Package 'TunePareto'

October 2, 2023

Type Package

Title Multi-Objective Parameter Tuning for Classifiers

Version 2.5.3

Author Christoph Müssel, Ludwig Lausser, Hans Kestler

Maintainer Hans Kestler <hans.kestler@uni-ulm.de>

Description Generic methods for parameter tuning of classification algorithms using multiple scoring functions (Muessel et al. (2012), <doi:10.18637/jss.v046.i05>).

Suggests snowfall, igraph, gsl, class, tree, e1071, randomForest, klaR

License GPL-2

LazyLoad yes

Encoding UTF-8

NeedsCompilation yes

Repository CRAN

Date/Publication 2023-10-02 14:40:05 UTC

R topics documented:

`TunePareto-package` *Multi-objective parameter tuning for classifiers*

Description

Generic methods for parameter tuning of classification algorithms using multiple scoring functions

Details

The methods of this package allow to assess the performance of classifiers with respect to certain parameter values and multiple scoring functions, such as the cross-validation error or the sensitivity. It provides the `tunePareto` function which can be configured to run most common classification methods implemented in R. Several sampling strategies for parameters are supplied, including Latin Hypercube sampling, quasi-random sequences, and evolutionary algorithms.

Classifiers are wrapped in generic `TuneParetoClassifier` objects which can be created using `tuneParetoClassifier`. For state-of-the-art classifiers, the package includes the corresponding wrapper objects (see `tunePareto.knn`, `tunePareto.tree`, `tunePareto.randomForest`, `tunePareto.svm`, `tunePareto.NaiveBayes`).

The method tests combinations of the supplied classifier parameters according to the supplied scoring functions and calculates the Pareto front of optimal parameter configurations. The Pareto fronts can be visualized using `plotDominationGraph`, `plotParetoFronts2D` and `plotObjectivePairs`.

A number of predefined scoring functions are provided (see `predefinedObjectiveFunctions`), but the user is free to implement own scores (see `createObjective`).

Author(s)

Christoph Müssel, Ludwig Lausser, Hans Kestler

Maintainer: Hans Kestler <hans.kestler@uni-ulm.de>

References

Christoph Müssel, Ludwig Lausser, Markus Maucher, Hans A. Kestler (2012). Multi-Objective Parameter Selection for Classifiers. Journal of Statistical Software, 46(5), 1-27. DOI https://doi.org/10.18637/jss.v046.i05.

Examples

```
# optimize the 'cost' and 'kernel' parameters of an SVM according
# to CV error and CV Specificity on the 'iris' data set
# using several predefined values for the cost
r <- tunePareto(data = iris[, -ncol(iris)],
                labels = iris[, ncol(iris)],
                classifier=tunePareto.svm(),
                cost=c(0.001,0.01,0.1,1,10),
                kernel=c("linear", "polynomial",
                         "radial", "sigmoid"),
                objectiveFunctions=list(cvError(10, 10),
                                        cvSpecificity(10, 10, caseClass="setosa")))

# print Pareto-optimal solutions
print(r)

# use a continuous interval for the 'cost' parameter
# and optimize it using evolutionary algorithms and
# parallel execution with snowfall
library(snowfall)
sfInit(parallel=TRUE, cpus=2, type="SOCK")
r <- tunePareto(data = iris[, -ncol(iris)],
                 labels = iris[, ncol(iris)],
                 classifier = tunePareto.svm(),
                 cost = as.interval(0.001,10),
                 kernel = c("linear", "polynomial",
                            "radial", "sigmoid"),
                 sampleType="evolution",
                 numCombinations=20,
                 numIterations=20,
                 objectiveFunctions = list(cvError(10, 10),
                                           cvSensitivity(10, 10, caseClass="setosa"),
                                           cvSpecificity(10, 10, caseClass="setosa")),
                useSnowfall=TRUE)
sfStop()

# print Pareto-optimal solutions
print(r)

# plot the Pareto fronts
plotDominationGraph(r, legend.x="topright")
```

`allCombinations` *Build a list of all possible combinations of parameter values*

Description

Builds a list of all possible combinations of parameter values from supplied ranges of parameter values. That is, each of the specified values is combined with all specified values for other parameters.

The resulting lists can be used in the `classifierParameterCombinations` and `predictorParameterCombinations` parameters of `tunePareto`.

Usage

```
allCombinations(parameterRanges)
```

Arguments

`parameterRanges`	A list of lists of parameter ranges. That is, each element of the list specifies the values of a single parameter to be tested and is named according to this parameter. It is also possible to set parameters to fixed values by specifying only one value.

Value

Returns a list of lists, where each of the inner lists represents one parameter combination and consists of named elements for the parameters.

See Also

`tunePareto`

Examples

```
library(class)
# Combine only valid combinations of 'k' and 'l'
# for the k-NN classifier:
comb <- c(allCombinations(list(k=1,l=0)),
          allCombinations(list(k=3,l=0:2)),
          allCombinations(list(k=5,l=0:4)),
          allCombinations(list(k=7,l=0:6)))
print(comb)

print(tunePareto(data = iris[, -ncol(iris)],
                 labels = iris[, ncol(iris)],
                 classifier = tunePareto.knn(),
                 parameterCombinations = comb,
                 objectiveFunctions = list(cvError(10, 10),
                                           reclassError())))
```

---

as.interval *Specify a continous interval*

Description

Specifies a continous interval by supplying a lower and upper bound. Such intervals can be supplied as parameter value ranges in `tunePareto`.

Usage

```
as.interval(lower, upper)
```

Arguments

lower	The lower bound of the interval
upper	The upper bound of the interval

Value

A list of class `Interval` specifying the lower and upper bound.

See Also

tunePareto

---

createObjective *Create a new objective function*

Description

Creates a new `TuneParetoObjective` object. An objective consists of two parts: The precalculation function, which applies the classifier to the data, and the objective itself, which is calculated from the predicted class labels.

Usage

```
createObjective(precalculationFunction,
                precalculationParams = NULL,
                objectiveFunction,
                objectiveFunctionParams = NULL,
                direction = c("minimize", "maximize"),
                name)
```

Arguments

precalculationFunction	The name of the precalculation function that applies the classifiers to the data. Two predefined precalculation functions are `reclassification` and `crossValidation`.
precalculationParams	A named list of parameters for the precalculation function.
objectiveFunction	The name of the objective function that calculates the objective from the precalculated class labels.
objectiveFunctionParams	A named list of further parameters for the objective function.
direction	Specifies whether the objective is minimized or maximized.
name	A readable name of the objective.

Details

The objective calculation is divided into a precalculation step and the objective calculation itself. The main reason for this is the possibility to aggregate precalculation across objectives. For example, if both the specificity and the sensitivity of a cross-validation (with the same parameters) are required, the cross-validation is run only once to save computational time. Afterwards, the results are passed to both objective functions.

A precalculation function has the following parameters:

data The data set to be used for the precalculation. This is usually a matrix or data frame with the samples in the rows and the features in the columns.

labels A vector of class labels for the samples in `data`.

classifier A `TuneParetoClassifier` wrapper object containing the classifier to tune. A number of state-of-the-art classifiers are included in **TunePareto** (see `predefinedClassifiers`). Custom classifiers can be employed using `tuneParetoClassifier`.

classifierParams A named list of parameter assignments for the classifier.

predictorParams If the classifier has separate training and prediction functions, a named list of parameter assignments for the predictor.

Additionally, the function can have further parameters which are supplied in `precalculationParams`. To train a classifier and obtain predictions, the precalculation function can call the generic `trainTuneParetoClassifier` and `predict.TuneParetoModel` functions.

The precalculation function usually returns the predicted labels, the true labels and the model, but the only requirement of the return value is that it can be processed by the corresponding objective function. Predefined precalculation functions are `reclassification` and `crossValidation`.

The objective function has a single obligatory parameter named `result` which supplies the result of the precalculation. Furthermore, optional parameters can be specified. Their values are taken from `objectiveFunctionParams`. The function either returns a single number specifying the objective value, or a list with a `score` component containing the objective value and a `additionalData` component that contains additional information to be stored in the `additionalData` component of the `TuneParetoResult` object (see `tunePareto`).

Value

Retuns an object of class `TuneParetoObjective` with the following components:

`precalculationFunction`
: The supplied precalculation function

`precalculationParams`
: The additional parameters to be passed to the precalculation function

`objectiveFunction`
: The objective function

`minimize`
: `TRUE` if the objective is minimized, `FALSE` if it is maximized.

`name`
: The readable name of the objective.

See Also

`predefinedObjectiveFunctions`, `trainTuneParetoClassifier`, `predict.TuneParetoModel`

Examples

```
# create new objective minimizing the number of support vectors
# for a support vector machine

reclassSupportVectors <- function (saveModel = FALSE)
{
    createObjective(precalculationFunction = reclassification,
        precalculationParams = NULL, objectiveFunction =
        function(result, saveModel)
        {
        if(result$model$classifier$name != "svm")
        stop("This objective function can only be applied
              to classifiers of type tunePareto.svm()")

       res <- result$model$model$tot.nSV

      if (saveModel)
      # return a list containing the objective value as well as the model
      {
         return(list(additionalData = result$model, fitness = res))
      }
      else
      # only return the objective value
        return(res)
        },
        objectiveFunctionParams = list(saveModel = saveModel),
        direction = "minimize",
        name = "Reclass.SupportVectors")
}

# tune error vs. number of support vectors on the 'iris' data set
r <- tunePareto(data = iris[, -ncol(iris)],
                labels = iris[, ncol(iris)],
                classifier = tunePareto.svm(),
                cost=c(0.001,0.005,0.01,0.05,0.1,0.5,1,5,10,50),
                objectiveFunctions=list(reclassError(), reclassSupportVectors()))

print(r)
```

`generateCVRuns` *Generate cross-validation partitions*

Description

This function generates a set of partitions for a cross-validation. It can be employed if the same cross-validation settings should be used in the objective functions of several experiments. The resulting fold list can be passed to the cross-validation objective functions (see `predefinedObjectiveFunctions`) and the internal cross-validation precalculation function `crossValidation`.

Usage

```
generateCVRuns(labels,
               ntimes = 10,
               nfold = 10,
               leaveOneOut = FALSE,
               stratified = FALSE)
```

Arguments

labels	A vector of class labels of the data set to be used for the cross-validation.
nfold	The number of groups of the cross-validation. Ignored if leaveOneOut=TRUE.
ntimes	The number of repeated runs of the cross-validation. Ignored if leaveOneOut=TRUE.
leaveOneOut	If this is true, a leave-one-out cross-validation is performed, i.e. each sample is left out once in the training phase and used as a test sample
stratified	If set to true, a stratified cross-validation is carried out. That is, the percentage of samples from different classes in the cross-validation folds corresponds to the class sizes in the complete data set. If set to false, the folds may be unbalanced.

Value

A list with ntimes elements, each representing a cross-validation run. Each of the runs is a list of nfold vectors specifying the indices of the samples to be left out in the folds.

See Also

predefinedObjectiveFunctions, crossValidation

Examples

```
# precalculate the cross-validation partitions
foldList <- generateCVRuns(labels = iris[, ncol(iris)],
                           ntimes = 10,
                           nfold = 10,
                           stratified=TRUE)

 # build a list of objective functions
objectiveFunctions <- list(cvError(foldList=foldList),
                           cvSensitivity(foldList=foldList,caseClass="setosa"))

# pass them to tunePareto
print(tunePareto(data = iris[, -ncol(iris)],
                 labels = iris[, ncol(iris)],
                 classifier = tunePareto.knn(),
                 k = c(3,5,7,9),
                 objectiveFunctions = objectiveFunctions))
```

mergeTuneParetoResults *Calculate optimal solutions from several calls of tunePareto*

Description

Merges the results of multiple TuneParetoResult objects as returned by tunePareto, and recalculates the optimal solutions for the merged solution set. All supplied TuneParetoResult objects must use the same objective functions.

Usage

```
mergeTuneParetoResults(...)
```

Arguments

...	A set of TuneParetoResult objects to be merged.

Value

A TuneParetoResult object containing the parameter configurations of all objects in the ... argument and selecting the Pareto-optimal solutions among all these configurations. For more details on the object structure, refer to tunePareto.

See Also

tunePareto, recalculateParetoSet

Examples

```
# optimize an SVM with small costs on
# the 'iris' data set
r1 <- tunePareto(classifier = tunePareto.svm(),
                 data = iris[, -ncol(iris)],
                 labels = iris[, ncol(iris)],
                 cost=seq(0.01,0.1,0.01),
                 objectiveFunctions=list(cvWeightedError(10, 10),
                                         cvSensitivity(10, 10, caseClass="setosa")))
print(r1)

# another call to tunePareto with higher costs
r2 <- tunePareto(classifier = tunePareto.svm(),
                 data = iris[, -ncol(iris)],
                 labels = iris[, ncol(iris)],
                 cost=seq(0.5,10,0.5),
                 objectiveFunctions=list(cvWeightedError(10, 10),
                                         cvSensitivity(10, 10, caseClass="setosa")))
print(r2)
```

```
# merge the results
print(mergeTuneParetoResults(r1,r2))
```

---

plotDominationGraph *Visualize the Pareto fronts of parameter configuration scores*

Description

Draws the Pareto fronts and domination relations of tested parameter configurations in a graph. Here, the leftmost column of nodes represents the non-dominated configurations (i.e. the first Pareto front). The second column contains the second Pareto front, i.e. the configurations that are only dominated by the first Pareto front, and so on. An edge between two configurations indicate that the first configuration is dominated by the second.

Usage

```
plotDominationGraph(tuneParetoResult,
                    transitiveReduction = TRUE,
                    drawDominatedObjectives = TRUE,
                    drawLabels = TRUE,
                    drawLegend = TRUE,
                    x.legend = "topleft",
                    cex.legend = 0.7,
                    col.indicator,
                    pch.indicator = 15,
                    cex.indicator = 0.8,
                    ...)
```

Arguments

`tuneParetoResult`
: An object of class `TuneParetoResult` as returned by `tunePareto`.

`transitiveReduction`
: If this is true, transitive edges in the graph are removed to enhance readability. That is, if configuration `c1` dominates configuration `c2` and `c2` dominates `c3`, no edge from `c3` to `c1` is drawn.

`drawDominatedObjectives`
: If set to true, color indicators are drawn next to the nodes. Here, each color corresponds to one objective. The color is drawn next to a node if this node has the best score in this objectives among all solutions of the same Pareto front (i.e., column of the graph).

`drawLabels`
: Specifies whether the parameter configurations should be printed next to the corresponding edges.

`drawLegend`
: If `drawDominatedObjectives=TRUE`, this specifies whether a legend with the objective colors should be drawn.

x.legend	The position of the legend. For details, refer to the x parameter of legend.
cex.legend	Specifies the size of the text in the legend if drawLegend is true.
col.indicator	Specifies an optional list of colors, one for each objective function. These colors will be used for the indicators if drawDominatedObjectives is true. By default, a predefined set of colors is used.
pch.indicator	Specifies a single plotting character or a list of plotting characters for the objective functions in the indicators which is used for the indicators if drawDominatedObjectives is true.
cex.indicator	Specifies the size of the symbols in the indicators which is be used for the indicators if drawDominatedObjectives is true. This can also be a vector of sizes for the symbols of the objectives.
...	Further graphical parameters for plot.igraph.

Value

Invisibly returns the igraph object representing the graph.

See Also

tunePareto

Examples

```
# call tunePareto using a k-NN classifier
# with different 'k' and 'l' on the 'iris' data set
x <- tunePareto(data = iris[, -ncol(iris)],
                labels = iris[, ncol(iris)],
                classifier = tunePareto.knn(),
                k = c(5,7,9),
                l = c(1,2,3),
                objectiveFunctions=list(cvError(10, 10),
                                        cvSpecificity(10, 10, caseClass="setosa")))

# plot the graph
plotDominationGraph(x)
```

---

plotObjectivePairs *Plot a matrix of Pareto front panels*

Description

Plots a matrix of Pareto front panels for each pair of objectives. The plot for n objectives consists of n x n panels, where the panel in row i and column j depicts the Pareto fronts of the i-th and the j-th objective. Each of the panels is drawn in the same way as plotParetoFronts2D.

Usage

```
plotObjectivePairs(tuneParetoResult,
                   drawLabels = TRUE,
                   drawBoundaries = TRUE,
                   labelPos = 4,
                   fitLabels=TRUE,
                   cex.conf=0.5,
                   lty.fronts=1,
                   pch.fronts=8,
                   col.fronts,
                   ...)
```

Arguments

`tuneParetoResult`	An object of class `TuneParetoResult` as returned by `tunePareto`.
`drawLabels`	If set to true, the descriptions of the configurations are printed next to the points in the plot.
`drawBoundaries`	If set to true, the upper or lower objective limits supplied in the `objectiveBoundaries` parameter of `tunePareto` are drawn as horizontal and vertical lines.
`labelPos`	The position of the configuration labels in the plot (if `drawLabels` is true). Values of 1, 2, 3 and 4 denote positions below, to the left of, above and to the right of the points on the Pareto fronts.
`fitLabels`	If this parameter is true (and `drawLabels` is true), overlapping or partially hidden configuration labels are removed from the plot to improve the readability of the remaining labels.
`cex.conf`	The size of the configuration labels in the plots (if `drawLabels` is true).
`lty.fronts`	A vector of line types to use for the Pareto fronts. By default, straight lines are drawn for all fronts.
`pch.fronts`	A vector of symbols to use for points on the Pareto fronts. All points on the same front will have the same symbol. By default, an asterisk is used.
`col.fronts`	A vector of colors to use for the Pareto fronts. By default, a predefined set of colors is used.
`...`	Further graphical parameters to be passed to the `plot` function.

Value

This function does not have a return value.

See Also

`tunePareto`, `plotParetoFronts2D`, `plotDominationGraph`

Examples

```
# optimize the 'cost' parameter of an SVM according
# to CV error, CV error variance, and CV Specificity
# on two classes of the 'iris' data set
r <- tunePareto(data = iris[, -ncol(iris)],
                labels = iris[, ncol(iris)],
                classifier = tunePareto.svm(),
                cost=c(0.001,0.005,0.01,0.05,0.1,0.5,1,5,10,50),
                objectiveFunctions=list(cvError(10, 10),
                                        cvErrorVariance(10, 10),
                                        cvSpecificity(10, 10, caseClass="virginica")))

# plot the matrix of Pareto fronts
plotObjectivePairs(r)
```

---

plotParetoFronts2D *A classical 2-dimensional plot of Pareto fronts*

Description

Draws a classical Pareto front plot of 2 objectives of a parameter tuning. The first objective is on the x axis of the plot, and the second objective is on the y axis. Points on a Pareto front are connected by lines. Each Pareto front is drawn in a different color.

Usage

```
plotParetoFronts2D(tuneParetoResult,
                   objectives,
                   drawLabels = TRUE,
                   drawBoundaries = TRUE,
                   labelPos = 4,
                   fitLabels=TRUE,
                   cex.conf=0.5,
                   lty.fronts=1,
                   pch.fronts=8,
                   col.fronts,
                   ...)
```

Arguments

`tuneParetoResult`	An object of class `TuneParetoResult` as returned by `tunePareto`.
`objectives`	The names or indices of the two objectives to plot. Pareto-optimality is determined only on the basis of these two objectives. Optional if the parameter tuning has exactly two objectives.

drawLabels	If set to true, the descriptions of the configurations are printed next to the points in the plot.
drawBoundaries	If set to true, the upper or lower objective limits supplied in the objectiveBoundaries parameter of tunePareto are drawn as horizontal and vertical lines.
labelPos	The position of the configuration labels in the plot (if drawLabels is true). Values of 1, 2, 3 and 4 denote positions below, to the left of, above and to the right of the points on the Pareto fronts.
fitLabels	If this parameter is true (and drawLabels is true), overlapping or partially hidden configuration labels are removed from the plot to improve the readability of the remaining labels.
cex.conf	The size of the configuration labels in the plots (if drawLabels is true).
lty.fronts	A vector of line types to use for the Pareto fronts. By default, straight lines are drawn for all fronts.
pch.fronts	A vector of symbols to use for points on the Pareto fronts. All points on the same front will have the same symbol. By default, an asterisk is used.
col.fronts	A vector of colors to use for the Pareto fronts. By default, a predefined set of colors is used.
...	Further graphical parameters to be passed to the plot function.

Value

This function does not have a return value.

See Also

tunePareto, plotDominationGraph

Examples

```
# optimize the 'cost' parameter according
# to CV error and CV Specificity on the 'iris' data set
r <- tunePareto(data = iris[, -ncol(iris)],
                labels = iris[, ncol(iris)],
                classifier = tunePareto.svm(),
                cost=c(0.001,0.005,0.01,0.05,0.1,0.5,1,5,10,50),
                objectiveFunctions=list(cvError(10, 10),
                                        cvSpecificity(10, 10, caseClass="setosa")))

# plot the Pareto graph
plotParetoFronts2D(r)
```

precalculation *Predefined precalculation functions for objectives*

Description

These predefined precalculation functions can be employed to create own objectives using `createObjective`. They perform a reclassification or a cross-validation and return the true labels and the predictions.

Usage

```
reclassification(data, labels,
                 classifier, classifierParams, predictorParams)

crossValidation(data, labels,
                classifier, classifierParams, predictorParams,
                ntimes = 10, nfold = 10,
                leaveOneOut = FALSE, stratified = FALSE,
                foldList = NULL)
```

Arguments

`data`	The data set to be used for the precalculation. This is usually a matrix or data frame with the samples in the rows and the features in the columns.
`labels`	A vector of class labels for the samples in `data`.
`classifier`	A `TuneParetoClassifier` wrapper object containing the classifier to tune. A number of state-of-the-art classifiers are included in **TunePareto** (see `predefinedClassifiers`). Custom classifiers can be employed using `tuneParetoClassifier`.
`classifierParams`	A named list of parameter assignments for the training routine of the classifier.
`predictorParams`	If the classifier consists of separate training and prediction functions, a named list of parameter assignments for the predictor function.
`nfold`	The number of groups of the cross-validation. Ignored if `leaveOneOut=TRUE`.
`ntimes`	The number of repeated runs of the cross-validation. Ignored if `leaveOneOut=TRUE`.
`leaveOneOut`	If this is true, a leave-one-out cross-validation is performed, i.e. each sample is left out once in the training phase and used as a test sample
`stratified`	If set to true, a stratified cross-validation is carried out. That is, the percentage of samples from different classes in the cross-validation folds corresponds to the class sizes in the complete data set. If set to false, the folds may be unbalanced.
`foldList`	If this parameter is set, the other cross-validation parameters (`ntimes`, `nfold`, `leaveOneOut`, `stratified`) are ignored. Instead, the precalculated cross-validation partition supplied in `foldList` is used. This allows for using the same cross-validation experiment in multiple `tunePareto` calls. Partitions can be generated using `generateCVRuns`.

Details

`reclassification` trains the classifier with the full data set. Afterwards, the classifier is applied to the same data set.

`crossValidate` partitions the samples in the data set into a number of groups (depending on `nfold` and `leaveOneOut`). Each of these groups is left out once in the training phase and used for prediction. The whole procedure is repeated several times (as specified in `ntimes`).

Value

`reclassification` returns a list with the following components:

trueLabels The original labels of the dataset as supplied in `labels`

predictedLabels A vector of predicted labels of the data set

model The `TuneParetoModel` object resulting from the classifier training

`crossValidation` returns a nested list structure. At the top level, there is one list element for each run of the cross-validation. Each of these elements consists of a list of sub-structures for each fold. The sub-structures have the following components:

trueLabels The original labels of the test samples in the fold

predictedLabels A vector of predicted labels of the test samples in the fold

model The `TuneParetoModel` object resulting from the classifier training in the fold

That is, for a cross-validation with `n` runs and `m` folds, there are `n` top-level lists, each having `m` sub-lists comprising the true labels and the predicted labels.

See Also

`createObjective`, `generateCVRuns`.

Examples

```
# create new objective minimizing the
# false positives of a reclassification

cvFalsePositives <- function(nfold=10, ntimes=10, leaveOneOut=FALSE, foldList=NULL, caseClass)
{
  return(createObjective(
            precalculationFunction = "crossValidation",
            precalculationParams = list(nfold=nfold,
                                        ntimes=ntimes,
                                        leaveOneOut=leaveOneOut,
                                        foldList=foldList),
            objectiveFunction =
            function(result, caseClass)
            {

              # take mean value over the cv runs
              return(mean(sapply(result,
```

```
                    function(run)
                    # iterate over runs of cross-validation
                    {
                      # extract all predicted labels in the folds
                      predictedLabels <-
                            unlist(lapply(run,
                                          function(fold)fold$predictedLabels))

                      # extract all true labels in the folds
                      trueLabels <-
                            unlist(lapply(run,
                                           function(fold)fold$trueLabels))

                      # calculate number of false positives in the run
                      return(sum(predictedLabels == caseClass &
                                 trueLabels != caseClass))
                    }))
            },
            objectiveFunctionParams = list(caseClass=caseClass),
            direction = "minimize",
            name = "CV.FalsePositives"))
}

# use the objective in an SVM cost parameter tuning on the 'iris' data set
r <- tunePareto(data = iris[, -ncol(iris)],
                labels = iris[, ncol(iris)],
                classifier = tunePareto.svm(),
                cost = c(0.001,0.005,0.01,0.05,0.1,0.5,1,5,10,50),
                objectiveFunctions=list(cvFalsePositives(10, 10, caseClass="setosa")))
print(r)
```

---

predefinedClassifiers *TunePareto wrappers for certain classifiers*

Description

Creates TunePareto classifier objects for the k-Nearest Neighbour classifier, support vector machines, and trees.

Usage

```
tunePareto.knn()

tunePareto.svm()

tunePareto.tree()

tunePareto.randomForest()

tunePareto.NaiveBayes()
```

Details

tunePareto.knn encapsulates a k-Nearest Neighbour classifier as defined in link[class]{knn} in package **class**. The classifier allows for supplying and tuning the following parameters of link[class]{knn}:

k, l, use.all

tunePareto.svm encapsulates the support vector machine svm classifier in package **e1071**. The classifier allows for supplying and tuning the following parameters:

kernel, degree, gamma, coef0, cost, nu, class.weights, cachesize, tolerance, epsilon, scale, shrinking, fitted, subset, na.action

tunePareto.tree encapsulates the CART classifier tree in package **tree**. The classifier allows for supplying and tuning the following parameters:

weights, subset, na.action, method, split, mincut, minsize, mindev

as well as the type parameter of predict.tree.

tunePareto.randomForest encapsulates the randomForest classifier in package **randomForest**. The classifier allows for supplying and tuning the following parameters:

subset, na.action, ntree, mtry, replace, classwt, cutoff, strata, sampsize, nodesize, maxnodes

tunePareto.NaiveBayes encapsulates the NaiveBayes classifier in package **klaR**. The classifier allows for supplying and tuning the following parameters:

prior, usekernel, fL, subset, na.action, bw, adjust, kernel, weights, window, width, give.Rkern, n, from, to, cut, na.rm

Value

Returns objects of class TuneParetoClassifier as described in tuneParetoClassifier. These can be passed to functions like tunePareto or trainTuneParetoClassifier.

See Also

tuneParetoClassifier, tunePareto, trainTuneParetoClassifier

Examples

```
# tune a k-NN classifier with different 'k' and 'l'
# on the 'iris' data set
print(tunePareto(classifier = tunePareto.knn(),
                 data = iris[, -ncol(iris)],
                 labels = iris[, ncol(iris)],
                 k = c(5,7,9),
                 l = c(1,2,3),
                 objectiveFunctions=list(cvError(10, 10),
                                         cvSpecificity(10, 10, caseClass="setosa"))))

# tune an SVM with different costs on
# the 'iris' data set
# using Halton sequences for sampling
```

```
print(tunePareto(classifier = tunePareto.svm(),
                 data = iris[, -ncol(iris)],
                 labels = iris[, ncol(iris)],
                 cost = as.interval(0.001,10),
                 sampleType = "halton",
                 numCombinations=20,
                 objectiveFunctions=list(cvWeightedError(10, 10),
                                         cvSensitivity(10, 10, caseClass="setosa"))))

# tune a CART classifier with different
# splitting criteria on the 'iris' data set
print(tunePareto(classifier = tunePareto.tree(),
                 data = iris[, -ncol(iris)],
                 labels = iris[, ncol(iris)],
                 split = c("deviance","gini"),
                 objectiveFunctions=list(cvError(10, 10),
                                         cvErrorVariance(10, 10))))

# tune a Random Forest with different numbers of trees
# on the 'iris' data set
print(tunePareto(classifier = tunePareto.randomForest(),
                 data = iris[, -ncol(iris)],
                 labels = iris[, ncol(iris)],
                 ntree = seq(50,300,50),
                 objectiveFunctions=list(cvError(10, 10),
                                         cvSpecificity(10, 10, caseClass="setosa"))))

# tune a Naive Bayes classifier with different kernels
# on the 'iris' data set
print(tunePareto(classifier = tunePareto.NaiveBayes(),
                 data = iris[, -ncol(iris)],
                 labels = iris[, ncol(iris)],
                 kernel = c("gaussian", "epanechnikov", "rectangular",
                            "triangular", "biweight",
                            "cosine", "optcosine"),
                 objectiveFunctions=list(cvError(10, 10),
                                         cvSpecificity(10, 10, caseClass="setosa"))))
```

`predefinedObjectiveFunctions` *Predefined objective functions for parameter tuning*

Description

Predefined objective functions that calculate several performance criteria of reclassification or cross-validation experiments.

Usage

```
reclassAccuracy(saveModel = FALSE)
reclassError(saveModel = FALSE)
reclassWeightedError(saveModel = FALSE)
reclassSensitivity(caseClass, saveModel = FALSE)
reclassRecall(caseClass, saveModel = FALSE)
reclassTruePositive(caseClass, saveModel = FALSE)
reclassSpecificity(caseClass, saveModel = FALSE)
reclassTrueNegative(caseClass, saveModel = FALSE)
reclassFallout(caseClass, saveModel = FALSE)
reclassFalsePositive(caseClass, saveModel = FALSE)
reclassMiss(caseClass, saveModel = FALSE)
reclassFalseNegative(caseClass, saveModel = FALSE)
reclassPrecision(caseClass, saveModel = FALSE)
reclassPPV(caseClass, saveModel = FALSE)
reclassNPV(caseClass, saveModel = FALSE)
reclassConfusion(trueClass, predictedClass, saveModel = FALSE)

cvAccuracy(nfold = 10, ntimes = 10,
           leaveOneOut = FALSE, stratified = FALSE,
           foldList = NULL,
           saveModel = FALSE)
cvError(nfold = 10, ntimes = 10,
        leaveOneOut = FALSE, stratified=FALSE,
        foldList=NULL,
        saveModel = FALSE)
cvErrorVariance(nfold = 10, ntimes = 10,
                leaveOneOut = FALSE, stratified=FALSE,
                foldList=NULL,
                saveModel = FALSE)
cvWeightedError(nfold = 10, ntimes = 10,
                leaveOneOut = FALSE, stratified=FALSE,
                foldList=NULL,
                saveModel = FALSE)
cvSensitivity(nfold = 10, ntimes = 10,
              leaveOneOut = FALSE, stratified=FALSE,
              foldList=NULL, caseClass,
              saveModel = FALSE)
cvRecall(nfold = 10, ntimes = 10,
         leaveOneOut = FALSE, stratified=FALSE,
         foldList=NULL, caseClass,
         saveModel = FALSE)
cvTruePositive(nfold = 10, ntimes = 10,
               leaveOneOut = FALSE, stratified=FALSE,
               foldList=NULL, caseClass,
               saveModel = FALSE)
cvSpecificity(nfold = 10, ntimes = 10,
              leaveOneOut = FALSE, stratified=FALSE,
```

```
              foldList=NULL, caseClass,
              saveModel = FALSE)
cvTrueNegative(nfold = 10, ntimes = 10,
               leaveOneOut = FALSE, stratified=FALSE,
               foldList=NULL, caseClass,
               saveModel = FALSE)
cvFallout(nfold = 10, ntimes = 10,
          leaveOneOut = FALSE, stratified=FALSE,
          foldList=NULL, caseClass,
          saveModel = FALSE)
cvFalsePositive(nfold = 10, ntimes = 10,
                leaveOneOut = FALSE, stratified=FALSE,
                foldList=NULL, caseClass,
                saveModel = FALSE)
cvMiss(nfold = 10, ntimes = 10,
       leaveOneOut = FALSE, stratified=FALSE,
       foldList=NULL, caseClass,
       saveModel = FALSE)
cvFalseNegative(nfold = 10, ntimes = 10,
                leaveOneOut = FALSE, stratified=FALSE,
                foldList=NULL, caseClass,
                saveModel = FALSE)
cvPrecision(nfold = 10, ntimes = 10,
            leaveOneOut = FALSE, stratified=FALSE,
            foldList=NULL, caseClass,
            saveModel = FALSE)
cvPPV(nfold = 10, ntimes = 10,
      leaveOneOut = FALSE, stratified=FALSE,
      foldList=NULL, caseClass,
      saveModel = FALSE)
cvNPV(nfold = 10, ntimes = 10,
      leaveOneOut = FALSE, stratified=FALSE,
      foldList=NULL, caseClass,
      saveModel = FALSE)
cvConfusion(nfold = 10, ntimes = 10,
            leaveOneOut = FALSE, stratified=FALSE,
            foldList=NULL, trueClass, predictedClass,
            saveModel = FALSE)
```

Arguments

`nfold`	The number of groups of the cross-validation. Ignored if `leaveOneOut=TRUE`.
`ntimes`	The number of repeated runs of the cross-validation. Ignored if `leaveOneOut=TRUE`.
`leaveOneOut`	If this is true, a leave-one-out cross-validation is performed, i.e. each sample is left out once in the training phase and used as a test sample
`stratified`	If set to true, a stratified cross-validation is carried out. That is, the percentage of samples from different classes in the cross-validation folds corresponds to the class sizes in the complete data set. If set to false, the folds may be unbalanced.

`foldList` If this parameter is set, the other cross-validation parameters (`ntimes`, `nfold`, `leaveOneOut`, `stratified`) are ignored. Instead, the precalculated cross-validation partition supplied in `foldList` is used. This allows for using the same cross-validation experiment in multiple `tunePareto` calls. Partitions can be generated using `generateCVRuns`.

`caseClass` The class containing the positive samples for the calculation of specificity and sensitivity. All samples with different class labels are regarded as controls (negative samples).

`trueClass` When calculating the confusion of two classes, the class to which a sample truly belongs.

`predictedClass` When calculating the confusion of two classes, the class to which a sample is erreneously assigned.

`saveModel` If set to true, the trained model(s) are stored to the `additionalData` component of the resulting `TuneParetoResult` objects (see `tunePareto` for details). In case of a reclassification, a single model is stored. In case of a cross-validation, a list of length `nruns`, each containing a sub-list of `nfold` models, is stored. If the size of a model is large, setting `saveModel = TRUE` can result in a high memory consumption. As the model information is the same for all reclassification objectives or for cross-validation objectives with the same parameters, it is usually sufficient to set `saveModel=TRUE` for only one of the objective functions.

Details

The functions do not calculate the objectives directly, but return a structure of class `TuneParetoObjectives` that provides all information on the objective function for later use in `tunePareto`.

The behaviour of the functions in `tunePareto` is as follows:

The reclassification functions train the classifiers with the full data set. Afterwards, the classifiers are applied to the same data set. `reclassAccuracy` measures the fraction of correctly classified samples, while `reclassError` calculates the fraction of misclassified samples. `reclassWeightedError` calculates the sum of fractions of misclassified samples in each class weighted by the class size. `reclassSensitivity` measures the sensitivity, and `reclassSpecificity` measures the specificity of the reclassification experiment. `reclassTruePositive` and `reclassRecall` are aliases for `reclassSensitivity`, and `reclassTrueNegative` is an alias for `reclassSpecificity`. `reclassFallout` and its equivalent alias `reclassFalsePositive` give the ratio of false positives to all negative samples, and `reclassMiss` and its alias `reclassFalseNegative` measure the ratio of false negatives to all positive samples. `reclassPrecision` calculates the precision of the reclassification experiment, i.e. the ratio of true positives to all samples classified as positive. This is equivalent to the positive predictive value (`reclassPPV`). `reclassNPV` measures the negative predictive value, i.e. the ratio of true negatives to all samples classified as negative. `reclassConfusion` calculates the fraction of samples in `trueClass` that have been confused with `predictedClass`.

`reclassError`, `reclassWeightedError`, `reclassFallout`, `reclassFalsePositive`, `reclassMiss`, `reclassFalsePositive` and `reclassConfusion` are minimization objectives, whereas `reclassAccuracy`, `reclassSensitivity`, `reclassTruePositive`, `reclassRecall`, `reclassSpecificity`, `reclassTrueNegative` `reclassPrecision`, `reclassPPV` and `reclassNPV` are maximization objectives.

The cross-validation functions partition the samples in the data set into a number of groups (depending on `nfold` and `leaveOneOut`). Each of these groups is left out once in the training phase and

used for prediction. The whole procedure is usually repeated several times (as specified in ntimes), and the results are averaged. Similar to the reclassification functions, cvAccuracy calculates the fraction of correctly classified samples over the runs, cvError calculates the average fraction of misclassified samples over the runs, and cvWeightedError calculates the mean sum of fractions of misclassified samples in each class weighted by the class size. cvErrorVariance calculates the variance of the cross-validation error. cvSensitivity, cvRecall and cvTruePositive calculate the average sensitivity, and cvSpecificity and cvTrueNegative calculate the average specificity. cvFallout and cvFalsePositive calculate the average false positive rate over the runs. cvMiss and cvFalseNegative calculate the average false negative rate over the runs. cvPrecision and cvPPV calculate the average precision/positive predictive rate. cvNPV gives the average negative predictive rate over all runs. cvConfusion calculates the average fraction of samples in trueClass that have been confused with predictedClass.

cvError, cvWeightedError, cvErrorVariance, cvFallout, cvFalsePositive, cvMiss, cvFalseNegative and cvConfusion are minimization objectives, and cvAccuracy, cvSensitivity, cvRecall, cvTruePositive, cvSpecificity, cvTrueNegative, cvPrecision, cvPPV and cvNPV are maximization objectives.

Value

An object of class TuneParetoObjective representing the objective function. For more details, see createObjective.

See Also

createObjective, tunePareto, generateCVRuns

Examples

```
# build a list of objective functions
objectiveFunctions <- list(cvError(10, 10),
                           reclassSpecificity(caseClass="setosa"),
                           reclassSensitivity(caseClass="setosa"))

# pass them to tunePareto
print(tunePareto(data = iris[, -ncol(iris)],
                 labels = iris[, ncol(iris)],
                 classifier = tunePareto.knn(),
                 k = c(3,5,7,9),
                 objectiveFunctions = objectiveFunctions))
```

---

predict.TuneParetoModel

Prediction method for TuneParetoClassifier objects

Description

S3 method that predicts the labels of unknown samples using a trained TuneParetеModel model of a TuneParetoClassifier object.

Usage

```
## S3 method for class 'TuneParetoModel'
predict(object, newdata, ...)
```

Arguments

object	A TuneParetoTraining object as returned by trainTuneParetoClassifier.
newdata	The samples whose class labels are predicted. This is usually a matrix or data frame with the samples in the rows and the features in the columns.
...	Further parameters for the predictor. These must be defined in the predictorParamNames argument of tuneParetoClassifier.

Value

Returns a vector of class labels for the samples in newdata-

See Also

tuneParetoClassifier, predefinedClassifiers, trainTuneParetoClassifier

Examples

```
# train an SVM classifier
cl <- tunePareto.svm()
tr <- trainTuneParetoClassifier(cl,
                                iris[,-ncol(iris)],
                                iris[,ncol(iris)],
                                cost=0.001)

# re-apply the classifier to predict the training data
print(iris[,ncol(iris)])
print(predict(tr, iris[,-ncol(iris)]))
```

---

print.TuneParetoResult

Print method for objects used in TunePareto

Description

Customized printing methods for several objects used in TunePareto: For TuneParetoResult objects, the Pareto-optimal parameter configurations are printed. For TuneParetoClassifier and TuneParetoModel objects, information on the classifier and its parameters is printed.

Usage

```
## S3 method for class 'TuneParetoResult'
print(x, ...)
## S3 method for class 'TuneParetoClassifier'
print(x, ...)
## S3 method for class 'TuneParetoModel'
print(x, ...)
```

Arguments

x	An object of class TuneParetoResult, TuneParetoClassifier or TuneParetoModel to be printed.
...	Further parameters (currently unused).

Value

Invisibly returns the printed object.

See Also

tunePareto, tuneParetoClassifier, trainTuneParetoClassifier

---

rankByDesirability *Rank results according to their desirabilities*

Description

Calculates the desirability index for each Pareto-optimal combination (or for all combinations), and ranks the combinations according to this value. The desirability index was introduced by Harrington in 1965 for multicriteria optimization. Desirability functions specify the desired values of each objective and are aggregated in a single desirability index.

Usage

```
rankByDesirability(tuneParetoResult,
                   desirabilityIndex,
                   optimalOnly = TRUE)
```

Arguments

tuneParetoResult	A TuneParetoResult object containing the parameter configurations to be examined
desirabilityIndex	A function accepting a vector of objective values and returning a desirability index in [0,1].
optimalOnly	If set to true, only the Pareto-optimal solutions are ranked. Otherwise, all tested solutions are ranked. Defaults to TRUE.

Value

A matrix of objective values with an additional column for the desirability index. The rows of the matrix are sorted according to the index.

Examples

```
# optimize the 'cost' parameter of an SVM on
# the 'iris' data set
res <- tunePareto(classifier = tunePareto.svm(),
                  data = iris[, -ncol(iris)],
                  labels = iris[, ncol(iris)],
                  cost=c(0.01,0.05,0.1,0.5,1,5,10,50,100),
                  objectiveFunctions=list(cvWeightedError(10, 10),
                                          cvSensitivity(10, 10, caseClass="setosa"),
                                          cvSpecificity(10, 10, caseClass="setosa")))

# create desirability functions
# aggregate functions in desirability index (e.g. harrington desirability function)
# here, for the sake of simplicity a random number generator
di <- function(x) {runif(1)}

# rank all tuning results according to their desirabilities
print(rankByDesirability(res,di,optimalOnly=FALSE))
```

---

recalculateParetoSet *Recalculate Pareto-optimal solutions*

Description

Recalculates the Pareto-optimal solutions in a `TuneParetoResult` according to the specified objectives only, and returns another `TuneParetoResult` object reduced to these objectives. This avoids time-consuming recalculations of objective values if only a subset of objectives should be considered for a previously evaluated set of parameter combinations.

Usage

```
recalculateParetoSet(tuneParetoResult,
                     objectives)
```

Arguments

`tuneParetoResult`
: The `TuneParetoResult` object containing the parameter configurations to be examined

objectives — A vector of objective function indices. The Pareto set is recalculated according to these objectives, i.e. omitting other objectives. If this argument is not supplied, all objectives are used, which usually returns a copy of the input.

Value

Returns a reduced TuneParetoResult object. For more details on the object structure, refer to tunePareto.

See Also

tunePareto, mergeTuneParetoResults

Examples

```
# optimize the 'cost' parameter of an SVM on
# the 'iris' data set
res <- tunePareto(classifier = tunePareto.svm(),
                  data = iris[, -ncol(iris)],
                  labels = iris[, ncol(iris)],
                  cost=seq(0.01,0.1,0.01),
                  objectiveFunctions=list(cvWeightedError(10, 10),
                                          cvSensitivity(10, 10, caseClass="setosa"),
                                          cvSpecificity(10, 10, caseClass="setosa")))
print(res)

# select only specificity and sensitivity
print(recalculateParetoSet(res, 2:3))
```

---

trainTuneParetoClassifier

Train a TunePareto classifier

Description

Trains a classifier wrapped in a TuneParetoClassifier object. The trained classifier model can then be passed to predict.TuneParetoModel.

Usage

```
trainTuneParetoClassifier(classifier, trainData, trainLabels, ...)
```

Arguments

classifier	A TuneParetoClassifier object as returned by tuneParetoClassifier or one of the predefined classification functions (see predefinedClassifiers).
trainData	The data set to be used for the classifier training. This is usually a matrix or data frame with the samples in the rows and the features in the columns.
trainLabels	A vector of class labels for the samples in trainData.
...	Further parameters to be passed to the classifier. These must be parameters specified in the classifierParameterNames parameter of tuneParetoClassifier and usually correspond to the tuned parameters.

Value

Returns an object of class TuneParetoModel with the following entries

classifier	The classifier object supplied in the classifier parameter
classifierParams	The additional parameters supplied to the classifier in the ... parameter
trainData	If classifier is an all-in-one classifier without a separate prediction method, this stores the input training data.
trainLabels	If classifier is an all-in-one classifier without a separate prediction method, this stores the input training labels.
model	If classifier consists of separate training and prediction methods, this contains the trained classifier model.

See Also

tuneParetoClassifier, predefinedClassifiers, predict.TuneParetoModel

Examples

```
# train an SVM classifier
cl <- tunePareto.svm()
tr <- trainTuneParetoClassifier(cl,
                                iris[,-ncol(iris)],
                                iris[,ncol(iris)],
                                cost=0.001)

# re-apply the classifier to predict the training data
print(iris[,ncol(iris)])
print(predict(tr, iris[,-ncol(iris)]))
```

tunePareto *Generic function for multi-objective parameter tuning of classifiers*

Description

This generic function tunes parameters of arbitrary classifiers in a multi-objective setting and returns the Pareto-optimal parameter combinations.

Usage

```
tunePareto(..., data, labels,
           classifier, parameterCombinations,
           sampleType = c("full","uniform",
                          "latin","halton",
                          "niederreiter","sobol",
                          "evolution"),
           numCombinations,
           mu=10, lambda=20, numIterations=100,
           objectiveFunctions, objectiveBoundaries,
           keepSeed = TRUE, useSnowfall = FALSE, verbose=TRUE)
```

Arguments

data	The data set to be used for the parameter tuning. This is usually a matrix or data frame with the samples in the rows and the features in the columns.
labels	A vector of class labels for the samples in data.
classifier	A TuneParetoClassifier wrapper object containing the classifier to tune. A number of state-of-the-art classifiers are included in **TunePareto** (see predefinedClassifiers). Custom classifiers can be employed using tuneParetoClassifier.
parameterCombinations	If not all combinations of parameter ranges for the classifier are meaningful, you can set this parameter instead of specifying parameter values in the ... argument. It holds an explicit list of possible combinations, where each element of the list is a named sublist with one entry for each parameter.
sampleType	Determines the way parameter configurations are sampled. If type="full", all possible combinations are tried. This is only possible if all supplied parameter ranges are discrete or if the combinations are supplied explicitly in parameterCombinations. If type="uniform", numCombinations combinations are drawn uniformly at random. If type="latin", Latin Hypercube sampling is applied. This is particularly encouraged when tuning using continuous parameters. If type="halton","niederreiter","sobol", numCombinations parameter combinations are drawn on the basis of the corresponding quasi-random sequences (initialized at a random step to ensure that different values are drawn in repeated

runs). This is particularly encouraged when tuning using continuous parameters. `type="niederreiter"` and `type="sobol"` require the **gsl** package to be installed.

If `type="evolution"`, an evolutionary algorithm is applied. In details, this employs `mu+lambda` Evolution Strategies with uncorrelated mutations and non-dominated sorting for survivor selection. This is encouraged when the space of possible parameter configurations is very large. For smaller parameter spaces, the above sampling methods may be faster.

`numCombinations`

If this parameter is set, at most `numCombinations` randomly chosen parameter configurations are tested. Otherwise, all possible combinations of the supplied parameter ranges are tested.

`mu` The number of individuals used in the Evolution Strategies if `type="evolution"`.

`lambda` The number of offspring per generation in the Evolution Strategies if `type="evolution"`.

`numIterations` The number of iterations/generations the evolutionary algorithm is run if `type="evolution"`.

`objectiveFunctions`

A list of objective functions used to tune the parameters. There are a number of predefined objective functions (see `predefinedObjectiveFunctions`). Custom objective functions can be created using `createObjective`.

`objectiveBoundaries`

If this parameter is set, it specifies boundaries of the objective functions for valid solutions. That is, each element of the supplied vector specifies the upper or lower limit of an objective (depending on whether the objective is maximized or minimized). Parameter combinations that do not meet all these restrictions are not included in the result set, even if they are Pareto-optimal. If only some of the objectives should have bounds, supply `NA` for the remaining objectives.

`keepSeed` If this is true, the random seed is reset to the same value for each of the tested parameter configurations. This is an easy way to guarantee comparability in randomized objective functions. E.g., cross-validation runs of the classifiers will all start with the same seed, which results in the same partitions.

Attention: If you set this parameter to `FALSE`, you must ensure that all configuration are treated equally in the objective functions: There may be randomness in processes such as classifier training, but there should be no random difference in the rating itself. In particular, the choice of subsets for subsampling experiments should always be the same for all configurations. For example, you can provide precalculated fold lists to the cross-validation objectives in the `foldList` parameter. If parameter configurations are rated under varying conditions, this may yield over-optimistic or over-pessimistic ratings for some configurations due to outliers.

`useSnowfall` If this parameter is true, the routine loads the **snowfall** package and processes the parameter configurations in parallel. Please note that the **snowfall** cluster has to be initialized properly before running the tuning function and stopped after the run.

`verbose` If this parameter is true, status messages are printed. In particular, the algorithm prints the currently tested combination.

`...` The parameters of the classifier and predictor functions that should be tuned. The names of the parameters must correspond to the parameters specified in `classifierParameterNames` and `predictorParameterNames` of `tuneParetoClassifier`. Each supplied argument describes the possible values of a single parameter. These can be specified in two ways: Discrete parameter ranges are specified as lists of possible values. Continous parameter ranges are specified as intervals using `as.interval`. The algorithm then generates combinations of possible parameter values. Alternatively, the combinations can be defined explicitly using the `parameterCombinations` parameter.

Details

This is a generic function that allows for parameter tuning of a wide variety of classifiers. You can either specify the values or intervals of tuned parameters in the `...` argument, or supply selected combinations of parameter values using `parameterCombinations`. In the first case, combinations of parameter values specified in the `...` argument are generated. If `sampleType="uniform"`, `sampleType="latin"`, `sampleType="halton"`, `sampleType="niederreiter"` or `sampleType="sobol"`, a random subset of the possible combinations is drawn.
If `sampleType="evolution"`, random parameter combinations are generated and optimized using Evolution Strategies.

In the latter case, only the parameter combinations specified explicitly in `parameterCombinations` are tested. This is useful if certain parameter combinations are invalid. You can create parameter combinations by concatenating results of calls to `allCombinations`. Only `sampleType="full"` is allowed in this mode.

For each of the combinations, the specified objective functions are calculated. This usually involves training and testing a classifier. From the resulting objective values, the non-dominated parameter configurations are calculated and returned.

The `...` argument is the first argument of `tunePareto` for technical reasons (to prevent partial matching of the supplied parameters with argument names of `tunePareto`. This requires all arguments to be named.

Value

Returns a list of class `TuneParetoResult` with the following components:

`bestCombinations`
: A list of Pareto-optimal parameter configurations. Each element of the list consists of a sub-list with named elements corresponding to the parameter values.

`bestObjectiveValues`
: A matrix containing the objective function values of the Pareto-optimal configurations in `bestCombinations`. Each row corresponds to a parameter configuration, and each column corresponds to an objective function.

`testedCombinations`
: A list of all tested parameter configurations with the same structure as `bestCombinations`.

`testedObjectiveValues`
: A matrix containing the objective function values of all tested configurations with the same structure as `bestObjectiveValues`.

`dominationMatrix`
: A Boolean matrix specifying which parameter configurations dominate each other. If a configuration `i` dominates a configuration `j`, the entry in the `i`th row and the `j`th column is `TRUE`.

`minimizeObjectives`
: A Boolean vector specifying which of the objectives are minimization objectives. This is derived from the objective functions supplied to `tunePareto`.

`additionalData`
: A list containing additional data that may have been returned by the objective functions. The list has one element for each tested parameter configuration, each comprising one sub-element for each objective function that returned additional data. The structure of these sub-elements depends on the corresponding objective function. For example, the predefined objective functions (see `predefinedObjectiveFunctions`) save the trained models here if `saveModel` is true.

See Also

`predefinedClassifiers`, `predefinedObjectiveFunctions`, `createObjective`, `allCombinations`

Examples

```
# tune 'k' of a k-NN classifier
# on two classes of the 'iris' data set --
# see ?knn
print(tunePareto(data = iris[, -ncol(iris)],
                 labels = iris[, ncol(iris)],
                 classifier = tunePareto.knn(),
                 k = c(1,3,5,7,9),
                 objectiveFunctions = list(cvError(10, 10),
                                           reclassError())))

# example using predefined parameter configurations,
# as certain combinations of k and l are invalid:
comb <- c(allCombinations(list(k=1,l=0)),
          allCombinations(list(k=3,l=0:2)),
          allCombinations(list(k=5,l=0:4)),
          allCombinations(list(k=7,l=0:6)))

print(tunePareto(data = iris[, -ncol(iris)],
                 labels = iris[, ncol(iris)],
                 classifier = tunePareto.knn(),
                 parameterCombinations = comb,
                 objectiveFunctions = list(cvError(10, 10),
                                           reclassError())))


# tune 'cost' and 'kernel' of an SVM on
# the 'iris' data set using Latin Hypercube sampling --
# see ?svm and ?predict.svm
```

```
print(tunePareto(data = iris[, -ncol(iris)],
                 labels = iris[, ncol(iris)],
                 classifier = tunePareto.svm(),
                 cost = as.interval(0.001,10),
                 kernel = c("linear", "polynomial",
                          "radial", "sigmoid"),
                 sampleType="latin",
                 numCombinations=20,
                 objectiveFunctions = list(cvError(10, 10),
                                           cvSensitivity(10, 10, caseClass="setosa"))))

# tune the same parameters using Evolution Strategies
print(tunePareto(data = iris[, -ncol(iris)],
                 labels = iris[, ncol(iris)],
                 classifier = tunePareto.svm(),
                 cost = as.interval(0.001,10),
                 kernel = c("linear", "polynomial",
                          "radial", "sigmoid"),
                 sampleType="evolution",
                 numCombinations=20,
                 numIterations=20,
                 objectiveFunctions = list(cvError(10, 10),
                                           cvSensitivity(10, 10, caseClass="setosa"),
                                           cvSpecificity(10, 10, caseClass="setosa"))))
```

`tuneParetoClassifier` *Create a classifier object*

Description

Creates a wrapper object mapping all information necessary to call a classifier which can be passed to `tunePareto`.

Usage

```
tuneParetoClassifier(name,
                     classifier,
                     classifierParamNames = NULL,
                     predefinedClassifierParams = NULL,
                     predictor = NULL,
                     predictorParamNames = NULL,
                     predefinedPredictorParams = NULL,
                     useFormula = FALSE,
                     formulaName = "formula",
                     trainDataName = "x",
                     trainLabelName = "y",
                     testDataName = "newdata",
                     modelName = "object",
                     requiredPackages = NULL)
```

Arguments

`name`	A human-readable name of the classifier
`classifier`	The classification function to use. If `predictor` is `NULL`, this function is an all-in-one classification method that receives both training data and test data and returns the predicted labels for the test data. If `predictor` is not `NULL`, this is the training function of the classifier that builds a model from the training data. This model is then passed to `predictor` along with the test data to obtain the predicted labels for the test data.
`classifierParamNames`	A vector of names of possible arguments for `classifier`.
`predefinedClassifierParams`	A named list of default values for the classifier parameters.
`predictor`	If the classification method consists of separate training and prediction functions, this points to the prediction function that receives a model and the test data as inputs and returns the predicted class labels.
`predictorParamNames`	If `predictor != NULL`, a vector of names of possible arguments for `predictor`.
`predefinedPredictorParams`	If `predictor != NULL`, a named list of default values for the parameters of `predictor`.
`useFormula`	Set this to true if the classifier expects a formula to describe the relation between features and class labels. The formula itself is built automatically.
`formulaName`	If `useFormula` is true, this is the name of the parameter of the classifier's training function that holds the formula.
`trainDataName`	The name of the paramater of the classifier's training function that holds the training data.
`trainLabelName`	If `useFormula=FALSE`, the name of the paramater of the classifier's training function that holds the training labels. Otherwise, the training labels are added to the training data and supplied in parameter `trainDataName`.
`testDataName`	If `predictor=NULL`, this is the name of the parameter of `classifier` that receives the test data. Otherwise, it is the parameter of `predictor` that holds the test data.
`modelName`	If `predictor` is not NULL, this is the name of the parameter of `predictor` that receives the training model (i.e., the return value of `classifier`).
`requiredPackages`	A vector containing the names of packages that are required to run the classifier. These packages are loaded automatically when running the classifier using `tunePareto`. They are also loaded in the **snowfall** cluster if necessary.

Details

TunePareto classifier objects are wrappers containing all information necessary to run the classifier, including the training and prediction function, the required packages, and the names of certain arguments. **TunePareto** provides a set of predefined objects for state-of-the-art classifiers (see `predefinedClassifiers`).

The main tunePareto routine evaluates TuneParetoClassifier objects to call the training and prediction methods. Furthermore, direct calls to the classifiers are possible using trainTuneParetoClassifier and predict.TuneParetoModel.

Value

An object of class TuneParetoClassifier with components corresponding to the above parameters.

See Also

trainTuneParetoClassifier, predict.TuneParetoModel, tunePareto, predefinedClassifiers

Examples

```
# equivalent to tunePareto.svm()
cl <- tuneParetoClassifier(name = "svm",
                           classifier = svm,
                           predictor = predict,
                           classifierParamNames = c("kernel", "degree", "gamma",
                                                    "coef0", "cost", "nu",
                                                    "class.weights", "cachesize",
                                                    "tolerance", "epsilon",
                                                    "subset", "na.action"),
                            useFormula = FALSE,
                            trainDataName = "x",
                            trainLabelName = "y",
                            testDataName = "newdata",
                            modelName = "object",
                            requiredPackages="e1071")

# call TunePareto with the classifier
print(tunePareto(classifier = cl,
                 data = iris[, -ncol(iris)],
                 labels = iris[, ncol(iris)],
                 cost = c(0.001,0.01,0.1,1,10),
                 objectiveFunctions=
                   list(cvError(10, 10),
                        cvSpecificity(10, 10,
                          caseClass="setosa"))))
```

Index