Package 'bluster'

March 29, 2021

Version 1.0.0

Date 2020-10-15

Title Clustering Algorithms for Bioconductor

Description Wraps common clustering algorithms in an easily extended S4 framework. Backends are implemented for hierarchical, k-means and graph-based clustering. Several utilities are also provided to compare and evaluate clustering results.

Imports stats, methods, utils, Matrix, Rcpp, igraph, S4Vectors, BiocParallel, BiocNeighbors

Suggests knitr, rmarkdown, testthat, BiocStyle, dynamicTreeCut, scRNAseq, scuttle, scater, scran, pheatmap, viridis

biocViews ImmunoOncology, Software, GeneExpression, Transcriptomics, SingleCell, Clustering

LinkingTo Rcpp

License GPL-3

NeedsCompilation yes

VignetteBuilder knitr

SystemRequirements C++11

RoxygenNote 7.1.1

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

git_branch RELEASE_3_12

git_last_commit 3727538

git_last_commit_date 2020-10-27

Date/Publication 2021-03-29

Author Aaron Lun [aut, cre]

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

R topics documented:

`approxSilhouette` *Approximate silhouette width*

Description

Given a clustering, compute a fast approximate silhouette width for each cell.

Usage

```
approxSilhouette(x, clusters)
```

Arguments

`x`	A numeric matrix-like object containing observations in rows and variables in columns.
`clusters`	Vector of length equal to `ncol(x)`, specifying the cluster assigned to each observation.

Details

The silhouette width is a general-purpose method for evaluating the separation between clusters but requires calculating the average distance between pairs of observations within or between clusters. This function instead approximates the average distances for faster computation in large datasets.

For a given observation, let $\tilde{D}$ be the approximate average distance to all cells in cluster X. This is defined as the square root of the sum of:

- The squared distance from the current observation to the centroid of cluster X. This is most accurate when the observation is distant to X relative to the latter's variation.
- The summed variance of all variables across observations in cluster X. This is most accurate when the observation lies close to the close to the centroid of X.

This is also equivalent to the root-square-mean distance from the current observation to all cells in X.

The approximate silhouette width for each cell can then be calculated with the relevant two values of $\tilde{D}$, computed by setting X to the cluster of the current cell or the closest other cluster.

Value

A DataFrame with one row per cell in `x` and the columns:

- `width`, a numeric field containing the approximate silhouette width of the current cell.
- `other`, the closest cluster other than the one to which the current cell is assigned.

Row names are defined as the row names of `x`.

Author(s)

Aaron Lun

See Also

`silhouette` from the **cluster** package, for the exact calculation.

`neighborPurity`, for another method of evaluating cluster separation.

Examples

```
m <- matrix(rnorm(10000), ncol=10)
clusters <- clusterRows(m, BLUSPARAM=KmeansParam(5))
out <- approxSilhouette(m, clusters)
boxplot(split(out$width, clusters))

# Mocking up a stronger example:
centers <- matrix(rnorm(30), nrow=3)
clusters <- sample(1:3, 1000, replace=TRUE)

y <- centers[clusters,]
y <- y + rnorm(length(y), sd=0.1)

out2 <- approxSilhouette(y, clusters)
boxplot(split(out2$width, clusters))
```

BlusterParam-class *The BlusterParam class*

Description

The BlusterParam class is a virtual base class controlling S4 dispatch in `clusterRows` and friends. Concrete subclasses specify the choice of clustering algorithm, while the slots of an instance of such a subclass represent the parameters for that algorithm.

Available methods

In the following code snippets, `x` is a BlusterParam object or one of its subclasses.

- `x[[i]]` will return the value of the parameter `i`. Refer to the documentation for each concrete subclass for more details on the available parameters.
- `x[[i]] <-value` will set the value of the parameter `i` to `value`.
- `show(x)` will print some information about the class instance.

Author(s)

Aaron Lun

See Also

HclustParam, KmeansParam and NNGraphParam for some examples of concrete subclasses.

`bootstrapStability` *Assess cluster stability by bootstrapping*

Description

Generate bootstrap replicates and recluster on them to determine the stability of clusters with respect to sampling noise.

Usage

```
bootstrapStability(
  x,
  FUN = clusterRows,
  clusters = NULL,
  iterations = 20,
  average = c("median", "mean"),
  ...,
  compare = NULL,
  mode = "ratio",
  adjusted = TRUE,
  transposed = FALSE
)
```

Arguments

`x`	A numeric matrix-like object containing observations in the rows and variables in the columns. If `transposed=TRUE`, observations are assumed to be in the columns instead.
`FUN`	A function that takes `x` as its first argument and returns a vector or factor of cluster identities.
`clusters`	A vector or factor of cluster identities equivalent to that obtained by calling `FUN(x,...)`. This is provided as an additional argument in the case that the clusters have already been computed, in which case we can save a single round of computation.
`iterations`	A positive integer scalar specifying the number of bootstrap iterations.
`average`	String specifying the method to use to average across bootstrap iterations.
`...`	Further arguments to pass to `FUN` to control the clustering procedure.
`compare`	A function that accepts the original clustering and the bootstrapped clustering, and returns a numeric vector or matrix containing some measure of similarity between them - see Details.
`mode, adjusted`	Further arguments to pass to `pairwiseRand` when `compare=NULL`.
`transposed`	Logical scalar indicating that resampling should be done on the columns instead.

Details

Bootstrapping is conventionally used to evaluate the precision of an estimator by applying it to an *in silico*-generated replicate dataset. We can (ab)use this framework to determine the stability of the clusters given the original dataset. We sample observations with replacement from `x`, perform clustering with `FUN` and compare the new clusters to `clusters`.

For comparing clusters, we compute the ratio matrix from `pairwiseRand` and average its values across bootstrap iterations. High on-diagonal values indicate that the corresponding cluster remains coherent in the bootstrap replicates, while high off-diagonal values indicate that the corresponding pair of clusters are still separated in the replicates. If a single value is necessary, we can instead average the adjusted Rand indices across iterations with `mode="index"`.

We use the ratio matrix by default as it is more interpretable than a single value like the ARI or the Jaccard index (see the **fpc** package). It focuses on the relevant differences between clusters, allowing us to determine which aspects of a clustering are stable. For example, A and B may be well separated but A and C may not be, which is difficult to represent in a single stability measure for A. If our main interest lies in the A/B separation, we do not want to be overly pessimistic about the stability of A, even though it might not be well-separated from all other clusters.

Value

If `compare=NULL` and `mode="ratio"`, a numeric matrix is returned with upper triangular entries set to the ratio of the adjusted observation pair counts (see `?pairwiseRand`) for each pair of clusters in `clusters`. Each ratio is averaged across bootstrap iterations as specified by `average`.

If `compare=NULL` and `mode="index"`, a numeric scalar containing the average ARI between `clusters` and the bootstrap replicates across iterations is returned.

If `compare` is provided, a numeric array of the same type as the output of `compare` is returned, containing the average statistic(s) across bootstrap replicates.

Using another comparison function

We can use a different method for comparing clusterings by setting `compare`. This is expected to be a function that takes two arguments - the original clustering first, and the bootstrapped clustering second - and returns some kind of numeric scalar, vector or matrix containing statistics for the similarity or difference between the original and bootstrapped clustering. These statistics are then averaged across all bootstrap iterations.

Any numeric output of `compare` is acceptable as long as the dimensions are only dependent on the *levels* of the original clustering - including levels that have no observations, due to resampling! - and thus do not change across bootstrap iterations.

Statistical note on bootstrap comparisons

Technically speaking, some mental gymnastics are required to compare the original and bootstrap clusters in this manner. After bootstrapping, the sampled observations represent distinct entities from the original dataset (otherwise it would be difficult to treat them as independent replicates) for which the original clusters do not immediately apply. Instead, we assume that we perform label transfer using a nearest-neighbors approach - which, in this case, is the same as using the original label for each observation, as the nearest neighbor of each resampled observation to the original dataset is itself.

Needless to say, bootstrapping will only generate replicates that differ by sampling noise. Real replicates will differ due to composition differences, variability in expression across individuals, etc. Thus, any stability inferences from bootstrapping are likely to be overly optimistic.

Author(s)

Aaron Lun

See Also

clusterRows, for the default clustering function.

pairwiseRand, for the calculation of the ARI.

Examples

```
m <- matrix(runif(10000), ncol=10)

# BLUSPARAM just gets passed to the default FUN=clusterRows:
bootstrapStability(m, BLUSPARAM=KmeansParam(4), iterations=10)

# Defining your own clustering function:
kFUN <- function(x) kmeans(x, 2)$cluster
bootstrapStability(m, FUN=kFUN)

# Using an alternative comparison, in this case the Rand index:
bootstrapStability(m, FUN=kFUN, compare=pairwiseRand)
```

---

clusterRows *Cluster rows of a matrix*

Description

Cluster rows of a matrix-like object with a variety of algorithms.

Usage

```
clusterRows(x, BLUSPARAM, full = FALSE)
```

Arguments

x	A numeric matrix-like object where rows represent observations and columns represent variables.
BLUSPARAM	A BlusterParam object specifying the algorithm to use.
full	Logical scalar indicating whether the full clustering statistics should be returned for each method.

Details

This generic allows users to write agile code that can use a variety of clustering algorithms. By simply changing BLUSPARAM, we can tune the clustering procedure in analysis workflows and package functions.

Value

By default, a factor of length equal to nrow(x) containing cluster assignments for each row of x.

If full=TRUE, a list is returned containing clusters, a factor as described above; and objects, an arbitrary object containing algorithm-specific statistics or intermediate objects.

Author(s)

Aaron Lun

See Also

HclustParam, KmeansParam and NNGraphParam for some examples of values for BLUSPARAM.

Examples

```
m <- matrix(runif(10000), ncol=10)

clusterRows(m, KmeansParam(10L))
clusterRows(m, HclustParam())
clusterRows(m, NNGraphParam())
```

HclustParam-class *Hierarchical clustering*

Description

Run the base hclust function on a distance matrix within clusterRows.

Usage

```
HclustParam(
  metric = "euclidean",
  method = "complete",
  cut.fun = NULL,
  cut.dynamic = FALSE,
  cut.height = NULL,
  cut.number = NULL,
  ...
)

## S4 method for signature 'ANY,HclustParam'
clusterRows(x, BLUSPARAM, full = FALSE)
```

Arguments

metric	String specifying the distance metric to use in dist.
method	String specifying the agglomeration method to use in hclust.
cut.fun	Function specifying the method to use to cut the dendrogram. The first argument of this function should be the output of hclust, and the return value should be an atomic vector specifying the cluster assignment for each observation. Defaults to cutree if cut.dynamic=FALSE and cutreeDynamic otherwise.
cut.dynamic	Logical scalar indicating whether a dynamic tree cut should be performed using the **dynamicTreeCut** package.
cut.height	Numeric scalar specifying the cut height to use for the tree cut when cut.fun=NULL. If NULL, defaults to half the tree height. Ignored if cut.number is set.

cut.number	Integer scalar specifying the number of clusters to generate from the tree cut when cut.fun=NULL.
...	Further arguments to pass to cut.fun, when cut.dynamic=TRUE or cut.fun is non-NULL.
x	A numeric matrix-like object where rows represent observations and columns represent variables.
BLUSPARAM	A HclustParam object.
full	Logical scalar indicating whether the hierarchical clustering statistics should be returned.

Details

To modify an existing HclustParam object x, users can simply call x[[i]] or x[[i]] <-value where i is any argument used in the constructor.

Value

The HclustParam constructor will return a HclustParam object with the specified parameters.

The clusterRows method will return a factor of length equal to nrow(x) containing the cluster assignments. If full=TRUE, a list is returned with clusters (the factor, as above) and objects (the hclust output).

Author(s)

Aaron Lun

See Also

dist, hclust and cutree, which actually do all the heavy lifting.

cutreeDynamic, for an alternative tree cutting method to use in cut.fun.

Examples

```
clusterRows(iris[,1:4], HclustParam())
clusterRows(iris[,1:4], HclustParam(method="ward.D2"))
```

KmeansParam-class *K-means clustering*

Description

Run the base kmeans function with the specified number of centers within clusterRows.

Usage

```
KmeansParam(centers, ...)

## S4 method for signature 'ANY,KmeansParam'
clusterRows(x, BLUSPARAM, full = FALSE)
```

Arguments

centers	An integer scalar specifying the number of centers. Alternatively, a function that takes the number of observations and returns the number of centers.
...	Further arguments to pass to kmeans.
x	A numeric matrix-like object where rows represent observations and columns represent variables.
BLUSPARAM	A KmeansParam object.
full	Logical scalar indicating whether the full k-means statistics should be returned.

Details

This class usually requires the user to specify the number of clusters beforehand. However, we can also allow the number of clusters to vary as a function of the number of observations. The latter is occasionally useful, e.g., to allow the clustering to automatically become more granular for large datasets.

To modify an existing KmeansParam object x, users can simply call x[[i]] or x[[i]] <-value where i is any argument used in the constructor.

Value

The KmeansParam constructor will return a KmeansParam object with the specified parameters.

The clusterRows method will return a factor of length equal to nrow(x) containing the cluster assignments. If full=TRUE, a list is returned with clusters (the factor, as above) and objects; the latter will contain the direct output of kmeans.

Author(s)

Aaron Lun

See Also

kmeans, which actually does all the heavy lifting.

Examples

```
clusterRows(iris[,1:4], KmeansParam(centers=4))
clusterRows(iris[,1:4], KmeansParam(centers=4, algorithm="Lloyd"))
clusterRows(iris[,1:4], KmeansParam(centers=sqrt))
```

---

makeSNNGraph *Build a nearest-neighbor graph*

Description

Build a shared or k-nearest-neighbors graph of observations for downstream community detection.

Usage

```
makeSNNGraph(
  x,
  k = 10,
  type = c("rank", "number", "jaccard"),
  BNPARAM = KmknnParam(),
  BPPARAM = SerialParam()
)

makeKNNGraph(
  x,
  k = 10,
  directed = FALSE,
  BNPARAM = KmknnParam(),
  BPPARAM = SerialParam()
)

neighborsToSNNGraph(indices, type = c("rank", "number", "jaccard"))

neighborsToKNNGraph(indices, directed = FALSE)
```

Arguments

x	A matrix-like object containing expression values for each observation (row) and dimension (column).
k	An integer scalar specifying the number of nearest neighbors to consider during graph construction.
type	A string specifying the type of weighting scheme to use for shared neighbors.
BNPARAM	A BiocNeighborParam object specifying the nearest neighbor algorithm.
BPPARAM	A BiocParallelParam object to use for parallel processing.
directed	A logical scalar indicating whether the output of buildKNNGraph should be a directed graph.
indices	An integer matrix where each row corresponds to an observation and contains the indices of the k nearest neighbors (by increasing distance and excluding self) from that observation.

Details

The makeSNNGraph method builds a shared nearest-neighbour graph using observations as nodes. For each observation, its k nearest neighbours are identified using the findKNN function, based on distances between their expression profiles (Euclidean by default). An edge is drawn between all pairs of observations that share at least one neighbour, weighted by the characteristics of the shared nearest neighbors:

- If type="rank", the weighting scheme defined by Xu and Su (2015) is used. The weight between two nodes is $k - r/2$ where r is the smallest sum of ranks for any shared neighboring node. For example, if one node was the closest neighbor of each of two nodes, the weight between the two latter nodes would be $k - 1$. For the purposes of this ranking, each node has a rank of zero in its own nearest-neighbor set. More shared neighbors, or shared neighbors that are close to both observations, will generally yield larger weights.

- If `type="number"`, the weight between two nodes is simply the number of shared nearest neighbors between them. The weight can range from zero to $k + 1$, as the node itself is included in its own nearest-neighbor set. This is a simpler scheme that is also slightly faster but does not account for the ranking of neighbors within each set.
- If `type="jaccard"`, the weight between two nodes is the Jaccard similarity between the two sets of neighbors for those nodes. This weight can range from zero to 1, and is a monotonic transformation of the weight used by `type="number"`. It is provided for consistency with other clustering algorithms such as those in **seurat**.

The aim is to use the SNN graph to perform clustering of observations via community detection algorithms in the **igraph** package. This is faster and more memory efficient than hierarchical clustering for large numbers of observations. In particular, it avoids the need to construct a distance matrix for all pairs of observations. Only the identities of nearest neighbours are required, which can be obtained quickly with methods in the **BiocNeighbors** package.

The choice of `k` controls the connectivity of the graph and the resolution of community detection algorithms. Smaller values of `k` will generally yield smaller, finer clusters, while increasing `k` will increase the connectivity of the graph and make it more difficult to resolve different communities. The value of `k` can be roughly interpreted as the anticipated size of the smallest subpopulation. If a subpopulation in the data has fewer than `k+1` observations, `buildSNNGraph` and `buildKNNGraph` will forcibly construct edges between observations in that subpopulation and observations in other subpopulations. This increases the risk that the subpopulation will not form its own cluster as it is more interconnected with the rest of the observations in the dataset.

Note that the setting of `k` here is slightly different from that used in SNN-Cliq. The original implementation considers each observation to be its first nearest neighbor that contributes to `k`. In `buildSNNGraph`, the `k` nearest neighbours refers to the number of *other* observations.

The `makeKNNGraph` method builds a simpler k-nearest neighbour graph. Observations are again nodes, and edges are drawn between each observation and its k-nearest neighbours. No weighting of the edges is performed. In theory, these graphs are directed as nearest neighour relationships may not be reciprocal. However, by default, `directed=FALSE` such that an undirected graph is returned.

The `neighborsToSNNGraph` and `neighborsToKNNGraph` functions operate directly on a matrix of nearest neighbor indices, obtained using functions like `findKNN`. This may be useful for constructing a graph from precomputed nearest-neighbor search results. Note that the user is responsible for ensuring that the indices are valid (i.e., `range(indices)` is positive and no greater than `max(indices)`).

Value

A graph where nodes are cells and edges represent connections between nearest neighbors. For `buildSNNGraph`, these edges are weighted by the number of shared nearest neighbors. For `buildKNNGraph`, edges are not weighted but may be directed if `directed=TRUE`.

Author(s)

Aaron Lun, with KNN code contributed by Jonathan Griffiths.

References

Xu C and Su Z (2015). Identification of cell types from single-cell transcriptomes using a novel clustering method. *Bioinformatics* 31:1974-80

See Also

See make_graph for details on the graph output object.

See cluster_walktrap, cluster_louvain and related functions in **igraph** for clustering based on the produced graph.

Also see findKNN for specifics of the nearest-neighbor search.

Examples

```
m <- matrix(rnorm(10000), ncol=10)

g <- makeSNNGraph(m)
clusters <- igraph::cluster_fast_greedy(g)$membership
table(clusters)

# Any clustering method from igraph can be used:
clusters <- igraph::cluster_walktrap(g)$membership
table(clusters)

# Smaller 'k' usually yields finer clusters:
g <- makeSNNGraph(m, k=5)
clusters <- igraph::cluster_walktrap(g)$membership
table(clusters)
```

---

mergeCommunities *Merge communities from graph-based clustering*

Description

Adjust the resolution of a graph-based community detection algorithm by greedily merging clusters together. At each step, the pair of clusters that yield the highest modularity are merged.

Usage

```
mergeCommunities(graph, clusters, number = NULL, steps = NULL)
```

Arguments

graph	A graph object from **igraph**, usually where each node represents an observation.
clusters	Factor specifying the cluster identity for each node.
number	Integer scalar specifying the number of clusters to obtain. Ignored if steps is specified.
steps	Integer scalar specifying the number of merge steps.

Details

This function is similar to the cut_at function from the **igraph** package, but works on clusters that were not generated by a hierarchical algorithm. The aim is to facilitate rapid adjustment of the number of clusters without having to repeat the clustering - or, even worse, repeating the graph construction, e.g., in makeSNNGraph.

Value

A vector or factor of the same length as `clusters`, containing the desired number of merged clusters.

Author(s)

Aaron Lun

See Also

`cut_at`, for a faster and more natural adjustment when using a hierarchical community detection algorithm.

NNGraphParam, for a one-liner to generate graph-based clusters.

Examples

```
output <- clusterRows(iris[,1:4], NNGraphParam(k=5), full=TRUE)
table(output$clusters)

merged <- mergeCommunities(output$objects$graph, output$clusters, number=3)
table(merged)
```

---

neighborPurity *Compute neighborhood purity*

Description

Use a hypersphere-based approach to compute the "purity" of each cluster based on the number of contaminating observations from different clusters in its neighborhood.

Usage

```
neighborPurity(
  x,
  clusters,
  k = 50,
  weighted = TRUE,
  BNPARAM = KmknnParam(),
  BPPARAM = SerialParam()
)
```

Arguments

`x`	A numeric matrix-like object containing observations in rows and variables in columns.
`clusters`	Vector of length equal to `ncol(x)`, specifying the cluster assigned to each observation.
`k`	Integer scalar specifying the number of nearest neighbors to use to determine the radius of the hyperspheres.

weighted	A logical scalar indicating whether to weight each observation in inverse proportion to the size of its cluster. Alternatively, a numeric vector of length equal to clusters containing the weight to use for each observation.
BNPARAM	A BiocNeighborParam object specifying the nearest neighbor algorithm. This should be an algorithm supported by findNeighbors.
BPPARAM	A BiocParallelParam object indicating whether and how parallelization should be performed across genes.

Details

The purity of a cluster is quantified by creating a hypersphere around each observation in the cluster and computing the proportion of observations in that hypersphere from the same cluster. If all observations in a cluster have proportions close to 1, this indicates that the cluster is highly pure, i.e., there are few observations from other clusters in its region of the coordinate space. The distribution of purities for each cluster can be used as a measure of separation from other clusters.

In most cases, the majority of observations of a cluster will have high purities, corresponding to observations close to the cluster center. A fraction of observations will have low values as these lie at the boundaries of two adjacent clusters. A high degree of over-clustering will manifest as a majority of observations with purities close to zero. The maximum field in the output can be used to determine the identity of the cluster with the greatest presence in a observation's neighborhood, usually an adjacent cluster for observations lying on the boundary.

The choice of k is used only to determine an appropriate value for the hypersphere radius. We use hyperspheres as this is robust to changes in density throughout the coordinate space, in contrast to computing purity based on the proportion of k-nearest neighbors in the same cluster. For example, the latter will fail most obviously when the size of the cluster is less than k.

Value

A DataFrame with one row per observation in x and the columns:

- purity, a numeric field containing the purity value for the current observation.
- maximum, the cluster with the highest proportion of observations neighboring the current observation.

Row names are defined as the row names of x.

Weighting by frequency

By default, purity values are computed after weighting each observation by the reciprocal of the number of observations in the same cluster. Otherwise, clusters with more observations will have higher purities as any contamination is offset by the bulk of observations, which would compromise comparisons of purities between clusters. One can interpret the weighted purities as the expected value after downsampling all clusters to the same size.

Advanced users can achieve greater control by manually supplying a numeric vector of weights to weighted. For example, we may wish to check the purity of batches after batch correction in single-cell RNA-seq. In this application, clusters should be set to the *batch blocking factor* (not the cluster identities!) and weighted should be set to 1 over the frequency of each combination of cell type and batch. This accounts for differences in cell type composition between batches when computing purities.

If weighted=FALSE, no weighting is performed.

Author(s)

Aaron Lun

Examples

```
m <- matrix(runif(1000), ncol=10)
clusters <- clusterRows(m, BLUSPARAM=NNGraphParam())
out <- neighborPurity(m, clusters)
boxplot(split(out$purity, clusters))

# Mocking up a stronger example:
centers <- matrix(rnorm(30), nrow=3)
clusters <- sample(1:3, 1000, replace=TRUE)
y <- centers[clusters,,drop=FALSE]
y <- y + rnorm(length(y))

out2 <- neighborPurity(y, clusters)
boxplot(split(out2$purity, clusters))
```

NNGraphParam-class *Graph-based clustering*

Description

Run community detection algorithms on a nearest-neighbor (NN) graph within clusterRows.

Usage

```
NNGraphParam(
  shared = TRUE,
  ...,
  cluster.fun = "walktrap",
  cluster.args = list()
)

## S4 method for signature 'ANY,NNGraphParam'
clusterRows(x, BLUSPARAM, full = FALSE)
```

Arguments

shared	Logical scalar indicating whether a shared NN graph should be constructed.
...	Further arguments to pass to makeSNNGraph (if shared=TRUE) or makeKNNGraph.
cluster.fun	Function specifying the method to use to detect communities in the NN graph. The first argument of this function should be the NN graph and the return value should be a communities object. Alternatively, this may be a string containing the suffix of any **igraph** community detection algorithm. For example, cluster.fun="louvain" will instruct clusterRows to use cluster_louvain. Defaults to cluster_walktrap.
cluster.args	Further arguments to pass to the chosen cluster.fun.

x	A numeric matrix-like object where rows represent observations and columns represent variables.
BLUSPARAM	A NNGraphParam object.
full	Logical scalar indicating whether the graph-based clustering objects should be returned.

Details

To modify an existing NNGraphParam object x, users can simply call x[[i]] or x[[i]] <-value where i is any argument used in the constructor.

Value

The NNGraphParam constructor will return a NNGraphParam object with the specified parameters.

The clusterRows method will return a factor of length equal to nrow(x) containing the cluster assignments. If full=TRUE, a list is returned with clusters (the factor, as above) and objects; the latter is a list with graph (the graph) and communities (the output of cluster.fun).

Author(s)

Aaron Lun

See Also

makeSNNGraph and related functions, to build the graph.

cluster_walktrap and related functions, to perform community detection.

Examples

```
clusterRows(iris[,1:4], NNGraphParam())
clusterRows(iris[,1:4], NNGraphParam(k=5))
clusterRows(iris[,1:4], NNGraphParam(cluster.fun="louvain"))
```

---

pairwiseModularity *Compute pairwise modularity*

Description

Calculate the modularity of each pair of clusters from a graph, based on a null model of random connections between nodes.

Usage

```
pairwiseModularity(graph, clusters, get.weights = FALSE, as.ratio = FALSE)
```

Arguments

graph	A graph object from **igraph**, usually where each node represents an observation.
clusters	Factor specifying the cluster identity for each node.
get.weights	Logical scalar indicating whether the observed and expected edge weights should be returned, rather than the modularity.
as.ratio	Logical scalar indicating whether the log-ratio of observed to expected weights should be returned.

Details

This function computes a modularity score in the same manner as that from modularity. The modularity is defined as the (scaled) difference between the observed and expected number of edges between nodes in the same cluster. The expected number of edges is defined by a null model where edges are randomly distributed among nodes. The same logic applies for weighted graphs, replacing the number of edges with the summed weight of edges.

Whereas modularity returns a modularity score for the entire graph, pairwiseModularity provides scores for the individual clusters. The sum of the diagonal elements of the output matrix should be equal to the output of modularity (after supplying weights to the latter, if necessary). A well-separated cluster should have mostly intra-cluster edges and a high modularity score on the corresponding diagonal entry, while two closely related clusters that are weakly separated will have many inter-cluster edges and a high off-diagonal score.

In practice, the modularity may not the most effective metric for evaluating cluster separatedness. This is because the modularity is proportional to the number of observations, so larger clusters will naturally have a large score regardless of separation. An alternative approach is to set as.ratio=TRUE, which returns the ratio of the observed to expected weights for each entry of the matrix. This adjusts for differences in cluster size and improves resolution of differences between clusters.

Directed graphs are treated as undirected inputs with mode="each" in as.undirected. In the rare case that self-loops are present, these will also be handled correctly.

Value

By default, an upper triangular numeric matrix of order equal to the number of clusters is returned. Each entry corresponds to a pair of clusters and is proportional to the difference between the observed and expected edge weights between those clusters.

If as.ratio=TRUE, an upper triangular numeric matrix is again returned. Here, each entry is equal to the ratio between the observed and expected edge weights.

If get.weights=TRUE, a list is returned containing two upper triangular numeric matrices. The observed matrix contains the observed sum of edge weights between and within clusters, while the expected matrix contains the expected sum of edge weights under the random model.

Author(s)

Aaron Lun

See Also

makeSNNGraph, for one method to construct graph.

modularity, for the calculation of the entire graph modularity.

pairwiseRand, which applies a similar breakdown to the Rand index.

Examples

```
m <- matrix(runif(10000), ncol=10)
clust.out <- clusterRows(m, BLUSPARAM=NNGraphParam(), full=TRUE)
clusters <- clust.out$clusters
g <- clust.out$objects$graph

# Examining the modularity values directly.
out <- pairwiseModularity(g, clusters)
out

# Compute the ratio instead, for visualization
# (log-transform to improve range of colors).
out <- pairwiseModularity(g, clusters, as.ratio=TRUE)
image(log2(out+1))

# This can also be used to construct a graph of clusters,
# for use in further plotting, a.k.a. graph abstraction.
# (Fiddle with the scaling values for a nicer plot.)
g2 <- igraph::graph_from_adjacency_matrix(out, mode="upper",
    diag=FALSE, weighted=TRUE)
plot(g2, edge.width=igraph::E(g2)$weight*10,
    vertex.size=sqrt(table(clusters))*2)

# Alternatively, get the edge weights directly:
out <- pairwiseModularity(g, clusters, get.weights=TRUE)
out
```

`pairwiseRand` *Compute pairwise Rand indices*

Description

Breaks down the Rand index calculation to report values for each cluster and pair of clusters in a reference clustering compared to an alternative clustering.

Usage

```
pairwiseRand(ref, alt, mode = c("ratio", "pairs", "index"), adjusted = TRUE)
```

Arguments

`ref`	A character vector or factor containing one set of groupings, considered to be the reference.
`alt`	A character vector or factor containing another set of groupings, to be compared to `alt`.
`mode`	String indicating whether to return the ratio, the number of pairs or the Rand index.
`adjusted`	Logical scalar indicating whether the adjusted Rand index should be returned.

Details

Recall that the Rand index calculation consists of four numbers:

a The number of pairs of cells in the same cluster in `ref` and the same cluster in `alt`.

b The number of pairs of cells in different clusters in `ref` and different clusters in `alt`.

c The number of pairs of cells in the same cluster in `ref` and different clusters in `alt`.

d The number of pairs of cells in different clusters in `ref` but the same cluster in `alt`.

The Rand index is then computed as $a + b$ divided by $a + b + c + d$, i.e., the total number of pairs.

We can break these numbers down into values for each cluster or pair of clusters in `ref`. For each cluster, we compute its value of a, i.e., the number of pairs of cells in *that* cluster that are also in the same cluster in `alt`. Similarly, for each pair of clusters in `ref`, we compute its value of b, i.e., the number of pairs of cells that have one cell in each of those clusters and also belong in different clusters in `alt`.

This process provides more information about the specific similarities or differences between `ref` and `alt`, rather than coalescing all the values into a single statistic. For example, it is now possible to see which specific clusters from `ref` are not reproducible in `alt`, or which specific partitions between pairs of clusters are not reproducible. In the default output, such events can be diagnosed by looking for low entries in the ratio matrix; on the other hand, values close to 1 indicate that `ref` is almost perfectly recapitulated by `alt`.

If `adjusted=TRUE`, we adjust all counts by subtracting their expected values under a model of random permutations. This accounts for differences in the number and sizes of clusters within and between `ref` and `alt`, in a manner that mimics the calculation of adjusted Rand index (ARI). We subtract expectations on a per-cluster or per-cluster-pair basis for a and b, respectively; we also redefine the "total" number of cell pairs for each cluster or cluster pair based on the denominator of the ARI.

Value

If `mode="ratio"`, a square numeric matrix is returned with number of rows equal to the number of unique levels in `ref`. Each diagonal entry is the ratio of the per-cluster a to the total number of pairs of cells in that cluster. Each off-diagonal entry is the ratio of the per-cluster-pair b to the total number of pairs of cells for that pair of clusters. Lower-triangular entries are set to `NA`. If `adjusted=TRUE`, counts and totals are both adjusted prior to computing the ratio.

If `mode="pairs"`, a list is returned containing `correct` and `total`, both of which are square numeric matrices of the same arrangement as described above. However, `correct` contains the actual numbers a (diagonal) and b (off-diagonal) rather than the ratios, while `total` contains the total number of cell pairs in each cluster or pair of clusters. If `adjusted=TRUE`, both matrices are adjusted by subtracting the random expectations from the counts.

If `mode="index"`, a numeric scalar is returned containing the Rand index (or ARI, if `adjusted=TRUE`).

Author(s)

Aaron Lun

See Also

`pairwiseModularity`, which applies the same breakdown to the cluster modularity.

Examples

```
m <- matrix(runif(10000), ncol=10)

clust1 <- kmeans(m,3)$cluster
clust2 <- kmeans(m,5)$cluster

ratio <- pairwiseRand(clust1, clust2)
ratio

# Getting the raw counts:
pairwiseRand(clust1, clust2, mode="pairs")

# Computing the original Rand index.
pairwiseRand(clust1, clust2, mode="index")
```

TwoStepParam-class *Two step clustering with vector quantization*

Description

For large datasets, we can perform vector quantization (e.g., with k-means clustering) to create centroids. These centroids are then subjected to a slower clustering technique such as graph-based community detection. The label for each cell is set to the label of the centroid to which it was assigned.

Usage

```
TwoStepParam(first = KmeansParam(centers = sqrt), second = NNGraphParam())

## S4 method for signature 'ANY,TwoStepParam'
clusterRows(x, BLUSPARAM, full = FALSE)
```

Arguments

first	A BlusterParam object specifying a fast vector quantization technique.
second	A BlusterParam object specifying the second clustering technique on the centroids.
x	A numeric matrix-like object where rows represent observations and columns represent variables.
BLUSPARAM	A KmeansParam object.
full	Logical scalar indicating whether the clustering statistics from both steps should be returned.

Details

Here, the idea is to use a fast clustering algorithm to perform vector quantization and reduce the size of the dataset, followed by a slower algorithm that aggregates the centroids for easier interpretation. The exact choice of the number of clusters is less relevant to the first clustering step as long as not too many centroids are generated but the clusters are still sufficiently granular. The second step

can take more care (and computational time) summarizing the centroids into meaningful "meta-clusters".

The default choice is to use k-means for the first step, with number of clusters set to the root of the number of observations; and graph-based clustering for the second step, which automatically detects a suitable number of clusters. K-means also eliminates density differences in the data that can introduce variable resolution from graph-based methods.

To modify an existing TwoStepParam object `x`, users can simply call `x[[i]]` or `x[[i]] <- value` where `i` is any argument used in the constructor.

Value

The `TwoStepParam` constructor will return a TwoStepParam object with the specified parameters.

The `clusterRows` method will return a factor of length equal to `nrow(x)` containing the cluster assignments. If `full=TRUE`, a list is returned with a `clusters` factor and an `objects` list containing:

- `first`, a list of objects from the first clustering step. This is equal to the `objects` list in the output of `clusterRows` with the `first` BlusterParam.
- `centroids`, a numeric matrix of centroids generated from the first clustering step.
- `second`, a list of objects from the second clustering step on the centroids. This is equal to the `objects` list in the output of `clusterRows` with the `second` BlusterParam.

Author(s)

Aaron Lun

Examples

```
m <- matrix(runif(100000), ncol=10)
stuff <- clusterRows(m, TwoStepParam())
table(stuff)
```

Index