Package 'mosmafs'

April 5, 2020

Title Multi-Objective Simultaneous Model and Feature Selection

Description Performs simultaneous hyperparameter tuning and feature selection through both single-objective and multi-objective optimization as described in Binder, Moosbauer et al. (2019) <arXiv:1912.12912>. Uses the 'ecr'-package as basis but adds mixed integer evolutionary strategies and multi-fidelity functionality as well as operators specific for the problem of feature selection.

URL https://github.com/compstat-lmu/mosmafs

BugReports https://github.com/compstat-lmu/mosmafs/issues

License MIT + file LICENSE

Encoding UTF-8

ByteCompile yes

Version 0.1.2

VignetteBuilder knitr

Depends ecr (>= 2.1.0)

Imports BBmisc, checkmate (>= 1.9.0), ParamHelpers, MASS, smoof, mlrCPO (>= 0.3.4), mlr, parallelMap

Suggests knitr, ggplot2, magrittr, testthat, rpart, parallel, praznik, mlrMBO, emoa, DiceKriging, rgenoud, randomForest, digest

RoxygenNote 7.0.2

Collate 'customnsga2.R' 'datagen.R' 'ecrshims.R' 'evalmosmafs.R' 'filtermat.R' 'objective.R' 'utils.R' 'operators.R' 'plotting.R' 'selectorcpo.R' 'terminators.R' 'zzz.R'

NeedsCompilation no

Author Martin Binder [aut, cre], Susanne Dandl [aut], Julia Moosbauer [aut]

Maintainer Martin Binder <developer.mb706@mb706.com>

Repository CRAN

Date/Publication 2020-04-05 21:40:03 UTC

R topics documented:

`availableAttributes` *List Attributes that can be Aggregated*

Description

Looks at the first individuum in the first generation and returns its attributes. If `check` is TRUE it checks that the log object is consistent and throws an error if not. "Consistent" here means that all individuals in all generations have the same attributes.

Usage

```
availableAttributes(log, check = FALSE)
```

Arguments

`log`	[ecr_logger] ecr log object.
`check`	[logical(1)] whether to check consistency.

Value

[character] attributes of individuals in population.

`clonetask` *Replace Data in Task with new Data*

Description

Create new task identical to the old one, but with `newdata` instead of old data. This should either preserve the `orig.features` of the original task, or should add new noise-features, in which case `orig.features` should mark the features that correspond to the full original task.

Usage

```
clonetask(
  task,
  newdata,
  newid,
  orig.features = rep(TRUE, ncol(newdata) - length(getTaskTargetNames(task)))
)
```

Arguments

task	[Task] mlr Task to use.
newdata	[data.frame] data to replace task data with; must include the target column with same name.
newid	[character(1)] ID to use for new Task.
orig.features	[logical] features that correspond to original task's data.

Value

Task

See Also

Other Artificial Datasets: create.hypersphere.data(), create.linear.data(), create.linear.toy.data(), create.regr.task(), task.add.permuted.cols(), task.add.random.cols()

---

collectResult *Collect Result Information*

Description

Merge both log and log.newinds data for a complete data.frame with information about progress (both on training data and on holdout data) and ressource usage.

Usage

```
collectResult(
  ecr.object,
  aggregate.perresult = list(domHV = function(x) computeHV(x, ref.point)),
  aggregate.perobjective = list("min", "mean", "max"),
  ref.point = smoof::getRefPoint(ecr.object$control$task$fitness.fun),
  cor.fun = cor
)
```

Arguments

ecr.object	[MosmafsResult] slickEcr() result to analyse.
aggregate.perresult	[list] list of functions to apply to fitness and holdout fitness. Every entry must either be a character(1) naming the function to use, or a function, in which that entry must have a name. Each function must return exactly one numeric value when fed a fitness matrix of one generation. This is ignored for single-objective runs.
aggregate.perobjective	[list] list of functions to apply to fitness and holdout fitness matrix rows, formatted like aggregate.perresult. Each function must return exactly one numeric value when fed a fitness vector.

ref.point	[numeric] reference point to use for HV computation.
cor.fun	[function] function to use for calculation of correlation between objective and holdout. Must take two numeric arguments and return a numeric(1).

Value

data.frame

Examples

```
library(mlrCPO)

# Setup of optimization problem
ps.simple <- pSS(
 a: numeric [0, 10],
 selector.selection: logical^10)

mutator.simple <- combine.operators(ps.simple,
 a = mutGauss,
 selector.selection = mutBitflipCHW)

crossover.simple <- combine.operators(ps.simple,
 a = recSBX,
 selector.selection = recPCrossover)

initials <- sampleValues(ps.simple, 30, discrete.names = TRUE)

fitness.fun <- smoof::makeMultiObjectiveFunction(
 sprintf("simple test"),
 has.simple.signature = FALSE, par.set = ps.simple, n.objectives = 2,
 noisy = TRUE,
 ref.point = c(10, 1),
 fn = function(args, fidelity = NULL, holdout = FALSE) {
   pfeat <- mean(args$selector.selection)
   c(perform = args$a, pfeat = pfeat)
 })
fitness.fun.single <- smoof::makeMultiObjectiveFunction(
 sprintf("simple test"),
 has.simple.signature = FALSE, par.set = ps.simple, n.objectives = 1,
 noisy = TRUE,
 ref.point = c(10),
 fn = function(args, fidelity = NULL, holdout = FALSE) {
   propfeat <- mean(args$selector.selection)
   c(propfeat = propfeat)
 })

# Run NSGA-II
results <- slickEcr(fitness.fun = fitness.fun, lambda = 10, population = initials,
 mutator = mutator.simple, recombinator = crossover.simple, generations = 10)

# Collect results
colres <- collectResult(results)
```

```
print(colres)
```

`combine.operators` *Combine ECR-Operators*

Description

Combine operators to be applied to individuals that conform to parameter set `param.set`. Parameters are the `param.set`, and the names / types of params with the operator to use. Parameter groups that use a single operator can be defined using .params.<groupname> = [character].

Say `param.set` has three logical params 'l1', 'l2', 'l3' and two numeric params 'n1', 'n2'. To use operatorA for 'l1' and 'l2', operatorB for 'l3', and operatorC for all numeric params, call `combineOperator(param.set,.params.group1 = c("l1","l2"),group1 = operatorA,l3 = operatorB,numeric = operatorC)`.

Use arguments by types, names of parameters, or group name. Valid types are 'numeric', 'logical', 'integer', 'discrete'. Operators given for groups or individual parameters supercede operators given for types.

Strategy parameters can be created by using .strategy.<groupname|parametername|type>. They must be a *function* taking a named list of parameter values (i.e. an individuum) as input and return a named list of parameter values to be given to the respective group's / parameter's or type's operator. If, in the example above, `operatorA` has a parameter `sigma` that should also be treated as a parameter under evolution (and in fact be equal to `l3`), then the above call would become `combineOperator(param.set,.params.group1 = c("l1","l2"),group1 = operatorA,.strategy.group1 = function(x) list(sigma = x$l3),l3 = operatorB,numeric = operatorC)`.

If .binary.discrete.as.logical is TRUE, then binary discrete params are handled as logical params.

Operators for logical parameters must have only one argument. Operators for discrete parameters must have an additional argument 'values'. Operators for continuous or integer parameters must have an additional argument 'lower', 'upper'.

Use the ecr::setup function to set parameters for operators ("currying").

Usage

```
combine.operators(param.set, ..., .binary.discrete.as.logical = TRUE)
```

Arguments

`param.set`	[ParamSet] `ParamSet` that defines the search space.
`...`	additional parameters. See description.
`.binary.discrete.as.logical`	[logical(1)] whether to treat binary discrete parameters as `logical` parameters and use bitwise operators.

Value

`ecr_operator` ecr operator.

Examples

```
library(mlrCPO)

# Create parameter set
ps <- pSS(
logi: logical,
disc: discrete[yes, no],
discvec: discrete[letters]^3,
numer: numeric[0, 10])

# Define mutators for groups of parameters
combo.mut <- combine.operators(ps,
.params.group1 = c("logi", "disc"), # define group for which same mutator is used
group1 = ecr::setup(mutBitflip, p = 1), # set probability for mutation to 1
discrete = mutRandomChoice, # define operator for all other discrete parameters
numer = mutGauss) # specific operator for parameter numer

combo.mut(list(logi = FALSE, disc = "yes", discvec = c("a", "x", "y"),
numer = 2.5))

# Define mutator with strategy parameter
combo.strategy <- combine.operators(ps,
logical = ecr::setup(mutBitflip, p = 0),
discrete = mutRandomChoice,
numeric = mutGauss,
.strategy.numeric = function(ind) {
if (ind$disc == "yes") {
return(list(p = 1L))
} else {
return(list(p = 0L))
}
})

combo.strategy(list(logi = FALSE, disc = "no", discvec = c("a", "x", "y"),
numer = 2.5))

# Define recombinators for groups of parameters
combo.rec <- combine.operators(ps,
.params.group1 = c("logi", "disc"), # define group for which same mutator is used
group1 = recPCrossover,
discrete = recPCrossover,
numer = recGaussian)

combo.rec(list(list(logi = FALSE, disc = "no", discvec = c("a", "x", "y"),
numer = 2.5), list(logi = TRUE, disc = "yes", discvec = c("c", "e", "g"),
numer = 7.5)))
```

`constructEvalSetting` *Parameter Configuration of Mosmafs*

Description

Create a SMOOF function for parameter configuration of mosmafs, with parameter set.

The resulting function takes a list of values according to its `getParamSet()`. Additionally the list can contain an $INSTANCE, an integer between 1 and 1000. If it is not given, the instance will be chosen randomly. It corresponds to the resampling instance to use if `fixed.ri` is `TRUE`.

Usage

```
constructEvalSetting(
  task,
  learner,
  ps,
  measure = getDefaultMeasure(task),
  worst.measure = NULL,
  cpo = NULLCPO,
  nfeat = getTaskNFeats(task %>>% cpo),
  evals = 1e+05,
  outer.resampling = makeResampleDesc("CV", iters = 10, stratify = TRUE),
  savedir = NULL
)
```

Arguments

`task`	[Task] the task to optimize.
`learner`	[Learner] the learner to optimize.
`ps`	[ParamSet] the parameter set of the learner (and `cpo`) alone.
`measure`	[Measure] measure to optimize.
`worst.measure`	[numeric(1)] worst value for measure to consider, for dominated hypervolume calculation. Will be extracted from the given measure if not given, but will raise an error if the extracted (or given) value is infinite.
`cpo`	[CPO] cpo to prepend feature selection.
`nfeat`	[integer(1)] number of features.
`evals`	[integer(1)] number of evals to perform. Note this concerns fidelity evaluations (i.e. single CV folds). When not using multifid the number of points evaluated is 1/10th the `evals` value.
`outer.resampling`	outer resampling to use.
`savedir`	[character(1) | NULL] the directory to save every trace to. If this is `NULL` (the default) evaluations are not saved.

Value

`function` a smoof function.

`cpoSelector` *CPO that Selects Features*

Description

CPO that Selects Features

Usage

```
cpoSelector(
  selection,
  id,
  export = "export.default",
  affect.type = NULL,
  affect.index = integer(0),
  affect.names = character(0),
  affect.pattern = NULL,
  affect.invert = FALSE,
  affect.pattern.ignore.case = FALSE,
  affect.pattern.perl = FALSE,
  affect.pattern.fixed = FALSE
)
```

Arguments

`selection`	[logical] Logical vector indicating if a features was selected or not. Must have the same length as number of features.
`id`	[`character(1)`] id to use as prefix for the CPO's hyperparameters. this must be used to avoid name clashes when composing two CPOs of the same type, or with learners or other CPOS with hyperparameters with clashing names.
`export`	[`character`] Either a character vector indicating the parameters to export as hyperparameters, or one of the special values "export.all" (export all parameters), "export.default" (export all parameters that are exported by default), "export.set" (export all parameters that were set during construction), "export.default.set" (export the intersection of the "default" and "set" parameters), "export.unset" (export all parameters that were *not* set during construction) or "export.default.unset" (export the intersection of the "default" and "unset" parameters). Default is "export.default".
`affect.type`	[`character` | `NULL`] Type of columns to affect. A subset of "numeric", "factor", "ordered", "other", or `NULL` to not match by column type. Default is `NULL`.

affect.index [numeric]
Indices of feature columns to affect. The order of indices given is respected. Target column indices are not counted (since target columns are always included). Default is integer(0).

affect.names [character]
Feature names of feature columns to affect. The order of names given is respected. Default is character(0).

affect.pattern [character(1) | NULL]
grep pattern to match feature names by. Default is NULL (no pattern matching)

affect.invert [logical(1)]
Whether to affect all features *not* matched by other affect.* parameters.

affect.pattern.ignore.case
[logical(1)]
Ignore case when matching features with affect.pattern; see grep. Default is FALSE.

affect.pattern.perl
[logical(1)]
Use Perl-style regular expressions for affect.pattern; see grep. Default is FALSE.

affect.pattern.fixed
[logical(1)]
Use fixed matching instead of regular expressions for affect.pattern; see grep. Default is FALSE.

Value

[CPO]

Examples

```
library("mlr")
library("mlrCPO")

# Dataset has originally four features
iris.task$task.desc$n.feat

iris.task.subset = iris.task %>>% cpoSelector(c(TRUE, TRUE, FALSE, FALSE))

# Now only two were selected
iris.task.subset$task.desc$n.feat
```

---

create.hypersphere.data

Create Hypersphere Data

Description

Creates hypersphere data with X as a `n * dim` matrix of sampled columns from `dist`. `dist` must be a function n -> vector length (n) and should (probably) sample randomly to create `X`.

Y is a vector with entries `Y[i] = +1` if the L_norm of `X[i,]` is < radius^norm, and `Y[i] = -1` otherwise.

Usage

```
create.hypersphere.data(
  dim,
  n,
  dist = function(x) runif(x, -1, 1),
  norm = 2,
  radius = 1
)
```

Arguments

`dim`	[integer(1)] number of columns to create.
`n`	[integer(1)] number of sample to create.
`dist`	[function] function `n -> numeric(n)` that is used to sample points dimension-wise.
`norm`	[numeric(1)] Norm exponent.
`radius`	[numeric(1)] Radius to check against.

Value

list(X = [Matrix], Y = [vector], orig.features = logical)

See Also

Other Artificial Datasets: `clonetask()`, `create.linear.data()`, `create.linear.toy.data()`, `create.regr.task()`, `task.add.permuted.cols()`, `task.add.random.cols()`

`create.linear.data` *Create Linear Model Data*

Description

Create linear model data `Y = X * beta + epsilon` with X as a `n * p` matrix of multivariate normal distributed rows with covariance matrix

```
1     rho rho^2 rho^3 ... rho^p
rho     1 rho   rho^2 ... rho^(p-1)
rho^2 rho   1   rho ...   rho^(p-2)
...
rho^p ...
```

epsilon is standard normally distributed and beta[i] = beta0 * q ^ (i -1) for i = 1,..., p.

If permute == TRUE, columns of X as well as beta are permuted before the linear model equation is evaluated to generate Y. These permuted values are also the ones returned in the result. $orig.features are the features with beta > 1 / sqrt(n).

Usage

```
create.linear.data(n, p, q = exp(-1), beta0 = 1, rho = 0, permute = TRUE)
```

Arguments

n	[integer(1)] number of rows to generate.
p	[integer(1)] number of columns to generate.
q	[numeric(1)] attenuation factor for beta coefficients.
beta0	[numeric(1)] size of first coefficient.
rho	[numeric(1)] parameter for correlation matrix.
permute	[logical(1)] whether to permute columns of X and coefficient vector (beta).

Value

list(X=[Matrix], Y=[vector], beta = [vector], orig.features = logical)

See Also

Other Artificial Datasets: clonetask(), create.hypersphere.data(), create.linear.toy.data(), create.regr.task(), task.add.permuted.cols(), task.add.random.cols()

---

create.linear.toy.data

Linear Toy Data

Description

Based on Weston (2000) Feature Selection for SVMs.

Creates matrix X and vector Y with six dimensions out of 202 relevant and equal probability of y = 1 or -1.

With a prob of 0.7 we draw xi = y * norm(i,1) for i = 1, 2, 3 and xi = norm(0,1) for i = 4, 5, 6. Otherwise: xi = norm(0,1) for i = 1, 2, 3 and xi = y * norm(i -3,1) for i = 4, 5, 6.

All other features are noise.

Usage

```
create.linear.toy.data(n)
```

Arguments

n	[integer(1)] number of samples to draw.

Value

list(X = [Matrix], Y = [vector], orig.features = logical)

See Also

Other Artificial Datasets: clonetask(), create.hypersphere.data(), create.linear.data(), create.regr.task(), task.add.permuted.cols(), task.add.random.cols()

---

create.regr.task *Create mlr-Task from Data*

Description

Both create.regr.task and create.classif.task take a numeric target column Y, but create.classif.task binarizes it on cutoff to create a classification task, while create.regr.task creates a regression task.

Usage

```
create.regr.task(id, data)

create.classif.task(id, data, cutoff = 0)
```

Arguments

id	[character(1)] ID to use for Task.
data	[named list] with columns entries $X, $Y, and $orig.features.
cutoff	[numeric(1)] cutoff at which to binarize target.

Value

Task

See Also

Other Artificial Datasets: clonetask(), create.hypersphere.data(), create.linear.data(), create.linear.toy.data(), task.add.permuted.cols(), task.add.random.cols()

Other Artificial Datasets: clonetask(), create.hypersphere.data(), create.linear.data(), create.linear.toy.data(), task.add.permuted.cols(), task.add.random.cols()

fitnesses *Extract Fitnesses from ECR Log*

Description

Extract fitnesses for each generation from ECR log.

Usage

```
fitnesses(results, trafo = identity)
```

Arguments

results	[ecr_multi_objective_result] ecr run log.
trafo	[function] function matrix|data.frame -> matrix|data.frame to transforms individual generation matrices.

Value

data.frame of fitnesses from ecr run log, with extra column iter.

See Also

Other Utility Functions: paretoEdges()

getPopulations *Get Populations*

Description

Get populations from ecr_logger. Replaces ecr::getPopulation because original is buggy.

Usage

```
getPopulations(log)
```

Arguments

log	[ecr_logger] ecr log object

Value

list of populations.

getStatistics *Get Statistics*

Description

Get statistics from ecr_logger. Replaces `ecr::getStatistics` because original is buggy.

Usage

```
getStatistics(log)
```

Arguments

log	[ecr_logger] ecr log object

Value

`data.frame` of logged statistics.

initSelector *Initialize Selector*

Description

Sample the `vector.name` variable such that the number of ones has a given distribution.

Usage

```
initSelector(
  individuals,
  vector.name = "selector.selection",
 distribution = function() floor(runif(1, 0, length(individuals[[1]][[vector.name]]) +
    1)),
  soften.op = NULL,
  soften.op.strategy = NULL,
  soften.op.repeat = 1,
  reject.condition = function(x) !any(x)
)
```

Arguments

individuals	[list of named lists] the individuals to initialize.
vector.name	[character(1)] the variable name, whose entries are sampled.
distribution	[function] function that returns a random integer from 0 to the length of each individual's `vector.name` slot. Defaults to the uniform distribution from 1 to `length()`.

soften.op [ecr_mutator] an optional mutator to apply to the vector.name variable.

soften.op.strategy
function an optional function that can set the soften.op's parameters. See combine.operators strategy parameters. Ignored if soften.op is not given.

soften.op.repeat
[integer(1)] how often to repeat soften.op application. Ignored if soften.op is not given.

reject.condition
[function | NULL] reject condition as a function applied to newly generated values of vector.name. If set to NULL, no rejection is done.

Value

list of named lists the individuals with initialized [[vector.name]].

Examples

```
library(mlrCPO)

# Initialize parameter set and sample candidates
ps <- pSS(
 maxdepth: integer[1, 30],
 minsplit: integer[2, 30],
 cp: numeric[0.001, 0.999],
 selector.selection: logical^5)

initials <- sampleValues(ps, 15, discrete.names = TRUE)

# Resample logical vector selector.selection of initials
# with binomial distribution
initSelector(initials, distribution = function() rbinom(n = 5, size = 5,
 prob = 0.5))
```

---

intifyMutator *Turn Continuous-Space Operators into Integer-Space Operators*

Description

The input operator is wrapped: individuals are fed to it as-is, and output is rounded. Upper and lower bounds are both shifted by 0.5 down or up, respectively, to retain a fair distribution.

Usage

```
intifyMutator(operator)

intifyRecombinator(operator)
```

Arguments

operator [ecr_operator] ecr_operator that supports continuous variables.

Value

ecr_operator operator that operates on integers.

Examples

```
library(mlrCPO)

# Create parameter set
ps <- pSS(
numb: numeric[1, 10],
int: integer[0, 5])

# Define mutator
# If Gaussian mutator is applied to integer parameter,
# it does not return an integer
combo.mut <- combine.operators(ps,
numeric = mutGauss,
int = mutGauss)
combo.mut(list(numb = 1.5, int = 3))

# Turn continuous-space operator mutGauss into integer-space operator
mutGaussInt <- intifyMutator(mutGauss)
combo.mut.int <- combine.operators(ps,
numeric = mutGauss,
int = mutGaussInt)
combo.mut.int(list(numb = 1.5, int = 3))

# Turn continuous-space operator recSBX into integer-space operator
recSBXInt <- intifyRecombinator(recSBX)
combo.rec.int <- combine.operators(ps,
numeric = recSBX,
int = recSBXInt)
combo.rec.int(list(list(numb = 1.5, int = 3), list(numb = 3, int = 0)))
```

---

listToDf *List to data.frame*

Description

Converts a list to a data.frame based on given parameter set.

List elements must have the correct type with respect to parameter set. Exceptions are discrete parameters, whose values should be factors, only characters are accepted and factors are returned.

Returned data.frame has column names equal to parameter ids. In case of vector parameters column names will be numbered.

Usage

```
listToDf(list.object, par.set)
```

Arguments

list.object	[list] list of individuals, each with elements named by parameter ids.
par.set	[ParamSet] parameter set.

Value

[data.frame]

Examples

```
library(mlrCPO)

# Create parameter set
temp <- c("a", "b", "c")
ps.simple <- pSS(
 num: numeric [0, 10],
 int: integer[0, 10] [[trafo = function(x) x / 10]],
 char: discrete [temp],
 selector.selection: logical^10)

# Sample values as list and convert list to data frame
init.list <- sampleValues(ps.simple, 5, discrete.names = TRUE)
result <- listToDf(init.list, ps.simple)
result
```

---

makeBaselineObjective *Create mlrMBO Objective Function*

Description

"Baseline" performance measure: Creates an objective function that performs normal parameter optimization by evaluating filters with additional parameters: `mosmafs.nselect` (how many features to select), `mosmafs.iselect` (vector integer parameter that selects explicit features that are not necessary the best according to filter values) and `mosmafs.select.weights` (numeric parameter vector that does weighting between filter values to use.

Usage

```
makeBaselineObjective(
  learner,
  task,
  filters,
  ps,
  resampling,
```

```
  measure = NULL,
  num.explicit.featsel = 0,
  holdout.data = NULL,
  worst.measure = NULL,
  cpo = NULLCPO,
  numfeats = getTaskNFeats(task)
)
```

Arguments

learner	[Learner] the base learner to use.
task	[Task] the task to optimize.
filters	[character] filter values to evaluate and use.
ps	[ParamSet] the ParamSet of the learner to evaluate. Should not include selector.selection etc., only parameters of the actual learner.
resampling	[ResampleDesc | ResampleInstance] the resampling strategy to use.
measure	[Measure] the measure to evaluate. If measure needs to be maximized, the measure is multiplied by -1, to make it a minimization task.
num.explicit.featsel	[integer(1)] additional number of parameters to add for explicit feature selection.
holdout.data	[Task | NULL] the holdout data to consider.
worst.measure	[numeric(1)] worst value to impute for failed evals.
cpo	[CPO] CPO pipeline to apply before feature selection.
numfeats	[integer(1)] number of features to consider. Is extracted from the task but should be given if cpo changes the number of features.

Value

function that can be used for mlrMBO; irace possibly needs some adjustments.

---

makeFilterMat *Create a Filter-Matrix*

Description

A Filter-Matrix can be used in combination with mutUniformMetaReset for heuristic-supported biased mutation.

Usage

```
makeFilterMat(
  task,
  filters,
  expectfeatfrac = 0.5,
  expectfeats = getTaskNFeats(task) * expectfeatfrac,
  minprob = 0,
  maxprob = 1
)
```

Arguments

task	[Task] The task to generate filter information for.
filters	[character] The filters to use. Special vilter "DUMMY" gives a constant column of expectfeatfrac.
expectfeatfrac	[numeric(1)] The expected fraction of features to have in equilibrium. Ignored if expectfeats is given.
expectfeats	[numeric(1)] The expected number of features to have in equilibrium.
minprob	[numeric(1)] The minimum probability for each feature.
maxprob	[numeric(1)] The maximum probability for each feature.

Value

matrix

Examples

```
library("mlr")

# Example for iris task
filters <- c("praznik_JMI", "anova.test", "variance", "DUMMY")
fima <- makeFilterMat(iris.task, filters = filters)
print(fima)
```

---

makeFilterStrategy *Create a Filter Strategy Function*

Description

Creates a strategy function that uses the weight.param.name entry of individuals as a weighting vector reset.dist.weights and reset.dists for mutUniformMetaReset and mutUniformMetaResetSHW.

Usage

```
makeFilterStrategy(reset.dists, weight.param.name)
```

Arguments

reset.dists	[matrix] see reset.dists in mutUniformMetaReset.
weight.param.name	[character(1)] name of parameter to use as reset.dist.weights in mutUniformMetaReset.

Value

function

makeObjective *Create ecr Objective Function*

Description

Creates an objective function that resamples `learner` on `task` with `resampling` and measures `measure` (optional), together with the number of features selected. If measure needs to be maximized, it is multiplied by -1 to make it a minimization task.

The `ParamSet` used to generate individuals for the ecr must include parameters for `learner`, not a `logical` parameter with length equal to `getTaskNFeats(task)` for feature selection, as it is automatically added named as `selector.selection`. It can be accessed via `getParamSet()` with the object created by `makeObjective()` as input.

`learner` must *not* include a `cpoSelector()` applied to it, this happens automatically within `makeObjective`.

Usage

```
makeObjective(
  learner,
  task,
  ps,
  resampling,
  measure = NULL,
  holdout.data = NULL,
  worst.measure = NULL,
  cpo = NULLCPO
)
```

Arguments

`learner`	[Learner] A `Learner` object to optimize.
`task`	[Task] The `mlr::Task` object to optimize on.
`ps`	[ParamSet] The `ParamSet` to optimize over, only parameters of the actual learner.
`resampling`	[ResampleDesc | ResampleInst | function] The `ResampleDesc` or `ResampleInst` object to use. This may be a function `numeric(1)` -> `ResampleDesc`/`ResampleInst` which maps fidelity to the resampling to use. If this is used, then the resampling should be chosen such that an average value, weighted by fidelity, makes sense. For example, the function could map an integer to a corresponding number of resampling folds or repetitions.
`measure`	[Measure | NULL] The `Measure` to optimize for. The default is `NULL`, which uses the `task`'s default `Measure`. If measure needs to be maximized, the measure is multiplied by -1, to make it a minimization task.
`holdout.data`	[Task] Additional data on which to predict each configuration after training on `task`.
`worst.measure`	[numeric(1)] worst value for measure to consider, for dominated hypervolume calculation. Will be extracted from the given measure if not given, but will raise

an error if the extracted (or given) value is infinite. Measure is multiplied by -1, if measure needs to be maximized.

cpo [CPO] CPO pipeline to apply before feature selection. (A CPO that should be applied *after* feature selection should already be part of `learner` when given). Care should be taken that the `selector.selection` parameter in `ps` has the appropriate length of the data that `cpo` emits.

Value

`function` an objective function for `ecr::ecr`.

Examples

```
library("mlr")
library("rpart")

task.whole <- bh.task
rows.whole <- sample(nrow(getTaskData(task.whole)))
task <- subsetTask(task.whole, rows.whole[1:250])
task.hout <- subsetTask(task.whole, rows.whole[251])
lrn <- makeLearner("regr.rpart")

ps.simple <- mlrCPO::pSS(
  maxdepth: integer[1, 30],
  minsplit: integer[2, 30],
  cp: numeric[0.001, 0.999])
  nRes <- function(n) {
  makeResampleDesc("Subsample", split = 0.9, iters = n)
}

fitness.fun.mos <- makeObjective(lrn, task, ps.simple, nRes,
  measure = mse,
  holdout.data = task.hout, worst.measure = 100)

# extract param set from objective
ps.obj  <- getParamSet(fitness.fun.mos)
getParamIds(ps.obj) # automatically added parameter ' for selecting features

exp <- sampleValue(ps.obj)
res <- fitness.fun.mos(exp, fidelity = 2, holdout = FALSE)
```

---

mosmafsTermEvals *Termination Function Creator*

Description

These create functions that can be given to `slickEcr`'s `generations` argument

The stagnation terminators only count stagnation from the last time the fidelity was changed in a way that led to population re-evaluation.

Usage

```
mosmafsTermEvals(evals)

mosmafsTermGenerations(generations)

mosmafsTermTime(time)

mosmafsTermFidelity(fidelity)

mosmafsTermStagnationHV(stag, stag.index = "generations")

mosmafsTermStagnationObjStatistic(
  stag,
  stag.index = "generations",
  obj.stat = "mean",
  objective.index = TRUE
)
```

Arguments

evals	[integer(1)] limit evals.
generations	[integer(1)] limit generations. Initial population does not count.
time	[numeric(1)] limit evaluation time (which does not count holdout fitting time.
fidelity	[numeric(1)] total fidelity evaluation to limit.
stag	[integer(1)] number of generations (or other measures) without progress in hypervolume or mean objective value.
stag.index	[character(1)] one of "generations" (default), "evals", "time", "fidelity": What index to count stag against when aborting after stagnation.
obj.stat	[character(1)] what statistic of the objective to test. One of "min", "mean", "max". Default "mean".
objective.index	[integer | logical] index of objective(s) to consider. Terminates if all the objectives listed here stagnate. TRUE for all objectives. Default TRUE.

Value

function a terminator function

---

mutBitflipCHW *Bitflip (Approximately, in Expectation) Conserving Hamming Weight*

Description

If a given bitvector has m 1s and n 0s, then a bit is flipped from 0 to 1 with probability 2p(m+1)/(m+n+2) and from 1 to 0 with probability 2p(n+1)/(m+n+2). This is equivalent with choosing bits uniformly at random with probability 2 * p and drawing them from a bernoulli-distribution with parameter (m+1)/(m+n+2).

Usage

```
mutBitflipCHW(ind, p = 0.1, ...)
```

Arguments

ind	[integer] binary individual.
p	[numeric] average flip probability, must be between 0 and 0.5.
...	further arguments passed on to the method.

Value

[integer] mutated binary individual.

---

mutDoubleGeom *Double Geometric Distribution Mutator*

Description

"Double Geometric" mutation operator for integer parameters: with probability p a random geometrically distributed value is added, and another (different) one subtracted.

mutDoubleGeomScaled scales sdev with each component's range and then uses geomp = (sqrt(2 * sdev^2 + 1) -1) / sdev^2.

Usage

```
mutDoubleGeom(ind, p = 1, geomp = 0.9, lower, upper)

mutDoubleGeomScaled(ind, p = 1, sdev = 0.05, lower, upper)
```

Arguments

ind	[integer] individual to mutate.
p	[numeric(1)] per-entry probability to perform mutation.
geomp	[numeric] geometric distribution parameter.
lower	[integer] lower bounds of ind values. May have same length as ind or may be a single number, if the lower bounds are the same for all values.
upper	[integer] upper bounds of ind values. May have same length as ind or may be a single number, if the upper bounds are the same for all values.
sdev	[numeric] standard deviation, relative to upper -lower.

Value

[integer]

See Also

Other operators: mutGaussIntScaled(), mutGaussInt(), mutGaussScaled(), mutPolynomialInt(), mutRandomChoice(), mutUniformInt(), recGaussian(), recIntIntermediate(), recIntSBX()

mutGaussInt *Integer Gaussian Mutator*

Description

See ecr::mutGauss

Usage

```
mutGaussInt(ind, ..., lower, upper)
```

Arguments

ind	[integer] integer vector/individual to mutate.
...	further arguments passed on to the method.
lower	[integer] vector of minimal values for each parameter of the decision space. Must have the same length as ind.
upper	[integer] vector of maximal values for each parameter of the decision space. Must have the same length as ind.

Value

[integer] mutated individual.

See Also

Other operators: mutDoubleGeom(), mutGaussIntScaled(), mutGaussScaled(), mutPolynomialInt(), mutRandomChoice(), mutUniformInt(), recGaussian(), recIntIntermediate(), recIntSBX()

mutGaussIntScaled *Integer Scaled Gaussian Mutator*

Description

See mutGaussScaled.

Usage

```
mutGaussIntScaled(ind, ..., lower, upper)
```

Arguments

ind	[integer] integer vector/individual to mutate.
...	further arguments passed on to the method.
lower	[integer] vector of minimal values for each parameter of the decision space. Must have the same length as ind.
upper	[integer] vector of maximal values for each parameter of the decision space. Must have the same length as ind.

Value

[integer] mutated individual.

See Also

Other operators: mutDoubleGeom(), mutGaussInt(), mutGaussScaled(), mutPolynomialInt(), mutRandomChoice(), mutUniformInt(), recGaussian(), recIntIntermediate(), recIntSBX()

---

mutGaussScaled *Scaled Gaussian Mutator*

Description

See ecr::mutGauss. Allows a vector of standard deviations. Scales standard deviations to the range of [lower, upper].

Usage

```
mutGaussScaled(ind, p = 1, sdev = 0.05, lower, upper)
```

Arguments

ind	[numeric] Numeric vector / individual to mutate.
p	[numeric(1)] Probability of mutation for the gauss mutation operator.
sdev	[numeric] standard deviation(s) of the Gauss mutation.
lower	[numeric] Vector of minimal values for each parameter of the decision space.
upper	[numeric] Vector of maximal values for each parameter of the decision space.

Value

[numeric] mutated individual.

See Also

Other operators: mutDoubleGeom(), mutGaussIntScaled(), mutGaussInt(), mutPolynomialInt(), mutRandomChoice(), mutUniformInt(), recGaussian(), recIntIntermediate(), recIntSBX()

mutPolynomialInt *Integer Polynomial Mutator*

Description

See ecr::mutPolynomial

Usage

```
mutPolynomialInt(ind, ..., lower, upper)
```

Arguments

ind	[integer] integer vector/individual to mutate.
...	further arguments passed on to the method.
lower	[integer] vector of minimal values for each parameter of the decision space. Must have the same length as ind.
upper	[integer] vector of maximal values for each parameter of the decision space. Must have the same length as ind.

Value

[integer] mutated individual.

See Also

Other operators: mutDoubleGeom(), mutGaussIntScaled(), mutGaussInt(), mutGaussScaled(), mutRandomChoice(), mutUniformInt(), recGaussian(), recIntIntermediate(), recIntSBX()

mutRandomChoice *Random Choice Mutator*

Description

"Random Choice" mutation operator for discrete parameters: with probability p chooses one of the available categories at random (this *may* be the original value!)

Usage

```
mutRandomChoice(ind, values, p = 0.1)
```

Arguments

ind	[character] individual to mutate.
values	[list of character] set of possible values for ind entries to take. May be a list of length 1, in which case it is recycled.
p	[numeric(1)] per-entry probability to perform mutation.

Value

[character]

See Also

Other operators: mutDoubleGeom(), mutGaussIntScaled(), mutGaussInt(), mutGaussScaled(), mutPolynomialInt(), mutUniformInt(), recGaussian(), recIntIntermediate(), recIntSBX()

---

mutUniformInt *Integer Uniform Mutator*

Description

See ecr::mutUniform

Usage

```
mutUniformInt(ind, ..., lower, upper)
```

Arguments

ind	[integer] integer vector/individual to mutate.
...	further arguments passed on to the method.
lower	[integer] vector of minimal values for each parameter of the decision space. Must have the same length as ind.
upper	[integer] vector of maximal values for each parameter of the decision space. Must have the same length as ind.

Value

[integer] mutated individual.

See Also

Other operators: mutDoubleGeom(), mutGaussIntScaled(), mutGaussInt(), mutGaussScaled(), mutPolynomialInt(), mutRandomChoice(), recGaussian(), recIntIntermediate(), recIntSBX()

mutUniformMetaReset *Parametrised Uniform Reset for Binary Parameters*

Description

Performs mutUniformReset with reset.dist = reset.dists %*% reset.dist.weights.

Usage

```
mutUniformMetaReset(ind, p = 0.1, reset.dists, reset.dist.weights)
```

Arguments

ind	[integer] binary individual with values 0 or 1.
p	[numeric(1)] entry-wise reset probability.
reset.dists	[matrix] columns of probabilities to draw 1-bit per entry, if reset is performed. Must have length(ind) rows and length(reset.dist.weights) columns.
reset.dist.weights	[numeric] weight vector to select among reset.dists columns.

Value

[integer] the mutated individual

mutUniformParametric *Parametric Uniform Mutation*

Description

Adds a variable delta to each component ind[i] with probability p, where delta is uniformly distributed between pmax(lower -ind[x],-lx/2) and pmin(upper -ind[i],lx/2).

Usage

```
mutUniformParametric(ind, p, lx, lower, upper)

mutUniformParametricScaled(ind, p, sdev, lower, upper)

mutUniformParametricInt(ind, ..., lower, upper)

mutUniformParametricIntScaled(ind, ..., lower, upper)
```

Arguments

ind	[numeric | integer] individual to mutate.
p	[numeric] per-entry probability to perform mutation.
lx	[numeric] uniform distribution bandwidth.
lower	[integer] lower bounds of ind values. May have same length as ind or may be a single number, if the lower bounds are the same for all values.
upper	[integer] upper bounds of ind values. May have same length as ind or may be a single number, if the upper bounds are the same for all values.
sdev	[numeric] standard deviation, will be scaled to upper -lower.
...	further arguments passed on to the method.

Value

[numeric | integer] mutated individual.

---

mutUniformReset *Uniform Reset for Binary Parameters*

Description

For each bit individually, decide with probability p to "reset" it to an equilibrium distribution which is specified by reset.dist: a bit being reset is set to 1 with probability reset.dist and set to 0 with probability (1 - reset.dist).

Usage

```
mutUniformReset(ind, p = 0.1, reset.dist)
```

Arguments

ind	[integer] binary individual with values 0 or 1.
p	[numeric(1)] entry-wise reset probability.
reset.dist	[numeric] probability to draw 1-bit per entry, if reset is performed. reset.dist can be length 1 or same length as ind (which uses a different distribution for each bit).

Value

[integer] the mutated individual.

mutUniformResetSHW *Uniform Reset Scaled by Hamming Weight*

Description

Combination of the idea of mutBitflipCHW with mutUniformReset.

If a given bitvector has m 1s and n 0s, then, with probability p for each bit, it is drawn anew from the distribution ((m + 1) * reset.dist) / (m * reset.dist + n * (1 -reset.dist) + 1).

The reasoning behind this is that, without Laplace smoothing, drawing from m * reset.dist / (m * reset.dist + n * (1 - reset.dist))would lead to probabilities of drawing a "0" or "1" such that-mean(P("1") / P("0")) = m / n * mean(reset.dist / (1 - reset.dist))‘.

The mutUniformMetaResetSHW does reset with a weighted mean of distributions.

Usage

```
mutUniformResetSHW(ind, p = 0.1, reset.dist, ...)

mutUniformMetaResetSHW(ind, p = 0.1, reset.dists, reset.dist.weights, ...)
```

Arguments

ind	[integer] binary individual.
p	[numeric] average reset probability, must be between 0 and 1.
reset.dist	[numeric] approximate probability to draw 1-bit per entry.
...	further arguments passed on to the method.
reset.dists	[matrix] columns of probabilities, with length(ind) cols and length(reset.dist.weights) rows.
reset.dist.weights	[numeric] weight vector to select among reset.dist columns.

Value

[integer] the mutated individual

naiveHoldoutDomHV *Naive Hypervolume on Holdout Data*

Description

Calculate dominated hypervolume on holdout data. The result is biased depending on noise in holdout data performance.

Usage

```
naiveHoldoutDomHV(fitness, holdout, refpoint)
```

Arguments

fitness	[matrix] fitness matrix of training data.
holdout	[matrix] fitness matrix of holdout data.
refpoint	[numeric] reference point.

Value

numeric

---

overallRankMO — *Rank by Nondominated Front and Crowding Distance or Hypervolume Contribution*

Description

Rank individuals by nondominating sorted front first and by hypervolume contribution or crowding distance second.

Ties are broken randomly by adding random noise of relative magnitude `.Machine$double.eps * 2^10` to points.

Usage

```
overallRankMO(fitness, sorting = "crowding", ref.point)
```

Arguments

fitness	[matrix] fitness matrix, one column per individual.
sorting	[character(1)] one of "domhv" or "crowding" (default).
ref.point	[numeric] reference point for hypervolume, must be given if sorting is "domhv".

Value

[integer] vector of ranks with length `ncol(fitness)`, lower ranks are associated with individuals that tend to dominate more points and that tend to have larger crowding distance or hypervolume contribution.

paretoEdges *Get Pareto Front Edges from Fitness Matrix*

Description

Get the edges defining a 2D pareto front for plotting.

Usage

```
paretoEdges(fitness, refpoint)
```

Arguments

fitness	[matrix | data.frame] matrix or (numeric) data.frame with two columns and rows for each individuum.
refpoint	[numeric(2)] reference point.

Value

data.frame with three columns: The points on the pareto front, and a logical column point indicating whether the point is on the pareto front (TRUE) or an auxiliary point for plotting (FALSE).

See Also

Other Utility Functions: fitnesses()

popAggregate *Aggregate Population Results*

Description

Extract attributes saved for individuums in a log object to a more accessible matrix or data.frame.

Usage

```
popAggregate(log, extract, simplify = TRUE, data.frame = FALSE)
```

Arguments

log	[ecr_logger] ecr log object.
extract	[character] names of attributes to extract, currently "names", "fitness", "runtime", "fitness.holdout" and "fidelity" (if used) are supported.
simplify	[logical(1)] whether to create a matrix/data.frame for each generation (default). Otherwise a list is returned for each generation containing the value (if length(extract) == 1) or a named list of values.

data.frame	[logical(1)] whether to return a data.frame with rows for each individuum (if TRUE) or to return a matrix with columns for each individuum compatible as fitness matrix with various ecr tools (if FALSE, default). Only effective if simplify is TRUE.

Value

[matrix] if simplify is TRUE, [list] otherwise.

---

recGaussian *Gaussian Intermediate Recombinator*

Description

Gaussian intermediate recombinator samples component-wise from a normal distribution with mean as the component-wise mean and standard deviation as halved components-wise absolute distance of the two given parents. It is applicable only for numeric representations.

See also ecr::recIntermediate.

Usage

```
recGaussian(inds, lower, upper)
```

Arguments

inds	[list of numeric] list of two individuals to recombinate.
lower	[numeric] lower bounds of inds values. May have same length as one individual or may be a single number, if the lower bounds are the same for all values.
upper	[numeric] upper bounds of inds values. May have same length as one individual or may be a single number, if the upper bounds are the same for all values.

Value

[list of numeric] recombined individuals.

See Also

Other operators: mutDoubleGeom(), mutGaussIntScaled(), mutGaussInt(), mutGaussScaled(), mutPolynomialInt(), mutRandomChoice(), mutUniformInt(), recIntIntermediate(), recIntSBX()

recIntIntermediate *Integer Intermediate Recombinator*

Description

See ecr::recIntermediate

Usage

```
recIntIntermediate(inds, ..., lower, upper)
```

Arguments

inds	[inds] parents, i.e., list of exactly two integer vectors of equal length.
...	further arguments passed on to the method.
lower	[integer] vector of minimal values for each parameter of the decision space.
upper	[integer] vector of maximal values for each parameter of the decision space.

Value

[integer] mutated individual.

See Also

Other operators: mutDoubleGeom(), mutGaussIntScaled(), mutGaussInt(), mutGaussScaled(), mutPolynomialInt(), mutRandomChoice(), mutUniformInt(), recGaussian(), recIntSBX()

recIntSBX *Integer SBX Recombinator*

Description

See ecr::recSBX

Usage

```
recIntSBX(inds, ..., lower, upper)
```

Arguments

inds	[integer] parents, i.e., list of exactly two numeric vectors of equal length.
...	further arguments passed on to the method.
lower	[integer] vector of minimal values for each parameter of the decision space.
upper	[integer] vector of maximal values for each parameter of the decision space.

Value

[integer] mutated individual.

See Also

Other operators: mutDoubleGeom(), mutGaussIntScaled(), mutGaussInt(), mutGaussScaled(), mutPolynomialInt(), mutRandomChoice(), mutUniformInt(), recGaussian(), recIntIntermediate()

---

recPCrossover *General Uniform Crossover*

Description

Crossover recombination operator that crosses over each position iid with prob. p and can also be used for non-binary operators.

Usage

```
recPCrossover(inds, p = 0.1, ...)
```

Arguments

inds	[list of any] list of two individuals to perform uniform crossover on
p	[numeric(1)] per-entry probability to perform crossover.
...	further arguments passed on to the method.

Value

[list of any] The mutated individuals.

---

selSimpleUnique *Simple Selector without Replacement*

Description

Simple Selector without Replacement

Usage

```
selSimpleUnique(fitness, n.select)
```

Arguments

fitness	[matrix] fitness matrix, one column per individual.
n.select	[integer(1)] number of individuals to select.

Value

[matrix] selected individuals.

See Also

Other Selectors: `selTournamentMO()`

`selTournamentMO` *Multi-Objective k-Tournament Selector*

Description

k individuals are chosen randomly and the best one is chosen. This process is repeated `n.select` times.

Choice is primarily by dominated sorting and secondarily by either dominated hypervolume or crowding distance, depending on `sorting`.

Ties are broken randomly by adding random noise of relative magnitude `.Machine$double.eps * 2^10` to points.

Usage

```
selTournamentMO(
  fitness,
  n.select,
  sorting = "crowding",
  ref.point,
  k = 2,
  return.unique = FALSE
)
```

Arguments

`fitness`	[matrix] fitness matrix, one column per individual.
`n.select`	[integer(1)] number of individuals to select.
`sorting`	[character(1)] one of `"domhv"` or `"crowding"` (default).
`ref.point`	[numeric] reference point for hypervolume, must be given if `sorting` is `"domhv"`.
`k`	[integer(1)] number of individuals to select at once.
`return.unique`	[logical(1)] whether returned individual indices must be unique.

Value

[integer] vector of selected individuals.

See Also

Other Selectors: `selSimpleUnique()`

setMosmafsVectorized *Set mosmafs.vectorize*

Description

Set or change attribute `mosmafs.vectorized` in fitness function.

Usage

```
setMosmafsVectorized(fn, vectorize = TRUE)
```

Arguments

`fn`	`smoof_multi_objective_function` fitness function.
`vectorize`	[logical(1)] whether to force `slickEvaluateFitness` to pass candidates to fitness function as `data.frame` or not.

Value

`smoof_multi_objective_function`.

slickEcr *Modified Interface to ECR*

Description

Mostly `ecr::ecr`, with some simplifications and extensions.

`slickEcr` does mostly what `ecr::ecr` does, with different default values at places. Note that `fitness.fun` must be a "`smoof`" function.

`initEcr` only evaluates fitness for the initial population and does not perform any mutation or selection.

`continueEcr` continues a run for another number of `generations`. Only `ecr.object` (a result from a previous `initEcr`, `slickEcr`, or `continueEcr` call) and `generations` must be given, the other arguments are optional. *If* they were set in a previous `slickEcr` or `continueEcr` call, the values from the previous run are used. Otherwise it is possible to supply any combination of these values to set them to new values.

Note, for `fidelity`, that the generation continues counting from previous runs, so if `initEcr` was ran for 5 generations and `continueEcr` is called with a `fidelity` with first column values `c(1,8)`, then the fidelity given in the first row is applied for 2 generations, after which the fidelity given in the second row applies.

Usage

```
slickEcr(
  fitness.fun,
  lambda,
  population,
  mutator,
  recombinator,
  generations = 100,
  parent.selector = selSimple,
  survival.selector = selNondom,
  p.recomb = 0.7,
  p.mut = 0.3,
  survival.strategy = "plus",
  n.elite = 0,
  fidelity = NULL,
  unbiased.fidelity = TRUE,
  log.stats = NULL,
 log.stats.newinds = c(list(runtime = list("mean", "sum")), if (!is.null(fidelity))
    list(fidelity = list("sum")))
)

initEcr(
  fitness.fun,
  population,
  fidelity = NULL,
  log.stats = NULL,
 log.stats.newinds = c(list(runtime = list("mean", "sum")), if (!is.null(fidelity))
    list(fidelity = list("sum"))),
  unbiased.fidelity = TRUE
)

continueEcr(
  ecr.object,
  generations,
  lambda = NULL,
  mutator = NULL,
  recombinator = NULL,
  parent.selector = NULL,
  survival.selector = NULL,
  p.recomb = NULL,
  p.mut = NULL,
  survival.strategy = NULL,
  n.elite = NULL,
  fidelity = NULL,
  unbiased.fidelity = NULL
)
```

Arguments

`fitness.fun`	[smoof_multi_objective_function] fitness function, must be a `"smoof"` function.
`lambda`	[integer(1)] number of individuals to add in each generation.
`population`	[list] list of individuals to start off from.
`mutator`	[ecr_mutator] mutation operator.
`recombinator`	[ecr_recombinator] recombination operator.
`generations`	[integer(1) | list of function] number of iterations to evaluate if it is an integer, or terminator function. If this is an integer, it counts the *new* generations to evaluate; otherwise the terminator functions are applied to the whole combined trace of evaluation.
`parent.selector`	[ecr_selector] parent selection operator.
`survival.selector`	[ecr_selector] survival selection operator.
`p.recomb`	[numeric(1)] probability to apply a recombination operator.
`p.mut`	[numeric(1)] probability to apply mutation operator.
`survival.strategy`	[character(1)|function] one of `"plus"` or `"comma"` or a function. If function, arguments must be the same as for `ecr::replaceMuPlusLambda`.
`n.elite`	[integer(1)] Number of elites to keep, only used if `survival.strategy` is `"comma"`
`fidelity`	[data.frame | NULL] If this is given, it controls the fidelity of the function being evaluated, via its `fidelity` argument. It must then be a `data.frame` with two or three columns. The first column gives the generation at which the fidelity first applies; the second column controls the fidelity at that generation or later; the third column, if given, controls the additional fidelity whenever the result of the first evaluation is not dominated by any result of the previous generation. The entries in the first column must be strictly ascending. The first element of the first column must always be 1. Whenever fidelity changes, the whole population is re-evaluated, so it is recommended to use only few different fidelity jumps throughout all generations.
`unbiased.fidelity`	[logical(1)] Whether generations do not have to be re-evaluated when fidelity jumps downward.
`log.stats`	[list] information to log for each generation. Defaults to min, mean, and max of each objective as well as dominated hypervolume.
`log.stats.newinds`	[list] information to log for each newly evaluated individuals
`ecr.object`	[MosmafsResult] an object retrieved from previous runs of `initEcr`, `slickEcr`, or `continueEcr`

Value

[MosmafsResult] the terminated optimization state.

Examples

```
library("mlr")
library("magrittr")
library("mlrCPO")

# Define tasks
task.whole <- create.hypersphere.data(3, 2000) %>%
create.classif.task(id = "sphere") %>%
task.add.permuted.cols(10)
rows.whole <- sample(2000)
task <- subsetTask(task.whole, rows.whole[1:500])
task.hout <- subsetTask(task.whole, rows.whole[501:2000])

# Create learner
lrn <- makeLearner("classif.rpart", maxsurrogate = 0)

# Create parameter set to optimize over
ps <- pSS(
 maxdepth: integer[1, 30],
 minsplit: integer[2, 30],
 cp: numeric[0.001, 0.999])

# Create fitness function
fitness.fun <- makeObjective(lrn, task, ps, cv5,
 holdout.data = task.hout)

#  Receive parameter set from fitness function
ps.objective <- getParamSet(fitness.fun)

# Define mutators and recombinators
mutator <- combine.operators(ps.objective,
 numeric = ecr::setup(mutGauss, sdev = 0.1),
 integer = ecr::setup(mutGaussInt, sdev = 3),
 selector.selection = mutBitflipCHW)
crossover <- combine.operators(ps.objective,
 numeric = recPCrossover,
 integer = recPCrossover,
 selector.selection = recPCrossover)

# Initialize population and evaluate it
initials <- sampleValues(ps.objective, 32, discrete.names = TRUE)
run.init <- initEcr(fitness.fun = fitness.fun, population = initials)

# Run NSGA-II for 5 generations with run.init as input
run.gen <- continueEcr(run.init, generations = 5, lambda = 5, mutator = mutator,
 recombinator = crossover, parent.selector = selSimple,
 survival.selector = selNondom,
 p.recomb = 0.7, p.mut = 0.3, survival.strategy = "plus")

# Or instead of initEcr and continueEcr use the shortcut function slickEcr
run.simple <- slickEcr(
 fitness.fun = fitness.fun, lambda = 5, population = initials,
 mutator = mutator,
 recombinator = crossover,
```

```
 generations = 5)

print(run.simple)
```

`slickEvaluateFitness` *Compute the Fitness of Individuals*

Description

Takes a list of individuals `population` and evaluates the fitness with varying `fidelity`, if specified.

A list is returned with two elements, one being the list of individuals and one being the matrix of fitness values. In the matrix each column represents the fitness values of one individual. For consistency, a matrix is also returned for single objective fitness function.

Usage

```
slickEvaluateFitness(ctrl, population, fidelity = NULL, previous.points = NULL)
```

Arguments

`ctrl`	[ecr_control] control object.
`population`	[list] list of individuals to evaluate.
`fidelity`	[numeric] vector of fidelity, with one or two elements. If this has one element, it is directly passed on to the fitness function. If it has two elements, the fitness function is first evaluated with the first fidelity; if the resulting point dominates the population given in `population` it is again evaluated with the second fidelity given, and the result is averaged weighted by the fidelity parameter.
`previous.points`	[matrix] population to compare points to if `fidelity` has two elements. Otherwise not used.

Value

`list(population = list, fitness = matrix)`

task.add.permuted.cols *Add Permuted Noise-Features to Task*

Description

Adds num copies of the task with permuted rows.

The feature names of the ith permuted copy have PERM.i. prepended to them. The returned task has a new member $orig.features which is a logical vector indicating the features that were originally in the task.

If the $orig.features slot is already present in the input task, then the output will have added FALSE entries at appropriate positions.

Usage

```
task.add.permuted.cols(task, num)
```

Arguments

task	[Task] the input task.
num	[integer(1)] Number of noise features to add.

Value

Task

See Also

Other Artificial Datasets: clonetask(), create.hypersphere.data(), create.linear.data(), create.linear.toy.data(), create.regr.task(), task.add.random.cols()

task.add.random.cols *Add Sampled Noise-Features to Task*

Description

Adds num new features sampled from dist to task. New features are inserted at random positions in the task and named RANDOM.1...RANDOM.[num]

The returned Task has a $orig.features which is a logical vector indicating the features that were originally in the Task.

If the $orig.features slot is already present in the input task, then the output will have added FALSE entries at appropriate positions.

Usage

```
task.add.random.cols(task, num, dist = rnorm)
```

Arguments

task	[Task] the input task.
num	[integer(1)] number of noise features to add.
dist	[function] function n -> numeric(n) that samples random noise features.

Value

Task

See Also

Other Artificial Datasets: clonetask(), create.hypersphere.data(), create.linear.data(), create.linear.toy.data(), create.regr.task(), task.add.permuted.cols()

---

unbiasedHoldoutDomHV *Unbiased Dominated Hypervolume on Holdout Data*

Description

Calculate dominated hypervolume on holdout data. The result is unbiased with respect to (uncorrelated with respect to objectives) noise in holdout data performance, but it is *not* an estimate of real "dominated hypervolume".

Only works on two-objective performance matrices.

Usage

```
unbiasedHoldoutDomHV(fitness, holdout, refpoint)
```

Arguments

fitness	[matrix] fitness matrix of training data.
holdout	[matrix] fitness matrix of holdout data.
refpoint	[numeric] reference point.

Value

numeric

valuesFromNames *Convert Discrete Parameters from Names to Values*

Description

Convert parameter values sampled with ParamHelpers::sampleValue() and discrete.names = TRUE to true parameter values.

Usage

```
valuesFromNames(paramset, value)
```

Arguments

paramset	[ParamSet] The ParamSet used to generate the value.
value	[named list] Names list of parameters sampled from paramset.

Value

named list of parameter values, with character entries representing names of values of discrete params converted to the actual values.

Index