Package 'HEMDAG'

January 20, 2025

Title Hierarchical Ensemble Methods for Directed Acyclic Graphs

Version 2.7.4

Author Marco Notaro [aut, cre] (<https://orcid.org/0000-0003-4309-2200>),
Alessandro Petrini [ctb] (<https://orcid.org/0000-0002-0587-1484>),
Giorgio Valentini [aut] (<https://orcid.org/0000-0002-5694-3919>)

Maintainer Marco Notaro <marco.notaro@unimi.it>

Description An implementation of several Hierarchical Ensemble Methods (HEMs) for Directed Acyclic Graphs (DAGs). 'HEMDAG' package: 1) reconciles flat predictions with the topology of the ontology; 2) can enhance the predictions of virtually any flat learning methods by taking into account the hierarchical relationships between ontology classes; 3) provides biologically meaningful predictions that always obey the true-path-rule, the biological and logical rule that governs the internal coherence of biomedical ontologies; 4) is specifically designed for exploiting the hierarchical relationships of DAG-structured taxonomies, such as the Human Phenotype Ontology (HPO) or the Gene Ontology (GO), but can be safely applied to tree-structured taxonomies as well (as FunCat), since trees are DAGs; 5) scales nicely both in terms of the complexity of the taxonomy and in the cardinality of the examples; 6) provides several utility functions to process and analyze graphs; 7) provides several performance metrics to evaluate HEMs algorithms. (Marco Notaro, Max Schubach, Peter N. Robinson and Giorgio Valentini (2017) <doi:10.1186/s12859-017-1854-y>).

URL https://hemdag.readthedocs.io

https://github.com/marconotaro/hemdag

https://anaconda.org/bioconda/r-hemdag

BugReports https://github.com/marconotaro/hemdag/issues

Depends R (>= 2.10)

License GPL (>= 3)

Encoding UTF-8

Repository CRAN

LazyLoad true

NeedsCompilation yes

Imports graph, RBGL, precrec, preprocessCore, methods, plyr, foreach, doParallel, parallel

Suggests Rgraphviz, testthat

RoxygenNote 7.1.1

Date/Publication 2021-02-12 15:00:06 UTC

Contents

`HEMDAG-package` *HEMDAG: Hierarchical Ensemble Methods for Directed Acyclic Graphs*

Description

The HEMDAG package:

- provides an implementation of several Hierarchical Ensemble Methods (HEMs) for Directed Acyclic Graphs (DAGs);
- reconciles flat predictions with the topology of the ontology;
- can enhance predictions of virtually any flat learning methods by taking into account the hierarchical relationships between ontology classes;
- provides biologically meaningful predictions that obey the true-path-rule, the biological and logical rule that governs the internal coherence of biomedical ontologies;
- is specifically designed for exploiting the hierarchical relationships of DAG-structured taxonomies, such as the Human Phenotype Ontology (HPO) or the Gene Ontology (GO), but can be safely applied to tree-structured taxonomies as well (as FunCat), since trees are DAGs;
- scales nicely both in terms of the complexity of the taxonomy and in the cardinality of the examples;
- provides several utility functions to process and analyze graphs;
- provides several performance metrics to evaluate HEMs algorithms;

A comprehensive tutorial showing how to apply HEMDAG to real case bio-medical case studies is available at https://hemdag.readthedocs.io.

Details

The HEMDAG package implements the following Hierarchical Ensemble Methods for DAGs:

1. **HTD-DAG**: Hierarchical Top Down (`htd`);
2. **GPAV-DAG**: Generalized Pool-Adjacent Violators, *Burdakov et al.* (`gpav`);
3. **TPR-DAG**: True-Path Rule (`tpr.dag`);
4. **DESCENS**: Descendants Ensemble Classifier (`tpr.dag`);
5. **ISO-TPR**: Isotonic-True-Path Rule (`tpr.dag`);
6. **Max**, **And**, **Or**: Heuristic Methods, *Obozinski et al.* (`obozinski.heuristic.methods`);

Author(s)

Marco Notaro[1] (https://orcid.org/0000-0003-4309-2200);
Alessandro Petrini[1] (https://orcid.org/0000-0002-0587-1484);
Giorgio Valentini[1] (https://orcid.org/0000-0002-5694-3919);

Maintainer: *Marco Notaro* <marco.notaro@unimi.it>

[1] AnacletoLab, Computational Biology and Bioinformatics Laboratory, Computer Science Department, University of Milan, Italy

References

Marco Notaro, Max Schubach, Peter N. Robinson and Giorgio Valentini, *Prediction of Human Phenotype Ontology terms by means of Hierarchical Ensemble methods*, BMC Bioinformatics 2017, 18(1):449, doi: 10.1186/s128590171854y

---

`adj.upper.tri` *Binary upper triangular adjacency matrix*

Description

Compute a binary square upper triangular matrix where rows and columns correspond to the nodes' name of the graph g.

Usage

```
adj.upper.tri(g)
```

Arguments

g	a graph of class `graphNELL` representing the hierarchy of the class.

Details

The nodes of the matrix are topologically sorted (by using the `tsort` function of the **RBGL** package). Let's denote with `adj` our adjacency matrix. Then `adj` represents a partial order data set in which the class `j` dominates the class `i`. In other words, `adj[i,j]=1` means that `j` dominates `i`; `adj[i,j]=0` means that there is no edge between the class `i` and the class `j`. Moreover the nodes of `adj` are ordered such that `adj[i,j]=1` implies $i < j$, i.e. `adj` is upper triangular.

Value

An adjacency matrix which is square, logical and upper triangular.

Examples

```
data(graph);
adj <- adj.upper.tri(g);
```

---

auprc *AUPRC measures*

Description

Compute the Area under the Precision Recall Curve (AUPRC) through **precrec** package.

Usage

```
auprc.single.class(labels, scores, folds = NULL, seed = NULL)

auprc.single.over.classes(target, predicted, folds = NULL, seed = NULL)
```

Arguments

`labels`	vector of the true labels (0 negative, 1 positive examples).
`scores`	a numeric vector of the values of the predicted labels (scores).
`folds`	number of folds on which computing the AUPRC. If `folds=NULL` (`def.`), the AUPRC is computed one-shot, otherwise the AUPRC is computed averaged across folds.
`seed`	initialization seed for the random generator to create folds. Set `seed` only if `folds`$\neq$`NULL`. If `seed=NULL` and `folds`$\neq$`NULL`, the AUPRC averaged across folds is computed without seed initialization.
`target`	matrix with the target multilabel: rows correspond to examples and columns to classes. $target[i,j] = 1$ if example i belongs to class j, $target[i,j] = 0$ otherwise.
`predicted`	a numeric matrix with predicted values (scores): rows correspond to examples and columns to classes.

Details

The AUPRC (for a single class or for a set of classes) is computed either one-shot or averaged across stratified folds.

`auprc.single.class` computes the AUPRC just for a given class.

`auprc.single.over.classes` computes the AUPRC for a set of classes, returning also the averaged values across the classes.

For all those classes having zero annotations, the AUPRC is set to 0. These classes are discarded in the computing of the AUPRC averaged across classes, both when the AUPRC is computed one-shot or averaged across stratified folds.

Names of rows and columns of `labels` and `predicted` matrix must be provided in the same order, otherwise a stop message is returned.

Value

`auprc.single.class` returns a numeric value corresponding to the AUPRC for the considered class; `auprc.single.over.classes` returns a list with two elements:

1. average: the average AUPRC across classes;
2. per.class: a named vector with AUPRC for each class. Names correspond to classes.

Examples

```
data(labels);
data(scores);
data(graph);
root <- root.node(g);
L <- L[,-which(colnames(L)==root)];
S <- S[,-which(colnames(S)==root)];
prc.single.class <- auprc.single.class(L[,3], S[,3], folds=5, seed=23);
prc.over.classes <- auprc.single.over.classes(L, S, folds=5, seed=23);
```

---

auroc *AUROC measures*

Description

Compute the Area under the ROC Curve (AUROC) through **precrec** package.

Usage

```
auroc.single.class(labels, scores, folds = NULL, seed = NULL)

auroc.single.over.classes(target, predicted, folds = NULL, seed = NULL)
```

Arguments

labels	vector of the true labels (0 negative, 1 positive examples).
scores	a numeric vector of the values of the predicted labels (scores).
folds	number of folds on which computing the AUROC. If folds=NULL (def.), the AUROC is computed one-shot, otherwise the AUROC is computed averaged across folds.
seed	initialization seed for the random generator to create folds. Set seed only if folds≠NULL. If seed=NULL and folds≠NULL, the AUROC averaged across folds is computed without seed initialization.
target	annotation matrix: rows correspond to examples and columns to classes. $target[i,j] = 1$ if example i belongs to class j, $target[i,j] = 0$ otherwise.
predicted	a numeric matrix with predicted values (scores): rows correspond to examples and columns to classes.

Details

The AUROC (for a single class or for a set of classes) is computed either one-shot or averaged across stratified folds.

auroc.single.class computes the AUROC just for a given class.

auroc.single.over.classes computes the AUROC for a set of classes, including their average values across all the classes.

For all those classes having zero annotations, the AUROC is set to 0.5. These classes are included in the computing of the AUROC averaged across classes, both when the AUROC is computed one-shot or averaged across stratified folds.

The AUROC is set to 0.5 to all those classes having zero annotations. Names of rows and columns of labels and predicted must be provided in the same order, otherwise a stop message is returned.

Value

auroc.single.class returns a numeric value corresponding to the AUROC for the considered class; auprc.single.over.classes returns a list with two elements:

1. average: the average AUROC across classes;
2. per.class: a named vector with AUROC for each class. Names correspond to classes.

Examples

```
data(labels);
data(scores);
data(graph);
root <- root.node(g);
L <- L[,-which(colnames(L)==root)];
S <- S[,-which(colnames(S)==root)];
auc.single.class <- auroc.single.class(L[,3], S[,3], folds=5, seed=23);
auc.over.classes <- auroc.single.over.classes(L, S, folds=5, seed=23);
```

`build.ancestors` *Build ancestors*

Description

Build ancestors for each node of a graph.

Usage

```
build.ancestors(g)

build.ancestors.per.level(g, levels)

build.ancestors.bottom.up(g, levels)
```

Arguments

`g`	a graph of class `graphNEL`. It represents the hierarchy of the classes.
`levels`	a list of character vectors. Each component represents a graph level and the elements of any component correspond to nodes. The level 0 coincides with the root node.

Value

`build.ancestos` returns a named list of vectors. Each component corresponds to a node x of the graph and its vector is the set of its ancestors including also x.

`build.ancestors.per.level` returns a named list of vectors. Each component corresponds to a node x of the graph and its vector is the set of its ancestors including also x. The nodes are ordered from root (included) to leaves.

`build.ancestors.bottom.up` a named list of vectors. Each component corresponds to a node x of the graph and its vector is the set of its ancestors including also x. The nodes are ordered from leaves to root (included).

Examples

```
data(graph);
root <- root.node(g);
anc <- build.ancestors(g);
lev <- graph.levels(g, root=root);
anc.tod <-build.ancestors.per.level(g,lev);
anc.bup <- build.ancestors.bottom.up(g,lev);
```

build.children *Build children*

Description

Build children for each node of a graph.

Usage

```
build.children(g)

build.children.top.down(g, levels)

build.children.bottom.up(g, levels)
```

Arguments

g	a graph of class graphNEL. It represents the hierarchy of the classes.
levels	a list of character vectors. Each component represents a graph level and the elements of any component correspond to nodes. The level 0 coincides with the root node.

Value

build.children returns a named list of vectors. Each component corresponds to a node x of the graph and its vector is the set of its children.

build.children.top.down returns a named list of character vectors. Each component corresponds to a node x of the graph (i.e. parent node) and its vector is the set of its children. The nodes are ordered from root (included) to leaves.

build.children.bottom.up returns a named list of character vectors. Each component corresponds to a node x of the graph (i.e. parent node) and its vector is the set of its children. The nodes are ordered from leaves (included) to root.

Examples

```
data(graph);
root <- root.node(g);
children <- build.children(g);
lev <- graph.levels(g, root=root);
children.tod <- build.children.top.down(g,lev);
children.bup <- build.children.bottom.up(g,lev);
```

build.consistent.graph *Build consistent graph*

Description

Build a graph in which all nodes are reachable from root.

Usage

```
build.consistent.graph(g = g, root = "00")
```

Arguments

g	an object of class graphNEL.
root	name of the class that is on the top-level of the hierarchy (def. root="00").

Details

All nodes not accessible from root (if any) are removed from the graph and printed on stdout.

Value

A graph (as an object of class graphNEL) in which all nodes are accessible from root.

Examples

```
data(graph);
root <- root.node(g);
G <- graph::addNode(c("X","Y","Z"), g);
G <- graph::addEdge(c("X","Y","Z"), c("HP:0011844","HP:0009810","HP:0012385"), G);
G <- build.consistent.graph(G, root=root);
```

build.descendants *Build descendants*

Description

Build descendants for each node of a graph.

Usage

```
build.descendants(g)

build.descendants.per.level(g, levels)

build.descendants.bottom.up(g, levels)
```

Arguments

g	a graph of class graphNEL. It represents the hierarchy of the classes.
levels	a list of character vectors. Each component represents a graph level and the elements of any component correspond to nodes. The level 0 coincides with the root node.

Value

build.descendants returns a named list of vectors. Each component corresponds to a node x of the graph, and its vector is the set of its descendants including also x.

build.descendants.per.level returns a named list of vectors. Each component corresponds to a node x of the graph and its vector is the set of its descendants including also x. The nodes are ordered from root (included) to leaves.

build.descendants.bottom.up returns a named list of vectors. Each component corresponds to a node x of the graph and its vector is the set of its descendants including also x. The nodes are ordered from leaves to root (included).

Examples

```
data(graph);
root <- root.node(g);
desc <- build.descendants(g);
lev <- graph.levels(g, root=root);
desc.tod <- build.descendants.per.level(g,lev);
desc.bup <- build.descendants.bottom.up(g,lev);
```

---

build.edges.from.hpo.obo

Parse an HPO obo file

Description

Read an HPO obo file (HPO) and write the edges of the dag on a plain text file. The format of the file is a sequence of rows and each row corresponds to an edge represented through a pair of vertexes separated by blank.

Usage

```
build.edges.from.hpo.obo(obofile = "hp.obo", file = "edge.file")
```

Arguments

obofile	an HPO obo file. The extension of the obofile can be plain (".txt") or compressed (".gz").
file	name of the file of the edges to be written. The extension of the file can be plain (".txt") or compressed (".gz").

Details

A faster and more flexible parser to handle *obo* file can be found here.

Value

A text file representing the edges in the format: source destination (i.e. one row for each edge).

Examples

```
## Not run:
hpobo <- "http://purl.obolibrary.org/obo/hp.obo";
build.edges.from.hpo.obo(obofile=hpobo, file="hp.edge");
## End(Not run)
```

`build.parents` *Build parents*

Description

Build parents for each node of a graph.

Usage

```
build.parents(g, root = "00")

build.parents.top.down(g, levels, root = "00")

build.parents.bottom.up(g, levels, root = "00")

build.parents.topological.sorting(g, root = "00")
```

Arguments

`g`	a graph of class `graphNEL`. It represents the hierarchy of the classes.
`root`	name of the class that it is on the top-level of the hierarchy (`def. root="00"`).
`levels`	a list of character vectors. Each component represents a graph level and the elements of any component correspond to nodes. The level 0 represents the root node.

Value

`build.parents` returns a named list of character vectors. Each component corresponds to a node x of the graph (i.e. child node) and its vector is the set of its parents (the root node is not included).

`build.parents.top.down` returns a named list of character vectors. Each component corresponds to a node x of the graph (i.e. child node) and its vector is the set of its parents. The order of nodes follows the levels of the graph from root (excluded) to leaves.

`build.parents.bottom.up` returns a named list of character vectors. Each component corresponds to a node x of the graph (i.e. child node) and its vector is the set of its parents. The nodes are ordered from leaves to root (excluded).

`build.parents.topological.sorting` a named list of character vectors. Each component corresponds to a node x of the graph (i.e. child node) and its vector is the set of its parents. The nodes are ordered according to a topological sorting, i.e. parents node come before children node.

Examples

```
data(graph);
root <- root.node(g)
parents <- build.parents(g, root=root);
lev <- graph.levels(g, root=root);
parents.tod <- build.parents.top.down(g, lev, root=root);
parents.bup <- build.parents.bottom.up(g, lev, root=root);
parents.tsort <- build.parents.topological.sorting(g, root=root);
```

---

build.scores.matrix *Build scores matrix*

Description

Build a scores matrix from file

Usage

```
build.scores.matrix.from.list(file = "scores.list.txt", split = "[(\t,|)]")

build.scores.matrix.from.tupla(file = "scores.tupla.txt")
```

Arguments

`file`	name of the text file to be read. The matrix of the input file can be either a list (e.g in the form `example nodeX|score`), or a tupla (i.e. in the form `example nodeX score`).The file extension can be plain (".txt") or compressed (".gz").
`split`	character vector containing a regular expression use for splitting.

Value

A named scores matrix.

Examples

```
file.list  <- system.file("extdata/scores.list.txt.gz", package="HEMDAG");
file.tupla <- system.file("extdata/scores.tupla.txt.gz", package="HEMDAG");
S <- build.scores.matrix.from.list(file.list, split="[(\t,|)]");
S <- build.scores.matrix.from.tupla(file.tupla);
```

build.subgraph *Build subgraph*

Description

Build a subgraph with only the supplied nodes and any edges between them.

Usage

```
build.subgraph(nd, g, edgemode = "directed")
```

Arguments

nd	a vector with the nodes for which the subgraph must be built.
g	a graph of class graphNEL. It represents the hierarchy of the classes.
edgemode	can be "directed" or "undirected".

Value

A subgraph with only the supplied nodes.

Examples

```
data(graph);
anc <- build.ancestors(g);
nd <- anc[["HP:0001371"]];
subg <- build.subgraph(nd, g, edgemode="directed");
```

build.submatrix *Build submatrix*

Description

Terms having less than n annotations are pruned. Terms having exactly n annotations are discarded as well.

Usage

```
build.submatrix(ann, n)
```

Arguments

ann	the annotation matrix (0/1). Rows are examples and columns are functional terms.
n	an integer number representing the number of annotations to be pruned.

Value

An annotation matrix having only those terms with more than n annotations.

Examples

```
data(labels);
subm <- build.submatrix(L,5);
```

---

check.annotation.matrix.integrity

Annotation matrix checker

Description

Assess the integrity of an annotation matrix where a transitive closure of annotations was performed.

Usage

```
check.annotation.matrix.integrity(anc, ann.spec, ann)
```

Arguments

anc	the ancestor list.
ann.spec	the annotation matrix of the most specific annotations (0/1): rows are genes and columns are terms.
ann	the full annotation matrix (0/1), i.e. the matrix where the transitive closure of the annotation was performed. Rows are examples and columns are terms.

Value

If the transitive closure of the annotations is performed correctly, OK is returned, otherwise an error message is printed on the stdout.

Examples

```
data(graph);
data(labels);
anc <- build.ancestors(g);
tca <- transitive.closure.annotations(L, anc);
check.annotation.matrix.integrity(anc, L, tca);
```

check.dag.integrity *DAG checker*

Description

Check the integrity of a dag.

Usage

```
check.dag.integrity(g, root = "00")
```

Arguments

g	a graph of class graphNEL. It represents the hierarchy of the classes.
root	name of the class that is on the top-level of the hierarchy (def. root="00").

Value

If all the nodes are accessible from the root "dag is ok" is printed, otherwise a message error and the list of the not accessible nodes is printed on the stdout.

Examples

```
data(graph);
root <- root.node(g);
check.dag.integrity(g, root=root);
```

compute.flipped.graph *Flip graph*

Description

Compute a directed graph with edges in the opposite direction.

Usage

```
compute.flipped.graph(g)
```

Arguments

g	a graphNEL directed graph

Value

A graph (as an object of class graphNEL) with edges in the opposite direction w.r.t. g.

Examples

```
data(graph);
g.flipped <- compute.flipped.graph(g);
```

`constraints.matrix` *Constraints matrix*

Description

Return a matrix with two columns and as many rows as there are edges. The entries of the first columns are the index of the node the edge comes from (i.e. children nodes), the entries of the second columns indicate the index of node the edge is to (i.e. parents nodes). Referring to a dag this matrix defines a partial order.

Usage

```
constraints.matrix(g)
```

Arguments

g	a graph of class `graphNELL`. It represents the hierarchy of the classes.

Value

A constraints matrix w.r.t the graph g.

Examples

```
data(graph);
m <- constraints.matrix(g);
```

`create.stratified.fold.df`
DataFrame for stratified cross validation

Description

Create a data frame for stratified cross-validation.

Usage

```
create.stratified.fold.df(labels, scores, folds = 5, seed = 23)
```

Arguments

labels	vector of the true labels (0 negative, 1 positive).
scores	a numeric vector of the values of the predicted labels.
folds	number of folds of the cross validation (def. folds=5).
seed	initialization seed for the random generator to create folds (def. seed=23). If seed=NULL, the stratified folds are generated without seed initialization.

Details

Folds are *stratified*, i.e. contain the same amount of positive and negative examples.

Value

A data frame with three columns:

- scores: contains the predicted scores;
- labels: contains the labels as pos or neg;
- folds: contains the index of the fold in which the example falls. The index can range from 1 to the number of folds.

Examples

```
data(labels);
data(scores);
df <- create.stratified.fold.df(L[,3], S[,3], folds=5, seed=23);
```

---

distances.from.leaves *Distances from leaves*

Description

Compute the minimum distance of each node from one of the leaves of the graph.

Usage

```
distances.from.leaves(g)
```

Arguments

g	a graph of class graphNEL. It represents the hierarchy of the classes.

Value

A named vector. The names are the names of the nodes of the graph g, and their values represent the distance from the leaves. A value equal to 0 is assigned to the leaves, 1 to nodes with distance 1 from a leaf and so on.

Examples

```
data(graph);
dist.leaves <- distances.from.leaves(g);
```

`example.datasets` *Small real example datasets*

Description

Collection of real sub-datasets used in the examples of the **HEMDAG** package

Usage

```
data(graph)
data(labels)
data(scores)
data(wadj)
data(test.index)
```

Details

The DAG g contained in `graph` data is an object of class `graphNEL`. The graph g has 23 nodes and 30 edges and represents the "ancestors view" of the HPO term *Camptodactyly of finger* (`"HP:0100490"`).

The matrix `L` contained in the `labels` data is a 100 X 23 matrix, whose rows correspond to genes (*Entrez GeneID*) and columns to HPO classes. $L[i,j] = 1$ means that the gene i belong to class j, $L[i,j] = 0$ means that the gene i does not belong to class j. The classes of the matrix `L` correspond to the nodes of the graph g.

The matrix `S` contained in the `scores` data is a named 100 X 23 flat scores matrix, representing the likelihood that a given gene belongs to a given class: higher the value higher the likelihood. The classes of the matrix S correspond to the nodes of the graph g.

The matrix `W` contained in the `wadj` data is a named 100 X 100 symmetric weighted adjacency matrix, whose rows and columns correspond to genes.The genes names (*Entrez GeneID*) of the adjacency matrix `W` correspond to the genes names of the flat scores matrix `S` and to genes names of the target multilabel matrix `L`.

The vector of integer numbers `test.index` contained in the `test.index` data refers to the index of the examples of the scores matrix `S` to be used in the test set. It is useful only in holdout experiments.

Note

Some examples of full data sets for the prediction of HPO terms are available at the following link. Note that the processing of the full datasets should be done similarly to the processing of the small data examples provided directly in this package. Please read the `README` clicking the link above to know more details about the available full datasets.

`find.best.f` *Best hierarchical F-score*

Description

Select the best hierarchical F-score by choosing an appropriate threshold in the scores.

Usage

```
find.best.f(
  target,
  predicted,
  n.round = 3,
  verbose = TRUE,
  b.per.example = FALSE
)
```

Arguments

`target`	matrix with the target multilabel: rows correspond to examples and columns to classes. $target[i, j] = 1$ if example i belongs to class j, $target[i, j] = 0$ otherwise.
`predicted`	a numeric matrix with continuous predicted values (scores): rows correspond to examples and columns to classes.
`n.round`	number of rounding digits to be applied to predicted (`default=3`).
`verbose`	a boolean value. If `TRUE` (def.) the number of iterations are printed on stdout.
`b.per.example`	a boolean value. • `TRUE`: results are returned for each example; • `FALSE`: only the average results are returned;

Details

All the examples having no positive annotations are discarded. The predicted scores matrix (`predicted`) is rounded according to parameter `n.round` and all the values of `predicted` are divided by `max(predicted)`. Then all the thresholds corresponding to all the different values included in `predicted` are attempted, and the threshold leading to the maximum F-measure is selected.

Names of rows and columns of `target` and `predicted` matrix must be provided in the same order, otherwise a stop message is returned.

Value

Two different outputs respect to the input parameter `b.per.example`:

- b.per.example==FALSE: a list with a single element average. A named vector with 7 elements relative to the best result in terms of the F.measure: Precision (P), Recall (R), Specificity (S), F.measure (F), av.F.measure (av.F), Accuracy (A) and the best selected Threshold (T). F is the F-measure computed as the harmonic mean between the average precision and recall; av.F is the F-measure computed as the average across examples and T is the best selected threshold;
- b.per.example==FALSE: a list with two elements:
  1. average: a named vector with with 7 elements relative to the best result in terms of the F.measure: Precision (P), Recall (R), Specificity (S), F.measure (F), av.F.measure (av.F), Accuracy (A) and the best selected Threshold (T);
  2. per.example: a named matrix with the Precision (P), Recall (R), Specificity (S), Accuracy (A), F-measure (F), av.F-measure (av.F) and the best selected Threshold (T) for each example. Row names correspond to examples, column names correspond respectively to Precision (P), Recall (R), Specificity (S), Accuracy (A), F-measure (F), av.F-measure (av.F) and the best selected Threshold (T);

Examples

```
data(graph);
data(labels);
data(scores);
root <- root.node(g);
L <- L[,-which(colnames(L)==root)];
S <- S[,-which(colnames(S)==root)];
fscore <- find.best.f(L, S, n.round=3, verbose=TRUE, b.per.example=TRUE);
```

---

find.leaves *Leaves*

Description

Find leaves of a directed graph.

Usage

```
find.leaves(g)
```

Arguments

g	a graph of class graphNEL. It represents the hierarchy of the classes.

Value

A vector with the names of the leaves of g.

Examples

```
data(graph);
leaves <- find.leaves(g);
```

fmax *Compute Fmax*

Description

Compute the best hierarchical Fmax either one-shot or averaged across folds

Usage

```
compute.fmax(
  target,
  predicted,
  n.round = 3,
  verbose = TRUE,
  b.per.example = FALSE,
  folds = NULL,
  seed = NULL
)
```

Arguments

target	matrix with the target multilabel: rows correspond to examples and columns to classes. $target[i, j] = 1$ if example i belongs to class j, $target[i, j] = 0$ otherwise.
predicted	a numeric matrix with predicted values (scores): rows correspond to examples and columns to classes.
n.round	number of rounding digits to be applied to predicted (default=3).
verbose	a boolean value. If TRUE (def.) the number of iterations are printed on stdout.
b.per.example	a boolean value. • TRUE: results are returned for each example; • FALSE: only the average results are returned;
folds	number of folds on which computing the Fmax If folds=NULL (def.), the Fmax is computed one-shot, otherwise the Fmax is computed averaged across folds.
seed	initialization seed for the random generator to create folds. Set seed only if folds≠NULL. If seed=NULL and folds≠NULL, the Fmax averaged across folds is computed without seed initialization.

Details

Names of rows and columns of target and predicted matrix must be provided in the same order, otherwise a stop message is returned.

Value

Two different outputs respect to the input parameter `b.per.example`:

- `b.per.example==FALSE`: a list with a single element average. A named vector with 7 elements relative to the best result in terms of the F.measure: Precision (P), Recall (R), Specificity (S), F.measure (F), av.F.measure (av.F), Accuracy (A) and the best selected Threshold (T). F is the F-measure computed as the harmonic mean between the average precision and recall; av.F is the F-measure computed as the average across examples and T is the best selected threshold;
- `b.per.example==FALSE`: a list with two elements:
  1. average: a named vector with with 7 elements relative to the best result in terms of the F.measure: Precision (P), Recall (R), Specificity (S), F.measure (F), av.F.measure (av.F), Accuracy (A) and the best selected Threshold (T);
  2. per.example: a named matrix with the Precision (P), Recall (R), Specificity (S), Accuracy (A), F-measure (F), av.F-measure (av.F) and the best selected Threshold (T) for each example. Row names correspond to examples, column names correspond respectively to Precision (P), Recall (R), Specificity (S), Accuracy (A), F-measure (F), av.F-measure (av.F) and the best selected Threshold (T);

Examples

```
data(graph);
data(labels);
data(scores);
root <- root.node(g);
L <- L[,-which(colnames(L)==root)];
S <- S[,-which(colnames(S)==root)];
fmax <- compute.fmax(L, S, n.round=3, verbose=TRUE, b.per.example=TRUE, folds=5, seed=23);
```

`full.annotation.matrix`

Full annotation matrix

Description

Build a full annotations matrix using the ancestor list and the most specific annotations matrix w.r.t. a given weighted adjacency matrix (wadj). The rows of the full annotation matrix correspond to all the examples of the given weighted adjacency matrix and the columns to the class/terms. The transitive closure of the annotations is performed.

Usage

```
full.annotation.matrix(W, anc, ann.spec)
```

Arguments

`W`	a symmetric adjacency weighted matrix of the graph.
`anc`	the ancestor list.
`ann.spec`	the annotation matrix of the most specific annotations (0/1): rows are genes and columns are terms.

Details

The examples present in the annotation matrix (ann.spec) but not in the adjacency weighted matrix (W) are purged.

Value

A full annotation table T, that is a matrix where the transitive closure of annotations is performed. Rows correspond to genes of the weighted adjacency matrix and columns to terms. $T[i, j] = 1$ means that gene i is annotated for the term j, $T[i, j] = 0$ means that gene i is not annotated for the term j.

Examples

```
data(wadj);
data(graph);
data(labels);
anc <- build.ancestors(g);
full.ann <- full.annotation.matrix(W, anc, L);
```

---

gpav *Generalized Pool-Adjacent Violators (GPAV)*

Description

Implementation of GPAV (Generalized Pool-Adjacent Violators) algorithm. (*Burdakov et al., In: Di Pillo G, Roma M, editors. An O(n2) Algorithm for Isotonic Regression. Boston, MA: Springer US; 2006. p. 25–33. Available from: doi: 10.1007/0387300651_3*

Usage

```
gpav(Y, W = NULL, adj)
```

Arguments

Y	vector of scores relative to a single example. Y must be a numeric named vector, where names correspond to classes' names, i.e. nodes of the graph g (root node included).
W	vector of weight relative to a single example. If W=NULL (def.) it is assumed that W is a unitary vector of the same length of the columns' number of the matrix S (root node included).
adj	adjacency matrix of the graph which must be sparse, logical and upper triangular. Number of columns of adj must be equal to the length of Y and W.

Details

Given the constraints adjacency matrix of the graph, a vector of scores $\hat{y} \in R^n$ and a vector of strictly positive weights $w \in R^n$, the GPAV algorithm returns a vector $\bar{y}$ which is as close as possible, in the least-squares sense, to the response vector $\hat{y}$ and whose components are partially ordered in accordance with the constraints matrix `adj`. In other words, GPAV solves the following problem:

$$\bar{y} = \begin{cases} \min \sum_{i \in V} (\hat{y}_i - \bar{y}_i)^2 \\ \\ \forall i, \quad j \in par(i) \Rightarrow \bar{y}_j \geq \bar{y}_i \end{cases}$$

where V are the number of vertexes of the graph.

Value

A list of 3 elements:

- `YFit`: a named vector with the scores of the classes corrected according to the GPAV algorithm.
- `blocks`: list of vectors, containing the partitioning of nodes (represented with an integer number) into blocks;
- `W`: vector of weights.

Examples

```
data(graph);
data(scores);
Y <- S[3,];
adj <- adj.upper.tri(g);
Y.gpav <- gpav(Y,W=NULL,adj);
```

`gpav.holdout` *GPAV holdout*

Description

Correct the computed scores in a hierarchy according to the GPAV algorithm by applying a classical holdout procedure.

Usage

```
gpav.holdout(
  S,
  g,
  testIndex,
  W = NULL,
  parallel = FALSE,
  ncores = 1,
  norm = TRUE,
  norm.type = NULL
)
```

Arguments

S	a named flat scores matrix with examples on rows and classes on columns (root node included).
g	a graph of class graphNEL. It represents the hierarchy of the classes.
testIndex	a vector of integer numbers corresponding to the indexes of the elements (rows) of the scores matrix S to be used in the test set.
W	vector of weight relative to a single example. If W=NULL (def.) it is assumed that W is a unitary vector of the same length of the columns' number of the matrix S (root node included).
parallel	a boolean value. Should the parallel version GPAV be run? • TRUE: execute the parallel implementation of GPAV (gpav.parallel); • FALSE (def.): execute the sequential implementation of GPAV (gpav.over.examples);
ncores	number of cores to use for parallel execution. Set ncores=1 if parallel=FALSE, otherwise set ncores to the desired number of cores.
norm	a boolean value. Should the flat score matrix be normalized? By default norm=FALSE. If norm=TRUE the matrix S is normalized according to the normalization type selected in norm.type.
norm.type	a string character. It can be one of the following values: 1. NULL (def.): none normalization is applied (norm=FALSE) 2. maxnorm: each score is divided for the maximum value of each class; 3. qnorm: quantile normalization. **preprocessCore** package is used;

Value

A named matrix with the scores of the classes corrected according to the GPAV algorithm. Rows of the matrix are shrunk to testIndex.

Examples

```
data(graph);
data(scores);
data(test.index);
S.gpav <- gpav.holdout(S, g, testIndex=test.index, norm=FALSE, norm.type=NULL);
```

---

gpav.over.examples *GPAV over examples*

Description

Compute GPAV across all the examples.

Usage

```
gpav.over.examples(S, g, W = NULL)
```

Arguments

S	a named flat scores matrix with examples on rows and classes on columns (root node included).
g	a graph of class graphNEL. It represents the hierarchy of the classes.
W	vector of weight relative to a single example. If W=NULL (def.) it is assumed that W is a unitary vector of the same length of the columns' number of the matrix S (root node included).

Value

A named matrix with the scores of the classes corrected according to the GPAV algorithm.

See Also

gpav.parallel

Examples

```
data(graph);
data(scores);
S.gpav <- gpav.over.examples(S,W=NULL,g);
```

---

gpav.parallel *GPAV over examples – parallel implementation*

Description

Compute GPAV across all the examples (parallel implementation).

Usage

```
gpav.parallel(S, g, W = NULL, ncores = 8)
```

Arguments

S	a named flat scores matrix with examples on rows and classes on columns (root node included).
g	a graph of class graphNEL. It represents the hierarchy of the classes.
W	vector of weight relative to a single example. If W=NULL (def.) it is assumed that W is a unitary vector of the same length of the columns' number of the matrix S (root node included).
ncores	number of cores to use for parallel execution (def. 8). If ncores=0, the maximum number of cores minus one are used.

Value

A named matrix with the scores of the classes corrected according to the GPAV algorithm.

Examples

```
data(graph);
data(scores);
if(Sys.info()['sysname']!="Windows"){
   S.gpav <- gpav.parallel(S,W=NULL,g,ncores=2);
}
```

---

gpav.vanilla *GPAV vanilla*

Description

Correct the computed scores in a hierarchy according to the GPAV algorithm.

Usage

```
gpav.vanilla(
  S,
  g,
  W = NULL,
  parallel = FALSE,
  ncores = 1,
  norm = FALSE,
  norm.type = NULL
)
```

Arguments

S	a named flat scores matrix with examples on rows and classes on columns (root node included).
g	a graph of class graphNEL. It represents the hierarchy of the classes.
W	vector of weight relative to a single example. If W=NULL (def.) it is assumed that W is a unitary vector of the same length of the columns' number of the matrix S (root node included).
parallel	a boolean value. Should the parallel version GPAV be run? • TRUE: execute the parallel implementation of GPAV (gpav.parallel); • FALSE (def.): execute the sequential implementation of GPAV (gpav.over.examples);
ncores	number of cores to use for parallel execution. Set ncores=1 if parallel=FALSE, otherwise set ncores to the desired number of cores.
norm	a boolean value. Should the flat score matrix be normalized? By default norm=FALSE. If norm=TRUE the matrix S is normalized according to the normalization type selected in norm.type.
norm.type	a string character. It can be one of the following values: 1. NULL (def.): none normalization is applied (norm=FALSE) 2. maxnorm: each score is divided for the maximum value of each class; 3. qnorm: quantile normalization. **preprocessCore** package is used;

Value

A named matrix with the scores of the classes corrected according to the GPAV algorithm.

Examples

```
data(graph);
data(scores);
S.gpav <- gpav.vanilla(S, g, W=NULL, parallel=FALSE, ncores=1, norm=FALSE, norm.type=NULL);
```

---

graph.levels *Build graph levels*

Description

Group a set of nodes in according to their maximum depth in the graph. Firstly, it inverts the weights of the graph and then it applies the Bellman Ford algorithm to find the shortest path, achieving in this way the longest path.

Usage

```
graph.levels(g, root = "00")
```

Arguments

g	an object of class graphNEL.
root	name of the class that it is on the top-level of the hierarchy (def. root="00").

Value

A list of the nodes grouped w.r.t. the distance from the root: the first element of the list corresponds to the root node (level 0), the second to nodes at maximum distance 1 (level 1), the third to the node at maximum distance 3 (level 2) and so on.

Examples

```
data(graph);
root <- root.node(g);
lev <- graph.levels(g, root=root);
```

`hierarchical.checkers` *Hierarchical constraints checker*

Description

Check if the true path rule is violated or not. In other words this function checks if the score of a parent or an ancestor node is always larger or equal than that of its children or descendants nodes.

Usage

```
check.hierarchy.single.sample(y.hier, g, root = "00")

check.hierarchy(S.hier, g, root = "00")
```

Arguments

`y.hier`	vector of scores relative to a single example. It must be a named numeric vector (names are functional classes).
`g`	a graph of class `graphNEL`. It represents the hierarchy of the classes.
`root`	name of the class that is on the top-level of the hierarchy (`def. root="00"`).
`S.hier`	the matrix with the scores of the classes corrected in according to hierarchy. It must be a named matrix: rows are examples and columns are functional classes.

Value

A list of 3 elements:

- status:
  - `OK` if none hierarchical constraints have bee broken;
  - `NOTOK` if there is at least one hierarchical constraints broken;
- hierarchy_constraints_broken:
  - TRUE: example did not respect the hierarchical constraints;
  - FALSE: example broke the hierarchical constraints;
- hierarchy_constraints_satisfied: how many terms satisfied the hierarchical constraint;

Examples

```
data(graph);
data(scores);
root <- root.node(g);
S.hier <- htd(S,g,root);
S.hier.single.example <- S.hier[sample(ncol(S.hier),1),];
check.hierarchy.single.sample(S.hier.single.example, g, root=root);
check.hierarchy(S.hier, g, root);
```

htd *HTD-DAG*

Description

Implementation of the top-down procedure to correct the scores of the hierarchy according to the constraints that the score of a node cannot be greater than a score of its parents.

Usage

```
htd(S, g, root = "00")
```

Arguments

S	a named flat scores matrix with examples on rows and classes on columns.
g	a graph of class graphNEL. It represents the hierarchy of the classes.
root	name of the class that it is the top-level of the hierarchy (def:00).

Details

The HTD-DAG algorithm modifies the flat scores according to the hierarchy of a DAG G through a unique run across the nodes of the graph. For a given example x, the flat predictions $f(x) = \hat{y}$ are hierarchically corrected to $\bar{y}$, by per-level visiting the nodes of the DAG from top to bottom according to the following simple rule:

$$\bar{y}_i := \begin{cases} \hat{y}_i & \text{if} \quad i \in root(G) \\ \min_{j \in par(i)} \bar{y}_j & \text{if} \quad \min_{j \in par(i)} \bar{y}_j < \hat{y}_i \\ \hat{y}_i & \text{otherwise} \end{cases}$$

The node levels correspond to their maximum path length from the root.

Value

A matrix with the scores of the classes corrected according to the HTD-DAG algorithm.

Examples

```
data(graph);
data(scores);
root <- root.node(g);
S.htd <- htd(S,g,root);
```

htd.holdout *HTD-DAG holdout*

Description

Correct the computed scores in a hierarchy according to the HTD-DAG algorithm applying a classical holdout procedure.

Usage

```
htd.holdout(S, g, testIndex, norm = FALSE, norm.type = NULL)
```

Arguments

S	a named flat scores matrix with examples on rows and classes on columns.
g	a graph of class graphNEL. It represents the hierarchy of the classes.
testIndex	a vector of integer numbers corresponding to the indexes of the elements (rows) of the scores matrix S to be used in the test set.
norm	a boolean value. Should the flat score matrix be normalized? By default norm=FALSE. If norm=TRUE the matrix S is normalized according to the normalization type selected in norm.type.
norm.type	a string character. It can be one of the following values: 1. NULL (def.): none normalization is applied (norm=FALSE) 2. maxnorm: each score is divided for the maximum value of each class; 3. qnorm: quantile normalization. **preprocessCore** package is used;

Value

A matrix with the scores of the classes corrected according to the HTD-DAG algorithm. Rows of the matrix are shrunk to testIndex.

Examples

```
data(graph);
data(scores);
data(test.index);
S.htd <- htd.holdout(S, g, testIndex=test.index, norm=FALSE, norm.type=NULL);
```

htd.vanilla *HTD-DAG vanilla*

Description

Correct the computed scores in a hierarchy according to the HTD-DAG algorithm.

Usage

```
htd.vanilla(S, g, norm = FALSE, norm.type = NULL)
```

Arguments

S	a named flat scores matrix with examples on rows and classes on columns.
g	a graph of class graphNEL. It represents the hierarchy of the classes.
norm	a boolean value. Should the flat score matrix be normalized? By default norm=FALSE. If norm=TRUE the matrix S is normalized according to the normalization type selected in norm.type.
norm.type	a string character. It can be one of the following values: 1. NULL (def.): none normalization is applied (norm=FALSE) 2. maxnorm: each score is divided for the maximum value of each class; 3. qnorm: quantile normalization. **preprocessCore** package is used;

Value

A matrix with the scores of the classes corrected according to the HTD-DAG algorithm.

Examples

```
data(graph);
data(scores);
S.htd <- htd.vanilla(S, g, norm=FALSE, norm.type=NULL);
```

lexicographical.topological.sort *Lexicographical topological sorting*

Description

Nodes of a graph are sorted according to a lexicographical topological ordering.

Usage

```
lexicographical.topological.sort(g)
```

Arguments

g	an object of class graphNEL.

Details

A topological sorting is a linear ordering of the nodes such that given an edge from u to v, the node u comes before node v in the ordering. Topological sorting is not possible if the graph g contains self-loop. To implement the topological sorting algorithm we applied the Kahn's algorithm.

Value

A vector in which the nodes of the graph g are sorted according to a lexicographical topological order.

Examples

```
data(graph);
T <- lexicographical.topological.sort(g);
```

---

multilabel.F.measure *multilabel F-measure*

Description

Method for computing Precision, Recall, Specificity, Accuracy and F-measure for multiclass and multilabel classification.

Usage

```
F.measure.multilabel(target, predicted, b.per.example = FALSE)

## S4 method for signature 'matrix,matrix'
F.measure.multilabel(target, predicted, b.per.example = FALSE)
```

Arguments

target	matrix with the target multilabel: rows correspond to examples and columns to classes. $target[i,j] = 1$ if example i belongs to class j, $target[i,j] = 0$ otherwise.
predicted	a numeric matrix with discrete predicted values: rows correspond to examples and columns to classes. $predicted[i,j] = 1$ if example i is predicted belonging to class j, $target[i,j] = 0$ otherwise.
b.per.example	a boolean value. • TRUE: results are returned for each example; • FALSE: only the average results are returned;

Details

Names of rows and columns of `target` and `predicted` matrix must be provided in the same order, otherwise a stop message is returned.

Value

Two different outputs respect to the input parameter `b.per.example`:

- `b.per.example==FALSE`: a list with a single element average. A named vector with average precision (P), recall (R), specificity (S), F-measure (F), average F-measure (avF) and Accuracy (A) across examples. F is the F-measure computed as the harmonic mean between the average precision and recall; av.F is the F-measure computed as average across examples;
- `b.per.example==FALSE`: a list with two elements:
  1. average: a named vector with average precision (P), recall (R), specificity (S), F-measure (F), average F-measure (avF) and Accuracy (A) across examples;
  2. per.example: a named matrix with the Precision (P), Recall (R), Specificity (S), Accuracy (A), F-measure (F) and av.F-measure (av.F) for each example. Row names correspond to examples, column names correspond respectively to Precision (P), Recall (R), Specificity (S), Accuracy (A), F-measure (F) and av.F-measure (av.F);

Examples

```
data(labels);
data(scores);
data(graph);
root <- root.node(g);
L <- L[,-which(colnames(L)==root)];
S <- S[,-which(colnames(S)==root)];
S[S>0.7] <- 1;
S[S<0.7] <- 0;
fscore <- F.measure.multilabel(L,S);
```

`normalize.max` *Max normalization*

Description

Normalize the scores of a scores matrix by dividing the score values of each class for the maximum score of the class.

Usage

```
normalize.max(S)
```

Arguments

`S`	a scores matrix. Rows are examples and columns are classes.

Value

A scores matrix with the scores normalized.

Examples

```
data(scores);
maxnorm <- normalize.max(S);
```

`obozinski.heuristic.methods`

Obozinski heuristic methods

Description

Implementation of the Obozinski's heuristic methods `Max`, `And`, `Or` (*Obozinski et al., Genome Biology, 2008, doi: 10.1186/gb20089s1s6*).

Usage

```
obozinski.max(S, g, root = "00")

obozinski.and(S, g, root = "00")

obozinski.or(S, g, root = "00")
```

Arguments

`S`	a named flat scores matrix with examples on rows and classes on columns.
`g`	a graph of class `graphNEL`. It represents the hierarchy of the classes.
`root`	name of the class that it is the top-level of the hierarchy (`def:00`).

Details

Obozinski's heuristic methods:

1. **Max**: reports the largest logistic regression (LR) value of self and all descendants: $p_i = max_{j \in descendants(i)} \hat{p_j}$;
2. **And**: reports the product of LR values of all ancestors and self. This is equivalent to computing the probability that all ancestral terms are "on" assuming that, conditional on the data, all predictions are independent: $p_i = \prod_{j \in ancestors(i)} \hat{p_j}$;
3. **Or**: computes the probability that at least one of the descendant terms is "on" assuming again that, conditional on the data, all predictions are independent: $1 - p_i = \prod_{j \in descendants(i)} (1 - \hat{p_j})$;

Value

A matrix with the scores of the classes corrected according to the chosen Obozinski's heuristic algorithm.

Examples

```
data(graph);
data(scores);
root  <- root.node(g);
S.max <- obozinski.max(S,g,root);
S.and <- obozinski.and(S,g,root);
S.or  <- obozinski.or(S,g,root);
```

---

obozinski.holdout *Obozinski's heuristic methods – holdout*

Description

Compute the Obozinski's heuristic methods Max, And, Or (*Obozinski et al., Genome Biology, 2008*) applying a classical holdout procedure.

Usage

```
obozinski.holdout(
  S,
  g,
  testIndex,
  heuristic = "and",
  norm = FALSE,
  norm.type = NULL
)
```

Arguments

S	a named flat scores matrix with examples on rows and classes on columns.
g	a graph of class graphNEL. It represents the hierarchy of the classes.
testIndex	a vector of integer numbers corresponding to the indexes of the elements (rows) of the scores matrix S to be used in the test set.
heuristic	a string character. It can be one of the following three values: 1. "max": run the method heuristic.max; 2. "and": run the method heuristic.and; 3. "or": run the method heuristic.or;
norm	a boolean value. Should the flat score matrix be normalized? By default norm=FALSE. If norm=TRUE the matrix S is normalized according to the normalization type selected in norm.type.
norm.type	a string character. It can be one of the following values: 1. NULL (def.): none normalization is applied (norm=FALSE) 2. maxnorm: each score is divided for the maximum value of each class; 3. qnorm: quantile normalization. **preprocessCore** package is used;

Value

A matrix with the scores of the classes corrected according to the chosen heuristic algorithm. Rows of the matrix are shrunk to `testIndex`.

Examples

```
data(graph);
data(scores);
data(test.index);
S.and <- obozinski.holdout(S, g, testIndex=test.index, heuristic="and", norm=FALSE, norm.type=NULL);
```

`obozinski.methods` *Obozinski's heuristic methods calling*

Description

Compute the Obozinski's heuristic methods `Max`, `And`, `Or` (*Obozinski et al., Genome Biology, 2008*).

Usage

```
obozinski.methods(S, g, heuristic = "and", norm = FALSE, norm.type = NULL)
```

Arguments

`S`	a named flat scores matrix with examples on rows and classes on columns.
`g`	a graph of class `graphNEL`. It represents the hierarchy of the classes.
`heuristic`	a string character. It can be one of the following three values: 1. "max": run the method `obozinski.max`; 2. "and": run the method `obozinski.and`; 3. "or": run the method `obozinski.or`;
`norm`	a boolean value. Should the flat score matrix be normalized? By default `norm=FALSE`. If `norm=TRUE` the matrix `S` is normalized according to the normalization type selected in `norm.type`.
`norm.type`	a string character. It can be one of the following values: 1. `NULL` (def.): none normalization is applied (`norm=FALSE`) 2. `maxnorm`: each score is divided for the maximum value of each class; 3. `qnorm`: quantile normalization. **preprocessCore** package is used;

Value

A matrix with the scores of the classes corrected according to the chosen heuristic algorithm.

Examples

```
data(graph);
data(scores);
S.and <- obozinski.methods(S, g, heuristic="and", norm=TRUE, norm.type="maxnorm");
```

pxr *Precision-Recall curves*

Description

Compute the Precision-Recall (PxR) values through **precrec** package.

Usage

```
precision.at.all.recall.levels.single.class(labels, scores)

precision.at.given.recall.levels.over.classes(
  target,
  predicted,
  folds = NULL,
  seed = NULL,
  recall.levels = seq(from = 0.1, to = 1, by = 0.1)
)
```

Arguments

labels	vector of the true labels (0 negative, 1 positive examples).
scores	a numeric vector of the values of the predicted labels (scores).
target	matrix with the target multilabel: rows correspond to examples and columns to classes. $target[i, j] = 1$ if example i belongs to class j, $target[i, j] = 0$ otherwise.
predicted	a numeric matrix with predicted values (scores): rows correspond to examples and columns to classes.
folds	number of folds on which computing the PXR. If folds=NULL (def.), the PXR is computed one-shot, otherwise the PXR is computed averaged across folds.
seed	initialization seed for the random generator to create folds. Set seed only if folds≠NULL. If seed=NULL and folds≠NULL, the PXR averaged across folds is computed without seed initialization.
recall.levels	a vector with the desired recall levels (def: from:0.1, to:0.9, by:0.1).

Details

precision.at.all.recall.levels.single.class computes the precision at all recall levels just for a single class.

precision.at.given.recall.levels.over.classes computes the precision at fixed recall levels over classes.

Value

precision.at.all.recall.levels.single.class returns a two-columns matrix, representing a pair of precision and recall values. The first column is the precision, the second the recall; precision.at.given.recall.levels.over.classes returns a list with two elements:

1. average: a vector with the average precision at different recall levels across classes;
2. fixed.recall: a matrix with the precision at different recall levels: rows are classes, columns precision at different recall levels;

Examples

```
data(labels);
data(scores);
data(graph);
root <- root.node(g);
L <- L[,-which(colnames(L)==root)];
S <- S[,-which(colnames(S)==root)];
labels <- L[,1];
scores <- S[,1];
rec.levels <- seq(from=0.25, to=1, by=0.25);
pxr.single <- precision.at.all.recall.levels.single.class(labels, scores);
pxr <- precision.at.given.recall.levels.over.classes(L, S, folds=5, seed=23,
         recall.levels=rec.levels);
```

---

read.graph *Read a directed graph from a file*

Description

Read a directed graph from a file and build a graphNEL object.

Usage

```
read.graph(file = "graph.txt.gz")
```

Arguments

file	name of the file to be read. The format of the file is a sequence of rows and each row corresponds to an edge represented through a pair of vertexes separated by blanks. The extension of the file can be plain (".txt") or compressed (".gz").

Value

An object of class graphNEL.

Examples

```
ed <- system.file("extdata/graph.edges.txt.gz", package= "HEMDAG");
g <- read.graph(file=ed);
```

read.undirected.graph *Read an undirected graph from a file*

Description

Read a graph from a file and build a graphNEL object. The format of the input file is a sequence of rows. Each row corresponds to an edge represented through a pair of vertexes (blank separated) and the weight of the edge.

Usage

```
read.undirected.graph(file = "graph.txt.gz")
```

Arguments

file	name of the file to be read. The extension of the file can be plain (".txt") or compressed (".gz").

Value

A graph of class graphNEL.

Examples

```
edges <- system.file("extdata/edges.txt.gz", package="HEMDAG");
g <- read.undirected.graph(file=edges);
```

root.node *Root node*

Description

Find the root node of a directed graph.

Usage

```
root.node(g)
```

Arguments

g	a graph of class graphNEL. It represents the hierarchy of the classes.

Value

Name of the root node.

Examples

```
data(graph);
root <- root.node(g);
```

`scores.normalization` *Scores normalization function*

Description

Normalize a scores matrix w.r.t. max normalization (`maxnorm`) or quantile normalization (`qnorm`)

Usage

```
scores.normalization(norm.type = "maxnorm", S)
```

Arguments

`norm.type`	can be one of the following two values: • maxnorm (`def.`): each score is divided w.r.t. the max of each class; • qnorm: a quantile normalization is applied. Package preprocessCore is used;
`S`	A named flat scores matrix with examples on rows and classes on columns.

Details

To apply the quantile normalization the **preprocessCore** package must be properly installed.

Value

The matrix of the scores flat normalized w.r.t. `maxnorm` or `qnorm`.

Examples

```
data(scores);
norm.types <- c("maxnorm","qnorm");
for(norm.type in norm.types){
    scores.normalization(norm.type=norm.type, S=S);
}
```

specific.annotation.list

Specific annotations list

Description

Build the annotation list starting from the matrix of the most specific annotations.

Usage

```
specific.annotation.list(ann)
```

Arguments

ann	an annotation matrix (0/1). Rows are examples and columns are the most specific functional terms. It must be a named matrix.

Value

A named list, where names of each component correspond to examples (genes) and elements of each component are the associated functional terms.

Examples

```
data(labels);
spec.list <- specific.annotation.list(L);
```

specific.annotation.matrix

Specific annotation matrix

Description

Build the annotation matrix of the most specific functional terms.

Usage

```
specific.annotation.matrix(file = "gene2pheno.txt.gz")
```

Arguments

file	text file representing the associations gene-OBO terms. The file must be written as sequence of rows. Each row represents a gene/protein and all its associations with an ontology term (pipe separated), i.e. in the form *e.g.: gene1 |obo1|obo2|...|oboN*.

Details

The input plain text file (representing the associations gene-OBO terms) can be obtained by cloning the GitHub repository obogaf-parser, a perl5 module specifically designed to handle HPO and GO obo file and their gene annotation file (gaf file).

Value

The annotation matrix of the most specific annotations (0/1): rows are genes and columns are functional terms (such as GO or HPO). Let's denote M the labels matrix. If $M[i,j] = 1$, means that the gene i is annotated with the class j, otherwise $M[i,j] = 0$.

Examples

```
gene2pheno <- system.file("extdata/gene2pheno.txt.gz", package="HEMDAG");
spec.ann <- specific.annotation.matrix(file=gene2pheno);
```

---

stratified.cross.validation

Stratified cross validation

Description

Generate data for the stratified cross-validation.

Usage

```
stratified.cv.data.single.class(examples, positives, kk = 5, seed = NULL)

stratified.cv.data.over.classes(labels, examples, kk = 5, seed = NULL)
```

Arguments

examples	indices or names of the examples. Can be either a vector of integers or a vector of names.
positives	vector of integers or vector of names. The indices (or names) refer to the indices (or names) of 'positive' examples.
kk	number of folds (def. kk=5).
seed	seed of the random generator (def. seed=NULL). If is set to NULL no initialization is performed.
labels	labels matrix. Rows are genes and columns are classes. Let's denote M the labels matrix. If $M[i,j] = 1$, means that the gene i is annotated with the class j, otherwise $M[i,j] = 0$.

Details

Folds are *stratified*, i.e. contain the same amount of positive and negative examples.

Value

`stratified.cv.data.single.class` returns a list with 2 two component:

- fold.non.positives: a list with k components. Each component is a vector with the indices (or names) of the non-positive elements. Indexes (or names) refer to row numbers (or names) of a data matrix;
- fold.positives: a list with k components. Each component is a vector with the indices (or names) of the positive elements. Indexes (or names) refer to row numbers (or names) of a data matrix;

`stratified.cv.data.over.classes` returns a list with n components, where n is the number of classes of the labels matrix. Each component n is in turn a list with k elements, where k is the number of folds. Each fold contains an equal amount of positives and negatives examples.

Examples

```
data(labels);
examples.index <- 1:nrow(L);
examples.name <- rownames(L);
positives <- which(L[,3]==1);
x <- stratified.cv.data.single.class(examples.index, positives, kk=5, seed=23);
y <- stratified.cv.data.single.class(examples.name, positives, kk=5, seed=23);
z <- stratified.cv.data.over.classes(L, examples.index, kk=5, seed=23);
k <- stratified.cv.data.over.classes(L, examples.name, kk=5, seed=23);
```

`tpr.dag` *TPR-DAG ensemble variants*

Description

Collection of the true-path-rule-based hierarchical learning ensemble algorithms and its variants.

`TPR-DAG` is a family of algorithms on the basis of the choice of the **bottom-up** step adopted for the selection of *positive* children (or descendants) and of the **top-down** step adopted to assure ontology-based predictions. Indeed, in their more general form the `TPR-DAG` algorithms adopt a two step learning strategy:

1. in the first step they compute a *per-level bottom-up* visit from leaves to root to propagate *positive* predictions across the hierarchy;
2. in the second step they compute a *per-level top-down* visit from root to leaves in order to assure the consistency of the predictions.

It is worth noting that levels (both in the first and second step) are defined in terms of the maximum distance from the root node (see `graph.levels`).

Usage

```
tpr.dag(
  S,
  g,
  root = "00",
  positive = "children",
  bottomup = "threshold.free",
  topdown = "gpav",
  t = 0,
  w = 0,
  W = NULL,
  parallel = FALSE,
  ncores = 1
)
```

Arguments

S	a named flat scores matrix with examples on rows and classes on columns.
g	a graph of class graphNEL. It represents the hierarchy of the classes.
root	name of the class that it is on the top-level of the hierarchy (def. root="00").
positive	choice of the *positive* nodes to be considered in the bottom-up strategy. Can be one of the following values: • children (def.): positive children are are considered for each node; • descendants: positive descendants are are considered for each node;
bottomup	strategy to enhance the flat predictions by propagating the positive predictions from leaves to root. It can be one of the following values: • threshold.free (def.): positive nodes are selected on the basis of the threshold.free strategy; • threshold: positive nodes are selected on the basis of the threshold strategy; • weighted.threshold.free: positive nodes are selected on the basis of the weighted.threshold.free strategy; • weighted.threshold: positive nodes are selected on the basis of the weighted.threshold strategy; • tau: positive nodes are selected on the basis of the tau strategy. NOTE: tau is only a DESCENS variant. If you select tau strategy you must set positive=descendants;
topdown	strategy to make scores "hierarchy-aware". It can be one of the following values: • htd: HTD-DAG strategy is applied (htd); • gpav (def.): GPAV strategy is applied (gpav);
t	threshold for the choice of positive nodes (def. t=0). Set t only for the variants requiring a threshold for the selection of the positive nodes, otherwise set t=0.
w	weight to balance between the contribution of the node i and that of its positive nodes. Set w only for the *weighted* variants, otherwise set w=0.

`W` vector of weight relative to a single example. If `W=NULL` (def.) it is assumed that `W` is a unitary vector of the same length of the columns' number of the matrix `S` (root node included). Set `W` only if `topdown=gpav`.

`parallel` a boolean value:

- `TRUE`: execute the parallel implementation of GPAV (`gpav.parallel`);
- `FALSE` (def.): execute the sequential implementation of GPAV (`gpav.over.examples`);

Use `parallel` only if `topdown=GPAV`; otherwise set `parallel=FALSE`.

`ncores` number of cores to use for parallel execution. Set `ncores=1` if `parallel=FALSE`, otherwise set `ncores` to the desired number of cores. Set `ncores` if and only if `topdown=GPAV`; otherwise set `ncores=1`.

Details

The *vanilla* `TPR-DAG` adopts a per-level bottom-up traversal of the DAG to correct the flat predictions $\hat{y}_i$ according to the following formula:

$$\bar{y}_i := \frac{1}{1+|\phi_i|}(\hat{y}_i + \sum_{j \in \phi_i} \bar{y}_j)$$

where ϕ_i are the positive children of i. Different strategies to select the positive children ϕ_i can be applied:

1. **threshold-free** strategy: the positive nodes are those children that can increment the score of the node i, that is those nodes that achieve a score higher than that of their parents:

$$\phi_i := \{j \in child(i) | \bar{y}_j > \hat{y}_i\}$$

2. **threshold** strategy: the positive children are selected on the basis of a threshold that can be selected in two different ways:
   (a) for each node a constant threshold $\bar{t}$ is a priori selected:

$$\phi_i := \{j \in child(i) | \bar{y}_j > \bar{t}\}$$

   For instance if the predictions represent probabilities it could be meaningful to a priori select $\bar{t} = 0.5$.
   (b) the threshold is selected to maximize some performance metric $\mathcal{M}$ estimated on the training data, as for instance the Fmax or the AUPRC. In other words the threshold is selected to maximize some measure of accuracy of the predictions $\mathcal{M}(j,t)$ on the training data for the class j with respect to the threshold t. The corresponding set of positives $\forall i \in V$ is:

$$\phi_i := \{j \in child(i) | \bar{y}_j > t_j^*, t_j^* = \arg\max_t \mathcal{M}(j,t)\}$$

   For instance t_j^* can be selected from a set of $t \in (0,1)$ through internal cross-validation techniques.

The weighted `TPR-DAG` version can be designed by adding a weight $w \in [0,1]$ to balance between the contribution of the node i and that of its positive children ϕ, through their convex combination:

$$\bar{y}_i := w\hat{y}_i + \frac{(1-w)}{|\phi_i|} \sum_{j \in \phi_i} \bar{y}_j$$

If $w = 1$ no weight is attributed to the children and the TPR-DAG reduces to the HTD-DAG algorithm, since in this way only the prediction for node i is used in the bottom-up step of the algorithm. If $w = 0$ only the predictors associated to the children nodes vote to predict node i. In the intermediate cases we attribute more importance to the predictor for the node i or to its children depending on the values of w. By combining the weighted and the threshold variant, we design the weighted-threshold variant.

Since the contribution of the descendants of a given node decays exponentially with their distance from the node itself, to enhance the contribution of the most specific nodes to the overall decision of the ensemble we design the ensemble variant DESCENS. The novelty of DESCENS consists in strongly considering the contribution of all the descendants of each node instead of only that of its children. Therefore DESCENS predictions are more influenced by the information embedded in the leaves nodes, that are the classes containing the most informative and meaningful information from a biological and medical standpoint. For the choice of the "positive" descendants we use the same strategies adopted for the selection of the "positive" children shown above. Furthermore, we designed a variant specific only for DESCENS, that we named DESCENS-τ. The DESCENS-τ variant balances the contribution between the "positives" children of a node i and that of its "positives" descendants excluding its children by adding a weight $\tau \in [0, 1]$:

$$\bar{y}_i := \frac{\tau}{1 + |\phi_i|}(\hat{y}_i + \sum_{j \in \phi_i} \bar{y}_j) + \frac{1 - \tau}{1 + |\delta_i|}(\hat{y}_i + \sum_{j \in \delta_i} \bar{y}_j)$$

where ϕ_i are the "positive" children of i and $\delta_i = \Delta_i \setminus \phi_i$ the descendants of i without its children. If $\tau = 1$ we consider only the contribution of the "positive" children of i; if $\tau = 0$ only the descendants that are not children contribute to the score, while for intermediate values of τ we can balance the contribution of ϕ_i and δ_i positive nodes.

Simply by replacing the HTD-DAG top-down step (htd) with the GPAV approach (gpav) we design the ISO-TPR variant. The most important feature of ISO-TPR is that it maintains the hierarchical constraints by construction and it selects the closest solution (in the least square sense) to the bottom-up predictions that obeys the *True Path Rule*.

Value

A named matrix with the scores of the classes corrected according to the chosen TPR-DAG ensemble algorithm.

See Also

gpav, htd

Examples

```
data(graph);
data(scores);
data(labels);
root <- root.node(g);
S.tpr <- tpr.dag(S, g, root, positive="children", bottomup="threshold.free",
topdown="gpav", t=0, w=0, W=NULL, parallel=FALSE, ncores=1);
```

tpr.dag.cv *TPR-DAG cross-validation experiments*

Description

Correct the computed scores in a hierarchy according to the a TPR-DAG ensemble variant.

Usage

```
tpr.dag.cv(
  S,
  g,
  ann,
  norm = FALSE,
  norm.type = NULL,
  positive = "children",
  bottomup = "threshold",
  topdown = "gpav",
  W = NULL,
  parallel = FALSE,
  ncores = 1,
  threshold = seq(from = 0.1, to = 0.9, by = 0.1),
  weight = 0,
  kk = 5,
  seed = 23,
  metric = "auprc",
  n.round = NULL
)
```

Arguments

S	a named flat scores matrix with examples on rows and classes on columns.
g	a graph of class graphNEL. It represents the hierarchy of the classes.
ann	an annotation matrix: rows correspond to examples and columns to classes. $ann[i,j] = 1$ if example i belongs to class j, $ann[i,j] = 0$ otherwise. ann matrix is necessary to maximize the hyper-parameter(s) of the chosen parametric TPR-DAG ensemble variant respect to the metric selected in metric. For the parametric-free ensemble variant set ann=NULL.
norm	a boolean value. Should the flat score matrix be normalized? By default norm=FALSE. If norm=TRUE the matrix S is normalized according to the normalization type selected in norm.type.
norm.type	a string character. It can be one of the following values: 1. NULL (def.): none normalization is applied (norm=FALSE) 2. maxnorm: each score is divided for the maximum value of each class (scores.normalization); 3. qnorm: quantile normalization. **preprocessCore** package is used (scores.normalization);

`positive` choice of the *positive* nodes to be considered in the bottom-up strategy. Can be one of the following values:

- `children` (`def.`): positive children are are considered for each node;
- `descendants`: positive descendants are are considered for each node;

`bottomup` strategy to enhance the flat predictions by propagating the positive predictions from leaves to root. It can be one of the following values:

- `threshold.free`: positive nodes are selected on the basis of the `threshold.free` strategy;
- `threshold` (`def.`): positive nodes are selected on the basis of the `threshold` strategy;
- `weighted.threshold.free`: positive nodes are selected on the basis of the `weighted.threshold.free` strategy;
- `weighted.threshold`: positive nodes are selected on the basis of the `weighted.threshold` strategy;
- `tau`: positive nodes are selected on the basis of the `tau` strategy. NOTE: `tau` is only a `DESCENS` variant. If you select `tau` strategy you must set `positive=descendants`;

`topdown` strategy to make the scores hierarchy-consistent. It can be one of the following values:

- `htd`: `HTD-DAG` strategy is applied (`htd`);
- `gpav` (`def.`): `GPAV` strategy is applied (`gpav`);

`W` vector of weight relative to a single example. If `W=NULL` (def.) it is assumed that `W` is a unitary vector of the same length of the columns' number of the matrix `S` (root node included). Set `W` only if `topdown=gpav`.

`parallel` a boolean value:

- `TRUE`: execute the parallel implementation of GPAV (`gpav.parallel`);
- `FALSE` (def.): execute the sequential implementation of GPAV (`gpav.over.examples`);

Use `parallel` only if `topdown=gpav`; otherwise set `parallel=FALSE`.

`ncores` number of cores to use for parallel execution. Set `ncores=1` if `parallel=FALSE`, otherwise set `ncores` to the desired number of cores. Set `ncores` if `topdown=gpav`, otherwise set `ncores=1`.

`threshold` range of threshold values to be tested in order to find the best threshold (`def: from:0.1, to:0.9, by:0.1`). The denser the range is, the higher the probability to find the best threshold is, but the execution time will be higher. For the *threshold-free* variants, set `threshold=0`.

`weight` range of weight values to be tested in order to find the best weight (`def: from:0.1, to:0.9, by:0.1`). The denser the range is, the higher the probability to find the best threshold is, but the execution time will be higher. For the *weight-free* variants, set `weight=0`.

`kk` number of folds of the cross validation (`def: kk=5`) on which tuning the parameters `threshold`, `weight` and `tau` of the parametric ensemble variants. For the parametric-free variants (i.e. if `bottomup = threshold.free`), set `kk=NULL`.

seed	initialization seed for the random generator to create folds (def. 23). If seed=NULL folds are generated without seed initialization. If bottomup=threshold.free, set seed=NULL.
metric	a string character specifying the performance metric on which maximizing the parametric ensemble variant. It can be one of the following values: 1. auprc (def.): the parametric ensemble variant is maximized on the basis of AUPRC (auprc); 2. fmax: the parametric ensemble variant is maximized on the basis of Fmax (multilabel.F.measure; 3. NULL: threshold.free variant is parameter-free, so none optimization is needed.
n.round	number of rounding digits (def. 3) to be applied to the hierarchical scores matrix for choosing the best threshold on the basis of the best Fmax. If bottomup==threshold.free or metric="auprc", set n.round=NULL.

Details

The parametric hierarchical ensemble variants are cross-validated maximizing the parameter on the metric selected in metric.

Value

A named matrix with the scores of the functional terms corrected according to the chosen TPR-DAG ensemble algorithm.

Examples

```
data(graph);
data(scores);
data(labels);
S.tpr <- tpr.dag.cv(S, g, ann=NULL, norm=FALSE, norm.type=NULL, positive="children",
bottomup="threshold.free", topdown="gpav", W=NULL, parallel=FALSE, ncores=1,
threshold=0, weight=0, kk=NULL, seed=NULL, metric=NULL, n.round=NULL);
```

---

tpr.dag.holdout *TPR-DAG holdout experiments*

Description

Correct the computed scores in a hierarchy according to the selected TPR-DAG ensemble variant by applying a classical holdout procedure.

Usage

```
tpr.dag.holdout(
  S,
  g,
  ann,
  testIndex,
  norm = FALSE,
  norm.type = NULL,
  W = NULL,
  parallel = FALSE,
  ncores = 1,
  positive = "children",
  bottomup = "threshold",
  topdown = "htd",
  threshold = seq(from = 0.1, to = 0.9, by = 0.1),
  weight = seq(from = 0.1, to = 0.9, by = 0.1),
  kk = 5,
  seed = 23,
  metric = "auprc",
  n.round = NULL
)
```

Arguments

S	a named flat scores matrix with examples on rows and classes on columns.
g	a graph of class graphNEL. It represents the hierarchy of the classes.
ann	an annotation matrix: rows correspond to examples and columns to classes. $ann[i,j] = 1$ if example i belongs to class j, $ann[i,j] = 0$ otherwise. ann matrix is necessary to maximize the hyper-parameter(s) of the chosen parametric TPR-DAG ensemble variant respect to the metric selected in metric. For the parametric-free ensemble variant set ann=NULL.
testIndex	a vector of integer numbers corresponding to the indexes of the elements (rows) of the scores matrix S to be used in the test set.
norm	a boolean value. Should the flat score matrix be normalized? By default norm=FALSE. If norm=TRUE the matrix S is normalized according to the normalization type selected in norm.type.
norm.type	a string character. It can be one of the following values: 1. NULL (def.): none normalization is applied (norm=FALSE) 2. maxnorm: each score is divided for the maximum value of each class; 3. qnorm: quantile normalization. **preprocessCore** package is used;
W	vector of weight relative to a single example. If W=NULL (def.) it is assumed that W is a unitary vector of the same length of the columns' number of the matrix S (root node included). Set W only if topdown=gpav.
parallel	a boolean value: • TRUE: execute the parallel implementation of GPAV (gpav.parallel);

- FALSE (def.): execute the sequential implementation of GPAV (gpav.over.examples);

Use parallel only if topdown=gpav; otherwise set parallel=FALSE.

ncores — number of cores to use for parallel execution. Set ncores=1 if parallel=FALSE, otherwise set ncores to the desired number of cores. Set ncores if and only if topdown=gpav; otherwise set ncores=1.

positive — choice of the *positive* nodes to be considered in the bottom-up strategy. Can be one of the following values:

- children (def.): positive children are are considered for each node;
- descendants: positive descendants are are considered for each node;

bottomup — strategy to enhance the flat predictions by propagating the positive predictions from leaves to root. It can be one of the following values:

- threshold.free: positive nodes are selected on the basis of the threshold.free strategy (def.);
- threshold (def.): positive nodes are selected on the basis of the threshold strategy;
- weighted.threshold.free: positive nodes are selected on the basis of the weighted.threshold.free strategy;
- weighted.threshold: positive nodes are selected on the basis of the weighted.threshold strategy;
- tau: positive nodes are selected on the basis of the tau strategy. NOTE: tau is only a DESCENS variant. If you select tau strategy you must set positive=descendants;

topdown — strategy to make the scores hierarchy-consistent. It can be one of the following values:

- htd: HTD-DAG strategy is applied (htd);
- gpav (def.): GPAV strategy is applied (gpav);

threshold — range of threshold values to be tested in order to find the best threshold (def: from:0.1, to:0.9, by:0.1). The denser the range is, the higher the probability to find the best threshold is, but the execution time will be higher. For the *threshold-free* variants, set threshold=0.

weight — range of weight values to be tested in order to find the best weight (def: from:0.1, to:0.9, by:0.1). The denser the range is, the higher the probability to find the best threshold is, but the execution time will be higher. For the *weight-free* variants, set weight=0.

kk — number of folds of the cross validation (def: kk=5) on which tuning the parameters threshold, weight and tau of the parametric ensemble variants. For the parametric-free variants (i.e. if bottomup = threshold.free), set kk=NULL.

seed — initialization seed for the random generator to create folds (def. 23). If seed=NULL folds are generated without seed initialization. If bottomup=threshold.free, set seed=NULL.

metric — a string character specifying the performance metric on which maximizing the parametric ensemble variant. It can be one of the following values:

1. auprc (def.): the parametric ensemble variant is maximized on the basis of AUPRC (auprc);

2. fmax: the parametric ensemble variant is maximized on the basis of Fmax (multilabel.F.measure;
3. NULL: threshold.free variant is parameter-free, so none optimization is needed.

n.round — number of rounding digits (def. 3) to be applied to the hierarchical scores matrix for choosing the best threshold on the basis of the best Fmax. If bottomup==threshold.free or metric="auprc", set n.round=NULL.

Details

The parametric hierarchical ensemble variants are cross-validated maximizing the parameter on the metric selected in metric,

Value

A named matrix with the scores of the classes corrected according to the chosen TPR-DAG ensemble algorithm. Rows of the matrix are shrunk to testIndex.

Examples

```
data(graph);
data(scores);
data(labels);
data(test.index);
S.tpr <- tpr.dag.holdout(S, g, ann=NULL, testIndex=test.index, norm=FALSE, norm.type=NULL,
positive="children", bottomup="threshold.free", topdown="gpav", W=NULL, parallel=FALSE,
ncores=1, threshold=0, weight=0, kk=NULL, seed=NULL, metric=NULL, n.round=NULL);
```

---

transitive.closure.annotations

Transitive closure of annotations

Description

Perform the transitive closure of the annotations using ancestors and the most specific annotation matrix. The annotations are propagated from bottom to top, enriching the most specific annotations table. Rows correspond to genes and columns to functional terms.

Usage

```
transitive.closure.annotations(ann.spec, anc)
```

Arguments

ann.spec — the annotation matrix of the most specific annotations (0/1): rows are genes and columns are functional terms.

anc — the ancestor list.

Value

The annotation table T: rows correspond to genes and columns to OBO terms. $T[i, j] = 1$ means that gene i is annotated for the term j, $T[i, j] = 0$ means that gene i is not annotated for the term j.

Examples

```
data(graph);
data(labels);
anc <- build.ancestors(g);
tca <- transitive.closure.annotations(L, anc);
```

---

tupla.matrix *Tupla matrix*

Description

Transform a named score matrix in a tupla, i.e. in the form `nodeX nodeY score`.

Usage

```
tupla.matrix(m, output.file = "net.file.gz", digits = 3)
```

Arguments

`m`	a named score matrix. It can be either a `m` x `n` matrix (where `m` are example and `n` are functional terms, e.g. GO terms) or it can be a square named matrix `m` x `m`, where `m` are examples.
`output.file`	name of the file on which the matrix must be written.
`digits`	number of digits to be used to save scores of `m` (`def. digits=3`). The extension of the file can be plain (".txt") or compressed (".gz").

Details

Only the *non-zero* interactions are kept, while the *zero* interactions are discarded.

Value

A tupla score matrix stored in output.file.

Examples

```
data(wadj);
file <- tempfile();
tupla.matrix(W, output.file=file, digits=3);
```

unstratified.cv.data *Unstratified cross validation*

Description

This function splits a dataset in k-fold in an unstratified way, i.e. a fold does not contain an equal amount of positive and negative examples. This function is used to perform k-fold cross-validation experiments in a hierarchical correction contest where splitting dataset in a stratified way is not needed.

Usage

```
unstratified.cv.data(S, kk = 5, seed = NULL)
```

Arguments

S	matrix of the flat scores. It must be a named matrix, where rows are example (e.g. genes) and columns are classes/terms (e.g. GO terms).
kk	number of folds in which to split the dataset (def. k=5).
seed	seed for random generator. If NULL (def.) no initialization is performed.

Value

A list with $k = kk$ components (folds). Each component of the list is a character vector contains the index of the examples, i.e. the index of the rows of the matrix S.

Examples

```
data(scores);
foldIndex <- unstratified.cv.data(S, kk=5, seed=23);
```

weighted.adjacency.matrix *Weighted adjacency matrix*

Description

Build a symmetric weighted adjacency matrix (wadj matrix) of a graph.

Usage

```
weighted.adjacency.matrix(file = "edges.txt")
```

Arguments

file	name of the plain text file to be read (def. edges). The format of the file is a sequence of rows. Each row corresponds to an edge represented through a pair of vertexes (blank separated) and the weight of the edges. For instance: nodeX nodeY score. The file extension can be plain (".txt") or compressed (".gz").

Value

A named symmetric weighted adjacency matrix of the graph.

Examples

```
edges <- system.file("extdata/edges.txt.gz", package="HEMDAG");
W <- weighted.adjacency.matrix(file=edges);
```

---

write.graph *Write a directed graph on file*

Description

Read an object of class graphNEL and write the graph as sequence of rows on a plain text file.

Usage

```
write.graph(g, file = "graph.txt.gz")
```

Arguments

g	a graph of class graphNEL.
file	name of the file to be written. The extension of the file can be plain (".txt") or compressed (".gz").

Value

A plain text file representing the graph. Each row corresponds to an edge represented through a pair of vertexes separated by blank.

Examples

```
data(graph);
file <- tempfile();
write.graph(g, file=file);
```

Index