Package ‘scrapper’

July 25, 2025

Version 1.3.5

Date 2025-07-13

Title Bindings to C++ Libraries for Single-Cell Analysis

Description Implements R bindings to C++ code for analyzing single-cell (expression) data, mostly from various libscran libraries.
Each function performs an individual step in the single-cell analysis workflow, ranging from quality control to clustering and marker detection.
It is mostly intended for other Bioconductor package developers to build more user-friendly end-to-end workflows.

License MIT + file LICENSE

Imports methods, Rcpp, beachmat (>= 2.25.1), DelayedArray, BiocNeighbors (>= 1.99.0), Rigraphlib, parallel

Suggests testthat, knitr, rmarkdown, BiocStyle, MatrixGenerics, sparseMatrixStats, Matrix, S4Vectors, SummarizedExperiment, SingleCellExperiment, scRNAseq, igraph

LinkingTo Rcpp, assorthead (>= 1.3.5), beachmat, BiocNeighbors

biocViews Normalization, RNASeq, Software, GeneExpression, Transcriptomics, SingleCell, BatchEffect, QualityControl, DifferentialExpression, FeatureExtraction, PrincipalComponent, Clustering

SystemRequirements C++17, GNU make

VignetteBuilder knitr

Encoding UTF-8

RoxygenNote 7.3.2

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

git_branch devel

git_last_commit 3924f5b

git_last_commit_date 2025-07-14

Repository Bioconductor 3.22

Date/Publication 2025-07-24

Author Aaron Lun [cre, aut]

Maintainer Aaron Lun `<infinite.monkeys.with.keyboards@gmail.com>`

Contents

`adt_quality_control` *Quality control for ADT count data*

Description

Compute per-cell QC metrics from an initialized matrix of ADT counts, and use the metrics to suggest filter thresholds to retain high-quality cells.

Usage

```
computeAdtQcMetrics(x, subsets, num.threads = 1)

suggestAdtQcThresholds(
  metrics,
  block = NULL,
  min.detected.drop = 0.1,
```

```
  num.mads = 3
)

filterAdtQcMetrics(thresholds, metrics, block = NULL)
```

Arguments

`x`	A matrix-like object where rows are ADTs and columns are cells. Values are expected to be counts.
`subsets`	List of vectors specifying tag subsets of interest, typically control tags like IgGs. Each vector may be logical (whether to keep each row), integer (row indices) or character (row names).
`num.threads`	Integer scalar specifying the number of threads to use.
`metrics`	List with the same structure as produced by `computeAdtQcMetrics`.
`block`	Factor specifying the block of origin (e.g., batch, sample) for each cell in `metrics`. Alternatively `NULL` if all cells are from the same block. For `filterAdtQcMetrics`, a blocking factor should be provided if `block` was used to construct `thresholds`.
`min.detected.drop`	Minimum drop in the number of detected features from the median, in order to consider a cell to be of low quality.
`num.mads`	Number of median from the median, to define the threshold for outliers in each metric.
`thresholds`	List with the same structure as produced by `suggestAdtQcThresholds`.

Value

For `computeAdtQcMetrics`, a list is returned containing:

- `sum`, a numeric vector containing the total ADT count for each cell.
- `detected`, an integer vector containing the number of detected tags per cell.
- `subsets`, a list of numeric vectors containing the total count of each control subset.

Each vector is of length equal to the number of cells.

For `suggestAdtQcThresholds`, a named list is returned:

- If `block=NULL`, the list contains:
  - `detected`, a numeric scalar containing the lower bound on the number of detected tags.
  - `subsets`, a numeric vector containing the upper bound on the sum of counts in each control subset.
- Otherwise, if `block` is supplied, the list contains:
  - `detected`, a numeric vector containing the lower bound on the number of detected tags for each blocking level.
  - `subsets`, a list of numeric vectors containing the upper bound on the sum of counts in each control subset for each blocking level.

  Each vector is of length equal to the number of levels in `block` and is named accordingly.

For `filterAdtQcMetrics`, a logical vector of length `ncol(x)` is returned indicating which cells are of high quality.

Author(s)

Aaron Lun

See Also

The `compute_adt_qc_metrics`, `compute_adt_qc_filters` and `compute_adt_qc_filters_blocked` functions in `https://libscran.github.io/scran_qc/`, for the rationale of QC filtering on ADT counts.

Examples

```
# Mocking a matrix:
library(Matrix)
x <- round(abs(rsparsematrix(1000, 100, 0.1) * 100))

# Mocking up a control set.
sub <- list(IgG=rbinom(nrow(x), 1, 0.1) > 0)

qc <- computeAdtQcMetrics(x, sub)
str(qc)

filt <- suggestAdtQcThresholds(qc)
str(filt)

keep <- filterAdtQcMetrics(filt, qc)
summary(keep)
```

`aggregateAcrossCells` *Aggregate expression across cells*

Description

Aggregate expression values across cells based on one or more grouping factors. This is primarily used to create pseudo-bulk profiles for each cluster/sample combination.

Usage

```
aggregateAcrossCells(x, factors, num.threads = 1)
```

Arguments

`x`	A matrix-like object where rows correspond to genes or genomic features and columns correspond to cells. Values are typically expected to be counts.
`factors`	A list or data frame containing one or more grouping factors, see `combineFactors`.
`num.threads`	Integer specifying the number of threads to be used for aggregation.

Value

A list containing:

- `sums`, a numeric matrix where each row corresponds to a gene and each column corresponds to a unique combination of grouping levels. Each entry contains the summed expression across all cells with that combination.
- `detected`, an integer matrix where each row corresponds to a gene and each column corresponds to a unique combination of grouping levels. Each entry contains the number of cells with detected expression in that combination.
- `combinations`, a data frame describing the levels for each unique combination of factors. Rows of this data frame correspond to columns of `sums` and `detected`, while columns correspond to the factors in `factors`.
- `counts`, the number of cells associated with each combination. Each entry corresponds to a row of `combinations`.
- `index`, an integer vector of length equal to the number of cells in `x`. This specifies the combination in `combinations` to which each cell was assigned.

Author(s)

Aaron Lun

See Also

The `aggregate_across_cells` function in `https://libscran.github.io/scran_aggregate/`, for the underlying implementation.

`aggregateAcrossGenes`, to aggregate expression values across gene sets.

Examples

```
# Mocking a matrix:
library(Matrix)
x <- round(abs(rsparsematrix(1000, 100, 0.1) * 100))

# Simple aggregation:
clusters <- sample(LETTERS, 100, replace=TRUE)
agg <- aggregateAcrossCells(x, list(cluster=clusters))
str(agg)

# Multi-factor aggregation
samples <- sample(1:5, 100, replace=TRUE)
agg2 <- aggregateAcrossCells(x, list(cluster=clusters, sample=samples))
str(agg2)
```

`aggregateAcrossGenes` *Aggregate expression across genes*

Description

Aggregate expression values across genes, potentially with weights. This is typically used to summarize expression values for gene sets into a single per-cell score.

Usage

```
aggregateAcrossGenes(x, sets, average = FALSE, num.threads = 1)
```

Arguments

`x`	A matrix-like object where rows correspond to genes or genomic features and columns correspond to cells. Values are typically expected to be counts.
`sets`	A list of integer vectors containing the row indices of genes in each set. Alternatively, each entry may be a list of length 2, containing an integer vector (row indices) and a numeric vector (weights).
`average`	Logical scalar indicating whether to compute the average rather than the sum.
`num.threads`	Integer specifying the number of threads to be used for aggregation.

Value

A list of length equal to that of `sets`. Each entry is a numeric vector of length equal to the number of columns in `x`, containing the (weighted) sum/mean of expression values for the corresponding set across all cells.

Author(s)

Aaron Lun

See Also

The `aggregate_across_genes` function in https://libscran.github.io/scran_aggregate/, for the underlying implementation.

`aggregateAcrossCells`, to aggregate expression values across groups of cells.

Examples

```
# Mocking a matrix:
library(Matrix)
x <- round(abs(rsparsematrix(1000, 100, 0.1) * 100))

# Unweighted aggregation:
sets <- list(
   foo = sample(nrow(x), 20),
   bar = sample(nrow(x), 10)
)
agg <- aggregateAcrossGenes(x, sets)
str(agg)

# Weighted aggregation:
sets <- list(
   foo = list(sample(nrow(x), 20), runif(20)),
   bar = list(sample(nrow(x), 10), runif(10))
)
agg2 <- aggregateAcrossGenes(x, sets, average = TRUE)
str(agg2)
```

`analyze` *Analyze single-cell data*

Description

Execute a simple single-cell analysis pipeline, starting from a count matrix and ending with clusters, visualizations and markers. This also supports integration of multiple modalities and correction of batch effects.

Usage

```
analyze(
  rna.x,
  adt.x = NULL,
  crispr.x = NULL,
  block = NULL,
  rna.subsets = list(),
  adt.subsets = list(),
  suggestRnaQcThresholds.args = list(),
  suggestAdtQcThresholds.args = list(),
  suggestCrisprQcThresholds.args = list(),
  filter.cells = TRUE,
  centerSizeFactors.args = list(),
  computeClrm1Factors.args = list(),
  normalizeCounts.args = list(),
  modelGeneVariances.args = list(),
  chooseHighlyVariableGenes.args = list(),
  runPca.args = list(),
  use.rna.pcs = TRUE,
  use.adt.pcs = TRUE,
  use.crispr.pcs = TRUE,
  scaleByNeighbors.args = list(),
  correctMnn.args = list(),
  runUmap.args = list(),
  runTsne.args = list(),
  buildSnnGraph.args = list(),
  clusterGraph.args = list(),
  runAllNeighborSteps.args = list(),
  kmeans.clusters = NULL,
  clusterKmeans.args = list(),
  clusters.for.markers = c("graph", "kmeans"),
  scoreMarkers.args = list(),
  BNPARAM = AnnoyParam(),
  rna.assay = 1L,
  adt.assay = 1L,
  crispr.assay = 1L,
  num.threads = 3L
)
```

Arguments

`rna.x`	Matrix-like object containing RNA counts. This should have the same number of columns as the other `*.x` arguments. Alternatively, a SummarizedExperiment instance containing such a matrix in its `rna.assay`. Alternatively `NULL`, if no RNA counts are available.
`adt.x`	Matrix-like object containing ADT counts. This should have the same number of columns as the other `*.x` arguments. Alternatively, a SummarizedExperiment instance containing such a matrix in its `adt.assay`. Alternatively `NULL`, if no ADT counts are available.
`crispr.x`	Matrix-like object containing ADT counts. This should have the same number of columns as the other `*.x` arguments. Alternatively, a SummarizedExperiment instance containing such a matrix in its `crispr.assay`. Alternatively `NULL`, if no ADT counts are available.
`block`	Factor specifying the block of origin (e.g., batch, sample) for each cell in the `*_x` matrices. Alternatively `NULL`, if all cells are from the same block.
`rna.subsets`	Gene subsets for quality control, typically used for mitochondrial genes. See the `subsets` arguments in `computeRnaQcMetrics` for details.
`adt.subsets`	ADT subsets for quality control, typically used for IgG controls. See the `subsets` arguments in `computeAdtQcMetrics` for details.
`suggestRnaQcThresholds.args`	Named list of arguments to pass to `suggestRnaQcThresholds`.
`suggestAdtQcThresholds.args`	Named list of arguments to pass to `suggestAdtQcThresholds`.
`suggestCrisprQcThresholds.args`	Named list of arguments to pass to `suggestCrisprQcThresholds`.
`filter.cells`	Logical scalar indicating whether to filter the count matrices to only retain high-quality cells in all modalities. If `FALSE`, QC metrics and thresholds are still computed but are not used to filter the count matrices.
`centerSizeFactors.args`	Named list of arguments to pass to `centerSizeFactors`.
`computeClrm1Factors.args`	Named list of arguments to pass to `computeClrm1Factors`. Only used if `adt.x` is provided.
`normalizeCounts.args`	Named list of arguments to pass to `normalizeCounts`.
`modelGeneVariances.args`	Named list of arguments to pass to `modelGeneVariances`. Only used if `rna.x` is provided.
`chooseHighlyVariableGenes.args`	Named list of arguments to pass to `chooseHighlyVariableGenes`. Only used if `rna.x` is provided.
`runPca.args`	Named list of arguments to pass to `runPca`.
`use.rna.pcs`	Logical scalar indicating whether to use the RNA-derived PCs for downstream steps (i.e., clustering, visualization). Only used if `rna.x` is provided.

`use.adt.pcs`	Logical scalar indicating whether to use the ADT-derived PCs for downstream steps (i.e., clustering, visualization). Only used if `adt.x` is provided.
`use.crispr.pcs`	Logical scalar indicating whether to use the CRISPR-derived PCs for downstream steps (i.e., clustering, visualization). Only used if `crispr.x` is provided.
`scaleByNeighbors.args`	Named list of arguments to pass to `scaleByNeighbors`. Only used if multiple modalities are available and their corresponding `use.*.pcs` arguments are `TRUE`.
`correctMnn.args`	Named list of arguments to pass to `correctMnn`. Only used if `block` is supplied.
`runUmap.args`	Named list of arguments to pass to `runUmap`. If `NULL`, UMAP is not performed.
`runTsne.args`	Named list of arguments to pass to `runTsne`. If `NULL`, t-SNE is not performed.
`buildSnnGraph.args`	Named list of arguments to pass to `buildSnnGraph`. Ignored if `clusterGraph.args = NULL`.
`clusterGraph.args`	Named list of arguments to pass to `clusterGraph`. If `NULL`, graph-based clustering is not performed.
`runAllNeighborSteps.args`	Named list of arguments to pass to `runAllNeighborSteps`.
`kmeans.clusters`	Integer scalar specifying the number of clusters to use in k-means clustering. If `NULL`, k-means clustering is not performed.
`clusterKmeans.args`	Named list of arguments to pass to `clusterKmeans`. Ignored if `kmeans.clusters = NULL`.
`clusters.for.markers`	Character vector of clustering algorithms (either `"graph"` or `"kmeans"`, specifying the clustering to be used for marker detection. The first available clustering will be chosen.
`scoreMarkers.args`	Named list of arguments to pass to `scoreMarkers`. Ignored if no suitable clusterings are available.
`BNPARAM`	A BiocNeighborParam instance specifying the nearest-neighbor search algorithm to use.
`rna.assay`	Integer scalar or string specifying the assay to use if `rna.x` is a SummarizedExperiment.
`adt.assay`	Integer scalar or string specifying the assay to use if `adt.x` is a SummarizedExperiment.
`crispr.assay`	Integer scalar or string specifying the assay to use if `crispr.x` is a SummarizedExperiment.
`num.threads`	Integer scalar specifying the number of threads to use in each step.

Value

List containing the results of the entire analysis:

`rna.qc.metrics`**:** Results of `computeRnaQcMetrics`. If RNA data is not available, this is set to `NULL` instead.

`rna.qc.thresholds`: Results of `suggestRnaQcThresholds`. If RNA data is not available, this is set to `NULL` instead.

`rna.qc.filter`: Results of `filterRnaQcMetrics`. If RNA data is not available, this is set to `NULL` instead.

`adt.qc.metrics`: Results of `computeAdtQcMetrics`. If ADT data is not available, this is set to `NULL` instead.

`adt.qc.thresholds`: Results of `suggestAdtQcThresholds`. If ADT data is not available, this is set to `NULL` instead.

`adt.qc.filter`: Results of `filterAdtQcMetrics`. If ADT data is not available, this is set to `NULL` instead.

`crispr.qc.metrics`: Results of `computeCrisprQcMetrics`. If CRISPR data is not available, this is set to `NULL` instead.

`crispr.qc.thresholds`: Results of `suggestCrisprQcThresholds`. If CRISPR data is not available, this is set to `NULL` instead.

`crispr.qc.filter`: Results of `filterCrisprQcMetrics`. If CRISPR data is not available, this is set to `NULL` instead.

`combined.qc.filter`: Logical vector indicating which cells are of high quality and should be retained for downstream analyses.

`rna.filtered`: Matrix of RNA counts that has been filtered to only contain the high-quality cells in `combined.qc.filter`. If RNA data is not available, this is set to `NULL` instead.

`adt.filtered`: Matrix of ADT counts that has been filtered to only contain the high-quality cells in `combined.qc.filter`. If ADT data is not available, this is set to `NULL` instead.

`crispr.filtered`: Matrix of CRISPR counts that has been filtered to only contain the high-quality cells in `combined.qc.filter`. If CRISPR data is not available, this is set to `NULL` instead.

`rna.size.factors`: Size factors for the RNA count matrix, derived from the sum of counts for each cell and centered with `centerSizeFactors`. If RNA data is not available, this is set to `NULL` instead.

`rna.normalized`: Matrix of (log-)normalized expression values derived from RNA counts, as computed by `normalizeCounts` using `rna.size.factors`. If RNA data is not available, this is set to `NULL` instead.

`adt.size.factors`: Size factors for the ADT count matrix, computed by `computeClrm1Factors` and centered with `centerSizeFactors`. If ADT data is not available, this is set to `NULL` instead.

`adt.normalized`: Matrix of (log-)normalized expression values derived from ADT counts, as computed by `normalizeCounts` using `adt.size.factors`. If ADT data is not available, this is set to `NULL` instead.

`crispr.size.factors`: Size factors for the CRISPR count matrix, derived from the sum of counts for each cell and centered with `centerSizeFactors`. If CRISPR data is not available, this is set to `NULL` instead.

`crispr.normalized`: Matrix of (log-)normalized expression values derived from CRISPR counts, as computed by `normalizeCounts` using `crispr.size.factors`. If CRISPR data is not available, this is set to `NULL` instead.

`rna.gene.variances`: Results of `modelGeneVariances`. If RNA data is not available, this is set to `NULL` instead.

`rna.highly.variable.genes`: Results of `chooseHighlyVariableGenes`. If RNA data is not available, this is set to `NULL` instead.

`rna.pca`**:** Results of calling `runPca` on `rna.normalized` with the `rna.highly.variable.genes` subset. If RNA data is not available, this is set to `NULL` instead.

`adt.pca`**:** Results of calling `runPca` on `adt.normalized`. If ADT data is not available, this is set to `NULL` instead.

`crispr.pca`**:** Results of calling `runPca` on `crispr.normalized`. If CRISPR data is not available, this is set to `NULL` instead.

`combined.pca`**:** If only one modality is used for the downstream analysis, this is a string specifying the list element containing the components to be used, e.g., `"rna.pca"`. If multiple modalities are to be combined for downstream analysis, this contains the results of `scaleByNeighbors` on the PCs of those modalities.

`block`**:** Vector or factor containing the blocking factor for all cells (after filtering, if `filter.cells = TRUE`). This is set to `NULL` if no blocking factor was supplied.

`mnn.corrected`**:** Results of `correctMnn` on the PCs in or referenced by `combined.pca`. If no blocking factor is supplied, this is set to “None“ instead.

`tsne`**:** Results of `runTsne`. This is `NULL` if t-SNE was not performed.

`umap`**:** Results of `runUmap`. This is `NULL` if UMAP was not performed.

`snn.graph`**:** Results of `buildSnnGraph`. This is `NULL` if graph-based clustering was not performed, or if `return.graph=FALSE` in `runAllNeighborSteps`.

`graph.clusters`**:** Results of `clusterGraph`. This is `NULL` if graph-based clustering was not performed.

`kmeans.clusters`**:** Results of `clusterKmeans`. This is `NULL` if k-means clustering was not performed.

`clusters`**:** Integer vector containing the cluster assignment for each cell (after filtering, if `filter.cells = TRUE`). This may be derived from `graph.clusters` or `kmeans.clusters` depending on the choice of `clusters.for.markers`. If no suitable clusterings are available, this is set to NULL.

`rna.markers`**:** Results of calling `scoreMarkers` on `rna.normalized`. This is `NULL` if RNA data is not available or no suitable clusterings are available.

`adt.markers`**:** Results of calling `scoreMarkers` on `adt.normalized`. This is `NULL` if ADT data is not available or no suitable clusterings are available.

`crispr.markers`**:** Results of calling `scoreMarkers` on `crispr.normalized`. This is `NULL` if CRISPR data is not available or no suitable clusterings are available.

Author(s)

Aaron Lun

See Also

C++ libraries in `https://github.com/libscran`, which implement all of these steps.

`convertAnalyzeResults`, to convert the results into a SingleCellExperiment.

Examples

```
library(scRNAseq)
sce <- fetchDataset("zeisel-brain-2015", "2023-12-14", realize.assays=TRUE)
sce <- sce[,1:500] # smaller dataset for a faster runtime for R CMD check.
res <- analyze(
    sce,
```

```
    rna.subsets=list(mito=grep("^mt-", rownames(sce))),
    num.threads=2 # keep R CMD check happy
)
str(res)
convertAnalyzeResults(res)
```

---

buildSnnGraph *Build a shared nearest neighbor graph*

Description

Build a shared nearest neighbor (SNN) graph where each node is a cell. Edges are formed between cells that share one or more nearest neighbors, weighted by the number or importance of those shared neighbors.

Usage

```
buildSnnGraph(
  x,
  num.neighbors = 10,
  weight.scheme = "ranked",
  num.threads = 1,
  BNPARAM = AnnoyParam(),
  as.pointer = FALSE
)
```

Arguments

`x`	For `buildSnnGraph`, a numeric matrix where rows are dimensions and columns are cells, typically containing a low-dimensional representation from, e.g., `runPca`. Alternatively, a named list of nearest-neighbor search results. This should contain `index`, an integer matrix where rows are neighbors and columns are cells. Each column contains 1-based indices for the nearest neighbors of the corresponding cell, ordered by increasing distance. The number of neighbors for each cell should be equal to `num.neighbors`, otherwise a warning is raised. Alternatively, an index constructed by `buildIndex`.
`num.neighbors`	Integer scalar specifying the number of neighbors to use to construct the graph.
`weight.scheme`	String specifying the weighting scheme to use for constructing the SNN graph. This can be `"ranked"` (default), `"jaccard"` or `"number"`.
`num.threads`	Integer scalar specifying the number of threads to use. Only used if `x` is not a list of existing nearest-neighbor search results.
`BNPARAM`	A BiocNeighborParam object specifying the algorithm to use. Only used if `x` is not a list of existing nearest-neighbor search results.
`as.pointer`	Logical scalar indicating whether to return an external pointer for direct use in `clusterGraph`. This avoids the extra memory usage caused by conversion to/from an R list.

Value

If `as.pointer=FALSE`, a list is returned containing:

- `vertices`, an integer scalar specifying the number of vertices in the graph (i.e., cells in `x`).
- `edges`, an integer vector of 1-based indices for graph edges. Pairs of values represent the endpoints of an (undirected) edge, i.e., `edges[1:2]` form the first edge, `edges[3:4]` form the second edge and so on.
- `weights`, a numeric vector of weights for each edge. This has length equal to half the length of `edges`.

If `as.pointer=TRUE`, an external pointer to the graph is returned that can be directly used in `clusterGraph`.

Author(s)

Aaron Lun

See Also

The `build_snn_graph` function in `https://libscran.github.io/scran_graph_cluster/`, for details on the weighting scheme.

`clusterGraph`, to define clusters (i.e., communities) from the graph.

Examples

```
data <- matrix(rnorm(10000), ncol=1000)
out <- buildSnnGraph(data)
str(out)

# We can use this to make an igraph::graph.
g <- igraph::make_undirected_graph(out$edges, n = out$vertices)
igraph::E(g)$weight <- out$weight
```

`centerSizeFactors` *Center size factors*

Description

Scale the size factors so they are centered at unity, which ensures that the scale of the counts is preserved (on average) after normalization.

Usage

```
centerSizeFactors(size.factors, block = NULL, mode = c("lowest", "per-block"))
```

Arguments

`size.factors`	Numeric vector of size factors across cells.
`block`	Vector or factor of length equal to `size.factors`, specifying the block of origin for each cell. Alternatively `NULL`, in which case all cells are assumed to be in the same block.
`mode`	String specifying how to scale size factors across blocks. `"lowest"` will compute the average size factor in each block, identify the lowest average across all blocks, and then scale all size factors by that value. `"per-block"` will compute the average size factor in each block, and then scale each size factor by the average of block to which it belongs. Only used if `block` is provided.

Value

Numeric vector of length equal to `size.factors`, containing the centered size factors.

Author(s)

Aaron Lun

See Also

The `center_size_factors` and `center_size_factors_blocked` functions in `https://libscran.github.io/scran_norm/`, for the rationale behind centering the size factors.

Examples

```
centerSizeFactors(runif(100))

centerSizeFactors(runif(100), block=sample(3, 100, replace=TRUE))
```

`chooseHighlyVariableGenes` *Choose highly variable genes*

Description

Choose highly variable genes (HVGs) based on a variance-related statistic.

Usage

```
chooseHighlyVariableGenes(
  stats,
  top = 4000,
  larger = TRUE,
  keep.ties = TRUE,
  bound = NULL
)
```

Arguments

`stats`	Numeric vector of variances (or a related statistic) across all genes. Typically the residuals from `modelGeneVariances` are used here.
`top`	Integer specifying the number of top genes to retain. Note that the actual number of retained genes may not be equal to `top`, depending on the other options.
`larger`	Logical scalar indicating whether larger values of `stats` correspond to more variable genes. If `TRUE`, HVGs are defined as those with the largest values of `stats`.
`keep.ties`	Logical scalar indicating whether to keep tied values of `stats`, even if `top` may be exceeded.
`bound`	Numeric scalar specifying the lower bound (if `larger=TRUE`) or upper bound (otherwise) to be applied to `stats`. Genes are not considered to be HVGs if they do not pass this bound, even if they are within the `top` genes. Ignored if `NULL`.

Value

Integer vector containing the indices of genes in `stats` that are considered to be highly variable.

Author(s)

Aaron Lun

See Also

The `choose_highly_variable_genes` function in `https://libscran.github.io/scran_variances/`, for the underlying implementation.

Examples

```
resids <- rexp(10000)
str(chooseHighlyVariableGenes(resids))
```

---

choosePseudoCount *Choose a suitable pseudo-count*

Description

Choose a suitable pseudo-count to control the bias introduced by log-transformation of normalized counts.

Usage

```
choosePseudoCount(size.factors, quantile = 0.05, max.bias = 1, min.value = 1)
```

Arguments

`size.factors`	Numeric vector of size factors for all cells.
`quantile`	Numeric scalar specifying the quantile to use for defining extreme size factors.
`max.bias`	Numeric scalar specifying the maximum allowed bias.
`min.value`	Numeric scalar specifying the minimum value for the pseudo-count.

Value

A choice of pseudo-count for `normalizeCounts`.

Author(s)

Aaron Lun

See Also

The `choose_pseudo_count` function in `https://libscran.github.io/scran_norm/`, for the motivation behind calculating a larger pseudo-count.

Examples

```
sf <- runif(100)
choosePseudoCount(sf)
choosePseudoCount(sf, quantile=0.01)
choosePseudoCount(sf, max.bias=0.5)
```

`clusterGraph` *Graph-based clustering of cells*

Description

Identify clusters of cells using a variety of community detection methods from a graph where similar cells are connected.

Usage

```
clusterGraph(
  x,
  method = c("multilevel", "leiden", "walktrap"),
  multilevel.resolution = 1,
  leiden.resolution = 1,
  leiden.objective = c("modularity", "cpm"),
  walktrap.steps = 4,
  seed = 42
)
```

Arguments

`x`	List containing graph information or an external pointer to a graph, as returned by `buildSnnGraph`. Alternatively, an igraph object with edge weights.
`method`	String specifying the algorithm to use.
`multilevel.resolution`	Numeric scalar specifying the resolution when `method="multilevel"`.
`leiden.resolution`	Numeric scalar specifying the resolution when `method="leiden"`.
`leiden.objective`	String specifying the objective function when `method="leiden"`.

`walktrap.steps`	Integer scalar specifying the number of steps to use when `method="walktrap"`.
`seed`	Integer scalar specifying the random seed to use for `method="multilevel"` or `"leiden"`.

Value

A list containing `membership`, a factor containing the cluster assignment for each cell; and `status`, an integer scalar indicating whether the algorithm completed successfully (0) or not (non-zero). Additional fields may be present depending on the `method`:

- For `method="multilevel"`, the `levels` list contains the clustering result at each level of the algorithm. A `modularity` numeric vector also contains the modularity at each level, the highest of which corresponds to the reported `membership`.
- For `method="leiden"`, a `quality` numeric scalar containg the quality of the partitioning.
- For `method="walktrap"`, a `merges` matrix specifies the pair of cells or clusters that were merged at each step of the algorithm. A `modularity` numeric scalar also contains the modularity of the final partitioning.

Author(s)

Aaron Lun

See Also

`https://igraph.org/c/html/latest/igraph-Community.html`, for the underlying implementation of each clustering method.

The various `cluster_*` functions in `https://libscran.github.io/scran_graph_cluster/`, for wrappers around the **igraph** code.

Examples

```
data <- matrix(rnorm(10000), ncol=1000)
gout <- buildSnnGraph(data)
str(gout)

str(clusterGraph(gout))
str(clusterGraph(gout, method="leiden"))
str(clusterGraph(gout, method="walktrap"))
```

`clusterKmeans` *K-means clustering*

Description

Perform k-means clustering with a variety of different initialization and refinement algorithms.

Usage

```
clusterKmeans(
  x,
  k,
  init.method = c("var-part", "kmeans++", "random"),
  refine.method = c("hartigan-wong", "lloyd"),
  var.part.optimize.partition = TRUE,
  var.part.size.adjustment = 1,
  lloyd.iterations = 100,
  hartigan.wong.iterations = 10,
  hartigan.wong.quick.transfer.iterations = 50,
  hartigan.wong.quit.quick.transfer.failure = FALSE,
  seed = 5489L,
  num.threads = 1
)
```

Arguments

`x`	Numeric matrix where rows are dimensions and columns are cells.
`k`	Integer scalar specifying the number of clusters.
`init.method`	String specifying the initialization method: variance partitioning (`"var-part"`), kmeans++ (`"kmeans++"`) or random initialization (`"random"`).
`refine.method`	String specifying the refinement method: Lloyd's algorithm (`"lloyd"`) or the Hartigan-Wong algorithm (`"hartigan-wong"`).
`var.part.optimize.partition`	Logical scalar indicating whether each partition boundary should be optimized to reduce the sum of squares in the child partitions. Only used if `init.method = "var.part"`.
`var.part.size.adjustment`	Numeric scalar between 0 and 1, specifying the adjustment to the cluster size when prioritizing the next cluster to partition. Setting this to 0 will ignore the cluster size while setting this to 1 will generally favor larger clusters. Only used if `init.method = "var.part"`.
`lloyd.iterations`	Integer scalar specifying the maximmum number of iterations for the Lloyd algorithm.
`hartigan.wong.iterations`	Integer scalar specifying the maximmum number of iterations for the Hartigan-Wong algorithm.
`hartigan.wong.quick.transfer.iterations`	Integer scalar specifying the maximmum number of quick transfer iterations for the Hartigan-Wong algorithm.
`hartigan.wong.quit.quick.transfer.failure`	Logical scalar indicating whether to quit the Hartigan-Wong algorithm upon convergence failure during quick transfer iterations.
`seed`	Integer scalar specifying the seed to use for random or kmeans++ initialization.
`num.threads`	Integer scalar specifying the number of threads to use.

Value

By default, a list is returned containing:

- `clusters`, a factor containing the cluster assignments for each cell.
- `centers`, a numeric matrix with the coordinates of the cluster centroids (dimensions in rows, centers in columns).
- `iterations`, an integer scalar specifying the number of refinement iterations that were performed.
- `status`, an integer scalar specifying the convergence status. Any non-zero value indicates a convergence failure though the exact meaning depends on the choice of `refine.method`.

Author(s)

Aaron Lun

See Also

`https://ltla.github.io/CppKmeans/`, which describes the various initialization and refinement algorithms in more detail.

Examples

```
x <- t(as.matrix(iris[,1:4]))
clustering <- clusterKmeans(x, k=3)
table(clustering$clusters, iris[,"Species"])
```

`combineFactors` *Combine multiple factors*

Description

Combine multiple categorical factors based on the unique combinations of levels from each factor.

Usage

```
combineFactors(factors, keep.unused = FALSE)
```

Arguments

`factors`	List of vectors or factors of the same length. Corresponding elements across all vectors/factors represent the combination of levels for a single observation. For factors, any existing levels are respected. For other vectors, the sorted and unique values are used as levels. Alternatively, a data frame where each column is a vector or factor and each row corresponds to an observation.
`keep.unused`	Logical scalar indicating whether to report unused combinations of levels.

Value

List containing `levels`, a data frame containing the sorted and unique combinations of levels from `factors`; and `index`, an integer vector specifying the index into `levels` for each observation.

Author(s)

Aaron Lun

See Also

The `combine_factors` function in https://libscran.github.io/scran_aggregate/, which provides the underlying implementation.

Examples

```
combineFactors(list(
    sample(LETTERS[1:5], 100, replace=TRUE),
    sample(3, 100, replace=TRUE)
))

combineFactors(list(
    factor(sample(LETTERS[1:5], 10, replace=TRUE), LETTERS[1:5]),
    factor(sample(5, 10, replace=TRUE), 1:5)
), keep.unused=TRUE)
```

`computeClrm1Factors` *Compute size factors for ADT counts*

Description

Compute size factors from an ADT count matrix using the CLRm1 method.

Usage

```
computeClrm1Factors(x, num.threads = 1)
```

Arguments

`x`	A matrix-like object containing ADT count data. Rows correspond to tags and columns correspond to cells.
`num.threads`	Number of threads to use.

Value

Numeric vector containing the CLRm1 size factor for each cell. Note that these size factors are not centered and should be passed through, e.g., `centerSizeFactors` before normalization.

Author(s)

Aaron Lun

See Also

https://github.com/libscran/clrm1, for a description of the CLRm1 method.

Examples

```
library(Matrix)
x <- abs(rsparsematrix(1000, 100, 0.1) * 10)
head(computeClrm1Factors(x))
```

`convertAnalyzeResults` *Convert analysis results into a SingleCellExperiment*

Description

Convert results from `analyze` into a SingleCellExperiment for further analysis with Bioconductor packages.

Usage

```
convertAnalyzeResults(
  results,
  main.modality = NULL,
  flatten.qc.subsets = TRUE,
  include.per.block.variances = FALSE
)
```

Arguments

`results`	List of results produced by `analyze`.
`main.modality`	String specifying the modality to use as the main experiment of a SingleCellExperiment.
`flatten.qc.subsets`	Logical scalar indicating whether QC metrics for subsets should be flattened in the column data. If `FALSE`, subset metrics are reported as a nested DataFrame.
`include.per.block.variances`	Logical scalar indicating whether the per-block variances should be reported as a nested DataFrame in the row data.

Value

A SingleCellExperiment containing most of the analysis results. Filtered and normalized matrices are stored in the assays. QC metrics, size factors and clusterings are stored in the column data. Gene variances are stored in the row data. PCA, t-SNE and UMAP results are stored in the reduced dimensions. Further modalities are stored as alternative experiments.

Author(s)

Aaron Lun

See Also

`analyze`, to generate `results`.

`correctMnn` *Batch correction with mutual nearest neighbors*

Description

Apply mutual nearest neighbor (MNN) correction to remove batch effects from a low-dimensional matrix.

Usage

```
correctMnn(
  x,
  block,
  num.neighbors = 15,
  num.steps = 1,
  merge.policy = c("rss", "size", "variance", "input"),
  num.mads = NULL,
  robust.iterations = NULL,
  robust.trim = NULL,
  mass.cap = NULL,
  order = NULL,
  reference.policy = NULL,
  BNPARAM = AnnoyParam(),
  num.threads = 1
)
```

Arguments

`x`	Numeric matrix where rows are dimensions and columns are cells, typically containing low-dimensional coordinates (e.g., from `runPca`).
`block`	Factor specifying the block of origin (e.g., batch, sample) for each cell in `x`.
`num.neighbors`	Integer scalar specifying the number of neighbors to use when identifying MNN pairs.
`num.steps`	Integer scalar specifying the number of steps for the recursive neighbor search to compute the center of mass.
`merge.policy`	String specifying the policy to use to choose the order of batches to merge. This can be based on the size of the batch (`"size"`), the variance within each batch (`"variance"`), the residual sum of squares of each batch (`"rss"`), or the input order (`"input"`).
`num.mads`	Deprecated and ignored.
`robust.iterations`	Deprecated and ignored.
`robust.trim`	Deprecated and ignored.
`mass.cap`	Deprecated and ignored.
`order`	Deprecated and ignored, the merge order is now always automatically determined.
`reference.policy`	Deprecated, use `merge.policy` instead.
`BNPARAM`	A BiocNeighborParam object specifying the nearest-neighbor algorithm to use.
`num.threads`	Integer scalar specifying the number of threads to use.

Value

List containing `corrected`, a numeric matrix of the same dimensions as `x`, containing the corrected values.

Author(s)

Aaron Lun

See Also

`https://libscran.github.io/mnncorrect/`, for more details on the underlying implementation.

Examples

```
# Mocking up a dataset with multiple batches.
x <- matrix(rnorm(10000), nrow=10)
b <- sample(3, ncol(x), replace=TRUE)
x[,b==2] <- x[,b==2] + 3
x[,b==3] <- x[,b==3] + 5
lapply(split(colMeans(x), b), mean) # indeed the means differ...

corrected <- correctMnn(x, b)
str(corrected)
lapply(split(colMeans(corrected$corrected), b), mean) # now merged.
```

`crispr_quality_control` *Quality control for CRISPR count data*

Description

Compute per-cell QC metrics from an initialized matrix of CRISPR counts, and use the metrics to suggest filter thresholds to retain high-quality cells.

Usage

```
computeCrisprQcMetrics(x, num.threads = 1)

suggestCrisprQcThresholds(metrics, block = NULL, num.mads = 3)

filterCrisprQcMetrics(thresholds, metrics, block = NULL)
```

Arguments

`x`	A matrix-like object where rows are CRISPRs and columns are cells. Values are expected to be counts.
`num.threads`	Integer scalar specifying the number of threads to use.
`metrics`	List with the same structure as produced by `computeCrisprQcMetrics`.

`block`	Factor specifying the block of origin (e.g., batch, sample) for each cell in `metrics`. Alternatively `NULL` if all cells are from the same block. For `filterCrisprQcMetrics`, a blocking factor should be provided if `block` was used to construct `thresholds`.
`num.mads`	Number of median from the median, to define the threshold for outliers in each metric.
`thresholds`	List with the same structure as produced by `suggestCrisprQcThresholds`.

Value

For `computeCrisprQcMetrics`, a list is returned containing:

- `sum`, a numeric vector containing the total CRISPR count for each cell.
- `detected`, an integer vector containing the number of detected guides per cell.
- `max.value`, a numeric vector containing the count for the most abundant guide in cell.
- `max.index`, an integer vector containing the row index of the most abundant guide in cell.

Each vector is of length equal to the number of cells.

For `suggestCrisprQcThresholds`, a named list is returned.

- If `block=NULL`, the list contains:
    - `max.value`, a numeric scalar containing the lower bound on the maximum counts for each blocking level.
- Otherwise, if `block` is supplied, the list contains:
    - `max.value`, a numeric vector containing the lower bound on the maximum counts for each blocking level.

  Each vector is of length equal to the number of levels in `block` and is named accordingly.

For `filterCrisprQcMetrics`, a logical vector of length `ncol(x)` is returned indicating which cells are of high quality.

Author(s)

Aaron Lun

See Also

The `compute_crispr_qc_metrics`, `compute_crispr_qc_filters` and `compute_crispr_qc_filters_blocked` functions in `https://libscran.github.io/scran_qc/`, for the rationale of QC filtering on CRISPR counts.

Examples

```
# Mocking a matrix:
library(Matrix)
x <- round(abs(rsparsematrix(100, 100, 0.1) * 100))

qc <- computeCrisprQcMetrics(x)
str(qc)

filt <- suggestCrisprQcThresholds(qc)
str(filt)
```

```
keep <- filterCrisprQcMetrics(filt, qc)
summary(keep)
```

`fitVarianceTrend` *Fit a mean-variance trend*

Description

Fit a trend to the per-cell variances with respect to the mean.

Usage

```
fitVarianceTrend(
  means,
  variances,
  mean.filter = TRUE,
  min.mean = 0.1,
  transform = TRUE,
  span = 0.3,
  use.min.width = FALSE,
  min.width = 1,
  min.window.count = 200,
  num.threads = 1
)
```

Arguments

`means`	Numeric vector containing the mean (log-)expression for each gene.
`variances`	Numeric vector containing the variance in the (log-)expression for each gene.
`mean.filter`	Logical scalar indicating whether to filter on the means before trend fitting.
`min.mean`	Numeric scalar specifying the minimum mean of genes to use in trend fitting. Only used if `mean.filter=TRUE`.
`transform`	Logical scalar indicating whether a quarter-root transformation should be applied before trend fitting.
`span`	Numeric scalar specifying the span of the LOWESS smoother. Ignored if `use.min.width=TRUE`.
`use.min.width`	Logical scalar indicating whether a minimum width constraint should be applied to the LOWESS smoother. Useful to avoid overfitting in high-density intervals.
`min.width`	Minimum width of the window to use when `use.min.width=TRUE`.
`min.window.count`	Minimum number of observations in each window. Only used if `use.min.width=TRUE`.
`num.threads`	Number of threads to use.

Value

List containing `fitted`, the fitted values of the trend for each gene; and `residuals`, the residuals from the trend.

Author(s)

Aaron Lun

See Also

The `fit_variance_trend` function in `https://libscran.github.io/scran_variances/`, for the underlying implementation.

Examples

```
x <- runif(1000)
y <- 2^rnorm(1000)
out <- fitVarianceTrend(x, y)

plot(x, y)
o <- order(x)
lines(x[o], out$fitted[o], col="red")
```

`modelGeneVariances` *Model per-gene variances in expression*

Description

Compute the variance in (log-)expression values for each gene, and model the trend in the variances with respect to the mean.

Usage

```
modelGeneVariances(
  x,
  block = NULL,
  block.weight.policy = c("variable", "equal", "none"),
  variable.block.weight = c(0, 1000),
  mean.filter = TRUE,
  min.mean = 0.1,
  transform = TRUE,
  span = 0.3,
  use.min.width = FALSE,
  min.width = 1,
  min.window.count = 200,
  num.threads = 1
)
```

Arguments

`x`	A matrix-like object where rows correspond to genes or genomic features and columns correspond to cells. It is typically expected to contain log-expression values, e.g., from `normalizeCounts`.
`block`	Factor specifying the block of origin (e.g., batch, sample) for each cell in `x`. Alternatively `NULL` if all cells are from the same block.

`block.weight.policy`	String specifying the policy to use for weighting different blocks when computing the average for each statistic Only used if `block` is not `NULL`.
`variable.block.weight`	Numeric vector of length 2, specifying the parameters for variable block weighting. The first and second values are used as the lower and upper bounds, respectively, for the variable weight calculation. Only used if `block` is not `NULL` and `block.weight.policy = "variable"`.
`mean.filter`	Logical scalar indicating whether to filter on the means before trend fitting.
`min.mean`	Numeric scalar specifying the minimum mean of genes to use in trend fitting. Only used if `mean.filter=TRUE`.
`transform`	Logical scalar indicating whether a quarter-root transformation should be applied before trend fitting.
`span`	Numeric scalar specifying the span of the LOWESS smoother. Ignored if `use.min.width=TRUE`.
`use.min.width`	Logical scalar indicating whether a minimum width constraint should be applied to the LOWESS smoother. Useful to avoid overfitting in high-density intervals.
`min.width`	Minimum width of the window to use when `use.min.width=TRUE`.
`min.window.count`	Minimum number of observations in each window. Only used if `use.min.width=TRUE`.
`num.threads`	Integer scalar specifying the number of threads to use.

Value

A list containing `statistics`. This is a data frame with the columns `means`, `variances`, `fitted` and `residuals`, each of which is a numeric vector containing the statistic of the same name across all genes.

If `block` is supplied, each of the column vectors described above contains the average across all blocks. The list will also contain `per.block`, a list of data frames containing the equivalent statistics for each block.

Author(s)

Aaron Lun

See Also

The `model_gene_variances` function in `https://libscran.github.io/scran_variances/`, for the underlying implementation.

`fitVarianceTrend`, which fits the mean-variance trend.

Examples

```
library(Matrix)
x <- abs(rsparsematrix(1000, 100, 0.1) * 10)
out <- modelGeneVariances(x)
str(out)

# Throwing in some blocking.
block <- sample(letters[1:4], ncol(x), replace=TRUE)
out <- modelGeneVariances(x, block=block)
str(out)
```

`normalizeCounts` *Normalize the count matrix*

Description

Apply scaling normalization to a count matrix to obtain log-transformed normalized expression values.

Usage

```
normalizeCounts(
  x,
  size.factors,
  log = TRUE,
  pseudo.count = 1,
  log.base = 2,
  preserve.sparsity = FALSE,
  delayed = TRUE
)
```

Arguments

`x`	A matrix-like object where rows correspond to genes or genomic features and columns correspond to cells. Values are typically expected to be counts. Alternatively, an external pointer created by `initializeCpp`.
`size.factors`	A numeric vector of length equal to the number of cells in `x`, containing positive size factors for all cells.
`log`	Logical scalar indicating whether log-transformation should be performed.
`pseudo.count`	Numeric scalar specifying the positive pseudo-count to add before any log-transformation. Ignored if `log=FALSE`.
`log.base`	Numeric scalar specifying the base of the log-transformation. Ignored if `log=FALSE`.
`preserve.sparsity`	Logical scalar indicating whether to preserve sparsity for `pseudo.count != 1`. If `TRUE`, users should manually add `log(pseudo.count, log.base)` to the returned matrix to obtain the desired log-transformed expression values. Ignored if `log = FALSE` or `pseudo.count = 1`.
`delayed`	Logical scalar indicating whether operations on a matrix-like `x` should be delayed. This improves memory efficiency at the cost of some speed in downstream operations.

Value

If `x` is a matrix-like object and `delayed=TRUE`, a DelayedArray is returned containing the (log-transformed) normalized expression matrix. If `delayed=FALSE`, the type of the (log-)normalized matrix will depend on the operations applied to `x`.

If `x` is an external pointer produced by `initializeCpp`, a new external pointer is returned containing the normalized expression matrix.

Author(s)

Aaron Lun

See Also

The `normalize_counts` function in https://libscran.github.io/scran_norm/, for the rationale behind normalization and log-transformation.

Examples

```
# Mocking a matrix:
library(Matrix)
x <- round(abs(rsparsematrix(1000, 100, 0.1) * 100))
sf <- centerSizeFactors(colSums(x))
normed <- normalizeCounts(x, size.factors=sf)
normed

# Passing a pointer.
ptr <- beachmat::initializeCpp(x)
optr <- normalizeCounts(ptr, sf)
optr
```

`reportGroupMarkerStatistics`

Report marker statistics for a single group

Description

Combine all marker statistics for a single group into a data frame for easy inspection.

Usage

```
reportGroupMarkerStatistics(
  results,
  group,
  effect.sizes = NULL,
  summaries = NULL,
  include.mean = TRUE,
  include.detected = TRUE
)
```

Arguments

`results`	Named list of marker statistics, typically generated by `scoreMarkers` with `all.pairwise=FALSE`.
`group`	String or integer scalar specifying the group of interest.
`effect.sizes`	Character vector specifying the effect sizes of interest. If `NULL`, all effect sizes are reported in the returned data frame.
`summaries`	Character vector specifying the summary statistics of interest. If `NULL`, all summaries are reported in the returned data frame.

`include.mean`	Logical scalar indicating whether the mean expression should be reported in the returned data frame.
`include.detected`	Logical scalar indicating whether the proportion of detected cells should be reported in the returned data frame.

Value

Data frame where each row corresponds to a gene. Each column contains the requested statistics for `group`. Effect size summary columns are named as `<EFFECT>.<SUMMARY>`.

Author(s)

Aaron Lun

See Also

`scoreMarkers`, to generate `results`.

`rna_quality_control` *Quality control for RNA count data*

Description

Compute per-cell QC metrics from an initialized matrix of RNA counts, and use the metrics to suggest filter thresholds to retain high-quality cells.

Usage

```
computeRnaQcMetrics(x, subsets, num.threads = 1)

suggestRnaQcThresholds(metrics, block = NULL, num.mads = 3)

filterRnaQcMetrics(thresholds, metrics, block = NULL)
```

Arguments

`x`	A matrix-like object where rows are genes and columns are cells. Values are expected to be counts.
`subsets`	List of vectors specifying gene subsets of interest, typically for control-like features like mitochondrial genes or spike-in transcripts. Each vector may be logical (whether to keep each row), integer (row indices) or character (row names).
`num.threads`	Integer scalar specifying the number of threads to use.
`metrics`	List with the same structure as produced by `computeRnaQcMetrics`.
`block`	Factor specifying the block of origin (e.g., batch, sample) for each cell in `metrics`. Alternatively `NULL` if all cells are from the same block. For `filterRnaQcMetrics`, a blocking factor should be provided if `block` was used to construct `thresholds`.
`num.mads`	Number of median from the median, to define the threshold for outliers in each metric.
`thresholds`	List with the same structure as produced by `suggestRnaQcThresholds`.

Value

For `computeRnaQcMetrics`, a list is returned containing:

- `sum`, a numeric vector containing the total RNA count for each cell.
- `detected`, an integer vector containing the number of detected genes per cell.
- `subsets`, a list of numeric vectors containing the proportion of counts in each feature subset.

Each vector is of length equal to the number of cells.

For `suggestRnaQcThresholds`, a named list is returned.

- If `block=NULL`, the list contains:
  - `sum`, a numeric scalar containing the lower bound on the sum.
  - `detected`, a numeric scalar containing the lower bound on the number of detected genes.
  - `subsets`, a numeric vector containing the upper bound on the sum of counts in each feature subset.
- Otherwise, if `block` is supplied, the list contains:
  - `sum`, a numeric vector containing the lower bound on the sum for each blocking level.
  - `detected`, a numeric vector containing the lower bound on the number of detected genes for each blocking level.
  - `subsets`, a list of numeric vectors containing the upper bound on the sum of counts in each feature subset for each blocking level.

  Each vector is of length equal to the number of levels in `block` and is named accordingly.

For `filterRnaQcMetrics`, a logical vector of length `ncol(x)` is returned indicating which cells are of high quality.

Author(s)

Aaron Lun

See Also

The `compute_rna_qc_metrics`, `compute_rna_qc_filters` and `compute_rna_qc_filters_blocked` functions in `https://libscran.github.io/scran_qc/`, for the rationale of QC filtering on ADT counts.

Examples

```
# Mocking a matrix:
library(Matrix)
x <- round(abs(rsparsematrix(1000, 100, 0.1) * 100))

# Mocking up a control set.
sub <- list(mito=rbinom(nrow(x), 1, 0.1) > 0)

qc <- computeRnaQcMetrics(x, sub)
str(qc)

filt <- suggestRnaQcThresholds(qc)
str(filt)

keep <- filterRnaQcMetrics(filt, qc)
summary(keep)
```

`runAllNeighborSteps` *Run all neighbor-related steps*

Description

Run all steps that require a nearest-neighbor search. This includs `runUmap`, `runTsne` and `buildSnnGraph` with `clusterGraph`. The idea is to build the index once, perform the neighbor search, and run each task in parallel to save time.

Usage

```
runAllNeighborSteps(
  x,
  runUmap.args = list(),
  runTsne.args = list(),
  buildSnnGraph.args = list(),
  clusterGraph.args = list(),
  BNPARAM = AnnoyParam(),
  return.graph = FALSE,
  collapse.search = FALSE,
  num.threads = 3
)
```

Arguments

`x`	Numeric matrix where rows are dimensions and columns are cells, typically containing a low-dimensional representation from, e.g., `runPca`. Alternatively, an index constructed by `buildIndex`.
`runUmap.args`	Named list of further arguments to pass to `runUmap`. This can be set to `NULL` to omit the UMAP.
`runTsne.args`	Named list of further arguments to pass to `runTsne`. This can be set to `NULL` to omit the t-SNE.
`buildSnnGraph.args`	Named list of further arguments to pass to `buildSnnGraph`. Ignored if `clusterGraph.args=NULL`.
`clusterGraph.args`	Named list of further arguments to pass to `clusterGraph`. This can be set to `NULL` to omit the clustering.
`BNPARAM`	A BiocNeighborParam instance specifying the nearest-neighbor search algorithm to use.
`return.graph`	Logical scalar indicating whether to return the output of `buildSnnGraph`. By default, only the output of `clusterGraph` is returned.
`collapse.search`	Logical scalar indicating whether to collapse the nearest-neighbor search for each step into a single search. Steps that need fewer neighbors will take a subset of the neighbors from the collapsed search. This is faster but may not give the same results as separate searches for some algorithms (e.g., approximate searches).
`num.threads`	Integer scalar specifying the number of threads to use. At least one thread should be available for each step.

Value

A named list containing the results of each step. See each individual function for the format of the results.

Author(s)

Aaron Lun

Examples

```
x <- t(as.matrix(iris[,1:4]))
# (Turning down the number of threads so that R CMD check is happy.)
res <- runAllNeighborSteps(x, num.threads=2)
str(res)
```

`runPca` *Principal components analysis*

Description

Run a PCA on the gene-by-cell log-expression matrix to obtain a low-dimensional representation for downstream analyses.

Usage

```
runPca(
  x,
  number = 25,
  scale = FALSE,
  block = NULL,
  block.weight.policy = c("variable", "equal", "none"),
  variable.block.weight = c(0, 1000),
  components.from.residuals = FALSE,
  extra.work = 7,
  iterations = 1000,
  seed = 5489,
  realized = TRUE,
  num.threads = 1
)
```

Arguments

`x`	A matrix-like object where rows correspond to genes or genomic features and columns correspond to cells. Typically, the matrix is expected to contain log-expression values, and the rows should be filtered to relevant (e.g., highly variable) genes.
`number`	Integer scalar specifying the number of PCs to retain.
`scale`	Logical scalar indicating whether to scale all genes to have the same variance.
`block`	Factor specifying the block of origin (e.g., batch, sample) for each cell in `x`. Alternatively `NULL` if all cells are from the same block.

`block.weight.policy`	String specifying the policy to use for weighting different blocks when computing the average for each statistic. Only used if `block` is not `NULL`.
`variable.block.weight`	Numeric vector of length 2, specifying the parameters for variable block weighting. The first and second values are used as the lower and upper bounds, respectively, for the variable weight calculation. Only used if `block` is not `NULL` and `block.weight.policy = "variable"`.
`components.from.residuals`	Logical scalar indicating whether to compute the PC scores from the residuals in the presence of a blocking factor. By default, the residuals are only used to compute the rotation matrix, and the original expression values of the cells are projected onto this new space. Only used if `block` is not `NULL`.
`extra.work`	Integer scalar specifying the extra dimensions for the IRLBA workspace.
`iterations`	Integer scalar specifying the maximum number of restart iterations for IRLBA.
`seed`	Integer scalar specifying the seed for the initial random vector in IRLBA.
`realized`	Logical scalar indicating whether to realize `x` into an optimal memory layout for IRLBA. This speeds up computation at the cost of increased memory usage.
`num.threads`	Number of threads to use.

Value

List containing:

- `components`, a matrix of PC scores. Rows are dimensions (i.e., PCs) and columns are cells.
- `rotation`, the rotation matrix. Rows are genes and columns are dimensions.
- `variance.explained`, the vector of variances explained by each PC.
- `total.variance`, the total variance in the dataset.
- `center`, a numeric vector containing the mean for each gene. If `block` is provided, this is instead a matrix containing the mean for each gene (column) in each block (row).
- `scale`, a numeric vector containing the scaling for each gene. Only reported if `scale=TRUE`.

Author(s)

Aaron Lun

See Also

`https://libscran.github.io/scran_pca/`, for more details on the PCA. In particular, the documentation for the `blocked_pca` function explains the blocking strategy.

Examples

```
library(Matrix)
x <- abs(rsparsematrix(1000, 100, 0.1) * 10)
y <- normalizeCounts(x, size.factors=centerSizeFactors(colSums(x)))

# A simple PCA:
out <- runPca(y)
str(out)
```

```
# Blocking on uninteresting factors:
block <- sample(LETTERS[1:3], ncol(y), replace=TRUE)
bout <- runPca(y, block=block)
str(bout)
```

`runTsne` *t-stochastic neighbor embedding*

Description

Compute t-SNE coordinates to visualize similarities between cells.

Usage

```
runTsne(
  x,
  perplexity = 30,
  num.neighbors = tsnePerplexityToNeighbors(perplexity),
  max.depth = 20,
  leaf.approximation = FALSE,
  max.iterations = 500,
  seed = 42,
  num.threads = 1,
  BNPARAM = AnnoyParam()
)

tsnePerplexityToNeighbors(perplexity)
```

Arguments

`x`	Numeric matrix where rows are dimensions and columns are cells, typically containing a low-dimensional representation from, e.g., `runPca`. Alternatively, a named list of nearest-neighbor search results like that returned by `findKNN`. This should contain `index`, an integer matrix where rows are neighbors and columns are cells; and `distance`, a numeric matrix of the same dimensions containing the distances to each neighbor. Each column contains 1-based indices for the nearest neighbors of the corresponding cell, ordered by increasing distance. The number of neighbors should be the same as `num.neighbors`, otherwise a warning is raised. Alternatively, an index constructed by `buildIndex`.
`perplexity`	Numeric scalar specifying the perplexity to use in the t-SNE algorithm.
`num.neighbors`	Integer scalar specifying the number of neighbors, typically derived from `perplexity`.
`max.depth`	Integer scalar specifying the maximum depth of the Barnes-Hut quadtree. Smaller values (7-10) improve speed at the cost of accuracy.
`leaf.approximation`	Logical scalar indicating whether to use the "leaf approximation" approach, which sacrifices some accuracy for greater speed. Only effective when `max.depth` is small enough for multiple cells to be assigned to the same leaf node of the quadtree.

`max.iterations`	Integer scalar specifying the maximum number of iterations to perform.
`seed`	Integer scalar specifying the seed to use for generating the initial coordinates.
`num.threads`	Integer scalar specifying the number of threads to use.
`BNPARAM`	A BiocNeighborParam object specifying the algorithm to use. Only used if `x` is not a prebuilt index or a list of existing nearest-neighbor search results.

Value

For `runTsne`, a numeric matrix where rows are cells and columns are the two dimensions of the embedding.

For `tsnePerplexityToNeighbors`, an integer scalar specifying the number of neighbors to use for a given `perplexity`.

Author(s)

Aaron Lun

See Also

`https://libscran.github.io/qdtsne/`, for an explanation of the approximations.

Examples

```
x <- t(as.matrix(iris[,1:4]))
embedding <- runTsne(x)
plot(embedding[,1], embedding[,2], col=iris[,5])
```

`runUmap` *Uniform manifold approxation and projection*

Description

Compute UMAP coordinates to visualize similarities between cells.

Usage

```
runUmap(
  x,
  num.dim = 2,
  num.neighbors = 15,
  num.epochs = NULL,
  min.dist = 0.1,
  seed = 1234567890,
  num.threads = 1,
  parallel.optimization = FALSE,
  BNPARAM = AnnoyParam()
)
```

Arguments

`x`	Numeric matrix where rows are dimensions and columns are cells, typically containing a low-dimensional representation from, e.g., `runPca`. Alternatively, a named list of nearest-neighbor search results like that returned by `findKNN`. This should contain `index`, an integer matrix where rows are neighbors and columns are cells; and `distance`, a numeric matrix of the same dimensions containing the distances to each neighbor. Each column contains 1-based indices for the nearest neighbors of the corresponding cell, ordered by increasing distance. The number of neighbors should be the same as `num.neighbors`, otherwise a warning is raised. Alternatively, an index constructed by `buildIndex`.
`num.dim`	Integer scalar specifying the number of dimensions of the output embedding.
`num.neighbors`	Integer scalar specifying the number of neighbors to use in the UMAP algorithm.
`num.epochs`	Integer scalar specifying the number of epochs to perform. If set to `NULL`, an appropriate number of epochs is chosen based on `ncol(x)`.
`min.dist`	Numeric scalar specifying the minimum distance between points.
`seed`	Integer scalar specifying the seed to use.
`num.threads`	Integer scalar specifying the number of threads to use.
`parallel.optimization`	Logical scalar specifying whether to parallelize the optimization step.
`BNPARAM`	A BiocNeighborParam object specifying the algorithm to use. Only used if `x` is not a prebuilt index or a list of existing nearest-neighbor search results.

Value

A numeric matrix where rows are cells and columns are the two dimensions of the embedding.

Author(s)

Aaron Lun

See Also

`https://libscran.github.io/umappp/`, for details on the underlying implementation.

Examples

```
x <- t(as.matrix(iris[,1:4]))
embedding <- runUmap(x)
plot(embedding[,1], embedding[,2], col=iris[,5])
```

`sanitizeSizeFactors` *Sanitize size factors*

Description

Replace invalid size factors, i.e., zero, negative, infinite or NaN values.

Usage

```
sanitizeSizeFactors(
  size.factors,
  replace.zero = TRUE,
  replace.negative = TRUE,
  replace.infinite = TRUE,
  replace.nan = TRUE
)
```

Arguments

`size.factors`	Numeric vector of size factors across cells.
`replace.zero`	Logical scalar indicating whether to replace size factors of zero with the lowest positive factor. If `FALSE`, zeros are retained.
`replace.negative`	Logical scalar indicating whether to replace negative size factors with the lowest positive factor. If `FALSE`, negative values are retained.
`replace.infinite`	Logical scalar indicating whether to replace infinite size factors with the largest positive factor. If `FALSE`, infinite values are retained.
`replace.nan`	Logical scalar indicating whether to replace NaN size factors with unity. If `FALSE`, NaN values are retained.

Value

Numeric vector of length equal to `size.factors`, containing the sanitized size factors.

Author(s)

Aaron Lun

See Also

The `sanitize_size_factors` function in `https://libscran.github.io/scran_norm/`, for more details on the sanitization.

Examples

```
sf <- 2^rnorm(100)
sf[1] <- 0
sf[2] <- -1
sf[3] <- Inf
sf[4] <- NaN
```

```
sanitizeSizeFactors(sf)
```

`scaleByNeighbors` *Scale and combine multiple embeddings*

Description

Scale multiple embeddings (usually derived from different modalities across the same set of cells) so that their within-population variances are comparable, and then combine them into a single embedding matrix for combined downstream analysis.

Usage

```
scaleByNeighbors(
  x,
  num.neighbors = 20,
  num.threads = 1,
  weights = NULL,
  BNPARAM = AnnoyParam()
)
```

Arguments

`x`	List of numeric matrices of principal components or other embeddings, one for each modality. For each entry, rows are dimensions and columns are cells. All entries should have the same number of columns but may have different numbers of rows.
`num.neighbors`	Integer scalar specifying the number of neighbors to use to define the scaling factor.
`num.threads`	Integer scalar specifying the number of threads to use.
`weights`	Numeric vector of length equal to that of `x`, specifying the weights to apply to each modality. Each value represents a multiplier of the within-population variance of its modality, i.e., larger values increase the contribution of that modality in the combined output matrix. `NULL` is equivalent to an all-1 vector, i.e., all modalities are scaled to have the same within-population variance.
`BNPARAM`	A BiocNeighborParam object specifying how to perform the neighbor search.

Value

List containing `scaling`, a vector of scaling factors to be aplied to each embedding; and `combined`, a numeric matrix formed by scaling each entry of `x` and then `rbind`ing them together.

Author(s)

Aaron Lun

See Also

`https://libscran.github.io/mumosa/`, for the basis and caveats of this approach.

Examples

```
pcs <- list(
    gene = matrix(rnorm(10000), ncol=200),
    protein = matrix(rnorm(1000, sd=3), ncol=200),
    guide = matrix(rnorm(2000, sd=5), ncol=200)
)

out <- scaleByNeighbors(pcs)
out$scaling
dim(out$combined)
```

`scoreGeneSet` *Score gene set activity for each cell*

Description

Compute per-cell scores for a gene set, defined as the column sums of a rank-1 approximation to the submatrix for the gene set. This uses the same approach implemented in the **GSDecon** package by Jason Hackney.

Usage

```
scoreGeneSet(
  x,
  set,
  rank = 1,
  scale = FALSE,
  block = NULL,
  block.weight.policy = c("variable", "equal", "none"),
  variable.block.weight = c(0, 1000),
  extra.work = 7,
  iterations = 1000,
  seed = 5489,
  realized = TRUE,
  num.threads = 1
)
```

Arguments

`x`	A matrix-like object where rows correspond to genes or genomic features and columns correspond to cells. Typically, the matrix is expected to contain log-expression values.
`set`	Integer, logical or character vector specifying the rows that belong to the gene set.
`rank`	Integer scalar specifying the rank of the approximation.
`scale`	Logical scalar indicating whether to scale all genes to have the same variance.
`block`	Factor specifying the block of origin (e.g., batch, sample) for each cell in `x`. Alternatively `NULL` if all cells are from the same block.
`block.weight.policy`	String specifying the policy to use for weighting different blocks when computing the average for each statistic. Only used if `block` is not `NULL`.

`variable.block.weight`	Numeric vector of length 2, specifying the parameters for variable block weighting. The first and second values are used as the lower and upper bounds, respectively, for the variable weight calculation. Only used if `block` is not `NULL` and `block.weight.policy = "variable"`.
`extra.work`	Integer scalar specifying the extra dimensions for the IRLBA workspace.
`iterations`	Integer scalar specifying the maximum number of restart iterations for IRLBA.
`seed`	Integer scalar specifying the seed for the initial random vector in IRLBA.
`realized`	Logical scalar indicating whether to realize `x` into an optimal memory layout for IRLBA. This speeds up computation at the cost of increased memory usage.
`num.threads`	Number of threads to use.

Value

List containing `scores`, a numeric vector of per-cell scores for each column in `x`; and `weights`, a numeric vector of per-gene weights for each gene in `set`.

Author(s)

Aaron Lun

See Also

`https://libscran.github.io/gsdecon/`, for more details on the underlying algorithm. In particular, the documentation for the `compute_blocked` function explains the blocking strategy.

Examples

```
library(Matrix)
x <- round(abs(rsparsematrix(1000, 100, 0.1) * 100))
normed <- normalizeCounts(x, size.factors=centerSizeFactors(colSums(x)))
scoreGeneSet(normed, set=c(1,3,5,10,20,100))
```

`scoreMarkers` *Score marker genes*

Description

Score marker genes for each group using a variety of effect sizes from pairwise comparisons between groups. This includes Cohen's d, the area under the curve (AUC), the difference in the means (delta-mean) and the difference in the proportion of detected cells (delta-detected).

Usage

```
scoreMarkers(
  x,
  groups,
  block = NULL,
  block.weight.policy = c("variable", "equal", "none"),
  variable.block.weight = c(0, 1000),
```

```
  compute.delta.mean = TRUE,
  compute.delta.detected = TRUE,
  compute.cohens.d = TRUE,
  compute.auc = TRUE,
  threshold = 0,
  all.pairwise = FALSE,
  num.threads = 1
)
```

Arguments

`x`	A matrix-like object where rows correspond to genes or genomic features and columns correspond to cells. It is typically expected to contain log-expression values, e.g., from `normalizeCounts`.
`groups`	A vector specifying the group assignment for each cell in `x`.
`block`	Factor specifying the block of origin (e.g., batch, sample) for each cell in `x`. Alternatively `NULL` if all cells are from the same block.
`block.weight.policy`	String specifying the policy to use for weighting different blocks when computing the average for each statistic Only used if `block` is not `NULL`.
`variable.block.weight`	Numeric vector of length 2, specifying the parameters for variable block weighting. The first and second values are used as the lower and upper bounds, respectively, for the variable weight calculation. Only used if `block` is not `NULL` and `block.weight.policy = "variable"`.
`compute.delta.mean`	Logical scalar indicating whether to compute the delta-means, i.e., the log-fold change when `x` contains log-expression values.
`compute.delta.detected`	Logical scalar indicating whether to compute the delta-detected, i.e., differences in the proportion of cells with detected expression.
`compute.cohens.d`	Logical scalar indicating whether to compute Cohen's d.
`compute.auc`	Logical scalar indicating whether to compute the AUC. Setting this to `FALSE` can improve speed and memory efficiency.
`threshold`	Non-negative numeric scalar specifying the minimum threshold on the differences in means (i.e., the log-fold change, if `x` contains log-expression values). This is incorporated into the effect sizes for Cohen's d and the AUC.
`all.pairwise`	Logical scalar indicating whether to report the full effects for every pairwise comparison between groups.
`num.threads`	Integer scalar specifying the number of threads to use.

Value

If `all.pairwise=FALSE`, a named list is returned containing:

- `mean`, a numeric matrix containing the mean expression for each group. Each row is a gene and each column is a group.
- `detected`, a numeric matrix containing the proportion of detected cells in each group. Each row is a gene and each column is a group.

- `cohens.d`, a list of data frames where each data frame corresponds to a group. Each row of each data frame represents a gene, while each column contains a summary of Cohen's d from pairwise comparisons to all other groups. This includes the `min`, `mean`, `median`, `max` and `min.rank` - check out `?summarizeEffects` for details. Omitted if `compute.cohens.d=FALSE`.
- `auc`, a list like `cohens.d` but containing the summaries of the AUCs from each pairwise comparison. Omitted if `compute.auc=FALSE`.
- `delta.mean`, a list like `cohens.d` but containing the summaries of the delta-mean from each pairwise comparison. Omitted if `compute.delta.mean=FALSE`.
- `delta.detected`, a list like `cohens.d` but containing the summaries of the delta-detected from each pairwise comparison. Omitted if `compute.delta.detected=FALSE`.

If `all.pairwise=TRUE`, a list is returned containing:

- `mean`, a numeric matrix containing the mean expression for each group. Each row is a gene and each column is a group.
- `detected`, a numeric matrix containing the proportion of detected cells in each group. Each row is a gene and each column is a group.
- `cohens.d`, a 3-dimensional numeric array containing the Cohen's d from each pairwise comparison between groups. The extents of the first two dimensions are equal to the number of groups, while the extent of the final dimension is equal to the number of genes. The entry `[i, j, k]` represents Cohen's d from the comparison of group `j` over group `i` for gene `k`. Omitted if `compute.cohens.d=FALSE`.
- `auc`, an array like `cohens.d` but containing the AUCs from each pairwise comparison. Omitted if `compute.auc=FALSE`.
- `delta.mean`, an array like `cohens.d` but containing the delta-mean from each pairwise comparison. Omitted if `compute.delta.mean=FALSE`.
- `delta.detected`, an array like `cohens.d` but containing the delta-detected from each pairwise comparison. Omitted if `compute.delta.detected=FALSE`.

See Also

The `score_markers_summary` and the `score_markers_pairwise` function (for `all.pairwise=FALSE` and `TRUE`, respectively) in `https://libscran.github.io/scran_markers/`, which describes the rationale behind the choice of effect sizes and summary statistics. Also see their blocked equivalents `score_markers_summary_blocked` and `score_markers_pairwise_blocked` when `block` is not `NULL`.

`summarizeEffects`, to summarize the pairwise effects returned when `all.pairwise=TRUE`.

`reportGroupMarkerStatistics`, to consolidate the statistics for a single group into its own data frame.

Examples

```
# Mocking a matrix:
library(Matrix)
x <- round(abs(rsparsematrix(1000, 100, 0.1) * 100))
normed <- normalizeCounts(x, size.factors=centerSizeFactors(colSums(x)))

# Compute marker summaries for each group:
g <- sample(letters[1:4], ncol(x), replace=TRUE)
scores <- scoreMarkers(normed, g)
names(scores)
```

```
head(scores$mean)
head(scores$cohens.d[["a"]])

# Report marker statistics for a single group:
reportGroupMarkerStatistics(scores, "b")
```

`subsampleByNeighbors` *Subsample cells based on their neighbors*

Description

Subsample a dataset by selecting cells to represent all of their nearest neighbors.

Usage

```
subsampleByNeighbors(
  x,
  num.neighbors = 20,
  min.remaining = 10,
  num.threads = 1,
  BNPARAM = AnnoyParam()
)
```

Arguments

`x` A numeric matrix where rows are dimensions and columns are cells, typically containing a low-dimensional representation from, e.g., `runPca`.

Alternatively, an index constructed by `buildIndex`.

Alternatively, a list containing existing nearest-neighbor search results. This should contain:

- `index`, an integer matrix where rows are neighbors and columns are cells. Each column contains 1-based indices for the nearest neighbors of the corresponding cell, ordered by increasing distance.
- `distance`, a numeric matrix of the same dimensions as `index`, containing the distances to each of the nearest neighbors.

The number of neighbors should be equal to `num.neighbors`, otherwise a warning is raised.

`num.neighbors` Integer scalar specifying the number of neighbors to use. Larger values result in greater downsampling. Only used if `x` does not contain existing nearest-neighbor results.

`min.remaining` Integer scalar specifying the minimum number of remaining (i.e., unselected) neighbors that a cell must have in order to be considered for selection. This should be less than or equal to `num.neighbors`.

`num.threads` Integer scalar specifying the number of threads to use for the nearest-neighbor search. Only used if `x` does not contain existing nearest-neighbor results.

`BNPARAM` A BiocNeighborParam object specifying the algorithm to use. Only used if `x` does not contain existing nearest-neighbor results.

Value

Integer vector with the indices of the selected cells in the subsample.

Author(s)

Aaron Lun

See Also

`https://libscran.github.io/nenesub/`, for more details on the underlying algorithm.

Examples

```
x <- matrix(rnorm(10000), nrow=2)
keep <- subsampleByNeighbors(x, 10)
plot(x[1,], x[2,])
points(x[1,keep], x[2,keep], col="red")
legend('topright', col=c('black', 'red'), legend=c('all', 'subsample'), pch=1)
```

`summarizeEffects` *Summarize pairwise effect sizes for each group*

Description

For each group, summarize the effect sizes for all pairwise comparisons to other groups. This yields a set of summary statistics that can be used to rank marker genes for each group.

Usage

```
summarizeEffects(effects, num.threads = 1)
```

Arguments

`effects`	A 3-dimensional numeric containing the effect sizes from each pairwise comparison between groups. The extents of the first two dimensions are equal to the number of groups, while the extent of the final dimension is equal to the number of genes. The entry `[i, j, k]` represents the effect size from the comparison of group `j` against group `i` for gene `k`. See also the output of `scoreMarkers` with `all.pairwise=TRUE`.
`num.threads`	Integer scalar specifying the number of threads to use.

Details

Each summary statistic can be used to prioritize different sets of marker genes for the group of interest, by ranking them in decreasing order according to said statistic:

- `min` contains the minimum effect size across all comparisons involving the group of interest. Using this to define markers will focus on genes that are upregulated in all comparisons.
- `mean` contains the mean effect size across all comparisons involving the group of interest. Using this to define markers will focus on genes that are generally upregulated.

- `median` contains the median effect size across all comparisons involving the group of interest. Using this to define markers will focus on genes that are upregulated in most comparisons.
- `max` contains the maximum effect size across all comparisons involving the group of interest. Using this to define markers will focus on genes that are upregulated in at least one comparison.

The `min.rank` is a more exotic summary statistic, containing the minimum rank for each gene across all comparisons involving the group of interest. This is defined by ranking the effect sizes across genes within each comparison, and then taking the minimum of these ranks across comparisons. Taking all genes with `min.rank` <= `T` will yield a set containing the top `T` genes from each comparison; the idea is to ensure that there are at least `T` genes that can distinguish the group of interest from the others.

Value

List of data frames containing summary statistics for the effect sizes. Each data frame corresponds to a group, each row corresponds to a gene, and each column contains a single summary.

Author(s)

Aaron Lun

See Also

The `summarize_effects` function in `https://libscran.github.io/scran_markers/`, for more details on the statistics.

`scoreMarkers`, to compute the pairwise effects in the first place.

Examples

```
# Mocking a matrix:
library(Matrix)
x <- round(abs(rsparsematrix(1000, 100, 0.1) * 100))
normed <- normalizeCounts(x, size.factors=centerSizeFactors(colSums(x)))

g <- sample(letters[1:4], ncol(x), replace=TRUE)
effects <- scoreMarkers(normed, g, all.pairwise=TRUE)

summarized <- summarizeEffects(effects$cohens.d)
str(summarized)
```

`testEnrichment` *Test for gene set enrichment*

Description

Perform a hypergeometric test for enrichment of interesting genes (e.g., markers) in one or more pre-defined gene sets.

Usage

```
testEnrichment(x, sets, universe, log = FALSE, num.threads = 1)
```

Arguments

`x`	Vector of identifiers for the interesting genes.
`sets`	List of vectors of identifiers for the pre-defined gene sets.
`universe`	Vector of identifiers for the universe of genes in the dataset. It is expected that `x` is a subset of `universe`. Alternatively, an integer scalar specifying the size of the universe.
`log`	Logical scalar indicating whether to report the log-transformed p-values.
`num.threads`	Integer scalar specifying the number of threads to use.

Value

Numeric vector of (log-transformed) p-values to test for significant enrichment of `x` in each entry of `sets`.

Author(s)

Aaron Lun

See Also

`phyper` and `https://libscran.github.io/phyper/`, which is the basis for the underlying calculation.

Examples

```
testEnrichment(
    x=LETTERS[1:5],
    sets=list(
        first=LETTERS[1:10],
        second=LETTERS[1:5 * 2],
        third=LETTERS[10:20]
    ),
    universe=LETTERS
)
```

Index