Package 'D2MCS'

May 7, 2021

Type Package

Title Data Driving Multiple Classifier System

Version 1.0.0

Description Provides a novel framework to able to automatically develop and deploy an accurate Multiple Classifier System based on the feature-clustering distribution achieved from an input dataset. 'D2MCS' was developed focused on four main aspects: (i) the ability to determine an effective method to evaluate the independence of features, (ii) the identification of the optimal number of feature clusters, (iii) the training and tuning of ML models and (iv) the execution of voting schemes to combine the outputs of each classifier comprising the Multiple Classifier System.

Date 2021-05-05

License GPL-3

URL https://github.com/drordas/D2MCS

BugReports https://github.com/drordas/D2MCS/issues

Depends R (>= 4.0)

Imports caret, devtools, dplyr, FSelector, ggplot2, ggrepel, gridExtra, infotheo, mccr, mltools, ModelMetrics, questionr, recipes, R6, tictoc, varhandle

Suggests grDevices, knitr, rmarkdown, testthat (>= 3.0.2)

VignetteBuilder knitr

RoxygenNote 7.1.1

Encoding UTF-8

NeedsCompilation no

Config/testthat/edition 2

Author David Ruano-Ordás [aut, ctb], Miguel Ferreiro-Díaz [aut, cre], José Ramón Méndez [aut, ctb], University of Vigo [cph]

Maintainer Miguel Ferreiro-Díaz <miguel.ferreiro.diaz@gmail.com>

Repository CRAN

Date/Publication 2021-05-07 09:30:03 UTC

R topics documented:

`Accuracy` *Computes the Accuracy measure.*

Description

Computes the ratio of number of correct predictions to the total number of input samples.

Details

$$Accuracy = (NumberCorrectPredictions)/(TotalNumberofPredictions)$$

Super class

`D2MCS::MeasureFunction` -> `Accuracy`

Methods

Public methods:

- `Accuracy$new()`
- `Accuracy$compute()`
- `Accuracy$clone()`

Method `new()`**:** Method for initializing the object arguments during runtime.

Usage:

```
Accuracy$new(performance.output = NULL)
```

Arguments:

`performance.output` An optional `ConfMatrix` used as basis to compute the performance.

Method `compute()`**:** The function computes the **Accuracy** achieved by the M.L. model.

Usage:

```
Accuracy$compute(performance.output = NULL)
```

Arguments:

`performance.output` An optional `ConfMatrix` parameter to define the type of object used as basis to compute the **Accuracy** measure.

Details: This function is automatically invoke by the `ClassificationOutput` object.

Returns: A numeric vector of size 1 or NULL if an error occurred.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Accuracy$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`MeasureFunction`, `ClassificationOutput`, `ConfMatrix`.

`BinaryPlot` *Plotting feature clusters following bi-class problem.*

Description

The `BinaryPlot` implements a basic plot for bi-class problem.

Super class

`D2MCS::GenericPlot` -> `BinaryPlot`

Methods

Public methods:

- BinaryPlot$new()
- BinaryPlot$plot()
- BinaryPlot$clone()

Method `new()`**:** Empty function used to initialize the object arguments in runtime.

Usage:

```
BinaryPlot$new()
```

Method `plot()`**:** Plots feature-clustering data from a bi-class problem.

Usage:

```
BinaryPlot$plot(summary)
```

Arguments:

`summary` A data.frame comprising the elements to be plotted.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
BinaryPlot$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

GenericPlot

`ChiSquareHeuristic` *Feature-clustering based on ChiSquare method.*

Description

Performs feature-clustering based on ChiSquare method.

Super class

D2MCS::GenericHeuristic -> ChiSquareHeuristic

Methods

Public methods:

- ChiSquareHeuristic$new()
- ChiSquareHeuristic$heuristic()
- ChiSquareHeuristic$clone()

Method `new()`**:** Empty function used to initialize the object arguments in runtime.

Usage:

```
ChiSquareHeuristic$new()
```

Method `heuristic()`**:** Functions responsible of performing the ChiSquare feature-clustering operation.

Usage:

```
ChiSquareHeuristic$heuristic(col1, col2, column.names = NULL)
```

Arguments:

`col1` A numeric vector or matrix required to perform the clustering operation.

`col2` A numeric vector or matrix to perform the clustering operation.

`column.names` An optional character vector with the names of both columns.

Returns: A numeric vector of length 1 or NA if an error occurs.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
ChiSquareHeuristic$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Dataset, chisq.test

`ClassificationOutput` *D2MCS Classification Output.*

Description

Allows computing the classification performance values achieved by D2MCS. The class is automatically created when D2MCS classification method is invoked.

Methods

Public methods:

- ClassificationOutput$new()
- ClassificationOutput$getMetrics()
- ClassificationOutput$getPositiveClass()
- ClassificationOutput$getModelInfo()
- ClassificationOutput$getPerformances()
- ClassificationOutput$savePerformances()
- ClassificationOutput$plotPerformances()
- ClassificationOutput$getPredictions()
- ClassificationOutput$savePredictions()
- ClassificationOutput$clone()

Method `new()`**:** Method for initializing the object arguments during runtime.

Usage:

```
ClassificationOutput$new(voting.schemes, models)
```

Arguments:

`voting.schemes` A list containing the voting schemes used (inherited from `VotingStrategy`.

`models` A list containing the used `Model` during classification stage.

Method `getMetrics()`**:** The function returns the measures used during training stage.

Usage:

```
ClassificationOutput$getMetrics()
```

Returns: A character vector or NULL if training was not performed.

Method `getPositiveClass()`**:** The function gets the name of the positive class used for training/classification.

Usage:

```
ClassificationOutput$getPositiveClass()
```

Returns: A character vector of size 1.

Method `getModelInfo()`**:** The function compiled all the information concerning to the M.L. models used during training/classification.

Usage:

```
ClassificationOutput$getModelInfo(metrics = NULL)
```

Arguments:

`metrics` A character vector defining the metrics used during training/classification.

Returns: A list with the information of each M.L. model.

Method `getPerformances()`**:** The function is used to compute the performance of D2MCS.

Usage:

```
ClassificationOutput$getPerformances(
  test.set,
  measures,
  voting.names = NULL,
  metric.names = NULL,
  cutoff.values = NULL
)
```

Arguments:

`test.set` A Subset object used to compute the performance.

`measures` A character vector with the measures to be used to compute performance value (inherited from MeasureFunction).

`voting.names` A character vector with the name of the voting schemes to analyze the performance. If not defined, all the voting schemes used during classification stage will be taken into account.

`metric.names` A character containing the measures used during training stage. If not defined, all training metrics used during classification will be taken into account.

`cutoff.values` A character vector defining the minimum probability used to perform a a positive classification. If is not defined, all cutoffs used during classification stage will be taken into account.

`dir.path` A character vector with location where the plot will be saved.

Returns: A list of performance values.

Method `savePerformances()`**:** The function is used to save the computed predictions into a CSV file.

Usage:

```
ClassificationOutput$savePerformances(
  dir.path,
  test.set,
  measures,
  voting.names = NULL,
  metric.names = NULL,
  cutoff.values = NULL
)
```

Arguments:

`dir.path` A character vector with location where the plot will be saved.

`test.set` A Subset object used to compute the performance.

`measures` A character vector with the measures to be used to compute performance value (inherited from MeasureFunction).

`voting.names` A character vector with the name of the voting schemes to analyze the performance. If not defined, all the voting schemes used during classification stage will be taken into account.

`metric.names` A character containing the measures used during training stage. If not defined, all training metrics used during classification will be taken into account.

`cutoff.values` A character vector defining the minimum probability used to perform a a positive classification. If is not defined, all cutoffs used during classification stage will be taken into account.

Method `plotPerformances()`**:** The function allows to graphically visualize the computed performance.

Usage:

```
ClassificationOutput$plotPerformances(
  dir.path,
  test.set,
  measures,
  voting.names = NULL,
  metric.names = NULL,
  cutoff.values = NULL
)
```

Arguments:

`dir.path` A character vector with location where the plot will be saved.

`test.set` A `Subset` object used to compute the performance.

`measures` A character vector with the measures to be used to compute performance value (inherited from `MeasureFunction`).

`voting.names` A character vector with the name of the voting schemes to analyze the performance. If not defined, all the voting schemes used during classification stage will be taken into account.

`metric.names` A character containing the measures used during training stage. If not defined, all training metrics used during classification will be taken into account.

`cutoff.values` A character vector defining the minimum probability used to perform a positive classification. If is not defined, all cutoffs used during classification stage will be taken into account.

Method `getPredictions()`**:** The function is used to obtain the computed predictions.

Usage:

```
ClassificationOutput$getPredictions(
  voting.names = NULL,
  metric.names = NULL,
  cutoff.values = NULL,
  type = NULL,
  target = NULL,
  filter = FALSE
)
```

Arguments:

`voting.names` A character vector with the name of the voting schemes to analyze the performance. If not defined, all the voting schemes used during classification stage will be taken into account.

`metric.names` A character containing the measures used during training stage. If not defined, all training metrics used during classification will be taken into account.

`cutoff.values` A character vector defining the minimum probability used to perform a a positive classification. If is not defined, all cutoffs used during classification stage will be taken into account.

`type` A character to define which type of predictions should be returned. If not defined all type of probabilities will be returned. Conversely if "prob" or "raw" is defined then computed 'probabilistic' or 'class' values are returned.

`target` A character defining the value of the positive class.

`filter` A logical value used to specify if only predictions matching the target value should be returned or not. If TRUE the function returns only the predictions matching the target value. Conversely if FALSE (by default) the function returns all the predictions.

Returns: A `PredictionOutput` object.

Method `savePredictions()`**:** The function saves the predictions into a CSV file.

Usage:

```
ClassificationOutput$savePredictions(
  dir.path,
  voting.names = NULL,
  metric.names = NULL,
  cutoff.values = NULL,
  type = NULL,
  target = NULL,
  filter = FALSE
)
```

Arguments:

`dir.path` A character vector with location defining the location of the CSV file.

`voting.names` A character vector with the name of the voting schemes to analyze the performance. If not defined, all the voting schemes used during classification stage will be taken into account.

`metric.names` A character containing the measures used during training stage. If not defined, all training metrics used during classification will be taken into account.

`cutoff.values` A character vector defining the minimum probability used to perform a positive classification. If is not defined, all cutoffs used during classification stage will be taken into account.

`type` A character to define which type of predictions should be returned. If not defined all type of probabilities will be returned. Conversely if "prob" or "raw" is defined then computed 'probabilistic' or 'class' values are returned.

`target` A character defining the value of the positive class.

`filter` A logical value used to specify if only predictions matching the target value should be returned or not. If TRUE the function returns only the predictions matching the target value. Conversely if FALSE (by default) the function returns all the predictions.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
ClassificationOutput$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`D2MCS`

ClassMajorityVoting *Implementation of Majority Voting voting.*

Description

Implementation of the parliamentary 'majority voting' procedure. The majority class value is defined as final class. All class values have the same importance.

Super class

D2MCS::SimpleVoting -> ClassMajorityVoting

Methods

Public methods:

- ClassMajorityVoting$new()
- ClassMajorityVoting$getMajorityClass()
- ClassMajorityVoting$getClassTie()
- ClassMajorityVoting$execute()
- ClassMajorityVoting$clone()

Method `new()`**:** Method for initializing the object arguments during runtime.

Usage:

```
ClassMajorityVoting$new(cutoff = 0.5, class.tie = NULL, majority.class = NULL)
```

Arguments:

`cutoff` A character vector defining the minimum probability used to perform a positive classification. If is not defined, 0.5 will be used as default value.

`class.tie` A character used to define the target class value used when a tie is found. If NULL positive class value will be assigned.

`majority.class` A character defining the value of the majority class. If NULL will be used same value as training stage.

Method `getMajorityClass()`**:** The function returns the value of the majority class.

Usage:

```
ClassMajorityVoting$getMajorityClass()
```

Returns: A character vector of length 1 with the name of the majority class.

Method `getClassTie()`**:** The function gets the class value assigned to solve ties.

Usage:

```
ClassMajorityVoting$getClassTie()
```

Returns: A character vector of length 1.

Method `execute()`**:** The function implements the majority voting procedure.

Usage:

```
ClassMajorityVoting$execute(predictions, verbose = FALSE)
```

Arguments:

`predictions` A `ClusterPredictions` object containing all the predictions achieved for each cluster.

`verbose` A logical value to specify if more verbosity is needed.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
ClassMajorityVoting$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`D2MCS`, `ClassMajorityVoting`, `ClassWeightedVoting`, `ProbAverageVoting`, `ProbAverageWeightedVoting`, `ProbBasedMethodology`

`ClassWeightedVoting` *Implementation Weighted Voting scheme.*

Description

A new implementation of `ClassMajorityVoting` where each class value has different values (weights).

Super class

`D2MCS::SimpleVoting` -> `ClassWeightedVoting`

Methods

Public methods:

- `ClassWeightedVoting$new()`
- `ClassWeightedVoting$getWeights()`
- `ClassWeightedVoting$setWeights()`
- `ClassWeightedVoting$execute()`
- `ClassWeightedVoting$clone()`

Method `new()`**:** Method for initializing the object arguments during runtime.

Usage:

```
ClassWeightedVoting$new(cutoff = 0.5, weights = NULL)
```

Arguments:

`cutoff` A character vector defining the minimum probability used to perform a positive classification. If is not defined, 0.5 will be used as default value.

`weights` A numeric vector with the weights of each cluster. If NULL performance achieved during training will be used as default.

Method `getWeights()`**:** The function returns the weights used to perform the voting scheme.

Usage:

```
ClassWeightedVoting$getWeights()
```

Returns: A numeric vector.

Method `setWeights()`**:** The function allows changing the value of the weights.

Usage:

```
ClassWeightedVoting$setWeights(weights)
```

Arguments:

`weights` A numeric vector containing the new weights.

Method `execute()`**:** The function implements the cluster-weighted majority voting procedure.

Usage:

```
ClassWeightedVoting$execute(predictions, verbose = FALSE)
```

Arguments:

`predictions` A `ClusterPredictions` object containing all the predictions achieved for each cluster.

`verbose` A logical value to specify if more verbosity is needed.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
ClassWeightedVoting$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`D2MCS`, `ClassMajorityVoting`, `ClassWeightedVoting`, `ProbAverageVoting`, `ProbAverageWeightedVoting`, `ProbBasedMethodology`

`ClusterPredictions` *Manages the predictions achieved on a cluster.*

Description

Stores the predictions achieved by the best M.L. of each cluster.

Methods

Public methods:

- ClusterPredictions$new()
- ClusterPredictions$add()
- ClusterPredictions$get()
- ClusterPredictions$getAll()
- ClusterPredictions$size()
- ClusterPredictions$getPositiveClass()
- ClusterPredictions$getClassValues()
- ClusterPredictions$clone()

Method `new()`**:** Method for initializing the object arguments during runtime.

Usage:

```
ClusterPredictions$new(class.values, positive.class)
```

Arguments:

`class.values` A character vector containing the values of the target class.

`positive.class` A character with the value of the positive class.

Method `add()`**:** The function is used to add the prediction achieved by a specific M.L. model.

Usage:

```
ClusterPredictions$add(prediction)
```

Arguments:

`prediction` A `Prediction` object containing the computed predictions.

Method `get()`**:** The function returns the predictions placed at specific position.

Usage:

```
ClusterPredictions$get(position)
```

Arguments:

`position` A numeric value indicating the position of the predictions to be obtained.

Returns: A `Prediction` object.

Method `getAll()`**:** The function returns all the predictions.

Usage:

```
ClusterPredictions$getAll()
```

Returns: A list containing all computed predictions.

Method `size()`**:** The function returns the number of computed predictions.

Usage:

```
ClusterPredictions$size()
```

Returns: A numeric value.

Method `getPositiveClass()`**:** The function gets the value of the positive class.

Usage:

```
ClusterPredictions$getPositiveClass()
```

Returns: A character vector of size 1.

Method `getClassValues()`**:** The function returns all the values of the target class.

Usage:

```
ClusterPredictions$getClassValues()
```

Returns: A character vector containing all target values.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
ClusterPredictions$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`D2MCS`, `ClassificationOutput`, `Prediction`

`CombinedMetrics` *Abstract class to compute the class prediction based on combination between metrics.*

Description

Abstract class used as a template to define new customized strategies to combine the class predictions made by different metrics.

Methods

Public methods:

- `CombinedMetrics$new()`
- `CombinedMetrics$getRequiredMetrics()`
- `CombinedMetrics$getFinalPrediction()`
- `CombinedMetrics$clone()`

Method `new()`**:** Method for initializing the object arguments during runtime.

Usage:

```
CombinedMetrics$new(required.metrics)
```

Arguments:

`required.metrics` A character vector of length greater than 2 with the name of the required metrics.

Method `getRequiredMetrics()`**:** The function returns the required metrics that will participate in the combined metric process.

Usage:

```
CombinedMetrics$getRequiredMetrics()
```

Returns: A character vector of length greater than 2 with the name of the required metrics.

Method `getFinalPrediction()`**:** Function used to implement the strategy to obtain the final prediction based on different metrics.

Usage:

```
CombinedMetrics$getFinalPrediction(
  raw.pred,
  prob.pred,
  positive.class,
  negative.class
)
```

Arguments:

`raw.pred` A character list of length greater than 2 with the class value of the predictions made by the metrics.

`prob.pred` A numeric list of length greater than 2 with the probability of the predictions made by the metrics.

`positive.class` A character with the value of the positive class.

`negative.class` A character with the value of the negative class.

Returns: A logical value indicating if the instance is predicted as positive class or not.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
CombinedMetrics$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`CombinedVoting`

`CombinedVoting` *Implementation of Combined Voting.*

Description

Calculates the final prediction by performing the result of the predictions of different metrics obtained through a `SimpleVoting` class.

Super class

`D2MCS::VotingStrategy` -> `CombinedVoting`

Methods

Public methods:

- CombinedVoting$new()
- CombinedVoting$getCombinedMetrics()
- CombinedVoting$getMethodology()
- CombinedVoting$getFinalPred()
- CombinedVoting$execute()
- CombinedVoting$clone()

Method `new()`**:** Method for initializing the object arguments during runtime.

Usage:

```
CombinedVoting$new(voting.schemes, combined.metrics, methodology, metrics)
```

Arguments:

`voting.schemes` A list of elements inherited from `SimpleVoting`.

`combined.metrics` An object defining the metrics used to combine the voting schemes. The object must inherit from `CombinedMetrics` class.

`methodology` An object specifying the methodology used to execute the combined voting. Object inherited from `Methodology` object

`metrics` A character vector with the name of the metrics used to perform the combined voting operations. Metrics should be previously defined during training stage.

Method `getCombinedMetrics()`**:** The function returns the metrics used to combine the metrics results.

Usage:

```
CombinedVoting$getCombinedMetrics()
```

Returns: An object inherited from `CombinedMetrics` class.

Method `getMethodology()`**:** The function gets the methodology used to execute the combined votings.

Usage:

```
CombinedVoting$getMethodology()
```

Returns: An object inherited from `Methodology` class.

Method `getFinalPred()`**:** The function returns the predictions obtained after executing the combined-voting methodology.

Usage:

```
CombinedVoting$getFinalPred(type = NULL, target = NULL, filter = NULL)
```

Arguments:

`type` A character to define which type of predictions should be returned. If not defined all type of probabilities will be returned. Conversely if "prob" or "raw" is defined then computed 'probabilistic' or 'class' values are returned.

`target` A character defining the value of the positive class.

`filter` A logical value used to specify if only predictions matching the target value should be returned or not. If TRUE the function returns only the predictions matching the target value. Conversely if FALSE (by default) the function returns all the predictions.

Returns: A data.frame with the computed predictions.

Method `execute()`**:** The function implements the combined voting scheme.

Usage:

```
CombinedVoting$execute(predictions, verbose = FALSE)
```

Arguments:

`predictions` A `ClusterPredictions` object containing the predictions computed for each cluster.

`verbose` A logical value to specify if more verbosity is needed.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
CombinedVoting$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`D2MCS`, `ClassMajorityVoting`, `ClassWeightedVoting`, `ProbAverageVoting`, `ProbAverageWeightedVoting`, `ProbBasedMethodology`, `SimpleVoting`

`ConfMatrix` *Confusion matrix wrapper.*

Description

Creates a `R6` confusion matrix from the `confusionMatrix` caret package.

Methods

Public methods:

- `ConfMatrix$new()`
- `ConfMatrix$getConfusionMatrix()`
- `ConfMatrix$getTP()`
- `ConfMatrix$getTN()`
- `ConfMatrix$getFN()`
- `ConfMatrix$getFP()`
- `ConfMatrix$clone()`

Method `new()`**:** Method to create a confusion matrix object from a `caret` `confusionMatrix`

Usage:

```
ConfMatrix$new(confMatrix)
```

Arguments:

`confMatrix` A `caret` confusionMatrix argument.

Method `getConfusionMatrix()`**:** The function obtains the `confusionMatrix` following the same structured as defined in the `caret` package

Usage:

```
ConfMatrix$getConfusionMatrix()
```

Returns: A `confusionMatrix` object.

Method `getTP()`**:** The function is used to compute the number of True Positive values achieved.

Usage:

```
ConfMatrix$getTP()
```

Returns: A numeric vector of size 1.

Method `getTN()`**:** The function computes the True Negative values.

Usage:

```
ConfMatrix$getTN()
```

Returns: A numeric vector of size 1.

Method `getFN()`**:** The function returns the number of Type II errors (False Negative).

Usage:

```
ConfMatrix$getFN()
```

Returns: A numeric vector of size 1.

Method `getFP()`**:** The function returns the number of Type I errors (False Negative).

Usage:

```
ConfMatrix$getFP()
```

Returns: A numeric vector of size 1.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
ConfMatrix$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`D2MCS`, `MeasureFunction`, `ClassificationOutput`

D2MCS *Data Driven Multiple Classifier System.*

Description

The class is responsible of managing the whole process. Concretely builds the M.L. models (optimizes models hyperparameters), selects the best M.L. model for each cluster and executes the classification stage.

Methods

Public methods:

- `D2MCS$new()`
- `D2MCS$train()`
- `D2MCS$classify()`
- `D2MCS$getAvailableModels()`
- `D2MCS$clone()`

Method `new()`**:** The function is used to initialize all parameters needed to build a Multiple Classifier System.

Usage:

```
D2MCS$new(
  dir.path,
  num.cores = NULL,
  socket.type = "PSOCK",
  outfile = NULL,
  serialize = FALSE
)
```

Arguments:

`dir.path` A character defining location were the trained models should be saved.

`num.cores` An optional numeric value specifying the number of CPU cores used for training the models (only if parallelization is allowed). If not defined (num.cores - 2) cores will be used.

`socket.type` A character value defining the type of socket used to communicate the workers. The default type, `"PSOCK"`, calls makePSOCKcluster. Type `"FORK"` calls makeForkCluster. For more information see `makeCluster`

`outfile` Where to direct the stdout and stderr connection output from the workers. "" indicates no redirection (which may only be useful for workers on the local machine). Defaults to '/dev/null'

`serialize` A `logical` value. If TRUE (default) serialization will use XDR: where large amounts of data are to be transferred and all the nodes are little-endian, communication may be substantially faster if this is set to false.

Method `train()`**:** The function is responsible of performing the M.L. model training stage.

Usage:

```
D2MCS$train(
  train.set,
  train.function,
  num.clusters = NULL,
  model.recipe = DefaultModelFit$new(),
  ex.classifiers = c(),
  ig.classifiers = c(),
  metrics = NULL,
  saveAllModels = FALSE
)
```

Arguments:

`train.set` A `Trainset` object used as training input for the M.L. models

`train.function` A `TrainFunction` defining the training configuration options.

`num.clusters` An numeric value used to define the number of clusters from the `Trainset` that should be utilized during the training stage. If not defined all clusters will we taken into account for training.

`model.recipe` An unprepared recipe object inherited from `GenericModelFit` class.

`ex.classifiers` A character vector containing the name of the M.L. models used in training stage. See `getModelInfo` and `https://topepo.github.io/caret/available-models.html` for more information about all the available models.

`ig.classifiers` A character vector containing the name of the M.L. that should be ignored when performing the training stage. See `getModelInfo` and `https://topepo.github.io/caret/available-models.html` for more information about all the available models.

`metrics` A character vector containing the metrics used to perform the M.L. model hyperparameter optimization during the training stage. See `SummaryFunction`, `UseProbability` and `NoProbability` for more information.

`saveAllModels` A logical parameter. A TRUE saves all trained models while A FALSE saves only the M.L. model achieving the best performance on each cluster.

Returns: A `TrainOutput` object containing all the information computed during the training stage.

Method `classify()`**:** The function is responsible for executing the classification stage.

Usage:

```
D2MCS$classify(train.output, subset, voting.types, positive.class = NULL)
```

Arguments:

`train.output` The `TrainOutput` object computed in the train stage.

`subset` A `Subset` containing the data to be classified.

`voting.types` A list containing `SingleVoting` or `CombinedVoting` objects.

`positive.class` An optional character parameter used to define the positive class value.

Returns: A `ClassificationOutput` with all the values computed during classification stage.

Method `getAvailableModels()`**:** The function obtains all the available M.L. models.

Usage:

```
D2MCS$getAvailableModels()
```

Returns: A data.frame containing the information of the available M.L. models.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
D2MCS$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Dataset, Subset, Trainset

Examples

```
# Specify the random number generation
set.seed(1234)

## Create Dataset Handler object.
loader <- DatasetLoader$new()

## Load 'hcc-data-complete-balanced.csv' dataset file.
data <- loader$load(filepath = system.file(file.path("examples",
                                                     "hcc-data-complete-balanced.csv"),
                                           package = "D2MCS"),
                    header = TRUE, normalize.names = TRUE)
## Get column names
data$getColumnNames()

## Split data into 4 partitions keeping balance ratio of 'Class' column.
data$createPartitions(num.folds = 4, class.balance = "Class")

## Create a subset comprising the first 2 partitions for clustering purposes.
cluster.subset <- data$createSubset(num.folds = c(1, 2), class.index = "Class",
                                    positive.class = "1")

## Create a subset comprising second and third partitions for trainning purposes.
train.subset <- data$createSubset(num.folds = c(2, 3), class.index = "Class",
                                  positive.class = "1")

## Create a subset comprising last partitions for testing purposes.
test.subset <- data$createSubset(num.folds = 4, class.index = "Class",
                                 positive.class = "1")

## Distribute the features into clusters using MCC heuristic.
distribution <- SimpleStrategy$new(subset = cluster.subset,
                                   heuristic = MCCHeuristic$new())
distribution$execute()

## Get the best achieved distribution
```

```
distribution$getBestClusterDistribution()

## Create a train set from the computed clustering distribution
train.set <- distribution$createTrain(subset = train.subset)

## Not run:

## Initialization of D2MCS configuration parameters.
##  - Defining training operation.
##    + 10-fold cross-validation
##    + Use only 1 CPU core.
##    + Seed was set to ensure straightforward reproductivity of experiments.
trFunction <- TwoClass$new(method = "cv", number = 10, savePredictions = "final",
                           classProbs = TRUE, allowParallel = TRUE,
                           verboseIter = FALSE, seed = 1234)

#' ## - Specify the models to be trained
ex.classifiers <- c("ranger", "lda", "lda2")

## Initialize D2MCS
#' d2mcs <- D2MCS$new(dir.path = tempdir(),
                      num.cores = 1)

## Execute training stage for using 'MCC' and 'PPV' measures to optimize model hyperparameters.
trained.models <- d2mcs$train(train.set = train.set,
                              train.function = trFunction,
                              ex.classifiers = ex.classifiers,
                              metrics = c("MCC", "PPV"))

## Execute classification stage using two different voting schemes
predictions <- d2mcs$classify(train.output = trained.models,
                              subset = test.subset,
                              voting.types = c(
                            SingleVoting$new(voting.schemes = c(ClassMajorityVoting$new(),
                                                                ClassWeightedVoting$new()),
                                                    metrics = c("MCC", "PPV"))))

## Compute the performance of each voting scheme using PPV and MMC measures.
predictions$getPerformances(test.subset, measures = list(MCC$new(), PPV$new()))

## Execute classification stage using multiple voting schemes (simple and combined)
predictions <- d2mcs$classify(train.output = trained.models,
                              subset = test.subset,
                              voting.types = c(
                            SingleVoting$new(voting.schemes = c(ClassMajorityVoting$new(),
                                                                ClassWeightedVoting$new()),
                                                     metrics = c("MCC", "PPV")),
                            CombinedVoting$new(voting.schemes = ClassMajorityVoting$new(),
                                                          combined.metrics = MinimizeFP$new(),
                                                       methodology = ProbBasedMethodology$new(),
                                                               metrics = c("MCC", "PPV"))))

## Compute the performance of each voting scheme using PPV and MMC measures.
```

```
predictions$getPerformances(test.subset, measures = list(MCC$new(), PPV$new()))

## End(Not run)
```

`Dataset` *Simple Dataset handler.*

Description

Creates a valid simple dataset object.

Methods

Public methods:

- `Dataset$new()`
- `Dataset$getColumnNames()`
- `Dataset$getDataset()`
- `Dataset$getNcol()`
- `Dataset$getNrow()`
- `Dataset$getRemovedColumns()`
- `Dataset$cleanData()`
- `Dataset$removeColumns()`
- `Dataset$createPartitions()`
- `Dataset$createSubset()`
- `Dataset$createTrain()`

Method `new()`**:** Method for initializing the object arguments during runtime.

Usage:

```
Dataset$new(
  filepath,
  header = TRUE,
  sep = ",",
  skip = 0,
  normalize.names = FALSE,
  string.as.factor = FALSE,
  ignore.columns = NULL
)
```

Arguments:

`filepath` The name of the file which the data are to be read from. Each row of the table appears as one line of the file. If it does not contain an _absolute_ path, the file name is _relative_ to the current working directory, '`getwd()`'.

`header` A logical value indicating whether the file contains the names of the variables as its first line. If missing, the value is determined from the file format: 'header' is set to 'TRUE' if and only if the first row contains one fewer field than the number of columns.

`sep` The field separator character. Values on each line of the file are separated by this character.

`skip` Defines the number of header lines should be skipped.

`normalize.names` A logical value indicating whether the columns names should be automatically renamed to ensure R compatibility.

`string.as.factor` A logical value indicating if character columns should be converted to factors (`default = FALSE`).

`ignore.columns` Specify the columns from the input file that should be ignored.

Method `getColumnNames()`**:** Get the name of the columns comprising the dataset.

Usage:

```
Dataset$getColumnNames()
```

Returns: A character vector with the name of each column.

Method `getDataset()`**:** Gets the full dataset.

Usage:

```
Dataset$getDataset()
```

Returns: A data.frame with all the loaded information.

Method `getNcol()`**:** Obtains the number of columns present in the dataset.

Usage:

```
Dataset$getNcol()
```

Returns: An integer of length 1 or NULL

Method `getNrow()`**:** Obtains the number of rows present in the dataset.

Usage:

```
Dataset$getNrow()
```

Returns: An integer of length 1 or NULL

Method `getRemovedColumns()`**:** Get the columns removed or ignored.

Usage:

```
Dataset$getRemovedColumns()
```

Returns: A list containing the name of the removed columns.

Method `cleanData()`**:** Removes data.frame columns matching some criterion.

Usage:

```
Dataset$cleanData(remove.funcs = NULL, remove.na = TRUE, remove.const = FALSE)
```

Arguments:

`remove.funcs` A vector of functions use to define which columns must be removed.

`remove.na` A logical value indicating whether NA values should be removed.

`remove.const` A logical value used to indicate if constant values should be removed.

Method `removeColumns()`**:** Applies `cleanData` function over an specific set of columns.

Usage:

```
Dataset$removeColumns(
  columns,
  remove.funcs = NULL,
  remove.na = FALSE,
  remove.const = FALSE
)
```

Arguments:

`columns` Set of columns (numeric or character) where removal operation should be applied.

`remove.funcs` A vector of functions use to define which columns must be removed.

`remove.na` A logical value indicating whether `NA` values should be removed.

`remove.const` A logical value used to indicate if constant values should be removed.

Method `createPartitions()`**:** Creates a k-folds partition from the initial dataset.

Usage:

```
Dataset$createPartitions(
  num.folds = NULL,
  percent.folds = NULL,
  class.balance = NULL
)
```

Arguments:

`num.folds` A numeric for the number of folds (partitions)

`percent.folds` A numeric vector with the percentage of instances containing each fold.

`class.balance` A logical value indicating if class balance should be kept.

Method `createSubset()`**:** Creates a `Subset` for testing or classification purposes. A target class should be provided for testing purposes.

Usage:

```
Dataset$createSubset(
  num.folds = NULL,
  opts = list(remove.na = TRUE, remove.const = FALSE),
  class.index = NULL,
  positive.class = NULL
)
```

Arguments:

`num.folds` A numeric defining the number of folds that should we used to build the Subset.

`opts` A list with optional parameters. Valid arguments are `remove.na` (removes columns with NA values) and `remove.const` (ignore columns with constant values).

`class.index` A numeric value identifying the column representing the target class

`positive.class` Defines the positive class value.

Returns: A Subset object.

Method `createTrain()`**:** Creates a set for training purposes. A class should be defined to guarantee full-compatibility with supervised models.

Usage:

```
Dataset$createTrain(
  class.index,
  positive.class,
  num.folds = NULL,
  opts = list(remove.na = TRUE, remove.const = FALSE)
)
```

Arguments:

`class.index` A numeric value identifying the column representing the target class

`positive.class` Defines the positive class value.

`num.folds` A numeric defining the number of folds that should we used to build the `Subset`.

`opts` A list with optional parameters. Valid arguments are `remove.na` (removes columns with NA values) and `remove.const` (ignore columns with constant values).

Returns: A `Trainset` object.

See Also

`HDDataset`

`DatasetLoader` *Dataset creation.*

Description

Wrapper class able to automatically create a `Dataset`, `HDDataset` according to the input data.

Methods

Public methods:

- `DatasetLoader$new()`
- `DatasetLoader$load()`

Method `new()`**:** Empty function used to initialize the object arguments in runtime.

Usage:

```
DatasetLoader$new()
```

Method `load()`**:** Stores the input source into a `Dataset` or `HDDataset` type object.

Usage:

```
DatasetLoader$load(
  filepath,
  header = TRUE,
  sep = ",",
  skip.lines = 0,
  normalize.names = FALSE,
  string.as.factor = FALSE,
  ignore.columns = NULL
)
```

Arguments:

`filepath` The name of the file which the data are to be read from. Each row of the table appears as one line of the file. If it does not contain an _absolute_ path, the file name is _relative_ to the current working directory, 'getwd()'.

`header` A logical value indicating whether the file contains the names of the variables as its first line. If missing, the value is determined from the file format: 'header' is set to 'TRUE' if and only if the first row contains one fewer field than the number of columns.

`sep` The field separator character. Values on each line of the file are separated by this character.

`skip.lines` Defines the number of header lines should be skipped.

`normalize.names` A logical value indicating whether the columns names should be automatically renamed to ensure R compatibility.

`string.as.factor` A logical value indicating if character columns should be converted to factors (default = FALSE).

`ignore.columns` Specify the columns from the input file that should be ignored.

Returns: A `Dataset` or `HDDataset` object.

See Also

`Dataset`, `HDDataset`

Examples

```
## Not run:
# Create Dataset Handler object.
  loader <- DatasetLoader$new()

  # Load input file.
  data <- loader$load(filepath = system.file(file.path("examples",
                                                       "hcc-data-complete-balanced.csv"),
                                             package = "D2MCS"),
                      header = T, normalize.names = T)

## End(Not run)
```

`DefaultModelFit` *Default model fitting implementation.*

Description

Creates a default `recipe` and `formula` objects used in model training stage.

Super class

`D2MCS::GenericModelFit` -> `DefaultModelFit`

Methods

Public methods:

- DefaultModelFit$new()
- DefaultModelFit$createFormula()
- DefaultModelFit$createRecipe()
- DefaultModelFit$clone()

Method `new()`**:** Method for initializing the object arguments during runtime.

Usage:

```
DefaultModelFit$new()
```

Method `createFormula()`**:** The function is responsible of creating a `formula` for M.L. model.

Usage:

```
DefaultModelFit$createFormula(instances, class.name, simplify = FALSE)
```

Arguments:

`instances` A data.frame containing the instances used to create the recipe.

`class.name` A character vector representing the name of the target class.

`simplify` A logical argument defining whether the formula should be generated as simple as possible.

Returns: A `formula` object.

Method `createRecipe()`**:** The function is responsible of creating a `recipe` with five operations over the data: `step_zv`, `step_nzv`, `step_corr`, `step_center`, `step_scale`

Usage:

```
DefaultModelFit$createRecipe(instances, class.name)
```

Arguments:

`instances` A `data.frame` containing the instances used to create the recipe.

`class.name` A `character` vector representing the name of the target class.

Details: This function is automatically invoked by `D2MCS` during model training stage.

Returns: An object of class `recipe`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
DefaultModelFit$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`GenericModelFit`, `train`

DependencyBasedStrategy

Clustering strategy based on dependency between features.

Description

Features are distributed according to their independence values. This strategy is divided into two steps. The first phase focuses on forming groups with those features most dependent on each other. This step also identifies those that are independent from all the others in the group. The second step is to try out different numbers of clusters until you find the one you think is best. These clusters are formed by inserting in all the independent characteristics identified previously and trying to distribute the features of the groups formed in the previous step in separate clusters. In this way, it seeks to ensure that the features are as independent as possible from those found in the same cluster.

Details

The strategy is suitable only for binary and real features. Other features are automatically grouped into a specific cluster named as 'unclustered'. This class requires the `StrategyConfiguration` type object implements the following methods:

- `getBinaryCutoff()`: The function is used to define the interval to consider the dependency between binary features.

- `getRealCutoff()`: The function allows defining the cutoff to consider the dependency between real features.

- `tiebreak(feature,clus.candidates,fea.dep.dist.clus,corpus,heuristic,class,class.name)`: The function solves the ties between two (or more) features.

- `qualityOfCluster(clusters,metrics)`: The function determines the quality of a cluster

- `isImprovingClustering(clusters.deltha)`: The function indicates if clustering is getting better as the number of them increases.

An example of implementation with the description of each parameter is the `DependencyBasedStrategyConfiguration` class.

Super class

`D2MCS::GenericClusteringStrategy` -> `DependencyBasedStrategy`

Methods

Public methods:

- `DependencyBasedStrategy$new()`
- `DependencyBasedStrategy$execute()`
- `DependencyBasedStrategy$getDistribution()`
- `DependencyBasedStrategy$createTrain()`
- `DependencyBasedStrategy$plot()`
- `DependencyBasedStrategy$saveCSV()`

- DependencyBasedStrategy$clone()

Method `new()`**:** Method for initializing the object parameters during runtime.

Usage:

```
DependencyBasedStrategy$new(
  subset,
  heuristic,
  configuration = DependencyBasedStrategyConfiguration$new()
)
```

Arguments:

`subset` The Subset used to apply the feature-clustering strategy.

`heuristic` The heuristic used to compute the relevance of each feature. Must inherit from GenericHeuristic abstract class.

`configuration` optional parameter to customize configuration parameters for the strategy. Must inherited from StrategyConfiguration abstract class.

Method `execute()`**:** Function responsible of performing the dependency-based feature clustering strategy over the defined Subset.

Usage:

```
DependencyBasedStrategy$execute(verbose = TRUE)
```

Arguments:

`verbose` A logical value to specify if more verbosity is needed.

Method `getDistribution()`**:** Function used to obtain a specific cluster distribution.

Usage:

```
DependencyBasedStrategy$getDistribution(
  num.clusters = NULL,
  num.groups = NULL,
  include.unclustered = FALSE
)
```

Arguments:

`num.clusters` A numeric value to select the number of clusters (define the distribution).

`num.groups` A single or numeric vector value to identify a specific group that forms the clustering distribution.

`include.unclustered` A logical value to determine if unclustered features should be included.

Returns: A list with the features comprising an specific clustering distribution.

Method `createTrain()`**:** The function is used to create a Trainset object from a specific clustering distribution.

Usage:

```
DependencyBasedStrategy$createTrain(
  subset,
  num.clusters = NULL,
  num.groups = NULL,
  include.unclustered = FALSE
)
```

Arguments:

`subset` The `Subset` object used as a basis to create the train set (see `Trainset` class).

`num.clusters` A numeric value to select the number of clusters (define the distribution).

`num.groups` A single or numeric vector value to identify a specific group that forms the clustering distribution.

`include.unclustered` A logical value to determine if unclustered features should be included.

Details: If `num.clusters` and `num.groups` are not defined, best clustering distribution is used to create the train set.

Method `plot()`**:** The function is responsible for creating a plot to visualize the clustering distribution.

Usage:

```
DependencyBasedStrategy$plot(dir.path = NULL, file.name = NULL)
```

Arguments:

`dir.path` An optional argument to define the name of the directory where the exported plot will be saved. If not defined, the file path will be automatically assigned to the current working directory, '`getwd()`'.

`file.name` A character to define the name of the PDF file where the plot is exported.

Method `saveCSV()`**:** The function is used to save the clustering distribution to a CSV file.

Usage:

```
DependencyBasedStrategy$saveCSV(
  dir.path = NULL,
  name = NULL,
  num.clusters = NULL
)
```

Arguments:

`dir.path` The name of the directory to save the CSV file.

`name` Defines the name of the CSV file.

`num.clusters` An optional parameter to select the number of clusters to be saved. If not defined, all cluster distributions will be saved.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
DependencyBasedStrategy$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`GenericClusteringStrategy`, `StrategyConfiguration`, `DependencyBasedStrategyConfiguration`

DependencyBasedStrategyConfiguration *Custom Strategy Configuration handler for the DependencyBasedStrategy strategy.*

Description

Define the default configuration parameters for the DependencyBasedStrategy strategy.

Super class

D2MCS::StrategyConfiguration -> DependencyBasedStrategyConfiguration

Methods

Public methods:

- DependencyBasedStrategyConfiguration$new()
- DependencyBasedStrategyConfiguration$minNumClusters()
- DependencyBasedStrategyConfiguration$maxNumClusters()
- DependencyBasedStrategyConfiguration$getBinaryCutoff()
- DependencyBasedStrategyConfiguration$getRealCutoff()
- DependencyBasedStrategyConfiguration$setBinaryCutoff()
- DependencyBasedStrategyConfiguration$setRealCutoff()
- DependencyBasedStrategyConfiguration$tiebreak()
- DependencyBasedStrategyConfiguration$qualityOfCluster()
- DependencyBasedStrategyConfiguration$isImprovingClustering()
- DependencyBasedStrategyConfiguration$clone()

Method `new()`**:** Method for initializing the object arguments during runtime.

Usage:

```
DependencyBasedStrategyConfiguration$new(
  binaryCutoff = 0.6,
  realCutoff = 0.6,
  tiebreakMethod = "lfdc",
  metric = "dep.tar"
)
```

Arguments:

`binaryCutoff` The numeric value of binary cutoff.

`realCutoff` The numeric value of real cutoff.

`tiebreakMethod` The character value of tie-break method. The two tiebreak methods available are "lfdc" (less dependence cluster with the features) and "ltdc" (less dependence cluster with the target). These methods are used to add the features in the candidate feature clusters.

`metric` The character value of the metric to apply the mean to obtain the quality of a cluster. The two metrics available are "dep.tar" (Dependence of cluster features on the target) and "dep.fea" (Dependence between cluster features).

Method `minNumClusters()`**:** Function used to return the minimum number of clusters distributions used. By default the minimum is set in 2.

Usage:

```
DependencyBasedStrategyConfiguration$minNumClusters(...)
```

Arguments:

`...` Further arguments passed down to `minNumClusters` function.

Returns: A numeric vector of length 1.

Method `maxNumClusters()`**:** The function is responsible of returning the maximum number of cluster distributions used. By default the maximum number is set in 50.

Usage:

```
DependencyBasedStrategyConfiguration$maxNumClusters(...)
```

Arguments:

`...` Further arguments passed down to `maxNumClusters` function.

Returns: A numeric vector of length 1.

Method `getBinaryCutoff()`**:** Gets the cutoff to consider the dependency between binary features.

Usage:

```
DependencyBasedStrategyConfiguration$getBinaryCutoff()
```

Returns: The numeric value of binary cutoff.

Method `getRealCutoff()`**:** Gets the cutoff to consider the dependency between real features.

Usage:

```
DependencyBasedStrategyConfiguration$getRealCutoff()
```

Returns: The numeric value of real cutoff.

Method `setBinaryCutoff()`**:** Sets the cutoff to consider the dependency between binary features.

Usage:

```
DependencyBasedStrategyConfiguration$setBinaryCutoff(cutoff)
```

Arguments:

`cutoff` The new numeric value of binary cutoff.

Method `setRealCutoff()`**:** Sets the cutoff to consider the dependency between real features.

Usage:

```
DependencyBasedStrategyConfiguration$setRealCutoff(cutoff)
```

Arguments:

`cutoff` The new numeric value of real cutoff.

Method `tiebreak()`**:** The function solves the ties between two (or more) features.

Usage:

```
DependencyBasedStrategyConfiguration$tiebreak(
  feature,
  clus.candidates,
  fea.dep.dist.clus,
  corpus,
  heuristic,
  class,
  class.name
)
```

Arguments:

`feature` A character containing the name of the feature

`clus.candidates` A single or numeric vector value to identify the candidate groups to insert the feature.

`fea.dep.dist.clus` A list containing the groups chosen for the features.

`corpus` A data.frame containing the features of the initial data.

`heuristic` The heuristic used to compute the relevance of each feature. Must inherit from GenericHeuristic abstract class.

`class` A character vector containing all the values of the target class.

`class.name` A character value representing the name of the target class.

Method `qualityOfCluster()`**:** The function determines the quality of a cluster.

Usage:

```
DependencyBasedStrategyConfiguration$qualityOfCluster(clusters, metrics)
```

Arguments:

`clusters` A list with the feature distribution of each cluster.

`metrics` A numeric list with the metrics associated to the cluster (dependency between all features and dependency between the features and the class).

Returns: A numeric vector of length 1.

Method `isImprovingClustering()`**:** The function indicates if clustering is getting better as the number of them increases.

Usage:

```
DependencyBasedStrategyConfiguration$isImprovingClustering(clusters.deltha)
```

Arguments:

`clusters.deltha` A numeric vector value with the quality values of the built clusters.

Returns: A numeric vector of length 1.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
DependencyBasedStrategyConfiguration$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

StrategyConfiguration, DependencyBasedStrategy

FisherTestHeuristic *Feature-clustering based on Fisher's Exact Test.*

Description

Performs feature-clustering based on Fisher's exact test for testing the null of independence of rows and columns in a contingency table with fixed marginals.

Super class

D2MCS::GenericHeuristic -> FisherTestHeuristic

Methods

Public methods:

- FisherTestHeuristic$new()
- FisherTestHeuristic$heuristic()
- FisherTestHeuristic$clone()

Method new(): Empty function used to initialize the object arguments in runtime.

Usage:

```
FisherTestHeuristic$new()
```

Method heuristic(): Performs the Fisher's exact test for testing the null of independence between two columns (col1 and col2).

Usage:

```
FisherTestHeuristic$heuristic(col1, col2, column.names = NULL)
```

Arguments:

col1 A numeric vector or matrix required to perform the clustering operation.

col2 A numeric vector or matrix to perform the clustering operation.

column.names An optional character vector with the names of both columns.

Returns: A numeric vector of length 1 or NA if an error occurs.

Method clone(): The objects of this class are cloneable with this method.

Usage:

```
FisherTestHeuristic$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Dataset, fisher.test

FN *Computes the False Negative errors.*

Description

Computes the ratio of number of Type II errors achieved by the final M.L. model.

Super class

D2MCS::MeasureFunction -> FN

Methods

Public methods:

- FN$new()
- FN$compute()
- FN$clone()

Method `new()`**:** Method for initializing the object arguments during runtime.

Usage:

```
FN$new(performance.output = NULL)
```

Arguments:

`performance.output` An optional ConfMatrix parameter to define the type of object used to compute the **FN** measure.

Method `compute()`**:** The function computes the **FN** achieved by the M.L. model.

Usage:

```
FN$compute(performance.output = NULL)
```

Arguments:

`performance.output` An optional ConfMatrix parameter to define the type of object used as basis to compute the **FN** measure

Details: This function is automatically invoked by the ClassificationOutput framework.

Returns: A numeric vector of size 1 or NULL if an error occurred.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
FN$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

MeasureFunction, ClassificationOutput, ConfMatrix

FP *Computes the False Positive value.*

Description

This is the number of individuals with a negative condition for which the test result is positive. The value entered here must be non-negative.

Super class

`D2MCS::MeasureFunction` -> `FP`

Methods

Public methods:

- `FP$new()`
- `FP$compute()`
- `FP$clone()`

Method `new()`**:** Method for initializing the object arguments during runtime.

Usage:

```
FP$new(performance.output = NULL)
```

Arguments:

`performance.output` An optional `ConfMatrix` parameter used as basis to define the type of compute the `FP` measure.

Method `compute()`**:** The function computes the **FP** achieved by the M.L. model.

Usage:

```
FP$compute(performance.output = NULL)
```

Arguments:

`performance.output` An optional `ConfMatrix` parameter to define the type of object used as basis to compute the `FP` measure.

Details: This function is automatically invoked by the `ClassificationOutput` object.

Returns: A numeric vector of size 1 or NULL if an error occurred.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
FP$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`MeasureFunction`, `ClassificationOutput`, `ConfMatrix`

GainRatioHeuristic *Feature-clustering based on GainRatio methodology.*

Description

Performs the feature-clustering using entropy-based filters.

Super class

D2MCS::GenericHeuristic -> GainRatioHeuristic

Methods

Public methods:

- GainRatioHeuristic$new()
- GainRatioHeuristic$heuristic()
- GainRatioHeuristic$clone()

Method new(): Empty function used to initialize the object arguments in runtime.

Usage:

```
GainRatioHeuristic$new()
```

Method heuristic(): The algorithms find weights of discrete attributes basing on their correlation with continuous class attribute.

Usage:

```
GainRatioHeuristic$heuristic(col1, col2, column.names = NULL)
```

Arguments:

col1 A numeric vector or matrix required to perform the clustering operation.

col2 A numeric vector or matrix to perform the clustering operation.

column.names An optional character vector with the names of both columns.

Returns: A numeric vector of length 1 or NA if an error occurs.

Method clone(): The objects of this class are cloneable with this method.

Usage:

```
GainRatioHeuristic$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Dataset, gain.ratio

GenericClusteringStrategy *Abstract Feature Clustering Strategy class.*

Description

Abstract class used as a template to ensure the proper definition of new customized clustering strategies.

Details

The GenericClusteringStrategy is an archetype class so it cannot be instantiated.

Methods

Public methods:

- `GenericClusteringStrategy$new()`
- `GenericClusteringStrategy$getDescription()`
- `GenericClusteringStrategy$getHeuristic()`
- `GenericClusteringStrategy$getConfiguration()`
- `GenericClusteringStrategy$getBestClusterDistribution()`
- `GenericClusteringStrategy$getUnclustered()`
- `GenericClusteringStrategy$execute()`
- `GenericClusteringStrategy$getDistribution()`
- `GenericClusteringStrategy$createTrain()`
- `GenericClusteringStrategy$plot()`
- `GenericClusteringStrategy$saveCSV()`
- `GenericClusteringStrategy$clone()`

Method `new()`**:** A function responsible for creating a GenericClusteringStrategy object.

Usage:

```
GenericClusteringStrategy$new(subset, heuristic, description, configuration)
```

Arguments:

`subset` A `Subset` object to perform the clustering strategy.

`heuristic` The heuristic to be applied. Must inherit from `GenericHeuristic` class.

`description` A character vector describing the strategy operation.

`configuration` Optional customized configuration parameters for the strategy. Must inherited from `StrategyConfiguration` abstract class.

Method `getDescription()`**:** The function is used to obtain the description of the strategy.

Usage:

```
GenericClusteringStrategy$getDescription()
```

Returns: A character vector of NULL if not defined.

Method `getHeuristic()`**:** The function returns the heuristic applied for the clustering strategy.

Usage:

```
GenericClusteringStrategy$getHeuristic()
```

Returns: An object inherited from `GenericClusteringStrategy` class.

Method `getConfiguration()`**:** The function returns the configuration parameters used to perform the clustering strategy.

Usage:

```
GenericClusteringStrategy$getConfiguration()
```

Returns: An object inherited from `StrategyConfiguration` class.

Method `getBestClusterDistribution()`**:** The function obtains the best clustering distribution.

Usage:

```
GenericClusteringStrategy$getBestClusterDistribution()
```

Returns: A list of clusters. Each list element represents a feature group.

Method `getUnclustered()`**:** The function is used to return the features that cannot be clustered due to incompatibilities with the used heuristic.

Usage:

```
GenericClusteringStrategy$getUnclustered()
```

Returns: A character vector containing the unclassified features.

Method `execute()`**:** Abstract function responsible of performing the clustering strategy over the defined `Subset`.

Usage:

```
GenericClusteringStrategy$execute(verbose, ...)
```

Arguments:

`verbose` A logical value to specify if more verbosity is needed.

`...` Further arguments passed down to `execute` function.

Method `getDistribution()`**:** Abstract function used to obtain the set of features following an specific clustering distribution.

Usage:

```
GenericClusteringStrategy$getDistribution(
  num.clusters = NULL,
  num.groups = NULL,
  include.unclustered = FALSE
)
```

Arguments:

`num.clusters` A numeric value to select the number of clusters (define the distribution).

`num.groups` A single or numeric vector value to identify a specific group that forms the clustering distribution.

`include.unclustered` A logical value to determine if unclustered features should be included.

Returns: A list with the features comprising an specific clustering distribution.

Method `createTrain()`**:** Abstract function in charge of creating a `Trainset` object for training purposes.

Usage:

```
GenericClusteringStrategy$createTrain(
  subset,
  num.cluster = NULL,
  num.groups = NULL,
  include.unclustered = FALSE
)
```

Arguments:

`subset` A `Subset` object used as a basis to create the Trainset

`num.cluster` A numeric value to select the number of clusters (define the distribution).

`num.groups` A single or numeric vector value to identify a specific group that forms the clustering distribution.

`include.unclustered` A logical value to determine if unclustered features should be included.

Method `plot()`**:** Abstract function responsible of creating a plot to visualize the clustering distribution.

Usage:

```
GenericClusteringStrategy$plot(dir.path = NULL, file.name = NULL, ...)
```

Arguments:

`dir.path` An optional character argument to define the name of the directory where the exported plot will be saved. If not defined, the file path will be automatically assigned to the current working directory, '`getwd()`'.

`file.name` The name of the PDF file where the plot is exported.

`...` Further arguments passed down to `execute` function.

Method `saveCSV()`**:** Abstract function to save the clustering distribution to a CSV file.

Usage:

```
GenericClusteringStrategy$saveCSV(dir.path, name, num.clusters = NULL)
```

Arguments:

`dir.path` The name of the directory to save the CSV file.

`name` Defines the name of the CSV file.

`num.clusters` An optional parameter to select the number of clusters to be saved. If not defined, all clusters will be saved.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
GenericClusteringStrategy$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`Subset`, `GenericHeuristic`

GenericHeuristic *Abstract Feature Clustering heuristic object.*

Description

Abstract class used as a template to define new customized clustering heuristics.

Details

The GenericHeuristic is an archetype class so it cannot be instantiated.

Methods

Public methods:

- `GenericHeuristic$new()`
- `GenericHeuristic$heuristic()`
- `GenericHeuristic$clone()`

Method `new()`**:** Empty function used to initialize the object arguments in runtime.

Usage:

```
GenericHeuristic$new()
```

Method `heuristic()`**:** Function used to implement the clustering heuristic.

Usage:

```
GenericHeuristic$heuristic(col1, col2, column.names = NULL, ...)
```

Arguments:

`col1` A numeric vector or matrix required to perform the clustering operation.

`col2` A numeric vector or matrix to perform the clustering operation.

`column.names` An optional character vector with the names of both columns

`...` Further arguments passed down to `heuristic` function.

Returns: A numeric vector of length 1.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
GenericHeuristic$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`Dataset`

GenericModelFit *Abstract class for defining model fitting method.*

Description

Template to create a recipe or formula objects used in model training stage.

Methods

Public methods:

- GenericModelFit$new()
- GenericModelFit$createFormula()
- GenericModelFit$createRecipe()
- GenericModelFit$clone()

Method new(): Method for initializing the object arguments during runtime.

Usage:

```
GenericModelFit$new()
```

Method createFormula(): The function is responsible of creating a formula for M.L. model.

Usage:

```
GenericModelFit$createFormula(instances, class.name, simplify = TRUE)
```

Arguments:

instances A data.frame containing the instances used to create the recipe.

class.name A character vector representing the name of the target class.

simplify A logical argument defining whether the formula should be generated as simple as possible.

Returns: A formula object.

Method createRecipe(): The function is responsible of creating a recipe for M.L. model.

Usage:

```
GenericModelFit$createRecipe(instances, class.name)
```

Arguments:

instances A data.frame containing the instances used to create the recipe.

class.name A character vector representing the name of the target class.

Returns: A object of class recipe.

Method clone(): The objects of this class are cloneable with this method.

Usage:

```
GenericModelFit$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

DefaultModelFit, train

GenericPlot *Pseudo-abstract class for creating feature clustering plots.*

Description

The GenericPlot implements a basic plot.

Methods

Public methods:

- GenericPlot$new()
- GenericPlot$plot()
- GenericPlot$clone()

Method `new()`**:** Empty function used to initialize the object arguments in runtime.

Usage:

```
GenericPlot$new()
```

Method `plot()`**:** Implements a generic plot to visualize basic feature-clustering data.

Usage:

```
GenericPlot$plot(summary)
```

Arguments:

`summary` A data.frame comprising the elements to be plotted.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
GenericPlot$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

BinaryPlot

HDDataset *High Dimensional Dataset handler.*

Description

Creates a high dimensional dataset object. Only the required instances are loaded in memory to avoid unnecessary of resources and memory.

Methods

Public methods:

- HDDataset$new()
- HDDataset$getColumnNames()
- HDDataset$getNcol()
- HDDataset$createSubset()

Method `new()`**:** Method for initializing the object arguments during runtime.

Usage:

```
HDDataset$new(
  filepath,
  header = TRUE,
  sep = ",",
  skip = 0,
  normalize.names = FALSE,
  ignore.columns = NULL
)
```

Arguments:

`filepath` The name of the file which the data are to be read from. Each row of the table appears as one line of the file. If it does not contain an _absolute_ path, the file name is _relative_ to the current working directory, '`getwd()`'.

`header` A logical value indicating whether the file contains the names of the variables as its first line. If missing, the value is determined from the file format: '`header`' is set to '`TRUE`' if and only if the first row contains one fewer field than the number of columns.

`sep` The field separator character. Values on each line of the file are separated by this character.

`skip` Defines the number of header lines should be skipped.

`normalize.names` A logical value indicating whether the columns names should be automatically renamed to ensure R compatibility.

`ignore.columns` Specify the columns from the input file that should be ignored.

Method `getColumnNames()`**:** Gets the name of the columns comprising the dataset

Usage:

```
HDDataset$getColumnNames()
```

Returns: A character vector with the name of each column.

Method `getNcol()`**:** Obtains the number of columns present in the dataset.

Usage:

```
HDDataset$getNcol()
```

Returns: An integer of length 1 or NULL

Method `createSubset()`**:** Creates a blinded HDSubset for classification purposes.

Usage:

```
HDDataset$createSubset(column.id = FALSE, chunk.size = 1e+05)
```

Arguments:

`column.id` An integer or character indicating the column (number or name respectively) identifier. Default NULL value is valid ignores defining a identification column.

`chunk.size` an integer value indicating the size of chunks taken over each iteration.

Returns: A `HDSubset` object.

See Also

`Dataset`, `HDSubset`, `DatasetLoader`

`HDSubset` *High Dimensional Subset handler.*

Description

Creates a high dimensional subset from a `HDDataset` object. Only the required instances are loaded in memory to avoid unnecessary use of resources and memory.

Details

Use `HDDataset` to ensure the creation of a valid `HDSubset` object.

Methods

Public methods:

- `HDSubset$new()`
- `HDSubset$getColumnNames()`
- `HDSubset$getNcol()`
- `HDSubset$getID()`
- `HDSubset$getIterator()`
- `HDSubset$isBlinded()`
- `HDSubset$clone()`

Method `new()`**:** Method for initializing the object arguments during runtime.

Usage:

```
HDSubset$new(
  file.path,
  feature.names,
  feature.id,
  start.at = 0,
  sep = ",",
  chunk.size
)
```

Arguments:

`file.path` The name of the file which the data are to be read from. Each row of the table appears as one line of the file. If it does not contain an _absolute_ path, the file name is _relative_ to the current working directory, '`getwd()`'.

`feature.names` A character vector specifying the name of the features that should be included in the `HDDataset` object.

`feature.id` An integer or character indicating the column (number or name respectively) identifier. Default NULL value is valid ignores defining a identification column.

`start.at` A numeric value to identify the reading start position.

`sep` the field separator character. Values on each line of the file are separated by this character.

`chunk.size` an integer value indicating the size of chunks taken over each iteration. By default chunk.size is defined as 10000.

Method `getColumnNames()`**:** Gets the name of the columns comprising the subset.

Usage:

```
HDSubset$getColumnNames()
```

Returns: A character vector containing the name of each column.

Method `getNcol()`**:** Obtains the number of columns present in the dataset.

Usage:

```
HDSubset$getNcol()
```

Returns: A numeric value or 0 if is empty.

Method `getID()`**:** Obtains the column identifier.

Usage:

```
HDSubset$getID()
```

Returns: A character vector of size 1.

Method `getIterator()`**:** Creates the `FIterator` object.

Usage:

```
HDSubset$getIterator(chunk.size = private$chunk.size, verbose = FALSE)
```

Arguments:

`chunk.size` An integer value indicating the size of chunks taken over each iteration. By default `chunk.size` is defined as 10000.

`verbose` A logical value to specify if more verbosity is needed.

Returns: A `FIterator` object to transverse through `HDSubset` instances

Method `isBlinded()`**:** Checks if the subset contains a target class.

Usage:

```
HDSubset$isBlinded()
```

Returns: A logical to specify if the subset contains a target class or not.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
HDSubset$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`HDDataset`, `DatasetLoader`

`InformationGainHeuristic` *Feature-clustering based on InformationGain methodology.*

Description

Performs the feature-clustering using entropy-based filters.

Super class

`D2MCS::GenericHeuristic` -> `InformationGainHeuristic`

Methods

Public methods:

- `InformationGainHeuristic$new()`
- `InformationGainHeuristic$heuristic()`
- `InformationGainHeuristic$clone()`

Method `new()`**:** Empty function used to initialize the object arguments in runtime.

Usage:

```
InformationGainHeuristic$new()
```

Method `heuristic()`**:** The algorithm find weights of discrete attributes basing on their correlation with continuous class attribute. Particularly Information Gain uses `H(Class) + H(Attribute) -H(Class,Attribute)`

Usage:

```
InformationGainHeuristic$heuristic(col1, col2, column.names = NULL)
```

Arguments:

col1 A numeric vector or matrix required to perform the clustering operation.

col2 A numeric vector or matrix to perform the clustering operation.

column.names an optional character vector with the names of both columns.

Returns: A numeric vector of length 1 or NA if an error occurs.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
InformationGainHeuristic$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Dataset, information.gain

Kappa *Computes the Kappa Cohen value.*

Description

Cohen's Kappa measures the agreement between two raters who each classify N items into C mutually exclusive categories.

Details

$$\kappa \; is \; equivalent \; to \; (p_o - p_e)/(1 - p_e) = 1 - (1 - p_0)/(1 - p_e)$$

Super class

D2MCS::MeasureFunction -> Kappa

Methods

Public methods:

- Kappa$new()
- Kappa$compute()
- Kappa$clone()

Method `new()`**:** Method for initializing the object arguments during runtime.

Usage:

```
Kappa$new(performance.output = NULL)
```

Arguments:

performance.output An optional ConfMatrix used as basis to compute the performance.

Method `compute()`**:** The function computes the **Kappa** achieved by the M.L. model.

Usage:

```
Kappa$compute(performance.output = NULL)
```

Arguments:

`performance.output` An optional `ConfMatrix` parameter to define the type of object used as basis to compute the Kappa measure.

Details: This function is automatically invoked by the ClassificationOutput object.

Returns: A numeric vector of size 1 or NULL if an error occurred.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Kappa$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`MeasureFunction`, `ClassificationOutput`, `ConfMatrix`

`KendallHeuristic` *Feature-clustering based on Kendall Correlation Test.*

Description

Performs the feature-clustering using Kendall correlation tests.

Details

The method estimate the association between paired samples and compute a test of the value being zero. They use different measures of association, all in the range [-1, 1] with 0 indicating no association. Method valid only for bi-class problems.

Super class

`D2MCS::GenericHeuristic` -> `KendallHeuristic`

Methods

Public methods:

- `KendallHeuristic$new()`
- `KendallHeuristic$heuristic()`
- `KendallHeuristic$clone()`

Method `new()`**:** Empty function used to initialize the object arguments in runtime.

Usage:

```
KendallHeuristic$new()
```

Method `heuristic()`**:** Test for association between paired samples using Kendall's tau value.

Usage:

```
KendallHeuristic$heuristic(col1, col2, column.names = NULL)
```

Arguments:

`col1` A numeric vector or matrix required to perform the clustering operation.

`col2` A numeric vector or matrix to perform the clustering operation.

`column.names` An optional character vector with the names of both columns.

Returns: a numeric vector of length 1 or NA if an error occurs.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
KendallHeuristic$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`Dataset`, `cor.test`

`MCC` *Computes the Matthews correlation coefficient.*

Description

The Matthews correlation coefficient is used in machine learning as a measure of the quality of binary (two-class) classifications. It takes into account true and false positives and negatives and is generally regarded as a balanced measure which can be used even if the classes are of very different sizes. The MCC is in essence a correlation coefficient between the observed and predicted binary classifications; it returns a value between -1 and +1.

Details

$$MCC = (TP(TN - FP)FN)/(\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)})$$

Super class

`D2MCS::MeasureFunction` -> `MCC`

Methods

Public methods:

- MCC$new()
- MCC$compute()
- MCC$clone()

Method `new()`**:** Method for initializing the object arguments during runtime.

Usage:

```
MCC$new(performance.output = NULL)
```

Arguments:

`performance.output` An optional ConfMatrix parameter used as basis to compute the MCC measure.

Method `compute()`**:** The function computes the **MCC** achieved by the M.L. model.

Usage:

```
MCC$compute(performance.output = NULL)
```

Arguments:

`performance.output` An optional ConfMatrix parameter to define the type of object used as basis to compute the MCC measure.

Details: This function is automatically invoke by the ClassificationOutput object.

Returns: A numeric vector of size 1 or NULL if an error occurred.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
MCC$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

MeasureFunction, ClassificationOutput, ConfMatrix

`MCCHeuristic` *Feature-clustering based on Matthews Correlation Coefficient score.*

Description

Performs the feature-clustering using MCC score. Valid for both bi-class and multi-class problems

Super class

D2MCS::GenericHeuristic -> MCCHeuristic

Methods

Public methods:

- MCCHeuristic$new()
- MCCHeuristic$heuristic()
- MCCHeuristic$clone()

Method `new()`**:** Empty function used to initialize the object arguments in runtime.

Usage:

```
MCCHeuristic$new()
```

Method `heuristic()`**:** Calculates the Matthews correlation Coefficient (MCC) score.

Usage:

```
MCCHeuristic$heuristic(col1, col2, column.names = NULL)
```

Arguments:

`col1` A numeric vector or matrix required to perform the clustering operation.

`col2` A numeric vector or matrix to perform the clustering operation.

`column.names` An optional character vector with the names of both columns.

Returns: A numeric vector of length 1 or NA if an error occurs.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
MCCHeuristic$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Dataset, mccr

`MeasureFunction` *Archetype to define customized measures.*

Description

Abstract class used as a template to define new M.L. performance measures.

Details

The GenericHeuristic is an full-abstract class so it cannot be instantiated. To ensure the proper operation, `compute` method is automatically invoke by D2MCS framework when needed.

Methods

Public methods:

- `MeasureFunction$new()`
- `MeasureFunction$compute()`
- `MeasureFunction$clone()`

Method `new()`**:** Method for initializing the object arguments during runtime.

Usage:

```
MeasureFunction$new(performance = NULL)
```

Arguments:

`performance` An optional `ConfMatrix` parameter to define the type of object used to compute the measure.

Method `compute()`**:** The function implements the metric used to measure the performance achieved by the M.L. model.

Usage:

```
MeasureFunction$compute(performance.output = NULL)
```

Arguments:

`performance.output` An optional `ConfMatrix` parameter to define the type of object used to compute the measure.

Details: This function is automatically invoke by the `D2MCS` framework.

Returns: A numeric vector of size 1 or NULL if an error occurred.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
MeasureFunction$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`MeasureFunction`

`Methodology` *Abstract class to compute the probability prediction based on combination between metrics.*

Description

Abstract class used as a template to define new customized strategies to combine the probability predictions made by different metrics.

Methods

Public methods:

- Methodology$new()
- Methodology$getRequiredMetrics()
- Methodology$compute()
- Methodology$clone()

Method `new()`**:** Method for initializing the object arguments during runtime.

Usage:

```
Methodology$new(required.metrics)
```

Arguments:

`required.metrics` A character vector of length greater than 2 with the name of the required metrics.

Method `getRequiredMetrics()`**:** The function returns the required metrics that will participate in the methodology to compute a metric based on all of them.

Usage:

```
Methodology$getRequiredMetrics()
```

Returns: A character vector of length greater than 2 with the name of the required metrics.

Method `compute()`**:** Function to compute the probability of the final prediction based on different metrics.

Usage:

```
Methodology$compute(raw.pred, prob.pred, positive.class, negative.class)
```

Arguments:

`raw.pred` A character list of length greater than 2 with the class value of the predictions made by the metrics.

`prob.pred` A numeric list of length greater than 2 with the probability of the predictions made by the metrics.

`positive.class` A character with the value of the positive class.

`negative.class` A character with the value of the negative class.

Returns: A numeric value indicating the probability of the instance is predicted as positive class.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Methodology$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

ProbBasedMethodology

MinimizeFN *Combined metric strategy to minimize FN errors.*

Description

Calculates if the positive class is the predicted one in any of the metrics, otherwise, the instance is not considered to have the positive class associated.

Super class

D2MCS::CombinedMetrics -> MinimizeFN

Methods

Public methods:

- MinimizeFN$new()
- MinimizeFN$getFinalPrediction()
- MinimizeFN$clone()

Method `new()`**:** Method for initializing the object arguments during runtime.

Usage:

```
MinimizeFN$new(required.metrics = c("MCC", "PPV"))
```

Arguments:

`required.metrics` A character vector of length 1 with the name of the required metrics.

Method `getFinalPrediction()`**:** Function to obtain the final prediction based on different metrics.

Usage:

```
MinimizeFN$getFinalPrediction(
  raw.pred,
  prob.pred,
  positive.class,
  negative.class
)
```

Arguments:

`raw.pred` A character list of length greater than 2 with the class value of the predictions made by the metrics.

`prob.pred` A numeric list of length greater than 2 with the probability of the predictions made by the metrics.

`positive.class` A character with the value of the positive class.

`negative.class` A character with the value of the negative class.

Returns: A logical value indicating if the instance is predicted as positive class or not.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
MinimizeFN$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`CombinedMetrics`

`MinimizeFP` *Combined metric strategy to minimize FP errors.*

Description

Calculates if the positive class is the predicted one in all metrics, otherwise, the instance is not considered to have the positive class associated.

Super class

`D2MCS::CombinedMetrics` -> `MinimizeFP`

Methods

Public methods:

- `MinimizeFP$new()`
- `MinimizeFP$getFinalPrediction()`
- `MinimizeFP$clone()`

Method `new()`**:** Method for initializing the object arguments during runtime.

Usage:

```
MinimizeFP$new(required.metrics = c("MCC", "PPV"))
```

Arguments:

`required.metrics` A character vector of length greater than 2 with the name of the required metrics.

Method `getFinalPrediction()`**:** Function to obtain the final prediction based on different metrics.

Usage:

```
MinimizeFP$getFinalPrediction(
  raw.pred,
  prob.pred,
  positive.class,
  negative.class
)
```

Arguments:

`raw.pred` A character list of length greater than 2 with the class value of the predictions made by the metrics.

`prob.pred` A numeric list of length greater than 2 with the probability of the predictions made by the metrics.

`positive.class` A character with the value of the positive class.

`negative.class` A character with the value of the negative class.

Returns: A logical value indicating if the instance is predicted as positive class or not.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
MinimizeFP$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`CombinedMetrics`

`MultinformationHeuristic`

Feature-clustering based on Mutual Information Computation theory.

Description

Performs the feature-clustering using MCC score. Valid for both bi-class and multi-class problems. Only valid for bi-class problems.

Super class

`D2MCS::GenericHeuristic` -> `MultinformationHeuristic`

Methods

Public methods:

- `MultinformationHeuristic$new()`
- `MultinformationHeuristic$heuristic()`
- `MultinformationHeuristic$clone()`

Method `new()`**:** Empty function used to initialize the object arguments in runtime.

Usage:

```
MultinformationHeuristic$new()
```

Method `heuristic()`**:** Mutinformation takes two random variables as input and computes the mutual information in nats according to the entropy estimator method.

Usage:

```
MultinformationHeuristic$heuristic(col1, col2, column.names = NULL)
```

Arguments:

`col1` A vector/factor denoting a random variable or a data.frame denoting a random vector where columns contain variables/features and rows contain outcomes/samples.

`col2` An another random variable or random vector (vector/factor or data.frame).

`column.names` An optional character vector with the names of both columns.

Returns: Returns the mutual information I(X;Y) in nats.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
MultinformationHeuristic$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`Dataset`, `mutinformation`

`NoProbability` *Compute performance across resamples.*

Description

Computes the performance across resamples when class probabilities cannot be computed.

Super class

`D2MCS::SummaryFunction` -> `NoProbability`

Methods

Public methods:

- `NoProbability$new()`
- `NoProbability$execute()`
- `NoProbability$clone()`

Method `new()`**:** The function defined during runtime the usage of five measures: 'Kappa', 'Accuracy', 'TCR_9', 'MCC' and 'PPV'.

Usage:

```
NoProbability$new()
```

Method `execute()`**:** The function computes the performance across resamples using the previously defined measures.

Usage:

```
NoProbability$execute(data, lev = NULL, model = NULL)
```

Arguments:

`data` A data.frame containing the data used to compute the performance.

`lev` An optional value used to define the levels of the target class.

`model` An optional value used to define the M.L. model used.

Returns: A vector of performance estimates.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
NoProbability$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`SummaryFunction`

`NPV` *Computes the Negative Predictive Value.*

Description

Negative Predictive Values are the proportions of negative results in statistics and diagnostic tests that are true negative results.

Details

$$NPV = TN/(TN + FN)$$

Super class

`D2MCS::MeasureFunction` -> `NPV`

Methods

Public methods:

- `NPV$new()`
- `NPV$compute()`
- `NPV$clone()`

Method `new()`**:** Method for initializing the object arguments during runtime.

Usage:

```
NPV$new(performance.output = NULL)
```

Arguments:

`performance.output` An optional `ConfMatrix` parameter to define the type of object used as basis to compute the `NPV` measure.

Method `compute()`**:** The function computes the **NPV** achieved by the M.L. model.

Usage:

```
NPV$compute(performance.output = NULL)
```

Arguments:

`performance.output` An optional `ConfMatrix` parameter to define the type of object used as basis to compute the **NPV** measure.

Details: This function is automatically invoke by the ClassificationOutput object.

Returns: A numeric vector of size 1 or NULL if an error occurred.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
NPV$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`MeasureFunction`, `ClassificationOutput`, `ConfMatrix`

`OddsRatioHeuristic` *Feature-clustering based on Odds Ratio measure.*

Description

Performs the feature-clustering using Odds Ratio methodology. Valid only for bi-class problems.

Super class

`D2MCS::GenericHeuristic` -> `OddsRatioHeuristic`

Methods

Public methods:

- `OddsRatioHeuristic$new()`
- `OddsRatioHeuristic$heuristic()`
- `OddsRatioHeuristic$clone()`

Method `new()`**:** Empty function used to initialize the object arguments in runtime.

Usage:

```
OddsRatioHeuristic$new()
```

Method `heuristic()`**:** Calculates the Odds Ratio method.

Usage:

```
OddsRatioHeuristic$heuristic(col1, col2, column.names = NULL)
```

Arguments:

`col1` The object from whom odds ratio will be computed.

`col2` A second factor or numeric object.

`column.names` An optional character vector with the names of both columns.

Returns: A numeric vector of length 1 or NA if an error occurs.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
OddsRatioHeuristic$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`Dataset`, `odds.ratio`

`PearsonHeuristic` *Feature-clustering based on Pearson Correlation Test.*

Description

Performs the feature-clustering using Pearson correlation tests. Valid for both, bi-class and multi-class problems.

Details

The test statistic is based on Pearson's product moment correlation coefficient cor(x, y) and follows a t distribution with length(x)-2 degrees of freedom if the samples follow independent normal distributions. If there are at least 4 complete pairs of observation, an asymptotic confidence interval is given based on Fisher's Z transform.

Super class

`D2MCS::GenericHeuristic` -> `PearsonHeuristic`

Methods

Public methods:

- PearsonHeuristic$new()
- PearsonHeuristic$heuristic()
- PearsonHeuristic$clone()

Method `new()`**:** Creates a PearsonHeuristic object.

Usage:

```
PearsonHeuristic$new()
```

Method `heuristic()`**:** Test for association between paired samples using Pearson test.

Usage:

```
PearsonHeuristic$heuristic(col1, col2, column.names = NULL)
```

Arguments:

`col1` A numeric vector or matrix required to perform the clustering operation.

`col2` A numeric vector or matrix to perform the clustering operation.

`column.names` An optional character vector with the names of both columns.

Returns: A numeric vector of length 1 or NA if an error occurs.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
PearsonHeuristic$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Dataset, cor

PPV *Computes the Positive Predictive Value.*

Description

Positive Predictive Values are the proportions of positive results in statistics and diagnostic tests that are true positive results.

Details

$$PPV = TP/(TP + FP)$$

Super class

D2MCS::MeasureFunction -> PPV

Methods

Public methods:

- PPV$new()
- PPV$compute()
- PPV$clone()

Method `new()`**:** Method for initializing the object arguments during runtime.

Usage:

```
PPV$new(performance.output = NULL)
```

Arguments:

`performance.output` An optional ConfMatrix parameter to define the type of object used as basis to compute the **PPV** measure.

Method `compute()`**:** The function computes the **PPV** achieved by the M.L. model.

Usage:

```
PPV$compute(performance.output = NULL)
```

Arguments:

`performance.output` An optional ConfMatrix parameter to define the type of object used as basis to compute the **PPV** measure.

Details: This function is automatically invoke by the ClassificationOutput object.

Returns: A numeric vector of size 1 or NULL if an error occurred.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
PPV$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

MeasureFunction, ClassificationOutput, ConfMatrix

`Precision` *Computes the Precision Value.*

Description

Precision is the fraction of relevant instances among the retrieved instances

Details

$$precision = TP/(TP + FP)$$

Super class

D2MCS::MeasureFunction -> Precision

Methods

Public methods:

- Precision$new()
- Precision$compute()
- Precision$clone()

Method new(): Method for initializing the object arguments during runtime.

Usage:

```
Precision$new(performance.output = NULL)
```

Arguments:

performance.output An optional ConfMatrix parameter to define the type of object used as basis to compute the measure.

Method compute(): The function computes the **Precision** achieved by the M.L. model.

Usage:

```
Precision$compute(performance.output = NULL)
```

Arguments:

performance.output An optional ConfMatrix parameter to define the type of object used as basis to compute the **Precision** measure.

Details: This function is automatically invoke by the ClassificationOutput object.

Returns: A numeric vector of size 1 or NULL if an error occurred.

Method clone(): The objects of this class are cloneable with this method.

Usage:

```
Precision$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

MeasureFunction, ClassificationOutput, ConfMatrix

PredictionOutput *Encapsulates the achieved predictions.*

Description

The class used to encapsulates all the computed predictions to facilitate their access and maintenance.

Methods

Public methods:

- PredictionOutput$new()
- PredictionOutput$getPredictions()
- PredictionOutput$getType()
- PredictionOutput$getTarget()
- PredictionOutput$clone()

Method `new()`**:** Method for initializing the object arguments during runtime.

Usage:

```
PredictionOutput$new(predictions, type, target)
```

Arguments:

`predictions` A list of `FinalPred` elements.

`type` A character to define which type of predictions should be returned. If not defined all type of probabilities will be returned. Conversely if "prob" or "raw" is defined then computed 'probabilistic' or 'class' values are returned.

`target` A character defining the value of the positive class.

Method `getPredictions()`**:** The function returns the final predictions.

Usage:

```
PredictionOutput$getPredictions()
```

Returns: A list containing the final predictions or NULL if classification stage was not successfully performed.

Method `getType()`**:** The function returns the type of prediction should be returned. If "prob" or "raw" is defined then computed 'probabilistic' or 'class' values are returned.

Usage:

```
PredictionOutput$getType()
```

Returns: A character value.

Method `getTarget()`**:** The function returns the value of the target class.

Usage:

```
PredictionOutput$getTarget()
```

Returns: A character value.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
PredictionOutput$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

D2MCS

`ProbAverageVoting` *Implementation of Probabilistic Average voting.*

Description

Computes the final prediction by performing the mean value of the probability achieved by each prediction.

Super class

`D2MCS::SimpleVoting` -> `ProbAverageVoting`

Methods

Public methods:

- `ProbAverageVoting$new()`
- `ProbAverageVoting$getMajorityClass()`
- `ProbAverageVoting$getClassTie()`
- `ProbAverageVoting$execute()`
- `ProbAverageVoting$clone()`

Method `new()`**:** Method for initializing the object arguments during runtime.

Usage:

```
ProbAverageVoting$new(cutoff = 0.5, class.tie = NULL, majority.class = NULL)
```

Arguments:

`cutoff` A character vector defining the minimum probability used to perform a positive classification. If is not defined, 0.5 will be used as default value.

`class.tie` A character used to define the target class value used when a tie is found. If NULL positive class value will be assigned.

`majority.class` A character defining the value of the majority class. If NULL will be used same value as training stage.

Method `getMajorityClass()`**:** The function returns the value of the majority class.

Usage:

```
ProbAverageVoting$getMajorityClass()
```

Returns: A character vector of length 1 with the name of the majority class.

Method `getClassTie()`**:** The function gets the class value assigned to solve ties.

Usage:

```
ProbAverageVoting$getClassTie()
```

Returns: A character vector of length 1.

Method `execute()`**:** The function implements the majority voting procedure.

Usage:

```
ProbAverageVoting$execute(predictions, verbose = FALSE)
```

Arguments:

`predictions` A `ClusterPredictions` object containing all the predictions achieved for each cluster.

`verbose` A logical value to specify if more verbosity is needed.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
ProbAverageVoting$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`D2MCS`, `ClassMajorityVoting`, `ClassWeightedVoting`, `ProbAverageVoting`, `ProbAverageWeightedVoting`, `ProbBasedMethodology`

`ProbAverageWeightedVoting` *Implementation of Probabilistic Average Weighted voting.*

Description

Computes the final prediction by performing the weighted mean of the probability achieved by each cluster prediction. By default, weight values are consistent with the performance value achieved by the best M.L. model on each cluster.

Super class

`D2MCS::SimpleVoting` -> `ProbAverageWeightedVoting`

Methods

Public methods:

- ProbAverageWeightedVoting$new()
- ProbAverageWeightedVoting$getClassTie()
- ProbAverageWeightedVoting$getWeights()
- ProbAverageWeightedVoting$setWeights()
- ProbAverageWeightedVoting$execute()
- ProbAverageWeightedVoting$clone()

Method `new()`**:** Method for initializing the object arguments during runtime.

Usage:

```
ProbAverageWeightedVoting$new(cutoff = 0.5, class.tie = NULL, weights = NULL)
```

Arguments:

`cutoff` A character vector defining the minimum probability used to perform a positive classification. If is not defined, 0.5 will be used as default value.

`class.tie` A character used to define the target class value used when a tie is found. If NULL positive class value will be assigned.

`weights` A numeric vector with the weights of each cluster. If NULL performance achieved during training will be used as default.

Method `getClassTie()`**:** The function gets the class value assigned to solve ties.

Usage:

```
ProbAverageWeightedVoting$getClassTie()
```

Returns: A character vector of length 1.

Method `getWeights()`**:** The function returns the value of the majority class.

Usage:

```
ProbAverageWeightedVoting$getWeights()
```

Returns: A character vector of length 1 with the name of the majority class.

Method `setWeights()`**:** The function allows changing the value of the weights.

Usage:

```
ProbAverageWeightedVoting$setWeights(weights)
```

Arguments:

`weights` A numeric vector containing the new weights.

Method `execute()`**:** The function implements the cluster-weighted probabilistic voting procedure.

Usage:

```
ProbAverageWeightedVoting$execute(predictions, verbose = FALSE)
```

Arguments:

`predictions` A `ClusterPredictions` object containing all the predictions achieved for each cluster.

`verbose` A logical value to specify if more verbosity is needed.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
ProbAverageWeightedVoting$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`D2MCS`, `ClassMajorityVoting`, `ClassWeightedVoting`, `ProbAverageVoting`, `ProbAverageWeightedVoting`, `ProbBasedMethodology`

`ProbBasedMethodology` *Methodology to obtain the combination of the probability of different metrics.*

Description

Calculates the mean of the probabilities of the different metrics.

Super class

`D2MCS::Methodology` -> `ProbBasedMethodology`

Methods

Public methods:

- `ProbBasedMethodology$new()`
- `ProbBasedMethodology$compute()`
- `ProbBasedMethodology$clone()`

Method `new()`**:** Method for initializing the object arguments during runtime.

Usage:

```
ProbBasedMethodology$new(required.metrics = c("MCC", "PPV"))
```

Arguments:

`required.metrics` A character vector of length greater than 2 with the name of the required metrics.

Method `compute()`**:** Function to compute the probability of the final prediction based on different metrics.

Usage:

```
ProbBasedMethodology$compute(
  raw.pred,
  prob.pred,
  positive.class,
  negative.class
)
```

Arguments:

`raw.pred` A character list of length greater than 2 with the class value of the predictions made by the metrics.

`prob.pred` A numeric list of length greater than 2 with the probability of the predictions made by the metrics.

`positive.class` A character with the value of the positive class.

`negative.class` A character with the value of the negative class.

Returns: A numeric value indicating the probability of the instance is predicted as positive class.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
ProbBasedMethodology$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Methodology

`Recall` *Computes the Recall Value.*

Description

Recall (also known as sensitivity) is the fraction of the total amount of relevant instances that were actually retrieved.

Details

$$recall = TP/(TP + FN)$$

Super class

D2MCS::MeasureFunction -> Recall

Methods

Public methods:

- `Recall$new()`
- `Recall$compute()`
- `Recall$clone()`

Method `new()`**:** Method for initializing the object arguments during runtime.

Usage:

```
Recall$new(performance.output = NULL)
```

Arguments:

`performance.output` An optional `ConfMatrix` parameter to define the type of object used as basis to compute the measure.

Method `compute()`**:** The function computes the **Recall** achieved by the M.L. model.

Usage:

```
Recall$compute(performance.output = NULL)
```

Arguments:

`performance.output` An optional `ConfMatrix` parameter to define the type of object used as basis to compute the **Recall** measure.

Details: This function is automatically invoke by the `ClassificationOutput` object.

Returns: A numeric vector of size 1 or NULL if an error occurred.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Recall$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`MeasureFunction`, `ClassificationOutput`, `ConfMatrix`

`Sensitivity` *Computes the Sensitivity Value.*

Description

Sensitivity is a measure of the proportion of actual positive cases that got predicted as positive (or true positive).

Details

$$Sensitivity = TP/(TP + FN)$$

Super class

D2MCS::MeasureFunction -> Sensitivity

Methods

Public methods:

- Sensitivity$new()
- Sensitivity$compute()
- Sensitivity$clone()

Method new(): Method for initializing the object arguments during runtime.

Usage:

```
Sensitivity$new(performance.output = NULL)
```

Arguments:

performance.output An optional ConfMatrix parameter to define the type of object used as basis to compute the Sensitivity measure.

Method compute(): The function computes the **Sensitivity** achieved by the M.L. model.

Usage:

```
Sensitivity$compute(performance.output = NULL)
```

Arguments:

performance.output An optional ConfMatrix parameter to define the type of object used as basis to compute the **Sensitivity** measure.

Details: This function is automatically invoke by the ClassificationOutput object.

Returns: A numeric vector of size 1 or NULL if an error occurred.

Method clone(): The objects of this class are cloneable with this method.

Usage:

```
Sensitivity$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

MeasureFunction, ClassificationOutput, ConfMatrix

SimpleStrategy *Simple feature clustering strategy.*

Description

Features are sorted by descendant according to the relevance value obtained after applying an specific heuristic. Next, features are distributed into N clusters following a card-dealing methodology. Finally best distribution is assigned to the distribution having highest homogeneity.

Details

The strategy is suitable for all features that are valid for the indicated heuristics. Invalid features are automatically grouped into a specific cluster named as 'unclustered'.

Super class

`D2MCS::GenericClusteringStrategy` -> `SimpleStrategy`

Methods

Public methods:

- `SimpleStrategy$new()`
- `SimpleStrategy$execute()`
- `SimpleStrategy$getBestClusterDistribution()`
- `SimpleStrategy$getUnclustered()`
- `SimpleStrategy$getDistribution()`
- `SimpleStrategy$createTrain()`
- `SimpleStrategy$plot()`
- `SimpleStrategy$saveCSV()`
- `SimpleStrategy$clone()`

Method `new()`**:** Method for initializing the object arguments during runtime.

Usage:

```
SimpleStrategy$new(
  subset,
  heuristic,
  configuration = StrategyConfiguration$new()
)
```

Arguments:

`subset` The `Subset` used to apply the feature-clustering strategy.

`heuristic` The heuristic used to compute the relevance of each feature. Must inherit from `GenericHeuristic` abstract class.

`configuration` Optional parameter to customize configuration parameters for the strategy. Must inherited from `StrategyConfiguration` abstract class.

Method `execute()`**:** Function responsible of performing the clustering strategy over the defined `Subset`.

Usage:

```
SimpleStrategy$execute(verbose = FALSE)
```

Arguments:

`verbose` A logical value to specify if more verbosity is needed.

Method `getBestClusterDistribution()`**:** The function obtains the best clustering distribution.

Usage:

```
SimpleStrategy$getBestClusterDistribution()
```

Returns: A list of clusters. Each list element represents a feature group.

Method `getUnclustered()`**:** The function is used to return the features that cannot be clustered due to incompatibilities with the used heuristic.

Usage:

```
SimpleStrategy$getUnclustered()
```

Returns: A character vector containing the unclassified features.

Method `getDistribution()`**:** Function used to obtain a specific cluster distribution.

Usage:

```
SimpleStrategy$getDistribution(
  num.clusters = NULL,
  num.groups = NULL,
  include.unclustered = FALSE
)
```

Arguments:

`num.clusters` A numeric value to select the number of clusters (define the distribution).

`num.groups` A single or numeric vector value to identify a specific group that forms the clustering distribution.

`include.unclustered` A logical value to determine if unclustered features should be included.

Returns: A list with the features comprising an specific clustering distribution.

Method `createTrain()`**:** The function is used to create a `Trainset` object from a specific clustering distribution.

Usage:

```
SimpleStrategy$createTrain(
  subset,
  num.clusters = NULL,
  num.groups = NULL,
  include.unclustered = FALSE
)
```

Arguments:

`subset` The `Subset` object used as a basis to create the train set (see `Trainset` class).

`num.clusters` A numeric value to select the number of clusters (define the distribution).

`num.groups` A single or numeric vector value to identify a specific group that forms the clustering distribution.

`include.unclustered` A logical value to determine if unclustered features should be included.

Details: If `num.clusters` and `num.groups` are not defined, best clustering distribution is used to create the train set.

Returns: A `Trainset` object.

Method `plot()`**:** The function is responsible for creating a plot to visualize the clustering distribution.

Usage:

```
SimpleStrategy$plot(dir.path = NULL, file.name = NULL)
```

Arguments:

`dir.path` An optional argument to define the name of the directory where the exported plot will be saved. If not defined, the file path will be automatically assigned to the current working directory, '`getwd()`'.

`file.name` A character to define the name of the PDF file where the plot is exported.

Method `saveCSV()`**:** The function is used to save the clustering distribution to a CSV file.

Usage:

```
SimpleStrategy$saveCSV(dir.path, name = NULL, num.clusters = NULL)
```

Arguments:

`dir.path` The name of the directory to save the CSV file.

`name` Defines the name of the CSV file.

`num.clusters` An optional parameter to select the number of clusters to be saved. If not defined, all cluster distributions will be saved.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
SimpleStrategy$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`GenericClusteringStrategy`, `StrategyConfiguration`

SimpleVoting *Abtract class to define simple voting schemes.*

Description

Abstract class used as a template to define new customized simple voting schemes.

Methods

Public methods:

- SimpleVoting$new()
- SimpleVoting$getCutoff()
- SimpleVoting$getFinalPred()
- SimpleVoting$execute()
- SimpleVoting$clone()

Method `new()`**:** Method for initializing the object arguments during runtime.

Usage:

```
SimpleVoting$new(cutoff = NULL)
```

Arguments:

`cutoff` A character vector defining the minimum probability used to perform a positive classification. If is not defined, 0.5 will be used as default value.

Method `getCutoff()`**:** The function obtains the minimum probabilistic value used to perform a positive classification.

Usage:

```
SimpleVoting$getCutoff()
```

Returns: A numeric value.

Method `getFinalPred()`**:** The function is used to return the prediction values computed by a voting strategy.

Usage:

```
SimpleVoting$getFinalPred(type = NULL, target = NULL, filter = NULL)
```

Arguments:

`type` A character to define which type of predictions should be returned. If not defined all type of probabilities will be returned. Conversely if 'prob' or 'raw' is defined then computed 'probabilistic' or 'class' values are returned.

`target` A character defining the value of the positive class.

`filter` A logical value used to specify if only predictions matching the target value should be returned or not. If TRUE the function returns only the predictions matching the target value. Conversely if FALSE (by default) the function returns all the predictions.

Returns: A FinalPred object.

Method `execute()`**:** Abstract function used to implement the operation of the voting scheme.

Usage:

```
SimpleVoting$execute(predictions, verbose = FALSE)
```

Arguments:

`predictions` A `ClusterPredictions` object containing all the predictions achieved for each cluster.

`verbose` A logical value to specify if more verbosity is needed.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
SimpleVoting$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`D2MCS`, `ClassMajorityVoting`, `ClassWeightedVoting`, `ProbAverageVoting`, `ProbAverageWeightedVoting`, `ProbBasedMethodology`, `CombinedVoting`

`SingleVoting` *Manages the execution of Simple Votings.*

Description

The class is responsible of initializing and executing voting schemes. Additionally, to ensure a proper operation, the class automatically checks the compatibility of defined voting schemes.

Super class

`D2MCS::VotingStrategy` -> `SingleVoting`

Methods

Public methods:

- `SingleVoting$new()`
- `SingleVoting$execute()`
- `SingleVoting$clone()`

Method `new()`**:** The function initializes the object arguments during runtime.

Usage:

```
SingleVoting$new(voting.schemes, metrics)
```

Arguments:

`voting.schemes` A vector of voting schemes inheriting from `SimpleVoting` class.

`metrics` A list containing the metrics used as basis to perform the voting strategy.

Method `execute()`**:** The function is used to execute all the previously defined (and compatible) voting schemes.

Usage:

```
SingleVoting$execute(predictions, verbose = FALSE)
```

Arguments:

`predictions` A `ClusterPredictions` object containing all the predictions computed in the classification stage.

`verbose` A logical value to specify if more verbosity is needed.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
SingleVoting$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`D2MCS`, `SimpleVoting`, `CombinedVoting`

`SpearmanHeuristic` *Feature-clustering based on Spearman Correlation Test.*

Description

Performs the feature-clustering using Spearman's rho statistic.

Details

Spearman's rho statistic is to estimate a rank-based measure of association. These tests may be used if the data do not necessarily come from a bivariate normal distribution.

Super class

`D2MCS::GenericHeuristic` -> `SpearmanHeuristic`

Methods

Public methods:

- `SpearmanHeuristic$new()`
- `SpearmanHeuristic$heuristic()`
- `SpearmanHeuristic$clone()`

Method `new()`**:** Creates a SpearmanHeuristic object.

Usage:

```
SpearmanHeuristic$new()
```

Method `heuristic()`**:** Test for correlation between paired samples using Spearman rho statistic.

Usage:

```
SpearmanHeuristic$heuristic(col1, col2, column.names = NULL)
```

Arguments:

`col1` A numeric vector or matrix required to perform the clustering operation.

`col2` A numeric vector or matrix to perform the clustering operation.

`column.names` An optional character vector with the names of both columns.

Returns: A numeric vector of length 1 or NA if an error occurs.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
SpearmanHeuristic$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`Dataset`, `cor.test`

`Specificity` *Computes the Specificity Value.*

Description

Specificity is defined as the proportion of actual negatives, which got predicted as the negative (or true negative). This implies that there will be another proportion of actual negative, which got predicted as positive and could be termed as false positives.

Details

$$Specificity = TrueNegative/(TrueNegative + FalsePositive)$$

Super class

`D2MCS::MeasureFunction` -> `Specificity`

Methods

Public methods:

- `Specificity$new()`
- `Specificity$compute()`
- `Specificity$clone()`

Method `new()`**:** Method for initializing the object arguments during runtime.

Usage:

```
Specificity$new(performance.output = NULL)
```

Arguments:

`performance.output` An optional `ConfMatrix` parameter to define the type of object used as basis to compute the measure.

Method `compute()`**:** The function computes the **Specificity** achieved by the M.L. model.

Usage:

```
Specificity$compute(performance.output = NULL)
```

Arguments:

`performance.output` An optional `ConfMatrix` parameter to define the type of object used as basis to compute the **Specificity** measure.

Details: This function is automatically invoke by the `ClassificationOutput` object.

Returns: A numeric vector of size 1 or NULL if an error occurred.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Specificity$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`MeasureFunction`, `ClassificationOutput`, `ConfMatrix`

`StrategyConfiguration` *Default Strategy Configuration handler.*

Description

Define default configuration parameters for the clustering strategies.

Details

The `StrategyConfiguration` can be used to define the default configuration parameters for a feature clustering strategy or as an archetype to define new customized parameters.

Methods

Public methods:

- StrategyConfiguration$new()
- StrategyConfiguration$minNumClusters()
- StrategyConfiguration$maxNumClusters()
- StrategyConfiguration$clone()

Method `new()`**:** Empty function used to initialize the object arguments in runtime.

Usage:

```
StrategyConfiguration$new()
```

Method `minNumClusters()`**:** Function used to return the minimum number of clusters distributions used. By default the minimum is set in 2.

Usage:

```
StrategyConfiguration$minNumClusters(...)
```

Arguments:

`...` Further arguments passed down to `minNumClusters` function.

Returns: A numeric vector of length 1.

Method `maxNumClusters()`**:** The function is responsible of returning the maximum number of cluster distributions used. By default the maximum number is set in 50.

Usage:

```
StrategyConfiguration$maxNumClusters(...)
```

Arguments:

`...` Further arguments passed down to `maxNumClusters` function.

Returns: A numeric vector of length 1.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
StrategyConfiguration$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

DependencyBasedStrategyConfiguration

`Subset` *Classification set.*

Description

The `Subset` is used for testing or classification purposes. If a target class is defined the `Subset` can be used as test and classification, otherwise the `Subset` only classification is compatible.

Details

Use `Dataset` to ensure the creation of a valid `Subset` object.

Methods

Public methods:

- `Subset$new()`
- `Subset$getColumnNames()`
- `Subset$getFeatures()`
- `Subset$getID()`
- `Subset$getIterator()`
- `Subset$getClassValues()`
- `Subset$getClassBalance()`
- `Subset$getClassIndex()`
- `Subset$getClassName()`
- `Subset$getNcol()`
- `Subset$getNrow()`
- `Subset$getPositiveClass()`
- `Subset$isBlinded()`

Method `new()`**:** Method for initializing the object arguments during runtime.

Usage:

```
Subset$new(
  dataset,
  class.index = NULL,
  class.values = NULL,
  positive.class = NULL,
  feature.id = NULL
)
```

Arguments:

`dataset` A fully filled data.frame.

`class.index` A numeric value identifying the column representing the target class

`class.values` A character vector containing all the values of the target class.

`positive.class` A character value representing the positive class value.

`feature.id` A numeric value specifying the column number used as identifier.

Method `getColumnNames()`**:** Get the name of the columns comprising the subset.

Usage:

```
Subset$getColumnNames()
```

Returns: A character vector containing the name of each column.

Method `getFeatures()`**:** Gets the values of all features or those indicated by arguments.

Usage:

```
Subset$getFeatures(feature.names = NULL)
```

Arguments:

`feature.names` A character vector comprising the name of the features to be obtained.

Returns: A character vector or NULL if subset is empty.

Method `getID()`**:** Gets the column name used as identifier.

Usage:

```
Subset$getID()
```

Returns: A character vector of size 1 of NULL if column id is not defined.

Method `getIterator()`**:** Creates the DIterator object.

Usage:

```
Subset$getIterator(chunk.size = private$chunk.size, verbose = FALSE)
```

Arguments:

`chunk.size` An integer value indicating the size of chunks taken over each iteration. By default chunk.size is defined as 10000.

`verbose` A logical value to specify if more verbosity is needed.

Returns: A `DIterator` object to transverse through `Subset` instances.

Method `getClassValues()`**:** Gets all the values of the target class.

Usage:

```
Subset$getClassValues()
```

Returns: A factor vector with all the values of the target class.

Method `getClassBalance()`**:** The function is used to compute the ratio of each class value in the `Subset`.

Usage:

```
Subset$getClassBalance(target.value = NULL)
```

Arguments:

`target.value` The class value used as reference to perform the comparison.

Returns: A numeric value.

Method `getClassIndex()`**:** The function is used to obtain the index of the column containing the target class.

Usage:

```
Subset$getClassIndex()
```

Returns: A numeric value.

Method `getClassName()`**:** The function is used to specify the name of the column containing the target class.

Usage:

```
Subset$getClassName()
```

Returns: A character value.

Method `getNcol()`**:** The function is in charge of obtaining the number of columns comprising the `Subset`. See `ncol` for more information.

Usage:

```
Subset$getNcol()
```

Returns: An integer of length 1 or NULL.

Method `getNrow()`**:** The function is used to determine the number of rows present in the `Subset`. See `nrow` for more information.

Usage:

```
Subset$getNrow()
```

Returns: An integer of length 1 or NULL.

Method `getPositiveClass()`**:** The function returns the value of the positive class.

Usage:

```
Subset$getPositiveClass()
```

Returns: A character vector of size 1 or NULL if not defined.

Method `isBlinded()`**:** The function is used to check if the Subset contains a target class.

Usage:

```
Subset$isBlinded()
```

Returns: A logical value where TRUE represents the absence of target class and FALSE its presence.

See Also

`Dataset`, `DatasetLoader`, `Trainset`

SummaryFunction *Abstract class to computing performance across resamples.*

Description

Abstract used as template to define customized metrics to compute model performance during train.

Details

This class is an archetype, so it cannot be instantiated.

Methods

Public methods:

- SummaryFunction$new()
- SummaryFunction$execute()
- SummaryFunction$getMeasures()
- SummaryFunction$clone()

Method `new()`**:** The function carries out the initialization of parameters during runtime.

Usage:

```
SummaryFunction$new(measures)
```

Arguments:

`measures` A character vector with the measures used.

Method `execute()`**:** Abstract function used to implement the performance calculator method. To guarantee a proper operation, this method is automatically invoked by D2MCS framework.

Usage:

```
SummaryFunction$execute()
```

Method `getMeasures()`**:** The function obtains the measures used to compute the performance across resamples.

Usage:

```
SummaryFunction$getMeasures()
```

Returns: A character vector of NULL if measures are not defined.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
SummaryFunction$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

NoProbability, UseProbability

TN *Computes the True Negative value.*

Description

This is the number of individuals with a negative condition for which the test result is negative. The value entered here must be non-negative.

Super class

`D2MCS::MeasureFunction` -> `TN`

Methods

Public methods:

- `TN$new()`
- `TN$compute()`
- `TN$clone()`

Method `new()`**:** Method for initializing the object arguments during runtime.

Usage:

```
TN$new(performance.output = NULL)
```

Arguments:

`performance.output` An optional `ConfMatrix` parameter to define the type of object used to compute the **TN** measure.

Method `compute()`**:** The function computes the **TN** achieved by the M.L. model.

Usage:

```
TN$compute(performance.output = NULL)
```

Arguments:

`performance.output` An optional `ConfMatrix` parameter to define the type of object used as basis to compute the **TN** measure.

Details: This function is automatically invoke by the `ClassificationOutput` object.

Returns: A numeric vector of size 1 or NULL if an error occurred.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
TN$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`MeasureFunction`, `ClassificationOutput`, `ConfMatrix`

TP *Computes the True Positive Value.*

Description

TP is the number of individuals with a positive condition for which the test result is positive. The value entered here must be non-negative.

Super class

`D2MCS::MeasureFunction` -> `TP`

Methods

Public methods:

- `TP$new()`
- `TP$compute()`
- `TP$clone()`

Method `new()`**:** Method for initializing the object arguments during runtime.

Usage:

```
TP$new(performance.output = NULL)
```

Arguments:

`performance.output` An optional `ConfMatrix` parameter to define the type of object used to compute the measure.

Method `compute()`**:** The function computes the **TP** achieved by the M.L. model.

Usage:

```
TP$compute(performance.output = NULL)
```

Arguments:

`performance.output` An optional `ConfMatrix` parameter to define the type of object used as basis to compute the **TP** measure.

Details: This function is automatically invoke by the `ClassificationOutput` object.

Returns: A numeric vector of size 1 or NULL if an error occurred.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
TP$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`MeasureFunction`, `ClassificationOutput`, `ConfMatrix`

TrainFunction *Control parameters for train stage.*

Description

Abstract class used as template to define customized functions to control the computational nuances of train function.

Methods

Public methods:

- TrainFunction$new()
- TrainFunction$create()
- TrainFunction$getResamplingMethod()
- TrainFunction$getNumberFolds()
- TrainFunction$getSavePredictions()
- TrainFunction$getClassProbs()
- TrainFunction$getAllowParallel()
- TrainFunction$getVerboseIter()
- TrainFunction$getTrFunction()
- TrainFunction$getMeasures()
- TrainFunction$getType()
- TrainFunction$getSeed()
- TrainFunction$setSummaryFunction()
- TrainFunction$setClassProbs()
- TrainFunction$clone()

Method `new()`**:** Function used to initialize the object parameters during execution time.

Usage:

```
TrainFunction$new(
  method,
  number,
  savePredictions,
  classProbs,
  allowParallel,
  verboseIter,
  seed
)
```

Arguments:

`method` The resampling method: "boot", "boot632", "optimism_boot", "boot_all", "cv", "repeatedcv", "LOOCV", "LGOCV" (for repeated training/test splits), "none" (only fits one model to the entire training set), "oob" (only for random forest, bagged trees, bagged earth, bagged flexible discriminant analysis, or conditional tree forest models), timeslice, "adaptive_cv", "adaptive_boot" or "adaptive_LGOCV"

`number` Either the number of folds or number of resampling iterations

`savePredictions` An indicator of how much of the hold-out predictions for each resample should be saved. Values can be either "all", "final", or "none". A logical value can also be used that convert to "all" (for true) or "none" (for false). "final" saves the predictions for the optimal tuning parameters.

`classProbs` A logical value. Should class probabilities be computed for classification models (along with predicted values) in each resample?

`allowParallel` A logical value. If a parallel backend is loaded and available, should the function use it?

`verboseIter` A logical for printing a training log.

`seed` An optional integer that will be used to set the seed during model training stage.

Method `create()`**:** Creates a `trainControl` requires for the training stage.

Usage:

```
TrainFunction$create(summaryFunction, search.method = "grid", class.probs)
```

Arguments:

`summaryFunction` An object inherited from `SummaryFunction` class.

`search.method` Either "grid" or "random", describing how the tuning parameter grid is determined.

`class.probs` A logical indicating if class probabilities should be computed for classification models (along with predicted values) in each resample.

Method `getResamplingMethod()`**:** Returns the resampling method used during training staged.

Usage:

```
TrainFunction$getResamplingMethod()
```

Returns: A character vector or length 1 or NULL if not defined.

Method `getNumberFolds()`**:** Returns the number or folds or number of iterations used during training.

Usage:

```
TrainFunction$getNumberFolds()
```

Returns: An integer vector or length 1 or NULL if not defined.

Method `getSavePredictions()`**:** Indicates if the predictions for each resample should be saved.

Usage:

```
TrainFunction$getSavePredictions()
```

Returns: A logical value or NULL if not defined.

Method `getClassProbs()`**:** Indicates if class probabilities should be computed for classification models in each resample.

Usage:

```
TrainFunction$getClassProbs()
```

Returns: A logical value.

Method `getAllowParallel()`**:** Determines if model training is performed in parallel.

Usage:

```
TrainFunction$getAllowParallel()
```

Returns: A logical value. TRUE indicates parallelization is enabled and FALSE otherwise.

Method `getVerboseIter()`**:** Determines if training log should be printed.

Usage:

```
TrainFunction$getVerboseIter()
```

Returns: A logical value. TRUE indicates training log is enabled and FALSE otherwise.

Method `getTrFunction()`**:** Function used to return the `trainControl` object.

Usage:

```
TrainFunction$getTrFunction()
```

Returns: A `trainControl` object.

Method `getMeasures()`**:** Returns the measures used to optimize model hyperparameters.

Usage:

```
TrainFunction$getMeasures()
```

Returns: A character vector.

Method `getType()`**:** Obtains the type of classification problem ("Bi-class" or "Multi-class").

Usage:

```
TrainFunction$getType()
```

Returns: A character vector with length 1. Either "Bi-class" or "Multi-class".

Method `getSeed()`**:** Indicates seed used during model training stage.

Usage:

```
TrainFunction$getSeed()
```

Returns: An integer value or NULL if not defined.

Method `setSummaryFunction()`**:** Function used to change the `SummaryFunction` used in the training stage.

Usage:

```
TrainFunction$setSummaryFunction(summaryFunction)
```

Arguments:

`summaryFunction` An object inherited from `SummaryFunction` class.

Method `setClassProbs()`**:** The function allows changing the class computation capabilities.

Usage:

```
TrainFunction$setClassProbs(class.probs)
```

Arguments:

`class.probs` A logical indicating if class probabilities should be computed for classification models (along with predicted values) in each resample

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
TrainFunction$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`TwoClass`

`TrainOutput` *Stores the results achieved during training.*

Description

This class manages the results achieved during training stage (such as optimized hyperparameters, model information, utilized metrics).

Methods

Public methods:

- `TrainOutput$new()`
- `TrainOutput$getModels()`
- `TrainOutput$getPerformance()`
- `TrainOutput$savePerformance()`
- `TrainOutput$plot()`
- `TrainOutput$getMetrics()`
- `TrainOutput$getClassValues()`
- `TrainOutput$getPositiveClass()`
- `TrainOutput$getSize()`
- `TrainOutput$clone()`

Method `new()`**:** Function used to initialize the object arguments during runtime.

Usage:

```
TrainOutput$new(models, class.values, positive.class)
```

Arguments:

`models` A list containing the best M.L. model for each cluster.

`class.values` A character vector containing the values of the target class.

`positive.class` A character with the value of the positive class.

Method `getModels()`**:** The function is used to obtain the best M.L. model of each cluster.

Usage:

```
TrainOutput$getModels(metric)
```

Arguments:

`metric` A character vector which specifies the metric(s) used for configuring M.L. hyperparameters.

Returns: A list is returned of class train.

Method `getPerformance()`**:** The function returns the performance value of M.L. models during training stage.

Usage:

```
TrainOutput$getPerformance(metrics = NULL)
```

Arguments:

`metrics` A character vector which specifies the metric(s) used to train the M.L. models.

Returns: A character vector containing the metrics used for configuring M.L. hyperparameters.

Method `savePerformance()`**:** The function is used to save into CSV file the performance achieved by the M.L. models during training stage.

Usage:

```
TrainOutput$savePerformance(dir.path, metrics = NULL)
```

Arguments:

`dir.path` The location to store the into a CSV file the performance of the trained M.L.

`metrics` An optional parameter specifying the metric(s) used to train the M.L. models. If not defined, all the metrics used in train stage will be saved.

Method `plot()`**:** The function is responsible for creating a plot to visualize the performance achieved by the best M.L. model on each cluster.

Usage:

```
TrainOutput$plot(dir.path, metrics = NULL)
```

Arguments:

`dir.path` The location to store the exported plot will be saved.

`metrics` An optional parameter specifying the metric(s) used to train the M.L. models. If not defined, all the metrics used in train stage will be plotted.

Method `getMetrics()`**:** The function returns all metrics used for configuring M.L. hyperparameters during train stage.

Usage:

```
TrainOutput$getMetrics()
```

Returns: A character value.

Method `getClassValues()`**:** The function is used to get the values of the target class.

Usage:

```
TrainOutput$getClassValues()
```

Returns: A character containing the values of the target class.

Method `getPositiveClass()`**:** The function returns the value of the positive class.

Usage:

```
TrainOutput$getPositiveClass()
```

Returns: A character vector of size 1.

Method `getSize()`**:** The function is used to get the number of the trained M.L. models. Each cluster contains the best M.L. model.

Usage:

```
TrainOutput$getSize()
```

Returns: A numeric value or NULL training was not successfully performed.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
TrainOutput$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`D2MCS`

`Trainset` *Trainning set.*

Description

The `Trainset` is used to perform training operations over M.L. models. A target class should be defined to guarantee a full compatibility with supervised models.

Details

Use `Dataset` object to ensure the creation of a valid `Trainset` object.

Methods

Public methods:

- `Trainset$new()`
- `Trainset$getPositiveClass()`
- `Trainset$getClassName()`
- `Trainset$getClassValues()`
- `Trainset$getColumnNames()`
- `Trainset$getFeatureValues()`
- `Trainset$getInstances()`
- `Trainset$getNumClusters()`

Method `new()`**:** Method for initializing the object arguments during runtime.

Usage:

```
Trainset$new(cluster.dist, class.name, class.values, positive.class)
```

Arguments:

`cluster.dist` The type of cluster distribution used as basis to build the `Trainset`. See `GenericClusteringStrategy` for more information.

`class.name` Used to specify the name of the column containing the target class.

`class.values` Specifies all the possible values of the target class.

`positive.class` A character with the value of the positive class.

Method `getPositiveClass()`**:** The function is used to obtain the value of the positive class.

Usage:

```
Trainset$getPositiveClass()
```

Returns: A numeric value with the positive class value.

Method `getClassName()`**:** The function is used to return the name of the target class.

Usage:

```
Trainset$getClassName()
```

Returns: A character vector with length 1.

Method `getClassValues()`**:** The function is used to compute all the possible target class values.

Usage:

```
Trainset$getClassValues()
```

Returns: A factor value.

Method `getColumnNames()`**:** The function returns the name of the columns comprising an specific cluster distribution.

Usage:

```
Trainset$getColumnNames(num.cluster)
```

Arguments:

`num.cluster` A numeric value used to specify the cluster number of the cluster distribution used when creating the `Trainset`.

Returns: A character vector with all column names.

Method `getFeatureValues()`**:** The function returns the values of the columns comprising an specific cluster distribution. Target class is omitted.

Usage:

```
Trainset$getFeatureValues(num.cluster)
```

Arguments:

`num.cluster` A numeric value used to specify the cluster number of the cluster distribution used when creating the `Trainset`.

Returns: A data.frame with the values of the features comprising the selected cluster distribution.

Method `getInstances()`**:** The function returns the values of the columns comprising an specific cluster distribution. Target class is included as the last column.

Usage:

```
Trainset$getInstances(num.cluster)
```

Arguments:

`num.cluster` A numeric value used to specify the cluster number of the cluster distribution used when creating the `Trainset`.

Returns: A data.frame with the values of the features comprising the selected cluster distribution.

Method `getNumClusters()`**:** The function obtains the number of groups (clusters) that forms the cluster distribution.

Usage:

```
Trainset$getNumClusters()
```

Returns: A numeric vector of size 1.

See Also

`Dataset`, `DatasetLoader`, `Subset`, `GenericClusteringStrategy`

`TwoClass` *Control parameters for train stage (Bi-class problem).*

Description

Implementation to control the computational nuances of train function for bi-class problems.

Super class

`D2MCS::TrainFunction` -> `TwoClass`

Methods

Public methods:

- `TwoClass$new()`
- `TwoClass$create()`
- `TwoClass$getTrFunction()`
- `TwoClass$setClassProbs()`
- `TwoClass$getMeasures()`
- `TwoClass$getType()`
- `TwoClass$setSummaryFunction()`
- `TwoClass$clone()`

Method `new()`**:**

Usage:

```
TwoClass$new(
  method,
  number,
  savePredictions,
  classProbs,
  allowParallel,
  verboseIter,
  seed = NULL
)
```

Arguments:

`method` The resampling method: "boot", "boot632", "optimism_boot", "boot_all", "cv", "repeatedcv", "LOOCV", "LGOCV" (for repeated training/test splits), "none" (only fits one model to the entire training set), "oob" (only for random forest, bagged trees, bagged earth, bagged flexible discriminant analysis, or conditional tree forest models), timeslice, "adaptive_cv", "adaptive_boot" or "adaptive_LGOCV"

`number` Either the number of folds or number of resampling iterations

`savePredictions` An indicator of how much of the hold-out predictions for each resample should be saved. Values can be either "all", "final", or "none". A logical value can also be used that convert to "all" (for true) or "none" (for false). "final" saves the predictions for the optimal tuning parameters.

`classProbs` A logical value. Should class probabilities be computed for classification models (along with predicted values) in each resample?

`allowParallel` A logical value. If a parallel backend is loaded and available, should the function use it?

`verboseIter` A logical for printing a training log.

`seed` An optional integer that will be used to set the seed during model training stage.

Method `create()`**:** Creates a `trainControl` requires for the training stage.

Usage:

```
TwoClass$create(summaryFunction, search.method = "grid", class.probs = NULL)
```

Arguments:

`summaryFunction` An object inherited from `SummaryFunction` class.

`search.method` Either "grid" or "random", describing how the tuning parameter grid is determined.

`class.probs` A logical indicating if class probabilities should be computed for classification models (along with predicted values) in each resample

Method `getTrFunction()`**:** Function used to return the `trainControl` object.

Usage:

```
TwoClass$getTrFunction()
```

Returns: A `trainControl` object.

Method `setClassProbs()`**:** The function allows changing the class computation capabilities.

Usage:

```
TwoClass$setClassProbs(class.probs)
```

Arguments:

`class.probs` A logical value. TRUE implies classification probabilities should be computed for classification models and FALSE otherwise.

Method `getMeasures()`**:** Returns the measures used to optimize model hyperparameters.

Usage:

```
TwoClass$getMeasures()
```

Returns: A character vector.

Method `getType()`**:** Obtains the type of classification problem ("Bi-class" or "Multi-class").

Usage:

```
TwoClass$getType()
```

Returns: A character vector with "Bi-class" value.

Method `setSummaryFunction()`**:** Function used to change the `SummaryFunction` used in the training stage.

Usage:

```
TwoClass$setSummaryFunction(summaryFunction)
```

Arguments:

`summaryFunction` An object inherited from `SummaryFunction` class.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
TwoClass$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`TrainFunction`

`TypeBasedStrategy` *Feature clustering strategy.*

Description

Features are sorted by descendant according to the relevance value obtained after applying an specific heuristic. Next, features are distributed into N clusters following a card-dealing methodology. Finally best distribution is assigned to the distribution having highest homogeneity.

Details

The strategy is suitable only for binary and real features. Other features are automatically grouped into a specific cluster named as 'unclustered'.

Super class

D2MCS::GenericClusteringStrategy -> TypeBasedStrategy

Methods

Public methods:

- TypeBasedStrategy$new()
- TypeBasedStrategy$execute()
- TypeBasedStrategy$getDistribution()
- TypeBasedStrategy$createTrain()
- TypeBasedStrategy$plot()
- TypeBasedStrategy$saveCSV()
- TypeBasedStrategy$clone()

Method `new()`**:** Method for initializing the object arguments during runtime.

Usage:

```
TypeBasedStrategy$new(
  subset,
  heuristic,
  configuration = StrategyConfiguration$new()
)
```

Arguments:

`subset` The Subset used to apply the feature-clustering strategy.

`heuristic` The heuristic used to compute the relevance of each feature. Must inherit from GenericHeuristic abstract class.

`configuration` Optional parameter to customize configuration parameters for the strategy. Must inherited from StrategyConfiguration abstract class.

Method `execute()`**:** Function responsible of performing the clustering strategy over the defined Subset.

Usage:

```
TypeBasedStrategy$execute(verbose = FALSE)
```

Arguments:

`verbose` A logical value to specify if more verbosity is needed.

Method `getDistribution()`**:** Function used to obtain a specific cluster distribution.

Usage:

```
TypeBasedStrategy$getDistribution(
  num.clusters = NULL,
  num.groups = NULL,
  include.unclustered = FALSE
)
```

Arguments:

`num.clusters` A numeric value to select the number of clusters (define the distribution).

num.groups A single or numeric vector value to identify a specific group that forms the clustering distribution.

include.unclustered A logical value to determine if unclustered features should be included.

Returns: A list with the features comprising an specific clustering distribution.

Method `createTrain()`**:** The function is used to create a Trainset object from a specific clustering distribution.

Usage:

```
TypeBasedStrategy$createTrain(
  subset,
  num.clusters = NULL,
  num.groups = NULL,
  include.unclustered = FALSE
)
```

Arguments:

subset The Subset object used as a basis to create the train set (see Trainset class).

num.clusters A numeric value to select the number of clusters (define the distribution).

num.groups A single or numeric vector value to identify a specific group that forms the clustering distribution.

include.unclustered A logical value to determine if unclustered features should be included.

Details: If num.clusters and num.groups are not defined, best clustering distribution is used to create the train set.

Returns: A Trainset object.

Method `plot()`**:** The function is responsible for creating a plot to visualize the clustering distribution.

Usage:

```
TypeBasedStrategy$plot(dir.path = NULL, file.name = NULL)
```

Arguments:

dir.path An optional character argument to define the name of the directory where the exported plot will be saved. If not defined, the file path will be automatically assigned to the current working directory, 'getwd()'.

file.name A character to define the name of the PDF file where the plot is exported.

Method `saveCSV()`**:** The function is used to save the clustering distribution to a CSV file.

Usage:

```
TypeBasedStrategy$saveCSV(dir.path = NULL, name = NULL, num.clusters = NULL)
```

Arguments:

dir.path The name of the directory to save the CSV file.

name Defines the name of the CSV file.

num.clusters An optional parameter to select the number of clusters to be saved. If not defined, all cluster distributions will be saved.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
TypeBasedStrategy$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`GenericClusteringStrategy`, `StrategyConfiguration`

`UseProbability` *Compute performance across resamples.*

Description

Computes the performance across resamples when class probabilities can be computed.

Super class

`D2MCS::SummaryFunction` -> `UseProbability`

Methods

Public methods:

- `UseProbability$new()`
- `UseProbability$execute()`
- `UseProbability$clone()`

Method `new()`**:** The function defined during runtime the usage of seven measures: 'ROC', 'Sens', 'Kappa', 'Accuracy', 'TCR_9', 'MCC' and 'PPV'.

Usage:

```
UseProbability$new()
```

Method `execute()`**:** The function computes the performance across resamples using the previously defined measures.

Usage:

```
UseProbability$execute(data, lev = NULL, model = NULL)
```

Arguments:

`data` A data.frame containing the data used to compute the performance.

`lev` An optional value used to define the levels of the target class.

`model` An optional value used to define the M.L. model used.

Returns: A vector of performance estimates.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
UseProbability$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

SummaryFunction

VotingStrategy *Voting Strategy template.*

Description

Abstract class used to define new SingleVoting and CombinedVoting schemes.

Methods

Public methods:

- VotingStrategy$new()
- VotingStrategy$getVotingSchemes()
- VotingStrategy$getMetrics()
- VotingStrategy$execute()
- VotingStrategy$getName()
- VotingStrategy$clone()

Method `new()`**:** Abstract method used to initialize the object arguments during runtime.

Usage:

```
VotingStrategy$new()
```

Method `getVotingSchemes()`**:** The function returns the voting schemes that will participate in the voting strategy.

Usage:

```
VotingStrategy$getVotingSchemes()
```

Returns: A vector of object inheriting from VotingStrategy class.

Method `getMetrics()`**:** The function is used to get the metric that will be used during the voting strategy.

Usage:

```
VotingStrategy$getMetrics()
```

Returns: A character vector.

Method `execute()`**:** Abstract function used to implement the operation of the voting schemes.

Usage:

```
VotingStrategy$execute(predictions, ...)
```

Arguments:

`predictions` A ClusterPredictions object containing the prediction achieved for each cluster.

`...` Further arguments passed down to `execute` function.

Method `getName()`**:** The function returns the name of the voting scheme.

Usage:

```
VotingStrategy$getName()
```

Returns: A character vector of size 1.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
VotingStrategy$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`D2MCS`, `SingleVoting`, `CombinedVoting`

Index