Package 'scrapper'

January 20, 2026

Version 1.4.0

Date 2025-10-27

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 (>= 1.3.10), 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 7.3.3

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

git_branch RELEASE_3_22

git_last_commit 4de467a

git_last_commit_date 2025-10-29

Repository Bioconductor 3.22

Date/Publication 2026-01-19

Author Aaron Lun [cre, aut]

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

Contents

adt_quality_control . . . 2
aggregateAcrossCells . . . 5
aggregateAcrossGenes . . . 6
analyze . . . 7
buildSnnGraph . . . 12
centerSizeFactors . . . 14
chooseHighlyVariableGenes . . . 15
choosePseudoCount . . . 17
clusterGraph . . . 18
clusterKmeans . . . 19
combineFactors . . . 22
computeBlockWeights . . . 23
computeClrm1Factors . . . 24
convertAnalyzeResults . . . 25
correctMnn . . . 26
crispr_quality_control . . . 28
fitVarianceTrend . . . 30
modelGeneVariances . . . 32
normalizeCounts . . . 34
reportGroupMarkerStatistics . . . 35
rna_quality_control . . . 37
runAllNeighborSteps . . . 39
runPca . . . 40
runTsne . . . 42
runUmap . . . 45
sanitizeSizeFactors . . . 48
scaleByNeighbors . . . 49
scoreGeneSet . . . 51
scoreMarkers . . . 53
subsampleByNeighbors . . . 57
summarizeEffects . . . 58
testEnrichment . . . 60

Index **62**

`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. In theory, this represents the efficiency of library preparation and sequencing. Compared to RNA, the sum is less useful as a QC metric for ADT data as it is strongly influenced by biological variation in the abundance of the targeted features. Nonetheless, we compute it for diagnostic purposes.
- detected, an integer vector containing the number of detected tags per cell. Even though ADTs are typically used in situations where few features are highly abundant (e.g., cell type-specific markers), we still expect detectable coverage of most features due to ambient contamination, non-specific binding or some background expression. Low numbers of detected tags indicates that library preparation or sequencing depth was suboptimal.
- subsets, a list of numeric vectors containing the total count of each control subset. The exact interpretation depends on the nature of the feature subset but the most common use case involves isotype control (IgG) features. IgG antibodies should not bind to anything so a high subset sum suggests that non-specific binding is a problem, e.g., due to antibody conjugates. (Unlike RNA quality control, we do not use proportions here as it is entirely possible for a cell to have low counts for other tags due to the absence of their targeted features; this would result in a high proportion even if the cell has a "normal" level of non-specific binding.)

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. This is defined as the lower of (i) num.mads MADs below the median for the log-transformed values across all cells, and (ii) the product of 1 - min.detected.drop and the median across all cells. The latter avoids overly aggressive filtering when the MAD is zero.
  - subsets, a numeric vector containing the upper bound on the sum of counts in each control subset. This is defined as num.mads MADs above the median of the log-transformed metrics across all cells.

- 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. Here, the threshold is computed independently for each block, using the same method as the unblocked case.
  - subsets, a list of numeric vectors containing the upper bound on the sum of counts in each control subset for each blocking level. Here, the threshold is computed independently for each block, using the same method as the unblocked case.

  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. High-quality cells are defined as those with numbers of detected tags above the detected threshold and control subset sums below the subsets threshold.

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/.

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 usually applied to a count matrix to create pseudo-bulk profiles for each cluster/sample combination. These profiles can then be used as if they were counts from bulk data, e.g., for differential analyses with **edgeR**.

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 levels from `factors`. 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 levels from `factors`. Each entry contains the number of cells with detected expression in that combination.
- `combinations`, a data frame containing the unique combination of levels from `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/.

`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, convert = TRUE, num.threads = 1)
```

Arguments

x	A matrix-like object where rows correspond to genes or genomic features and columns correspond to cells. Values are usually normalized expression values, possibly log-transformed depending on the application.
sets	List of vectors where each entry corresponds to a gene set. Each entry may be an integer vector of row indices, a logical vector of length equal to the number of rows, or a character vector of row names. For integer and character vectors, duplicate elements are ignored. Alternatively, each entry may be a list of two vectors. The first vector should be either integer (row indices) or character (row names), specifying the genes in the set. The second vector should be numeric and of the same length as the first vector, specifying the weight associated with each gene. If duplicate genes are present, only the first occurrence is used. If the first vector contains gene names not present in x, those genes are ignored.
average	Logical scalar indicating whether to compute the average rather than the sum.
convert	Logical scalar indicating whether to convert gene identities to non-duplicate row indices in each entry of sets. Can be set to FALSE for greater efficiency if the sets already contains non-duplicated integer vectors.
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/.

`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 `NULL` 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

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 ranking of those shared neighbors. If two cells are close together but have distinct sets of neighbors, the corresponding edge is downweighted as the two cells are unlikely to be part of the same neighborhood. In this manner, strongly weighted edges will only form within highly interconnected neighborhoods where many cells share the same neighbors. This provides a more sophisticated definition of similarity between cells compared to a simpler (unweighted) nearest neighbor graph that just focuses on immediate proximity.

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. Larger values increase the connectivity of the graph and reduce the granularity of subsequent community detection steps, at the cost of speed.
weight.scheme	String specifying the weighting scheme to use for constructing the SNN graph. This can be one of: • "ranked", where the weight of the edge is defined by the smallest sum of ranks across all shared neighbors. More shared neighbors, or shared neighbors that are close to both observations, will generally yield larger weights. • "number", where the weight of the edge is the number of shared nearest neighbors between them. This is a simpler scheme that is also slightly faster but does not account for the ranking of neighbors within each set. • "jaccard", where the weight of the edge is the Jaccard index of their neighbor sets, This is a monotonic transformation of the weight used in "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/.

`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. This ensures that the original scale of the counts is preserved in the normalized values from `normalizeCounts`, which simplifies interpretation and ensures that any pseudo-count added prior to log-transformation has a predictable shrinkage effect.

Usage

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

Arguments

`size.factors`	Numeric vector of size factors across cells. Invalid size factors (e.g., non-positive, non-finite) will be ignored.
`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. This can be either `"lowest"` or `"per-block"`, see Details. Only used if `block` is provided.

Details

`"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. Here, our normalization strategy involves downscaling all blocks to match the coverage of the lowest-coverage block. This is useful for datasets with big differences in coverage between blocks as it avoids egregious upscaling of low-coverage blocks. Specifically, strong upscaling allows the log-transformation to ignore any shrinkage from the pseudo-count. This is problematic as it inflates differences between cells at log-values derived from low counts, increasing noise and overstating log-fold changes. Downscaling is safer as it allows the pseudo-count to shrink the log-differences between cells towards zero at low counts, effectively sacrificing some information in the higher-coverage batches so that they can be compared to the low-coverage batches (which is preferable to exaggerating the informativeness of the latter for comparison to the former).

`"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. The scaled size factors are identical to those obtained by separate invocations of ‘center_size_factors()‘ on the size factors for each block. This can be desirable to ensure consistency with independent analyses of each block - otherwise, the centering would depend on the size factors in other blocks. However, any systematic differences in the size factors between blocks are lost, i.e., systematic changes in coverage between blocks will not be normalized.

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/.

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. This is typically used to subset the gene-cell matrix prior to calling `runPca`.

Usage

```
chooseHighlyVariableGenes(
  stats,
  top = 4000,
  larger = TRUE,
  keep.ties = TRUE,
  bound = 0
)
```

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. This is typically the case for variances or related statistics, e.g., residuals.
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 satisfy this bound, even if they are within the top genes. For example, residuals from the fitted trend should be positive, which can be enforced by setting bound to zero. 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/.

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 from normalizeCounts. Larger pseudo-counts shrink log-expression values towards the zero-expression baseline, reducing the impact of the transformation bias at the cost of some sensitivity.

Usage

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

Arguments

size.factors	Numeric vector of size factors for all cells. It is expected that these have already been centered, e.g., with centerSizeFactors. Invalid size factors (e.g., non-positive, non-finite) will be ignored.
quantile	Numeric scalar specifying the quantile to use for finding the smallest/largest size factors. Setting this to zero will use the observed minimum and maximum, though in practice, this is usually too sensitive to outliers. The default is to take the 5th and 95th percentile to obtain a range that captures most of the distribution.
max.bias	Numeric scalar specifying the maximum allowed bias. This is the maximum absolute value of any spurious log2-fold change between the cells with the smallest and largest size factors.
min.value	Numeric scalar specifying the minimum value for the pseudo-count. Defaults to 1 to stabilize near-zero normalized expression values, otherwise these manifest as avoid large negative values.

Value

A choice of pseudo-count for normalizeCounts.

Author(s)

Aaron Lun

See Also

choose_pseudo_count in https://libscran.github.io/scran_norm/.

Lun ATL (2018). Overcoming systematic errors caused by log-transformation of normalized single-cell RNA sequencing data. *biorXiv* doi:10.1101/404962

Examples

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

clusterGraph *Graph-based clustering of cells*

Description

Identify clusters by applying community detection algorithms to a graph. This assumes that that the nodes on the graph represent cells and weighted edges are formed between related cells.

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" uses multi-level modularity optimization, also known as the Louvain algorithm, see https://igraph.org/c/doc/igraph-Community.html#igraph_community_multilevel for details. • "walktrap" uses the Walktrap community finding algorithm, see https://igraph.org/c/doc/igraph-Community.html#igraph_community_walktrap for details. • "leiden" uses the Leiden algorithm, see https://igraph.org/c/doc/igraph-Community.html#igraph_community_leiden for details.
multilevel.resolution	Numeric scalar specifying the resolution when method="multilevel". Lower values favor fewer, larger communities; higher values favor more, smaller communities.
leiden.resolution	Numeric scalar specifying the resolution when method="leiden". Lower values favor fewer, larger communities; higher values favor more, smaller communities.
leiden.objective	String specifying the objective function when method="leiden". "cpm" uses the Constant Potts Model, which typically yields more fine-grained clusters at the same leiden.resolution.
walktrap.steps	Integer scalar specifying the number of steps to use when method="walktrap". This determines the ability of the Walktrap algorithm to distinguish highly interconnected communities from the rest of the graph.
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

The various `cluster_*` functions in https://libscran.github.io/scran_graph_cluster/.

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. This typically contains a low-dimensional representation from, e.g., runPca.
k	Integer scalar specifying the number of clusters.
init.method	String specifying the initialization method for the centers: • "var-part" uses variance partitioning as described by Su and Dy (2007). The dataset is repeatedly split along the dimension of greatest variance until k partitions are formed, the centroids of which form the initial clusters. This approach is slower than the others but fully deterministic. • "kmeans++" uses the weighted sampling method described by Arthur and Vassilvitskii (2007). k points are sampled with probability based on the smallest distance to any previously sampled point. This improves the likelihood of choosing initial centroids that are far apart from each other. • "random" initialization involves choosing k random points as the initial centers. This is the simplest and fastest method but may not yield good starting points.
refine.method	String specifying the refinement method. • "lloyd" uses Lloyd's algorithm, which performs a batch update in each iteration. This is simple and amenable to parallelization but may not converge. • "hartigan-wong" uses the Hartigan-Wong algorithm, which transfers points between clusters to optimize the drop in the within-cluster sum of squares. This is slower but has a greater chance of convergence.
var.part.optimize.partition	Logical scalar indicating whether each partition boundary should be optimized to reduce the sum of squares in the child partitions. This is slower but improves the quality of the partition. 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 selecting the next cluster to partition. Setting this to 0 or 1 will select the cluster with the highest variance or sum of squares, respectively, for partitioning. In other words, a value of 0 will ignore the cluster size while setting a value of 1 will generally cause larger clusters to be selected. Only used if init.method = "var.part".
lloyd.iterations	Integer scalar specifying the maximum number of iterations for the Lloyd algorithm. Larger values increase the chance of convergence at the cost of increasing compute time. Only used if refine.method = "lloyd".
hartigan.wong.iterations	Integer scalar specifying the maximum number of iterations for the Hartigan-Wong algorithm. Larger values increase the chance of convergence at the cost of increasing compute time. Only used if refine.method = "hartigan-wong".

`hartigan.wong.quick.transfer.iterations`
: Integer scalar specifying the maximum number of quick transfer iterations for the Hartigan-Wong algorithm. Larger values increase the chance of convergence at the cost of increasing compute time. Only used if `refine.method = "hartigan-wong"`.

`hartigan.wong.quit.quick.transfer.failure`
: Logical scalar indicating whether to quit the Hartigan-Wong algorithm upon convergence failure during quick transfer iterations. Setting this to `FALSE` gives the algorithm another chance to converge by attempting another optimal transfer iteration, at the cost of more compute time. If `TRUE`, the function follows the same behavior as R's `kmeans`. Only used if `refine.method = "hartigan-wong"`.

`seed`
: Integer scalar specifying the seed for random number generation. Only used if `init.method = "random"` or `"kmeans++"`.

`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 assignment for each cell. The number of levels is no greater than `k`, where each level is an integer that refer to a column of `centers`.
- `centers`, a numeric matrix with the coordinates of the cluster centroids (dimensions in rows, centers in columns). The number of columns is no greater than `k`. Empty clusters are automatically removed.
- `iterations`, an integer scalar specifying the number of refinement iterations that were performed.
- `status`, an integer scalar specifying the completion status of the algorithm. A value of zero indicates success while the meaning of any non-zero value depends on the choice of `refine.method`:
  - For Lloyd, a value of 2 indicates convergence failure.
  - For Hartigan-Wong, a value of 2 indicates convergence failure in the optimal transfer iterations. A value of 4 indicates convergence failure in the quick transfer iterations when `hartigan.wong.quit.quick.transfer.failure = TRUE`.

Author(s)

Aaron Lun

References

Hartigan JA. and Wong MA (1979). Algorithm AS 136: A K-means clustering algorithm. *Applied Statistics* 28, 100-108.

Arthur D and Vassilvitskii S (2007). k-means++: the advantages of careful seeding. *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms* 1027-1035.

Su T and Dy JG (2007). In Search of Deterministic Methods for Initializing K-Means and Gaussian Mixture Clustering. *Intelligent Data Analysis* 11, 319-338.

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 factors into a single factor where each level of the latter is a unique combination of levels from the former.

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. In other words, for observation `i` and factor `j`, `levels[[[j]][index[i]]` will recover `factors[[j]][i]`.

Author(s)

Aaron Lun

See Also

The `combine_to_factor` function in https://ltla.github.io/factorize/.

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)
```

`computeBlockWeights` *Compute block weights*

Description

Compute a weight for each block based on the number of cells in each block. This is typically used to aggregate statistics across blocks, e.g., with weighted sums/averages.

Usage

```
computeBlockWeights(
  sizes,
  block.weight.policy = c("variable", "equal", "size", "none"),
  variable.block.weight = c(0, 1000)
)
```

Arguments

`sizes`	Numeric vector containing the size of (i.e., number of cells in) each block.
`block.weight.policy`	String specifying the policy to use for weighting different blocks. This should be one of: • `"size"`: the contribution of each block is proportional to its size. `"none"` is also a deprecated alias for `"size"`. • `"equal"`: blocks are equally weighted regardless of their size. The exception is that of empty blocks with no cells, which receive zero weight. • `"variable"`: blocks are equally weighted past a certain threshold size. Below that size, the contribution of each block is proportional to its size. This avoids outsized contributions from very large blocks.
`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.weight.policy = "variable"`.

Value

Numeric vector containing the relative block weights.

Author(s)

Aaron Lun

See Also

The `compute_weights` function from https://libscran.github.io/scran_blocks/.

Examples

```
computeBlockWeights(c(1, 10, 100, 1000, 10000))
computeBlockWeights(c(1, 10, 100, 1000, 10000), block.weight.policy="equal")
computeBlockWeights(c(1, 10, 100, 1000, 10000), variable.block.weight=c(50, 5000))
```

---

computeClrm1Factors *Compute size factors for ADT counts*

Description

Compute size factors from an ADT count matrix using the CLRm1 method. This is a variant of the centered log-ratio (CLR) method, where the size factors are defined from the geometric mean of counts within each cell.

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 embedding.

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 coordinates in a low-dimensional embedding (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 in the various search steps. Larger values improve the stability of the correction by increasing the number of MNN pairs and including more observations in each center of mass. However, this comes at the cost of reduced resolution when matching subpopulations across batches.
`num.steps`	Integer scalar specifying the number of steps for the recursive neighbor search to compute the center of mass. Larger values mitigate the kissing effect but increase the risk of including inappropriately distant subpopulations into the center of mass.
`merge.policy`	String specifying the policy to use to choose the order of batches to merge. • `"input"` will use the input order of the batches. Observations in the last batch are corrected first, and then the second-last batch, and so on. This allows users to control the merge order by simply changing the inputs. • `"size"` will merge batches in order of increasing size (i.e., the number of observations). So, the smallest batch is corrected first while the largest batch is unchanged. The aim is to lower compute time by reducing the number of observations that need to be reprocessed in later merge steps.

- "variance" will merge batches in order of increasing variance between observations. So, the batch with the lowest variance is corrected first while the batch with the highest variance is unchanged. The aim is to lower compute time by encouraging more observations to be corrected to the most variable batch, thus avoid reprocessing in later merge steps.
- "rss" will merge batches in order of increasing residual sum of squares (RSS). This is effectively a compromise between "variance" and "size".

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

References

Haghverdi L, Lun ATL, Morgan MD, Marioni JC (2018). Batch effects in single-cell RNA-sequencing data are corrected by matching mutual nearest neighbors. *Nat. Biotechnol.* 36(5):421-427

See Also

The compute function in https://libscran.github.io/mnncorrect/.

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`.

Details

In CRISPR data, a cell is considered to be of low quality if it has a low count for its most abundant guide. However, directly defining a MAD-based outlier threshold on the maximum count is somewhat tricky as unsuccessful transfection can be common. This often results in a large subpopulation with low maximum counts, inflating the MAD and compromising the threshold calculation. Instead, we use the following approach:

- Compute the proportion of counts in the most abundant guide (i.e., the maximum proportion) in each cell. Cells that were successfully transfected should have high maximum proportions. In contrast, unsuccessfully transfected cells will be dominated by ambient contamination and have low proportions.
- Subset the dataset to only retain those cells with maximum proportions above the median. This assumes that at least 50 Thus, we remove all of the unsucessful transfections and enrich for mostly-high-quality cells.
- Define a MAD-based threshold for low outliers on the log-transformed maximum count within the subset (see 'choose_filter_thresholds()' for details). This is now possible as we can assume that most of the remaining cells are of high quality.

Note that the maximum proportion is only used to define the subset for threshold calculation. Once the maximum count threshold is computed, it is applied to all cells regardless of their maximum proportions. This ensures that we correctly remove cells with low coverage, even if the proportion is high. It also allows us to retain cells transfected with multiple guides, as long as the maximum is high enough - such cells are not necessarily uninteresting, e.g., for examining interaction effects, so we will err on the side of caution and leave them in.

Value

For `computeCrisprQcMetrics`, a list is returned containing:

- `sum`, a numeric vector containing the total CRISPR count for each cell. Low counts indicate that the cell was not successfully transfected with a construct or that library preparation and sequencing failed.
- `detected`, an integer vector containing the number of detected guides per cell. In theory, this should be 1, as each cell should express no more than one guide construct. However, ambient contamination may introduce non-zero counts for multiple guides, without necessarily interfering with downstream analyses. As such, this metric is less useful for guide data, though we compute it anyway.
- `max.value`, a numeric vector containing the count for the most abundant guide in cell. Low values indicate that the cell was not successfully transfected or that library preparation and sequencing failed.
- `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 count. This is defined as `num.mads` MADs below the median of the log-transformed metrics across cells with high maximum proportions (see Details).
- 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. Here, the threshold is computed independently for each block, using the same method as the unblocked case.

  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. High-quality cells are defined as those with maximum counts above the `max.value` threshold.

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/.

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-gene variances with respect to their means, typically from normalized and log-transformed expression values.

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. The assumption is that there is a bulk of low-abundance genes that are uninteresting and should be removed to avoid skewing the windows of the LOWESS smoother.
`min.mean`	Numeric scalar specifying the minimum mean of genes to use in trend fitting. Genes with lower means do not participate in the LOWESS fit, to ensure that windows are not skewed towards the majority of low-abundance genes. Instead, the fitted values for these genes are defined by extrapolating the left edge of the fitted trend is extrapolated to the origin. The default value is chosen based on

the typical distribution of means of log-expression values across genes. Only used if `mean.filter=TRUE`.

`transform` Logical scalar indicating whether a quarter-root transformation should be applied before trend fitting. This transformation is copied from `limma::voom` and shrinks all values towards 1, flattening any sharp gradients in the trend for an easier fit. The default of `TRUE` assumes that the variances are computed from log-expression values, in which case there is typically a strong "hump" in the mean-variance relationship.

`span` Numeric scalar specifying the span of the LOWESS smoother, as a proportion of the total number of points. Larger values improve stability at the cost of sensitivity to changes in low-density regions. Ignored if `use.min.width=TRUE`.

`use.min.width` Logical scalar indicating whether a minimum width constraint should be applied to the LOWESS smoother. This replaces the proportion-based span for defining each window. Instead, the window for each point must be of a minimum width and is extended until it contains a minimum number of points. Setting this to 'TRUE' ensures that sensitivity is maintained in the trend fit at low-density regions for the distribution of means, e.g., at high abundances. It also avoids overfitting from very small windows in high-density intervals.

`min.width` Minimum width of the window to use when `use.min.width=TRUE`. The default value is chosen based on the typical range of means in single-cell RNA-seq data.

`min.window.count`
Minimum number of observations in each window. This ensures that each window contains at least a given number of observations for a stable fit. If the minimum width window contains fewer observations, it is extended using the standard LOWESS logic until the minimum number is achieved. Only used if `use.min.width=TRUE`.

`num.threads` Number of threads to use.

Value

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

Author(s)

Aaron Lun

See Also

`modelGeneVariances`, to compute the means and variances.

The `fit_variance_trend` function in https://libscran.github.io/scran_variances/.

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

Model the per-gene variances as a function of the mean in single-cell expression data. Highly variable genes can then be selected for downstream analyses.

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`. If provided, calculation of means/variances and trend fitting are performed within each block to ensure that block effects do not confound the estimates. The weighted average of each statistic across all blocks is reported for each gene. 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. See the argument of the same name in `computeBlockWeights` for more detail. Only used if `block` is not `NULL`.
`variable.block.weight`	Numeric vector of length 2, specifying the parameters for variable block weighting. See the argument of the same name in `computeBlockWeights` for more detail. 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. The assumption is that there is a bulk of low-abundance genes that are uninteresting and should be removed to avoid skewing the windows of the LOWESS smoother.
`min.mean`	Numeric scalar specifying the minimum mean of genes to use in trend fitting. Genes with lower means do not participate in the LOWESS fit, to ensure that windows are not skewed towards the majority of low-abundance genes. Instead,

the fitted values for these genes are defined by extrapolating the left edge of the fitted trend is extrapolated to the origin. The default value is chosen based on the typical distribution of means of log-expression values across genes. Only used if `mean.filter=TRUE`.

`transform` Logical scalar indicating whether a quarter-root transformation should be applied before trend fitting. This transformation is copied from `limma::voom` and shrinks all values towards 1, flattening any sharp gradients in the trend for an easier fit. The default of `TRUE` assumes that the variances are computed from log-expression values, in which case there is typically a strong "hump" in the mean-variance relationship.

`span` Numeric scalar specifying the span of the LOWESS smoother, as a proportion of the total number of points. Larger values improve stability at the cost of sensitivity to changes in low-density regions. Ignored if `use.min.width=TRUE`.

`use.min.width` Logical scalar indicating whether a minimum width constraint should be applied to the LOWESS smoother. This replaces the proportion-based span for defining each window. Instead, the window for each point must be of a minimum width and is extended until it contains a minimum number of points. Setting this to 'TRUE' ensures that sensitivity is maintained in the trend fit at low-density regions for the distribution of means, e.g., at high abundances. It also avoids overfitting from very small windows in high-density intervals.

`min.width` Minimum width of the window to use when `use.min.width=TRUE`. The default value is chosen based on the typical range of means in single-cell RNA-seq data.

`min.window.count`
Minimum number of observations in each window. This ensures that each window contains at least a given number of observations for a stable fit. If the minimum width window contains fewer observations, it is extended using the standard LOWESS logic until the minimum number is achieved. Only used if `use.min.width=TRUE`.

`num.threads` Integer scalar specifying the number of threads to use.

Details

We compute the mean and variance for each gene and fit a trend to the variances with respect to the means using `fitVarianceTrend`. We assume that most genes at any given abundance are not highly variable, such that the fitted value of the trend is interpreted as the "uninteresting" variance - this is mostly attributed to technical variation like sequencing noise, but can also represent constitutive biological noise like transcriptional bursting. Under this assumption, the residual can be treated as a measure of biologically interesting variation. Genes with large residuals can then be selected for downstream analyses, e.g., with `chooseHighlyVariableGenes`.

Value

A list containing `statistics`, a data frame with number of rows equal to the number of genes. This contains 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/.

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 and log-transformation to a count matrix. This yields normalized expression values that can be used in downstream procedures like PCA.

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 expected to be non-negative 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. Any invalid values should be replaced with sanitizeSizeFactors. For most applications, these size factors should also be centered with centerSizeFactors.
log	Logical scalar indicating whether log-transformation should be performed. This ensures that downstream analyses like t-tests and distance calculations focus on relative fold-changes rather than absolute differences. The log-transformation also provides some measure of variance stabilization so that the downstream analyses are not dominated by sampling noise at large counts.

pseudo.count — Numeric scalar specifying the positive pseudo-count to add before any log-transformation. Larger values shrink the differences between cells towards zero, reducing spurious differences (but also signal) at low counts - see choosePseudoCount for comments. 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/.

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. Users can pick one of the columns for sorting potential marker genes.

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`.

`summarizeEffects`, for the trade-offs between effect size summaries.

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. This represents the efficiency of library preparation and sequencing. Low totals indicate that the library was not successfully captured.
- detected, an integer vector containing the number of detected genes per cell. This also quantifies library preparation efficiency but with greater focus on capturing transcriptional complexity.
- subsets, a list of numeric vectors containing the proportion of counts in each feature subset. The exact interpretation of which depends on the nature of the subset. For example, if one subset contains all genes on the mitochondrial chromosome, higher proportions are representative of cell damage; the assumption is that cytoplasmic transcripts leak through tears in the cell membrane while the mitochondria are still trapped inside. The proportion of spike-in transcripts can be interpreted in a similar manner, where the loss of endogenous transcripts results in higher spike-in proportions.

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. This is defined as `num.mads` MADs below the median of the log-transformed metrics across all cells.
  - `detected`, a numeric scalar containing the lower bound on the number of detected genes. This is defined as `num.mads` MADs below the median of the log-transformed metrics across all cells.
  - `subsets`, a numeric vector containing the upper bound on the sum of counts in each feature subset. This is defined as `num.mads` MADs above the median across all cells.
- Otherwise, if `block` is supplied, the list contains:
  - `sum`, a numeric vector containing the lower bound on the sum for each blocking level. Here, the threshold is computed independently for each block, using the same method as the unblocked case.
  - `detected`, a numeric vector containing the lower bound on the number of detected genes for each blocking level. Here, the threshold is computed independently for each block, using the same method as the unblocked case.
  - `subsets`, a list of numeric vectors containing the upper bound on the sum of counts in each feature subset for each blocking level. Here, the threshold is computed independently for each block, using the same method as the unblocked case.

  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. High-quality cells are defined as those with sums and detected genes above their respective thresholds and subset proportions below the `subsets` threshold.

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/.

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 and extract the top principal components (PCs). This yields a low-dimensional representation that reduces noise and compute time in downstream analyses. For efficiency, the PCA itself is approximated using IRLBA.

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 (see normalizeCounts) for "interesting" genes (see chooseHighlyVariableGenes
number	Integer scalar specifying the number of top PCs to retain. More PCs will capture more biological signal at the cost of increasing noise and compute time. If this is greater than the maximum number of PCs (i.e., the smaller dimension of x), only the maximum number of PCs will be reported in the results.

scale
: Logical scalar indicating whether to scale all genes to have the same variance. This ensures that each gene contributes equally to the PCA, favoring consistent variation across many genes rather than large variation in a few genes. If block is specified, each gene's variance is calculated as a weighted sum of the variances from each block. Genes with zero variance are ignored.

block
: Factor specifying the block of origin (e.g., batch, sample) for each cell in x. The PCA will be performed on the residuals after regressing out the block effect, ensuring that differences between block do not dominate the variation in the dataset. Alternatively NULL if all cells are from the same block.

block.weight.policy
: String specifying the policy to use for weighting the contribution of different blocks to the PCA. See the argument of the same name in computeBlockWeights for more detail. Only used if block is not NULL.

variable.block.weight
: Numeric vector of length 2, specifying the parameters for variable block weighting. See the argument of the same name in computeBlockWeights for more detail. 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 (see Details). Only used if block is not NULL.

extra.work
: Integer scalar specifying the extra dimensions for the IRLBA workspace. Larger values improve accuracy at the cost of compute time.

iterations
: Integer scalar specifying the maximum number of restart iterations for IRLBA. Larger values improve accuracy at the cost of compute time.

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.

Details

When block is specified, the nature of the reported PC scores depends on the choice of components.from.residuals:

- If TRUE, the PC scores are computed from the matrix of residuals. This yields a low-dimensional space where inter-block differences have been removed, assuming that all blocks have the same subpopulation composition and the inter-block differences are consistent for all cell subpopulations. Under these assumptions, we could use these components for downstream analysis without any concern for block-wise effects.
- If FALSE, the rotation vectors are first computed from the matrix of residuals. To obtain PC scores, each cell is then projected onto the associated subspace using its original expression values. This approach ensures that inter-block differences do not contribute to the PCA but does not attempt to explicitly remove them.

In complex datasets, the assumptions mentioned for TRUE not hold and more sophisticated batch correction methods like MNN correction are required. Functions like correctMnn will accept a low-dimensional embedding of cells that can be created as described above with FALSE.

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. This can be used to divide variance.explained to obtain the proportion of variance explained by each PC.
- 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

The simple_pca and blocked_pca functions for https://libscran.github.io/scran_pca/.

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),
  theta = 1,
  early.exaggeration.iterations = 250,
```

```
  exaggeration.factor = 12,
  momentum.switch.iterations = 250,
  start.momentum = 0.5,
  final.momentum = 0.8,
  eta = 200,
  max.depth = 7,
  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. Higher perplexities will focus on global structure, at the cost of increased runtime and decreased local resolution.
num.neighbors	Integer scalar specifying the number of neighbors, typically derived from perplexity.
theta	Numeric scalar specifying the approximation level for the Barnes-Hut calculation of repulsive forces. Lower values increase accuracy at the cost of increased compute time. All values should be non-negative.
early.exaggeration.iterations	Integer scalar specifying the number of iterations of the early exaggeration phase, where clusters are artificially compacted to leave more empty space so that cells can easily relocate to find a good global organization. Larger values improve convergence within this phase at the cost of reducing the remaining iterations in max.iterations.
exaggeration.factor	Numeric scalar containing the exaggeration factor for the early exaggeration phase (see early.exaggeration.iterations). Larger values increase the attraction between nearest neighbors to favor local structure.
momentum.switch.iterations	Integer scalar specifying the number of iterations to perform before switching from the starting momentum to the final momentum. Higher momentums can improve convergence by increasing the step size and smoothing over local oscillations, at the risk of potentially skipping over relevant minima.
start.momentum	Numeric scalar containing the starting momentum, to be used in the iterations before the momentum switch at momentum.switch.iterations. This is usually lower than final.momentum to avoid skipping over suitable local minima.

`final.momentum` Numeric scalar containing the final momentum, to be used in the iterations after the momentum switch at `momentum.switch.iterations`. This is usually higher than `start.momentum` to accelerate convergence to the local minima once the observations are moderately well-organized.

`eta` Numeric scalar containing the learning rate, used to scale the updates for each cell. Larger values can speed up convergence at the cost of skipping over local minima.

`max.depth` Integer scalar specifying the maximum depth of the Barnes-Hut quadtree. If neighboring cells cannot be separated before the maximum depth is reached, they will be assigned to the same leaf node of the quadtree. Smaller values (7-10) improve speed by bounding the recursion depth at the cost of accuracy.

`leaf.approximation` Logical scalar indicating whether to use the "leaf approximation". If `TRUE`, repulsive forces are computed between leaf nodes and re-used across all cells assigned to that leaf node. This sacrifices some accuracy for greater speed, assuming that `max.depth` is small enough for multiple cells to be assigned to the same leaf.

`max.iterations` Integer scalar specifying the maximum number of iterations to perform. Larger values improve convergence at the cost of compute time.

`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

References

van der Maaten LJP and Hinton GE (2008). Visualizing high-dimensional data using t-SNE. *Journal of Machine Learning Research_* 9, 2579-2605.

van der Maaten LJP (2014). Accelerating t-SNE using tree-based algorithms. *Journal of Machine Learning Research* 15, 3221-3245.

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,
  local.connectivity = 1,
  bandwidth = 1,
  mix.ratio = 1,
  spread = 1,
  min.dist = 0.1,
  a = NULL,
  b = NULL,
  repulsion.strength = 1,
  initialize.method = c("spectral", "random", "none"),
  initial.coordinates = NULL,
  initialize.random.on.spectral.fail = TRUE,
  initialize.spectral.scale = 10,
  initialize.spectral.jitter = FALSE,
  initialize.spectral.jitter.sd = 1e-04,
  initialize.random.scale = 10,
  initialize.seed = 9876543210,
  num.epochs = NULL,
  learning.rate = 1,
  negative.sample.rate = 5,
  num.neighbors = 15,
  optimize.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.

local.connectivity
Numeric scalar specifying the number of nearest neighbors that are assumed to be always connected, with maximum membership confidence. Larger values increase the connectivity of the embedding and reduce the focus on local structure. This may be a fractional number of neighbors, in which case interpolation is performed when computing the membership confidence.

bandwidth Numeric scalar specifying the effective bandwidth of the kernel when converting the distance to a neighbor into a fuzzy set membership confidence. Larger values reduce the decay in confidence with respect to distance, increasing connectivity and favoring global structure.

mix.ratio Numeric scalar between 0 and 1 specifying the mixing ratio when combining fuzzy sets. A mixing ratio of 1 will take the union of confidences, a ratio of 0 will take the intersection, and intermediate values will interpolate between them. Larger values favor connectivity and more global structure.

spread Numeric scalar specifying the scale of the coordinates of the final low-dimensional embedding. Ignored if a and b are provided.

min.dist Numeric scalar specifying the minimum distance between observations in the final low-dimensional embedding. Smaller values will increase local clustering while larger values favor a more even distribution of observations throughout the low-dimensional space. This is interpreted relative to spread. Ignored if a and b are provided.

a Numeric scalar specifying the a parameter for the fuzzy set membership strength calculations. Larger values yield a sharper decay in membership strength with increasing distance between observations. If this or b are NULL, a suitable value for this parameter is automatically determined from spread and min.dist.

b Numeric scalar specifying the b parameter for the fuzzy set membership strength calculations. Larger values yield an earlier decay in membership strength with increasing distance between observations. If this or a are NULL, a suitable value for this parameter is automatically determined from spread and min.dist.

repulsion.strength
Numeric scalar specifying the modifier for the repulsive force. Larger values increase repulsion and favor local structure.

initialize.method
String specifying how to initialize the embedding. This should be one of:

- SPECTRAL: spectral decomposition of the normalized graph Laplacian. Specifically, the initial coordinates are defined from the eigenvectors corresponding to the smallest non-zero eigenvalues. This fails in the presence of multiple graph components or if the approximate SVD fails to converge.
- RANDOM: fills the embedding with random draws from a normal distribution.
- NONE: uses initial values from initial.coordinates.

initial.coordinates
Numeric matrix of initial coordinates, with number of rows equal to the number of observations and number of columns equal to num.dim. Only relevant if initialize.method = "NONE"; or initialize.method = "SPECTRAL" and spectral initialization fails and initialize.random.on.spectral.fail = FALSE.

initialize.random.on.spectral.fail
Logical scalar indicating whether to fall back to random sampling (i.e., same as RANDOM) if spectral initialization fails due to the presence of multiple components in the graph. If FALSE, the values in initial.coordinates will be used

instead, i.e., same as NONE. Only relevant if initialize.method = "SPECTRAL" and spectral initialization fails.

initialize.spectral.scale
: Numeric scalar specifying the maximum absolute magnitude of the coordinates after spectral initialization. All initial coordinates are scaled such that the maximum of the absolute values is equal to initialize.spectral.scale. This ensures that outlier observations will not have large absolute distances that may interfere with optimization. Only relevant if initialize.method = "SPECTRAL" and spectral initialization does not fail.

initialize.spectral.jitter
: Logical scalar indicating whether to jitter coordinates after spectral initialization to separate duplicate observations (e.g., to avoid overplotting). This is done using normally-distributed noise of mean zero and standard deviation of initialize.spectral.jitter.sd. Only relevant if initialize.method = "SPECTRAL" and spectral initialization does not fail.

initialize.spectral.jitter.sd
: Numeric scalar specifying the standard deviation of the jitter to apply after spectral initialization. Only relevant if initialize.method = "SPECTRAL" and spectral initialization does not fail and initialize.spectral.jitter = TRUE.

initialize.random.scale
: Numeric scalar specifying the scale of the randomly generated initial coordinates. Coordinates are sampled from a uniform distribution from $[-x, x)$ where x is initialize.random.scale. Only relevant if initialize.method = "RANDOM", or initialize.method = "SPECTRAL" and spectral initialization fails and initialize.random.on.spectral.fail = TRUE.

initialize.seed
: Numeric scalar specifying the seed for the random number generation during initialization. Only relevant if initialize.method = "RANDOM", or initialize.method = "SPECTRAL" and initialize.spectral.jitter = TRUE; or initialize.method = "SPECTRAL" and spectral initialization fails and initialize.random.on.spectral.fail = TRUE.

num.epochs
: Integer scalar specifying the number of epochs for the gradient descent, i.e., optimization iterations. Larger values improve accuracy at the cost of increased compute time. If NULL, a value is automatically chosen based on the size of the dataset:

  - For datasets with no more than 10000 observations, the default number of epochs is set to 500.
  - For larger datasets, the number of epochs is inversely proportional to the number of cells, starting from 500 and decreasing asymptotically to a lower limit of 200. This choice aims to reduce computational work for very large datasets.

learning.rate
: Numeric scalar specifying the initial learning rate used in the gradient descent. Larger values can accelerate convergence but at the risk of skipping over suitable local optima.

negative.sample.rate
: Numeric scalar specifying the rate of sampling negative observations to compute repulsive forces. Greater values will improve accuracy but increase compute time.

num.neighbors
: Integer scalar specifying the number of neighbors to use to define the fuzzy sets. Larger values improve connectivity and favor preservation of global structure, at

	the cost of increased compute time. If x contains pre-computed neighbor search result, the number of neighbors should be equal to num.neighbors.
optimize.seed	Numeric scalar specifying the seed to use for the optimization epochs.
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

References

McInnes L, Healy J, Melville J (2020). UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction. *arXiv*, https://arxiv.org/abs/1802.03426

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. Such size factors can occasionally arise if, e.g., insufficient quality control was performed upstream. Removing them ensures that the normalized values from normalizeCounts remain finite for sensible downstream processing.

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 in size.factors. This ensures that the normalized values will be large to reflect the extremity of the scaling, but still finite for sensible downstream processing. If FALSE, zeros are retained.
replace.negative	Logical scalar indicating whether to replace negative size factors with the lowest positive factor in size.factors. This ensures that the normalized values will be large to reflect the extremity of the scaling, but still finite for sensible downstream processing. If FALSE, negative values are retained.
replace.infinite	Logical scalar indicating whether to replace infinite size factors with the largest positive factor in size.factors. This ensures that any normalized values will be, at least, finite; the choice of a relatively large replacement value reflects the extremity of the scaling. If FALSE, infinite values are retained.
replace.nan	Logical scalar indicating whether to replace NaN size factors with unity, e.g., scaling normalization is a no-op. 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/.

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 for the same cells) so that their within-population variances are comparable, and then combine them into a single embedding matrix for further analyses like clustering, t-SNE, etc. The aim is to equalize uninteresting variance across modalities so that high technical variance in one modality does not drown out interesting biology in another modality.

Usage

```
scaleByNeighbors(
  x,
  num.neighbors = 20,
  block = NULL,
  block.weight.policy = c("variable", "equal", "none"),
  variable.block.weight = c(0, 1000),
  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.
block	Factor specifying the block of origin (e.g., batch, sample) for each cell in x. If provided, the scaling factor is computed as a weighted average across blocks to ensure that block effects do not inflate the within-population variance. 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 scaling factor. See the argument of the same name in computeBlockWeights for more detail. Only used if block is not NULL.
variable.block.weight	Numeric vector of length 2, specifying the parameters for variable block weighting. See the argument of the same name in computeBlockWeights for more detail. Only used if block is not NULL and block.weight.policy = "variable".
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 creating by scaling each entry of x by scaling and then rbinding 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 as the **GSDecon** package by Jason Hackney, adapted to use an approximate PCA (via IRLBA) and to support blocking.

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`	Vector specifying the rows of `x` that belong to the gene set. This may be an integer vector of row indices, a logical vector of length equal to the number of rows, or a character vector of row names. For integer and character vectors, duplicate entries are ignored.
`rank`	Integer scalar specifying the rank of the approximation. The default value of 1 assumes that each gene set only describes a single coordinated biological function.

scale — Logical scalar indicating whether to scale all genes to have the same variance. This ensures that each gene contributes equally to the PCA, favoring consistent variation across many genes rather than large variation in a few genes. If block is specified, each gene's variance is calculated as a weighted sum of the variances from each block. Genes with zero variance are ignored.

block — Factor specifying the block of origin (e.g., batch, sample) for each cell in x. The PCA will be performed on the residuals after regressing out the block effect, ensuring that differences between block do not dominate the variation in the dataset. Alternatively NULL if all cells are from the same block.

block.weight.policy — String specifying the policy to use for weighting the contribution of different blocks to the PCA. See the argument of the same name in computeBlockWeights for more detail. Only used if block is not NULL.

variable.block.weight — Numeric vector of length 2, specifying the parameters for variable block weighting. See the argument of the same name in computeBlockWeights for more detail. Only used if block is not NULL and block.weight.policy = "variable".

extra.work — Integer scalar specifying the extra dimensions for the IRLBA workspace. Larger values improve accuracy at the cost of compute time.

iterations — Integer scalar specifying the maximum number of restart iterations for IRLBA. Larger values improve accuracy at the cost of compute time.

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.
- weights, a data frame containing row, an integer vector of ordered and unique row indices corresponding to the genes in set; and weight, a numeric vector of per-gene weights for each gene in row.

Author(s)

Aaron Lun

See Also

The compute and compute_blocked functions in https://libscran.github.io/gsdecon/.

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). For each group, the strongest markers are those genes with the largest effect sizes (i.e., upregulated) when compared to all other groups.

Usage

```
scoreMarkers(
  x,
  groups,
  block = NULL,
  block.weight.policy = c("variable", "equal", "none"),
  variable.block.weight = c(0, 1000),
  compute.group.mean = TRUE,
  compute.group.detected = TRUE,
  compute.delta.mean = TRUE,
  compute.delta.detected = TRUE,
  compute.cohens.d = TRUE,
  compute.auc = TRUE,
  threshold = 0,
  all.pairwise = FALSE,
  min.rank.limit = 500,
  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. If provided, comparisons are performed within each block to ensure that block effects do not confound the estimates. The weighted average of the effect sizes across all blocks is reported for each gene. 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. See the argument of the same name in computeBlockWeights for more detail. Only used if block is not NULL.
variable.block.weight	Numeric vector of length 2, specifying the parameters for variable block weighting. See the argument of the same name in computeBlockWeights for more detail. Only used if block is not NULL and block.weight.policy = "variable".

compute.group.mean
: Logical scalar indicating whether to compute the group-wise mean expression for each gene.

compute.group.detected
: Logical scalar indicating whether to compute the group-wise proportion of detected cells for each gene.

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. Larger thresholds will favor genes with large differences at the expense of genes with low variance that would otherwise have comparable effect sizes.

all.pairwise
: Logical scalar indicating whether to report the effect sizes for every pairwise comparison between groups. Alternatively, an integer scalar indicating the number of top markers to report from each pairwise comparison between groups. If FALSE, only the summary statistics are reported.

min.rank.limit
: Integer scalar specifying the maximum value of the min-rank to report. Lower values improve memory efficiency at the cost of discarding information about lower-ranked genes. Only used if all.pairwise=FALSE.

num.threads
: Integer scalar specifying the number of threads to use.

Value

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

- 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 named list is returned containing:

- 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.

If all.pairwise is an integer, a named list is returned containing:

- cohens.d, a list of list of data frames containing the top genes with the largest Cohen's d for each pairwise comparison. Specifically, cohens.d[[i]][[j]] is a data frame contains the top all.pairwise genes from the comparison of group i over group j. Each data frame contains index, the row index of the gene; and effect, the Cohen's d for that gene. Omitted if compute.cohens.d=FALSE.
- auc, a list of list of data frames like cohens.d but containing the AUCs from each pairwise comparison. Omitted if compute.auc=FALSE.
- delta.mean, a list of list of data frames like cohens.d but containing the delta-mean from each pairwise comparison. Omitted if compute.delta.mean=FALSE.
- delta.detected, a list of list of data frames like cohens.d but containing the delta-detected from each pairwise comparison. Omitted if compute.delta.detected=FALSE.

All returned lists will also contain:

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

Choice of effect size

The delta-mean is the difference in the mean expression between groups. This is fairly straightforward to interpret - a positive delta-mean corresponds to increased expression in the first group compared to the second. The delta-mean can also be treated as the log-fold change if the input matrix contains log-transformed normalized expression values.

The delta-detected is the difference in the proportion of cells with detected expression between groups. This lies between 1 and -1, with the extremes occurring when a gene is silent in one group and detected in all cells of the other group. For this interpretation, we assume that the input matrix contains non-negative expression values, where a value of zero corresponds to lack of detectable expression.

Cohen's d is the standardized difference between two groups. This is defined as the difference in the mean for each group scaled by the average standard deviation across the two groups. (Technically, we should use the pooled variance; however, this introduces some unintuitive asymmetry depending on the variance of the larger group, so we take a simple average instead.) A positive value indicates that the gene has increased expression in the first group compared to the second. Cohen's d is analogous to the t-statistic in a two-sample t-test and avoids spuriously large effect sizes from comparisons between highly variable groups. We can also interpret Cohen's d as the number of standard deviations between the two group means.

The area under the curve (AUC) is the probability that a randomly chosen observation in one group is greater than a randomly chosen observation in the other group. Values greater than 0.5 indicate that a gene is upregulated in the first group. The AUC is closely related to the U-statistic used in the Wilcoxon rank sum test. The key difference between the AUC and Cohen's d is that the former

is less sensitive to the variance within each group, e.g., if two distributions exhibit no overlap, the AUC is the same regardless of the variance of each distribution. This may or may not be desirable as it improves robustness to outliers but reduces the information available to obtain a fine-grained ranking.

With a minimum change threshold

Setting a minimum change threshold (i.e., `threshold`) prioritizes genes with large shifts in expression instead of those with low variances. Currently, only positive thresholds are supported, which focuses on genes that are upregulated in the first group compared to the second. The effect size definitions are generalized when testing against a non-zero threshold:

- Cohen's d is redefined as the standardized difference between the difference in means and the specified threshold, analogous to the TREAT method from the **limma** package. Large positive values are only obtained when the observed difference in means is significantly greater than the threshold. For example, if we had a threshold of 2 and we obtained a Cohen's d of 3, this means that the observed difference in means was 3 standard deviations greater than 2. Note that a negative Cohen's d cannot be intepreted as downregulation, as the difference in means may still be positive but less than the threshold.
- The AUC is generalized to the probability of obtaining a random observation in one group that is greater than a random observation plus the threshold in the other group. For example, if we had a threshold of 2 and we obtained an AUC of 0.8, this means that, 80 the random observation from the first group would be greater than a random observation from the second group by 2 or more. Again, AUCs below 0.5 cannot be interpreted as downregulation, as it may be caused by a positive shift that is less than the threshold.

See Also

The `score_markers_summary`, `score_markers_pairwise` and `score_markers_best` functions in https://libscran.github.io/scran_markers/. See their blocked equivalents (e.g., `score_markers_summary_bloc` when `block` is specified.

`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. The aim is to preserve the relative density of the original dataset while guaranteeing representation of low-frequency subpopulations.

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 stronger downsampling. Only used if `x` does not contain existing nearest-neighbor results.
`min.remaining`	Integer scalar specifying the minimum number of remaining neighbors that a cell must have in order to be considered for selection. This should be less than or equal to `num.neighbors`. Larger values result in stronger downsampling.
`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.

Details

Starting from the densest region in the high-dimensional space, we select an observation for inclusion into the subsampled dataset. Every time we select an observation, we remove it and all of its

nearest neighbors from the dataset. We then select the next observation with the most remaining neighbors, with ties broken by density; this is repeated until there are no more observations.

The premise is that each selected observation serves as a representative for its nearest neighbors. This ensures that the subsampled points are well-distributed across the original dataset. Low-frequency subpopulations will always have at least a few representatives if they are sufficiently distant from other subpopulations. We also preserve the relative density of the original dataset as more representatives will be generated from high-density regions.

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. Genes with large values are upregulated in all comparisons. As such, it is the most stringent summary as markers will only have large values if they are uniquely upregulated in the group of interest compared to every other group.
- `mean` contains the mean effect size across all comparisons involving the group of interest. Genes with large values are upregulated on average compared to the other groups. This is a good general-purpose summary statistic.
- `median` contains the median effect size across all comparisons involving the group of interest. Genes with large values are upregulated compared to most (i.e., at least 50 Compared to the mean, this is more robust to outlier effects but less sensitive to strong effects in a minority of 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. As such, it is the least stringent summary as markers can achieve large values if they are upregulated in the group of interest compared to any one other group.

The exact definition of "large" depends on the choice of effect size. For signed effects like Cohen's d, delta-mean and delta-detected, the value must be positive to be considered "large". For the AUC, a value greater than 0.5 is considered "large". This interpretation is also affected by the choice of `threshold=` used to compute each effect size in `scoreMarkers`, e.g., a negative Cohen's d cannot be interpreted as downregulation when the threshold is positive.

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 any other group.

`NaN` effect sizes are allowed, e.g., if two groups do not exist in the same block for a blocked analysis in `scoreMarkers` with `block=`. This function will ignore `NaN` values when computing each summary. If all effects are NaN for a particular group, the summary statistic will also be `NaN`.

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 summary statistic.

Author(s)

Aaron Lun

See Also

The `summarize_effects` function in https://libscran.github.io/scran_markers/.

`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 gene sets in a list of interesting genes (e.g., markers).

Usage

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

Arguments

x	Vector of identifiers for some interesting genes, e.g., symbols or Ensembl IDs. This is usually derived from a selection of top markers, e.g., from scoreMarkers.
sets	List of vectors of identifiers for the pre-defined gene sets. Each inner vector corresponds to a gene set and should contain the same type of identifiers as x.
universe	Vector of identifiers for the universe of genes in the dataset. x and each vector in sets will be subsetted to only include those genes in universe. If NULL, the universe is defined as the union of all genes in x and sets. Alternatively, an integer scalar specifying the number of genes in the universe. This is assumed to be greater than or equal to the number of unique genes in x and sets.
log	Logical scalar indicating whether to report log-transformed p-values. This may be desirable to avoid underflow at near-zero p-values.
num.threads	Integer scalar specifying the number of threads to use.

Value

Data frame with one row per gene set and the following columns:

- overlap, the overlap between x and each entry of sets, i.e., the number of genes in the intersection.
- size, the set of each entry of sets.
- p.value, the (possibly log-transformed) p-value for overrepresentation of the gene set in x.

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

adt_quality_control, 2
aggregateAcrossCells, 5, *7*
aggregateAcrossGenes, *5*, 6
analyze, 7, *25*

BiocNeighborParam, *10*, *13*, *27*, *39*, *44*, *48*, *50*, *57*
buildIndex, *13*, *39*, *43*, *45*, *57*
buildSnnGraph, *9*, *11*, 12, *18*, *39*

centerSizeFactors, *9*, *11*, 14, *17*, *24*, *34*
chooseHighlyVariableGenes, *9*, *11*, 15, *33*, *40*
choosePseudoCount, 17, *35*
clusterGraph, *9*, *12–14*, 18, *39*
clusterKmeans, *10*, *12*, 19
combineFactors, *5*, 22
computeAdtQcMetrics, *9, 10*
computeAdtQcMetrics *(adt_quality_control)*, 2
computeBlockWeights, 23, *32*, *41*, *50*, *52, 53*
computeClrm1Factors, *9*, *11*, 24
computeCrisprQcMetrics, *10*
computeCrisprQcMetrics *(crispr_quality_control)*, 28
computeRnaQcMetrics, *9, 10*
computeRnaQcMetrics *(rna_quality_control)*, 37
convertAnalyzeResults, *12*, 25
correctMnn, *9*, *11*, 26, *41*
crispr_quality_control, 28

DataFrame, *25*
DelayedArray, *35*

filterAdtQcMetrics, *10*
filterAdtQcMetrics *(adt_quality_control)*, 2
filterCrisprQcMetrics, *10*
filterCrisprQcMetrics *(crispr_quality_control)*, 28
filterRnaQcMetrics, *10*
filterRnaQcMetrics *(rna_quality_control)*, 37
findKNN, *43*, *45*
fitVarianceTrend, 30, *33*

igraph, *18*
initializeCpp, *34, 35*

kmeans, *21*

modelGeneVariances, *9*, *11*, *16*, *31*, 32

normalizeCounts, *9*, *11*, *14*, *17*, *32*, 34, *40*, *48*, *53*

phyper, *61*

reportGroupMarkerStatistics, 35, *56*
rna_quality_control, 37
runAllNeighborSteps, *10, 11*, 39
runPca, *9*, *11*, *13*, *15*, *20*, *26*, *39*, 40, *43*, *45*, *57*
runTsne, *9*, *11*, *39*, 42
runUmap, *9*, *11*, *39*, 45

sanitizeSizeFactors, *34*, 48
scaleByNeighbors, *9*, *11*, 49
scoreGeneSet, 51
scoreMarkers, *10*, *12*, *36*, 53, *58–60*
SingleCellExperiment, *12*, *25*
subsampleByNeighbors, 57
suggestAdtQcThresholds, *9, 10*
suggestAdtQcThresholds *(adt_quality_control)*, 2
suggestCrisprQcThresholds, *9, 10*
suggestCrisprQcThresholds *(crispr_quality_control)*, 28
suggestRnaQcThresholds, *9, 10*
suggestRnaQcThresholds *(rna_quality_control)*, 37
SummarizedExperiment, *8*, *10*
summarizeEffects, *36*, *54*, *56*, 58

testEnrichment, 60
tsnePerplexityToNeighbors *(runTsne)*, 42