Package ‘CTD’

December 14, 2018

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 probabilistic estimation of multivariate feature sets, against a partial correlation network of features.

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:

`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`	- The igraph object associated with 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.

Examples

```
# Read in any network via its adjacency matrix
tmp = as.matrix(read.table("adjacency_matrix.txt", sep="\t", header=TRUE))
colnames(tmp) = rownames(tmp)
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 v

p0=0.1  # 10% of probability distributed uniformly
p1=0.9  # 90% of probability diffused based on edge weights in networks
G = vector(mode="list", length=length(V(ig)$name))
names(G) = names(V(ig)$name)
startNode = names(G)[1]
visitedNodes = NULL
probs_afterCurrDraw = graph.diffuseP1(p1, startNode, G, visitedNodes, 1)
```

`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)
```

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.

Examples

```
# Read in any network via its adjacency matrix
tmp = as.matrix(read.table("adjacency_matrix.txt", sep="\t", header=TRUE))
colnames(tmp) = rownames(tmp)
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 v
# 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
G = vector(mode="list", length=length(V(ig)$name))
names(G) = names(V(ig)$name)
# 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[[names(G)[n]]] = mle.getPermN(n, G)
}
# Decide what the largest subset size you will consider will be
kmx = 20
# Load your patient data (p features as rows x n observations as columns)
# data_mx = read.table("/your/own/data.txt", sep="\t", header=TRUE)
data(testData)
data_mx = testData
# Get bitstrings associated with each patient's top kmx variable subsets
ptBSbyK = list()
for (pt in 1:ncol(data_mx)) {
    ptID = colnames(data_mx)[pt]
    ptBSbyK[[ptID]] = mle.getPtBSbyK(data_mx, ptID, perms, kmx)
}
# Identify the most significant subset per patient, given the background graph
data_mx.pvals = 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)
    res = res[which.max(res[,"d.score"]),]
    print(res)
}
```

`mle.getPatientSimilarity` *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.getPatientSimilarity(p1.optBS, ptID, p2.optBS, ptID2, data_mx)
```

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.

Examples

```
# Read in any network via its adjacency matrix
tmp = as.matrix(read.table("adjacency_matrix.txt", sep="\t", header=TRUE))
colnames(tmp) = rownames(tmp)
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 v
# 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
G = vector(mode="list", length=length(V(ig)$name))
names(G) = names(V(ig)$name)
# 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[[names(G)[n]]] = mle.getPermN(n, G)
}
# Decide what the largest subset size you will consider will be
kmx = 20
```

```
# Load your patient data (p features as rows x n observations as columns)
# data_mx = read.table("/your/own/data.txt", sep="\t", header=TRUE)
data(testData)
data_mx = testData
# Get bitstrings associated with each patient's top kmx variable subsets
ptBSbyK = list()
for (pt in 1:ncol(data_mx)) {
    ptID = colnames(data_mx)[pt]
    ptBSbyK[[ptID]] = mle.getPtBSbyK(data_mx, ptID, perms, kmx)
}
# Get patient distances
patientSim = matrix(NA, nrow=ncol(data_mx), ncol=ncol(data_mx))
rownames(patientSim) = colnames(data_mx)
colnames(patientSim) = colnames(data_mx)
for (pt in 1:ncol(data_mx)) {
    ptID = colnames(data_mx)[pt]
    for (pt2 in 1:ncol(data_mx)) {
        ptID2 = colnames(data_mx)[pt2]
      patientSim[ptID, ptID2] = mle.getPatientSimilarity(ptBSbyK[ptID], ptID, ptBSbyK[ptID2], data_mx)
    }
}
```

`mle.getPermMovie` *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(subset.nodes, ig, output_filepath, movie = TRUE)
```

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

```
# Read in any network via its adjacency matrix
tmp = as.matrix(read.table("adjacency_matrix.txt", sep="\t", header=TRUE))
colnames(tmp) = rownames(tmp)
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 v
p0=0.1  # 10% of probability distributed uniformly
```

```
p1=0.9  # 90% of probability diffused based on edge weights in networks
G = vector(mode="list", length=length(V(ig)$name))
names(G) = names(V(ig)$name)
subset.nodes = names(G)[sample(1:length(G), 3)]
mle.getPermMovie(subset.nodes, ig, output_filepath = getwd(), movie=TRUE)
```

`mle.getPermN` *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(n, G)
```

Arguments

`n`	- The index (out of a vector of metabolite names) of the permutation you want to calculate.

Examples

```
# Read in any network via its adjacency matrix
tmp = as.matrix(read.table("adjacency_matrix.txt", sep="\t", header=TRUE))
colnames(tmp) = rownames(tmp)
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 v
p0=0.1  # 10% of probability distributed uniformly
p1=0.9  # 90% of probability diffused based on edge weights in networks
G = vector(mode="list", length=length(V(ig)$name))
names(G) = names(V(ig)$name)
perms = list()
for (n in 1:length(G)) {
    print(sprintf("Generating node permutation starting with node %s", names(G)[n]))
    perms[[names(G)[n]]] = mle.getPermN(n, 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(data_mx, ptID, perms, kmx)
```

Arguments

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

Examples

```
# Read in any network via its adjacency matrix
tmp = as.matrix(read.table("adjacency_matrix.txt", sep="\t", header=TRUE))
colnames(tmp) = rownames(tmp)
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 v
# 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
G = vector(mode="list", length=length(V(ig)$name))
names(G) = names(V(ig)$name)
# 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[[names(G)[n]]] = mle.getPermN(n, G)
}
# Decide what the largest subset size you will consider will be
kmx = 20
# Load your patient data (p features as rows x n observations as columns)
# data_mx = read.table("/your/own/data.txt", sep="\t", header=TRUE)
data(testData)
data_mx = testData
# Get bitstrings associated with each patient's top kmx variable subsets
ptBSbyK = list()
for (pt in 1:ncol(data_mx)) {
    ptID = colnames(data_mx)[pt]
    ptBSbyK[[ptID]] = mle.getPtBSbyK(data_mx, ptID, perms, kmx)
}
```

`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

```
# Read in any network via its adjacency matrix
tmp = as.matrix(read.table("adjacency_matrix.txt", sep="\t", header=TRUE))
colnames(tmp) = rownames(tmp)
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 v
# 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
G = vector(mode="list", length=length(V(ig)$name))
names(G) = names(V(ig)$name)
# 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[[names(G)[n]]] = mle.getPermN(n, G)
}
# Decide what the largest subset size you will consider will be
kmx = 20
# Load your patient data (p features as rows x n observations as columns)
# data_mx = read.table("/your/own/data.txt", sep="\t", header=TRUE)
data(testData)
data_mx = testData
# Get bitstrings associated with each patient's top kmx variable subsets
ptBSbyK = list()
for (pt in 1:ncol(data_mx)) {
    ptID = colnames(data_mx)[pt]
    ptBSbyK[[ptID]] = mle.getPtBSbyK(data_mx, ptID, perms, kmx)
}
# Get patient distances
patientSim = matrix(NA, nrow=ncol(data_mx), ncol=ncol(data_mx))
rownames(patientSim) = colnames(data_mx)
colnames(patientSim) = colnames(data_mx)
for (pt in 1:ncol(data_mx)) {
    ptID = colnames(data_mx)[pt]
    for (pt2 in 1:ncol(data_mx)) {
        ptID2 = colnames(data_mx)[pt2]
      patientSim[ptID, ptID2] = mle.getPatientSimilarity(ptBSbyK[ptID], ptID, ptBSbyK[ptID2], data_mx)
    }
}
# if you have diagnostic labels associated with the colnames(data_mx), send them using diagnoses parameter
diagnoses = colnames(data_mx)
```

```
diagnoses[1:50] = "diseased"
diagnoses[51:100] = "neg_control"
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(simMat, diagnoses, k, diag)
```

Arguments

`simMat`	- The patient similarity matrix.
`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.

Examples

```
# Read in any network via its adjacency matrix
tmp = as.matrix(read.table("adjacency_matrix.txt", sep="\t", header=TRUE))
colnames(tmp) = rownames(tmp)
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 v
# 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
G = vector(mode="list", length=length(V(ig)$name))
names(G) = names(V(ig)$name)
# 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[[names(G)[n]]] = mle.getPermN(n, G)
}
# Decide what the largest subset size you will consider will be
kmx = 20
# Load your patient data (p features as rows x n observations as columns)
# data_mx = read.table("/your/own/data.txt", sep="\t", header=TRUE)
data(testData)
data_mx = testData
# Get bitstrings associated with each patient's top kmx variable subsets
ptBSbyK = list()
```

```
for (pt in 1:ncol(data_mx)) {
    ptID = colnames(data_mx)[pt]
    ptBSbyK[[ptID]] = mle.getPtBSbyK(data_mx, ptID, perms, kmx)
}
# Get patient distances
data_mx.pvals = apply(data_mx, c(1,2), function(i) 2*pnorm(abs(i), lower.tail = FALSE))
patientSim = matrix(NA, nrow=ncol(data_mx), ncol=ncol(data_mx))
rownames(patientSim) = colnames(data_mx)
colnames(patientSim) = colnames(data_mx)
for (pt in 1:ncol(data_mx)) {
    ptID = colnames(data_mx)[pt]
    for (pt2 in 1:ncol(data_mx)) {
        ptID2 = colnames(data_mx)[pt2]
      patientSim[ptID, ptID2] = mle.getPatientSimilarity(ptBSbyK[ptID], ptID, ptBSbyK[ptID2], data_mx, data_
    }
}
# if you have diagnostic labels associated with the colnames(data_mx), send them using diagnoses parameter
diagnoses = colnames(data_mx)
diagnoses[1:50] = "diseased"
diagnoses[51:100] = "neg_control"
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.

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).

Examples

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

Index