Arguments

obj  
Result of generateCalibrationData.

smooth  
Whether to use a loess smoother. Default is FALSE.

reference  
Whether to plot a reference line showing perfect calibration. Default is TRUE.

rag  
Whether to include a rag plot which shows a rug plot on the top which pertains to positive cases and on the bottom which pertains to negative cases. Default is TRUE.

facet.wrap.nrow, facet.wrap.ncol

Number of rows and columns for facetting. Default for both is NULL. In this case ggplot's facet_wrap will choose the layout itself.

Value

ggplot2 plot object.
plotCritDifferences

Plot critical differences for a selected measure.

Description

Plots a critical-differences diagram for all classifiers and a selected measure. If a baseline is selected for the Bonferroni-Dunn test, the critical difference interval will be positioned around the baseline. If not, the best performing algorithm will be chosen as baseline.

The positioning of some descriptive elements can be moved by modifying the generated data.

Usage

plotCritDifferences(obj, baseline = NULL, pretty.names = TRUE)

Arguments

obj (critDifferencesData) Result of generateCritDifferencesData().
baseline (character(1)): (learner.id)
Overwrites baseline from `generateCritDifferencesData()`!
Select a learner.id as baseline for the critical difference diagram, the critical
difference will be positioned around this learner. Defaults to best performing
algorithm.
pretty.names (logical(1))
Whether to use the short name of the learner instead of its ID in labels. Defaults
to TRUE.

Value

ggplot2 plot object.

References

Janez Demsar, Statistical Comparisons of Classifiers over Multiple Data Sets, JMLR, 2006

See Also

Other plot: `createSpatialResamplingPlots()`, `plotBMRRanksAsBarChart()`,
`plotBMRSummary()`, `plotLearningCurve()`, `plotPartialDependence()`,
`plotROCCurves()`, `plotResiduals()`, `plotThreshVsPerf()`

Other benchmark: `BenchmarkResult`, `benchmark()`, `convertBMRToRankMatrix()`,
`friedmanPostHocTestBMR()`, `friedmanTestBMR()`, `generateCritDifferencesData()`, `getBMRAggrPerformances()`,
`getBMRFeatSelResults()`, `getBMRFilteredFeatures()`, `getBMRLearnerIds()`, `getBMRLearnerShortNames()`,
`getBMRLearners()`, `getBMRMeasureIds()`, `getBMRMeasures()`, `getBMRModels()`, `getBMRPerformances()`,
`getBMRPredictions()`, `getBMRTaskDescs()`, `getBMRTaskIds()`, `getBMRTuneResults()`, `plotBMRRanksAsBarChart()`,
`plotBMRSummary()`, `reduceBatchmarkResults()`

Examples

```r
# see benchmark

plotFilterValues() Plot filter values using ggplot2.

Description

Plot filter values using ggplot2.

Usage

plotFilterValues(
  fvalues,
  sort = "dec",
  n.show = nrow(fvalues$data),
  filter = NULL,
  feat.type.cols = FALSE
)
```
plotHyperParsEffect  

Arguments

- **fvalues** 
  - **(FilterValues)**
  - Filter values.

- **sort** 
  - **(character(1))**
  - Available options are:
    - "dec" -> descending
    - "inc" -> increasing
    - "none" -> no sorting
  - Default is decreasing.

- **n.show** 
  - **(integer(1))**
  - Number of features (maximal) to show. Default is to plot all features.

- **filter** 
  - **(character(1))** 
  - In case `fvalues` contains multiple filter methods, which method should be plotted?

- **feat.type.cols** 
  - **(logical(1))**
  - Whether to color different feature types (e.g. numeric | factor). Default is to use no colors (`feat.type.cols = FALSE`).

Value

- ggplot2 plot object.

See Also

Other filter: `filterFeatures()`, `generateFilterValuesData()`, `getFilteredFeatures()`, `listFilterEnsembleMethods()`, `makeFilter()`, `makeFilterEnsemble()`, `makeFilterWrapper()`

Other generate_plot_data: `generateCalibrationData()`, `generateCritDifferencesData()`, `generateFeatureImportanceData()`, `generateFilterValuesData()`, `generateLearningCurveData()`, `generatePartialDependenceData()`, `generateThreshVsPerfData()`

Examples

```r
fv = generateFilterValuesData(iris.task, method = "variance")
plotFilterValues(fv)
```

Description

Plot hyperparameter validation path. Automated plotting method for `HyperParsEffectData` object. Useful for determining the importance or effect of a particular hyperparameter on some performance measure and/or optimizer.
Usage

plotHyperParsEffect(
    hyperpars.effect.data,
    x = NULL,
    y = NULL,
    z = NULL,
    plot.type = "scatter",
    loess.smooth = FALSE,
    facet = NULL,
    global.only = TRUE,
    interpolate = NULL,
    show.experiments = FALSE,
    show.interpolated = FALSE,
    nested.agg = mean,
    partial.dep.learn = NULL
)

Arguments

hyperpars.effect.data
  (HyperParsEffectData)
  Result of \code{generateHyperParsEffectData}

x
  (character(1))
  Specify what should be plotted on the x axis. Must be a column from \code{HyperParsEffectData$data}. For partial dependence, this is assumed to be a hyperparameter.

y
  (character(1))
  Specify what should be plotted on the y axis. Must be a column from \code{HyperParsEffectData$data}

z
  (character(1))
  Specify what should be used as the extra axis for a particular geom. This could be for the fill on a heatmap or color aesthetic for a line. Must be a column from \code{HyperParsEffectData$data}. Default is \code{NULL}.

plot.type
  (character(1))
  Specify the type of plot: “scatter” for a scatterplot, “heatmap” for a heatmap, “line” for a scatterplot with a connecting line, or “contour” for a contour plot layered ontop of a heatmap. Default is “scatter”.

loess.smooth
  (logical(1))
  If TRUE, will add loess smoothing line to plots where possible. Note that this is probably only useful when \code{plot.type} is set to either “scatter” or “line”. Must be a column from \code{HyperParsEffectData$data}. Not used with partial dependence. Default is \code{FALSE}.

facet
  (character(1))
  Specify what should be used as the facet axis for a particular geom. When using nested cross validation, set this to “nested_cv_run” to obtain a facet for each outer loop. Must be a column from \code{HyperParsEffectData$data}. Please note that facetting is not supported with partial dependence plots! Default is \code{NULL}.

global.only
  (logical(1))
  If TRUE, will only plot the current global optima when setting x = "iteration" and
y as a performance measure from `HyperParsEffectData$measures`. Set this to FALSE to always plot the performance of every iteration, even if it is not an improvement. Not used with partial dependence. Default is TRUE.

`interpolate` (Learner | character(1))
If not NULL, will interpolate non-complete grids in order to visualize a more complete path. Only meaningful when attempting to plot a heatmap or contour. This will fill in “empty” cells in the heatmap or contour plot. Note that cases of irregular hyperparameter paths, you will most likely need to use this to have a meaningful visualization. Accepts either a regression Learner object or the learner as a string for interpolation. This cannot be used with partial dependence. Default is NULL.

`show.experiments` (logical(1))
If TRUE, will overlay the plot with points indicating where an experiment ran. This is only useful when creating a heatmap or contour plot with interpolation so that you can see which points were actually on the original path. Note: if any learner crashes occurred within the path, this will become TRUE. Not used with partial dependence. Default is FALSE.

`show.interpolated` (logical(1))
If TRUE, will overlay the plot with points indicating where interpolation ran. This is only useful when creating a heatmap or contour plot with interpolation so that you can see which points were interpolated. Not used with partial dependence. Default is FALSE.

`nested.agg` (function)
The function used to aggregate nested cross validation runs when plotting 2 hyperparameters. This is also used for nested aggregation in partial dependence. Default is mean.

`partial.dep.learn` (Learner | character(1))
The regression learner used to learn partial dependence. Must be specified if “partial.dep” is set to TRUE in `generateHyperParsEffectData`. Accepts either a Learner object or the learner as a string for learning partial dependence. Default is NULL.

Value
ggplot2 plot object.

Note
Any NAs incurred from learning algorithm crashes will be indicated in the plot (except in the case of partial dependence) and the NA values will be replaced with the column min/max depending on the optimal values for the respective measure. Execution time will be replaced with the max. Interpolation by its nature will result in predicted values for the performance measure. Use interpolation with caution. If “partial.dep” is set to TRUE in `generateHyperParsEffectData`, only partial dependence will be plotted.

Since a ggplot2 plot object is returned, the user can change the axis labels and other aspects of the plot using the appropriate ggplot2 syntax.
plotLearnerPrediction

Examples

# see generateHyperParsEffectData

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 ggplot2::ggplot. Good for teaching or exploring models.

For classification and clustering, 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 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 = "white",
  err.size = pointsize,
  greyscale = FALSE,
  pretty.names = TRUE
)

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. Default is the default measure for the task, see here `getDefalutMeasure`.

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 `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 “white”

err.size (integer(1))
For classification: Size of misclassified data points. Default is pointsize.

greyscale (logical(1))
Should the plot be greyscale completely? Default is FALSE.

pretty.names (logical(1))
Whether to use the short name of the learner instead of its ID in labels. Defaults to TRUE.

Value

The ggplot2 object.
plotLearningCurve

Plot learning curve data using ggplot2.

Description

Visualizes data size (percentage used for model) vs. performance measure(s).

Usage

plotLearningCurve(
  obj,
  facet = "measure",
  pretty.names = TRUE,
  facet.wrap.nrow = NULL,
  facet.wrap.ncol = NULL
)

Arguments

obj

(LearningCurveData)
Result of generateLearningCurveData, with class LearningCurveData.

facet

(character(1))
Selects "measure" or "learner" to be the facetting variable. The variable mapped to
facet must have more than one unique value, otherwise it will be ignored.
The variable not chosen is mapped to color if it has more than one unique value.
The default is "measure".

pretty.names

(logical(1))
Whether to use the Measure name instead of the id in the plot. Default is TRUE.

facet.wrap.nrow, facet.wrap.ncol

(integer)
Number of rows and columns for facetting. Default for both is NULL. In this case
ggplot's facet_wrap will choose the layout itself.

Value

ggplot2 plot object.

See Also

Other learning_curve: generateLearningCurveData()

Other plot: createSpatialResamplingPlots(), plotBMRBoxplots(), plotBMRRanksAsBarChart(),
plotBMRSummary(), plotCalibration(), plotCritDifferences(), plotPartialDependence(),
plotROCCurves(), plotResiduals(), plotThreshVsPerf()
plotPartialDependence  
Plot a partial dependence with ggplot2.

Description

Plot a partial dependence from generatePartialDependenceData using ggplot2.

Usage

plotPartialDependence(
  obj,
  geom = "line",
  facet = NULL,
  facet.wrap.nrow = NULL,
  facet.wrap.ncol = NULL,
  p = 1,
  data = NULL
)

Arguments

obj  
PartialDependenceData
Generated by generatePartialDependenceData.

geom  
(character(1))
The type of geom to use to display the data. Can be “line” or “tile”. For tiling at least two features must be used with interaction = TRUE in the call to generatePartialDependenceData. This may be used in conjunction with the facet argument if three features are specified in the call to generatePartialDependenceData. Default is “line”.

facet  
(character(1))
The name of a feature to be used for facetting. This feature must have been an element of the features argument to generatePartialDependenceData and is only applicable when said argument had length greater than 1. The feature must be a factor or an integer. If generatePartialDependenceData is called with the interaction argument FALSE (the default) with argument features of length greater than one, then facet is ignored and each feature is plotted in its own facet. Default is NULL.

facet.wrap.nrow, facet.wrap.ncol  
(integer)
Number of rows and columns for facetting. Default for both is NULL. In this case ggplot’s facet_wrap will choose the layout itself.

p  
(numeric(1))
If individual = TRUE then sample allows the user to sample without replacement from the output to make the display more readable. Each row is sampled with probability p. Default is 1.
plotResiduals

Description
Plots for model diagnostics. Provides scatterplots of true vs. predicted values and histograms of the model’s residuals.

Usage
plotResiduals(
  obj,
  type = "scatterplot",
  loess.smooth = TRUE,
  rug = TRUE,
  pretty.names = TRUE
)

Arguments

obj  
(Prediction | BenchmarkResult)
Input data.

type  
Type of plot. Can be “scatterplot”, the default. Or “hist”, for a histogram, or in case of classification problems a barplot, displaying the residuals.

loess.smooth  
(logical(1))
Should a loess smoother be added to the plot? Defaults to TRUE. Only applicable for regression tasks and if type is set to scatterplot.
rug (logical(1))
Should marginal distributions be added to the plot? Defaults to TRUE. Only applicable for regression tasks and if type is set to scatterplot.

pretty.names (logical(1))
Whether to use the short name of the learner instead of its ID in labels. Defaults to TRUE.
Only applicable if a BenchmarkResult is passed to obj in the function call, ignored otherwise.

Value

ggplot2 plot object.

See Also

Other plot: createSpatialResamplingPlots(), plotBMRRBoxplots(), plotBMRRanksAsBarChart(), plotBMRSSummary(), plotCalibration(), plotCritDifferences(), plotLearningCurve(), plotPartialDependence(), plotROCCurves(), plotThreshVsPerf()

plotROCCurves

Plots a ROC curve using ggplot2.

Description

Plots a ROC curve from predictions.

Usage

plotROCCurves(
  obj,
  measures,
  diagonal = TRUE,
  pretty.names = TRUE,
  facet.learner = FALSE
)

Arguments

obj (ThreshVsPerfData)
Result of generateThreshVsPerfData.

measures (list(2) of Measure)
Default is the first 2 measures passed to generateThreshVsPerfData.

diagonal (logical(1))
Whether to plot a dashed diagonal line. Default is TRUE.

pretty.names (logical(1))
Whether to use the Measure name instead of the id in the plot. Default is TRUE.
plotThreshVsPerf

Plot threshold vs. performance(s) for 2-class classification using ggplot2.

Description

Plots threshold vs. performance(s) data that has been generated with generateThreshVsPerfData.

Usage

plotThreshVsPerf(
    obj,
    measures = obj$measures,
    facet = "measure",
    mark.th = NA_real_,
    
    facet.learner (logical(1))
    Weather to use facetting or different colors to compare multiple learners. Default is FALSE.

Value

ggplot2 plot object.

See Also

Other plot: createSpatialResamplingPlots(), plotBMRBoxplots(), plotBMRRanksAsBarChart(), plotBMRSummary(), plotCalibration(), plotCritDifferences(), plotLearningCurve(), plotPartialDependence(), plotResiduals(), plotThreshVsPerf()

Other thresh_vs_perf: generateThreshVsPerfData(), plotThreshVsPerf()

Examples

lrn = makeLearner("classif.rpart", predict.type = "prob")
fit = train(lrn, sonar.task)
pred = predict(fit, task = sonar.task)
roc = generateThreshVsPerfData(pred, list(fpr, tpr))
plotROCCurves(roc)

r = bootstrapB632plus(lrn, sonar.task, iters = 3)
roc_r = generateThreshVsPerfData(r, list(fpr, tpr), aggregate = FALSE)
plotROCCurves(roc_r)

r2 = crossval(lrn, sonar.task, iters = 3)
roc_l = generateThreshVsPerfData(list(boot = r, cv = r2), list(fpr, tpr), aggregate = FALSE)
plotROCCurves(roc_l)
Arguments

**obj**  
(ThreshVsPerfData)  
Result of `generateThreshVsPerfData`.

**measures**  
(Measure | list of Measure)  
Performance measure(s) to plot. Must be a subset of those used in `generateThreshVsPerfData`. Default is all the measures stored in `obj` generated by `generateThreshVsPerfData`.

**facet**  
(character(1))  
Selects “measure” or “learner” to be the facetting variable. The variable mapped to `facet` must have more than one unique value, otherwise it will be ignored. The variable not chosen is mapped to color if it has more than one unique value. The default is “measure”.

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

**pretty.names**  
(logical(1))  
Whether to use the Measure name instead of the id in the plot. Default is TRUE.

**facet.wrap.nrow, facet.wrap.ncol**  
(integer)  
Number of rows and columns for facetting. Default for both is NULL. In this case `ggplot`’s `facet_wrap` will choose the layout itself.

Value

ggplot2 plot object.

See Also

Other plot: `createSpatialResamplingPlots()`, `plotBMRBoxplots()`, `plotBMRRanksAsBarChart()`, `plotBMRSummary()`, `plotCalibration()`, `plotCritDifferences()`, `plotLearningCurve()`, `plotPartialDependence()`

Other thresh_vs_perf: `generateThreshVsPerfData()`, `plotROCCurves()`

Examples

```r
lrn = makeLearner("classif.rpart", predict.type = "prob")
mod = train(lrn, sonar.task)
pred = predict(mod, sonar.task)
pvs = generateThreshVsPerfData(pred, list(acc, setAggregation(acc, train.mean)))
plotThreshVsPerf(pvs)
```
**plotTuneMultiCritResult**

Plots multi-criteria results after tuning using ggplot2.

**Description**

Visualizes the pareto front and possibly the dominated points.

**Usage**

```r
plotTuneMultiCritResult(
  res,
  path = TRUE,
  col = NULL,
  shape = NULL,
  pointsize = 2,
  pretty.names = TRUE
)
```

**Arguments**

- `res` *(TuneMultiCritResult)* Result of `tuneParamsMultiCrit`.
- `path` *(logical)* 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)* Which column of `res$opt.path` should be mapped to `ggplot2` color? Default is `NULL`, which means none.
- `shape` *(character)* Which column of `res$opt.path` should be mapped to `ggplot2` shape? Default is `NULL`, which means none.
- `pointsize` *(numeric)* Point size for `ggplot2` `ggplot2::geom_point` for data points. Default is 2.
- `pretty.names` *(logical)* Whether to use the ID of the measures instead of their name in labels. Defaults to `TRUE`.

**Value**

`ggplot2` plot object.

**See Also**

Other `tune_multicrit`: `TuneMultiCritControl`, `tuneParamsMultiCrit()`
**predict.WrappedModel**  

**Examples**

```r
# see tuneParamsMultiCrit
```

---

**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.

The row names of the input task or `newdata` are preserved in the output.

**Usage**

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

**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 | logical | NULL)*  
  Selected cases. Either a logical or an index vector. By default `NULL` if all observations are used.

- `...` *(any)*  
  Currently ignored.

**Value**

*(Prediction).*

**See Also**

Other `predict`: `asROCRPrediction()`, `getPredictionProbabilities()`, `getPredictionResponse()`, `getPredictionTaskDesc()`, `setPredictThreshold()`, `setPredictType()`
Examples

```r
# 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 probabilities 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)
getPredictionProbabilities(p)
```

---

**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

Your implementation must adhere to the following: Predictions for the observations in `.newdata` must be made based on the fitted model (`.model$learner.model`). All parameters in `...` must be passed to the underlying predict function.
Value

- For classification: Either a factor with class labels for type “response” or, if the learner supports this, a matrix of class probabilities for type “prob”. In the latter case the columns must be named with the class labels.
- For regression: Either a numeric vector for type “response” or, if the learner supports this, a matrix with two columns for type “se”. In the latter case the first column contains the estimated response (mean value) and the second column the estimated standard errors.
- For survival: Either a numeric vector with some sort of orderable risk for type “response” or, if supported, a numeric vector with time dependent probabilities for type “prob”.
- For clustering: Either an integer with cluster IDs for type “response” or, if supported, a matrix of membership probabilities for type “prob”.
- For multilabel: A logical matrix that indicates predicted class labels for type “response” or, if supported, a matrix of class probabilities for type “prob”. The columns must be named with the class labels.

reduceBatchmarkResults

Reduce results of a batch-distributed benchmark.

Description

This creates a BenchmarkResult from a batchtools::ExperimentRegistry. To setup the benchmark have a look at batchmark.

Usage

reduceBatchmarkResults(
  ids = NULL,
  keep.pred = TRUE,
  keep.extract = FALSE,
  show.info = getMlrOption("show.info"),
  reg = batchtools::getDefaultRegistry()
)

Arguments

ids (data.frame or integer)
A base::data.frame (or data.table::data.table) with a column named “job.id”. Alternatively, you may also pass a vector of integerish job ids. If not set, defaults to all successfully terminated jobs (return value of batchtools::findDone).

keep.pred (logical(1))
Keep the prediction data in the pred slot of the result object. If you do many experiments (on larger data sets) these objects might unnecessarily increase object size / mem usage, if you do not really need them. The default is set to TRUE.
keep.extract: (logical(1))
Keep the extract slot of the result object. When creating a lot of benchmark results with extensive tuning, the resulting R objects can become very large in size. That is why the tuning results stored in the extract slot are removed by default (keep.extract = FALSE). Note that when keep.extract = FALSE you will not be able to conduct analysis in the tuning results.

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

reg: (batchtools::ExperimentRegistry)
Registry, created by batchtools::makeExperimentRegistry. If not explicitly passed, uses the last created registry.

Value
(BenchmarkResult).

See Also
Other benchmark: BenchmarkResult, batchmark(), benchmark(), convertBMRToRankMatrix(),
friedmanPostHocTestBMR(), friedmanTestBMR(), generateCritDifferencesData(), getBMRAggrPerformances(),
getBMRFeatSelResults(), getBMRfilteredFeatures(), getBMRLearnerIds(), getBMRLearnerShortNames(),
getBMRlearners(), getBMRMeasureIds(), getBMRMeasures(), getBMRModels(), getBMRPerformances(),
getBMRpredictions(), getBMRTaskDescs(), getBMRTaskIds(), getBMRTuneResults(), plotBMRBboxplots(),
plotBMRRanksAsBarChart(), plotBMRSsummary(), plotCritDifferences()

---

reextractFDAFeatures  
**Re-extract features from a data set**

Description
This function accepts a data frame or a task and an extractFDAFeatDesc (a FDA feature extraction description) as returned by `extractFDAFeatures` to extract features from previously unseen data.

Usage

```r
reextractFDAFeatures(obj, desc, ...)
```

Arguments

- **obj** (Task | data.frame)
  Task or data.frame to extract functional features from. Must contain functional features as matrix columns.

- **desc** (extractFDAFeatDesc)
  FDAFeature extraction description as returned by `extractFDAFeatures`

- **...** (any)
  Further args passed on to methods.
reimpute

Value

data.frame or Task containing the extracted Features

Description

This function accepts a data frame or a task and an 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(obj, desc)

Arguments

obj (data.frame | Task)
Input data.

desc (ImputationDesc)
Imputation description as returned by `impute`.

Value

Imputated data.frame or task with imputed data.

See Also

Other impute: `imputations`, `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

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

Arguments

- **obj** *(data.frame | Task)*
  Input data.

- **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?) Note that if the feature has only missing values, it is always removed. Default is FALSE.

- **wrap.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(.Machine$double.eps)`.

- **show.info** *(logical(1))*
  Print verbose output on console? Default is set via `configureMlr`.

- **...**
  To ensure backward compatibility with old argument `tol`.

Value

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

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: LearnerProperties, getClassWeightParam(), getHyperPars(), getLearnerId(), getLearnerNote(), getLearnerPackages(), getLearnerParVals(), getLearnerParamSet(), getLearnerPredictType(), getLearnerShortName(), getLearnerType(), getParamSet(), helpLearner(), helpLearnerParam(), makeLearner(), makeLearners(), setHyperPars(), setId(), setLearnerId(), setPredictThreshold(), setPredictType()
resample

Fit models according to a resampling strategy.

Description

The function resample fits a model specified by Learner on a Task and calculates predictions and performance measures for all training and all test sets specified by a either a resampling description (ResampleDesc) or resampling instance (ResampleInstance).

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.

The remaining functions on this page are convenience wrappers for the various existing resampling strategies. Note that if you need to work with precomputed training and test splits (i.e., resampling instances), you have to stick with resample.

Usage

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

crossval(
  learner,
  task,
  iters = 10L,
  stratify = FALSE,
  measures,
  models = FALSE,
  keep.pred = TRUE,
  ..., 
  show.info = getMlrOption("show.info")
)

crepcv(
  learner,
  task,
  folds = 10L,
  reps = 10L,
stratify = FALSE,
measures,
models = FALSE,
keep.pred = TRUE,
...
show.info = getMlrOption("show.info")
)

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

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

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

bootstrapB632(
  learner,
  task,
resample

iters = 30,
stratify = FALSE,
measures,
models = FALSE,
keep.pred = TRUE,
...
show.info = getMlrOption("show.info")
)

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

growingcv(
  learner,
task,
horizon = 1,
initial.window = 0.5,
skip = 0,
measures,
models = FALSE,
keep.pred = TRUE,
...
show.info = getMlrOption("show.info")
)

fixedcv(
  learner,
task,
horizon = 1L,
initial.window = 0.5,
skip = 0,
measures,
models = FALSE,
keep.pred = TRUE,
...
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.

measures  (Measure | list of Measure)
Performance measure(s) to evaluate. Default is the default measure for the task, see here getDefaultMeasure.

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.

keep.pred  (logical(1))
Keep the prediction data in the pred slot of the result object. If you do many experiments (on larger data sets) these objects might unnecessarily increase object size / mem usage, if you do not really need them. The default is set to TRUE.

...  (any)
Further hyperparameters passed to learner.

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

iters  (integer(1))
See ResampleDesc.

stratify  (logical(1))
See ResampleDesc.

folds  (integer(1))
See ResampleDesc.

reps  (integer(1))
See ResampleDesc.

split  (numeric(1))
See ResampleDesc.

horizon  (numeric(1))
See ResampleDesc.

initial.window  (numeric(1))
See ResampleDesc.

skip  (integer(1))
See ResampleDesc.
ResamplePrediction

Value

(ResampleResult).

Note

If you would like to include results from the training data set, make sure to appropriately adjust the resampling strategy and the aggregation for the measure. See example code below.

See Also

Other resample: ResamplePrediction, ResampleResult, addRRMeasure(), getRRPredictionList(), getRRPredictions(), getRRTaskDesc(), getRRTaskDescription(), makeResampleDesc(), makeResampleInstance()

Examples

```r
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)

# include the training set performance as well
rdesc = makeResampleDesc("CV", iters = 2, predict = "both")
r = resample(makeLearner("classif.qda"), task, rdesc,
          measures = list(mmce, setAggregation(mmce, train.mean)))
print(r$aggr)
```

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 iter, 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: ResampleResult, addRRMeasure(), getRRPredictionList(), getRRPredictions(), getRRTaskDesc(), getRRTaskDescription(), makeResampleDesc(), makeResampleInstance(), resample()
ResampleResult

ResampleResult object.

Description

A container for resample results.

Details

Resample Result:

A resample result is created by resample and contains the following object members:

- **task.id** (character(1)): Name of the Task.
- **learner.id** (character(1)): Name of the Learner.
- **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>`.
- **errmsgs** (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`.
- **err.dumps** (list of list of dump.frames): List with length equal to number of resampling iterations. Contains lists of `dump.frames` objects that can be fed to `debugger()` to inspect error dumps generated on learner errors. One iteration can generate more than one error dump depending on which of training, prediction on training set, or prediction on test set, operations fail. Therefore the lists have named slots `$train`, `$predict.train`, or `$predict.test` if relevant. The error dumps are only saved when option `on.error.dump` is `TRUE`.
- **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`.
- **runtime** (numeric(1)): Time in seconds it took to execute the resampling.

The print method of this object gives a short overview, including task and learner ids, aggregated measures and runtime for the resampling.

See Also

Other resample: `ResamplePrediction`, `addRRMeasure()`, `getRRPredictionList()`, `getRRPredictions()`, `getRRTaskDesc()`, `getRRTaskDescription()`, `makeResampleDesc()`, `makeResampleInstance()`, `resample()`

Other debug: `FailureModel`, `getPredictionDump()`, `getRRDump()`
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 do not have to do this for the learner to work, but it is strongly recommended), and use property tags to define features of the learner.

For a general overview on how to integrate a learning algorithm into mlr's system, please read the section in the online tutorial: https://mlr.mlr-org.com/articles/tutorial/create_learner.html

To see all possible properties of a learner, go to: LearnerProperties.

Usage

```r
makeRLearner()
makeRLearnerClassif(c1, package, par.set, par.vals = list(), properties = character(0L), name = c1, short.name = c1, note = "", class.weights.param = NULL, callees = character(0L))
makeRLearnerMultilabel(c1, package, par.set, par.vals = list(), properties = character(0L), name = c1, short.name = c1, note = "", callees = character(0L))
makeRLearnerRegr(c1, package,
Arguments

cl (character(1))
Class of learner. By convention, all classification learners start with “classif.”
all regression learners with “regr.” all survival learners start with “surv.” all clustering learners with “cluster.” and all multilabel classification learners start with “multilabel.”. A list of all integrated learners is available on the learners help page.

**package**
(character)
Package(s) to load for the implementation of the learner.

**par.set**
(ParamHelpers::ParamSet)
Parameter set of (hyper)parameters and their constraints. Dependent parameters with a requires field must use quote and not expression to define it.

**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.

**properties**
(character)
Set of learner properties. See above. Default is character(0).

**name**
(character(1))
Meaningful name for learner. Default is id.

**short.name**
(character(1))
Short name for learner. Should only be a few characters so it can be used in plots and tables. Default is id.

**note**
(character(1))
Additional notes regarding the learner and its integration in mlr. Default is “”.

**class.weights.param**
(character(1))
Name of the parameter, which can be used for providing class weights.

**callees**
(character)
Character vector naming all functions of the learner's package being called which have a relevant R help page. Default is character(0).

**Value**

(RLearner). The specific subclass is one of R LearnerClassif, R LearnerCluster, R Learner Multilabel, R LearnerRegr, R LearnerSurv.

**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.
selectFeatures

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 behavior, 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. Default is the default measure for the task, see here getDefaultMeasure.
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. Has to be used together with bits.to.features.
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. The vector x will correspond to the bit.names and has to be of the same length.
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, analyzeFeatSelResult(), getFeatSelResult(), makeFeatSelWrapper()
Examples

```r
desc = 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.

See Also

Other performance: `ConfusionMatrix`, `calculateConfusionMatrix()`, `calculateROCMeasures()`, `estimateRelativeOverfitting()`, `makeCostMeasure()`, `makeCustomResampledMeasure()`, `makeMeasure()`, `measures`, `performance()`, `setMeasurePars()`
setHyperPars

Set the hyperparameters of a learner object.

Description

Set the hyperparameters of a learner object.

Usage

setHyperPars(learner, ..., par.vals = list())

Arguments

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

... (any)
  Optional named (hyper)parameters. If you want to set specific hyperparameters for a learner during model creation, these should go here. You can get a list of available hyperparameters usinggetParamSet(<learner>). Alternatively hyperparameters can be given using the par.vals argument but ... should be preferred!

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 ... for passing hyperparameters.

Value

Learner.

Note

If a named (hyper)parameter can’t be found for the given learner, the 3 closest (hyper)parameter names will be output in case the user mistyped.

See Also

Other learner: LearnerProperties, getClassWeightParam(), getHyperPars(), getLearnerId(), getLearnerNote(), getLearnerPackages(), getLearnerParVals(), getLearnerParamSet(), getLearnerPredictType(), getLearnerShortName(), getLearnerType(), getParamSet(), helpLearner(), helpLearnerParam(), makeLearner(), makeLearners(), removeHyperPars(), setId(), setLearnerId(), setPredictThreshold(), setPredictType()
Examples

```r
c11 = makeLearner("classif.ksvm", sigma = 1)
c12 = setHyperPars(c11, sigma = 10, par.vals = list(C = 2))
print(c11)
# note the now set and altered hyperparameters:
print(c12)
```

 setId

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

**Description**

Deprecated, use `setLearnerId` instead.

**Usage**

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

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>learner</td>
<td>(Learner) The learner. If you pass a string the learner will be created via <code>makeLearner</code>.</td>
</tr>
<tr>
<td>id</td>
<td>(character(1)) New id for learner.</td>
</tr>
</tbody>
</table>

setHyperPars2

*Only exported for internal use.*

**Description**

Only exported for internal use.

**Usage**

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

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>learner</td>
<td>(Learner)</td>
</tr>
<tr>
<td>par.vals</td>
<td>(list) List of named (hyper)parameter settings.</td>
</tr>
</tbody>
</table>

Set the id of a learner object.
**setLearnerId**

Set the ID of a learner object.

**Description**

Set the ID of the learner.

**Usage**

```r
setLearnerId(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: `LearnerProperties`, `getClassWeightParam()`, `getHyperPars()`, `getLearnerId()`, `getLearnerNote()`, `getLearnerPackages()`, `getLearnerParVals()`, `getLearnerParamSet()`, `getLearnerPredictType()`, `getLearnerShortName()`, `getLearnerType()`, `getParamSet()`, `helpLearner()`, `helpLearnerParam()`, `makeLearner()`, `makeLearners()`, `removeHyperPars()`, `setHyperPars()`, `setLearnerId()`, `setPredictThreshold()`, `setPredictType()`
**setMeasurePars**  
*Set parameters of performance measures*

**Description**
Sets hyperparameters of measures.

**Usage**
```
setMeasurePars(measure, ..., par.vals = list())
```

**Arguments**
- `measure` *(Measure)*
  Performance measure.
- `...` *(any)*
  Named (hyper)parameters with new settings. 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**
*Measure.*

**See Also**
Other performance: `ConfusionMatrix, calculateConfusionMatrix(), calculateROCMeasures(), estimateRelativeOverfitting(), makeCostMeasure(), makeCustomResampledMeasure(), makeMeasure(), measures, performance(), setAggregation()`

---

**setPredictThreshold**  
*Set the probability threshold the learner should use.*

**Description**
See `predict.threshold` in `makeLearner` and `setThreshold`.
For complex wrappers only the top-level `predict.type` is currently set.

**Usage**
```
setPredictThreshold(learner, predict.threshold)
```
setPredictType

Arguments

learner (Learner | character(1))
The learner. If you pass a string the learner will be created via makeLearner.
predict.threshold (numeric)
Threshold to produce class labels. Has to be a named vector, where names corre-
spond to class labels. Only for binary classification it can be a single numerical
threshold for the positive class. See setThreshold for details on how it is applied.
Default is NULL which means 0.5 / an equal threshold for each class.

Value
Learner.

See Also
Other predict: asROCRPrediction(), getPredictionProbabilities(),
getPredictionResponse(), getPredictionTaskDesc(), predict.WrappedModel(), setPredictType()
Other learner: LearnerProperties, getClassWeightParam(), getHyperPars(), getLearnerId(),
getLearnerNote(), getLearnerPackages(), getLearnerParamSet(),
getLearnerPredictType(), getLearnerShortName(), getLearnerType(),getParamSet(), helpLearner(),
helpLearnerParam(), makeLearner(), makeLearners(), removeHyperPars(), setHyperPars(),
setId(), setName(), setPredictType()
Value
Learner.

See Also
Other predict: asROCRPrediction(), getPredictionProbabilities(), getPredictionResponse(), getPredictionTaskDesc(), predict.WrappedModel(), setPredictThreshold()

Other learner: LearnerProperties, getClassWeightParam(), getHyperPars(), getLearnerId(), getLearnerNote(), getLearnerPackages(), getLearnerParVals(), getLearnerParamSet(), getLearnerPredictType(), getLearnerShortName(), getLearnerType(),getParamSet(), helpLearner(), helpLearnerParam(), makeLearner(), makeLearners(), removeHyperPars(), setHyperPars(), setId(), setLearnerId(), setPredictThreshold()

---

**setThreshold**

Set threshold of prediction object.

Description
Set threshold of prediction object for classification or multilabel 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. For multilabel classification: A label is predicted (with entry TRUE) if a probability matrix entry exceeds the threshold of the corresponding label.

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 for 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

```r
# 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)))
```

---

`simplifyMeasureNames`  
_Simplify measure names._

**Description**

Clips aggregation names from character vector. E.g: 'mmce.test.mean' becomes 'mmce'. Elements that don’t contain a measure name are ignored and returned unchanged.

**Usage**

`simplifyMeasureNames(xs)`

**Arguments**

- `xs`  
  (character)  
  Character vector that (possibly) contains aggregated measure names.

**Value**

(character).
smote

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

**Description**

In each iteration, samples one minority class element \(x_1\), then one of \(x_1\)’s nearest neighbors: \(x_2\). Both points are now interpolated / convex-combined, resulting in a new virtual data point \(x_3\) for the minority class.

The method handles factor features, too. The gower distance is used for nearest neighbor calculation, see `cluster::daisy`. For interpolation, the new factor level for \(x_3\) is sampled from the two given levels of \(x_1\) and \(x_2\) per feature.

**Usage**

```r
smote(task, rate, nn = 5L, standardize = TRUE, alt.logic = FALSE)
```

**Arguments**

- `task` *(Task)*
  The task.
- `rate` *(numeric(1))*
  Factor to upsample the smaller class. Must be between 1 and \(\text{Inf}\), 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.
- `standardize` *(integer(1))*
  Standardize input variables before calculating the nearest neighbors for data sets with numeric input variables only. For mixed variables (numeric and factor) the gower distance is used and variables are standardized anyway. Default is `TRUE`.
- `alt.logic` *(integer(1))*
  Use an alternative logic for selection of minority class observations. Instead of sampling a minority class element AND one of its nearest neighbors, each minority class element is taken multiple times (depending on rate) for the interpolation and only the corresponding nearest neighbor is sampled. Default is `FALSE`.

**Value**

*Task.*

**References**

See Also

Other imbalancy: `makeOverBaggingWrapper()`, `makeUndersampleWrapper()`, `oversample()`
subsetTask  Subset data in task.

Description
See title.

Usage
subsetTask(task, subset = NULL, features)

Arguments
- **task**  (Task) The task.
- **subset**  (integer | logical | NULL) Selected cases. Either a logical or an index vector. By default NULL if all observations are used.
- **features**  (character | integer | logical) Vector of selected inputs. You can either pass a character vector with the feature names, a vector of indices, or a logical vector. In case of an index vector each element denotes the position of the feature name returned by `getTaskFeatureNames`. Note that the target feature is always included in the resulting task, you should not pass it here. Default is to use all features.

Value
(Task). Task with subsetted data.

See Also
Other task: `getTaskClassLevels()`, `getTaskCosts()`, `getTaskData()`, `getTaskDesc()`, `getTaskFeatureNames()`, `getTaskFormula()`, `getTaskId()`, `getTaskNFeats()`, `getTaskSize()`, `getTaskTargetNames()`, `getTaskTargets()`, `getTaskType()`

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

Summarize columns of data.frame or task.

Description

Summarizes a data.frame, somewhat differently than the normal summary function of R. The function is mainly useful as a basic EDA tool on data.frames before they are converted to tasks, but can be used on tasks as well.

Columns can be of type numeric, integer, logical, factor, or character. Characters and logicals will be treated as factors.

Usage

```r
summarizeColumns(obj)
```

Arguments

- `obj` (data.frame | Task)
  Input data.

Value

(data.frame). With columns:

- `name` Name of column.
- `type` Data type of column.
- `na` Number of NAs in column.
- `disp` Measure of dispersion, for numerics and integers sd is used, for categorical columns the qualitative variation.
- `mean` Mean value of column, NA for categorical columns.
- `median` Median value of column, NA for categorical columns.
- `mad` MAD of column, NA for categorical columns.
- `min` Minimal value of column, for categorical columns the size of the smallest category.
- `max` Maximal value of column, for categorical columns the size of the largest category.
- `nlevs` For categorical columns, the number of factor levels, NA else.

See Also

Other eda_and_preprocess: capLargeValues(), createDummyFeatures(), dropFeatures(), mergeSmallFactorLevels(), normalizeFeatures(), removeConstantFeatures(), summarizeLevels()

Examples

```r
summarizeColumns(iris)
```
summarizeLevels  

Summarizes factors of a data.frame by tabling them.

Description
Characters and logicals will be treated as factors.

Usage
summarizeLevels(obj, cols = NULL)

Arguments

- **obj** (data.frame | Task)
  Input data.

- **cols** (character)
  Restrict result to columns in cols. Default is all factor, character and logical columns of obj.

Value
(list). Named list of tables.

See Also
Other eda_and_preprocess: capLargeValues(), createDummyFeatures(), dropFeatures(), mergeSmallFactorLevels(), normalizeFeatures(), removeConstantFeatures(), summarizeColumns()

Examples

summarizeLevels(iris)
• `getTaskTargets`, and
• `subsetTask`.

Object members:

`env` (environment) Environment where data for the task are stored. Use `getTaskData` in order to access it.

`weights` (numeric) See argument. NULL if not present.

`blocking` (factor) See argument. NULL if not present.

`task.desc` (TaskDesc) Encapsulates further information about the task.

Functional data can be added to a task via matrix columns. For more information refer to `make-FunctionalData`.

**Arguments**

**id**
(character(1))
Id string for object. Default is the name of the R variable passed to `data`.

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

**target**
(character(1) | character(2) | character(n.classes))
Name(s) of the target variable(s). For survival analysis these are the names of the survival time and event columns, so it has length 2. For multilabel classification it contains the names of the logical columns that encode whether a label is present or not and its length corresponds to the number of classes.

**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 (otherwise ignored and set to NA). 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 choices 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.

coordinates (data.frame)
Coordinates of a spatial data set that will be used for spatial partitioning of the data in a spatial cross-validation resampling setting. Coordinates have to be numeric values. Provided data.frame needs to have the same number of rows as data and consist of at least two dimensions.

Value
Task.

See Also
ClassifTask ClusterTask CostSensTask MultilabelTask RegrTask SurvTask

Examples
if (requireNamespace("mlbench")) {
  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])
}

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 and “cluster” for cluster analysis, “costsens” for cost-sensitive classification, and “multilabel” for multilabel classification.
- **target** (`character(0) | character(1) | character(2) | character(n.classes)`) Name(s) of the target variable(s). For “surv” these are the names of the survival time and event columns, so it has length 2. For “costsens” it has length 0, as there is no target column, but a cost matrix instead. For “multilabel” these are the names of logical columns that indicate whether a class label is present and the number of target variables corresponds to the number of classes.
- **size** (`integer(1)`) Number of cases in data set.
- **n.feat** (`integer(2)`) Number of features, named vector with entries: “numerics”, “factors”, “ordered”, “functionals”.
- **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”, “costsens”, and “multilabel”.
- **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 = NULL, 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 | logical | NULL)
Selected cases. Either a logical or an index vector. By default NULL if all observations are 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(seq_len(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.
- ...
  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 in ... must be passed to the underlying training function.

Value

(any). Model of the underlying learner.

---

TuneControl

Control object for tuning

Description

General tune control object.

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. Imputation is only active if on.learner.error is configured not to stop in configureMlr. 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.
**TuneMultiCritControl**

Create control structures for multi-criteria tuning.

**Description**

The following tuners are available:

- **makeTuneMultiCritControlGrid** 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 ParamHelpers::makeDiscreteParam in the par.set passed to tuneParams.
TuneMultiCritControl

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

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

**makeTuneMultiCritControlMBO** Model-based/ Bayesian optimization. All kinds of parameter types can be handled.

Usage

```r
makeTuneMultiCritControlGrid(
  same.resampling.instance = TRUE,
  resolution = 10L,
  log.fun = "default",
  final.dw.perc = NULL,
  budget = NULL
)

makeTuneMultiCritControlMBO(
  n.objectives = mbo.control$n.objectives,
  same.resampling.instance = TRUE,
  impute.val = NULL,
  learner = NULL,
  mbo.control = NULL,
  tune.threshold = FALSE,
  tune.threshold.args = list(),
  continue = FALSE,
  log.fun = "default",
  final.dw.perc = NULL,
  budget = NULL,
  mbo.design = NULL
)

makeTuneMultiCritControlNSGA2(
  same.resampling.instance = TRUE,
  impute.val = NULL,
  log.fun = "default",
  final.dw.perc = NULL,
  budget = NULL,
  ...
)

makeTuneMultiCritControlRandom(
  same.resampling.instance = TRUE,
  maxit = 100L,
  log.fun = "default",
  final.dw.perc = NULL,
  budget = NULL
)```
Arguments

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 ParamHelpers::generateGridDesign. Default is 10.

log.fun
  (function | character(1))
  Function used for logging. If set to “default” (the default), the evaluated design points, the resulting performances, and the runtime will be reported. If set to “memory” the memory usage for each evaluation will also be displayed, with character(1) small increase in run time. Otherwise character(1) function with arguments learner, resampling, measures, par.set, control, opt.path, dob, x, y, remove.nas, stage and prev.stage is expected. The default displays the performance measures, the time needed for evaluating, the currently used memory and the max memory ever used before (the latter two both taken from gc). See the implementation for details.

final.dw.perc
  (boolean)
  If a Learner wrapped by a makeDownsampleWrapper is used, you can define the value of dw.perc which is used to train the Learner with the final parameter setting found by the tuning. Default is NULL which will not change anything.

budget
  (integer(1))
  Maximum budget for tuning. This value restricts the number of function evaluations. In case of makeTuneMultiCritControlGrid this number must be identical to the size of the grid. For makeTuneMultiCritControlRandom the budget equals the number of iterations (maxit) performed by the random search algorithm. In case of makeTuneMultiCritControlNSGA2 the budget corresponds to the product of the maximum number of generations (max(generations)) + 1 (for the initial population) and the size of the population (popsize). For makeTuneMultiCritControlMBO the budget equals the number of objective function evaluations, i.e. the number of MBO iterations + the size of the initial design. If not NULL, this will overwrite existing stopping conditions in mbo.control.

n.objectives
  (integer(1))
  Number of objectives, i.e. number of Measures to optimize.

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. Imputation is only active if on.learner.error is configured not to stop in configureMlr. 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 $\text{Inf}$ instead. For multi-criteria optimization pass a vector of imputation values, one for each of your measures, in the same order as your measures.

**learner**

(Learner | NULL)

The surrogate learner: A regression learner to model performance landscape. For the default, NULL, *mlrMBO* will automatically create a suitable learner based on the rules described in *mlrMBO::makeMBOControl*.

**mbo.control**

(*mlrMBO::MBOControl | NULL*)

Control object for model-based optimization tuning. For the default, NULL, the control object will be created with all the defaults as described in *mlrMBO::makeMBOControl*.

**tune.threshold**

(logical(1))

Should the threshold be tuned for the measure at hand, after each hyperparameter evaluation, via *tuneThreshold*? Only works for classification if the predict type is “prob”. Default is FALSE.

**tune.threshold.args**

(list)

Further arguments for threshold tuning that are passed down to *tuneThreshold*. Default is none.

**continue**

(logical(1))

Resume calculation from previous run using *mlrMBO::mboContinue*? Requires “save.file.path” to be set. Note that the ParamHelpers::OptPath in the *mlrMBO::OptResult* will only include the evaluations after the continuation. The complete OptPath will be found in the slot $mbo.result$opt.path.

**mbo.design**

(data.frame | NULL)

Initial design as data frame. If the parameters have corresponding trafo functions, the design must not be transformed before it is passed! For the default, NULL, a default design is created like described in *mlrMBO::mbo*.

... (any)

Further control parameters passed to the control arguments of *cmaes::cma_es* or *GenSA::GenSA*, as well as towards the tunerConfig argument of *irace::irace*.

**maxit**

(integer(1))

Number of iterations for random search. Default is 100.

**Value**

*(TuneMultiCritControl)*. The specific subclass is one of TuneMultiCritControlGrid, TuneMultiCritControlRandom, TuneMultiCritControlNSGA2, TuneMultiCritControlMBO.

**See Also**

Other tune_multicrit: *plotTuneMultiCritResult*, *tuneParamsMultiCrit*
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 \( x \).
threshold  Currently NULL.
\( \text{opt.path} \) (ParamHelpers::OptPath)  Optimization path which lead to \( x \). Note that when you have trafos on some of your params, the \( \text{opt.path} \) always contains the UNTRANSFORMED values on the original scale. You can simply call \( \text{trafoOptPath}(\text{opt.path}) \) to transform them, or, \( \text{as.data.frame(trafoOptPath(\text{opt.path})} \)
\( \text{ind} \) (integer(n))  Indices of Pareto optimal params in \( \text{opt.path} \).
measures (list of Measure)  Performance measures.

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"),
  resample.fun = resample
)
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. Default is the
default measure for the task, see here getDefaultMeasure.

par.set (ParamHelpers::ParamSet)
Collection of parameters and their constraints for optimization. Dependent pa-
rameters with a requires field must use quote and not expression to define it.

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.

resample.fun (closure)
The function to use for resampling. Defaults to resample. If a user-given func-
tion is to be used instead, it should take the arguments “learner”, “task”, “re-
sampling”, “measures”, and “show.info”; see resample. Within this function, it
is easiest to call resample and possibly modify the result. However, it is pos-
sible to return a list with only the following essential slots: the “agr” slot for
general tuning, additionally the “pred” slot if threshold tuning is performed (see
TuneControl), and the “errmsgs” and “err.dumps” slots for error reporting. This
parameter must be the default when mbo tuning is performed.

Value

(TuneResult).

Note

If you would like to include results from the training data set, make sure to appropriately adjust the
resampling strategy and the aggregation for the measure. See example code below.

See Also

generateHyperParsEffectData

Other tune: TuneControl, getNestedTuneResultsOptPathDf(), getNestedTuneResultsX(),
getResamplingIndices(), getResult(), makeModelMultiplexer(), makeModelMultiplexerParamSet(),
Examples

set.seed(123)
# 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)
# access data for all evaluated points
df = as.data.frame(res$opt.path)
df1 = as.data.frame(res$opt.path, trafo = TRUE)
print(head(df[, -ncol(df)]))
print(head(df1[, -ncol(df1)]))
# access data for all evaluated points - alternative
df2 = generateHyperParsEffectData(res)
df3 = generateHyperParsEffectData(res, trafo = TRUE)
print(head(df2$data[, -ncol(df2$data)]))
print(head(df3$data[, -ncol(df3$data)]))
## Not run:
# we optimize the SVM over 3 kernels simultaneously
# 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 = 5, nbIterations = 1, minNbSurvival = 1)
rdesc = makeResampleDesc("Holdout")
res = tuneParams("classif.ksvm", iris.task, rdesc, par.set = ps, control = ctrl)
print(res)
df = as.data.frame(res$opt.path)
print(head(df[, -ncol(df)]))
# include the training set performance as well
rdesc = makeResampleDesc("Holdout", predict = "both")
res = tuneParams("classif.ksvm", iris.task, rdesc, par.set = ps,
    control = ctrl, measures = list(mmce, setAggregation(mmce, train.mean)))
print(res)
df2 = as.data.frame(res$opt.path)
print(head(df2[, -ncol(df2)]))

## End(Not run)

tuneParamsMultiCrit  Hyperparameter tuning for multiple measures at once.

Description
Optimizes the hyperparameters of a learner in a multi-criteria fashion. 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"),
  resample.fun = resample
)

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.
measures [list of Measure]
Performance measures to optimize simultaneously.
par.set (ParamHelpers::ParamSet)
Collection of parameters and their constraints for optimization. Dependent parameters with a requires field must use quote and not expression to define it.
control  
(TuneMultiCritControl)  
Control object for search method. Also selects the optimization algorithm for tuning.

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

resample.fun  
(closure)  
The function to use for resampling. Defaults to resample and should take the same arguments as, and return the same result type as, resample.

Value

(TuneMultiCritResult).

See Also

Other tune_multicrit: TuneMultiCritControl, 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 = 1L)  
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.

threshold (**numeric**)  Vector of finally found and used thresholds if tune.threshold was enabled in TuneControl, otherwise not present and hence NULL.

opt.path (**ParamHelpers::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.path)). If mlr option on.error.dump is TRUE, OptPath will have a .dump object in its extra column which contains error dump traces from failed optimization evaluations. It can be accessed by getOptPathEl(opt.path)$extra$.dump.

tuneThreshold  

**Tune prediction threshold.**

**Description**

Optimizes the threshold of predictions based on probabilities. Works for classification and multilabel tasks. Uses BBmisc::optimizeSubInts for normal binary class problems and GenSA::GenSA for multiclass and multilabel problems.

**Usage**

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

**Arguments**

- **pred** (**Prediction**)  Prediction object.
- **measure** (**Measure**)  Performance measure to optimize. Default is the default measure for the task.
- **task** (**Task**)  Learning task. Rarely needed, only when required for the performance measure.
- **model** (**WrappedModel**)  Fitted model. Rarely needed, only when required for the performance measure.
- **nsub** (**integer(1)**)  Passed to BBmisc::optimizeSubInts for 2class problems. Default is 20.
- **control** (**list**)  Control object for GenSA::GenSA when used. Default is empty list.
Value

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

See Also

Other tune: TuneControl, getNestedTuneResultsOptPathDf(), getNestedTuneResultsX(), getResamplingIndices(), getTuneResult(), makeModelMultiplexer(), makeModelMultiplexerParamSet(), makeTuneControlCMAES(), makeTuneControlDesign(), makeTuneControlGenSA(), makeTuneControlGrid(), makeTuneControlIrace(), makeTuneControlMBO(), makeTuneControlRandom(), makeTuneWrapper(), tuneParams()

wpbc.task  
Wisconsin Prognostic Breast Cancer (WPBC) survival task.

Description

Contains the task (wpbc.task).

References

See TH.data::wpbc. Incomplete cases have been removed from the task.

yeast.task  
Yeast multilabel classification task.

Description

Contains the task (yeast.task).

Source

https://archive.ics.uci.edu/ml/datasets/Yeast (In long instead of wide format)

References

Index

* benchmark
  batchmark, 13
  benchmark, 15
  BenchmarkResult, 17
  convertBMRToRankMatrix, 26
  friedmanPostHocTestBMR, 50
  friedmanTestBMR, 51
  generateCritDifferencesData, 54
  getBMRAggrPerformances, 67
  getBMRFeatSelResults, 68
  getBMRFilteredFeatures, 70
  getBMRLearnerIds, 71
  getBMRLearners, 72
  getBMRLearnerShortNames, 72
  getBMRMetricIds, 73
  getBMRMeasures, 74
  getBMRModels, 74
  getBMRFinalPerformances, 75
  getBMRFinalPredictions, 76
  getBMRTaskDescs, 78
  getBMRTaskIds, 79
  getBMRTuneResults, 80
  plotBMRExampleBoxplots, 226
  plotBMRRanksAsBarChart, 227
  plotBMRSummary, 228
  plotCritDifferences, 231
  reduceBatchmarkResults, 247

* calibration
  generateCalibrationData, 52
  plotCalibration, 230

* configure
  configureMlr, 23
  getMlrOptions, 96

* costsens
  makeCostSensClassifWrapper, 141
  makeCostSensRegrWrapper, 142
  makeCostSensTask, 143
  makeCostSensWeightedPairsWrapper, 144

* datasets
  aggregations, 11
  measures, 214
  spatial.task, 271

* data
  agri.task, 12
  bc.task, 15
  bh.task, 18
  costiris.task, 27
  fuelsubset.task, 52
  gunpoint.task, 117
  iris.task, 124
  lung.task, 133
  mtcars.task, 220
  phoneme.task, 225
  pid.task, 225
  sonar.task, 271
  spam.task, 271
  wpbc.task, 290
  yeast.task, 290

* debug
  FailureModel, 43
  getPredictionDump, 100
  getRRDump, 104
  ResampleResult, 257

* downsample
  downsample, 32
  makeDownsampleWrapper, 146

* eda_and_preprocess
  capLargeValues, 22
  createDummyFeatures, 28
  dropFeatures, 33
  mergeSmallFactorLevels, 218
  normalizeFeatures, 221
  removeConstantFeatures, 250
  summarizeColumns, 273
  summarizeLevels, 274

* extractFDAFeatures
  reextractFDAFeatures, 248

291
<table>
<thead>
<tr>
<th>Index</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>calculateROCMeasures</td>
<td>20</td>
</tr>
<tr>
<td>ConfusionMatrix</td>
<td>25</td>
</tr>
<tr>
<td>estimateRelativeOverfitting</td>
<td>34</td>
</tr>
<tr>
<td>makeCostMeasure</td>
<td>140</td>
</tr>
<tr>
<td>makeCustomResampledMeasure</td>
<td>145</td>
</tr>
<tr>
<td>makeMeasure</td>
<td>165</td>
</tr>
<tr>
<td>measures</td>
<td>214</td>
</tr>
<tr>
<td>performance</td>
<td>224</td>
</tr>
<tr>
<td>setAggregation</td>
<td>262</td>
</tr>
<tr>
<td>setMeasurePars</td>
<td>266</td>
</tr>
<tr>
<td>* plot</td>
<td></td>
</tr>
<tr>
<td>createSpatialResamplingPlots</td>
<td>29</td>
</tr>
<tr>
<td>plotBMRBoxplots</td>
<td>226</td>
</tr>
<tr>
<td>plotBMRRanksAsBarChart</td>
<td>227</td>
</tr>
<tr>
<td>plotBMRSummary</td>
<td>228</td>
</tr>
<tr>
<td>plotCalibration</td>
<td>230</td>
</tr>
<tr>
<td>plotCritDifferences</td>
<td>231</td>
</tr>
<tr>
<td>plotLearningCurve</td>
<td>238</td>
</tr>
<tr>
<td>plotPartialDependence</td>
<td>239</td>
</tr>
<tr>
<td>plotResiduals</td>
<td>240</td>
</tr>
<tr>
<td>plotROCCurves</td>
<td>241</td>
</tr>
<tr>
<td>plotThreshVsPerf</td>
<td>242</td>
</tr>
<tr>
<td>* predict</td>
<td></td>
</tr>
<tr>
<td>asROCRPrediction</td>
<td>13</td>
</tr>
<tr>
<td>getPredictionProbabilities</td>
<td>100</td>
</tr>
<tr>
<td>getPredictionResponse</td>
<td>101</td>
</tr>
<tr>
<td>getPredictionTaskDesc</td>
<td>102</td>
</tr>
<tr>
<td>predict.WrappedModel</td>
<td>245</td>
</tr>
<tr>
<td>setPredictThreshold</td>
<td>266</td>
</tr>
<tr>
<td>setPredictType</td>
<td>267</td>
</tr>
<tr>
<td>* resample</td>
<td></td>
</tr>
<tr>
<td>addRRMeasure</td>
<td>10</td>
</tr>
<tr>
<td>getRRRPredictionList</td>
<td>105</td>
</tr>
<tr>
<td>getRRRPredictions</td>
<td>105</td>
</tr>
<tr>
<td>getRRTaskDesc</td>
<td>106</td>
</tr>
<tr>
<td>getRRTaskDescription</td>
<td>107</td>
</tr>
<tr>
<td>makeResampleDesc</td>
<td>185</td>
</tr>
<tr>
<td>makeResampleInstance</td>
<td>188</td>
</tr>
<tr>
<td>ResamplePrediction</td>
<td>256</td>
</tr>
<tr>
<td>ResampleResult</td>
<td>257</td>
</tr>
<tr>
<td>* roc</td>
<td></td>
</tr>
<tr>
<td>asROCRPrediction</td>
<td>13</td>
</tr>
<tr>
<td>calculateROCMeasures</td>
<td>20</td>
</tr>
<tr>
<td>* task</td>
<td></td>
</tr>
<tr>
<td>getTaskClassLevels</td>
<td>108</td>
</tr>
<tr>
<td>getTaskCosts</td>
<td>108</td>
</tr>
<tr>
<td>getTaskData</td>
<td>109</td>
</tr>
<tr>
<td>getTaskDesc</td>
<td>110</td>
</tr>
<tr>
<td>getTaskFeatureNames</td>
<td>111</td>
</tr>
<tr>
<td>getTaskFormula</td>
<td>112</td>
</tr>
<tr>
<td>getTaskId</td>
<td>113</td>
</tr>
<tr>
<td>getTaskNFeats</td>
<td>113</td>
</tr>
<tr>
<td>getTaskSize</td>
<td>114</td>
</tr>
<tr>
<td>getTaskTargetNames</td>
<td>114</td>
</tr>
<tr>
<td>getTaskTargets</td>
<td>115</td>
</tr>
<tr>
<td>getTaskType</td>
<td>116</td>
</tr>
<tr>
<td>subsetTask</td>
<td>272</td>
</tr>
<tr>
<td>* thresh_vs_perf</td>
<td></td>
</tr>
<tr>
<td>generateThreshVsPerfData</td>
<td>66</td>
</tr>
<tr>
<td>plotROCCurves</td>
<td>241</td>
</tr>
<tr>
<td>plotThreshVsPerf</td>
<td>242</td>
</tr>
<tr>
<td>* tune_multicrit</td>
<td></td>
</tr>
<tr>
<td>plotTuneMultiCritResult</td>
<td>244</td>
</tr>
<tr>
<td>TuneMultiCritControl</td>
<td>280</td>
</tr>
<tr>
<td>tuneParamsMultiCrit</td>
<td>287</td>
</tr>
<tr>
<td>* tune</td>
<td></td>
</tr>
<tr>
<td>getNestedTuneResultsOptPathDf</td>
<td>97</td>
</tr>
<tr>
<td>getNestedTuneResultsX</td>
<td>98</td>
</tr>
<tr>
<td>getResamplingIndices</td>
<td>103</td>
</tr>
<tr>
<td>getTuneResult</td>
<td>116</td>
</tr>
<tr>
<td>makeModelMultiplexer</td>
<td>167</td>
</tr>
<tr>
<td>makeModelMultiplexerParamSet</td>
<td>169</td>
</tr>
<tr>
<td>makeTuneControlCMAES</td>
<td>195</td>
</tr>
<tr>
<td>makeTuneControlDesign</td>
<td>197</td>
</tr>
<tr>
<td>makeTuneControlGenSA</td>
<td>198</td>
</tr>
<tr>
<td>makeTuneControlGrid</td>
<td>200</td>
</tr>
<tr>
<td>makeTuneControlIrace</td>
<td>202</td>
</tr>
<tr>
<td>makeTuneControlMBO</td>
<td>204</td>
</tr>
<tr>
<td>makeTuneControlRandom</td>
<td>206</td>
</tr>
<tr>
<td>makeTuneWrapper</td>
<td>207</td>
</tr>
<tr>
<td>TuneControl</td>
<td>279</td>
</tr>
<tr>
<td>tuneParams</td>
<td>284</td>
</tr>
<tr>
<td>tuneThreshold</td>
<td>289</td>
</tr>
<tr>
<td>* wrapper</td>
<td></td>
</tr>
<tr>
<td>makeBaggingWrapper</td>
<td>134</td>
</tr>
<tr>
<td>makeClassificationViaRegressionWrapper</td>
<td>135</td>
</tr>
<tr>
<td>makeConstantClassWrapper</td>
<td>139</td>
</tr>
<tr>
<td>makeCostSensClassifWrapper</td>
<td>141</td>
</tr>
<tr>
<td>makeCostSensRegrWrapper</td>
<td>142</td>
</tr>
<tr>
<td>makeDownsampleWrapper</td>
<td>146</td>
</tr>
<tr>
<td>makeDummyFeaturesWrapper</td>
<td>147</td>
</tr>
<tr>
<td>makeExtractFDAFeatsWrapper</td>
<td>149</td>
</tr>
<tr>
<td>makeFeatSelWrapper</td>
<td>150</td>
</tr>
<tr>
<td>makeFilterWrapper</td>
<td>154</td>
</tr>
</tbody>
</table>
makeImputeWrapper, 159
makeMulticlassWrapper, 170
makeMultilabelBinaryRelevanceWrapper, 171
makeMultilabelClassifierChainsWrapper, 172
makeMultilabelDBRWrapper, 173
makeMultilabelNestedStackingWrapper, 175
makeMultilabelStackingWrapper, 176
makeOverBaggingWrapper, 179
makePreprocWrapper, 180
makePreprocWrapperCaret, 182
makeRemoveConstantFeaturesWrapper, 184
makeSMOTEWrapper, 190
makeTuneWrapper, 207
makeUndersampleWrapper, 209
makeWeightedClassesWrapper, 210

acc (measures), 214
addRRMeasure, 10, 105–107, 187, 189, 256, 257
Aggregation, 11, 12, 134, 166, 262
aggregations, 11, 12, 134, 262
agri.task, 12
analyzeFeatSelResult, 12, 47, 85, 151, 261
as.data.frame, 17
asROCRPrediction, 13, 22, 101, 102, 245, 267, 268
auc (measures), 214

b632 (aggregations), 11
b632plus (aggregations), 11
bac (measures), 214
base::data.frame, 247
base::expand.grid, 14
base::rank, 26
batchtools::ExperimentRegistry, 247, 248
batchtools::findDone, 247
batchtools::makeExperimentRegistry, 13, 15, 248
batchtools::Registry, 15
batchtools::submitJobs, 13, 14
batchtools::waitForJobs, 14

BBmisc::normalize, 221, 222
BBmisc::optimizeSubInts, 289
bc.task, 15
ber (measures), 214
bh.task, 18
bootstrapB632 (resample), 252
bootstrapB632plus (resample), 252
bootstrapOOB (resample), 252
brier (measures), 214
cache Helpers, 18
calculateConfusionMatrix, 19, 22, 25, 35, 83, 141, 146, 166, 217, 225, 262, 266
calculateROCMeasures, 13, 20, 21, 25, 35, 141, 146, 166, 217, 225, 262, 266
CalibrationData, 53, 230
CalibrationData
generateCalibrationData, 52
capLargeValues, 22, 28, 33, 218, 222, 251, 273, 274
caret::preProcess, 182
caret::preProcess, 182
caret::preProcess, 182
caret::preProcess, 182
caret::preProcess, 182
caret::preProcess, 182
cindex (measures), 214
ClassifTask, 139, 144, 179, 184, 195, 276
ClassifTask (makeClassifTask), 137
closure, 285, 288
class::agriculture, 12
class::daisy, 270
ClusterTask, 138, 144, 179, 184, 195, 276
ClusterTask (makeClusterTask), 138
cmaes::cma_es, 195, 196, 200, 203, 280, 283
calculateMlr, 16, 23, 43, 46, 62, 84, 85, 96, 100, 104, 124, 151, 162, 196, 197, 199, 201, 202, 204, 208, 248, 250, 255, 257, 261, 279, 282, 285, 288
INDEX

ConfusionMatrix, 19, 20, 22, 25, 35, 141, 146, 166, 217, 225, 262, 266
convertBMRToRankMatrix, 15, 17, 18, 26, 51, 52, 55, 68, 69, 71–77, 79, 81, 227–229, 232, 248
convertMLBenchObjToTask, 27
costiris.task, 27
CostSensClassifModel
   (makeCostSensClassifWrapper), 141
CostSensClassifWrapper
   (makeCostSensClassifWrapper), 141
CostSensRegrModel
   (makeCostSensRegrWrapper), 142
CostSensRegrWrapper
   (makeCostSensRegrWrapper), 142
CostSensTask, 108, 138, 139, 179, 184, 195, 276
CostSensTask(makeCostSensTask), 143
CostSensWeightedPairsModel
   (makeCostSensWeightedPairsWrapper), 144
CostSensWeightedPairsWrapper, 129
CostSensWeightedPairsWrapper
   (makeCostSensWeightedPairsWrapper), 144
cowplot::save_plot, 30
createDummyFeatures, 23, 28, 33, 147, 218, 222, 261, 273, 274
createSpatialResamplingPlots, 29, 227, 228, 230–232, 238, 240–243
crossover, 32, 32
crossval (resample), 252
cv10 (makeResampleDesc), 185
cv2 (makeResampleDesc), 185
cv3 (makeResampleDesc), 185
cv5 (makeResampleDesc), 185
data.table, 15
data.table::data.table, 247
datasets::iris, 27, 124
datasets::mtcars, 221
db (measures), 214
deleteCacheDir (cache_helpers), 18
downsampel, 32, 147
dropFeatures, 23, 28, 33, 218, 222, 251, 273, 274
dump.frames, 257
evironment, 112
estimateRelativeOverfitting, 20, 22, 25, 34, 141, 146, 166, 217, 225, 262, 266
estimateResidualVariance, 35
expvar (measures), 214
extractFDABsignal, 36, 37, 39, 40, 42, 43
extractFDAkern, 36, 37, 39, 40, 42, 43
extractFDAfeatures, 37, 38, 149, 150, 248
extractFDAfourier, 36, 37, 39, 40, 42, 43
extractFADPCA, 36, 37, 39, 40, 42, 43
extractFDAMultiResFeatures, 36, 37, 39, 40, 42, 43
extractFDATsfeatures, 36, 37, 39, 40, 41, 43
extractFDWavelets, 36, 37, 39, 40, 42, 43
f1 (measures), 214
factor, 134, 137, 139, 143, 146, 178, 183, 188, 194, 216, 275
FailureModel, 25, 43, 100, 104, 257
FDboost::bsign() (measures), 36
FDboost::FDboost, 52
fdr (measures), 214
feature (measures), 214
FeaSelControl, 13, 44, 47, 48, 85, 150, 151, 260, 261
FeaSelControlExhaustive, 47
FeaSelControlExhaustive (FeaSelControl), 44
FeaSelControlGA, 47
FeaSelControlGA (FeaSelControl), 44
FeaSelControlRandom, 47
FeaSelControlRandom (FeaSelControl), 44
FeaSelControlSequential, 47
FeaSelControlSequential (FeaSelControl), 44
FeaSelResult, 12, 47, 68, 85, 261
FeatureImportanceData
   (generateFeatureImportanceData), 56
getTaskTargetNames, 108–114, 114, 115, 116, 272
getTaskType, 108–115, 116, 272
getTuneResultOptPath, 117
ggplot2::coord_sf, 30
ggplot2::geom_point, 229, 237, 244
ggplot2::ggplot, 236
ggplot2::scale_x_log10, 228, 229
gmean (measures), 214
gpr (measures), 214
graphics::hist, 121
growingcv (resample), 252
gunpoint.task, 117
h2o::h2o.varimp(), 86
hasFunctionalFeatures, 118
hasLearnerProperties
  (LearnerProperties), 126
hasMeasureProperties
  (MeasureProperties), 213
hasProperties, 118
helpLearner, 82, 90, 92–95, 99, 119, 120, 127, 163, 164, 251, 263, 265, 267, 268
helpLearnerParam, 82, 90, 92–95, 99, 119, 119, 127, 163, 164, 251, 263, 265, 267, 268
hist, 53
holdout (resample), 252
hout (makeResampleDesc), 185
iauc.uno (measures), 214
ibrier (measures), 214
imputations, 120, 122, 123, 159, 160, 249
impute, 121, 122, 159, 160, 249
imputeConstant (imputations), 120
imputeHist (imputations), 120
imputeLearner (imputations), 120
imputeMax (imputations), 120
imputeMean (imputations), 120
imputeMedian (imputations), 120
imputeMin (imputations), 120
imputeMode (imputations), 120
imputeNormal (imputations), 120
imputeUniform (imputations), 120
irace::irace, 196, 200, 202, 280, 283
iris.task, 124
isFailureModel, 124
joinClassLevels, 125
kappa (measures), 214
kendalltau (measures), 214
kernlab::spam, 271
Learner (makeLearner), 160
learnerArgsToControl, 125
LearnerParam, 82
LearnerProperties, 82, 90, 92–95, 99, 119, 120, 126, 160, 163, 164, 251, 258, 263, 265, 267, 268
learners, 127, 161, 260
LearningCurveData, 62, 238
LearningCurveData
  (generateLearningCurveData), 61
list, 17, 30, 38, 46, 53, 59, 68, 69, 71–78, 80, 89, 93, 96, 103–105, 122, 123, 146, 149, 158, 159, 162, 166, 181, 196, 198, 199, 201, 203, 205, 206, 213, 257, 260, 263, 264, 266, 274, 280, 283, 284, 289, 290
list(), 38, 149
listFilterEnsembleMethods, 49, 59, 87, 127, 128, 153–155, 233
listFilterMethods, 48, 49, 58, 59, 87, 127, 128, 152–155, 233
listLearnerProperties, 129
listLearners, 127, 129
listMeasureProperties, 131
listMeasures, 132, 214
listTaskTypes, 132
logical, 32, 49, 88, 103, 108, 109, 155, 169, 213, 224, 245, 272, 278
logloss (measures), 214
lsr (measures), 214
lung.task, 133

mae (measures), 214
makeAggregation, 11, 133
makeBaggingWrapper, 134, 136, 140, 142, 147, 148, 150, 151, 155, 160, 170, 171, 173–175, 177, 180–182, 185, 191, 208, 210, 211, 223
makeClassificationViaRegressionWrapper, 135, 136, 140, 142, 147, 148, 150, 151, 155, 160, 170, 171, 173–175, 177, 180–182, 185, 191, 208, 210, 211
makeClasstask, 137
makeClusterTask, 138
makeConstantClassWrapper, 135, 136, 139, 142, 147, 148, 150, 151, 155, 160, 170, 171, 173–175, 177, 180–182, 185, 191, 208, 210, 211
makeCostMeasure, 20, 22, 25, 35, 140, 146, 166, 214, 217, 225, 262, 266
makeCostSensClassiﬁerWrapper, 135, 136, 140, 141, 142, 144, 145, 147, 148, 150, 151, 155, 160, 170, 171, 173–175, 177, 180–182, 185, 191, 208, 210, 211
makeCostSensRegressorWrapper, 135, 136, 140, 142, 144, 145, 147, 148, 150, 151, 155, 160, 170, 171, 173–175, 177, 180–182, 185, 191, 208, 210, 211, 223
makeCostSensTask, 142, 143, 145
makeCostSensWeightedPairsWrapper, 142, 144, 144
makeCustomResampledMeasure, 20, 22, 25, 35, 141, 145, 166, 217, 225, 262, 266
makeDummyFeaturesWrapper, 135, 136, 140, 142, 147, 147, 150, 151, 155, 160, 170, 171, 173–175, 177, 180–182, 185, 191, 208, 210, 211
makeExtractFDAFeatMethod, 38, 148, 150
makeExtractFDAFeatsWrapper, 38, 135, 136, 140, 142, 147–149, 149, 151, 155, 160, 170, 171, 173–175, 177, 180–182, 185, 191, 208, 210, 211
makeFeatSelControlExhaustive (FeatSelControl), 44
makeFeatSelControlGAM (FeatSelControl), 44
makeFeatSelControlRandom (FeatSelControl), 44
makeFeatSelControlSequential (FeatSelControl), 44
makeFeatSelWrapper, 13, 47, 85, 135, 136, 140, 142, 147, 148, 150, 151, 155, 160, 170, 171, 173–175, 177, 180–182, 185, 191, 208, 210, 211, 261
makeFilter, 49, 59, 87, 127, 128, 152, 153, 155, 233
makeFilterEnsemble, 49, 59, 87, 127, 128, 153, 155, 233
makeFixedHoldoutInstance, 157, 187
makeFunctionalData, 157, 275
makeImputeMethod, 121–123, 158, 160, 249
makeImputeWrappers, 121, 123, 135, 136, 140, 142, 147, 148, 150, 151, 154, 160, 170, 171, 173–175, 177, 180–182, 185, 191, 208, 210, 211, 233
makeLearners, 82, 90, 92–95, 99, 119, 120, 127, 163, 164, 251, 263, 265, 267, 268
makeMeasure, 20, 22, 25, 35, 141, 146, 165, 214, 217, 225, 262, 266
makeMultilabelBinaryRelevanceWrapper, makeModelMultiplexerParamSet, makeModelMultiplexer, makePreprocWrapper, makePrediction, makeMulticlassWrapper, makeOverBaggingWrapper, makeMultilabelBinaryRelevanceWrapper, makeMultilabelClassifierChainsWrapper, makeMultilabelDBRWrapper, makeMultilabelNestedStackingWrapper, makeMultilabelStackingWrapper, makeMultilabelTask, makeRegrTask, makeRemoveConstantFeaturesWrapper, makeRegrTask, makeResampleDesc, makeStackedLearner, makeSurvTask, makeTuneControlCMAES, makeTuneControlDesign, makeTuneControlGenSA, makeTuneControlGrid, makeTuneControlIRace, makeTuneControlMBO, makePreprocWrapperCaret, makeRemakeConstantFeaturesWrapper, makeResampleDesc, makeResampleInstance, makeRLearner (RLearner), makeRLearner.classif.fdausc.glm, makeRLearner.classif.fdausc.kernel, makeRLearner.classif.fdausc.np, makeRLearnerClassif (RLearner), makeRLearnerCluster (RLearner), makeRLearnerCostSens (RLearner), makeRLearnerMultilabel (RLearner), makeRLearnerRegr (RLearner), makeSMOTEWrapper, makeSurvTask.
INDEX

measureAUC (measures), 214
measureAUNP (measures), 214
measureAUNU (measures), 214
measureBAC (measures), 214
measureBER (measures), 214
measureBrier (measures), 214
measureBrierScaled (measures), 214
measureEXPVAR (measures), 214
measureF1 (measures), 214
measureFDR (measures), 214
measureFN (measures), 214
measureFNR (measures), 214
measureFP (measures), 214
measureFPR (measures), 214
measureGMEAN (measures), 214
measureGPR (measures), 214
measureKAPPA (measures), 214
measureKendallTau (measures), 214
measureLogloss (measures), 214
measureLSR (measures), 214
measureMAE (measures), 214
measureMAPE (measures), 214
measureMCC (measures), 214
measureMEDAE (measures), 214
measureMEDSE (measures), 214
measureMMCE (measures), 214
measureMSE (measures), 214
measureMSLE (measures), 214
measureMulticlassBrier (measures), 214
measureMultilabelACC (measures), 214
measureMultilabelF1 (measures), 214
measureMultilabelHamloss (measures), 214
measureMultilabelPPV (measures), 214
measureMultilabelSubset01 (measures), 214
measureMultilabelTPR (measures), 214
measureNPV (measures), 214
measurePPV (measures), 214
MeasureProperties (measures), 213
measureQSR (measures), 214
measureRAE (measures), 214
measureRMSE (measures), 214
measureRMSLE (measures), 214
measureRRSE (measures), 214
measureRSQ (measures), 214
measures, 20–22, 25, 35, 141, 146, 165, 166, 214, 225, 252, 262, 266
measureSAE (measures), 214
measures, 169, 197, 198, 200, 202, 204, 204, 207, 208, 280, 286, 290
makeTuneControlRandom, 97, 98, 103, 117, 167, 169, 197, 198, 200, 202, 204, 206, 208, 280, 286, 290
makeTuneMultiCritControlGrid (TuneMultiCritControl), 280
makeTuneMultiCritControlMBO (TuneMultiCritControl), 280
makeTuneMultiCritControlNSGA2 (TuneMultiCritControl), 280
makeTuneMultiCritControlRandom (TuneMultiCritControl), 280
makeWeightedClassesWrapper, 135, 136, 140, 142, 143, 147, 148, 150, 151, 155, 160, 170, 171, 173, 174, 176, 177, 180–182, 185, 191, 208, 210, 210
makeWrappedModel, 212
mape (measures), 214
matrix, 25, 26, 83, 140, 216, 284
mcc (measures), 214
mco::nsga2, 281
mcp (measures), 214
mean, 141
meancosts (measures), 214
Measure (makeMeasure), 165
measureACC (measures), 214
measureAU1P (measures), 214
measureAU1U (measures), 214
measureSpearmanRho (measures), 214
measureSSE (measures), 214
measureSSR (measures), 214
measureTN (measures), 214
measureTNR (measures), 214
measureTP (measures), 214
measureTPR (measures), 214
measureWKAPPA (measures), 214
medae (measures), 214
mse (measures), 214
mergeBenchmarkResults, 217
mergeSmallFactorLevels, 23, 28, 33, 218, 222, 251, 273, 274
mlbench::BostonHousing, 18
mlbench::BreastCancer, 15
mlbench::PimaIndiansDiabetes, 225
mlbench::Sonar, 271
mlr (mlr-package), 8
mlr-package, 8
mlrFamilies, 219
mlrMBO::makeMBOControl, 205, 283
mlrMBO::makeMBOControler, 204, 283
mlrMBO::mbo, 204, 205, 283
mlrMBO::mboContinue, 205, 283
mlrMBO::MBOControl, 205, 283
mlrMBO::OptResult, 205, 283
mmce (measures), 214
model.matrix, 28
ModelMultiplexer, 167, 169
ModelMultiplexer
(makeModelMultiplexer), 167
mse (measures), 214
msle (measures), 214
mtcars.task, 220
multiclass.aulp (measures), 214
multiclass.aulu (measures), 214
multiclass.aunp (measures), 214
multiclass.aunu (measures), 214
multiclass.brier (measures), 214
multilabel.acc (measures), 214
multilabel.f1 (measures), 214
multilabel.hamloss (measures), 214
multilabel.ppv (measures), 214
multilabel.subset01 (measures), 214
multilabel.tpr (measures), 214
MultilabelTask, 138, 139, 144, 184, 195, 276
MultilabelTask (makeMultilabelTask), 177
normalizeFeatures, 23, 28, 33, 218, 221,
228, 231, 232, 238, 240–243, 248
plotFilterValues, 49, 54, 55, 58, 59, 63, 66, 67, 87, 127, 128, 133, 155, 232
plotHyperParsEffect, 60, 61, 233
plotLearnerPrediction, 236
plotLearningCurve, 30, 63, 227, 228, 230–232, 238, 240–243
plotPartialDependence, 30, 64, 66, 227, 228, 230–232, 238, 240–243, 248
plotResiduals, 30, 227, 228, 230–232, 238, 240, 242, 243
plotROCCurves, 30, 67, 227, 228, 230–232, 238, 240, 241, 243
plotTuneMultiCritResult, 244, 283, 288
PMCMRplus::frdAllPairsNemenyiTest, 50, 51, 55
ppv (measures), 214
predict.WrappedModel, 13, 83, 101, 102, 245, 267, 268, 278
Prediction, 13, 19, 21, 34, 53, 66, 83, 96, 98–100, 102, 105, 134, 146, 166, 224, 240, 245, 256, 268, 289
predictLearner, 246
print.ConfusionMatrix (calculateConfusionMatrix), 19
print.ROCMeasures (calculateROCMeasures), 20
qsr (measures), 214
rae (measures), 214
randomForest::importance(), 86
ranger::importance(), 87
ranger::ranger, 162
ranger::ranger(), 87
rank, 228
reduceBatchmarkResults, 15, 17, 18, 26, 51, 52, 55, 68, 69, 71–77, 79, 81, 227–229, 232, 247
reextractFDAFeatures, 37, 149, 248
RegrTask, 35, 138, 139, 144, 179, 195, 276
RegrTask (makeRegrTask), 183
reimpute, 121–123, 159, 160, 249
removeConstantFeatures, 23, 28, 33, 184, 218, 222, 250, 273, 274
removeHyperPars, 82, 90, 92–95, 99, 119, 120, 127, 163, 164, 251, 263, 265, 267, 268
repcv (resample), 252
resample, 10, 29, 60, 67, 75, 76, 104–107, 187, 189, 252, 256, 257, 285, 288
ResampleDesc, 14, 16, 34, 62, 150, 187, 188, 193, 202, 208, 252, 255, 261, 285, 287
ResampleDesc (makeResampleDesc), 185
ResampleInstance, 16, 32, 62, 150, 157, 185, 189, 202, 208, 252, 255, 261, 285, 287
ResampleInstance (makeResampleInstance), 188
ResamplePrediction, 10, 34, 67, 76, 105–107, 146, 187, 189, 256, 257
ResampleResult, 10, 14, 16, 17, 29, 43, 53, 60, 66, 67, 97, 98, 100, 103–107, 187, 189, 256, 257
RLearner, 246, 258, 260, 279
RLearnerClassif, 260
RLearnerClassif (RLearner), 258
RLearnerCluster, 260
RLearnerCluster (RLearner), 258
RLearnerMultilabel, 260
RLearnerMultilabel (RLearner), 258
RLearnerRegr, 260
RLearnerRegr (RLearner), 258
RLearnerSurv, 260
RLearnerSurv (RLearner), 258
rmse (measures), 214
rmsle (measures), 214
rpart::rpart, 91
rrse (measures), 214
rsq (measures), 214
sae (measures), 214
sd, 273
selectFeatures, 12, 13, 44, 47, 85, 150, 151, 260
setAggregation, 20, 22, 25, 35, 134, 141, 146, 166, 187, 217, 225, 262, 266
setHyperPars, 82, 90, 92–95, 99, 119, 120, 127, 163, 164, 251, 263, 265, 267,
setHyperPars2, 264
setLearnerId, 82, 90, 92–95, 99, 119, 120, 127, 163, 164, 251, 263, 264, 265, 267, 268
setMeasurePars, 20, 22, 25, 35, 141, 146, 166, 214, 217, 225, 262, 266
setMeasurePars(), 166
setPredictThreshold, 13, 82, 90, 92–95, 99, 101, 102, 119, 120, 127, 163, 164, 245, 251, 263, 265, 266, 268
setPredictType, 13, 82, 90, 92–95, 99, 101, 102, 119, 120, 127, 134, 163, 164, 245, 251, 263, 265, 267, 267
setThreshold, 160, 161, 171, 245, 266, 267, 268
silhouette (measures), 214
simplifyMeasureNames, 269
smote, 180, 190, 210, 223, 270
sonar.task, 271
spam.task, 271
spatial.task, 271
spearmanrho (measures), 214
sse (measures), 214
ssr (measures), 214
stats::fft, 39
stats::friedman.test, 50–52
subsample (resample), 252
subsetTask, 108–116, 272, 275
summarizeColumns, 23, 28, 33, 218, 222, 251, 273, 274
summarizeLevels, 23, 28, 33, 218, 222, 251, 273, 274
summary, 273
survival::lung, 133
survival::Surv, 88, 110, 115
SurvTask, 138, 139, 144, 179, 184, 276
SurvTask (makeSurvTask), 194

test.join (aggregations), 11
test.max (aggregations), 11
test.mean, 166
test.mean (aggregations), 11
test.median (aggregations), 11
test.min (aggregations), 11
test.range (aggregations), 11
test.rmse (aggregations), 11
test.sd (aggregations), 11
test.sum (aggregations), 11
testgroup.mean (aggregations), 11
testgroup.sd (aggregations), 11
TH.data::wpbc, 290
ThreshVsPerfData, 67, 241, 243
ThreshVsPerfData (generateThreshVsPerfData), 66
timembooth (measures), 214
timetrain (measures), 214	n (measures), 214
tn (measures), 214
tnrr (measures), 214
tp (measures), 214
tpr (measures), 214
train, 35, 64, 91, 212, 245, 255, 277
train(), 86
train.max (aggregations), 11
train.mean (aggregations), 11
train.median (aggregations), 11
train.min (aggregations), 11
train.range (aggregations), 11
train.rmse (aggregations), 11
train.sd (aggregations), 11
train.sum (aggregations), 11
trainLearner, 109, 278
tsfeatures::tsfeatures(), 41
TuneControl, 97, 98, 103, 117, 167, 169, 197, 198, 200, 202, 203, 206–208, 279, 284, 285, 289, 290
TuneControlCMAES, 196
TuneControlCMAES (makeTuneControlCMAES), 195
TuneControlDesign, 198
TuneControlDesign (makeTuneControlDesign), 197
TuneControlGenSA, 200
TuneControlGenSA
(makeTuneControlGenSA), 198
TuneControlGrid, 202
TuneControlGrid (makeTuneControlGrid), 200
TuneControlIrace, 203
TuneControlIrace
  (makeTuneControlIrace), 202
TuneControlMBO, 205
TuneControlMBO (makeTuneControlMBO), 204
TuneControlRandom, 207
TuneControlRandom
  (makeTuneControlRandom), 206
TuneMultiCritControl, 244, 280, 283, 287, 288
TuneMultiCritControlGrid, 283
TuneMultiCritControlGrid
  (TuneMultiCritControl), 280
TuneMultiCritControlMBO, 283
TuneMultiCritControlMBO
  (TuneMultiCritControl), 280
TuneMultiCritControlNSGA2, 283
TuneMultiCritControlNSGA2
  (TuneMultiCritControl), 280
TuneMultiCritControlRandom, 283
TuneMultiCritControlRandom
  (TuneMultiCritControl), 280
TuneMultiCritResult, 244, 284, 288
tuneParams, 60, 81, 97, 98, 103, 117, 167, 169, 197, 198, 200, 202, 204, 206–208, 280, 284, 290
tuneParamsMultiCrit, 244, 283, 284, 287
TuneResult, 60, 80, 117, 285, 288
tuneThreshold, 46, 97, 98, 103, 117, 167, 169, 171, 196–208, 280, 283, 286, 289
undersample, 209
undersample (oversample), 222
wavelets::dwt, 42
wkappa (measures), 214
wpbc.task, 290
WrappedModel, 35, 43, 64, 74, 84–87, 91, 99, 107, 116, 124, 166, 213, 224, 245, 246, 255, 257, 278, 289
WrappedModel (makeWrappedModel), 212
yeast.task, 290