Package 'CTD'

January 26, 2019

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

Version 0.0.0.9000

Date 2017-05-25

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; do the subsets show significant similarity?

Depends R (>= 3.3.0),
igraph,
plotly,
gplots,
RColorBrewer

License MIT License

Encoding UTF-8

LazyData true

RoxygenNote 6.1.0.9000

R topics documented:

`data.surrogateProfiles` *Surrogate profiles*

Description

Fill in a data matrix with low n, high p with surrogate profiles.

Usage

```
data.surrogateProfiles(data, sd = 1)
```

Arguments

`data`	- Data matrix with observations as rows, features as columns.
`sd`	- The level of variability (standard deviation) around each feature's mean you want to add in surrogate profiles.

Value

data_mx - Data matrix with added surrogate profiles.

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

Description

Recursively diffuse probability from a starting node based on the connectivity of the background knowledge graph, representing the likelihood that a variable will be most influenced by a perturbation in the starting node.

Usage

```
graph.diffuseP1(p1, startNode, G, visitedNodes, graphNumber = 1,
  verbose = FALSE)
```

Arguments

`p1`	- The probability being dispersed from the starting node, startNode.
`startNode`	- The first variable drawn in the adaptive permutation node sequence, from which p1 gets dispersed.
`G`	- A list of probabilities, with names of the list being the node names in the background knowledge graph.
`visitedNodes`	- The history of previous draws in the permutation sequence.
`graphNumber`	- If testing against multiple background knowledge graphs, this is the index associated with the adjacency matrix that codes for G. Default value is 1.
`verbose`	- If debugging or tracking a diffusion event, verbose=TRUE will activate print statements. Default is FALSE.

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 background knowledge graph.

Examples

```
# Look at main_CTD.r script for full analysis script: https://github.com/BRL-BCM/CTD.
# Read in any network via its adjacency matrix
tmp=matrix(1, nrow=100, ncol=100)
for (i in 1:100) {
  for (j in 1:100) {
    tmp[i, j]=rnorm(1, mean=0, sd=1)
  }
}
colnames(tmp)=sprintf("MolPheno%d", 1:100)
ig=graph.adjacency(tmp, mode="undirected", weighted=TRUE, add.colnames="name")
V(ig)$name=tolower(V(ig)$name)
adjacency_matrix=list(as.matrix(get.adjacency(ig, attr="weight")))  # Must have this declared as a GLOBAL var
# Set other tuning parameters
p0=0.1  # 10% of probability distributed uniformly
p1=0.9  # 90% of probability diffused based on edge weights in networks
thresholdDiff=0.01
G=vector(mode="list", length=length(V(ig)$name))
names(G)=V(ig)$name
G=lapply(G, function(i) i[[1]]=0)
startNode=names(G)[1]
visitedNodes=G[1]
probs_afterCurrDraw=graph.diffuseP1(p1, startNode, G, visitedNodes, 1, TRUE)
```

---

graph.diffuseP1Movie *Make a movie of the diffusion of probability, P1, from a starting node.*

Description

Recursively diffuse probability from a starting node based on the connectivity of the background knowledge graph, representing the likelihood that a variable will be most influenced by a perturbation in the starting node.

Usage

```
graph.diffuseP1Movie(p1, startNode, G, visitedNodes, ig,
  recursion_level = 1, output_dir = getwd())
```

Arguments

p1	- The probability being dispersed from the starting node, startNode.
startNode	- The first variable drawn in the adaptive permutation node sequence, from which p1 gets dispersed.
G	- A list of probabilities, with names of the list being the node names in the background knowledge graph.
visitedNodes	- A character vector of node names, storing the history of previous draws in the permutation sequence.

graphNumber - If testing against multiple background knowledge graphs, this is the index associated with the adjacency matrix that codes for G. Default value is 1.

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 background knowledge graph.

---

mle.getEncodingLength *Minimum encoding length (MLE)*

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

```
# Look at main_CTD.r script for full analysis script: https://github.com/BRL-BCM/CTD.
# Identify the most significant subset per patient, given the background graph
data_mx.pvals = t(apply(data_mx, c(1,2), function(i) 2*pnorm(abs(i), lower.tail = FALSE)))
for (pt in 1:ncol(data_mx)) {
    ptID = colnames(data_mx)[pt]
    res = mle.getEncodingLength(ptBSbyK[[ptID]], data_mx.pvals, ptID, G)
    res = res[order(res[,"d.score"], decreasing=TRUE),]
    print(res)
}
```

`mle.getPermMovie_memory` *Capture the movement of the adaptive walk of the diffusion probability method.*

Description

Make a movie of the adaptive walk the diffusion probability method makes in search of a given patient's perturbed variables.

Usage

```
mle.getPermMovie_memory(subset.nodes, ig, output_filepath, movie = TRUE,
  subset = FALSE)
```

Arguments

`subset.nodes`	- The subset of variables, S, in a background graph, G.
`ig`	- The igraph object associated with the background knowledge graph.
`movie`	- If you want to make a movie, set to TRUE. This will produce a set of still images that you can stream together to make a movie. Default is TRUE. Alternatively (movie=FALSE), you could use this function to get the node labels returned for each permutation starting with a perturbed variable.

Examples

```
# Look at main_CTD.r script for full analysis script: https://github.com/BRL-BCM/CTD.
# Read in any network via its adjacency matrix
tmp = matrix(1, nrow=100, ncol=100)
for (i in 1:100) {
  for (j in 1:100) {
    tmp[i, j] = rnorm(1, mean=0, sd=1)
  }
}
colnames(tmp) = sprintf("MolPheno%d", 1:100)
ig = graph.adjacency(tmp, mode="undirected", weighted=TRUE, add.colnames="name")
V(ig)$name = tolower(V(ig)$name)
# Set other tuning parameters
p0=0.1  # 10% of probability distributed uniformly
p1=0.9  # 90% of probability diffused based on edge weights in networks
thresholdDiff=0.01
subset.nodes = names(G)[sample(1:length(G), 3)]
mle.getPermMovie_memory(subset.nodes, ig, output_filepath = getwd(), movie=TRUE)
```

`mle.getPermMovie_memoryless` *Capture the movement of the adaptive walk of the diffusion probability method.*

Description

Make a movie of the adaptive walk the diffusion probability method makes in search of a given patient's perturbed variables.

Usage

```
mle.getPermMovie_memoryless(subset.nodes, ig, output_filepath,
  movie = TRUE, subset = FALSE)
```

Arguments

`subset.nodes`	- The subset of variables, S, in a background graph, G.
`ig`	- The igraph object associated with the background knowledge graph.
`movie`	- If you want to make a movie, set to TRUE. This will produce a set of still images that you can stream together to make a movie. Default is TRUE. Alternatively (movie=FALSE), you could use this function to get the node labels returned for each permutation starting with a perturbed variable.

Examples

```
# Look at main_CTD.r script for full analysis script: https://github.com/BRL-BCM/CTD.
# Read in any network via its adjacency matrix
tmp = matrix(1, nrow=100, ncol=100)
for (i in 1:100) {
  for (j in 1:100) {
    tmp[i, j] = rnorm(1, mean=0, sd=1)
  }
}
colnames(tmp) = sprintf("MolPheno%d", 1:100)
ig = graph.adjacency(tmp, mode="undirected", weighted=TRUE, add.colnames="name")
V(ig)$name = tolower(V(ig)$name)
# Set other tuning parameters
p0=0.1  # 10% of probability distributed uniformly
p1=0.9  # 90% of probability diffused based on edge weights in networks
thresholdDiff=0.01
subset.nodes = names(G)[sample(1:length(G), 3)]
mle.getPermMovie_memoryless(subset.nodes, ig, output_filepath = getwd(), movie=TRUE)
mle.getPermMovie_memory(subset.nodes, ig, output_filepath = getwd(), movie=TRUE)
```

`mle.getPermN_memory` *Generate the "adaptive walk" node permutations, starting from a given perturbed variable*

Description

This function calculates the node permutation starting from a given perturbed variable in a subset of variables in the background knowledge graph.

Usage

```
mle.getPermN_memory(S, G)
```

Arguments

`S`	- A character vector of the node names for the subset of nodes you want to encode.

Value

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

Examples

```
# Look at main_CTD.r script for full analysis script: https://github.com/BRL-BCM/CTD.
# Get node permutations for graph
perms = list()
for (n in 1:length(G)) {
  print(sprintf("Generating node permutation starting with node %s", names(G)[n]))
  perms[[n]] = mle.getPermN_memory(n, G)
}
names(perms) = names(G)
```

`mle.getPermN_memoryless` *Generate the "adaptive walk" node permutations, starting from a given perturbed variable*

Description

This function calculates the node permutation starting from a given perturbed variable in a subset of variables in the background knowledge graph.

Usage

```
mle.getPermN_memoryless(n, G, S = NULL, misses.thresh = NULL)
```

Arguments

n	- The index (out of a vector of node names) of the permutation you want to calculate.
G	- A list of probabilities with list names being the node names of the background graph.

Value

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

Examples

```
# Look at main_CTD.r script for full analysis script: https://github.com/BRL-BCM/CTD.
# Get node permutations for graph
perms = list()
for (n in 1:length(G)) {
  print(sprintf("Generating node permutation starting with node %s", names(G)[n]))
  perms[[n]] = mle.getPermN(n, G)
}
names(perms) = names(G)
```

---

mle.getPtBSbyK *Generate patient-specific bitstrings from adaptive network walk.*

Description

This function calculates the bitstrings (1 is a hit; 0 is a miss) associated with the adaptive network walk made by the diffusion algorithm trying to find the variables in the encoded subset, given the background knowledge graph.

Usage

```
mle.getPtBSbyK(S, perms)
```

Arguments

perms	- The list of permutations calculated over all possible nodes, starting with each node in subset of interest.
data_mx	- The matrix that gives the perturbation strength (z-score) for all variables (rows) for each patient (columns).
ptID	- The identifier associated with the patient being processed.
kmx	- The maximum size of variable sets for which you want to calculate probabilities.

Value

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

Examples

```
# Look at main_CTD.r script for full analysis script: https://github.com/BRL-BCM/CTD.
# Get bitstrings associated with each patient's top kmx variable subsets
kmx = 15
ptBSbyK = list()
for (pt in 1:ncol(data_mx)) {
  S = data_mx[order(abs(data_mx[,pt]), decreasing=TRUE),pt][1:kmx]
  ptBSbyK[[ptID]] = mle.getPtBSbyK(S, perms)
}
```

`mle.getPtSim` *Patient similarity using mutual information MLE metric of patients' most modular, perturbed subsets.*

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 background knowledge graph.

Usage

```
mle.getPtSim(p1.optBS, ptID, p2.optBS, ptID2, data_mx, perms)
```

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.
`data_mx`	- The matrix that gives the perturbation strength (z-scores) for all variables (columns) for each patient (rows).
`ptID`	- The identifier associated with patient 2's sample.

Value

patientSim - a similarity matrix, where row and columns are patient identifiers.

Examples

```
# Look at main_CTD.r script for full analysis script: https://github.com/BRL-BCM/CTD.
# Get patient distances
data_mx.pvals = apply(data_mx, c(1,2), function(i) 2*pnorm(abs(i), lower.tail = FALSE))
res = list()
t = list(ncd=matrix(NA, nrow=ncol(data_mx), ncol=ncol(data_mx)),
         dir=matrix(NA, nrow=ncol(data_mx), ncol=ncol(data_mx)),
         jac=matrix(NA, nrow=ncol(data_mx), ncol=ncol(data_mx)))
rownames(t$ncd) = colnames(data_mx)
colnames(t$ncd) = colnames(data_mx)
rownames(t$dir) = colnames(data_mx)
colnames(t$dir) = colnames(data_mx)
rownames(t$jac) = colnames(data_mx)
colnames(t$jac) = colnames(data_mx)
```

```
for (i in 1:(kmx-1)) {
  res[[i]] = t
}
for (pt in 1:ncol(data_mx)) {
  print(pt)
  ptID = colnames(data_mx)[pt]
  for (pt2 in pt:ncol(data_mx)) {
    ptID2 = colnames(data_mx)[pt2]
    for (k in 1:(kmx-1)) {
     tmp = mle.getPtSim(ptBSbyK[[ptID]][k], ptID, ptBSbyK[[ptID2]][k], ptID2, data_mx, perms)
      res[[k]]$ncd[ptID, ptID2] = tmp$NCD
      res[[k]]$dir[ptID, ptID2] = tmp$dirSim
      res[[k]]$ncd[ptID2, ptID] = tmp$NCD
      res[[k]]$dir[ptID2, ptID] = tmp$dirSim

     p1.sig.nodes = rownames(data_mx)[order(abs(data_mx[,ptID]), decreasing = TRUE)][1:k]
     p2.sig.nodes = rownames(data_mx)[order(abs(data_mx[,ptID2]), decreasing = TRUE)][1:k]
      p1.dirs = data_mx[p1.sig.nodes, ptID]
      p1.dirs[which(!(p1.dirs>0))] = 0
      p1.dirs[which(p1.dirs>0)] = 1
      p2.dirs = data_mx[p2.sig.nodes, ptID2]
      p2.dirs[which(!(p2.dirs>0))] = 0
      p2.dirs[which(p2.dirs>0)] = 1
      p1.sig.nodes = sprintf("%s%d", p1.sig.nodes, p1.dirs)
      p2.sig.nodes = sprintf("%s%d", p2.sig.nodes, p2.dirs)
     res[[k]]$jac[ptID, ptID2] = 1 - (length(intersect(p1.sig.nodes, p2.sig.nodes))/length(union(p1.sig.node
     res[[k]]$jac[ptID2, ptID] = 1 - (length(intersect(p1.sig.nodes, p2.sig.nodes))/length(union(p1.sig.node
    }
  }
}
```

`mle.kraftMcMillian_memoryless` *Apply the Kraft-McMillian Inequality using a specific encoding algorithm.*

Description

A power analysis of the encoding algorithm using to encode subsets of S in G.

Usage

```
mle.kraftMcMillian_memoryless(G, k)
```

Arguments

G	- A character vector of all node names in the background knowledge graph.
k	- The size of the node name subsets of G.

Value

IA - a list of bitlengths associated with all outcomes in the N choose K outcome space, with the names of the list elements the node names of the encoded nodes

plot.hmSim *Generate heatmap plot of patient similarity matrix.*

Description

This function plots a heatmap of a patient similarity matrix.

Usage

```
## S3 method for class 'hmSim'
plot(simMat, path, diagnoses = NULL)
```

Arguments

simMat	- The patient similarity matrix.
path	- The filepath to a directory in which you want to store the .png file.
diagnoses	- A character vector of diagnostic labels associated with the rownames of simMat.

Examples

```
# Look at main_CTD.r script for full analysis script: https://github.com/BRL-BCM/CTD.
plot.hmSim(patientSim, path=getwd(), diagnoses)
```

plot.mdsSim *View patient clusters using multi-dimensional scaling.*

Description

This function plots the provided patient similarity matrix in a lower dimensional space using multi-dimensional scaling, which is well suited for similarity metrics.

Usage

```
## S3 method for class 'mdsSim'
plot(patientSim, diagnoses, k, diag)
```

Arguments

diagnoses	- A character vector of diagnostic labels associated with the rownames of simMat.
k	- The number of dimension you want to plot your data using multi-dimensional scaling.
diag	- The diagnosis associated with positive controls in your data.
simMat	- The patient similarity matrix.

Value

p - a plotly scatter plot colored by provided diagnostic labels.

Examples

```
# Look at main_CTD.r script for full analysis script: https://github.com/BRL-BCM/CTD.
# if you have diagnostic labels associated with the colnames(data_mx), send them using diagnoses parameter
p = plot.mdsSim(patientSim, diagnoses, k=2, diag="diseased")
p
p = plot.mdsSim(patientSim, diagnoses, k=3, diag="diseased")
p
```

---

stats.entropyFunction *Entropy of a bit-string*

Description

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

Usage

```
stats.entropyFunction(bitString)
```

Arguments

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

Value

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

Examples

```
stats.entropyFunction(c(1,0,0,0,1,0,0,0,0,0,0,0,0))
> 0.6193822
stats.entropyFunction(c(1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0))
> 1
stats.entropyFunction(c(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1))
> 0
```

---

stats.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

```
stats.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

```
stats.fishersMethod(c(0.2,0.1,0.3))
> 0.1152162
```

---

stats.iteratedLog2 *Iterated Logarithm (Base 2)*

Description

This function calculates the number of times the logarithm function (base 2) must be iteratively applied before the result is less than or equal to 1.

Usage

```
stats.iteratedLog2(num)
```

Arguments

num	- An integer.

Examples

```
iteratedLogarithm(4)
2
```

Index