Package: llama (via r-universe)

October 30, 2026

Type Package

Title Leveraging Learning to Automatically Manage Algorithms

Version 0.10.1

Date 2021-03-16

Author Lars Kotthoff [aut,cre], Bernd Bischl [aut], Barry Hurley [ctb], Talal Rahwan [ctb], Damir Pulatov [ctb]

Maintainer Lars Kotthoff <larsko@uwyo.edu>

Description Provides functionality to train and evaluate algorithm selection models for portfolios.

Depends R (>= 4.0), mlr (>= 2.5)

Imports rJava, parallelMap, ggplot2, checkmate, BBmisc, plyr, data.table

Suggests testthat, ParamHelpers

License BSD_3_clause + file LICENSE

URL https://bitbucket.org/lkotthoff/llama

NeedsCompilation no

Repository https://cran.r-universe.dev

Date/Publication 2021-03-16 21:40:12 UTC

RemoteUrl https://github.com/cran/llama

RemoteRef HEAD

RemoteSha f95206aaeb5c492db579878443655dbb526c254e

Contents

llama-package 2
analysis 4
bsFolds 5
classify 6
classifyPairs 8
cluster 10

cvFolds . . . 13
helpers . . . 14
imputeCensored . . . 15
input . . . 16
misc . . . 18
misclassificationPenalties . . . 20
normalize . . . 21
parscores . . . 22
plot . . . 24
regression . . . 25
regressionPairs . . . 28
satsolvers . . . 30
successes . . . 31
trainTest . . . 32
tune . . . 33

Index **36**

`llama-package` *Leveraging Learning to Automatically Manage Algorithms*

Description

Leveraging Learning to Automatically Manage Algorithms provides functionality to read and process performance data for algorithms, facilitate building models that predict which algorithm to use in which scenario and ways of evaluating them.

Details

The package provides functions to read performance data, build performance models that enable selection of algorithms (using external machine learning functions) and evaluate those models.

Data is input using `input` and can then be used to learn performance models. There are currently four main ways to create models. Classification (`classify`) creates a single machine learning model that predicts the algorithm to use as a label. Classification of pairs of algorithms (`classifyPairs`) creates a classification model for each pair of algorithms that predicts which one is better and aggregates these predictions to determine the best overall algorithm. Clustering (`cluster`) clusters the problems to solve and assigns the best algorithm to each cluster. Regression (`regression`) trains a separate or single model (depending on the types of features available) for all algorithms, predicts the performance on a problem independently and chooses the algorithm with the best predicted performance. Regression of pairs of algorithms (`regressionPairs`) is similar to `classifyPairs`, but predicts the performance difference between each pair of algorithms. Similar to `regression`, `regressionPairs` can also build a single model for all pairs of algorithms, depending on the types of features available to the function.

Various functions to split the data into training and test set(s) and to evaluate the performance of the learned models are provided.

LLAMA uses the mlr package to access the implementation of machine learning algorithms in R.

The model building functions are using the `parallelMap` package to parallelize across the data partitions (e.g. cross-validation folds) with level "llama.fold" and "llama.tune" for tuning. By default, everything is run sequentially. By loading a suitable backend (e.g. through `parallelStartSocket(2)` for parallelization across 2 CPUs using sockets), the model building will be parallelized automatically and transparently. Note that this does *not* mean that all machine learning algorithms used for building models can be parallelized safely. For functions that are not thread safe, use `parallelStartSocket` to run in separate processes.

Author(s)

Lars Kotthoff, Bernd Bischl

contributions by Barry Hurley, Talal Rahwan, Damir Pulatov

Maintainer: Lars Kotthoff <larsko@uwyo.edu>

References

Kotthoff, L. (2013) LLAMA: Leveraging Learning to Automatically Manage Algorithms. *arXiv:1306.1031*.

Kotthoff, L. (2014) Algorithm Selection for Combinatorial Search Problems: A survey. *AI Magazine*.

Examples

```
if(Sys.getenv("RUN_EXPENSIVE") == "true") {
data(satsolvers)
folds = cvFolds(satsolvers)

model = classify(classifier=makeLearner("classif.J48"), data=folds)
# print the total number of successes
print(sum(successes(folds, model)))
# print the total misclassification penalty
print(sum(misclassificationPenalties(folds, model)))
# print the total PAR10 score
print(sum(parscores(folds, model)))

# number of total successes for virtual best solver for comparison
print(sum(successes(satsolvers, vbs, addCosts = FALSE)))

# print predictions on the entire data set
print(model$predictor(subset(satsolvers$data, TRUE, satsolvers$features)))

# train a regression model
model = regression(regressor=makeLearner("regr.lm"), data=folds)
# print the total number of successes
print(sum(successes(folds, model)))
}
```

`analysis` *Analysis functions*

Description

Functions for analysing portfolios.

Usage

```
contributions(data = NULL)
```

Arguments

`data`	the data to use. The structure returned by `input`.

Details

`contributions` analyses the marginal contributions of the algorithms in the portfolio to its overall performance. More specifically, the Shapley value for a specific algorithm is computed as the "value" of the portfolio with the algorithm minus the "value" without the algorithm. This is done over all possible portfolio compositions.

It is automatically determined whether the performance value is to be minimised or maximised.

Value

A table listing the Shapley values for each algorithm in the portfolio. The higher the value, the more the respective algorithm contributes to the overall performance of the portfolio.

Author(s)

Lars Kotthoff

References

Rahwan, T., Michalak, T. (2013) A Game Theoretic Approach to Measure Contributions in Algorithm Portfolios. *Technical Report RR-13-11, University of Oxford*.

Examples

```
if(Sys.getenv("RUN_EXPENSIVE") == "true") {
data(satsolvers)

contributions(satsolvers)
}
```

bsFolds *Bootstrapping folds*

Description

Take data produced by `input` and amend it with (optionally) stratified folds determined through bootstrapping.

Usage

```
bsFolds(data, nfolds = 10L, stratify = FALSE)
```

Arguments

`data`	the data to use. The structure returned by `input`.
`nfolds`	the number of folds. Defaults to 10.
`stratify`	whether to stratify the folds. Makes really only sense for classification models. Defaults to `FALSE`.

Details

Partitions the data set into folds. Stratification, if requested, is done by the best algorithm, i.e. the one with the best performance. The distribution of the best algorithms in each fold will be approximately the same. For each fold, the training index set is assembled through .632 bootstrap. The remaining indices are used for testing. There is no guarantee on the sizes of either sets. The sets of indices are added to the original data set and returned.

If the data set has train and test partitions already, they are overwritten.

Value

`train`	a list of index sets for training.
`test`	a list of index sets for testing.
`...`	the original members of `data`. See `input`.

Author(s)

Lars Kotthoff

See Also

`cvFolds`, `trainTest`

Examples

```
data(satsolvers)
folds = bsFolds(satsolvers)

# use 5 folds instead of the default 10
folds5 = bsFolds(satsolvers, 5L)

# stratify
foldsU = bsFolds(satsolvers, stratify=TRUE)
```

---

classify *Classification model*

Description

Build a classification model that predicts the algorithm to use based on the features of the problem.

Usage

```
classify(classifier = NULL, data = NULL,
    pre = function(x, y=NULL) { list(features=x) },
    save.models = NA, use.weights = TRUE)
```

Arguments

classifier	the mlr classifier to use. See examples. The argument can also be a list of such classifiers.
data	the data to use with training and test sets. The structure returned by one of the partitioning functions.
pre	a function to preprocess the data. Currently only normalize. Optional. Does nothing by default.
save.models	Whether to serialize and save the models trained during evaluation of the model. If not NA, will be used as a prefix for the file name.
use.weights	Whether to use instance weights if supported. Default TRUE.

Details

classify takes the training and test sets in data and processes it using pre (if supplied). classifier is called to induce a classifier. The learned model is used to make predictions on the test set(s).

The evaluation across the training and test sets will be parallelized automatically if a suitable backend for parallel computation is loaded. The parallelMap level is "llama.fold".

If the given classifier supports case weights and use.weights is TRUE, the performance difference between the best and the worst algorithm is passed as a weight for each instance.

If a list of classifiers is supplied in classifier, ensemble classification is performed. That is, the models are trained and used to make predictions independently. For each instance, the final

prediction is determined by majority vote of the predictions of the individual models – the class that occurs most often is chosen. If the list given as `classifier` contains a member `.combine` that is a function, it is assumed to be a classifier with the same properties as the other ones and will be used to combine the ensemble predictions instead of majority voting. This classifier is passed the original features and the predictions of the classifiers in the ensemble.

If the prediction of a stacked learner is `NA`, the prediction will be `NA` for the score.

If `save.models` is not `NA`, the models trained during evaluation are serialized into files. Each file contains a list with members `model` (the mlr model), `train.data` (the mlr task with the training data), and `test.data` (the data frame with the test data used to make predictions). The file name starts with `save.models`, followed by the ID of the machine learning model, followed by "combined" if the model combines predictions of other models, followed by the number of the fold. Each model for each fold is saved in a different file.

Value

`predictions` — a data frame with the predictions for each instance and test set. The columns of the data frame are the instance ID columns (as determined by `input`), the algorithm, the score of the algorithm, and the iteration (e.g. the number of the fold for cross-validation). More than one prediction may be made for each instance and iteration. The score corresponds to the number of classifiers that predicted the respective algorithm, or the sum of probabilities that this classifier was the best. If stacking is used, the score corresponds to the output of the stacked classifier.

`predictor` — a function that encapsulates the classifier learned on the *entire* data set. Can be called with data for the same features with the same feature names as the training data to obtain predictions in the same format as the `predictions` member.

`models` — the list of models trained on the *entire* data set. This is meant for debugging/inspection purposes and does not include any models used to combine predictions of individual models.

Author(s)

Lars Kotthoff

References

Kotthoff, L., Miguel, I., Nightingale, P. (2010) Ensemble Classification for Constraint Solver Configuration. *16th International Conference on Principles and Practices of Constraint Programming*, 321–329.

See Also

`classifyPairs`, `cluster`, `regression`, `regressionPairs`

Examples

```
if(Sys.getenv("RUN_EXPENSIVE") == "true") {
data(satsolvers)
folds = cvFolds(satsolvers)
```

```
res = classify(classifier=makeLearner("classif.J48"), data=folds)
# the total number of successes
sum(successes(folds, res))
# predictions on the entire data set
res$predictor(satsolvers$data[satsolvers$features])

res = classify(classifier=makeLearner("classif.svm"), data=folds)

# use probabilities instead of labels
res = classify(classifier=makeLearner("classif.randomForest", predict.type = "prob"), data=folds)

# ensemble classification
rese = classify(classifier=list(makeLearner("classif.J48"),
                                makeLearner("classif.IBk"),
                                makeLearner("classif.svm")),
                data=folds)

# ensemble classification with a classifier to combine predictions
rese = classify(classifier=list(makeLearner("classif.J48"),
                                makeLearner("classif.IBk"),
                                makeLearner("classif.svm"),
                                .combine=makeLearner("classif.J48")),
                data=folds)
}
```

`classifyPairs` *Classification model for pairs of algorithms*

Description

Build a classification model for each pair of algorithms that predicts which one is better based on the features of the problem. Predictions are aggregated to determine the best overall algorithm.

Usage

```
classifyPairs(classifier = NULL, data = NULL,
    pre = function(x, y=NULL) { list(features=x) }, combine = NULL,
    save.models = NA, use.weights = TRUE)
```

Arguments

`classifier`	the mlr classifier to use. See examples.
`data`	the data to use with training and test sets. The structure returned by one of the partitioning functions.
`pre`	a function to preprocess the data. Currently only `normalize`. Optional. Does nothing by default.

combine	The classifier function to predict the overall best algorithm given the predictions for pairs of algorithms. Optional. By default, the overall best algorithm is determined by majority vote.
save.models	Whether to serialize and save the models trained during evaluation of the model. If not NA, will be used as a prefix for the file name.
use.weights	Whether to use instance weights if supported. Default TRUE.

Details

classifyPairs takes the training and test sets in data and processes it using pre (if supplied). classifier is called to induce a classifier for each pair of algorithms to predict which one is better. If combine is not supplied, the best overall algorithm is determined by majority vote. If it is supplied, it is assumed to be a classifier with the same properties as the other one. This classifier is passed the original features and the predictions for each pair of algorithms.

Which algorithm is better of a pair is determined by comparing their performance scores. Whether a lower performance number is better or not is determined by what was specified when the LLAMA data frame was created.

The evaluation across the training and test sets will be parallelized automatically if a suitable backend for parallel computation is loaded. The parallelMap level is "llama.fold".

If the given classifier supports case weights and use.weights is TRUE, the performance difference between the best and the worst algorithm is passed as a weight for each instance.

If all predictions of an underlying machine learning model are NA, it will count as 0 towards the score.

Training this model can take a very long time. Given n algorithms, choose(n, 2) models are trained and evaluated. This is significantly slower than the other approaches that train a single model or one for each algorithm.

If save.models is not NA, the models trained during evaluation are serialized into files. Each file contains a list with members model (the mlr model), train.data (the mlr task with the training data), and test.data (the data frame with the test data used to make predictions). The file name starts with save.models, followed by the ID of the machine learning model, followed by "combined" if the model combines predictions of other models, followed by the number of the fold. Each model for each fold is saved in a different file.

Value

predictions	a data frame with the predictions for each instance and test set. The columns of the data frame are the instance ID columns (as determined by input), the algorithm, the score of the algorithm, and the iteration (e.g. the number of the fold for cross-validation). More than one prediction may be made for each instance and iteration. The score corresponds to the number of times the respective algorithm was predicted to be better. If stacking is used, only the best algorithm for each algorithm-instance pair is predicted with a score of 1.
predictor	a function that encapsulates the classifier learned on the *entire* data set. Can be called with data for the same features with the same feature names as the training data to obtain predictions in the same format as the predictions member.

`models` the models for each pair of algorithms trained on the *entire* data set. This is meant for debugging/inspection purposes and does not include any models used to combine predictions of individual models.

Author(s)

Lars Kotthoff

References

Xu, L., Hutter, F., Hoos, H. H., Leyton-Brown, K. (2011) Hydra-MIP: Automated Algorithm Configuration and Selection for Mixed Integer Programming. *RCRA Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion*, 16–30.

See Also

`classify`, `cluster`, `regression`, `regressionPairs`

Examples

```
if(Sys.getenv("RUN_EXPENSIVE") == "true") {
data(satsolvers)
folds = cvFolds(satsolvers)

res = classifyPairs(classifier=makeLearner("classif.J48"), data=folds)
# the total number of successes
sum(successes(folds, res))
# predictions on the entire data set
res$predictor(satsolvers$data[satsolvers$features])

# use probabilities instead of labels
res = classifyPairs(classifier=makeLearner("classif.randomForest",
                                predict.type = "prob"), data=folds)

# combine predictions using J48 induced classifier instead of majority vote
res = classifyPairs(classifier=makeLearner("classif.J48"),
                    data=folds,
                    combine=makeLearner("classif.J48"))
}
```

`cluster` *Cluster model*

Description

Build a cluster model that predicts the algorithm to use based on the features of the problem.

Usage

```
cluster(clusterer = NULL, data = NULL,
    bestBy = "performance",
    pre = function(x, y=NULL) { list(features=x) },
    save.models = NA)
```

Arguments

clusterer	the mlr clustering function to use. See examples. The argument can also be a list of such functions.
data	the data to use with training and test sets. The structure returned by one of the partitioning functions.
bestBy	the criteria by which to determine the best algorithm in a cluster. Can be one of "performance", "count", "successes". Optional. Defaults to "performance".
pre	a function to preprocess the data. Currently only normalize. Optional. Does nothing by default.
save.models	Whether to serialize and save the models trained during evaluation of the model. If not NA, will be used as a prefix for the file name.

Details

cluster takes data and processes it using pre (if supplied). clusterer is called to cluster the data. For each cluster, the best algorithm is identified according to the criteria given in bestBy. If bestBy is "performance", the best algorithm is the one with the best overall performance across all instances in the cluster. If it is "count", the best algorithm is the one that has the best performance most often. If it is "successes", the best algorithm is the one with the highest number of successes across all instances in the cluster. The learned model is used to cluster the test data and predict algorithms accordingly.

The evaluation across the training and test sets will be parallelized automatically if a suitable backend for parallel computation is loaded. The parallelMap level is "llama.fold".

If a list of clusterers is supplied in clusterer, ensemble clustering is performed. That is, the models are trained and used to make predictions independently. For each instance, the final prediction is determined by majority vote of the predictions of the individual models – the class that occurs most often is chosen. If the list given as clusterer contains a member .combine that is a function, it is assumed to be a classifier with the same properties as classifiers given to classify and will be used to combine the ensemble predictions instead of majority voting. This classifier is passed the original features and the predictions of the classifiers in the ensemble.

If all predictions of an underlying machine learning model are NA, the prediction will be NA for the algorithm and -Inf for the score if the performance value is to be maximised, Inf otherwise.

If save.models is not NA, the models trained during evaluation are serialized into files. Each file contains a list with members model (the mlr model), train.data (the mlr task with the training data), and test.data (the data frame with the test data used to make predictions). The file name starts with save.models, followed by the ID of the machine learning model, followed by "combined" if the model combines predictions of other models, followed by the number of the fold. Each model for each fold is saved in a different file.

Value

`predictions` a data frame with the predictions for each instance and test set. The columns of the data frame are the instance ID columns (as determined by `input`), the algorithm, the score of the algorithm, and the iteration (e.g. the number of the fold for cross-validation). More than one prediction may be made for each instance and iteration. The score corresponds to the cumulative performance value for the algorithm of the cluster the instance was assigned to. That is, if `bestBy` is "performance", it is the sum of the performance over all training instances. If `bestBy` is "count", the score corresponds to the number of training instances that the respective algorithm was the best on, and if it is "successes" it corresponds to the number of training instances where the algorithm was successful. If more than one clustering algorithm is used, the score corresponds to the sum of all instances across all clusterers. If stacking is used, the prediction is simply the best algorithm with a score of 1.

`predictor` a function that encapsulates the model learned on the *entire* data set. Can be called with data for the same features with the same feature names as the training data to obtain predictions in the same format as the `predictions` member.

`models` the list of models trained on the *entire* data set. This is meant for debugging/inspection purposes and does not include any models used to combine predictions of individual models.

Author(s)

Lars Kotthoff

See Also

`classify`, `classifyPairs`, `regression`, `regressionPairs`

Examples

```
if(Sys.getenv("RUN_EXPENSIVE") == "true") {
data(satsolvers)
folds = cvFolds(satsolvers)

res = cluster(clusterer=makeLearner("cluster.XMeans"), data=folds, pre=normalize)
# the total number of successes
sum(successes(folds, res))
# predictions on the entire data set
res$predictor(satsolvers$data[satsolvers$features])

# determine best by number of successes
res = cluster(clusterer=makeLearner("cluster.XMeans"), data=folds,
    bestBy="successes", pre=normalize)
sum(successes(folds, res))

# ensemble clustering
rese = cluster(clusterer=list(makeLearner("cluster.XMeans"),
    makeLearner("cluster.SimpleKMeans"), makeLearner("cluster.EM")),
    data=folds, pre=normalize)
```

```
# ensemble clustering with a classifier to combine predictions
rese = cluster(clusterer=list(makeLearner("cluster.XMeans"),
    makeLearner("cluster.SimpleKMeans"), makeLearner("cluster.EM"),
    .combine=makeLearner("classif.J48")), data=folds, pre=normalize)
}
```

---

cvFolds *Cross-validation folds*

Description

Take data produced by `input` and amend it with (optionally) stratified folds for cross-validation.

Usage

```
cvFolds(data, nfolds = 10L, stratify = FALSE)
```

Arguments

`data`	the data to use. The structure returned by `input`.
`nfolds`	the number of folds. Defaults to 10. If -1 is given, leave-one-out cross-validation folds are produced.
`stratify`	whether to stratify the folds. Makes really only sense for classification models. Defaults to `FALSE`.

Details

Partitions the data set into folds. Stratification, if requested, is done by the best algorithm, i.e. the one with the best performance. The distribution of the best algorithms in each fold will be approximately the same. The folds are assembled into training and test sets by combining $n-1$ folds for training and using the remaining fold for testing. The sets of indices are added to the original data set and returned.

If the data set has train and test partitions already, they are overwritten.

Value

`train`	a list of index sets for training.
`test`	a list of index sets for testing.
`...`	the original members of `data`. See `input`.

Author(s)

Lars Kotthoff

See Also

`bsFolds`, `trainTest`

Examples

```
data(satsolvers)
folds = cvFolds(satsolvers)

# use 5 folds instead of the default 10
folds5 = cvFolds(satsolvers, 5L)

# stratify
foldsU = cvFolds(satsolvers, stratify=TRUE)
```

`helpers` *Helpers*

Description

S3 helper methods.

Usage

```
## S3 method for class 'llama.data'
print(x, ...)
## S3 method for class 'llama.model'
print(x, ...)
## S3 method for class 'classif.constant'
makeRLearner()
## S3 method for class 'classif.constant'
predictLearner(.learner, .model, .newdata, ...)
## S3 method for class 'classif.constant'
trainLearner(.learner, .task, .subset, .weights, ...)
```

Arguments

`x`	the object to print.
`.learner`	learner.
`.model`	model.
`.newdata`	new data.
`.task`	task.
`.subset`	subset.
`.weights`	weights.
`...`	ignored.

Author(s)

Lars Kotthoff

`imputeCensored` *Impute censored values*

Description

Impute the performance values that are censored, i.e. for which the respective algorithm was not successful.

Usage

```
imputeCensored(data = NULL, estimator = makeLearner("regr.lm"),
    epsilon = 0.1, maxit = 1000)
```

Arguments

`data`	the data to check for censored values to impute. The structure returned by `input`.
`estimator`	the mlr regressor to use to impute the censored values.
`epsilon`	the convergence criterion. Default 0.1.
`maxit`	the maximum number of iterations. Default 1000.

Details

The function checks for each algorithm if there are censored values by checking for which problem instances the algorithm was not successful. It trains a model to predict the performance value for those instances using the given estimator based on the performance values of the instances where the algorithm was successful and the problem features. It then uses the results of this initial prediction to train a new model on the entire data and predict the performance values for those problems where the algorithm was successful again. This process is repeated until the maximum difference between predictions in two successive iterations is less than `epsilon` or more than `maxit` iterations have been performed.

It is up to the user to check whether the imputed values make sense. In particular, for solver runtime data and timeouts one would expect that the imputed values are above the timeout threshold, indicating at what time the algorithms that have timed out would have solved the problem. No effort is made to enforce such application-specific constraints.

Value

The data structure with imputed censored values. The original data is saved in the `original_data` member.

Author(s)

Lars Kotthoff

References

Josef Schmee and Gerald J. Hahn (1979) A Simple Method for Regression Analysis with Censored Data. *Technometrics* 21, no. 4, 417-432.

Examples

```
if(Sys.getenv("RUN_EXPENSIVE") == "true") {
data(satsolvers)
imputed = imputeCensored(satsolvers)
}
```

`input` *Read data*

Description

Reads performance data that can be used to train and evaluate models.

Usage

```
input(features, performances, algorithmFeatures = NULL, successes = NULL, costs = NULL,
  extra = NULL, minimize = T, perfcol = "performance")
```

Arguments

`features`	data frame that contains the feature values for each problem instance and a non-empty set of ID columns.
`algorithmFeatures`	data frame that contains the feature values for each algorithm and a non-empty set of algorithm ID columns. Optional.
`performances`	data frame that contains the performance values for each problem instance and a non-empty set of ID columns.
`successes`	data frame that contains the success values (`TRUE`/`FALSE`) for each algorithm on each problem instance and a non-empty set of ID columns. The names of the columns in this data set should be the same as the names of the columns in `performances`. Optional.
`costs`	either a single number, a data frame or a list that specifies the cost of the features. If a number is specified, it is assumed to denote the cost for all problem instances (i.e. the cost is always the same). If a data frame is given, it is assumed to have one column for each feature with the same name as the feature where each value gives the cost and a non-empty set of ID columns. If a list is specified, it is assumed to have a member `groups` that specifies which features belong to which group and a member `values` that is a data frame in the same format as before. Optional.

extra	data frame containing any extra information about the instances and a non-empty set of ID columns. This is not used in modelling, but can be used e.g. for visualisation. Optional.
minimize	whether the minimum performance value is best. Default true.
perfcol	name of the column that stores performance values when algorithm features are provided. Default performance.

Details

input takes a list of data frames and processes them as follows. The feature and performance data are joined by looking for common column names in the two data frames (usually an ID of the problem instance). For each problem, the best algorithm according to the given performance data is computed. If more than one algorithm has the best performance, all of them are returned.

The data frame for algorithmic features is optional. When it is provided, the existing data is joined by algorithm names. The final data frame is reshaped into 'long' format.

The data frame that describes whether an algorithm was successful on a problem is optional. If parscores or successes are to be used to evaluate the learned models, this argument is required however and will lead to error messages if not supplied.

Similarly, feature costs are optional.

If successes is given, it is used to determine the best algorithm on each problem instance. That is, an algorithm can only be best if it was successful. If no algorithm was successful, the value will be NA. Special care should be taken when preparing the performance values for unsuccessful algorithms. For example, if the performance measure is runtime and success is determined by whether the algorithm was able to find a solution within a timeout, the performance value for unsuccessful algorithms should be the timeout value. If the algorithm failed because of some other reason in a short amount of time, specifying this small amount of time may confuse some of the algorithm selection model learners.

Value

data	the combined data (features, performance, successes).
best	a list of the best algorithms.
ids	a list of names denoting the instance ID columns.
features	a list of names denoting problem features.
algorithmFeatures	a list of names denoting algorithm features. 'NULL' if no algorithm features are provided.
algorithmNames	a list of algorithm names. 'NULL' if no algorithm features are provided. See 'performance' field in that case.
algos	a column that stores names of algorithms. 'NULL' if no algorithm features are provided.
performance	a list of names denoting algorithm performances. If algorithm features are provided, a column name that stores algorithm performances.
success	a list of names denoting algorithm successes. If algorithm features are provided, a column name that stores algorithm successes.

minimize	true if the smaller performance values are better, else false.
cost	a list of names denoting feature costs.
costGroups	a list of list of names denoting which features belong to which group. Only returned if cost groups are given as input.

Author(s)

Lars Kotthoff

Examples

```
# features.csv looks something like
# ID,width,height
# 0,1.2,3
# ...
# performance.csv:
# ID,alg1,alg2
# 0,2,5
# ...
# success.csv:
# ID,alg1,alg2
# 0,T,F
# ...
#input(read.csv("features.csv"), read.csv("performance.csv"),
#    read.csv("success.csv"), costs=10)

# costs.csv:
# ID,width,height
# 0,3,4.5
# ...
#input(read.csv("features.csv"), read.csv("performance.csv"),
#    read.csv("success.csv"), costs=read.csv("costs.csv"))

# costGroups.csv:
# ID,group1,group2
# 0,3,4.5
# ...
#input(read.csv("features.csv"), read.csv("performance.csv"),
#    read.csv("success.csv"),
#    costs=list(groups=list(group1=c("height"), group2=c("width")),
#               values=read.csv("costGroups.csv")))
```

---

misc *Convenience functions*

Description

Convenience functions for computing and working with predictions.

Usage

```
vbs(data = NULL)
singleBest(data = NULL)
singleBestByCount(data = NULL)
singleBestByPar(data = NULL, factor = 10)
singleBestBySuccesses(data = NULL)
predTable(predictions = NULL, bestOnly = TRUE)
```

Arguments

data	the data to use. The structure returned by input.
factor	the penalization factor to use for non-successful choices. Default 10.
predictions	the list of predictions.
bestOnly	whether to tabulate only the respective best algorithm for each instance. Default TRUE.

Details

vbs and singleBest take a data frame of input data and return predictions that correspond to the virtual best and the single best algorithm, respectively. The virtual best picks the best algorithm for each instance. If no algorithm solved in the instance, NA is returned. The single best picks the algorithm that has the best cumulative performance over the entire data set.

singleBestByCount returns the algorithm that has the best performance the highest number of times over the entire data set. Only whether or not an algorithm is the best matters for this, not the difference to other algorithms.

singleBestByPar aggregates the PAR score over the entire data set and returns the algorithm with the lowest overall PAR score. singleBestBySuccesses counts the number of successes over the data set and returns the algorithm with the highest overall number of successes.

predTable tabulates the predicted algorithms in the same way that table does. If bestOnly is FALSE, all algorithms are considered – for example for regression models, predictions are made for all algorithms, so the table will simply show the number of instances for each algorithm. Set bestOnly to TRUE to tabulate only the best algorithm for each instance.

Value

A data frame with the predictions for each instance. The columns of the data frame are the instance ID columns (as determined by input), the algorithm, the score of the algorithm, and the iteration (always 1). The score is 1 if the respective algorithm is chosen for the instance, 0 otherwise. More than one prediction may be made for each instance and iteration.

For predTable, a table.

Author(s)

Lars Kotthoff

Examples

```
if(Sys.getenv("RUN_EXPENSIVE") == "true") {
data(satsolvers)

# number of total successes for virtual best solver
print(sum(successes(satsolvers, vbs)))
# number of total successes for single best solver by count
print(sum(successes(satsolvers, singleBestByCount)))

# sum of PAR10 scores for single best solver by PAR10 score
print(sum(parscores(satsolvers, singleBestByPar)))

# number of total successes for single best solver by successes
print(sum(successes(satsolvers, singleBestBySuccesses)))

# print a table of the best solvers per instance
print(predTable(vbs(satsolvers)))
}
```

`misclassificationPenalties` *Misclassification penalty*

Description

Calculates the penalty incurred because of making incorrect decisions, i.e. choosing suboptimal algorithms.

Usage

```
misclassificationPenalties(data, model, addCosts = NULL)
```

Arguments

`data`	the data used to induce the model. The same as given to `classify`, `classifyPairs`, `cluster` or `regression`.
`model`	the algorithm selection model. Can be either a model returned by one of the model-building functions or a function that returns predictions such as `vbs` or the predictor function of a trained model.
`addCosts`	does nothing. Only here for compatibility with the other evaluation functions.

Details

Compares the performance of the respective chosen algorithm to the performance of the best algorithm for each datum. Returns the absolute difference. This denotes the penalty for choosing a suboptimal algorithm, e.g. the additional time required to solve a problem or reduction in solution quality incurred. The misclassification penalty of the virtual best is always zero.

If the model returns NA (e.g. because no algorithm solved the instance), 0 is returned as misclassification penalty.

data may contain a train/test partition or not. This makes a difference when computing the misclassification penalties for the single best algorithm. If no train/test split is present, the single best algorithm is determined on the entire data. If it is present, the single best algorithm is determined on each test partition. That is, the single best is local to the partition and may vary across partitions.

Value

A list of the misclassification penalties.

Author(s)

Lars Kotthoff

See Also

parscores, successes

Examples

```
if(Sys.getenv("RUN_EXPENSIVE") == "true") {
data(satsolvers)
folds = cvFolds(satsolvers)

model = classify(classifier=makeLearner("classif.J48"), data=folds)
sum(misclassificationPenalties(folds, model))
}
```

---

normalize *Normalize features*

Description

Normalize input data so that the values for all features cover the same range -1 to 1.

Usage

```
normalize(rawfeatures, meta = NULL)
```

Arguments

rawfeatures	data frame with the feature values to normalize.
meta	meta data to use for the normalization. If supplied should be a list with members minValues that contains the minimum values for all features and maxValues that contains the maximum values for all features. Will be computed if not supplied.

Details

normalize subtracts the minimum (supplied or computed) from all values of a feature, divides by the difference between maximum and minimum, multiplies by 2 and subtracts 1. The range of the values for all features will be -1 to 1.

Value

features	the normalized feature vectors.
meta	the minimum and maximum values for each feature before normalization. Can be used in subsequent calls to normalize for new data.

Author(s)

Lars Kotthoff

Examples

```
if(Sys.getenv("RUN_EXPENSIVE") == "true") {
data(satsolvers)
folds = cvFolds(satsolvers)

cluster(clusterer=makeLearner("cluster.XMeans"), data=folds, pre=normalize)
}
```

---

parscores *Penalized average runtime score*

Description

Calculates the penalized average runtime score which is commonly used for evaluating satisfiability solvers on a set of problems.

Usage

```
parscores(data, model, factor = 10, timeout, addCosts = NULL)
```

Arguments

data	the data used to induce the model. The same as given to classify, classifyPairs, cluster or regression.
model	the algorithm selection model. Can be either a model returned by one of the model-building functions or a function that returns predictions such as vbs or the predictor function of a trained model.
factor	the penalization factor to use for non-successful choices. Default 10.
timeout	the timeout value to be multiplied by the penalization factor. If not specified, the maximum performance value of all algorithms on the entire data is used.

addCosts — whether to add feature costs. You should not need to set this manually, the default of NULL will have LLAMA figure out automatically depending on the model whether to add costs or not. This should always be true (the default) except for comparison algorithms (i.e. single best and virtual best).

Details

Returns the penalized average runtime performances of the respective chosen algorithm on each problem instance.

If feature costs have been given and addCosts is TRUE, the cost of the used features or feature groups is added to the performance of the chosen algorithm. The used features are determined by examining the the features member of data, not the model. If after that the performance value is above the timeout value, the timeout value multiplied by the factor is assumed.

If the model returns NA (e.g. because no algorithm solved the instance), timeout * factor is returned as PAR score.

data may contain a train/test partition or not. This makes a difference when computing the PAR scores for the single best algorithm. If no train/test split is present, the single best algorithm is determined on the entire data. If it is present, the single best algorithm is determined on each test partition. That is, the single best is local to the partition and may vary across partitions.

Value

A list of the penalized average runtimes.

Author(s)

Lars Kotthoff

See Also

misclassificationPenalties, successes

Examples

```
if(Sys.getenv("RUN_EXPENSIVE") == "true") {
data(satsolvers)
folds = cvFolds(satsolvers)

model = classify(classifier=makeLearner("classif.J48"), data=folds)
sum(parscores(folds, model))

# use factor of 5 instead of 10.
sum(parscores(folds, model, 5))

# explicitly specify timeout.
sum(parscores(folds, model, timeout = 3600))
}
```

plot *Plot convenience functions to visualise selectors*

Description

Functions to plot the performance of selectors and compare them to others.

Usage

```
perfScatterPlot(metric, modelx, modely, datax, datay=datax,
    addCostsx=NULL, addCostsy=NULL, pargs=NULL, ...)
```

Arguments

metric	the metric used to evaluate the model. Can be one of misclassificationPenalties, parscores or successes.
modelx	the algorithm selection model to be plotted on the x axis. Can be either a model returned by one of the model-building functions or a function that returns predictions such as vbs or the predictor function of a trained model.
modely	the algorithm selection model to be plotted on the y axis. Can be either a model returned by one of the model-building functions or a function that returns predictions such as vbs or the predictor function of a trained model.
datax	the data used to evaluate modelx. Will be passed to the metric function.
datay	the data used to evaluate modely. Can be omitted if the same as for modelx. Will be passed to the metric function.
addCostsx	whether to add feature costs for modelx. You should not need to set this manually, the default of NULL will have LLAMA figure out automatically depending on the model whether to add costs or not. This should always be true (the default) except for comparison algorithms (i.e. single best and virtual best).
addCostsy	whether to add feature costs for modely. You should not need to set this manually, the default of NULL will have LLAMA figure out automatically depending on the model whether to add costs or not. This should always be true (the default) except for comparison algorithms (i.e. single best and virtual best).
pargs	any arguments to be passed to geom_points.
...	any additional arguments to be passed to the metrics. For example the penalisation factor for parscores.

Details

perfScatterPlot creates a scatter plot that compares the performances of two algorithm selectors. It plots the performance on each instance in the data set for modelx on the x axis versus modely on the y axis. In addition, a diagonal line is drawn to denote the line of equal performance for both selectors.

Value

A ggplot object.

Author(s)

Lars Kotthoff

See Also

misclassificationPenalties, parscores, successes

Examples

```
if(Sys.getenv("RUN_EXPENSIVE") == "true") {
data(satsolvers)
folds = cvFolds(satsolvers)
model = classify(classifier=makeLearner("classif.J48"), data=folds)

# Simple plot to compare our selector to the single best in terms of PAR10 score
library(ggplot2)
perfScatterPlot(parscores,
        model, singleBest,
        folds, satsolvers) +
    scale_x_log10() + scale_y_log10() +
    xlab("J48") + ylab("single best")

# additional aesthetics for points
perfScatterPlot(parscores,
        model, singleBest,
        folds, satsolvers,
        pargs=aes(colour = scorex)) +
    scale_x_log10() + scale_y_log10() +
    xlab("J48") + ylab("single best")
}
```

---

regression *Regression model*

Description

Build a regression model that predicts the algorithm to use based on the features of the problem and optionally features of the algorithms.

Usage

```
regression(regressor = NULL, data = NULL,
    pre = function(x, y=NULL) { list(features=x) },
    combine = NULL, expand = identity, save.models = NA,
    use.weights = TRUE)
```

Arguments

regressor	the mlr regressor to use. See examples.
data	the data to use with training and test sets. The structure returned by one of the partitioning functions.
pre	a function to preprocess the data. Currently only normalize. Optional. Does nothing by default.
combine	the function used to combine the predictions of the individual regression models for stacking. Default NULL. See details.
expand	a function that takes a matrix of performance predictions (columns are algorithms, rows problem instances) and transforms it into a matrix with the same number of rows. Only meaningful if combine is not null. Default is the identity function, which will leave the matrix unchanged. See examples.
save.models	Whether to serialize and save the models trained during evaluation of the model. If not NA, will be used as a prefix for the file name.
use.weights	Whether to use instance weights if supported. Default TRUE.

Details

regression takes data and processes it using pre (if supplied). If no algorithm features are provided, regressor is called to induce separate regression models for each of the algorithms to predict its performance. When algorithm features are present, regressor is called to induce one regression model for all algorithms to predict their performance. The best algorithm is determined from the predicted performances by examining whether performance is to be minimized or not, as specified when creating the data structure through input.

The evaluation across the training and test sets will be parallelized automatically if a suitable backend for parallel computation is loaded. The parallelMap level is "llama.fold".

If combine is not null, it is assumed to be an mlr classifier and will be used to learn a model to predict the best algorithm given the original features and the performance predictions for the individual algorithms. combine option is currently not supported with algorithm features. If this classifier supports weights and use.weights is TRUE, they will be passed as the difference between the best and the worst algorithm. Optionally, expand can be used to supply a function that will modify the predictions before giving them to the classifier, e.g. augment the performance predictions with the pairwise differences (see examples).

If all predictions of an underlying machine learning model are NA, the prediction will be NA for the algorithm and -Inf for the score if the performance value is to be maximised, Inf otherwise.

If save.models is not NA, the models trained during evaluation are serialized into files. Each file contains a list with members model (the mlr model), train.data (the mlr task with the training data), and test.data (the data frame with the test data used to make predictions). The file name starts with save.models, followed by the ID of the machine learning model, followed by "combined" if the model combines predictions of other models, followed by the number of the fold. Each model for each fold is saved in a different file.

Value

predictions	a data frame with the predictions for each instance and test set. The columns of the data frame are the instance ID columns (as determined by input), the

algorithm, the score of the algorithm, and the iteration (e.g. the number of the fold for cross-validation). More than one prediction may be made for each instance and iteration. The score corresponds to the predicted performance value. If stacking is used, each prediction is simply the best algorithm with a score of 1.

`predictor` a function that encapsulates the regression model learned on the *entire* data set. Can be called with data for the same features with the same feature names as the training data to obtain predictions in the same format as the `predictions` member.

`models` the list of models trained on the *entire* data set. This is meant for debugging/inspection purposes and does not include any models used to combine predictions of individual models.

Author(s)

Lars Kotthoff

References

Kotthoff, L. (2012) Hybrid Regression-Classification Models for Algorithm Selection. *20th European Conference on Artificial Intelligence*, 480–485.

See Also

`classify`, `classifyPairs`, `cluster`, `regressionPairs`

Examples

```
if(Sys.getenv("RUN_EXPENSIVE") == "true") {
data(satsolvers)
folds = cvFolds(satsolvers)

res = regression(regressor=makeLearner("regr.lm"), data=folds)
# the total number of successes
sum(successes(folds, res))
# predictions on the entire data set
res$predictor(satsolvers$data[satsolvers$features])

res = regression(regressor=makeLearner("regr.ksvm"), data=folds)

# combine performance predictions using classifier
ress = regression(regressor=makeLearner("regr.ksvm"),
                  data=folds,
                  combine=makeLearner("classif.J48"))

# add pairwise differences to performance predictions before running classifier
ress = regression(regressor=makeLearner("regr.ksvm"),
                  data=folds,
                  combine=makeLearner("classif.J48"),
                  expand=function(x) { cbind(x, combn(c(1:ncol(x)), 2,
                         function(y) { abs(x[,y[1]] - x[,y[2]]) })) })
```

```
}
```

`regressionPairs` *Regression model for pairs of algorithms*

Description

Builds regression models for each pair of algorithms that predict the performance difference based on the features of the problem and optionally features of the algorithms. The sum over all pairs that involve a particular algorithm is aggregated as the score of the algorithm.

Usage

```
regressionPairs(regressor = NULL, data = NULL,
    pre = function(x, y=NULL) { list(features=x) }, combine = NULL,
    save.models = NA, use.weights = TRUE)
```

Arguments

`regressor`	the regression function to use. Must accept a formula of the values to predict and a data frame with features. Return value should be a structure that can be given to `predict` along with new data. See examples.
`data`	the data to use with training and test sets. The structure returned by one of the partitioning functions.
`pre`	a function to preprocess the data. Currently only `normalize`. Optional. Does nothing by default.
`combine`	the function used to combine the predictions of the individual regression models for stacking. Default `NULL`. See details.
`save.models`	Whether to serialize and save the models trained during evaluation of the model. If not `NA`, will be used as a prefix for the file name.
`use.weights`	Whether to use instance weights if supported. Default `TRUE`.

Details

`regressionPairs` takes the training and test sets in `data` and processes it using `pre` (if supplied). If no algorithm features are provided, `regressor` is called to induce a regression model for each pair of algorithms to predict the performance difference between them. When algorithm features are present, `regressor` is called to induce one regression model for all pairs of algorithms to predict the performance difference between them. If `combine` is not supplied, the best overall algorithm is determined by summing the performance differences over all pairs for each algorithm and ranking them by this sum. The algorithm with the largest value is chosen. If it is supplied, it is assumed to be an mlr classifier. This classifier is passed the original features and the predictions for each pair of algorithms. `combine` option is currently not supported with algorithm features. If the classifier supports weights and `use.weights` is `TRUE`, the performance difference between the best and the worst algorithm is passed as weight.

The aggregated score for each algorithm quantifies how much better it is than the other algorithms, where bigger values are better. Positive numbers denote that the respective algorithm usually exhibits better performance than most of the other algorithms, while negative numbers denote that it is usually worse.

The evaluation across the training and test sets will be parallelized automatically if a suitable backend for parallel computation is loaded. The `parallelMap` level is "llama.fold".

Training this model can take a very long time. Given `n` algorithms, `choose(n, 2) * n` models are trained and evaluated. This is significantly slower than the other approaches that train a single model or one for each algorithm. Even with algorithmic features present, when only a single model is trained, the process still takes a long time due to the amount of data.

If all predictions of an underlying machine learning model are `NA`, the prediction will be `NA` for the algorithm and `-Inf` for the score if the performance value is to be maximised, `Inf` otherwise.

If `save.models` is not `NA`, the models trained during evaluation are serialized into files. Each file contains a list with members `model` (the mlr model), `train.data` (the mlr task with the training data), and `test.data` (the data frame with the test data used to make predictions). The file name starts with `save.models`, followed by the ID of the machine learning model, followed by "combined" if the model combines predictions of other models, followed by the number of the fold. Each model for each fold is saved in a different file.

Value

`predictions` a data frame with the predictions for each instance and test set. The columns of the data frame are the instance ID columns (as determined by `input`), the algorithm, the score of the algorithm, and the iteration (e.g. the number of the fold for cross-validation). More than one prediction may be made for each instance and iteration. The score corresponds to how much better performance the algorithm delivers compared to the other algorithms in the portfolio. If stacking is used, each prediction is simply the best algorithm with a score of 1.

`predictor` a function that encapsulates the classifier learned on the *entire* data set. Can be called with data for the same features with the same feature names as the training data to obtain predictions in the same format as the `predictions` member.

`models` the models for each pair of algorithms trained on the *entire* data set. This is meant for debugging/inspection purposes and does not include any models used to combine predictions of individual models.

Author(s)

Lars Kotthoff

See Also

`classify`, `classifyPairs`, `cluster`, `regression`

Examples

```
if(Sys.getenv("RUN_EXPENSIVE") == "true") {
data(satsolvers)
folds = cvFolds(satsolvers)
```

```
model = regressionPairs(regressor=makeLearner("regr.lm"), data=folds)
# the total number of successes
sum(successes(folds, model))
# predictions on the entire data set
model$predictor(satsolvers$data[satsolvers$features])

# combine predictions using J48 induced classifier
model = regressionPairs(regressor=makeLearner("regr.lm"), data=folds,
    combine=makeLearner("classif.J48"))
}
```

`satsolvers` *Example data for Leveraging Learning to Automatically Manage Algorithms*

Description

Performance data for 19 SAT solvers on 2433 SAT instances.

Usage

```
data(satsolvers)
```

Format

`satsolvers` is a list in the format returned by `input` and expected by the other functions of LLAMA. The list has the following components.

data: The original input data merged. That is, the data frames processed by `input` in a single data frame with the following additional columns.

best: The algorithm(s) with the best performance for each row.

***_success:** For each algorithm whether it was successful on the respective row.

features: The names of the columns that contain feature values.

performance: The names of the columns that contain performance data.

success: The names of the columns indicating whether an algorithm was successful.

minimize: Whether the performance is to be minimized.

cost: The names of the columns that contain the feature group computation cost for each instance.

costGroups: A list the maps the names of the feature groups to the list of feature names that are contained in it.

Details

Performance data for 19 SAT solvers on 2433 SAT instances. For each instance, 36 features were measured. In addition to the performance (time) on each instance, data on whether a solver timed out on an instance is included. The cost to compute all features is included as well.

Source

Hurley, B., Kotthoff, L., Malitsky, Y., O'Sullivan, B. (2014) Proteus: A Hierarchical Portfolio of Solvers and Transformations. *Eleventh International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) techniques in Constraint Programming*.

See Also

input

Examples

```
data(satsolvers)
```

---

successes *Success*

Description

Was the problem solved successfully using the chosen algorithm?

Usage

```
successes(data, model, timeout, addCosts = NULL)
```

Arguments

data	the data used to induce the model. The same as given to classify, classifyPairs, cluster or regression.
model	the algorithm selection model. Can be either a model returned by one of the model-building functions or a function that returns predictions such as vbs or the predictor function of a trained model.
timeout	the timeout value to be multiplied by the penalization factor. If not specified, the maximum performance value of all algorithms on the entire data is used.
addCosts	whether to add feature costs. You should not need to set this manually, the default of NULL will have LLAMA figure out automatically depending on the model whether to add costs or not. This should always be true (the default) except for comparison algorithms (i.e. single best and virtual best).

Details

Returns TRUE if the chosen algorithm successfully solved the problem instance, FALSE otherwise for each problem instance.

If feature costs have been given and addCosts is TRUE, the cost of the used features or feature groups is added to the performance of the chosen algorithm. The used features are determined by examining the the features member of data, not the model. If after that the performance value is above the timeout value, FALSE is assumed. If whether an algorithm was successful is

not determined by performance and feature costs, don't pass costs when creating the LLAMA data frame.

If the model returns NA (e.g. because no algorithm solved the instance), FALSE is returned as success.

data may contain a train/test partition or not. This makes a difference when computing the successes for the single best algorithm. If no train/test split is present, the single best algorithm is determined on the entire data. If it is present, the single best algorithm is determined on each test partition. That is, the single best is local to the partition and may vary across partitions.

Value

A list of the success values.

Author(s)

Lars Kotthoff

See Also

misclassificationPenalties, parscores

Examples

```
if(Sys.getenv("RUN_EXPENSIVE") == "true") {
data(satsolvers)
folds = cvFolds(satsolvers)

model = classify(classifier=makeLearner("classif.J48"), data=folds)
sum(successes(folds, model))
}
```

---

trainTest *Train / test split*

Description

Split a data set into train and test set.

Usage

```
trainTest(data, trainpart = 0.6, stratify = FALSE)
```

Arguments

data	the data to use. The structure returned by input.
trainpart	the fraction of the data to use for training. Default 0.6.
stratify	whether to stratify the folds. Makes really only sense for classification models. Defaults to FALSE.

Details

Partitions the data set into training and test set according to the specified fraction. The training and test index sets are added to the original data and returned. If requested, the distribution of the best algorithms in training and test set is approximately the same, i.e. the sets are stratified.

If the data set has train and test partitions already, they are overwritten.

Value

`train`	a (one-element) list of index sets for training.
`test`	a (one-element) list of index sets for testing.
`...`	the original members of `data`. See `input`.

Author(s)

Lars Kotthoff

See Also

`bsFolds`, `cvFolds`

Examples

```
data(satsolvers)
trainTest = trainTest(satsolvers)

# use 50-50 split instead of 60-40
trainTest1 = trainTest(satsolvers, 0.5)

# stratify
trainTestU = trainTest(satsolvers, stratify=TRUE)
```

---

tune — *Tune the hyperparameters of the machine learning algorithm underlying a model*

Description

Functions to tune the hyperparameters of the machine learning algorithm underlying a model with respect to a performance measure.

Usage

```
tuneModel(ldf, llama.fun, learner, design, metric = parscores, nfolds = 10L,
    quiet = FALSE)
```

Arguments

ldf	the LLAMA data to use. The structure returned by input.
llama.fun	the LLAMA model building function.
learner	the mlr learner to use.
design	the data frame denoting the parameter values to try. Can be produced with the ParamHelpers package. See examples.
metric	the metric used to evaluate the model. Can be one of misclassificationPenalties, parscores or successes.
nfolds	the number of folds. Defaults to 10. If -1 is given, leave-one-out cross-validation folds are produced.
quiet	whether to output information on the intermediate values and progress during tuning.

Details

tuneModel finds the hyperparameters from the set denoted by design of the machine learning algorithm learner that give the best performance with respect to the measure metric for the LLAMA model type llama.fun on data ldf. It uses a nested cross-validation internally; the number of inner folds is given through nfolds, the number of outer folds is either determined by any existing partitions of ldf or, if none are present, by nfolds as well.

During each iteration of the inner cross-validation, all parameter sets specified in design are evaluated and the one with the best performance value chosen. The mean performance over all instances in the data is logged for all evaluations. This parameter set is then used to build and evaluate a model in the outer cross-validation. The predictions made by this model along with the parameter values used to train it are returned.

Finally, a normal (not-nested) cross-validation is performed to find the best parameter values on the *entire* data set. The predictor of this model along with the parameter values used to train it is returned. The interface corresponds to the normal LLAMA model-building functions in that respect – the returned data structure is the same with a few additional values.

The evaluation across the folds sets will be parallelized automatically if a suitable backend for parallel computation is loaded. The parallelMap level is "llama.tune".

Value

predictions	a data frame with the predictions for each instance and test set. The structure is the same as for the underlying model building function and the predictions are the ones made by the models trained with the best parameter values for the respective fold.
predictor	a function that encapsulates the classifier learned on the *entire* data set with the best parameter values determined on the *entire* data set. Can be called with data for the same features with the same feature names as the training data to obtain predictions in the same format as the predictions member.
models	the list of models trained on the *entire* data set. This is meant for debugging/inspection purposes.

parvals	the best parameter values on the entire data set used for training the predictor model.
inner.parvals	the best parameter values during each iteration of the outer cross-validation. These parameters were used to train the models that made the predictions in predictions.

Author(s)

Bernd Bischl, Lars Kotthoff

Examples

```
if(Sys.getenv("RUN_EXPENSIVE") == "true") {
library(ParamHelpers)
data(satsolvers)

learner = makeLearner("classif.J48")
# parameter set for J48
ps = makeParamSet(makeIntegerParam("M", lower = 1, upper = 100))
# generate 10 random parameter sets
design = generateRandomDesign(10, ps)
# tune with respect to PAR10 score (default) with 10 outer and inner folds
# (default)
res = tuneModel(satsolvers, classify, learner, design)
}
```

Index

∗ **cluster**
cluster, 10
∗ **datasets**
satsolvers, 30
∗ **imputation**
imputeCensored, 15
∗ **models**
analysis, 4
bsFolds, 5
classify, 6
classifyPairs, 8
cvFolds, 13
input, 16
misc, 18
misclassificationPenalties, 20
normalize, 21
parscores, 22
plot, 24
regressionPairs, 28
successes, 31
trainTest, 32
tune, 33
∗ **package**
llama-package, 2
∗ **regression**
regression, 25

analysis, 4

bsFolds, 5, *13*, *33*

classify, 6, *10*, *12*, *27*, *29*
classifyPairs, *7*, 8, *12*, *27*, *29*
cluster, *7*, 10, *10*, *27*, *29*
contributions *(*analysis*)*, 4
cvFolds, *5*, 13, *33*

helpers, 14

imputeCensored, 15
input, *5*, *13*, 16, *31*, *33*

llama *(*llama-package*)*, 2
llama-package, 2

makeRLearner.classif.constant *(*helpers*)*, 14
misc, 18
misclassificationPenalties, 20, *23*, *25*, *32*

normalize, 21

parscores, *21*, 22, *25*, *32*
perfScatterPlot *(*plot*)*, 24
plot, 24
predictLearner.classif.constant *(*helpers*)*, 14
predTable *(*misc*)*, 18
print.llama.data *(*helpers*)*, 14
print.llama.model *(*helpers*)*, 14

regression, *7*, *10*, *12*, 25, *29*
regressionPairs, *7*, *10*, *12*, *27*, 28

satsolvers, 30
singleBest *(*misc*)*, 18
singleBestByCount *(*misc*)*, 18
singleBestByPar *(*misc*)*, 18
singleBestBySuccesses *(*misc*)*, 18
successes, *21*, *23*, *25*, 31

trainLearner.classif.constant *(*helpers*)*, 14
trainTest, *5*, *13*, 32
tune, 33
tuneModel *(*tune*)*, 33

vbs *(*misc*)*, 18