Package: NetRep (via r-universe)

October 12, 2024

Type Package

Title Permutation Testing Network Module Preservation Across Datasets

Version 1.2.7

BugReports `https://github.com/sritchie73/NetRep/issues`

Description Functions for assessing the replication/preservation of a network module's topology across datasets through permutation testing; Ritchie et al. (2015) <doi:10.1016/j.cels.2016.06.012>.

License GPL-2

Depends R (>= 3.6), methods

Imports foreach, Rcpp (>= 0.11), statmod, RhpcBLASctl, abind, RColorBrewer, utils, stats, graphics, grDevices

Suggests bigmemory, testthat, knitr, rmarkdown

LinkingTo Rcpp, BH, RcppArmadillo (>= 0.4)

RoxygenNote 7.2.3

VignetteBuilder knitr

Encoding UTF-8

NeedsCompilation yes

Author Scott Ritchie [aut, cre] (0000-0002-8454-9548)

Maintainer Scott Ritchie <`sritchie73@gmail.com`>

Repository CRAN

Date/Publication 2023-08-19 15:12:32 UTC

Contents

disk.matrix *The 'disk.matrix' class*

Description

A 'disk.matrix' contains a file path to a matrix stored on disk, along with meta data for how to read that file. This allows **NetRep** to load datasets into RAM only when required, i.e. one at a time. This significantly reduces the memory usage of R when analysing large datasets. 'disk.matrix' objects may be supplied instead of 'matrix' objects in the input list arguments 'network', 'data', and 'correlation', which are common to most of **NetRep**'s functions.

Usage

```
attach.disk.matrix(file, serialized = TRUE, ...)

serialize.table(file, ...)

is.disk.matrix(x)

as.disk.matrix(x, file, serialize = TRUE)

## S4 method for signature 'disk.matrix'
as.disk.matrix(x, file, serialize = TRUE)

## S4 method for signature 'matrix'
as.disk.matrix(x, file, serialize = TRUE)

## S4 method for signature 'ANY'
as.disk.matrix(x, file, serialize = TRUE)

## S4 method for signature 'disk.matrix'
as.matrix(x)

## S4 method for signature 'disk.matrix'
show(object)
```

Arguments

`file` for `attach.disk.matrix` the file name of a matrix on disk. For `as.disk.matrix` the file name to save the matrix to. For `serialize.table` the file name of a matrix in table format on disk.

`serialized` determines how the matrix will be loaded from disk into R by `as.matrix`. If `TRUE`, the `readRDS` function will be used. If `FALSE`, the `read.table` function will be used.

`...` arguments to be used by `read.table` when reading in matrix data from a file in table format.

`x` for `as.matrix` a `disk.matrix` object to load into R. For `as.disk.matrix` an object to convert to a `disk.matrix`. For `is.disk.matrix` an object to check if its a `disk.matrix`.

`serialize` determines how the matrix is saved to disk by `as.disk.matrix`. If `TRUE` it will be stored as a serialized R object using `saveRDS`. If `FALSE` it will be stored as a tab-separated file using `write.table`.

`object` a `'disk.matrix'` object.

Details

Matrices may either be stored as regular table files that can be read by `read.table`, or as serialized R objects that can be read by `readRDS`. Serialized objects are much faster to load, but cannot be read by other programs.

The `attach.disk.matrix` function creates a `disk.matrix` object from a file path. The `as.matrix` function will load the data from disk into the R session as a regular `matrix` object.

The `as.disk.matrix` function converts a matrix into a `disk.matrix` by saving its contents to the specified `file`. The `serialize` argument determines whether the data is stored as a serialized R object or as a tab-separated file (i.e. `sep="\t"`). We recommend storing the matrix as a serialized R object unless disk space is a concern. More control over the storage format can be obtained by using `saveRDS` or `write.table` directly.

The `serialize.matrix` function converts a file in table format to a serialized R object with the same file name, but with the ".rds" extension.

Value

A `disk.matrix` object (`attach.disk.matrix`, `as.disk.matrix`), a `matrix` (`as.matrix`), the file path to a serialized matrix (`serialize.table`), or a `TRUE` or `FALSE` indicating whether an object is a `disk.matrix` (`is.disk.matrix`).

Slots

`file` the name of the file where the matrix is saved.

`read.func` either `"read.table"` or `"readRDS"`.

`func.args` a list of arguments to be supplied to the `'read.func'`.

Warning

`attach.disk.matrix` does not check whether the specified file can be read into R. `as.matrix` will fail and throw an error if this is the case.

example-data *Example data*

Description

Example gene coexpression networks inferred from two independent datasets to demonstrate the usage of package functions.

Usage

```
data("NetRep")
```

Format

- **"discovery_network"** a `matrix` with 150 columns and 150 rows containing the network edge weights encoding the interaction strength between each pair of genes in the *discovery* dataset.
- **"discovery_data"** a `matrix` with 150 columns (genes) and 30 rows (samples) whose entries correspond to the expression level of each gene in each sample in the *discovery* dataset.
- **"discovery_correlation"** a `matrix` with 150 columns and 150 rows containing the correlation-coefficients between each pair of genes calculated from the "discovery_data" `matrix`.
- **\"module_labels"** a named `vector` with 150 entries containing the module assignment for each gene as identified in the *discovery* dataset.
- **"test_network"** a `matrix` with 150 columns and 150 rows containing the network edge weights encoding the interaction strength between each pair of genes in the *test* dataset.
- **"test_data"** a `matrix` with 150 columns (genes) and 30 rows (samples) whose entries correspond to the expression level of each gene in each sample in the *test* dataset.
- **"test_correlation"** a `matrix` with 150 columns and 150 rows containing the correlation-coefficients between each pair of genes calculated from the "test_data" `matrix`.

An object of class `matrix` (inherits from `array`) with 150 rows and 150 columns.

An object of class `matrix` (inherits from `array`) with 30 rows and 150 columns.

An object of class `matrix` (inherits from `array`) with 150 rows and 150 columns.

An object of class `numeric` of length 150.

An object of class `matrix` (inherits from `array`) with 150 rows and 150 columns.

An object of class `matrix` (inherits from `array`) with 30 rows and 150 columns.

An object of class `matrix` (inherits from `array`) with 150 rows and 150 columns.

Details

The preservation of network modules in a second dataset is quantified by measuring the preservation of topological properties between the *discovery* and *test* datasets. These properties are calculated not only from the interaction networks inferred in each dataset, but also from the data used to infer those networks (e.g. gene expression data) as well as the correlation structure between variables/nodes. Thus, all functions in the `NetRep` package have the following arguments:

`network`: a list of interaction networks, one for each dataset.

`data`: a list of data matrices used to infer those networks, one for each dataset.

`correlation`: a list of matrices containing the pairwise correlation coefficients between variables/nodes in each dataset.

`moduleAssignments`: a list of vectors, one for each *discovery* dataset, containing the module assignments for each node in that dataset.

`modules`: a list of vectors, one vector for each *discovery* dataset, containing the names of the modules from that dataset to analyse.

`discovery`: a vector indicating the names or indices of the previous arguments' lists to use as the *discovery* dataset(s) for the analyses.

`test`: a list of vectors, one vector for each *discovery* dataset, containing the names or indices of the `network`, `data`, and `correlation` argument lists to use as the *test* dataset(s) for the analysis of each *discovery* dataset.

This data is used to provide concrete examples of the usage of these arguments in each package function.

Simulation details

The *discovery* gene expression dataset (`"discovery_data"`) containing 30 samples and 150 genes was simulated to contain four distinct modules of sizes 20, 25, 30, and 35 genes. Data for each module were simulated as:

$$G^{(w)}_{simulated} = E^{(w)}r_i + \sqrt{1 - r_i^2}\epsilon$$

Where $E^{(w)}$ is the simulated module's *summary vector*, r is the simulated module's *node contributions* for each gene, and ϵ is the error term drawn from a standard normal distribution. $E^{(w)}$ and r were simulated by bootstrapping (sampling with replacement) samples and genes from the corresponding vectors in modules 63, 51, 57, and 50 discovered in the liver tissue gene expression data from a publicly available mouse dataset (see reference *(1)* for details on the dataset and network discovery). The remaining 40 genes that were not part of any module were simulated by randomly selecting 40 liver genes and bootstrapping 30 samples and adding the noise term, ϵ. A vector of module assignments was created ("module_labels") in which each gene was labelled with a number 1-4 corresponding to the module they were simulated to be coexpressed with, or a label of 0 for the for the 40 "background" genes not participating in any module. The correlation structure ("discovery_correlation") was calculated as the Pearson's correlation coefficient between genes (cor(discovery_data)). Edge weights in the interaction network ("discovery_network") were calculated as the absolute value of the correlation coefficient exponentiated to the power 5 (abs(discovery_correlation)^5).

An independent test dataset ("test_data") containing the same 150 genes as the *discovery* dataset but 30 different samples was simulated as above. Modules 1 and 4 (containing 20 and 35 genes respectively) were simulated to be preserved using the same equation above, where the *summary vector* $E^{(w)}$ was bootstrapped from the same liver modules (modules 63 and 50) as in the *discovery* and with identical *node contributions* r as in the *discovery* dataset. Genes in modules 2 and 3 were simulated as "background" genes, *i.e.* not preserved as described above. The correlation structure between genes in the *test* dataset ("test_correlation") and the interaction network ("test_network") were calculated the same way as in the *discovery* dataset.

The random seed used for the simulations was 37.

References

1. Ritchie, S.C., *et al.*, *A scalable permutation approach reveals replication and preservation patterns of network modules in large datasets*. Cell Systems. **3**, 71-82 (2016).

See Also

modulePreservation, plotModule, and networkProperties.

---

modulePreservation *Replication and preservation of network modules across datasets*

Description

Quantify the preservation of network modules (sub-graphs) in an independent dataset through permutation testing on module topology. Seven network statistics (see details) are calculated for each module and then tested by comparing to distributions generated from their calculation on random subsets in the test dataset.

Usage

```
modulePreservation(
  network,
  data,
  correlation,
  moduleAssignments,
  modules = NULL,
  backgroundLabel = "0",
  discovery = 1,
  test = 2,
  selfPreservation = FALSE,
  nThreads = NULL,
  nPerm = NULL,
  null = "overlap",
  alternative = "greater",
  simplify = TRUE,
  verbose = TRUE
)
```

Arguments

network — a list of interaction networks, one for each dataset. Each entry of the list should be a $n * n$ matrix or where each element contains the edge weight between nodes i and j in the inferred network for that dataset.

data — a list of matrices, one for each dataset. Each entry of the list should be the data used to infer the interaction network for that dataset. The columns should correspond to variables in the data (nodes in the network) and rows to samples in that dataset.

`correlation` a list of matrices, one for each dataset. Each entry of the list should be a $n * n$ matrix where each element contains the correlation coefficient between nodes i and j in the `data` used to infer the interaction network for that dataset.

`moduleAssignments`
a list of vectors, one for each *discovery* dataset, containing the module assignments for each node in that dataset.

`modules` a list of vectors, one for each `discovery` dataset, of modules to perform the analysis on. If unspecified, all modules in each `discovery` dataset will be analysed, with the exception of those specified in `backgroundLabel` argument.

`backgroundLabel`
a single label given to nodes that do not belong to any module in the `moduleAssignments` argument. Defaults to "0". Set to `NULL` if you do not want to skip the network background module.

`discovery` a vector of names or indices denoting the *discovery* dataset(s) in the `data`, `correlation`, `network`, `moduleAssignments`, `modules`, and `test` lists.

`test` a list of vectors, one for each `discovery` dataset, of names or indices denoting the *test* dataset(s) in the `data`, `correlation`, and `network` lists.

`selfPreservation`
logical; if `FALSE` (default) then module preservation analysis will not be performed within a dataset (*i.e.* where the `discovery` and `test` datasets are the same).

`nThreads` number of threads to parallelise the calculation of network properties over. Automatically determined as the number of cores - 1 if not specified.

`nPerm` number of permutations to use. If not specified, the number of permutations will be automatically determined (see details). When set to 0 the permutation procedure will be skipped and the observed module preservation will be returned without p-values.

`null` variables to include when generating the null distributions. Must be either "overlap" or "all" (see details).

`alternative` The type of module preservation test to perform. Must be one of "greater" (default), "less" or "two.sided" (see details).

`simplify` logical; if `TRUE`, simplify the structure of the output list if possible (see Return Value).

`verbose` logical; should progress be reported? Default is `TRUE`.

Details

Input data structures:: The preservation of network modules in a second dataset is quantified by measuring the preservation of topological properties between the *discovery* and *test* datasets. These properties are calculated not only from the interaction networks inferred in each dataset, but also from the data used to infer those networks (e.g. gene expression data) as well as the correlation structure between variables/nodes. Thus, all functions in the `NetRep` package have the following arguments:

- `network`: a list of interaction networks, one for each dataset.
- `data`: a list of data matrices used to infer those networks, one for each dataset.

- `correlation`: a list of matrices containing the pairwise correlation coefficients between variables/nodes in each dataset.
- `moduleAssignments`: a list of vectors, one for each *discovery* dataset, containing the module assignments for each node in that dataset.
- `modules`: a list of vectors, one for each *discovery* dataset, containing the names of the modules from that dataset to analyse.
- `discovery`: a vector indicating the names or indices of the previous arguments' lists to use as the *discovery* dataset(s) for the analyses.
- `test`: a list of vectors, one vector for each *discovery* dataset, containing the names or indices of the `network`, `data`, and `correlation` argument lists to use as the *test* dataset(s) for the analysis of each *discovery* dataset.

The formatting of these arguments is not strict: each function will attempt to make sense of the user input. For example, if there is only one `discovery` dataset, then input to the `moduleAssigments` and `test` arguments may be vectors, rather than lists.

Analysing large datasets:: Matrices in the `network`, `data`, and `correlation` lists can be supplied as `disk.matrix` objects. This class allows matrix data to be kept on disk and loaded as required by **NetRep**. This dramatically decreases memory usage: the matrices for only one dataset will be kept in RAM at any point in time.

Additional memory usage of the permutation procedure is directly proportional to the sum of module sizes squared multiplied by the number of threads. Very large modules may result in significant additional memory usage per core due to extraction of the correlation coefficient submatrix at each permutation.

Module Preservation Statistics:: Module preservation is assessed through seven module preservation statistics, each of which captures a different aspect of a module's topology; *i.e.* the structure of the relationships between its nodes *(1,2)*. Below is a description of each statistic, what they seek to measure, and where their interpretation may be inappropriate.

The *module coherence* (`'coherence'`), *average node contribution* (`'avg.contrib'`), and *concordance of node contribution* (`'cor.contrib'`) are all calculated from the data used to infer the network (provided in the `'data'` argument). They are calculated from the module's *summary profile*. This is the eigenvector of the 1st principal component across all observations for every node composing the module. For gene coexpression modules this can be interpreted as a "summary expression profile". It is typically referred to as the "module eigengene" in the weighted gene coexpression network analysis literature *(4)*.

The *module coherence* (`'coherence'`) quantifies the proportion of module variance explained by the module's "summary profile". The higher this value, the more "coherent" the data is, *i.e.* the more similar the observations are nodes for each sample. With the default alternate hypothesis, a small permutation *P*-value indicates that the module is more coherent than expected by chance.

The *average node contribution* (`'avg.contrib'`) and *concordance of node contribution* (`'cor.contrib'`) are calculated from the *node contribution*, which quantifies how similar each node is to the modules's *summary profile*. It is calculated as the Pearson correlation coefficient between each node and the module summary profile. In the weighted gene coexpression network literature it is typically called the "module membership" *(2)*.

The *average node contribution* (`'avg.contrib'`) quantifies how similar nodes are to the module summary profile in the test dataset. Nodes detract from this score where the sign of their node contribution flips between the discovery and test datasets, *e.g.* in the case of differential gene expression across conditions. A high *average node contribution* with a small permutation *P*-value indicates that the module remains coherent in the test dataset, and that the nodes are acting together in a similar way.

The *concordance of node contribution* ('cor.contrib') measures whether the relative rank of nodes (in terms of their node contribution) is preserved across datasets. If a module is coherent enough that all nodes contribute strongly, then this statistic will not be meaningful as its value will be heavily influenced by tiny variations in node rank. This can be assessed through visualisation of the module topology (see plotContribution.) Similarly, a strong 'cor.contrib' is unlikely to be meaningful if the 'avg.contrib' is not significant.

The *concordance of correlation strucutre* ('cor.cor') and *density of correlation structure* ('avg.cor') are calculated from the user-provided correlation structure between nodes (provided in the 'correlation' argument). This is referred to as "coexpression" when calculated on gene expression data.

The 'avg.cor' measures how strongly nodes within a module are correlation on average in the test dataset. This average depends on the correlation coefficients in the discovery dataset: the score is penalised where correlation coefficients change in sign between datasets. A high 'avg.cor' with a small permutation *P*-value indicates that the module is (a) more strongly correlated than expected by chance for a module of the same size, and (b) more consistently correlated with respect to the discovery dataset than expected by chance.

The 'cor.cor' measures how similar the correlation coefficients are across the two datasets. A high 'cor.cor' with a small permutation *P*-value indicates that the correlation structure within a module is more similar across datasets than expected by chance. If all nodes within a module are very similarly correlated then this statistic will not be meaningful, as its value will be heavily influenced by tiny, non-meaningful, variations in correlation strength. This can be assessed through visualisation of the module topology (see plotCorrelation.) Similarly, a strong 'cor.cor' is unlikely to be meaningful if the 'avg.cor' is not significant.

The *average edge weight* ('avg.weight') and *concordance of weighted degree* ('cor.degree') are both calculated from the interaction network (provided as adjacency matrices to the 'network' argument).

The 'avg.weight' measures the average connection strength between nodes in the test dataset. In the weighted gene coexpression network literature this is typically called the "module density" *(2)*. A high 'avg.weight' with a small permutation *P*-value indicates that the module is more strongly connected in the test dataset than expected by chance.

The 'cor.degree' calculates whether the relative rank of each node's *weighted degree* is similar across datasets. The *weighted degree* is calculated as the sum of a node's edge weights to all other nodes in the module. In the weighted gene coexpression network literature this is typically called the "intramodular connectivity" *(2)*. This statistic will not be meaningful where all nodes are connected to each other with similar strength, as its value will be heavily influenced by tiny, non-meaningful, variations in weighted degree. This can be assessed through visualisation of the module topology (see plotDegree.)

Both the 'avg.weight' and 'cor.degree' assume edges are weighted, and that the network is densely connected. Note that for sparse networks, edges with zero weight are included when calculating both statistics. Only the magnitude of the weights, not their sign, contribute to the score. If the network is *unweighted*, *i.e.* edges indicate presence or absence of a relationship, then the 'avg.weight' will be the proportion of the number of edges to the total number of possible edges while the *weighted degree* simply becomes the *degree*. A high 'avg.weight' in this case measures how interconnected a module is in the test dataset. A high *degree* indicates that a node is connected to many other nodes. The interpretation of the 'cor.degree' remains unchanged between weighted and unweighted networks. If the network is directed the interpretation of the 'avg.weight' remains unchanged, while the *cor.degree* will measure the concordance of the node *in*-degree in the test network. To measure the *out*-degree instead, the adjacency matrices provided to the 'network' argument should be transposed.

Sparse data:: Caution should be used when running NetRep on sparse data (*i.e.* where there are many zero values in the data used to infer the network). For this data, the *average node contribution* ('avg.contrib'), *concordance of node contribution* ('cor.contrib'), and *module coherence* ('coherence') will all be systematically underestimated due to their reliance on the Pearson correlation coefficient to calculate the *node contribution*.

Care should also be taken to use appropriate methods for inferring the correlation structure when the data is sparse for the same reason.

Proportional data:: Caution should be used when running NetRep on proportional data (*i.e.* where observations across samples all sum to the same value, *e.g.* 1). For this data, the *average node contribution* ('avg.contrib'), *concordance of node contribution* ('cor.contrib'), and *module coherence* ('coherence') will all be systematically overestimated due to their reliance on the Pearson correlation coefficient to calculate the *node contribution*.

Care should also be taken to use appropriate methods for inferring the correlation structure from proportional data for the same reason.

Hypothesis testing:: Three alternative hypotheses are available. "greater", the default, tests whether each module preservation statistic is larger than expected by chance. "lesser" tests whether each module preservation statistic is smaller than expected by chance, which may be useful for identifying modules that are extremely different in the *test* dataset. "two.sided" can be used to test both alternate hypotheses.

To determine whether a module preservation statistic deviates from chance, a permutation procedure is employed. Each statistic is calculated between the module in the *discovery* dataset and nPerm random subsets of the same size in the *test* dataset in order to assess the distribution of each statistic under the null hypothesis.

Two models for the null hypothesis are available: "overlap", the default, only nodes that are present in both the *discovery* and *test* networks are used when generating null distributions. This is appropriate under an assumption that nodes that are present in the *test* dataset, but not present in the *discovery* dataset, are unobserved: that is, they may fall in the module(s) of interest in the *discovery* dataset if they were to be measured there. Alternatively, "all" will use all nodes in the *test* network when generating the null distributions.

The number of permutations required for any given significance threshold is approximately 1 / the desired significance for one sided tests, and double that for two-sided tests. This can be calculated with requiredPerms. When nPerm is not specified, the number of permutations is automatically calculated as the number required for a Bonferroni corrected significance threshold adjusting for the total number of tests for each statistic, i.e. the total number of modules to be analysed multiplied by the number of *test* datasets each module is tested in. Although assessing the replication of a small numberof modules calls for very few permutations, we recommend using no fewer than 1,000 as fewer permutations are unlikely to generate representative null distributions. **Note:** the assumption used by requiredPerms to determine the correct number of permtutations breaks down when assessing the preservation of modules in a very small dataset (e.g. gene sets in a dataset with less than 100 genes total). However, the reported p-values will still be accurate (see permutationTest) *(3)*.

Value

A nested list structure. At the top level, the list has one element per `'discovery'` dataset. Each of these elements is a list that has one element per `'test'` dataset analysed for that `'discovery'` dataset. Each of these elements is also a list, containing the following objects:

- `observed`: A matrix of the observed values for the module preservation statistics. Rows correspond to modules, and columns to the module preservation statistics.
- `nulls`: A three dimensional array containing the values of the module preservation statistics evaluated on random permutation of module assignment in the test network. Rows correspond to modules, columns to the module preservation statistics, and the third dimension to the permutations.
- `p.values`: A matrix of p-values for the `observed` module preservation statistics as evaluated through a permutation test using the corresponding values in `nulls`.
- `nVarsPresent`: A vector containing the number of variables that are present in the test dataset for each module.
- `propVarsPresent`: A vector containing the proportion of variables present in the test dataset for each module. Modules where this is less than 1 should be investigated further before making judgements about preservation to ensure that the missing variables are not the most connected ones.
- `contingency`: If `moduleAssignments` are present for both the *discovery* and *test* datasets, then a contingency table showing the overlap between modules across datasets is returned. Rows correspond to modules in the *discovery* dataset, columns to modules in the *test* dataset.

When `simplify = TRUE` then the simplest possible structure will be returned. E.g. if module preservation is tested in only one dataset, then the returned list will have only the above elements.

When `simplify = FALSE` then a nested list of datasets will always be returned, i.e. each element at the top level and second level correspond to a dataset, e.g. `results[["Dataset1"]][["Dataset2"]]` indicates an analysis where modules discovered in "Dataset1" are assessed for preservation in "Dataset2". Dataset comparisons which have not been assessed will contain `NULL`.

References

1. Ritchie, S.C., *et al.*, *A scalable permutation approach reveals replication and preservation patterns of network modules in large datasets*. Cell Systems. **3**, 71-82 (2016).
2. Langfelder, P., Luo, R., Oldham, M. C. & Horvath, S. *Is my network module preserved and reproducible?* PLoS Comput. Biol. **7**, e1001057 (2011).
3. Phipson, B. & Smyth, G. K. *Permutation P-values should never be zero: calculating exact P-values when permutations are randomly drawn.* Stat. Appl. Genet. Mol. Biol. **9**, Article39 (2010).
4. Langfelder, P. & Horvath, S. *WGCNA: an R package for weighted correlation network analysis.* BMC Bioinformatics **9**, 559 (2008).

See Also

Functions for: visualising network modules, calculating module topology, calculating permutation test P-values, and splitting computation over multiple machines.

Examples

```
# load in example data, correlation, and network matrices for a discovery and test dataset:
data("NetRep")

# Set up input lists for each input matrix type across datasets. The list
# elements can have any names, so long as they are consistent between the
# inputs.
network_list <- list(discovery=discovery_network, test=test_network)
data_list <- list(discovery=discovery_data, test=test_data)
correlation_list <- list(discovery=discovery_correlation, test=test_correlation)
labels_list <- list(discovery=module_labels)

# Assess module preservation: you should run at least 10,000 permutations
preservation <- modulePreservation(
 network=network_list, data=data_list, correlation=correlation_list,
 moduleAssignments=labels_list, nPerm=1000, discovery="discovery",
 test="test", nThreads=2
)
```

NetRep *Fast permutation procedure for testing network module replication*

Description

Functions for assessing the replication/preservation of a network module's topology across datasets through permutation testing. This is suitable for networks that can be meaningfully inferred from multiple datasets. These include gene coexpression networks, protein-protein interaction networks, and microbial interaction networks. Modules within these networks consist of groups of nodes that are particularly interesting: for example a group of tightly connected genes associated with a disease, groups of genes annotated with the same term in the Gene Ontology database, or groups of interacting microbial species, i.e. communities. Application of this method can answer questions such as; (1) do the relationships between genes in a module replicate in an independent cohort? (2) are these gene coexpression modules preserved across tissues or tissue specific? (3) are these modules conserved across species? (4) are microbial communities preserved across multiple spatial locations?

Details

The main function for this package is `modulePreservation`. Several functions for downstream are also provided: `networkProperties` for calculating the topological properties of a module, and `plotModule` for visualising a module.

Author(s)

Maintainer: Scott Ritchie `<sritchie73@gmail.com>` (0000-0002-8454-9548)

See Also

Useful links:

- Report bugs at https://github.com/sritchie73/NetRep/issues

---

networkProperties *Calculate the topological properties for a network module*

Description

Calculates the network properties used to assess module preservation for one or more modules in a user specified dataset.

Usage

```
networkProperties(
  network,
  data,
  correlation,
  moduleAssignments = NULL,
  modules = NULL,
  backgroundLabel = "0",
  discovery = NULL,
  test = NULL,
  simplify = TRUE,
  verbose = TRUE
)
```

Arguments

`network` a list of interaction networks, one for each dataset. Each entry of the list should be a $n*n$ matrix or where each element contains the edge weight between nodes i and j in the inferred network for that dataset.

`data` a list of matrices, one for each dataset. Each entry of the list should be the data used to infer the interaction `network` for that dataset. The columns should correspond to variables in the data (nodes in the network) and rows to samples in that dataset.

`correlation` a list of matrices, one for each dataset. Each entry of the list should be a $n * n$ matrix where each element contains the correlation coefficient between nodes i and j in the `data` used to infer the interaction network for that dataset.

`moduleAssignments`
a list of vectors, one for each *discovery* dataset, containing the module assignments for each node in that dataset.

`modules` a list of vectors, one for each `discovery` dataset, of modules to perform the analysis on. If unspecified, all modules in each `discovery` dataset will be analysed, with the exception of those specified in `backgroundLabel` argument.

backgroundLabel
: a single label given to nodes that do not belong to any module in the `moduleAssignments` argument. Defaults to "0". Set to `NULL` if you do not want to skip the network background module.

discovery
: a vector of names or indices denoting the *discovery* dataset(s) in the `data`, `correlation`, `network`, `moduleAssignments`, `modules`, and `test` lists.

test
: a list of vectors, one for each `discovery` dataset, of names or indices denoting the *test* dataset(s) in the `data`, `correlation`, and `network` lists.

simplify
: logical; if `TRUE`, simplify the structure of the output list if possible (see Return Value).

verbose
: logical; should progress be reported? Default is `TRUE`.

Details

Input data structures:: The preservation of network modules in a second dataset is quantified by measuring the preservation of topological properties between the *discovery* and *test* datasets. These properties are calculated not only from the interaction networks inferred in each dataset, but also from the data used to infer those networks (e.g. gene expression data) as well as the correlation structure between variables/nodes. Thus, all functions in the `NetRep` package have the following arguments:

- `network`: a list of interaction networks, one for each dataset.
- `data`: a list of data matrices used to infer those networks, one for each dataset.
- `correlation`: a list of matrices containing the pairwise correlation coefficients between variables/nodes in each dataset.
- `moduleAssignments`: a list of vectors, one for each *discovery* dataset, containing the module assignments for each node in that dataset.
- `modules`: a list of vectors, one for each *discovery* dataset, containing the names of the modules from that dataset to analyse.
- `discovery`: a vector indicating the names or indices of the previous arguments' lists to use as the *discovery* dataset(s) for the analyses.
- `test`: a list of vectors, one vector for each *discovery* dataset, containing the names or indices of the `network`, `data`, and `correlation` argument lists to use as the *test* dataset(s) for the analysis of each *discovery* dataset.

The formatting of these arguments is not strict: each function will attempt to make sense of the user input. For example, if there is only one `discovery` dataset, then input to the `moduleAssigments` and `test` arguments may be vectors, rather than lists. If the `networkProperties` are being calculate within the *discovery* or *test* datasets, then the `discovery` and `test` arguments do not need to be specified, and the input matrices for the `network`, `data`, and `correlation` arguments do not need to be wrapped in a list.

Analysing large datasets:: Matrices in the `network`, `data`, and `correlation` lists can be supplied as `disk.matrix` objects. This class allows matrix data to be kept on disk and loaded as required by **NetRep**. This dramatically decreases memory usage: the matrices for only one dataset will be kept in RAM at any point in time.

Value

A nested list structure. At the top level, the list has one element per 'discovery' dataset. Each of these elements is a list that has one element per 'test' dataset analysed for that 'discovery' dataset. Each of these elements is a list that has one element per 'modules' specified. Each of these is a list containing the following objects:

- 'degree': The weighted within-module degree: the sum of edge weights for each node in the module.
- 'avgWeight': The average edge weight within the module.

If the 'data' used to infer the 'test' network is provided then the following are also returned:

- 'summary': A vector summarising the module across each sample. This is calculated as the first eigenvector of the module from a principal component analysis.
- 'contribution': The *node contribution*: the similarity between each node and the *module summary profile* ('summary').
- 'coherence': The proportion of module variance explained by the 'summary' vector.

When simplify = TRUE then the simplest possible structure will be returned. E.g. if the network properties are requested for only one module in only one dataset, then the returned list will have only the above elements.

When simplify = FALSE then a nested list of datasets will always be returned, i.e. each element at the top level and second level correspond to a dataset, and each element at the third level will correspond to modules discovered in the dataset specified at the top level if module labels are provided in the corresponding moduleAssignments list element. E.g. results[["Dataset1"]][["Dataset2"]][["module1"]] will contain the properties of "module1" as calculated in "Dataset2", where "module1" was indentified in "Dataset1". Modules and datasets for which calculation of the network properties have not been requested will contain NULL.

See Also

Getting nodes ordered by degree., and Ordering samples by module summary

Examples

```
# load in example data, correlation, and network matrices for a discovery and test dataset:
data("NetRep")

# Set up input lists for each input matrix type across datasets. The list
# elements can have any names, so long as they are consistent between the
# inputs.
network_list <- list(discovery=discovery_network, test=test_network)
data_list <- list(discovery=discovery_data, test=test_data)
correlation_list <- list(discovery=discovery_correlation, test=test_correlation)
labels_list <- list(discovery=module_labels)

# Calculate the topological properties of all network modules in the discovery dataset
props <- networkProperties(
  network=network_list, data=data_list, correlation=correlation_list,
  moduleAssignments=labels_list
)

# Calculate the topological properties in the test dataset for the same modules
test_props <- networkProperties(
  network=network_list, data=data_list, correlation=correlation_list,
  moduleAssignments=labels_list, discovery="discovery", test="test"
)
```

plotModule *Plot the topology of a network module*

Description

Plot the correlation structure, network edges, scaled weighted degree, node contribtuion, module data, and module summary vectors of one or more network modules.

Individual components of the module plot can be plotted using `plotCorrelation`, `plotNetwork`, `plotDegree`, `plotContribution`, `plotData`, and `plotSummary`.

Usage

```
plotModule(
  network,
  data,
  correlation,
  moduleAssignments = NULL,
  modules = NULL,
  backgroundLabel = "0",
  discovery = NULL,
  test = NULL,
  verbose = TRUE,
  orderSamplesBy = NULL,
  orderNodesBy = NULL,
  orderModules = TRUE,
  plotNodeNames = TRUE,
  plotSampleNames = TRUE,
  plotModuleNames = NULL,
  main = "Module Topology",
  main.line = 1,
  drawBorders = FALSE,
  lwd = 1,
  naxt.line = -0.5,
  saxt.line = -0.5,
  maxt.line = NULL,
  xaxt.line = -0.5,
  xaxt.tck = -0.025,
  xlab.line = 2.5,
  yaxt.line = 0,
  yaxt.tck = -0.15,
  ylab.line = 2.5,
  laxt.line = 2.5,
  laxt.tck = 0.04,
  cex.axis = 0.8,
  legend.main.line = 1.5,
  cex.lab = 1.2,
  cex.main = 2,
  dataCols = NULL,
  dataRange = NULL,
  corCols = correlation.palette(),
  corRange = c(-1, 1),
  netCols = network.palette(),
  netRange = c(0, 1),
  degreeCol = "#feb24c",
  contribCols = c("#A50026", "#313695"),
  summaryCols = c("#1B7837", "#762A83"),
  naCol = "#bdbdbd",
  dryRun = FALSE
)
```

Arguments

`network` a list of interaction networks, one for each dataset. Each entry of the list should be a $n*n$ matrix or where each element contains the edge weight between nodes i and j in the inferred network for that dataset.

`data` a list of matrices, one for each dataset. Each entry of the list should be the data used to infer the interaction `network` for that dataset. The columns should correspond to variables in the data (nodes in the network) and rows to samples in that dataset.

`correlation` a list of matrices, one for each dataset. Each entry of the list should be a $n * n$ matrix where each element contains the correlation coefficient between nodes i and j in the `data` used to infer the interaction network for that dataset.

`moduleAssignments`

a list of vectors, one for each *discovery* dataset, containing the module assignments for each node in that dataset.

`modules` a list of vectors, one for each `discovery` dataset, of modules to perform the analysis on. If unspecified, all modules in each `discovery` dataset will be analysed, with the exception of those specified in `backgroundLabel` argument.

`backgroundLabel`

a single label given to nodes that do not belong to any module in the `moduleAssignments` argument. Defaults to "0". Set to `NULL` if you do not want to skip the network background module.

`discovery` a vector of names or indices denoting the *discovery* dataset(s) in the `data`, `correlation`, `network`, `moduleAssignments`, `modules`, and `test` lists.

`test` a list of vectors, one for each `discovery` dataset, of names or indices denoting the *test* dataset(s) in the `data`, `correlation`, and `network` lists.

`verbose` logical; should progress be reported? Default is `TRUE`.

`orderSamplesBy` `NULL` (default), `NA`, or a vector containing a single dataset name or index. Controls how samples are ordered on the plot (see details).

`orderNodesBy` `NULL` (default), `NA`, or a vector of dataset names or indices. Controls how nodes are ordered on the plot (see details).

`orderModules` logical; if `TRUE` modules ordered by clustering their summary vectors. If `FALSE` modules are returned in the order provided.

`plotNodeNames` logical; controls whether the node names are drawed on the bottom axis.

`plotSampleNames` logical; controls whether the sample names are drawed on the left axis.

`plotModuleNames` logical; controls whether module names are drawed. The default is for module names to be drawed when multiple `modules` are drawn.

`main` title for the plot.

`main.line` the number of lines into the top margin at which the plot title will be drawn.

`drawBorders` logical; if `TRUE`, borders are drawn around the *weighted degree*, *node conribution*, and *module summary* bar plots.

`lwd` line width for borders and axes.

`naxt.line` the number of lines into the bottom margin at which the node names will be drawn.

`saxt.line` the number of lines into the left margin at which the sample names will be drawn.

`maxt.line` the number of lines into the bottom margin at which the module names will be drawn.

`xaxt.line` the number of lines into the bottom margin at which the x-axis tick labels will be drawn on the module summary bar plot.

`xaxt.tck` the size of the x-axis ticks for the module summary bar plot.

`xlab.line` the number of lines into the bottom margin at which the x axis label on the *module summary* bar plot(s) will be drawn.

`yaxt.line` the number of lines into the left margin at which the y-axis tick labels will be drawn on the weighted degree and node contribution bar plots.

`yaxt.tck` the size of the y-axis ticks for the weighted degree and node contribution bar plots.

`ylab.line` the number of lines into the left margin at which the y axis labels on the *weighted degree* and *node contribution* bar plots will be drawn.

`laxt.line` the distance from the legend to draw the legend axis labels, as multiple of `laxt.tck`.

`laxt.tck` size of the ticks on each axis legend relative to the size of the correlation, edge weights, and data matrix heatmaps.

`cex.axis` relative size of the node and sample names.

`legend.main.line`
the distance from the legend to draw the legend title.

`cex.lab` relative size of the module names and legend titles.

`cex.main` relative size of the plot titles.

`dataCols` a character vector of colors to create a gradient from for the data heatmap (see details). Automatically determined if `NA` or `NULL`.

`dataRange` the range of values to map to the `dataCols` gradient (see details). Automatically determined if `NA` or `NULL`.

`corCols` a character vector of colors to create a gradient from for the correlation structure heatmap (see details).

`corRange` the range of values to map to the `corCols` gradient (see details).

`netCols` a character vector of colors to create a gradient from for the network edge weight heatmap (see details).

`netRange` the range of values to map to the `corCols` gradient (see details). Automatically determined if `NA` or `NULL`.

`degreeCol` color to use for the weighted degree bar plot.

`contribCols` color(s) to use for the node contribution bar plot (see details).

`summaryCols` color(s) to use for the node contribution bar plot (see details).

`naCol` color to use for missing nodes and samples on the data, correlation structure, and network edge weight heat maps.

`dryRun` logical; if `TRUE`, only the axes and labels will be drawed.

Details

Input data structures:: The preservation of network modules in a second dataset is quantified by measuring the preservation of topological properties between the *discovery* and *test* datasets. These properties are calculated not only from the interaction networks inferred in each dataset, but also from the data used to infer those networks (e.g. gene expression data) as well as the correlation structure between variables/nodes. Thus, all functions in the `NetRep` package have the following arguments:

- `network`: a list of interaction networks, one for each dataset.
- `data`: a list of data matrices used to infer those networks, one for each dataset.
- `correlation`: a list of matrices containing the pairwise correlation coefficients between variables/nodes in each dataset.
- `moduleAssignments`: a list of vectors, one for each *discovery* dataset, containing the module assignments for each node in that dataset.
- `modules`: a list of vectors, one for each *discovery* dataset, containing the names of the modules from that dataset to analyse.
- `discovery`: a vector indicating the names or indices of the previous arguments' lists to use as the *discovery* dataset(s) for the analyses.
- `test`: a list of vectors, one vector for each *discovery* dataset, containing the names or indices of the `network`, `data`, and `correlation` argument lists to use as the *test* dataset(s) for the analysis of each *discovery* dataset.

The formatting of these arguments is not strict: each function will attempt to make sense of the user input. For example, if there is only one `discovery` dataset, then input to the `moduleAssigments` and `test` arguments may be vectors, rather than lists. If the node and sample ordering is being calculated within the same dataset being visualised, then the `discovery` and `test` arguments do not need to be specified, and the input matrices for the `network`, `data`, and `correlation` arguments do not need to be wrapped in a list.

Analysing large datasets:: Matrices in the `network`, `data`, and `correlation` lists can be supplied as `disk.matrix` objects. This class allows matrix data to be kept on disk and loaded as required by **NetRep**. This dramatically decreases memory usage: the matrices for only one dataset will be kept in RAM at any point in time.

Node, sample, and module ordering:: By default, nodes are ordered in decreasing order of *weighted degree* in the `discovery` dataset (see `nodeOrder`). Missing nodes are colored in grey. This facilitates the visual comparison of modules across datasets, as the node ordering will be preserved.

Alternatively, a vector containing the names or indices of one or more datasets can be provided to the `orderNodesBy` argument.

If a single dataset is provided, then nodes will be ordered in decreasing order of *weighted degree* in that dataset. Only nodes that are present in this dataset will be drawn when ordering nodes by a dataset that is not the `discovery` dataset for the requested modules(s).

If multiple datasets are provided then the *weighted degree* will be averaged across these datasets (see nodeOrder for more details). This is useful for obtaining a robust ordering of nodes by relative importance, assuming the modules displayed are preserved in those datasets.

Ordering of nodes by *weighted degree* can be suppressed by setting `orderNodesBy` to `NA`, in which case nodes will be ordered as in the matrices provided in the `network`, `data`, and `correlation` arguments.

When multiple modules are drawn, modules are ordered by the similarity of their summary vectors in the dataset(s) specified in `orderNodesBy` argument. If multiple datasets are provided to the `orderNodesBy` argument then the module summary vectors are concatenated across datasets.

By default, samples in the data heatmap and accompanying module summary bar plot are ordered in descending order of *module summary* in the drawn dataset (specified by the `test` argument). If multiple modules are drawn, samples are ordered as per the left-most module on the plot.

Alternatively, a vector containing the name or index of another dataset may be provided to the `orderSamplesBy` argument. In this case, samples will be ordered in descending order of *module summary* in the specified dataset. This is useful when comparing different measurements across samples, for example, gene expression data obtained from multiple tissues samples across the same individuals. If the dataset specified is the `discovery` dataset, then missing samples will be displayed as horizontal grey bars. If the dataset specified is one of the other datasets, samples present in both the specified dataset and the `test` dataset will be displayed first in order of the specified dataset, then samples present in only the test dataset will be displayed underneath a horizontal black line ordered by their module summary vector in the test dataset.

Order of samples by *module summary* can be suppressed by setting `orderSamplesBy` to `NA`, in which case samples will be order as in the matrix provided to the `data` argument for the drawn dataset.

Weighted degree scaling:: When drawn on a plot, the weighted degree of each node is scaled to the maximum weighted degree within its module. The scaled weighted degree is measure of relative importance for each node to its module. This makes visualisation of multiple modules with different sizes and densities possible. However, the scaled weighted degree should only be interpreted for groups of nodes that have an apparent module structure.

Plot layout and device size: For optimal results we recommend viewing single modules on a PNG device with a width of 1500, a height of 2700 and a nominal resolution of 300 (`png(filename, width=5*300, height=9*300, res=300))`).

Warning: PDF and other vectorized devices should not be used when plotting more than a hundred nodes. Large files will be generated which may cause image editing programs such as Inkscape or Illustrator to crash when polishing figures for publication.

When `dryRun` is `TRUE` only the axes, legends, labels, and title will be drawn, allowing for quick iteration of customisable parameters to get the plot layout correct.

If axis labels or legends are drawn off screen then the margins of the plot should be adjusted prior to plotting using the `par` command to increase the margin size (see the "mar" option in the `par` help page).

The size of text labels can be modified by increasing or decreasing the `cex.main`, `cex.lab`, and `cex.axis` arguments:

- `cex.main`: controls the size of the plot title (specified in the `main` argument).
- `cex.lab`: controls the size of the axis labels on the *weighted degree*, *node contribution*, and *module summary* bar plots as well as the size of the module labels and the heatmap legend titles.
- `cex.axis`: contols the size of the axis tick labels, including the node and sample labels.

The position of these labels can be changed through the following arguments:

- `xaxt.line`: controls the distance from the plot the x-axis tick labels are drawn on the *module summary* bar plot.
- `xlab.line`: controls the distance from the plot the x-axis label is drawn on the *module summary* bar plot.
- `yaxt.line`: controls the distance from the plot the y-axis tick labels are drawn on the *weighted degree* and *node contribution* bar plots.
- `ylab.line`: controls the distance from the plot the y-axis label is drawn on the *weighted degree* and *node contribution* bar plots.
- `main.line`: controls the distance from the plot the title is drawn.
- `naxt.line`: controls the distance from the plot the node labels are drawn.
- `saxt.line`: controls the distance from the plot the sample labels are drawn.
- `maxt.line`: controls the distance from the plot the module labels are drawn.
- `laxt.line`: controls the distance from the heatmap legends that the gradient legend labels are drawn.
- `legend.main.line`: controls the distance from the heatmap legends that the legend title is drawn.

The rendering of node, sample, and module names can be disabled by setting `plotNodeNames`, `plotSampleNames`, and `plotModuleNames` to `FALSE`.

The size of the axis ticks can be changed by increasing or decreasing the following arguments:

- `xaxt.tck`: size of the x-axis tick labels as a multiple of the height of the *module summary* bar plot

- yaxt.tck: size of the y-axis tick labels as a multiple of the width of the *weighted degree* or *node contribution* bar plots.
- laxt.tck: size of the heatmap legend axis ticks as a multiple of the width of the data, correlation structure, or network edge weight heatmaps.

The drawBorders argument controls whether borders are drawn around the weighted degree, node contribution, or module summary bar plots. The lwd argument controls the thickness of these borders, as well as the thickness of axes and axis ticks.

Modifying the color palettes:: The dataCols and dataRange arguments control the appearance of the data heatmap (see plotData). The gradient of colors used on the heatmap can be changed by specifying a vector of colors to interpolate between in dataCols and dataRange specifies the range of values that maps to this gradient. Values outside of the specified dataRange will be rendered with the colors used at either extreme of the gradient. The default gradient is determined based on the data shown on the plot. If all values in the data matrix are positive, then the gradient is interpolated between white and green, where white is used for the smallest value and green for the largest. If all values are negative, then the gradient is interpolated between purple and white, where purple is used for the smallest value and white for the value closest to zero. If the data contains both positive and negative values, then the gradient is interpolated between purple, white, and green, where white is used for values of zero. In this case the range shown is always centered at zero, with the values at either extreme determined by the value in the rendered data with the strongest magnitude (the maximum of the absolute value).

The corCols and corRange arguments control the appearance of the correlation structure heatmap (see plotCorrelation). The gradient of colors used on the heatmap can be changed by specifying a vector of colors to interpolate between in corCols. By default, strong negative correlations are shown in blue, and strong positive correlations in red, and weak correlations as white. corRange controls the range of values that this gradient maps to, by default, -1 to 1. Changing this may be useful for showing differences where range of correlation coefficients is small.

The netCols and netRange arguments control the appearance of the network edge weight heatmap (see plotNetwork). The gradient of colors used on the heatmap can be changed by specifying a vector of colors to interpolate between in netCols. By default, weak or non-edges are shown in white, while strong edges are shown in red. The netRange controls the range of values this gradient maps to, by default, 0 to 1. If netRange is set to NA, then the gradient will be mapped to values between 0 and the maximum edge weight of the shown network.

The degreeCol argument controls the color of the weighted degree bar plot (see plotDegree).

The contribCols argument controls the color of the node contribution bar plot (see plotContribution. This can be specified as single value to be used for all nodes, or as two colors: one to use for nodes with positive contributions and one to use for nodes with negative contributions.

The summaryCols argument controls the color of the module summary bar plot (see plotSummary. This can be specified as single value to be used for all samples, or as two colors: one to use for samples with a positive module summary value and one fpr samples with a negative module summary value.

The naCol argument controls the color of missing nodes and samples on the data, correlaton structure, and network edge weight heatmaps.

Embedding in Rmarkdown documents: The chunk option fig.keep="last" should be set to avoid an empty plot being embedded above the plot generated by plotModule. This empty plot is generated so that an error will be thrown as early as possible if the margins are too small to be displayed. Normally, these are drawn over with the actual plot components when drawing the plot on other graphical devices.

See Also

`plotCorrelation`, `plotNetwork`, `plotDegree`, `plotContribution`, `plotData`, and `plotSummary`.

Examples

```
# load in example data, correlation, and network matrices for a discovery
# and test dataset:
data(”NetRep”)

# Set up input lists for each input matrix type across datasets. The list
# elements can have any names, so long as they are consistent between the
# inputs.
network_list <- list(discovery=discovery_network, test=test_network)
data_list <- list(discovery=discovery_data, test=test_data)
correlation_list <- list(discovery=discovery_correlation, test=test_correlation)
labels_list <- list(discovery=module_labels)

# Plot module 1, 2 and 4 in the discovery dataset
plotModule(
  network=network_list, data=data_list, correlation=correlation_list,
  moduleAssignments=labels_list, modules=c(1, 2, 4)
)

# Now plot them in the test dataset (module 2 does not replicate)
plotModule(
  network=network_list,data=data_list, correlation=correlation_list,
  moduleAssignments=labels_list, modules=c(1, 2, 4), discovery=”discovery”,
  test=”test”
)

# Plot modules 1 and 4, which replicate, in the test datset ordering nodes
# by weighted degree averaged across the two datasets
plotModule(
  network=network_list, data=data_list, correlation=correlation_list,
  moduleAssignments=labels_list, modules=c(1, 4), discovery=”discovery”,
  test=”test”, orderNodesBy=c(”discovery”, ”test”)
)
```

`requiredPerms` *How many permutations do I need to test at my desired significance level?*

Description

How many permutations do I need to test at my desired significance level?

Usage

```
requiredPerms(alpha, alternative = "greater")
```

Arguments

`alpha`	desired significance threshold.
`alternative`	a character string specifying the alternative hypothesis, must be one of "greater" (default), "less", or "two.sided". You can specify just the initial letter.

Value

The minimum number of permutations required to detect any significant associations at the provided `alpha`. The minimum p-value will always be smaller than `alpha`.

Examples

```
data("NetRep")

# Set up input lists for each input matrix type across datasets. The list
# elements can have any names, so long as they are consistent between the
# inputs.
network_list <- list(discovery=discovery_network, test=test_network)
data_list <- list(discovery=discovery_data, test=test_data)
correlation_list <- list(discovery=discovery_correlation, test=test_correlation)
labels_list <- list(discovery=module_labels)

# How many permutations are required to Bonferroni adjust for the 4 modules
# in the example data?
nPerm <- requiredPerms(0.05/4)

# Note that we recommend running at least 10,000 permutations to make sure
# that the null distributions are representative.

preservation <- modulePreservation(
 network=network_list, data=data_list, correlation=correlation_list,
 moduleAssignments=labels_list, nPerm=nPerm, discovery="discovery",
 test="test"
)
```

Index