Package 'CTD'

September 8, 2020

Title CTD method for "connecting the dots" in weighted graphs

Version 1.0.0

Date 2017-05-25

Author Lillian Thistlethwaite [aut, cre]

Maintainer Lillian Thistlethwaite <lillian.thistlethwaite@bcm.edu>

Description An R package for pattern discovery in weighted graphs. Two use cases are achieved: 1) Given a weighted graph and a subset of its nodes, do the nodes show significant connectedness? 2) Given a weighted graph and two subsets of its nodes, are the subsets close neighbors or distant?

Depends R (>= 4.0), gmp, igraph, stats, grDevices, graphics

Suggests knitr,
rmarkdown,
huge,
plotly,
gplots,
RColorBrewer,
testthat

VignetteBuilder rmarkdown

biocViews BiomedicalInformatics, Metabolomics, SystemsBiology, GraphAndNetwork

License MIT + file LICENSE

Encoding UTF-8

LazyData true

RoxygenNote 7.1.1

R topics documented:

`cohorts_coded` *Disease cohorts with coded identifiers*

Description

Patient sample identifiers mapped to known clinical diagnoses.

Usage

```
data(cohorts_coded)
```

Format

cohorts_coded - A list object where elements contain all patient IDs associated with a given diagnosis, as included in the dataset Thistlethwaite2020.

Examples

```
data(cohorts_coded)
```

`data.combineData` *Combine datasets*

Description

Combine datasets

Usage

```
data.combineData(curr_data, more_data)
```

Arguments

`curr_data`	- Current data matrix
`more_data`	- Data matrix you want to combine with curr_data.

Value

combined.data - Combined data matrix.

Examples

```
# Row names and column names are required for both input matrices.
curr_data=matrix(rnorm(500), ncol=100)
rownames(curr_data)=sprintf("Feature%d",sample(seq_len(20),
                            nrow(curr_data),replace = FALSE))
colnames(curr_data)=sprintf("Sample%d", seq_len(ncol(curr_data)))
more_data=matrix(rnorm(500), ncol=100)
rownames(more_data)=sprintf("Feature%d",sample(seq_len(20),
                            nrow(curr_data),replace = FALSE))
colnames(more_data) = sprintf("Sample%d", seq_len(ncol(curr_data)))
combined.data = data.combineData(curr_data, more_data)
```

---

data.imputeData *Impute missing values*

Description

Impute missing values as lowest observed value in a reference population

Usage

```
data.imputeData(data, ref)
```

Arguments

data	- Normalized, imputed data. Data matrix with observations as rows, features as columns.
ref	- Reference samples normalized, imputed data.

Value

imputed.data - Z-transformed data.

Examples

```
data(Thistlethwaite2020)
data_mx = Thistlethwaite2020
# Data with missing values
dt_w_missing_vals = data_mx[,-seq_len(8)]
# Reference data can also have missing values
ref_data = data_mx[,grep("EDTA-REF", colnames(data_mx))]
fil.rate = apply(ref_data, 1, function(i) sum(is.na(i))/length(i))
# Can only impute data that are found in reference samples
dt_w_missing_vals = dt_w_missing_vals[which(fil.rate<1.0),]
ref_data = ref_data[which(fil.rate<1.0),]
imputed.data = data.imputeData(dt_w_missing_vals, ref_data)
print(any(is.na(imputed.data)))
```

data.surrogateProfiles *Generate surrogate profiles*

Description

Fill in a data matrix rank with surrogate profiles., when your data is low n, high p.

Usage

```
data.surrogateProfiles(data, std = 1, ref_data = NULL)
```

Arguments

data	- Data matrix with observations (e.g., patient samples) as columns, features (e.g., metabolites or genes) as rows
std	- The level of variability (standard deviation) around each observed feature's z-score you want to add to generate the surrogate profiles.
ref_data	- Data matrix for healthy control "reference" samples, observations (e.g., patient samples) as columns, features (e.g., metabolites or genes) as rows

Value

data_mx_surr - Data matrix with added surrogate profiles.

Examples

```
data("Miller2015")
data_mx=Miller2015[-1,grep("IEM_", colnames(Miller2015))]
data_mx=apply(data_mx, c(1,2), as.numeric)
diags=unlist(Miller2015["diagnosis",grep("IEM_", colnames(Miller2015))])
refs=data_mx[,which(diags=="No biochemical genetic diagnosis")]
ref_fill=as.numeric(Miller2015$`Times identifed in all 200 samples`[-1])/200
refs2=refs[which(ref_fill>0.8),]
diag_pts=names(diags[which(diags==unique(diags)[1])])
diag_data=data_mx[which(rownames(data_mx) %in% rownames(refs2)),
                  which(colnames(data_mx) %in% diag_pts)]
data_mx_surr=data.surrogateProfiles(data=diag_data, std=1, ref_data=refs2)
```

data.zscoreData *Z-transform available data*

Description

The z-transform is meant to work with normalized, imputed metabolomics data

Usage

```
data.zscoreData(data, ref)
```

Arguments

data	- Normalized, imputed data. Data matrix with observations as rows, features as columns.
ref	- Reference samples normalized, imputed data.

Value

zscored.data - Z-transformed data.

Examples

```
dis_data = matrix(rexp(500), ncol=100)
rownames(dis_data)=sprintf("Feature%d",seq_len(nrow(dis_data)))
colnames(dis_data)=sprintf("Sample%d",seq_len(ncol(dis_data)))
ref_data = matrix(rexp(500), ncol=100)
rownames(ref_data)=sprintf("Feature%d",seq_len(nrow(ref_data)))
colnames(ref_data)=sprintf("Sample%d",seq_len(ncol(ref_data)))
zscored.data=data.zscoreData(dis_data,ref_data)
```

---

graph.connectToExt *Connect a node to its unvisited "extended" neighbors*

Description

Connect a node to its unvisited "extended" neighbors

Usage

```
graph.connectToExt(adj_mat, startNode, visitedNodes)
```

Arguments

adj_mat	- The adjacency matrix that encodes the edge weights for the network.
startNode	- The node most recently visited by the network walker, from which p1 gets dispersed.
visitedNodes	- The history of previous draws in the node ranking sequence.

Value

adj_matAfter - The adjacency matrix where the startNode is now connected to its unvisited "extended" neighbors. An extended neighbor is the neighbor of a neighbor.

Examples

```
adj_mat = rbind(c(0,2,1,0,0,0,0), # A
                c(2,0,1,0,0,0,0), # B
                c(1,0,0,1,0,0,0), # C
                c(0,0,1,0,2,0,0), # D
                c(0,0,0,2,0,2,1), # E
                c(0,0,0,1,2,0,1), # F
                c(0,0,0,0,1,1,0)  # G
                )
```

```
rownames(adj_mat) = c("A", "B", "C", "D", "E", "F", "G")
colnames(adj_mat) = c("A", "B", "C", "D", "E", "F", "G")
ig = graph.adjacency(as.matrix(adj_mat), mode="undirected",weighted=TRUE)
G=vector(mode="list", length=7)
G[seq_len(length(G))] = 0
names(G) = c("A", "B", "C", "D", "E", "F", "G")
startNode = "A"
visitedNodes = c("B", "C")
coords = layout.fruchterman.reingold(ig)
V(ig)$x = coords[,1]
V(ig)$y = coords[,2]
adj_matAfter = graph.connectToExt(adj_mat, startNode, visitedNodes)
```

`graph.diffuseP1` *Diffuse Probability P1 from a starting node*

Description

Recursively diffuse probability from a starting node based on the connectivity of the network, representing the likelihood that a variable is most influenced by a perturbation in the starting node.

Usage

```
graph.diffuseP1(
  p1,
  sn,
  G,
  vNodes,
  thresholdDiff,
  adj_mat,
  verbose = FALSE,
  out_dir = "",
  r_level = 1,
  coords = NULL
)
```

Arguments

p1	- The probability being dispersed from the starting node, sn, which is preferentially distributed between network nodes by the probability diffusion algorithm based solely on network connectivity.
sn	- "Start node", or the node most recently visited by the network walker, from which p1 gets dispersed.
G	- A list of probabilities, with names of the list being the node names in the network.
vNodes	- "Visited nodes", or the history of previous draws in the node ranking sequence.
thresholdDiff	- When the probability diffusion algorithm exchanges this amount (thresholdDiff) or less between nodes, the algorithm returns up the call stack.
adj_mat	- The adjacency matrix that encodes the edge weights for the network, G.
verbose	- If debugging or tracking a diffusion event, verbose=TRUE will activate print statements. Default is FALSE.

out_dir	- If specified, a image sequence will generate in the output directory specified.
r_level	- "Recursion level", or the current depth in the call stack caused by a recursive algorithm. Only relevant if out_dir is specified.
coords	- The x and y coordinates for each node in the network, to remain static between images. Only relevant if out_dir is specified.

Value

G - A list of returned probabilities after the diffusion of probability has truncated, with names of the list being the node names in the network.

Examples

```
# Read in any network via its adjacency matrix
adj_mat=matrix(1, nrow=100, ncol=100)
for (i in seq_len(100)){for (j in seq_len(100)){adj_mat[i,j]=rnorm(1,0,1)}}
colnames(adj_mat)=sprintf("Metabolite%d", seq_len(100))
rownames(adj_mat)=colnames(adj_mat)
G=vector(mode="list", length=ncol(adj_mat))
names(G)=colnames(adj_mat)
G=lapply(G, function(i) i[[1]]=0)
probs_afterCurrDraw=graph.diffuseP1(p1=1.0, sn=names(G)[1], G=G,
                                    vNodes=names(G)[1],
                                    thresholdDiff=0.01, adj_mat, TRUE)
# Make a movie of the diffusion of probability from sn
.GlobalEnv$imgNum = 1
ig=graph.adjacency(adj_mat,mode="undirected",weighted=TRUE,
                   add.colnames="name")
coords = layout.fruchterman.reingold(ig)
probs_afterCurrDraw=graph.diffuseP1(p1=1.0, sn=names(G)[1], G=G,
                                    vNodes=names(G)[1],
                                    thresholdDiff=0.01, adj_mat, TRUE,
                                    getwd(), 1, coords)
```

---

graph.diffusionSnapShot

Capture the current state of probability diffusion

Description

Recursively diffuse probability from a starting node based on the connectivity in a network, G, where the probability represents the likelihood that a variable will be influenced by a perturbation in the starting node.

Usage

```
graph.diffusionSnapShot(
  adj_mat,
  G,
  output_dir,
  p1,
  startNode,
```

```
  visitedNodes,
  recursion_level = 1,
  coords
)
```

Arguments

adj_mat	- The adjacency matrix that encodes the edge weights for the network, G.
G	- A list of probabilities, with names of the list being the node names in the network.
output_dir	- The local directory at which you want still PNG images to be saved.
p1	- The probability being dispersed from the starting node, startNode, which is preferentially distributed between network nodes by the probability diffusion algorithm based solely on network connectivity.
startNode	- The first variable drawn in the node ranking, from which p1 gets dispersed.
visitedNodes	- A character vector of node names, storing the history of previous draws in the node ranking.
recursion_level	- The current depth in the call stack caused by a recursive algorithm.
coords	- The x and y coordinates for each node in the network, to remain static between images.

Value

0

Examples

```
# 7 node example graph illustrating diffusion of probability based on
# network connectivity.
adj_mat = rbind(c(0,2,1,0,0,0,0), # A
                c(2,0,1,0,0,0,0), # B
                c(1,0,0,1,0,0,0), # C
                c(0,0,1,0,2,0,0), # D
                c(0,0,0,2,0,2,1), # E
                c(0,0,0,1,2,0,1), # F
                c(0,0,0,0,1,1,0)  # G
                )
rownames(adj_mat) = c("A", "B", "C", "D", "E", "F", "G")
colnames(adj_mat) = c("A", "B", "C", "D", "E", "F", "G")
ig = graph.adjacency(as.matrix(adj_mat),mode="undirected",weighted=TRUE)
G=vector(mode="list", length=7)
G[seq_len(length(G))] = 0
names(G) = c("A", "B", "C", "D", "E", "F", "G")
coords = layout.fruchterman.reingold(ig)
V(ig)$x = coords[,1]
V(ig)$y = coords[,2]
.GlobalEnv$imgNum = 1
graph.diffusionSnapShot(adj_mat,G,getwd(),1.0,"A","A",1,coords)
```

graph.naivePruning *Network pruning for disease-specific network determination*

Description

Prune edges from a disease+control "differential" network that also occur in the control-only network.

Usage

```
graph.naivePruning(ig_dis, ig_ref)
```

Arguments

ig_dis	- The igraph object associated with the disease+reference trained differential network.
ig_ref	- The igraph object associated with the reference-only trained interaction network.

Value

ig_pruned - The pruned igraph object of the disease+reference differential network, with reference edges subtracted.

Examples

```
# Generate a 100 node "disease-control" network
adj_mat=matrix(0, nrow=100, ncol=100)
rows = sample(seq_len(100), 50, replace=TRUE)
cols = sample(seq_len(100), 50, replace=TRUE)
for (i in rows) {for (j in cols){adj_mat[i,j]=rnorm(1,0,1)}}
colnames(adj_mat)=sprintf("Metabolite%d", seq_len(100))
ig_dis = graph.adjacency(adj_mat, mode="undirected", weighted=TRUE)
# Generate a 100 node reference "control-only" network
adj_mat2=matrix(0, nrow=100, ncol=100)
rows2 = sample(seq_len(100), 50, replace=TRUE)
cols2 = sample(seq_len(100), 50, replace=TRUE)
for (i in rows2) {for (j in cols2){adj_mat2[i,j]=rnorm(1,0,1)}}
colnames(adj_mat2)=sprintf("Metabolite%d", seq_len(100))
ig_ref = graph.adjacency(adj_mat2, mode="undirected", weighted=TRUE)
ig_pruned=graph.naivePruning(ig_dis, ig_ref)
```

graph.netWalkSnapShot *Capture the current location of a network walker*

Description

A network walker steps towards the node that inherited the highest probability from the last node that it stepped into.

Usage

```
graph.netWalkSnapShot(
  adj_mat,
  G,
  output_dir,
  p1,
  visitedNodes,
  S,
  coords,
  imgNum = 1,
  useLabels = TRUE
)
```

Arguments

adj_mat	- The adjacency matrix that encodes the edge weights for the network, G.
G	- A list of probabilities, with names of the list being the node names in the network.
output_dir	- The local directory at which you want still PNG images to be saved.
p1	- The probability being dispersed from the starting node, startNode, which is preferentially distributed between network nodes by the probability diffusion algorithm based solely on network connectivity.
visitedNodes	- A character vector of node names, storing the history of previous draws in the node ranking.
S	- A character vector of node names in the subset you want the network walker to find.
coords	- The x and y coordinates for each node in the network, to remain static between images.
imgNum	- The image number for this snapshot. If images are being generated in a sequence, this serves as an iterator for file naming.
useLabels	- If TRUE, node names will display next to their respective nodes in the network. If FALSE, node names will not display.

Value

0

Examples

```
# 7 node example graph illustrating diffusion of probability based on network
# connectivity
adj_mat = rbind(c(0,2,1,0,0,0,0), # A
                c(2,0,1,0,0,0,0), # B
                c(1,0,0,1,0,0,0), # C
                c(0,0,1,0,2,0,0), # D
                c(0,0,0,2,0,2,1), # E
                c(0,0,0,1,2,0,1), # F
                c(0,0,0,0,1,1,0)  # G
                )
rownames(adj_mat) = c("A", "B", "C", "D", "E", "F", "G")
colnames(adj_mat) = c("A", "B", "C", "D", "E", "F", "G")
```

```
ig = graph.adjacency(as.matrix(adj_mat), mode="undirected", weighted=TRUE)
G=vector(mode="list", length=7)
G[seq_len(length(G))] = 0
names(G) = c("A", "B", "C", "D", "E", "F", "G")
S = c("A", "C")
coords = layout.fruchterman.reingold(ig)
graph.netWalkSnapShot(adj_mat,G,output_dir=getwd(),p1=1.0,
                        "A",S,coords,1,TRUE)
```

`Miller2015` *Miller et al. (2015)*

Description

Untargeted metabolomic analysis for the clinical screening of inborn errors of metabolism. Global metabolic profiling obtained by untargeted mass spectrometry-based metabolomic platform for the detection of novel and known inborn errors of metabolism. This untargeted approach collected z-score values for ~1200 unique compounds (including ~500 named human analytes) from human plasma. Data set contains 186 individual plasma samples (118 confirmed inborn errors of metabolism). The outcome describes excellent sensitivity and specificity for the detection of a wide rage of metabolic disorders and identified novel biomarkers for some diseases.

Usage

```
data(Miller2015)
```

Format

Miller2015 - The data frame with 1203 metabolite features as rows, and 186 untargeted metabolomics patient samples as columns, alongside 16 metabolite annotations. The first row also provides the biochemical diagnosis confirmed for each patient sample.

Source

Dataset

References

Miller et al. (2015) J Inherit Metab Dis. 2015; 38: 1029–1039 (PubMed)

Examples

```
data(Miller2015)
```

`mle.getEncodingLength` *Minimum encoding length*

Description

This function calculates the mininmum encoding length associated with a subset of variables given a background knowledge graph.

Usage

```
mle.getEncodingLength(bs, pvals, ptID, G)
```

Arguments

bs	- A list of bitstrings associated with a given patient's perturbed variables.
pvals	- The matrix that gives the perturbation strength significance for all variables (columns) for each patient (rows)
ptID	- The row name in data.pvals corresponding to the patient you specifically want encoding information for.
G	- A list of probabilities with list names being the node names of the background graph.

Value

df - a data.frame object, for every bitstring provided in bs input parameter, a row is returned with the following data: the patientID; the bitstring evaluated where T denotes a hit and 0 denotes a miss; the subsetSize, or the number of hits in the bitstring; the individual p-values associated with the variable's perturbations, delimited by '/'; the combined p-value of all variables in the set using Fisher's method; Shannon's entropy, IS.null; the minimum encoding length IS.alt; and IS.null-IS.alt, the d.score.

Examples

```
# Identify the most significantly connected subset for a given patients'
# perturbations, given the network G
data("Miller2015")
data_mx = Miller2015[-c(1,grep("x - ",rownames(Miller2015))),
                        grep("IEM", colnames(Miller2015))]
data_mx = apply(data_mx, c(1,2), as.numeric)
data_pval=t(apply(data_mx,c(1,2),
                    function(i)2*pnorm(abs(i),lower.tail=FALSE)))
# Choose patient #1's (i.e., IEM_1000's) top 15 perturbed metabolites
ptID = colnames(data_mx)[1]
S=rownames(data_mx)[order(abs(data_mx[,which(colnames(data_mx)==ptID)]),
                          decreasing=TRUE)[seq_len(15)]]
# Build a dummy metabolite network for all metabolites in data_mx
adj_mat=matrix(0, nrow=nrow(data_mx), ncol=nrow(data_mx))
rows=sample(seq_len(ncol(adj_mat)), 0.1*ncol(adj_mat))
cols=sample(seq_len(ncol(adj_mat)), 0.1*ncol(adj_mat))
for (i in rows){for (j in cols){adj_mat[i,j]=rnorm(1,mean=0,sd=1)}}
colnames(adj_mat) = rownames(data_mx)
rownames(adj_mat) = rownames(data_mx)
```

```
G = vector("numeric", length=ncol(adj_mat))
names(G)=colnames(adj_mat)
ranks = list()
for (n in seq_len(length(S))) {
    print(sprintf("%d / %d", n, length(S)))
    ind = which(names(G)==S[n])
    ranks[[n]]=singleNode.getNodeRanksN(ind,G,p1=0.9,thresholdDiff=0.01,
                                        adj_mat,S,log2(length(G)),FALSE)
}
names(ranks) = S
ptBSbyK = mle.getPtBSbyK(S, ranks)
res = mle.getEncodingLength(ptBSbyK, data_pval, ptID, G)
# Rows with d.scores > 4.32 are of interest. Anything less indicates
# no to weak signal.
res = res[order(res[,"d.score"], decreasing=TRUE),]
print(res)
```

`mle.getMinPtDistance` *Get minimum patient distances*

Description

Given a series of patient distance matrices, return the minimum distance between all pairwise patient comparisons made.

Usage

```
mle.getMinPtDistance(allSimMatrices)
```

Arguments

`allSimMatrices`	- A list of all similarity matrices, across all k for a given graph, or across many graphs.

Value

minPtSim - Pairwise patient distances representing the minimum patient distance observed across several distance matrices.

Examples

```
# Get patient distances for the first 2 patients in the Miller 2015 dataset.
data("Miller2015")
data_mx = Miller2015[-c(1,grep("x - ",rownames(Miller2015))),
                       grep("IEM", colnames(Miller2015))]
data_mx = apply(data_mx[,c(1,2)], c(1,2), as.numeric)
# Build a network, G
adj_mat = matrix(0, nrow=nrow(data_mx), ncol=nrow(data_mx))
rows = sample(seq_len(ncol(adj_mat)), 0.1*ncol(adj_mat))
cols = sample(seq_len(ncol(adj_mat)), 0.1*ncol(adj_mat))
for(i in rows){for(j in cols){adj_mat[i,j]=rnorm(1,0,1)}}
colnames(adj_mat) = rownames(data_mx)
rownames(adj_mat) = rownames(data_mx)
G = vector("numeric", length=ncol(adj_mat))
```

```
names(G)=colnames(adj_mat)
# Look at the top 15 metabolites for each patient.
kmx=15
topMets_allpts = c()
for(pt in seq_len(ncol(data_mx))){
    topMets_allpts=c(topMets_allpts,
                     rownames(data_mx)[order(abs(data_mx[,pt]),
                                             decreasing=TRUE)[seq_len(kmx)]])
}
topMets_allpts = unique(topMets_allpts)
# Pre-compute node ranks for all metabolites in topMets_allpts for
# faster distance calculations.
ranks = list()
for(n in seq_len(length(topMets_allpts))){
    ind=which(names(G)==topMets_allpts[n])
    ranks[[n]]=singleNode.getNodeRanksN(ind,G,0.9,0.01,adj_mat,
                                        topMets_allpts,log2(length(G)))
}
names(ranks) = topMets_allpts
# Also pre-compute patient bitstrings for faster distance calculations.
ptBSbyK = list()
for (pt in seq_len(ncol(data_mx))) {
    S=rownames(data_mx)[order(abs(data_mx[,pt]),
                              decreasing=TRUE)[seq_len(kmx)]]
    ptBSbyK[[pt]]=mle.getPtBSbyK(S, ranks)
}
# Build your results ("res") list object to store patient distances at
# different size k's.
res = list()
t = list(ncd=matrix(NA, nrow=ncol(data_mx), ncol=ncol(data_mx)))
rownames(t$ncd) = colnames(data_mx)
colnames(t$ncd) = colnames(data_mx)
for (i in seq_len(kmx)) { res[[i]] = t }
for (pt in seq_len(ncol(data_mx))) {
    print(pt)
    ptID = colnames(data_mx)[pt]
    for (pt2 in pt:ncol(data_mx)) {
        ptID2 = colnames(data_mx)[pt2]
        tmp = mle.getPtDist(ptBSbyK[[pt]], ptID, ptBSbyK[[pt2]], ptID2, data_mx,
                            ranks, p1=0.9, thresholdDiff=0.01, adj_mat)
        for (k in seq_len(kmx)) {
            res[[k]]$ncd[ptID, ptID2] = tmp$NCD[k]
            res[[k]]$ncd[ptID2, ptID] = tmp$NCD[k]
        }
    }
}
res_ncd = lapply(res, function(i) i$ncd)
minPtDist = mle.getMinPtDistance(res_ncd)
```

`mle.getPtBSbyK` *Generate patient-specific bitstrings*

Description

This function calculates the bitstrings (1 is a hit; 0 is a miss) associated with a network walker which tries to find all nodes in a given subset, S, in a given network, G.

Usage

```
mle.getPtBSbyK(S, ranks, num.misses = NULL)
```

Arguments

S	- A character vector of node names describing the node subset to be encoded.
ranks	- The list of node ranks calculated over all possible nodes, starting with each node in subset of interest.
num.misses	- The number of misses tolerated by the network walker before path truncation occurs.

Value

pt.byK - a list of bitstrings, with the names of the list elements the node names of the encoded nodes

Examples

```
# Get patient bitstrings for the first 2 patients in the Miller 2015 dataset.
data("Miller2015")
data_mx=Miller2015[-c(1,grep("x - ", rownames(Miller2015))),
                   grep("IEM", colnames(Miller2015))]
data_mx=apply(data_mx[,c(1,2)], c(1,2), as.numeric)
# Build an adjacency matrix for network G
adj_mat=matrix(0, nrow=nrow(data_mx), ncol=nrow(data_mx))
rows=sample(seq_len(ncol(adj_mat)), 0.1*ncol(adj_mat))
cols=sample(seq_len(ncol(adj_mat)), 0.1*ncol(adj_mat))
for(i in rows){for (j in cols){adj_mat[i, j]=rnorm(1,0,1)}}
colnames(adj_mat)=rownames(data_mx)
rownames(adj_mat)=rownames(data_mx)
G=vector("numeric", length=ncol(adj_mat))
names(G)=colnames(adj_mat)
# Look at the top 15 metabolites for each patient.
kmx=15
topMets_allpts=c()
for (pt in seq_len(ncol(data_mx))) {
    topMets_allpts=c(topMets_allpts,
                     rownames(data_mx)[order(abs(data_mx[,pt]),
                                             decreasing=TRUE)[seq_len(kmx)]])}
topMets_allpts=unique(topMets_allpts)
# Use a single-node or multi-node network walker.
# Here we use a single-node network walker.
ranks=list()
for (n in seq_len(length(topMets_allpts))) {
    ind=which(names(G)==topMets_allpts[n])
    ranks[[n]]=singleNode.getNodeRanksN(ind,G,0.9,0.01,adj_mat,
                                        topMets_allpts,log2(length(G)))
}
names(ranks)=topMets_allpts
ptBSbyK=list()
for (pt in seq_len(ncol(data_mx))) {
    S=rownames(data_mx)[order(abs(data_mx[,pt]),
                              decreasing=TRUE)[seq_len(kmx)]]
    ptBSbyK[[pt]]=mle.getPtBSbyK(S, ranks)
}
```

`mle.getPtDist` *CTDncd: A network-based distance metric.*

Description

This function calculates the universal distance between patients, using a mutual information metric, where self-information comes from the minimum encoding length of each patient's encoded modular perturbations in the network.

Usage

```
mle.getPtDist(
  p1.optBS,
  ptID,
  p2.optBS,
  ptID2,
  data_mx,
  ranks,
  p1,
  thresholdDiff,
  adj_mat
)
```

Arguments

`p1.optBS`	- The optimal bitstring associated with patient 1.
`ptID`	- The identifier associated with patient 1's sample.
`p2.optBS`	- The optimal bitstring associated with patient 2.
`ptID2`	- The identifier associated with patient 2's sample.
`data_mx`	- The matrix that gives the perturbation strength (z-scores) for all variables (columns) for each patient (rows).
`ranks`	- The list of node ranks, starting with each node in patient 1&2's subsets of interest.
`p1`	- The probability that is preferentially distributed between network nodes by the probability diffusion algorithm based solely on network connectivity. The remaining probability (i.e., "p0") is uniformally distributed between network nodes, regardless of connectivity.
`thresholdDiff`	- When the probability diffusion algorithm exchanges this amount (thresholdDiff) or less between nodes, the algorithm returns up the call stack.
`adj_mat`	- The adjacency matrix that encodes the edge weights for the network, G.

Value

patientDistances - a distance matrix, where row and columns are patient identifiers.

Examples

```
# Get patient distances for the first 2 patients in the Miller 2015 dataset.
data("Miller2015")
data_mx = Miller2015[-c(1,grep("x - ", rownames(Miller2015))),
                        grep("IEM",colnames(Miller2015))]
data_mx = apply(data_mx[,c(1,2)], c(1,2), as.numeric)
# Build a network, G
adj_mat = matrix(0, nrow=nrow(data_mx), ncol=nrow(data_mx))
rows = sample(seq_len(ncol(adj_mat)), 0.1*ncol(adj_mat))
cols = sample(seq_len(ncol(adj_mat)), 0.1*ncol(adj_mat))
for (i in rows) {for (j in cols) {adj_mat[i,j]=rnorm(1,mean=0,sd=1)}}
colnames(adj_mat) = rownames(data_mx)
rownames(adj_mat) = rownames(data_mx)
G = vector("numeric", length=ncol(adj_mat))
names(G)=colnames(adj_mat)
# Look at the top 15 metabolites for each patient.
kmx=15
topMets_allpts = c()
for (pt in seq_len(ncol(data_mx))) {
    topMets_allpts=c(topMets_allpts,
                    rownames(data_mx)[order(abs(data_mx[,pt]),
                                            decreasing=TRUE)[seq_len(kmx)]])}
topMets_allpts = unique(topMets_allpts)
# Pre-compute node ranks for all metabolites in topMets_allpts
# for faster distance calculations.
ranks = list()
for (n in seq_len(length(topMets_allpts))) {
    ind = which(names(G)==topMets_allpts[n])
    ranks[[n]]=singleNode.getNodeRanksN(ind,G,0.9,0.01,adj_mat,
                                        topMets_allpts,log2(length(G)))
}
names(ranks) = topMets_allpts
# Also pre-compute patient bitstrings for faster distance calculations.
ptBSbyK = list()
for (pt in seq_len(ncol(data_mx))) {
    S=rownames(data_mx)[order(abs(data_mx[,pt]),
                              decreasing=TRUE)[seq_len(kmx)]]
    ptBSbyK[[pt]] = mle.getPtBSbyK(S, ranks)
}
# Build your results ("res") list object to store patient distances at
# different size k's.
res = list()
t = list(ncd=matrix(NA, nrow=ncol(data_mx), ncol=ncol(data_mx)))
rownames(t$ncd) = colnames(data_mx)
colnames(t$ncd) = colnames(data_mx)
for (i in seq_len(kmx)) { res[[i]] = t }
for (pt in seq_len(ncol(data_mx))) {
    print(pt)
    ptID = colnames(data_mx)[pt]
    for (pt2 in pt:ncol(data_mx)) {
        ptID2 = colnames(data_mx)[pt2]
        tmp=mle.getPtDist(ptBSbyK[[pt]],ptID,ptBSbyK[[pt2]],ptID2,
                          data_mx,ranks,p1=0.9,thresholdDiff=0.01,adj_mat)
        for (k in seq_len(kmx)) {
            res[[k]]$ncd[ptID, ptID2] = tmp$NCD[k]
            res[[k]]$ncd[ptID2, ptID] = tmp$NCD[k]
```

```
        }
    }
}
```

`multiNode.getNodeRanks` *Generate multi-node node rankings ("adaptive" walk)*

Description

This function calculates the node rankings starting from a given node in a subset of nodes in a given network, G.

Usage

```
multiNode.getNodeRanks(
  S,
  G,
  p1,
  thresholdDiff,
  adj_mat,
  num.misses = NULL,
  verbose = FALSE,
  out_dir = "",
  useLabels = FALSE,
  coords = NULL
)
```

Arguments

`S`	- A character vector of the node names for the subset of nodes you want to encode.
`G`	- A list of probabilities with list names being the node names of the network.
`p1`	- The probability that is preferentially distributed between network nodes by the probability diffusion algorithm based solely on network connectivity. The remaining probability, 1-p1, is uniformally distributed between network nodes, regardless of connectivity.
`thresholdDiff`	- When the probability diffusion algorithm exchanges this amount or less between nodes, the algorithm returns up the call stack.
`adj_mat`	- The adjacency matrix that encodes the edge weights for the network, G.
`num.misses`	- The number of "misses" the network walker will tolerate before switching to fixed length codes for remaining nodes to be found.
`verbose`	- If TRUE, print statements will execute as progress is made. Default is FALSE.
`out_dir`	- If specified, a image sequence will generate in the output directory specified.
`useLabels`	- If TRUE, node names will display next to their respective nodes in the network. If FALSE, node names will not display. Only relevant if out_dir is specified.
`coords`	- The x and y coordinates for each node in the network, to remain static between images.

Value

ranks - A list of character vectors of node names in the order they were drawn by the probability diffusion algorithm, from each starting node in S.

Examples

```
# Read in any network via its adjacency matrix
adj_mat=matrix(1, nrow=100, ncol=100)
for(i in seq_len(100)){for (j in seq_len(100)){adj_mat[i,j]=rnorm(1,0,1)}}
colnames(adj_mat)=sprintf("Metabolite%d", seq_len(100))
rownames(adj_mat)=colnames(adj_mat)
G=vector(mode="list", length=ncol(adj_mat))
names(G)=colnames(adj_mat)
S=names(G)[seq_len(3)]
ranks=multiNode.getNodeRanks(S, G, p1=0.9, thresholdDiff=0.01, adj_mat)
```

`singleNode.getNodeRanksN` *Generate single-node node rankings ("fixed" walk)*

Description

This function calculates the node rankings starting from a given perturbed variable in a subset of variables in the network.

Usage

```
singleNode.getNodeRanksN(
  n,
  G,
  p1,
  thresholdDiff,
  adj_mat,
  S = NULL,
  num.misses = NULL,
  verbose = FALSE,
  out_dir = "",
  useLabels = FALSE,
  coords = NULL
)
```

Arguments

n	- The index (out of a vector of node names) of the node ranking you want to calculate.
G	- A list of probabilities with list names being the node names of the network.
p1	- The probability that is preferentially distributed between network nodes by the probability diffusion algorithm based solely on network connectivity. The remaining probability (i.e., "p0") is uniformally distributed between network nodes, regardless of connectivity.

thresholdDiff	- When the probability diffusion algorithm exchanges this amount or less between nodes, the algorithm returns up the call stack.
adj_mat	- The adjacency matrix that encodes the edge weights for the network, G.
S	- A character vector of node names in the subset you want the network walker to find.
num.misses	- The number of "misses" the network walker will tolerate before switching to fixed length codes for remaining nodes to be found.
verbose	- If TRUE, print statements will execute as progress is made. Default is FALSE.
out_dir	- If specified, a image sequence will generate in the output directory specified.
useLabels	- If TRUE, node names will display next to their respective nodes in the network. If FALSE, node names will not display. Only relevant if out_dir is specified.
coords	- The x and y coordinates for each node in the network, to remain static between images.

Value

curr_ns - A character vector of node names in the order they were drawn by the probability diffusion algorithm.

Examples

```
data("Miller2015")
data_mx=Miller2015[-c(1,grep("x - ",rownames(Miller2015))),
                   grep("IEM", colnames(Miller2015))]
data_mx=apply(data_mx, c(1,2), as.numeric)
# Build an adjacency matrix for network G
adj_mat=matrix(0, nrow=nrow(data_mx),ncol=nrow(data_mx))
rows=sample(seq_len(ncol(adj_mat)),0.1*ncol(adj_mat))
cols=sample(seq_len(ncol(adj_mat)),0.1*ncol(adj_mat))
for(i in rows){for (j in cols){adj_mat[i,j]=rnorm(1,0,1)}}
colnames(adj_mat) = rownames(data_mx)
rownames(adj_mat) = rownames(data_mx)
G=vector("numeric", length=ncol(adj_mat))
names(G)=colnames(adj_mat)
# Get node rankings for the first metabolite in network G.
ranks=singleNode.getNodeRanksN(1,G,p1=0.9,thresholdDiff=0.01,adj_mat)
# Make a movie of the network walker
S=names(G)[sample(seq_len(length(G)), 3, replace=FALSE)]
ig=graph.adjacency(adj_mat,mode="undirected",weighted=TRUE,
                   add.colnames="name")
coords=layout.fruchterman.reingold(ig)
ranks = singleNode.getNodeRanksN(which(names(G)==S[1]),G,p1=0.9,
                                 thresholdDiff=0.01,adj_mat,S,
                                 log2(length(G)),FALSE,getwd())
```

---

stat.entropyFunction *Entropy of a bit-string*

Description

The entropy of a bitstring (ex: 1010111000) is calculated.

Usage

```
stat.entropyFunction(bitString)
```

Arguments

bitString	- A vector of 0's and 1's.

Value

e - a floating point percentage, between 0 and 1.

Examples

```
stat.entropyFunction(c(1,0,0,0,1,0,0,0,0,0,0,0,0))       # Output: 0.6193822
stat.entropyFunction(c(1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0)) # Output: 1
stat.entropyFunction(c(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1)) # Output: 0
```

---

stat.fishersMethod *Fisher's Combined P-value*

Description

Fisher's combined p-value, used to combine the results of individual statistical tests into an overall hypothesis.

Usage

```
stat.fishersMethod(x)
```

Arguments

x	- A vector of p-values (floating point numbers).

Value

a floating point number, a combined p-value using Fisher's method.

Examples

```
stat.fishersMethod(c(0.2,0.1,0.3))   # Output: 0.1152162
```

stat.getDirSim *DirSim: The Jaccard distance with directionality incorporated.*

Description

DirSim: The Jaccard distance with directionality incorporated.

Usage

```
stat.getDirSim(ptID, ptID2, kmx, data_mx)
```

Arguments

ptID	- The identifier associated with patient 1's sample.
ptID2	- The identifier associated with patient 2's sample.
kmx	- The number of top perturbations to consider in distance calculation.
data_mx	- The matrix that gives the perturbation strength (z-scores) for all variables (columns) for each patient (rows).

Value

dirSim - a distance matrix, where row and columns are patient identifiers.

Examples

```
# Get patient distances for the first 2 patients in the Miller 2015 dataset.
data("Miller2015")
data_mx = Miller2015[-c(1,grep("x - ", rownames(Miller2015))),
                        grep("IEM",colnames(Miller2015))]
data_mx = apply(data_mx[,c(1,2)], c(1,2), as.numeric)
# Build a network, G
adj_mat = matrix(0, nrow=nrow(data_mx), ncol=nrow(data_mx))
rows = sample(seq_len(ncol(adj_mat)), 0.1*ncol(adj_mat))
cols = sample(seq_len(ncol(adj_mat)), 0.1*ncol(adj_mat))
for (i in rows) {for (j in cols) {adj_mat[i,j]=rnorm(1,mean=0,sd=1)}}
colnames(adj_mat) = rownames(data_mx)
rownames(adj_mat) = rownames(data_mx)
G = vector("numeric", length=ncol(adj_mat))
names(G)=colnames(adj_mat)
# Look at the top 15 metabolites for each patient.
kmx=15
topMets_allpts = c()
for (pt in seq_len(ncol(data_mx))) {
    topMets_allpts=c(topMets_allpts,
                     rownames(data_mx)[order(abs(data_mx[,pt]),
                                             decreasing=TRUE)[seq_len(kmx)]])}
topMets_allpts = unique(topMets_allpts)
# Pre-compute node ranks for all metabolites in topMets_allpts
# for faster distance calculations.
ranks = list()
for (n in seq_len(length(topMets_allpts))) {
    ind = which(names(G)==topMets_allpts[n])
    ranks[[n]]=singleNode.getNodeRanksN(ind,G,0.9,0.01,adj_mat,
```

```
                                        topMets_allpts,log2(length(G)))
}
names(ranks) = topMets_allpts
# Also pre-compute patient bitstrings for faster distance calculations.
ptBSbyK = list()
for (pt in seq_len(ncol(data_mx))) {
    S=rownames(data_mx)[order(abs(data_mx[,pt]),
                              decreasing=TRUE)[seq_len(kmx)]]
    ptBSbyK[[pt]] = mle.getPtBSbyK(S, ranks)
}
# Build your results ("res") list object to store patient distances at
# different size k's.
res = list()
t = list(dir=matrix(NA, nrow=ncol(data_mx), ncol=ncol(data_mx)))
rownames(t$dir) = colnames(data_mx)
colnames(t$dir) = colnames(data_mx)
for (i in seq_len(kmx)) { res[[i]] = t }
for (pt in seq_len(ncol(data_mx))) {
    print(pt)
    ptID=colnames(data_mx)[pt]
    for (pt2 in pt:ncol(data_mx)) {
        ptID2=colnames(data_mx)[pt2]
        tmp=stat.getDirSim(ptID,ptID2,kmx,data_mx)
        for (k in seq_len(kmx)) {
            res[[k]]$dir[ptID, ptID2]=tmp[k]
            res[[k]]$dir[ptID2, ptID]=tmp[k]
        }
    }
}
```

`Thistlethwaite2020` *Thistlethwaite et al. (2020)*

Description

Clinical Diagnosis of Metabolic Disorders using Untargeted Metabolomic Profiling and Disease-specific Networks Learned from Patient Data. A meta-analysis of previous untargeted metabolomics studies describing 16 unique inborn errors of metabolism.

Usage

```
data(Thistlethwaite2020)
```

Format

Thistlethwaite2020 - A data frame with 1364 metabolite features as rows and 545 untargeted metabolomics patient samples as columns, alongside 8 metabolite annotations.

Source

Dataset

References

L.R. Thistlethwaite, et al. 2020. In review.

Examples

```
data(Thistlethwaite2020)
```

`Wangler2017` *Wangler et al. (2017)*

Description

A metabolomic map of Zellweger spectrum disorders reveals novel disease biomarkers. Global metabolic profiling obtained by untargeted mass spectrometry-based metabolomic platform for the detection of novel and known inborn errors of metabolism. This untargeted approach collected z-score values for >650 unique compounds from human plasma. Data set contains 19 individual plasma samples with confirmed biallelic pathogenic variants in the PEX1 gene. These samples revealed elevations in pipecolic acid and long-chain lysophosphatidylcholines, as well as an unanticipated reduction in multiple sphingomyelin species.

Usage

```
data(Wangler2017)
```

Format

Wangler2017 - The data matrix (metabolite features are rows, patient observations are columns) for 19 untargeted metabolomics patient samples, alongside metabolite annotations.

Source

Dataset

References

M.F. Wangler, et al. Genetics in Medicine, 2018, 00 (PubMed)

Examples

```
data(Wangler2017)
```

Index