Package 'diffuStats'

November 11, 2026

Type Package

Title Diffusion scores on biological networks

Version 1.32.0

Description Label propagation approaches are a widely used procedure in computational biology for giving context to molecular entities using network data. Node labels, which can derive from gene expression, genome-wide association studies, protein domains or metabolomics profiling, are propagated to their neighbours in the network, effectively smoothing the scores through prior annotated knowledge and prioritising novel candidates. The R package diffuStats contains a collection of diffusion kernels and scoring approaches that facilitates their computation, characterisation and benchmarking.

Depends R (>= 3.4)

Imports grDevices, stats, methods, Matrix, MASS, checkmate, expm, igraph, Rcpp, RcppArmadillo, RcppParallel, plyr, precrec

License GPL-3

LazyData true

Encoding UTF-8

RoxygenNote 7.1.1

Suggests testthat, knitr, rmarkdown, ggplot2, ggsci, igraphdata, BiocStyle, reshape2, utils

LinkingTo Rcpp, RcppArmadillo, RcppParallel

SystemRequirements GNU make

VignetteBuilder knitr

biocViews Network, GeneExpression, GraphAndNetwork, Metabolomics, Transcriptomics, Proteomics, Genetics, GenomeWideAssociation, Normalization

git_url https://git.bioconductor.org/packages/diffuStats

git_branch RELEASE_3_23

git_last_commit 1a7620f

git_last_commit_date 2026-04-28

Repository Bioconductor 3.23

Date/Publication 2026-11-11

Author Sergio Picart-Armada [aut, cre],
Alexandre Perera-Lluna [aut]

Maintainer Sergio Picart-Armada <sergi.picart@upc.edu>

Contents

.check_scores . . . 3
.connect_undirected_graph . . . 4
.default_graph_param . . . 5
convertSparse . . . 5
diffuse . . . 6
diffuse_mc . . . 11
diffuse_raw . . . 12
diffuStats . . . 13
generate_graph . . . 13
generate_input . . . 14
graph_toy . . . 15
is_kernel . . . 16
kernels . . . 16
largest_cc . . . 18
metric_auc . . . 19
moments . . . 21
named.list . . . 23
ParallelHeatrank . . . 23
perf . . . 24
perf_eval . . . 25
perf_wilcox . . . 26
scores2colours . . . 27
scores2shapes . . . 28
serialHeatrank . . . 29
sparsify2 . . . 29
to_list . . . 30
to_x_from_list . . . 30
which_format . . . 31

Index **32**

`.check_scores` *Sanity checks for input*

Description

.check_scores ensures that scores are suitable for diffusion

.available_methods is a character vector with the implemented scores

.check_method ensures that 'method' is a valid character

.check_metric ensures that 'metric' is a valid list of metric functions

.check_graph ensures that 'graph' is a valid `igraph` object

.check_K ensures that 'K' is a formally valid kernel. Does not check for spd

Usage

```
.check_scores(scores)

.available_methods

.check_method(method)

.check_metric(metric)

.check_graph(graph)

.check_K(K)
```

Arguments

`scores`	scores to check
`method`	object to test
`metric`	object to test
`graph`	object to test
`K`	object to test

Format

An object of class `character` of length 7.

Value

Functions return `invisible()` but throw warnings and errors as side effect

Examples

```
data(graph_toy)
diffuStats:::.check_scores(diffuStats:::to_list(graph_toy$input_mat))
diffuStats:::.check_method("raw")
diffuStats:::.check_metric(list(auc = metric_fun(curve = "ROC")))
data(graph_toy)
diffuStats:::.check_graph(graph_toy)
data(graph_toy)
diffuStats:::.check_K(regularisedLaplacianKernel(graph_toy))
```

.connect_undirected_graph

Function to connect a non connected graph

Description

Function to connect a non connected graph

Usage

```
.connect_undirected_graph(g)
```

Arguments

g	an igraph object

Value

a connected igraph object

Examples

```
library(igraph)
g <- diffuStats:::.connect_undirected_graph(
    graph.empty(10, directed = FALSE))
g
```

.default_graph_param *Generate data.frame with default vertex attributes*

Description

Generate data.frame with default vertex attributes

Default proportions for randomly generated graphs

Usage

```
.default_graph_param()

.default_prop
```

Format

An object of class `numeric` of length 3.

Value

data.frame with default node class attributes

named numeric with default class proportions

convertSparse *S4 sparse matrix to arma::sp_mat*

Description

Convert an S4 sparse matrix from the `Matrix` package to an arma `sp_mat`.

Usage

```
convertSparse(mat)
```

Arguments

`mat`	S4 sparse matrix from the `Matrix`

Value

an arma::sp_mat object

Source

http://gallery.rcpp.org/articles/armadillo-sparse-matrix/

`diffuse` *Diffuse scores on a network*

Description

Function `diffuse` takes a network in **igraph** format (or a graph kernel matrix stemming from a graph) and an initial state to score all the nodes in the network. The seven diffusion scores hereby provided differ on (a) how they distinguish positives, negatives and unlabelled examples, and (b) their statistical normalisation. The argument `method` offers the following options:

Methods without statistical normalisation:

- `raw`: positive nodes introduce unitary flow (`y_raw[i] = 1`) to the network, whereas neither negative nor unlabelled nodes introduce anything (`y_raw[j] = 0`) [Vandin, 2011]. They are computed as:

$$f_{raw} = K \cdot y_{raw}$$

  where `K` is a graph kernel, see `?kernels`. These scores treat negative and unlabelled nodes equivalently.

- `ml`: same as `raw`, but negative nodes introduce a negative unit of flow [Zoidi, 2015] and are therefore not equivalent to unlabelled nodes.

- `gm`: same as `ml`, but the unlabelled nodes are assigned a (generally non-null) bias term based on the total number of positives, negatives and unlabelled nodes [Mostafavi, 2008].

- `ber_s`: this is a quantification of the relative change in the node score before and after the network smoothing. The score for a particular node `i` can be written as

$$f_{ber_s,i} = \frac{f_{raw,i}}{y_{raw,i} + \epsilon}$$

  where `eps` is a parameter controlling the importance of the relative change.

Methods with statistical normalisation: the `raw` diffusion score of every node `i` is computed and compared to its own diffusion scores stemming from a permuted input.

- `mc`: the score of node `i` is based in its empirical p-value, computed by permuting the input `n.perm` times:

$$p_i = \frac{r_i + 1}{n.perm + 1}$$

  `p[i]` is roughly the proportion of input permutations that led to a diffusion score as high or higher than the original diffusion score (a total of `r[i]` for node `i`, in absolute terms). This assesses how likely a high diffusion score is to arise from chance, in absence of signal. To be consistent with the direction, `mc` is defined as:

$$f_{mc,i} = 1 - p_i$$

- `ber_p`: as used in [Bersanelli, 2016], this score combines `raw` and `mc`, in order to take into account both the magnitude of the `raw` scores and the effect of the network topology:

$$f_{ber_p,i} = -\log_{10}(p_i) \cdot f_{raw,i}$$

- z: this is a parametric alternative to mc. The raw score of node i is subtracted its mean value and divided by its standard deviation. The statistical moments have a closed analytical form, see the main vignette, and are inspired in [Harchaoui, 2013]. Unlike mc and ber_p, the z scores do not require actual permutations, giving them an advantage in terms of speed.

If the input labels are not quantitative, i.e. positive(1), negative(0) and possibly unlabelled, all the scores (raw, gm, ml, z, mc, ber_s, ber_p) can be used. Quantitative inputs are naturally defined on raw, z, mc, ber_s and ber_p by extending the definitions above, and are readily available in diffuStats. Further details on the scores can be found in the main vignette.

Usage

```
diffuse(graph, scores, method, ...)

diffuse_grid(scores, grid_param, ...)
```

Arguments

graph	**igraph** object for the diffusion. Alternatively, a kernel matrix can be provided through the argument K insted of the igraph object.
scores	scores to be smoothed; either a named numeric vector, a column-wise matrix whose rownames are nodes and colnames are different scores, or a named list of such matrices.
method	character, one of raw, gm, ml, z, mc, ber_s, ber_p. For batch analysis of several methods, see ?diffuse_grid.
...	additional arguments for the diffusion method. mc and ber_p accept n.perm (number of permutations), seed (for reproducibility, defaults to 1) and sample.prob, a list of named vectors -one per background- with sampling probabilities for the null model, uniform by default. More details available in ?diffuse_mc. On the other hand, ber_s accepts eps, a parameter controlling the importance of the relative change.
grid_param	data frame containing parameter combinations to explore. The column names should be the names of the parameters. Parameters that have a fixed value can be specified in the grid or through the additional arguments (...)

Details

Input scores can be specified in three formats. A single set of scores to smooth can be represented as (1) a named numeric vector, whereas if several of these vectors that share the node names need to be smoothed, they can be provided as (2) a column-wise matrix. However, if the unlabelled entities are not the same from one case to another, (3) a named list of such score matrices can be passed to this function. The input format will be kept in the output.

The implementation of mc and ber_p is optimized for sparse inputs. Dense inputs might take a longer time to compute. Another relevant note: z can give NaN for a particular node when the observed nodes are disconnected from the node being scored. This is because these nodes are neither annotated with experimental not network (topology) data.

Value

`diffuse` returns the diffusion scores, with the same format as `scores`

`diffuse_grid` returns a data frame containing the diffusion scores for the specified combinations of parameters

References

Scores "raw": Vandin, F., Upfal, E., & Raphael, B. J. (2011). Algorithms for detecting significantly mutated pathways in cancer. Journal of Computational Biology, 18(3), 507-522.

Scores "ml": Zoidi, O., Fotiadou, E., Nikolaidis, N., & Pitas, I. (2015). Graph-based label propagation in digital media: A review. ACM Computing Surveys (CSUR), 47(3), 48.

Scores "gm": Mostafavi, S., Ray, D., Warde-Farley, D., Grouios, C., & Morris, Q. (2008). GeneMANIA: a real-time multiple association network integration algorithm for predicting gene function. Genome biology, 9(1), S4.

Scores "mc", "ber_s", "ber_p": Bersanelli, M., Mosca, E., Remondini, D., Castellani, G., & Milanesi, L. (2016). Network diffusion-based analysis of high-throughput data for the detection of differentially enriched modules. Scientific reports, 6.

Scores "z": Harchaoui, Z., Bach, F., Cappe, O., & Moulines, E. (2013). Kernel-based methods for hypothesis testing: A unified view. IEEE Signal Processing Magazine, 30(4), 87-97.

Examples

```
##############################

library(igraph)
library(ggplot2)
data(graph_toy)
input_vec <- graph_toy$input_vec
n <- vcount(graph_toy)

##############################

# Examples for 'diffuse':

# Using a binary vector as input
diff_scores <- diffuse(
    graph = graph_toy,
    scores = input_vec,
    method = "raw")

# Using a matrix as input
diff_scores <- diffuse(
    graph = graph_toy,
    scores = graph_toy$input_mat,
    method = "raw")

# Using a list of matrices as input
diff_scores <- diffuse(
    graph = graph_toy,
```

```
    scores = list(myScores1 = graph_toy$input_mat,
        myScores2 = head(graph_toy$input_mat, n/2)),
    method = "raw")

##############################

# Examples for 'diffuse_grid':

# Using a single vector of scores and comparing the methods
# "raw", "ml", and "z"
df_diff <- diffuse_grid(
    graph = graph_toy,
    scores = graph_toy$input_vec,
    grid_param = expand.grid(method = c("raw", "ml", "z")))
head(df_diff)

# Same settings, but comparing several choices of the
# parameter epsilon ("eps") in the scores "ber_s"
df_diff <- diffuse_grid(
    graph = graph_toy,
    scores = graph_toy$input_vec,
    grid_param = expand.grid(method = "ber_s", eps = 1:5/5))
ggplot(df_diff, aes(x = factor(eps), fill = eps, y = node_score)) +
    geom_boxplot()

# Using a matrix with four set of scores
# called Single, Row, Small_sample, Large_sample
# See the 'quickstart' vignette for more details on these toy scores
# We compute scores for methods "ber_p" and "mc" and
# permute both 1e3 and 1e4 times in each run
df_diff <- diffuse_grid(
    graph = graph_toy,
    scores = graph_toy$input_mat,
    grid_param = expand.grid(
        method = c("mc", "ber_p"),
        n.perm = c(1e3, 1e4)))
dim(df_diff)
head(df_diff)

##############################

# Differences when using (1) a quantitative input and
# (2) different backgrounds.

# In this example, the
# small background contains binary scores and continuous scores for
# half of the nodes in the 'graph_toy' example graph.

# (1) Continuous scores have been generated by
# changing the positive labels to a random, positive numeric value.
# The user can see the impact of this in the scores 'raw', 'ber_s',
# 'ber_p', 'mc' and 'z'
```

```
# (2) The larger background is just the small background
# completed with zeroes, both for binary and continuous scores.
# This illustrates how 'raw' and 'ber_s' treat unlabelled
# and negative labels equally, whereas 'ml', 'gm', 'ber_p',
# 'mc' and 'z' do not.

# Examples:

# The input:
lapply(graph_toy$input_list, head)

# 'raw' scores treat equally unlabelled and negative nodes,
# and can account for continuous inputs
diff_raw <- diffuse(
    graph = graph_toy,
    scores = graph_toy$input_list,
    method = "raw")
lapply(diff_raw, head)

# 'z' scores distinguish unlabelled and negatives and accepts
# continuous inputs
diff_z <- diffuse(
    graph = graph_toy,
    scores = graph_toy$input_list,
    method = "z")
lapply(diff_z, head)

# 'ml' and 'gm' are the same score if there are no unobserved nodes
diff_compare <- diffuse_grid(
    graph = graph_toy,
    scores = input_vec,
    grid_param = expand.grid(method = c("raw", "ml", "gm"))
)
df_compare <- reshape2::acast(
    diff_compare,
    node_id~method,
    value.var = "node_score")
head(df_compare)

# 'ml' and 'gm' are different in presence of unobserved nodes
diff_compare <- diffuse_grid(
    graph = graph_toy,
    scores = head(input_vec, n/2),
    grid_param = expand.grid(method = c("raw", "ml", "gm"))
)
df_compare <- reshape2::acast(
    diff_compare,
    node_id~method,
    value.var = "node_score")
head(df_compare)
```

diffuse_mc *Compute the heatrank using permutations*

Description

Function diffuse_mc has an implemented parallelisation of the Monte Carlo trials for diffusion in a network. The input scores are assumed to be sparse and are internally sparsified, so very dense scores migth take time with current implementation.

Usage

```
diffuse_mc(
    graph,
    scores,
    n.perm = 10000,
    sample.prob = NULL,
    seed = 1,
    oneminusHeatRank = TRUE,
    K = NULL,
    ...
)
```

Arguments

graph	igraph object
scores	Recursive list, can have either binary or quantitative scores
n.perm	Numeric, number of permutations
sample.prob	Numeric, probabilities (needn't be scaled) to permute the input. This is passed to sample's prob argument. If NULL, sampling is uniform. It has to be in a list format, with the same names as scores, and each element of the list must be the sampling probability of each background.
seed	Numeric, seed for random number generator
oneminusHeatRank	Logical, should 1 - heatrank be returned instead of heatrank?
K	Kernel matrix (if precomputed). If K is not supplied, the regularised Laplacian will be computed on the fly and used.
...	currently ignored arguments

Value

A list containing matrices of heatrank scores

Examples

```
# Using a list as input (needed)
data(graph_toy)
list_input <- list(myInput1 = graph_toy$input_mat)
diff_mc <- diffuse_mc(
    graph = graph_toy,
    scores = list_input)
```

`diffuse_raw` *Diffuse scores on a network*

Description

Function `diffuse` takes a network in **igraph** format and an initial state to score all the nodes in the network.

Usage

```
diffuse_raw(graph, scores, z = FALSE, K = NULL, ...)
```

Arguments

`graph`	**igraph** object for the diffusion
`scores`	list of score matrices. For a single input with a single background, supply a list with a vector column
`z`	logical, should z-scores be computed instead of raw scores?
`K`	optional matrix, precomputed diffusion kernel
`...`	currently ignored arguments

Value

A list of scores, with the same length and dimensions as `scores`

Examples

```
# Using a list as input (needed)
data(graph_toy)
list_input <- list(myInput1 = graph_toy$input_mat)
diff_raw <- diffuse_raw(
    graph = graph_toy,
    scores = list_input)
diff_z <- diffuse_raw(
    graph = graph_toy,
    scores = list_input,
    z = TRUE)
```

diffuStats *diffuStats: an R package to compute and benchmark diffusion scores*

Description

The diffuStats package consists of (i) functions to compute graph kernels, see kernels, (ii) the function diffuse to compute the diffusion scores and (iii) the function perf_eval and its wrapper perf to compute performance measures. The user can find two vignettes in browseVignettes("diffuStats"): (1) a quick start with concise examples and (2) a detailed explanation of the implemented methods with a practical case study using a yeast protein dataset.

Author(s)

Sergio Picart-Armada <sergi.picart@upc.edu>, Alexandre Perera-Lluna

References

General references:

Most of the graph kernels can be found in: Smola, A. J., & Kondor, R. (2003, August). Kernels and regularization on graphs. In COLT (Vol. 2777, pp. 144-158).

The statistical normalisation of the diffusion scores, which has interest per se, has been introduced in: Bersanelli, M., Mosca, E., Remondini, D., Castellani, G., & Milanesi, L. (2016). Network diffusion-based analysis of high-throughput data for the detection of differentially enriched modules. Scientific reports, 6.

generate_graph *Generate a random graph*

Description

Function generate_graph generates a random network using **igraph** graph generators. Several models are available, and

Usage

```
generate_graph(
    fun_gen,
    param_gen,
    class_label = NULL,
    class_attr = .default_graph_param(),
    fun_curate = .connect_undirected_graph,
    seed = NULL
)
```

Arguments

fun_gen	function to generate the graphs. Typically from **igraph**, like barabasi.game, watts.strogatz.game, erdos.renyi.game, make_lattice, etc.
param_gen	list with parameters to pass to fun_gen
class_label	character vector with length equal to the number of nodes in the graph to generate. If left to NULL, the default classes are c("source", "filler", "end") with proportions of c(0.05, 0.45, 0.5).
class_attr	data.frame with vertex classes as rownames and a column for each vertex attribute. The name of the column will be used as the attribute name.
fun_curate	function to apply to the graph before returning it. Can be set to identity or NULL to skip this step. By default, the graph is connected: nodes not belonging to the largest connected component are randomly wired to a node in it.
seed	numeric, seed for random number generator

Value

An **igraph** object

Examples

```
g <- generate_graph(
    fun_gen = igraph::barabasi.game,
    param_gen = list(n = 100, m = 3, directed = FALSE),
    seed = 1)
g
```

---

generate_input *Generate a random input for graph diffusion*

Description

Function generate_input generates a random list of nodes from an **igraph** object. It also specifies the true solution generating the list. The graph object needs to have some attributes (automatically added through generate_graph)

Usage

```
generate_input(graph, order, length_inputs, return_matrix = TRUE, seed = NULL)
```

Arguments

graph	an **igraph** object, typically from generate_input
order	numeric or vector, order of the neighbourhoods that generate the list
length_inputs	numeric, number of nodes in the generated inputs
return_matrix	logical, should inputs be returned as a matrix?
seed	numeric, seed for random number generator

Value

A list whose elements are lists with three slots: pos for the true signal generators, neg for the nodes that did not generate signal and input for the signal itself

Examples

```
g <- generate_graph(
    fun_gen = igraph::barabasi.game,
    param_gen = list(n = 200, m = 3, directed = FALSE),
    seed = 1)
synth_input <- generate_input(
    g,
    order = 2,
    length_inputs = 3, return_matrix = TRUE)
str(synth_input)
```

---

graph_toy *Toy graph to play with diffusion*

Description

Small graph that can easily be plotted and experimented with. It has graphical parameters included, such as the vertex colour and the layout. It also includes an example input. Has graph attributes with example inputs and outputs, see input_* and output_* from list.graph.attributes(graph_toy)

Usage

```
graph_toy
```

Format

An object of class igraph of length 10.

Value

An **igraph** object

is_kernel *Check if a matrix is a valid kernel*

Description

This function checks whether the eigenvalues are non-negative

Usage

```
is_kernel(x, tol = 1e-08)
```

Arguments

x	numeric, symmetric matrix to be checked
tol	numeric, tolerance for zero eigenvalues

Value

scores in desired format

Examples

```
data(graph_toy)
K <- regularisedLaplacianKernel(graph_toy)
is_kernel(K)
is_kernel(K - 1)
```

kernels *Compute graph kernels*

Description

Function `commuteTimeKernel` computes the conmute-time kernel, which is the expected time of going back and forth between a couple of nodes. If the network is connected, then the commute time kernel will be totally dense, therefore reflecting global properties of the network. For further details, see [Yen, 2007]. This kernel can be computed using both the unnormalised and normalised graph Laplacian.

Function `diffusionKernel` computes the classical diffusion kernel that involves matrix exponentiation. It has a "bandwidth" parameter σ^2 that controls the extent of the spreading. Quoting [Smola, 2003]: K(x1,x2) can be visualized as the quantity of some substance that would accumulate at vertex x2 after a given amount of time if we injected the substance at vertex x1 and let it diffuse through the graph along the edges. This kernel can be computed using both the unnormalised and normalised graph Laplacian.

Function `inverseCosineKernel` computes the inverse cosine kernel, which is based on a cosine transform on the spectrum of the normalized Laplacian matrix. Quoting [Smola, 2003]: the inverse cosine kernel treats lower complexity functions almost equally, with a significant reduction in the upper end of the spectrum. This kernel is computed using the normalised graph Laplacian.

Function `pStepKernel` computes the p-step random walk kernel. This kernel is more focused on local properties of the nodes, because random walks are limited in terms of length. Therefore, if `p` is small, only a fraction of the values K(x1,x2) will be non-null if the network is sparse [Smola, 2003]. The parameter `a` is a regularising term that is summed to the spectrum of the normalised Laplacian matrix, and has to be 2 or greater. The p-step kernels can be cheaper to compute and have been successful in biological tasks, see the benchmark in [Valentini, 2014].

Function `regularisedLaplacianKernel` computes the regularised Laplacian kernel, which is a standard in biological networks. The regularised Laplacian kernel arises in numerous situations, such as the finite difference formulation of the diffusion equation and in Gaussian process estimation. Sticking to the heat diffusion model, this function allows to control the constant terms summed to the diagonal through `add_diag`, i.e. the strength of the leaking in each node. If a node has diagonal term of `0`, it is not allowed to disperse heat. The larger the diagonal term of a node, the stronger the first order heat dispersion in it, provided that it is positive. Every connected component in the graph should be able to disperse heat, i.e. have at least a node `i` with `add_diag[i] > 0`. If this is not the case, the result diverges. More details on the parameters can be found in [Smola, 2003]. This kernel can be computed using both the unnormalised and normalised graph Laplacian.

Usage

```
commuteTimeKernel(graph, normalized = FALSE)

diffusionKernel(graph, sigma2 = 1, normalized = TRUE)

inverseCosineKernel(graph)

pStepKernel(graph, a = 2, p = 5L)

regularisedLaplacianKernel(graph, sigma2 = 1, add_diag = 1, normalized = FALSE)
```

Arguments

`graph`	undirected igraph object. If the edges have weights, those should typically be non-negative.
`normalized`	logical, should the normalised (`TRUE`) or unnormalised (`FALSE`) graph Laplacian matrix be used?
`sigma2`	numeric value, parameter σ^2 of the kernel - higher values force more spreading in the network
`a`	numeric value greater or equal to 2, which acts as a regularisation term. Can also be a vector of length `vcount(graph)`
`p`	integer greater than 0, the number of steps for the random walk
`add_diag`	numeric value or vector of length `vcount(graph)`, term to regularise the spectrum of the Laplacian

Details

Please be aware that the kernel computation can be rather slow and memory demanding. This is a reference table of the peak memory usage and computing time for the regularised Laplacian kernel given the order of the network:

5k: 900MB & 250s

10k: 3,200MB & 2,200s

15k: 8,000MB & 8,000s

20k: 13,000MB & 21,000s

However, given a network to study, this step is a one-time task than can be stored and reused.

Value

A kernel matrix with adequate dimnames

References

The regularised Laplacian, diffusion, p-step and inverse cosine kernels: Smola, A. J., & Kondor, R. (2003, August). Kernels and regularization on graphs. In COLT (Vol. 2777, pp. 144-158).

The commute time kernel: Yen, L., Fouss, F., Decaestecker, C., Francq, P., & Saerens, M. (2007). Graph nodes clustering based on the commute-time kernel. Advances in Knowledge Discovery and Data Mining, 1037-1045.

Benchmark on kernels: Valentini, G., Paccanaro, A., Caniza, H., Romero, A. E., & Re, M. (2014). An extensive analysis of disease-gene associations using network integration and fast kernel-based gene prioritization methods. Artificial Intelligence in Medicine, 61(2), 63–78.

Examples

```
data(graph_toy)
K_lap <- regularisedLaplacianKernel(graph_toy)
K_diff <- diffusionKernel(graph_toy)
K_pstep <- pStepKernel(graph_toy)
K_ct <- commuteTimeKernel(graph_toy)
K_ic <- inverseCosineKernel(graph_toy)
is_kernel(K_lap)
```

---

largest_cc *Largest connected component*

Description

Obtain the largest connected component of an `igraph` object

Usage

```
largest_cc(g)
```

Arguments

g	igraph object

Value

A connected igraph object

Examples

```
library(igraph)
set.seed(1)
g <- erdos.renyi.game(30, p.or.m = .05)
largest_cc(g)
```

---

metric_auc *Compute the area under the curves (ROC, PRC)*

Description

Function metric_auc computes the AUROC (Area Under the Receiver Operating Characteristic Curve) and the AUPRC (Area Under the Precision Recall Curve), measures of goodness of a ranking in a binary classification problem. Partial areas are also supported. Important: the higher ranked classes are assumed to ideally target positives (label = 1) whereas lower ranks correspond to negatives (label = 0).

Function metric_fun is a wrapper on metric_auc that returns a function for performance evaluation. This function takes as input actual and predicted values and outputs a performance metric. This is needed for functions such as perf and perf_eval, which iterate over a list of such metric functions and return the performance measured through each of them.

Usage

```
metric_auc(
    actual,
    predicted,
    curve = "ROC",
    partial = c(0, 1),
    standardized = FALSE
)

metric_fun(...)
```

Arguments

actual	numeric, binary labels of the negatives (0) and positives (1)
predicted	numeric, prediction used to rank the entities - this will typically be the diffusion scores
curve	character, either "ROC" for computing the AUROC or "PRC" for the AUPRC
partial	vector with two numeric values for computing partial areas. The numeric values are the limits in the x axis of the curve, as implemented in the "xlim" argument in part. Defaults to c(0,1), i.e. the whole area
standardized	logical, should partial areas be standardised to range in [0, 1]? Defaults to FALSE and only affects partial areas.
...	parameters to pass to metric_auc

Details

The AUROC is a scalar value: the probability of a randomly chosen positive having a higher rank than a randomly chosen negative. AUROC is cutoff-free and an informative of the performance of a ranker. Likewise, AUPRC is the area under the Precision-Recall curve and is also a standard metric for binary classification. Both measures can be found in [Saito, 2017].

AUROC and AUPRC have their partial counterparts, in which only the area enclosed up to a certain false positive rate (AUROC) or recall (AUPRC) is accounted for. This can be useful when assessing the goodness of the ranking, focused on the top entities.

The user can, however, define his or her custom performance metric. AUROC and AUPRC are common choices, but other problem-specific metrics might be of interest. For example, number of hits in the top k nodes. Machine learning metrics can be found in packages such as Metrics and MLmetrics from the CRAN repository (http://cran.r-project.org/).

Value

metric_auc returns a numeric value, the area under the specified curve

metric_fun returns a function (performance metric)

References

Saito, T., & Rehmsmeier, M. (2017). Precrec: fast and accurate precision–recall and ROC curve calculations in R. Bioinformatics, 33(1), 145-147.

Examples

```
# generate class and numeric ranking
set.seed(1)
n <- 50
actual <- rep(0:1, each = n/2)
predicted <- ifelse(
    actual == 1,
    runif(n, min = 0.2, max = 1),
    runif(n, min = 0, max = 0.8))
```

```
# AUROC
metric_auc(actual, predicted, curve = "ROC")

# partial AUC (up until false positive rate of 10%)
metric_auc(
    actual, predicted, curve = "ROC",
    partial = c(0, 0.1))

# The same are, but standardised in (0, 1)
metric_auc(
    actual, predicted, curve = "ROC",
    partial = c(0, 0.1), standardized = TRUE)

# AUPRC
metric_auc(actual, predicted, curve = "PRC")

# Generate performance functions for perf and perf_eval
f_roc <- metric_fun(
    curve = "ROC", partial = c(0, 0.5),
    standardized = TRUE)
f_roc
f_roc(actual = actual, predicted = predicted)
```

`moments` *Compute exact statistical moments*

Description

Function `get_mu()` computes the exact expected values of the null distributions

Function `get_covar()` computes the exact covariance matrix of the null distributions (square matrix, same size as kernel matrix); the variances are the values in the matrix diagonal

Function `get_mu_reference()` computes the reference expected values (one scalar value for each node/entity)

Function `get_var_reference()` computes the reference variances (one scalar value for each node/entity), log10-transformed

Usage

```
get_mu(K, id_labelled = colnames(K), mu_y)

get_covar(K, id_labelled = colnames(K), var_y)

get_mu_reference(K, id_labelled = colnames(K))

get_var_reference(K, id_labelled = colnames(K))
```

Arguments

K	square matrix, precomputed diffusion graph kernel, see ?kernels
id_labelled	character, names of the labelled nodes (must be a subset of the colnames of K)
mu_y, var_y	(scalar) mean and variance of the input, see details

Details

These functions enable exploring the properties of the null distributions of diffusion scores. They provide the exact statistical moments mentioned in:

Sergio Picart-Armada, Wesley K Thompson, Alfonso Buil, Alexandre Perera-Lluna. The effect of statistical normalisation on network propagation scores. Bioinformatics, 2020, btaa896. https://doi.org/10.1093/bioinformatic

Specifically, `get_mu_reference()` and `get_var_reference()` provide the so-called 'Reference expected values' and 'Reference variances', which are input-independent (one only needs the kernel and the ids of the labelled nodes). Getting the actual expected values and variances requires providing the input expected value and variance, and can be achieved with `get_mu()` and `get_covar()`.

Value

`get_mu_reference()`, `get_var_reference()` and `get_mu()` return a vector, whereas `get_covar()` returns a square matrix.

References

Article: https://doi.org/10.1093/bioinformatics/btaa896 Functions: https://github.com/b2slab/diffuBench/blob/master/helper_

Examples

```
data(graph_toy)
## Kernel
K_pstep <- pStepKernel(graph_toy)
## Labelled nodes
ids <- head(rownames(K_pstep), ncol(K_pstep)/3)
## Reference values
get_mu_reference(K_pstep, ids)
get_var_reference(K_pstep, ids)
## Actual moments with an input y
y <- graph_toy$input_vec[ids]
mu_y <- mean(y)
var_y <- var(y)
mu <- get_mu(K_pstep, ids, mu_y = mu_y)
covar <- get_covar(K_pstep, ids, var_y = var_y)
## mean values
mu
## variances
diag(covar)
## covariances
covar[1:6, 1:6]
```

named.list *Create a named list*

Description

Create a list with variables and name the slots using the variables names

Usage

```
named.list(...)
```

Arguments

...	Variables to pack in a list

Value

A list of variables

Examples

```
diffuStats:::named.list(LETTERS, mean)
```

ParallelHeatrank *Compute heatrank in parallel*

Description

ParallelHeatrank is a wrapper that computes heatranks for (possibly) different backgrounds and for multiple inputs at once. It will reuse the permutations, which have to be passed to the function. The input must be binary for this implementation, so numeric values for each node are not supported.

Usage

```
ParallelHeatrank(R, perm, G)
```

Arguments

R	dense matrix with the diffusion kernel
perm	dense matrix with the permutations (indices in columns). This has to ensure that enough indices are sampled, i.e. at least as great as the largest list in the input (largest colSums in G)
G	S4 sparse matrix with the heat sources

Value

a matrix with the same amount of rows that R and columns in G, containing the heatrank scores. These scores are corrected using (r + 1)/(p + 1) instead of r/p. The smaller the score, the warmer the node.

---

perf *Compare diffusions to a target score on a grid of parameters*

Description

Function perf computes diffusion scores on a grid of parameters and evaluates them using the gold standard scores provided by the user.

Usage

```
perf(
    scores,
    validation,
    grid_param,
    metric = list(auc = metric_fun(curve = "ROC")),
    ...
)
```

Arguments

scores	scores to be smoothed; either a named numeric vector, a column-wise matrix whose rownames are nodes and colnames are different scores, or a named list of such matrices.
validation	target scores to which the smoothed scores will be compared to. Must have the same format as the input scores, although the number of rows may vary and only the matching rows will give a performance measure
grid_param	data frame containing parameter combinations to explore. The column names should be the names of the parameters.
metric	named list of metrics to apply. Each metric should accept the form f(actual, predicted)
...	additional named arguments for the diffusion method. It's important to input at least an igraph object or, alternative, a kernel matrix K

Details

Function perf takes a network in **igraph** format, an initial state to score all the nodes in the network, a target score set. To explore the parameter combinations, it needs a grid and a list of metrics to apply. The validation scores might be only a subset of the network nodes, in which case the metric will be restricted to this set as well.

Value

A data frame containing the performance of each diffusion score

Examples

```
# Using a single vector of scores
data(graph_toy)
df_perf <- perf(
    graph = graph_toy,
    scores = graph_toy$input_vec,
    validation = graph_toy$input_vec,
    grid_param = expand.grid(method = c("raw", "ml")))
df_perf
# Using a matrix with four set of scores
# called Single, Row, Small_sample, Large_sample
df_perf <- perf(
    graph = graph_toy,
    scores = graph_toy$input_mat,
    validation = graph_toy$input_mat,
    grid_param = expand.grid(method = c("raw", "ml")))
df_perf
```

---

perf_eval *Compute performance of diffusion scores on a single case*

Description

Function `perf_eval` directly compares a desired output with the scores from diffusion. It handles the possible shapes of the scores (vector, matrix, list of matrices) and gives the desired metrics.

Usage

```
perf_eval(
    prediction,
    validation,
    metric = list(auc = metric_fun(curve = "ROC"))
)
```

Arguments

`prediction`	smoothed scores; either a named numeric vector, a column-wise matrix whose rownames are nodes and colnames are different scores, or a named list of such matrices.
`validation`	target scores to which the smoothed scores will be compared to. Must have the same format as the input scores, although the number of rows may vary and only the matching rows will give a performance measure.
`metric`	named list of metrics to apply. Each metric should accept the form `f(actual, predicted)`

Value

A data frame containing the metrics for each comparable pair of output-validation.

Examples

```
# Using a matrix with four set of scores
# called Single, Row, Small_sample, Large_sample
data(graph_toy)
diff <- diffuse(
    graph = graph_toy,
    scores = graph_toy$input_mat,
    method = "raw")
df_perf <- perf_eval(
    prediction = diff,
    validation = graph_toy$input_mat)
df_perf
```

---

perf_wilcox *Compute column-wise statistics in a performance matrix*

Description

Function `perf_wilcox` compares all the columns of a matrix through a `wilcox.test`. The columns are assumed to be performance measures (e.g. AUROC) whereas the rows are instances.

Usage

```
perf_wilcox(
    perf_mat,
    adjust = function(p) stats::p.adjust(p, method = "fdr"),
    ci = 0.95,
    digits_ci = 2,
    digits_p = 3,
    ...
)
```

Arguments

`perf_mat`	Numeric matrix whose columns contain performance metrics of different methods.
`adjust`	Function to adjust the p-values for multiple testing. By default, `p.adjust` with its default parameters is used.
`ci`	Numeric, confidence interval (defaults to `0.95`)
`digits_ci`	Integer, digits to display in the confidence interval
`digits_p`	Integer, digits to display in the p-value
`...`	further arguments for `format`

Details

The statistical comparison of the columns is intended to ease comparisons between methods in a rigorous way. Methods are compared pairwise and a p-value for difference in performance. The function `perf_wilcox` returns a character matrix so that (1) the upper triangular matrix contains confidence intervals on the estimate of the difference between performances, and (2) the lower triangular matrix contains the two-tailed p-value that tests difference in performance, with multiple testing correction. The comparison takes place between row and column in that precise order: a positive difference favours the row and a negative one, the column.

Value

Character matrix. The upper triangular matrix contains a confidence interval and the estimate of the pairwise difference in performance. The lower triangular matrix shows the associated two-tailed p-value, with multiple testing correction.

Examples

```
# Dummy data frame to test
n <- 100
perf_mat <- cbind(
    good = runif(n = n, min = 0.5, max = 1),
    so_so = runif(n = n, min = 0.2, max = 0.7),
    bad = runif(n = n, min = 0, max = 0.5)
)
wilcox_mat <- perf_wilcox(perf_mat)

# See how the methods in the rows compare to those
# in the columns, confidence interval
# (upper) and p-value (lower)
wilcox_mat
```

`scores2colours` *Translate values into colours*

Description

Create a vector of hex colours from numeric values, typically diffusion scores

Usage

```
scores2colours(
    x,
    range = c(min(0, min(x)), max(x)),
    n.colors = 10,
    palette = colorRampPalette(c("#3C5488FF", "white", "#F39B7FFF"))
)
```

Arguments

x	numeric vector to be colorised
range	range of values to filter x (values out of the range will be collapsed to the closest limit)
n.colors	integer, number of colors in the palette
palette	palette function that generates a scale of colours given the number of desired colours. Defaults to a blue-white-red scale by colorRampPalette

Value

Character vector with hex colours

Examples

```
set.seed(1)
scores2colours(runif(20))
```

---

scores2shapes *Translate values into shapes*

Description

Translate 0/1 to shapes, by default "circle" and "square"

Usage

```
scores2shapes(x, shapes = c("circle", "square"))
```

Arguments

x	numeric vector to generate shapes from
shapes	character vector with two shapes, respectively zeroes and ones

Value

Character vector with shapes

Examples

```
set.seed(1)
scores2shapes(rbinom(n = 20, size = 1, prob = .5))
```

serialHeatrank *Compute heatrank for a single case*

Description

The heatrank incorporates the correction `(r + 1)/(p + 1)` instead of `r/p`

Usage

```
serialHeatrank(R, perm, G, ind)
```

Arguments

`R`	dense matrix with the diffusion kernel
`perm`	sparse matrix with the permutations
`G`	sparse matrix with the heat sources
`ind`	index of the G column for current source

Value

an arma::vec with node heatranks

sparsify2 *Sparsify arma::mat into arma::sp_mat*

Description

Return permutations as a numeric sparse matrix (can be binary or continuous)

Usage

```
sparsify2(perm, nrow, G)
```

Arguments

`perm`	dense matrix with the permutations
`nrow`	number of rows for the sparse matrix
`G`	sparse column matrix

Value

an arma::sp_mat object

to_list *Convert input to list format*

Description

Convert any input to list format

Usage

```
to_list(scores, dummy_column = "X1", dummy_list = "X1")
```

Arguments

scores	object to reformat
dummy_column, dummy_list	character, names for the dummy columns/items

Value

scores in list format

Examples

```
data(graph_toy)
x_v <- diffuStats:::to_list(graph_toy$input_vec)
x_m <- diffuStats:::to_list(graph_toy$input_mat)
```

to_x_from_list *Convert list format to desired format*

Description

Convert any list format to the convenient one

Usage

```
to_x_from_list(scores, x)
```

Arguments

scores	list to reformat
x	character, desired format

Value

scores in desired format

Examples

```
data(graph_toy)
x_v <- diffuStats:::to_x_from_list(
    diffuStats:::to_list(graph_toy$input_vec), "vector")
x_m <- diffuStats:::to_x_from_list(
    diffuStats:::to_list(graph_toy$input_vec), "matrix")
```

---

which_format *In which format is the input?*

Description

Tell apart vector, matrix or list of matrices

Usage

```
which_format(x)
```

Arguments

x	object to evaluate

Value

character: vector, matrix or list.

Examples

```
data(graph_toy)
diffuStats:::which_format(graph_toy$input_vec)
diffuStats:::which_format(graph_toy$input_mat)
```

Index

∗ **datasets**
    .check_scores, 3
    .default_graph_param, 5
    graph_toy, 15
.available_methods *(*.check_scores*)*, 3
.check_K *(*.check_scores*)*, 3
.check_graph *(*.check_scores*)*, 3
.check_method *(*.check_scores*)*, 3
.check_metric *(*.check_scores*)*, 3
.check_scores, 3
.connect_undirected_graph, 4
.default_graph_param, 5
.default_prop *(*.default_graph_param*)*, 5

barabasi.game, *14*

colorRampPalette, *28*
commuteTimeKernel *(*kernels*)*, 16
convertSparse, 5

diffuse, 6, *13*
diffuse_grid *(*diffuse*)*, 6
diffuse_mc, 11
diffuse_raw, 12
diffusionKernel *(*kernels*)*, 16
diffuStats, 13

erdos.renyi.game, *14*

format, *26*

generate_graph, 13
generate_input, *14*, 14
get_covar *(*moments*)*, 21
get_mu *(*moments*)*, 21
get_mu_reference *(*moments*)*, 21
get_var_reference *(*moments*)*, 21
graph_toy, 15

inverseCosineKernel *(*kernels*)*, 16
is_kernel, 16

kernels, *13*, 16

largest_cc, 18

make_lattice, *14*
Matrix, *5*
metric_auc, 19, *20*
metric_fun *(*metric_auc*)*, 19
moments, 21

named.list, 23

p.adjust, *26*
ParallelHeatrank, 23
part, *20*
perf, *13*, *19*, 24
perf_eval, *13*, *19*, 25
perf_wilcox, 26
pStepKernel *(*kernels*)*, 16

regularisedLaplacianKernel *(*kernels*)*, 16

sample, *11*
scores2colours, 27
scores2shapes, 28
serialHeatrank, 29
sparsify2, 29

to_list, 30
to_x_from_list, 30

watts.strogatz.game, *14*
which_format, 31
wilcox.test, *26*