Package: tgstat (via r-universe)

October 28, 2026

Type Package

Title Amos Tanay's Group High Performance Statistical Utilities

Version 2.4.0

Author Michael Hoichman [aut], Aviezer Lifshitz [aut, cre]

Maintainer Aviezer Lifshitz <aviezer.lifshitz@weizmann.ac.il>

Description A collection of high performance utilities to compute distance, correlation, auto correlation, clustering and other tasks. Contains graph clustering algorithm described in ``MetaCell: analysis of single-cell RNA-seq data using K-nn graph partitions" (Yael Baran, Akhiad Bercovich, Arnau Sebe-Pedros, Yaniv Lubling, Amir Giladi, Elad Chomsky, Zohar Meir, Michael Hoichman, Aviezer Lifshitz & Amos Tanay, 2019 <doi:10.1186/s13059-019-1812-2>).

License GPL-2

BugReports https://github.com/tanaylab/tgstat/issues

URL https://tanaylab.github.io/tgstat/

Depends R (>= 3.5.0)

Imports utils

Suggests knitr, Matrix, rmarkdown, testthat (>= 3.0.0)

VignetteBuilder knitr

Encoding UTF-8

LazyLoad yes

NeedsCompilation yes

OS_type unix

RoxygenNote 7.3.3

Config/testthat/edition 3

Repository https://cran.r-universe.dev

Date/Publication 2026-08-29 14:34:35 UTC

RemoteUrl https://github.com/cran/tgstat

RemoteRef HEAD

RemoteSha 077710a1aea79c1a279377689f57a7ac1c3e5d83

Contents

tgs_chi2 2
tgs_cor 3
tgs_dist 5
tgs_finite 6
tgs_graph 7
tgs_graph_cover 8
tgs_graph_cover_resample 9
tgs_knn 11
tgs_matrix_tapply 12

Index **14**

`tgs_chi2` *Vectorized chi-squared test for 2x2 contingency tables*

Description

Performs a chi-squared test with optional Yates' continuity correction on each row of a two-column count matrix. For each row, a 2x2 contingency table is constructed using the row counts and the column sums, and the chi-squared statistic and p-value are computed.

Usage

```
tgs_chi2(x, yates = TRUE)
```

Arguments

`x`	a numeric matrix or sparse matrix of `dgCMatrix` type with exactly 2 columns containing non-negative counts.
`yates`	logical; if `TRUE` (default), Yates' continuity correction is applied to the chi-squared statistic.

Details

This function is useful for differential gene expression analysis, where each row represents a gene and the two columns represent UMI counts in two conditions. The test determines whether the gene's proportion differs significantly between conditions.

For each row i, the contingency table is:

	Condition 1	Condition 2
Gene i	x[i,1]	x[i,2]

Other genes colSum1 - x[i,1] colSum2 - x[i,2]

Value

A numeric matrix with `nrow(x)` rows and 2 columns named `"chi2"` and `"pval"`. Row names from `x` are preserved. When the denominator of the chi-squared formula is zero, the statistic is set to 0 and the p-value to 1.

Examples

```
# Note: all the available CPU cores might be used

set.seed(42)
# Simulate UMI counts for 1000 genes in 2 conditions
mat <- matrix(rpois(2000, lambda = 100), ncol = 2)
rownames(mat) <- paste0("gene", 1:1000)
colnames(mat) <- c("condition1", "condition2")
result <- tgs_chi2(mat)
head(result)

# Without Yates' correction
result_no_yates <- tgs_chi2(mat, yates = FALSE)

# With sparse matrix
sparse_mat <- Matrix::Matrix(mat, sparse = TRUE)
result_sparse <- tgs_chi2(sparse_mat)
```

---

tgs_cor *Calculates correlation or auto-correlation*

Description

Calculates correlation between two matrices columns or auto-correlation between a matrix columns.

Usage

```
tgs_cor(
  x,
  y = NULL,
  pairwise.complete.obs = FALSE,
  spearman = FALSE,
  tidy = FALSE,
  threshold = 0
)
```

```
tgs_cor_knn(
  x,
  y,
  knn,
  pairwise.complete.obs = FALSE,
  spearman = FALSE,
  threshold = 0
)
```

Arguments

x	numeric matrix
y	numeric matrix
pairwise.complete.obs	see below
spearman	if 'TRUE' Spearman correlation is computed, otherwise Pearson
tidy	if 'TRUE' data is outputed in tidy format
threshold	absolute threshold above which values are outputed in tidy format
knn	the number of highest correlations returned per column

Details

'tgs_cor' is very similar to 'stats::cor'. Unlike the latter it uses all available CPU cores to compute the correlation in a much faster way. The basic implementation of 'pairwise.complete.obs' is also more efficient giving overall great run-time advantage.

Unlike 'stats::cor' 'tgs_cor' implements only two modes of treating data containing NA, which are equivalent to 'use="everything"' and 'use="pairwise.complete.obs". Please refer the documentation of this function for more details.

'tgs_cor(x, y, spearman = FALSE)' is equivalent to 'cor(x, y, method = "pearson")' 'tgs_cor(x, y, spearman = TRUE)' is equivalent to 'cor(x, y, method = "spearman")' 'tgs_cor(x, y, pairwise.complete.obs = TRUE, spearman = TRUE)' is equivalent to 'cor(x, y, use = "pairwise.complete.obs", method = "spearman")' 'tgs_cor(x, y, pairwise.complete.obs = TRUE, spearman = FALSE)' is equivalent to 'cor(x, y, use = "pairwise.complete.obs", method = "pearson")'

'tgs_cor' can output its result in "tidy" format: a data frame with three columns named 'col1', 'col2' and 'cor'. Only the correlation values which abs are equal or above the 'threshold' are reported. For auto-correlation (i.e. when 'y=NULL') a pair of columns numbered X and Y is reported only if X < Y.

'tgs_cor_knn' works similarly to 'tgs_cor'. Unlike the latter it returns only the highest 'knn' correlations for each column in 'x'. The result of 'tgs_cor_knn' is always outputed in "tidy" format.

One of the reasons to opt 'tgs_cor_knn' over a pair of calls to 'tgs_cor' and 'tgs_knn' is the reduced memory consumption of the former. For auto-correlation case (i.e. 'y=NULL') given that the number of columns NC exceeds the number of rows NR, then 'tgs_cor' memory consumption becomes a factor of NCxNC. In contrast 'tgs_cor_knn' would consume in the similar scenario a factor of max(NCxNR,NCxKNN). Similarly 'tgs_cor(x,y)' would consume memory as a factor of NCXxNCY, wherever 'tgs_cor_knn(x,y,knn)' would reduce that to max((NCX+NCY)xNR,NCXxKNN).

Value

'tgs_cor_knn' or 'tgs_cor' with 'tidy=TRUE' return a data frame, where each row represents correlation between two pairs of columns from 'x' and 'y' (or two columns of 'x' itself if 'y==NULL'). 'tgs_cor' with the 'tidy=FALSE' returns a matrix of correlation values, where `val[X,Y]` represents the correlation between columns X and Y of the input matrices (if 'y' is not 'NULL') or the correlation between columns X and Y of 'x' (if 'y' is 'NULL').

Examples

```
# Note: all the available CPU cores might be used

set.seed(seed = 0)
rows <- 100
cols <- 1000
vals <- sample(1:(rows * cols / 2), rows * cols, replace = TRUE)
m <- matrix(vals, nrow = rows, ncol = cols)
m[sample(1:(rows * cols), rows * cols / 1000)] <- NA

r1 <- tgs_cor(m, spearman = FALSE)
r2 <- tgs_cor(m, pairwise.complete.obs = TRUE, spearman = TRUE)
r3 <- tgs_cor_knn(m, NULL, 5, spearman = FALSE)
```

---

tgs_dist *Calculates distances between the matrix rows*

Description

Calculates distances between the matrix rows.

Usage

```
tgs_dist(x, diag = FALSE, upper = FALSE, tidy = FALSE, threshold = Inf)
```

Arguments

x	numeric matrix
diag	see 'dist' documentation
upper	see 'dist' documentation
tidy	if 'TRUE' data is outputed in tidy format
threshold	threshold below which values are outputed in tidy format

Details

This function is very similar to 'package:stats::dist'. Unlike the latter it uses all available CPU cores to compute the distances in a much faster way.

Unlike 'package:stats::dist' 'tgs_dist' uses always "euclidean" metrics (see 'method' parameter of 'dist' function). Thus:

'tgs_dist(x)' is equivalent to 'dist(x, method = "euclidean")'

'tgs_dist' can output its result in "tidy" format: a data frame with three columns named 'row1', 'row2' and 'dist'. Only the distances that are less or equal than the 'threshold' are reported. Distance between row number X and Y is reported only if X < Y. 'diag' and 'upper' parameters are ignored when the result is returned in "tidy" format.

Value

If 'tidy' is 'FALSE' - the output is similar to that of 'dist' function. If 'tidy' is 'TRUE' - 'tgs_dist' returns a data frame, where each row represents distances between two pairs of original rows.

Examples

```
# Note: all the available CPU cores might be used

set.seed(seed = 0)
rows <- 100
cols <- 1000
vals <- sample(1:(rows * cols / 2), rows * cols, replace = TRUE)
m <- matrix(vals, nrow = rows, ncol = cols)
m[sample(1:(rows * cols), rows * cols / 1000)] <- NA
r <- tgs_dist(m)
```

---

tgs_finite *Checks whether all the elements of the vector are finite*

Description

Checks whether all the elements of the vector are finite.

Usage

```
tgs_finite(x)
```

Arguments

x	numeric or integer vector or matrix

Details

'tgs_finite' returns 'TRUE' if all the elements of 'x' are finite numbers. (See: 'is.finite'.)

Value

'TRUE' if all the elements of 'x' are finite, otherwise 'FALSE'.

Examples

```
tgs_finite(1:10)
tgs_finite(c(1:10, NaN))
tgs_finite(c(1:10, Inf))
```

---

tgs_graph *Builds directed graph of correlations*

Description

Builds directed graph of correlations where the nodes are the matrix columns.

Usage

```
tgs_graph(x, knn, k_expand, k_beta = 3)
```

Arguments

x	see below
knn	maximal node degree
k_expand	see below
k_beta	see below

Details

This function builds a directed graph based on the edges in 'x' and their ranks.

'x' is a data frame containing 4 columns named: 'col1', 'col2', 'val', 'rank'. The third column ('val' can have a different name). The result in the compatible format is returned, for example, by 'tgs_knn' function.

'tgs_graph' prunes some of the edges in 'x' based on the following steps:

1. A pair of columns i, j that appears in 'x' in 'col1', 'col2' implies the edge in the graph from i to j: edge(i,j). Let the rank of i and j be rank(i,j).
2. Calculate symmetrised rank of i and j: sym_rank(i,j) = rank(i,j) * rank(j,i). If one of the ranks is missing from the previous result sym_rank is set to NA.
3. Prune the edges: remove edge(i,j) if sym_rank(i,j) == NA OR sym_rank(i,j) < knn * knn * k_expand

4. Prune excessive incoming edges: remove edge(i,j) if more than knn * k_beta edges of type edge(node,j) exist and sym_rank(i,j) is higher than sym_rank(node,j) for node != j.
5. Prune excessive outgoing edges: remove edge(i,j) if more than knn edges of type edge(i,node) exist and sym_rank(i,j) is higher than sym_rank(i,node) for node != i.

Value

The graph edges are returned in a data frame, with the weight of each edge. edge(i,j) receives weight 1 if its sym_rank is the lowest among all edges of type edge(i,node). Formally defined: weight(i,j) = 1 - (place(i,j) - 1) / knn, where place(i,j) is the location of edge(i,j) within the sorted set of edges outgoing from i, i.e. edge(i,node). The sort is done by sym_rank of the edges.

Examples

```
# Note: all the available CPU cores might be used

set.seed(seed = 1)
rows <- 100
cols <- 1000
vals <- sample(1:(rows * cols / 2), rows * cols, replace = TRUE)
m <- matrix(vals, nrow = rows, ncol = cols)
m[sample(1:(rows * cols), rows * cols / 1000)] <- NA

r1 <- tgs_cor(m, pairwise.complete.obs = FALSE, spearman = TRUE)
r2 <- tgs_knn(r1, knn = 30, diag = FALSE)
r3 <- tgs_graph(r2, knn = 3, k_expand = 10)
```

---

tgs_graph_cover *Clusters directed graph*

Description

Clusters directed graph.

Usage

```
tgs_graph_cover(graph, min_cluster_size, cooling = 1.05, burn_in = 10)
```

Arguments

graph	directed graph in the format returned by tgs_graph
min_cluster_size	used to determine the candidates for seeding (= min weight)
cooling	factor that is used to gradually increase the chance of a node to stay in the cluster
burn_in	number of node reassignments after which cooling is applied

Details

The algorithm is explained in a "MetaCell: analysis of single-cell RNA-seq data using K-nn graph partitions" paper, published in "Genome Biology" #20: https://genomebiology.biomedcentral.com/articles/10.1186/s13059-019-1812-2

Value

Data frame that maps each node to its cluster.

See Also

tgs_graph()

Examples

```
# Note: all the available CPU cores might be used

set.seed(seed = 0)
rows <- 100
cols <- 1000
vals <- sample(1:(rows * cols / 2), rows * cols, replace = TRUE)
m <- matrix(vals, nrow = rows, ncol = cols)
m[sample(1:(rows * cols), rows * cols / 1000)] <- NA

r1 <- tgs_cor(m, pairwise.complete.obs = FALSE, spearman = TRUE)
r2 <- tgs_knn(r1, knn = 30, diag = FALSE)
r3 <- tgs_graph(r2, knn = 3, k_expand = 10)
r4 <- tgs_graph_cover(r3, 5)
```

---

tgs_graph_cover_resample

Clusters directed graph multiple times with randomized sample subset

---

Description

Clusters directed graph multiple times with randomized sample subset.

Usage

```
tgs_graph_cover_resample(
  graph,
  knn,
  min_cluster_size,
  cooling = 1.05,
```

```
  burn_in = 10,
  p_resamp = 0.75,
  n_resamp = 500,
  method = "hash"
)
```

Arguments

graph	directed graph in the format returned by tgs_graph
knn	maximal number of edges used per node for each sample subset
min_cluster_size	used to determine the candidates for seeding (= min weight)
cooling	factor that is used to gradually increase the chance of a node to stay in the cluster
burn_in	number of node reassignments after which cooling is applied
p_resamp	fraction of total number of nodes used in each sample subset
n_resamp	number iterations the clustering is run on different sample subsets
method	method for calculating co_cluster and co_sample; valid values: "hash", "full", "edges"

Details

The algorithm is explained in a "MetaCell: analysis of single-cell RNA-seq data using K-nn graph partitions" paper, published in "Genome Biology" #20: https://genomebiology.biomedcentral.com/articles/10.1186/s13059-019-1812-2

Value

If method == "hash", a list with two members. The first member is a data frame with 3 columns: "node1", "node2" and "cnt". "cnt" indicates the number of times "node1" and "node2" appeared in the same cluster. The second member of the list is a vector of **number of nodes** length reflecting how many times each node was used in the subset.

If method == "full", a list containing two matrices: co_cluster and co_sample.

If method == "edges", a list containing two data frames: co_cluster and co_sample.

See Also

tgs_graph()

Examples

```
# Note: all the available CPU cores might be used

set.seed(seed = 0)
rows <- 100
cols <- 200
vals <- sample(1:(rows * cols / 2), rows * cols, replace = TRUE)
m <- matrix(vals, nrow = rows, ncol = cols)
```

```
r1 <- tgs_cor(m, pairwise.complete.obs = FALSE, spearman = TRUE)
r2 <- tgs_knn(r1, knn = 20, diag = FALSE)
r3 <- tgs_graph(r2, knn = 3, k_expand = 10)
r4 <- tgs_graph_cover_resample(r3, 10, 1)
```

---

tgs_knn *Returns k highest values of each column*

Description

Returns k highest values of each column.

Usage

```
tgs_knn(x, knn, diag = FALSE, threshold = 0)
```

Arguments

x	numeric matrix or data frame (see below)
knn	the number of highest values returned per column
diag	if 'F' values of row 'i' and col 'j' are skipped for each i == j
threshold	filter out values lower than threshold

Details

'tgs_knn' returns the highest 'knn' values of each column of 'x' (if 'x' is a matrix). 'x' can be also a sparse matrix given in a data frame of 'col', 'row', 'value' format.

'NA' and 'Inf' values are skipped as well as the values below 'threshold'. If 'diag' is 'F' values of the diagonal (row == col) are skipped too.

Value

A sparse matrix in a data frame format with 'col1', 'col2', 'val' and 'rank' columns. 'col1' and 'col2' represent the column and the row number of 'x'.

Examples

```
# Note: all the available CPU cores might be used

set.seed(seed = 1)
rows <- 100
cols <- 1000
vals <- sample(1:(rows * cols / 2), rows * cols, replace = TRUE)
m <- matrix(vals, nrow = rows, ncol = cols)
```

```
m[sample(1:(rows * cols), rows * cols / 1000)] <- NA
r <- tgs_knn(m, 3)
```

`tgs_matrix_tapply` *For each matrix row apply a function over a ragged array*

Description

For each matrix row apply a function to each cell of a ragged array, that is to each (non-empty) group of values given by a unique combination of the levels of certain factors.

Usage

```
tgs_matrix_tapply(x, index, fun, ...)
```

Arguments

`x`	a matrix or a sparse matrix of 'dgCMatrix' type
`index`	a 'list' of one or more 'factor's, each of same length as the number of columns in 'x'. The elements are coerced to factors by 'as.factor'.
`fun`	the function to be applied
`...`	optional arguments to 'fun'

Details

'tgs_matrix_tapply(x, index, fun)' is essentialy an efficient implementation of 'apply(mat, 1, function(x) tapply(x, index, fun))'.

Value

A matrix of length(index) X nrow(x) size. Each `[i,j]` element represents the result of applying 'fun' to `x[i,which(index==levels(index)[j])]`.
Note that the return value is a dense matrix even when `x` is sparse.

Examples

```
# Note: all the available CPU cores might be used

set.seed(seed = 1)
nr <- 6
nc <- 10
mat <- matrix(sample(c(rep(0, 6), 1:3), nr * nc, replace = TRUE), nrow = nr, ncol = nc)
index <- factor(rep_len(1:3, ncol(mat)), levels = 0:5)
r1 <- apply(mat, 1, function(x) tapply(x, index, sum))
```

```
r2 <- tgs_matrix_tapply(mat, index, sum)
```

Index

∗ **~apply**
    tgs_matrix_tapply, 12
∗ **~cluster**
    tgs_graph_cover, 8
    tgs_graph_cover_resample, 9
∗ **~correlation**
    tgs_cor, 3
∗ **~distance**
    tgs_dist, 5
∗ **~finite**
    tgs_finite, 6
∗ **~graph**
    tgs_graph, 7
∗ **~knn**
    tgs_knn, 11
∗ **~tapply**
    tgs_matrix_tapply, 12

tgs_chi2, 2
tgs_cor, 3
tgs_cor_knn *(*tgs_cor*)*, 3
tgs_dist, 5
tgs_finite, 6
tgs_graph, 7
tgs_graph(), *9, 10*
tgs_graph_cover, 8
tgs_graph_cover_resample, 9
tgs_knn, 11
tgs_matrix_tapply, 12