Package 'automlr'

July 16, 2017

Title Automatic Machine Learning in R

Description Package for interfacing mlr with algorithm configuration optimization libraries, giving functionality for optimizing across different ML algorithms.

URL https://github.com/mlr-org/automlr

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

License AGPL-3

Encoding UTF-8

Depends R (>= 3.0.2), mlr (>= 2.8), ParamHelpers (>= 1.7), utils, stats

Imports BBmisc (>= 1.9), checkmate (>= 1.7.1), parallelMap (>= 1.3)

Suggests mlrMBO (>= 1.0), smoof (>= 1.3), irace (>= 1.05), DiceKriging, fastICA, testthat, FSelector, randomForestSRC, glmnet, lqa, stepPlr, MASS, DiscriMiner, sparseLDA, elasticnet, rrlda, klaR, sda, caret, mda, sparsediscrim, kknn, class, rknn, FNN, RWeka, party, nodeHarvest, rpart, randomForest, extraTrees, ranger, rFerns, rotationForest, ada, mboost, adabag, gbm, xgboost, SwarmSVM, LiblineaR, kernlab, e1071, deepnet, nnet, neuralnet

Roxygen list(wrap = FALSE)

LazyData yes

ByteCompile yes

Version 0.1.0

RoxygenNote 6.0.1

Collate 'abortHandler.R' 'amexowrapper.R' 'aminterface.R' 'autolearner.R' 'defaults.R' 'lsambackends.R' 'mlrLearners.R' 'automlr.R' 'buildLearners.R' 'buildSearchspace.R' 'buildWrapperSearchspace.R' 'failImputationWrapper.R' 'optDummy.R' 'optGeneric.R' 'optIRace.R' 'optMBO.R' 'optRandom.R' 'runwithtimeout.R' 'runwithtimeoutfork.R' 'shims.R' 'timeconstraintWrapper.R' 'utils.R' 'wrappers.R' 'zzz.R'

NeedsCompilation no

Author mb706 [aut, cre]

Maintainer mb706 <mb706@uni>

R topics documented:

`amaddprior` *update the environment's prior by adding the information contained in 'prior'.*

Description

This is entirely backend-dependent.

Usage

```
amaddprior(env, prior)
```

Arguments

`env`	`[environment]` The private data of this backend
`prior`	[any] A 'prior' object.

`amaddprior.amdummy` *Reference implementation that exemplifies the backend interface.*

Description

Counts the number of priors used.

Usage

```
## S3 method for class 'amdummy'
amaddprior(env, prior)
```

Arguments

`env`	`[environment]` The private data of this backend
`prior`	[any] A 'prior' object.

`amfinish` *Converte the `AMState` object as returned by `automlr` to an `AMResult` object.*

Description

The result object contains information about the solution that is relatively backend-independent.

Usage

```
amfinish(amstate)
```

Arguments

`amstate`	`[AMState]` The AMState object which is to be converted.

Value

`AMResult` Object representing the optimum found by the `automlr` run.

Object members:

learner [`Learner`] The (constructed) learner that achieved the optimum.

opt.point [`list`] Optimal hyperparameters for learner.

opt.val [`numeric`] Optimum reached for the `AMState`'s `measure`.

opt.path [`OptPath`] Information about all the evaluations performed.

result [any] Furthe backend-dependent information about the optimum.

... (further elements) Elements of the `AMState` object.

`amgetprior` *return whatever kind of object this backend uses as 'prior'.*

Description

This is called usually after `amaddprior` to retrieve an updated prior object.

Usage

```
amgetprior(env)
```

Arguments

`env`	`[environment]` The private data of this backend.

Value

any An object representing the prior.

`amgetprior.amdummy` *Reference implementation that exemplifies the backend interface.*

Description

Just give the prior object, which will be a number in this case.

Usage

```
## S3 method for class 'amdummy'
amgetprior(env)
```

Arguments

`env`	`[environment]` The private data of this backend.

Value

any An object representing the prior.

`amoptimize` *Perform optimization, respecting the given budget.*

Description

return a vector detailing the spent budget.

optimization progress should be saved to 'env'.

Usage

```
amoptimize(env, stepbudget, verbosity, deadline)
```

Arguments

`env`	`[environment]` The private data of this backend.
`stepbudget`	`[numeric]` The budget for this optimization step with one or several of the entries `walltime` and `evals`. See `automlr` for details.
`verbosity`	`[numeric(1)]` Output options.
`deadline`	`[numeric(1)]` The number of seconds of runtime that this call should not exceed. While the time budget gives a soft limit and tries to finish calculations that have started by the time the budget is spent, this is a hard limit which should be kept as closely as possible, even if it means throwing away data.

Value

`numeric(4)` The budget spent during this invocation.

amoptimize.amdummy *Reference implementation that exemplifies the backend interface.*

Description

Simulate the spending of budget.

optimization progress should be saved to 'env'.

Usage

```
## S3 method for class 'amdummy'
amoptimize(env, stepbudget, verbosity, deadline)
```

Arguments

`env`	`[environment]` The private data of this backend.
`stepbudget`	`[numeric]` The budget for this optimization step with one or several of the entries `walltime` and `evals`. See `automlr` for details.
`verbosity`	`[numeric(1)]` Output options.
`deadline`	`[numeric(1)]` The number of seconds of runtime that this call should not exceed. While the time budget gives a soft limit and tries to finish calculations that have started by the time the budget is spent, this is a hard limit which should be kept as closely as possible, even if it means throwing away data.

Value

`numeric(4)` The budget spent during this invocation.

amresult *Give information about the optimum found.*

Description

This function can do some backend bound work to generate more information about the found optimum.

Usage

```
amresult(env)
```

Arguments

`env`	`[environment]` The private data of this backend.

Value

`list` A named list which will be inserted into the result object. Required elements are:

learner [`Learner`] The (constructed) learner that achieved the optimum.

opt.point [`list`] Optimal hyperparameters for learner.

opt.val [`numeric`] Optimum reached for the `AMState`'s `measure`.

opt.path [`OptPath`] Information about all the evaluations performed.

result [any] Furthe backend-dependent information about the optimum.

Further elements are possible, but names should not collide with `AMState` / `AMResult` slot names.

`amresult.amdummy` *Reference implementation that exemplifies the backend interface.*

Description

For the dummy backend, the result will be a random point in the search space.

Usage

```
## S3 method for class 'amdummy'
amresult(env)
```

Arguments

`env`	`[environment]` The private data of this backend.

Value

`list` A named list which will be inserted into the result object. Required elements are:

learner [`Learner`] The (constructed) learner that achieved the optimum.

opt.point [`list`] Optimal hyperparameters for learner.

opt.val [`numeric`] Optimum reached for the `AMState`'s `measure`.

opt.path [`OptPath`] Information about all the evaluations performed.

result [any] Furthe backend-dependent information about the optimum.

Further elements are possible, but names should not collide with `AMState` / `AMResult` slot names.

amsetup *Set up the backend private data.*

Description

This function is called once in the lifetime of the backend private data. All arguments except 'env' will not be passed to amoptimize, so they should be saved in 'env'.

Usage

```
amsetup(env, opt, prior, learner, task, measure, verbosity)
```

Arguments

`env`	`[environment]` The private data of this backend.
`opt`	`[AutomlrBackendConfig]` Options given for the backend. This is a list returned by a function that was registered for the backend using `registerBackend`.
`prior`	[any] The prior as passed to the `automlr` invocation.
`learner`	`[Learner]` The learner object that was built from the declared search space.
`task`	`[Task]` The task to optimize the `Learner` over.
`measure`	`[Measure]` The measure to optimize.
`verbosity`	`[numeric(1)]` Output options.

Value

`NULL`

amsetup.amdummy *Reference implementation that exemplifies the backend interface.*

Description

The dummy backend here just saves the information necessary to give some bogus result in `amresult.amdummy`.

Usage

```
## S3 method for class 'amdummy'
amsetup(env, opt, prior, learner, task, measure, verbosity)
```

Arguments

`env`	`[environment]` The private data of this backend.
`opt`	`[AutomlrBackendConfig]` Options given for the backend. This is a list returned by a function that was registered for the backend using `registerBackend`.
`prior`	[any] The prior as passed to the `automlr` invocation.
`learner`	`[Learner]` The learner object that was built from the declared search space.
`task`	`[Task]` The task to optimize the `Learner` over.
`measure`	`[Measure]` The measure to optimize.
`verbosity`	`[numeric(1)]` Output options.

Value

`NULL`

`areInterruptsSuspended` *Check whether interrupts are suspended*

Description

A debug function that gives info whether interrupts are suspended. Some R operations and some library calls reactivate interrupt processing, but this is apparently nowhere documented. This function allows one to take an empirical approach.

Usage

```
areInterruptsSuspended()
```

Value

logical(1) Whether interrupts are suspended.

argsToList *create a list of formal arguments*

Description

Return a list of formal arguments. Useful for registerBackend.

Usage

```
argsToList()
```

Examples

```
fun <- function(x = 1, y = 2) {
  argsToList()
}

fun(y = 10)
fun()
```

autolearner *Create Autolearner objects*

Description

Autolearner objects wrap mlr Learner objects as well as preprocessing functions and provide additional meta information. This is used to define the automlr searchspace.

Usage

```
autolearner(learner, searchspace = list(), stacktype = "learner")
```

Arguments

learner	[Learner|list] An mlr Learner object for a "learner" stacktype, otherwise an object returned by autoWrapper. If a Learner is required, it may also be the ID of the Learner which postpones allocation of memory and loading of packages and may therefore be preferred.
searchspace	[list of Searchparam] List of Searchparams reqpresenting the relevant parameter search space.
stacktype	[character(1)] Describing how this object can be connected with other learners. Must be one of "wrapper" or "learner".

automlr — *Automatically choose a model with parameters to fit.*

Description

This is the main entry point of automlr.

Usage

```
automlr(task, ...)

## S3 method for class 'Task'
automlr(task, measure = NULL, budget = 0,
  searchspace = mlrLearners, prior = NULL, savefile = NULL,
  save.interval = default.save.interval, backend, max.walltime.overrun = if
  ("walltime" %in% names(budget)) budget["walltime"] * 0.1 + 600 else 3600,
  max.learner.time = Inf, verbosity = 0, ...)

## S3 method for class 'character'
automlr(task, budget = NULL, prior = NULL,
  savefile = task, save.interval = default.save.interval,
  new.seed = FALSE, max.walltime.overrun = if ("walltime" %in%
  names(budget)) budget["walltime"] * 0.1 + 600 else 3600, verbosity = 0, ...)

## S3 method for class 'AMState'
automlr(task, budget = NULL, prior = NULL,
  savefile = NULL, save.interval = default.save.interval,
  new.seed = FALSE, max.walltime.overrun = if ("walltime" %in%
  names(budget)) budget["walltime"] * 0.1 + 600 else 3600, verbosity = 0, ...)
```

Arguments

`task`	[`Task` | `AMState` | `character(1)`] Either: The mlr `Task` object to fit a model on. Or: An `AMState` object, or a `character(1)` containing the file name of an `.rds` file containing such an object. The AMState usually contains progress of a past optimization run that was aborted by the user, a crash, or which ran out of budget.
`...`	No further arguments should be given.
`measure`	[`Measure`] The mlr `Measure` object to optimize for. If this is not given and the first argument is a `Task` object, uses the task's default measure.
`budget`	[`numeric` | list of `numeric`] A named list or named vector with one or several of the entries `walltime` time since invocation `evals` number of model fit evaluations (Time is always given in seconds.) When any of the budget criteria is exceeded, the optimization process will halt at the next possible point and return. In the current implementation, this is only checked between model fits and evaluations, so the time budgets may be exceeded, in some cases substantially. If the `taks` argument is an `AMState` object

or a `character(1)` referring to an `AMState` rds-file, this is optional and defaults to the referenced `AMState`'s budget *minus the already used up budget*. To continue an already finished run, therefore, one needs to pass a higher budget than the `$spent` slot indicates. Passing `0` will return an AMState object without performing any optimization or touching the file system.

`searchspace`
: [list of `Autolearner`]
Declaration of the searchspace: The mlr `Learners` to use and the parameter domains to consider for optimization. `Learners` can be chosen manually, either by creating custom `Autolearner` objects using `autolearner`, using elements of the provided `link{mlrLearners}` list, searching all implemented `Learners` by using `link{mlrLearners}` (default), or searching all `Learners` without considering preprocessing using `link{mlrLearnersNoWrap}`.
From the provided list, only `Learners` that fit the task characteristics and type will be used, the others will be ignored without notification.

`prior`
: [any]
A black box that contains some form of knowledge about the world at large that may help speed up the optimization process. Effect of this parameter depends on the backend implementation. Currently, this is ignored by all backends.

`savefile`
: [`character(1)`]
Name of a file or folder in which intermediate progress will be saved. This is will prevent data getting lost in case of a crash. The data written is an `AMState` object in an `.rds` file that can be read and run with another `automlr` call to resume optimization.
If `savefile` ends with a forward slash (`/`), it is assumed to refer to a directory in which a new file will be created. Otherwise it is assumed to refer to a specific file name, in which case the file will be created or overwritten *without warning*.
If the `task` argument is an `AMState` object, `savefile` will *not* default to the `AMState`'s *savefile* but must be supplied again; this is to prevent accidental file overwrites. If the first argument is a character, `savefile` defaults to `amstate` and therefore offers to seamlessly continue optimization runs.

`save.interval`
: [codenumeric(1)]
The inteval, in seconds, in between which to save the intermediate result to `savefile`. Ignored if `savefile` is `NULL`; set to `0` to only save at the end of optimization runs.

`backend`
: [`character(1)|BackendOptions`]
Refers to the back end used for optimization. Currently implemented and provided by automlr are `"random"`, `"irace"` and `"mbo"`. To list all backends, run `lsambackends`.

`max.walltime.overrun`
: [`numeric(1)`]
Defines a time in seconds for the automlr runtime beyond the `walltime` budget after which a learner function will be killed. Since the walltime (and other) budget is only checked in certain stages of the evaluation, this can sometimes lead to run times far greater than the walltime budget. Setting `max.walltime.overrun` to a finite value will agressively kill learner runs, potentially throwing away intermediate progress already made. There may still be a few seconds overhead runtime, especially when the learner code runs into a C function that can not be interrupted.

`max.learner.time`
: [`numeric(1)`]
Maximum time, in seconds, that one combined `train()`-`predict()` evaluation

of a learner may take after which it is aborted. Note that for performance measurements that use multiple evaluations, e.g. crossvalidation, a single datapoint takes a multiple of `max.learner.time` seconds.

`verbosity` `[integer(1)]`
Level of warning and info messages which to show.

0 Default: Only give essential warning messages and errors.

>=1 Output info about evaluated points.

>=2 Detailed warning messages about search space.

>=3 Detailed warning messages from learners.

>=4 Output from all learners.

>=5 Output memory usage stats.

>=6 Stop on learner error.

`new.seed` `[logical(1)]`
If `TRUE`, the random seed saved in the AMState object will not be used; instead the RNG state at time of the invocation will be used. The default behaviour (`FALSE`) is to use the saved rng state so that invocations with the same AMState object give a more deterministic result (insofar as execution time does not influence behaviour).
Warning: This is not yet tested and likely does not work with `Learners` that use external RNGs.

Value

`AMState` Object containing the result as well as info about the run. Use `amfinish` to extract the results.

Object members:

task [`Task`] The task being trained for.

measure [`Measure`] The measure for which is being ptimized.

budget [`numeric`] The budget of the current run.

spent [`numeric(4)`] The budget already spent.

searchspace [list of `Autolearner`] The `Learners` being considered for optimization.

prior [any] The prior of the current run. If the backend supports this, the prior is being updated during a run and can be given to another `automlr` invocation as an argument.

backend [`character(1)`] The backend of the optimization run.

creation.time [`numeric(1)`] The time at which the object was created.

finish.time [`numeric(1)`] The time at which the object was last touched by `automlr`; either by saving it on disk or by returning a result.

previous.versions [list of `AMState`] Backlog of previous invocations of `automlr` using this object. The objects in this list are reduced instances of `AMState`.

seed [`numeric`] The value of `.Random.seed` which to use for continuation.

Examples

```
## Not run:
library(mlr)
# almost minimal invocation. Will save progress to './iris.rds'.
automlr(iris.task, budget = c(evals = 1000), backend = "random",
  savefile = "iris")
```

```
> SOME RESULT

# optimize for another 1000 evaluations, loading the 'iris.rds' savefile
# automatically and saving back to it during evaluation.
automlr("iris", budget = c(evals = 2000))
> MORE RESULTS

## End(Not run)
```

---

autoWrapper — *Create a wrapper object for `autolearner`.*

Description

Build a wrapper object that can be plugged into `autolearner` to give wrapper functions to `buildLearners`.

Usage

```
autoWrapper(name, cpo, datatype, convertfrom = NULL)
```

Arguments

`name`	`[character(1)]` the name of the wrapper. Must not contain a `$` character.
`cpo`	`[CPO]` The cpo that will be attached to the learner and construct another `Learner`.
`datatype`	`[character(1)]` The data this wrapper operates on.
`convertfrom`	`[character(1)	NULL]` If this wrapper converts data from one type to another, “datatype” must be the target type, and “convertfrom” must be the source type. If the wrapper is an imputing wrapper, “convertfrom” must be “missings”, and “datatype” must be the type of columns that have their missings imputed. A given wrapper may only impute missings of one column type, even though it sees all columns.

---

buildLearners — *Take a list of autolearners and create a (big) mlr Learner object*

Description

Take a list of autolearners and create a (big) mlr Learner object

Usage

```
buildLearners(searchspace, task, verbosity = 0)
```

Arguments

searchspace	[list of Autolearner] List of autolearners.
task	[Task] The task that the searchspace is being created. buildLearners respects the task type, presence of NAs and type of covariates. Currently not supported: weights, request of probability assignment instead of class.
verbosity	[numeric(1)] Give detailed warnings. See the automlr parameter.

---

getSearchspace *Retrieve a suggested search space of the given learner*

Description

Learners created with buildLearners have a $searchspace slot that can be accessed with this function.

Usage

```
getSearchspace(learner)
```

Arguments

learner	[Learner] Learner

---

handleInterrupts *Abort evaluation of an expression when Ctrl-C is pressed*

Description

Evaluate an expression and abort its evaluation when Ctrl-C is pressed, without killing the program.

Usage

```
handleInterrupts(expr, onInterrupt)
```

Arguments

expr	[any] The expression to evaluate
onInterrupt	[any] The expression to evaluate and return when Ctrl-C was pressed during evaluation of expr.

Value

any the result of expr if it evaluated successfully, onInterrupt if an interrupt was caught.

isambackend — *Check if the given name is a valid automlr backend*

Description

Test whether whether all necessary functions for a backend are implemented.

Usage

```
isambackend(name)
```

Arguments

`name`	`[character(1)]` Name the name of the backend to check for.

lsambackends — *List the backends currently implemented which can be passed to `automlr`.*

Description

This lists all the backends that can be used by `automlr`, even user defined ones.

Usage

```
lsambackends(fenv = NULL)
```

Arguments

`fenv`	`[NULL]` Should be NULL

Examples

```
lsambackends()
```

`makeAMExoWrapper` *Create an mlr learner object that smoothes out the parameter space.*

Description

So what are we doing here? The AMExoWrapper mostly handles parameter space magic. Specifically, this is:

- introduce parameters that chooses which wrappers are used, and in which sequence
- introduce parameters that control whether the wrapper(s) are used to facilitate compatibility of learners with the data, i.e. remove missing values, convert data types that are not supportet
- set "pseudoparameters", which are some special variable values that the 'requires'-expressions of other parameters can use to specify that some parameters are only relevant in the presence of certain data types.
- remove parameters that always take on a single value (maybe dependent on a 'requires' from the search space) and set them internally before calling the wrapped model.
  - bonus: if a parameter name contains a suffix of .AMLRFIX#, where # is a number, it will be stripped from the parameter name. This way it is possible to define alternative ranges for a parameter, depending on requirements.

In the following, "XXX" and "YYY" will always be one of "numerics", "ordereds", "factors". The parameters that are introduced and exposed to the outside are:

- automlr.convert: Whether to do any conversion of input data.
- automlr.convert.XXX: Whether to convert input data of type "XXX".
- automlr.convert.XXX.to: What type to convert "XXX"-typed data into, if there is any choice.
- automlr.wrapafterconvert.XXX: Whether to apply preprocessing wrapper after conversion of data originally of type "XXX".
- automlr.wconverting.XXX.to.YYY: Which conversion wrapper to use for conversion of data from type "XXX" to type "YYY".
- automlr.impute: Whether to do imputation of missing values.
- automlr.missing.indicators: Whether imputation is supposed to introduce "factor" typed missing indicator variables.
- automlr.wimputing.XXX: Which imputation wrapper to use for imputation of "XXX"-typed data, if there is any choice.
- automlr.preproc.XXX: Which non-converting, non-imputing wrappers are used for preprocessing, with values of the format `outermostWrapper$wrapper..$wrapper$innermostwrapper`.

The following parameters can be used by wrappers and learners in their `$requires`-parameter; they will be replaced here:

- automlr.missing.indicators: Should be used by the imputing wrapper. Indicates whether imputation is supposed to introduce "factor" typed missing indicator variables.
- automlr.has.XXX: May be used by learners: Indicates that the "XXX" type is present in the data. Besides the types, mentioned above, "XXX" may also be "missings".
- automlr.targettype: One of "oneclass", "twoclass", "multiclass".

Usage

```
makeAMExoWrapper(modelmultiplexer, wrappers, taskdesc, missings, canHandleX,
  allLearners, modelTuneParsets)
```

Arguments

modelmultiplexer	[ModelMultiplexer] A modelmultiplexer object that should have a `$searchspace` slot.
wrappers	[list] A named list of wrappers. Names must not contain `$`-character.
taskdesc	[TaskDesc] The “TaskDesc” object of the task to be optimized over.
missings	[logical] A logical, with names according to the *present* feature data types (a subset of numerics, factors, ordered) indicating whether the columns in question have any missing values.
canHandleX	[list] A named list that maps "missings", "numerics", "factors", and "ordered" to a vector of learner names that can handle the respective data.
allLearners	[character] The list of all learner names.
modelTuneParsets	[list of ParamSet] List of each learner’s ParamSet, indexed by learner ID.

Value

`AMExoWrapper` A `Learner` that incorporates the wrappers and learners suitable for mlr learners.

The slot `$searchspace` should be used as `ParamSet` to tune parameters over.

`makeBackendconfIrace` *irace backend configuration*

Description

Create an `AutomlrBackendConfig` object that can be fed to `automlr` to perform optimization with the "irace" backend.

Usage

```
makeBackendconfIrace(nbIterations = 10, newpopulation = 10,
  resampling = hout)
```

Arguments

`nbIterations`	`[integer(1)]` Thinning of sampling distribution happens as if irace expected to run for `nbIterations` generations.
`newpopulation`	`[integer(1)]` Size of the population, *additinal to* the `2 + log2(dimParams)` elite size.
`resampling`	`[ResampleDesc]` resampling to evaluate model performance.

`makeBackendconfMbo` *mbo backend configuration*

Description

Create an `AutomlrBackendConfig` object that can be fed to `automlr` to perform optimization with the "mbo" backend.

Usage

```
makeBackendconfMbo(focussearch.restarts = 1, focussearch.maxit = 5,
  focussearch.points = 1000, mbo.save.mode = TRUE, resampling = hout)
```

Arguments

`focussearch.restarts`	`[integer(1)]` number of restarts to perform in focussearch surrogate model optimizer
`focussearch.maxit`	`[integer(1)]` number of iterations for one focussearch round
`focussearch.points`	`[integer(1)]` number of points to sample in focussearch.
`mbo.save.mode`	`[logical(1)]` Simplify search space for mbo backend. You should probably not change the default.
`resampling`	`[ResampleDesc]` resampling to evaluate model performance.

makeBackendconfRandom *random backend configuration*

Description

Create an AutomlrBackendConfig object that can be fed to automlr to perform optimization with the "random" backend.

Usage

```
makeBackendconfRandom(max.iters.per.round = 100, resampling = cv5)
```

Arguments

max.iters.per.round	[integer(1)] Number of iterations to perform between timeout checks. Do not set too small; you probably don't want to change this.
resampling	[ResampleDesc] resampling to evaluate model performance.

makeFailImputationWrapper *Create an mlr learner that catches errors and replaces them with the result of the simplest possible learner.*

Description

When a learner throws an error, its performance is treated by mlr as a "missing value". This behaviour is not always optimal, especially for tuning purposes, since a learner that sometimes fails but otherwise gives good results could still be better than a learner that never fails but gives awful results.

Usage

```
makeFailImputationWrapper(learner)
```

Arguments

learner	[Learner] The learner to be wrapped.

Value

FailImputationWrapper	A Learner that catches errors and returns a dummy model if an error occurs.

makeModelMultiplexerParamSetEx

Like mlr's `makeModelMultiplexerParamSet`*, but respecting requirements.*

Description

This is necessary since `makeModelMultiplexerParamSet` does not handle parameter requirements correctly.

Usage

```
makeModelMultiplexerParamSetEx(multiplexer, modelParsets, origParamNames)
```

Arguments

`multiplexer`	[`ModelMultiplexer`] The model multiplexer to use to create the ParamSet.
`modelParsets`	[`ParamSet`] The list of param sets that are used to create the mmps.
`origParamNames`	[list of `character`] Maps the IDs of the individual `Learners` to the set of parameters that are actually used and need to be multiplexed.

Value

`ModelMultiplexer` The `ModelMultiplexer` with repaired requirements.

makeTimeconstraintWrapper

Create an mlr learner that limits the runtime of a train/predict cycle.

Description

When max.learner.time, given to automlr, is overrun, we want to give an error. If the first resampling was an error, the other resamplings should also give errors without starting the run. Otherwise they should themselves run (with the correct timeout).

Usage

```
makeTimeconstraintWrapper(learner, time, timeFirstIter = NULL)
```

Arguments

`learner`	`[Learner]` The learner to be wrapped.
`time`	`[numeric(1)]` The maximum runtime, for `train` and `predict` *combined*, after which to abort.
`timeFirstIter`	`[numeric(1)]` If given, this is the maximum runtime for the first iteration in a resampling, given that (1) the evaluation happens inside `resample` and (2) the resampling will go through more than one iteration and (3) the the resampling is not being parallelized.

Value

`TimeconstraintWrapper` A `Learner` that runs only for the given time and will then return either an error or a dummy learner.

`mlrLearners` *A list of learners with corresponding* `par.set`*s that can be searched over.*

Description

This is the complete search space as suggested by the automlr package. It includes learners and wrappers.

Usage

```
mlrLearners
```

Format

An object of class `list` of length 0.

See Also

Other searchspace: `mlrLearnersNoWrap`, `mlrLightweightNoJava`, `mlrLightweightNoWrapNoJava`, `mlrLightweightNoWrap`, `mlrLightweight`, `mlrWrappers`

`mlrLearnersNoWrap` *A list of learners with corresponding* `par.set`*s that can be searched over.*

Description

This is a list of learners that the infinitely wise developer of this package collected *himself* that work well with automlr. `mlrLearnersNoWrap` does not include wrappers and therefore does not perform preprocessing and does not include meta-methods.

Usage

```
mlrLearnersNoWrap
```

Format

An object of class `list` of length 71.

See Also

Other searchspace: `mlrLearners`, `mlrLightweightNoJava`, `mlrLightweightNoWrapNoJava`, `mlrLightweightNoWrap`, `mlrLightweight`, `mlrWrappers`

`mlrLightweight` *A list of fast learners with corresponding* `par.sets` *that can be earched over.*

Description

This is a list similar to `mlrLearners`, only it excludes the slowest learners. This decreases average evaluation time significantly and also makes the search space small enough for the `mbo` backend.

Usage

```
mlrLightweight
```

Format

An object of class `list` of length 0.

See Also

Other searchspace: `mlrLearnersNoWrap`, `mlrLearners`, `mlrLightweightNoJava`, `mlrLightweightNoWrapNoJava`, `mlrLightweightNoWrap`, `mlrWrappers`

`mlrLightweightNoJava` *A list of fast learners with corresponding* `par.sets` *that can be earched over.*

Description

This is a list similar to `mlrLightweight`, but it also excludes java based learners. This makes it possible to use the “fork” backend for timeouts.

Usage

```
mlrLightweightNoJava
```

Format

An object of class `list` of length 0.

See Also

Other searchspace: mlrLearnersNoWrap, mlrLearners, mlrLightweightNoWrapNoJava, mlrLightweightNoWrap, mlrLightweight, mlrWrappers

---

mlrLightweightNoWrap *A list of fast learners with corresponding* par.sets *that can be earched over.*

Description

This is a list similar to mlrLearnersNoWrap, only it excludes the slowest learners. This decreases average evaluation time significantly and also makes the search space small enough for the mbo backend.

Usage

```
mlrLightweightNoWrap
```

Format

An object of class list of length 68.

See Also

Other searchspace: mlrLearnersNoWrap, mlrLearners, mlrLightweightNoJava, mlrLightweightNoWrapNoJava, mlrLightweight, mlrWrappers

---

mlrLightweightNoWrapNoJava *A list of fast learners with corresponding* par.sets *that can be earched over.*

Description

This is a list similar to mlrLightweightNoWrap, but it also excludes java based learners. This makes it possible to use the "fork" backend for timeouts.

Usage

```
mlrLightweightNoWrapNoJava
```

Format

An object of class list of length 62.

See Also

Other searchspace: mlrLearnersNoWrap, mlrLearners, mlrLightweightNoJava, mlrLightweightNoWrap, mlrLightweight, mlrWrappers

---

`mlrWrappers` *A list of wrappers with corresponding* `par.set`*s that can be searched over.*

Description

This is a list of wrappers that can be used as part of a searchspace. Currently this only includes a preprocessing wrapper, but may in future also include some meta-methods.

Usage

```
mlrWrappers
```

Format

An object of class `list` of length 0.

See Also

Other searchspace: `mlrLearnersNoWrap`, `mlrLearners`, `mlrLightweightNoJava`, `mlrLightweightNoWrapNoJava`, `mlrLightweightNoWrap`, `mlrLightweight`

`myCheckLearner` *Turn learner id string into learner object, if necessary*

Description

If the given learner is a `character(1)` identifying a learner, the corresponding learner is generated using `makeLearner`.

Usage

```
myCheckLearner(learner)
```

Arguments

`learner`	`[character(1)|Learner]` A character scalar or learner object.

print.AMObject *Give some cute info about a given AMState*

Description

Optionally give a little or a lot (if `verbose == TRUE`) of info.

Usage

```
## S3 method for class 'AMObject'
print(x, verbose = FALSE, ...)
```

Arguments

`x`	`[AMState|AMResult]` What to print
`verbose`	`[logical(1)]` Print detailed info
`...`	ignored

quickSuspendInterrupts *Quickly evaluate an expression that won't be killed by Ctrl-C*

Description

Like `suspendInterruptsFor` but more lightweight. Does not allow for internal handleInterrupts.

Usage

```
quickSuspendInterrupts(expr)
```

Arguments

`expr`	`[any]` Expression to evaluate that must not be interrupted by Ctrl-C and timeout events.

Value

`any` The result of evaluating `expr`.

registerBackend *Register a new backend*

Description

Make it possible to invoke automlr() with your own custom backend. Besides providing a creator function as an argument here, you need to make the S3 methods amaddprior, amgetprior, amsetup, amoptimize and amresult available for an object of class am[backendname].

Usage

```
registerBackend(name, creator)
```

Arguments

name	[character(1)] The name of your backend. The user will be able to invoke automlr with argument backend set to this name to perform optimization using your backend.
creator	[function] A function that can be run with no arguments and returns an object that will be given to the amsetup function of the backend. It may have more options that allow the user to set options for the backend. Note: The function that is given as an argument should not be used directly by the user. Instead, the return value of registerBackend must be used for that purpose.

Value

function The function that can be used to create a "BackendOptions" object that can then be given as the backend parameter to automlr.

Note

Use auxiliary function argsToList to return a list of all given formal arguments.

Examples

```
## Not run:
makeMyBackendOptions = registerBackend("mybackend",
    function(opt1 = 1, opt2 = "a") {
      list(opt1 = opt1, opt2 = opt2)
    })

# equivalent:
makeMyBackendOptions = registerBackend("mybackend",
    function(opt1 = 1, opt2 = "a") {
      argsToList()
    })

# the following works if you also defined amsetup.mybackend,
# amoptimize.mybackend, etc.
amresult1 = automlr(iris.task, budget = c(evals = 10),
    backend = makeMyBackendOptions(opt1 = 2, opt2 = "b"))
```

```
# the following also works; it uses the defaults as defined in the funciton
# header.
amresult2 = automlr(iris.task, budget = c(evals = 10), backend = "mybackend")

## End(Not run)
```

`runWithTimeout` *Run a given expression with a given (walltime) timeout.*

Description

Runs ‘expr‘ with timeout ‘time‘ (in seconds) and returns a logical(1) indicating success. The return value contains the result of the evaluated expression, as well as information about the execution.

`runWithTimeout` can also be called in a nested fashion.

Usage

```
runWithTimeout(expr, time, throwError = FALSE, backend)
```

Arguments

`expr`	[any] The expression that will be run with the given timeout.
`time`	`[numeric(1)]` The runtime, in seconds, after which to abort evaluation of expr. If this is smaller or equal zero, expr will not be run and a timeout will be reported.
`throwError`	`[logical(1)]` If `TRUE`, throw an error on timeout instead of just returning `FALSE`.
`backend`	`[character(1)]` One of “native”, “fork”: which backend to use. If not given, the backend set by `setDefaultRWTBackend` is used. If this was not set, the default is “native”.

Value

`list` a list with three items:

result	contains the result of the evaluated expression `expr`. If a timeout occurred and `throwError` is `FALSE`, this will be `NULL`.
timeout	if `throwError` is `FALSE`, this is `TRUE` if a timeout occurred, `FALSE` otherwise. If `throwError` is `TRUE`, this is always `TRUE`.
elapsed	contains the time spent evaluating `expr`. The value is guaranteed to be lower or equal to `time` if the returned `$timeout` is `FALSE`, and is greater than `TIME` otherwise.

`setDefaultRWTBackend` *Set the default runWithTimeout backend*

Description

Sets the default backend used by `runWithTimeout`.

Usage

```
setDefaultRWTBackend(backend)
```

Arguments

`backend`	`[character(1)]` The backend to use by `runWithTimeout`. The two implemented backends are `"fork"` (default) and `"native"`. The fork backend uses the operating system's `fork` functionality. This is safer and protects against crashes of most kind, but it prevents communication by reference (e.g. S4 objects or environments) and it only works on UNIX kinds of operating systems. The native backend uses the crude and unreliable R internal `setTimeLimit` functionality to facilitate timeout. This is system independent.

`sp` *Define the searchspace parameter in a short form*

Description

This function is used to define the search space related to a given learner. The priority here is that the function should be saving space: Much of the information about a parameter is inferred from the learner itself; In principle only the name, the type, and a range of a parameter are needed.

However, to get notifications about changes in the mlr package, also all the parameters that are not given should be referenced with an `sp()` of type `"def"`; otherwise, a warning will be given upon instantiation of the learner.

Usage

```
sp(name, type = "real", values = NULL, trafo = NULL, id = NULL,
  special = NULL, req = NULL, dim = 1, version = NULL)
```

Arguments

`name`	`[character(1)]` The name of the parameter which must match the id of the `Param` it refers to. May be suffixed with `.AMLRFIX#`, where `#` is a number, to expose one varible to the outside with different search space depending on requirements.
`type`	`[character(1)]` The type of the parameter. One of: `real` (numeric), `int` (integer), `cat` (discrete), `bool` (logical), `fix` (fixed value of whatever type), `def` (using default value / not setting the value), `fixdef` (fixed value, but warn if this is not the default)

`values`	`[numeric|character|list]` If `type` is `real` or `int`, this gives the lower and upper bound. If `type` is `cat`, it is a character vector of possible values. If `type` is `fix`, only one value (the one to be fixed) must be given. If `type` is codebool or `def`, it must be `NULL`. It is possible to give expression values instead of constant values, usually using `quote`. These expressions can involve the special values `n` (number of observations), `p` (number of features), the automlr pseudoparameters described in the docs of `makeAMExoWrapper`, and `PARAM.x` to refer to the value of parameter `x`. If at least one expression is given, `values` needs to be a list containing the expressions and singleton vectors of the appropriate type. A special value for "def" type parameters is "##", which leads to using the current default of the function without specifying it.
`trafo`	`[character|function|NULL]` May be "exp" for exponential transformation, or "invexp" for exponential transformation of the difference of the value and its upper bound. Transforms integer parameters in a smart way. Only applies for `real` and `int` values. May also be an R function.
`id`	`[character(1)|NULL]` May be given to identify parameters of different learners having similar effects in similar learners. This currently has no effect beyond warnings if different parameters with the same `id` have different value ranges.
`special`	`[character(1)|NULL]` May be `NULL`, `"dummy"` or `"inject"`. Set this to `"dummy"` if this is a dummy variable that will be hidden from the learner itself but visible to the outside. Set this to `"inject"` to create the parameter in the learners `ParamSet` if it does not exist.
`req`	`[language|NULL]` A requirement for the variable to have effect. May reference other parameters of the learner, as well as the automlr pseudoparameters described in `makeAMExoWrapper`. Must not call any parameter values as functions!
`dim`	`[integer(1)]` The number of dimensions of this variable.
`version`	`[character(1)|NULL]` Version of MLR to apply this parameter to. If not `NULL`, this must be a `character` made up of a comparison operator (one of ">", "<", ">=", "<=", or "==") and an MLR version number of the form "MAJOR.MINOR". If the mlr version that was found does not satisfy the requirement, the parameter will be ignored.

`suspendInterruptsFor` *Evaluate an expression that won't be killed by Ctrl-C*

Description

This function evaluates its first argument and prevents the user from killing the evaluation by pressing Ctrl-C. A timeout may be specified such that pressing Ctrl-C twice in the given interval will actually kill the evaluation. Together with handleInterrupts, this enables one to selectively abort part of an evaluation without killing the whole program.

The suspension is not complete, it looks like some socket operations override this. In any case, expensive operations should not happen while interrupts are suspended.

Calls to suspendInterruptsFor can be nested; interrupts will be disabled by the first call and only be reenabled when the first call exits.

Usage

```
suspendInterruptsFor(expr, hardKillInterval = 0)
```

Arguments

`expr`	`[any]` an expression to be evaluated
`hardKillInterval`	`[numeric(1)]` The interval (in seconds) in which a second Ctrl-C press after a first one will kill the run regardless. Ctrl-C presses are only evaluated within `handleInterrupts` calls; therefore this value is a minimum, the maximum being the time interval between exiting one `handleInterrupts` call and entering the next one.

Value

`any` The result of the evaluation of `expr`.

Examples

```
## Not run:
# This example will lapply expensive_function(i). If the user wants to abort
# one of these runs without killing the whole run, he can press Ctrl-C --
# the 'result' array will then record an "aborted" character. If the user
# accidentally presses Ctrl-C in between runs of expensive_function(),
# nothing happens.
suspendInterruptsFor({
    result = lapply(1:10, function(i) {
        handleInterrupts(expensive_function(i), "aborted")
      })
    }, hardKillInterval = 0.5)

## End(Not run)
```

Index