Package 'bluster'

October 15, 2023

Version 1.10.0

Date 2023-01-12

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, cluster, Matrix, Rcpp, igraph, S4Vectors, BiocParallel, BiocNeighbors

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

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

LinkingTo Rcpp

Collate AllClasses.R AllGenerics.R AgnesParam.R approxSilhouette.R bluster-package.R DbscanParam.R DianaParam.R AffinityParam.R BlusterParam.R bootstrapStability.R ClaraParam.R clusterRMSD.R clusterSweep.R compareClusterings.R FixedNumberParam.R HclustParam.R HierarchicalParam.R KmeansParam.R linkClusters.R makeSNNGraph.R MbkmeansParam.R mergeCommunities.R neighborPurity.R nestedClusters.R NNGraphParam.R pairwiseModularity.R pairwiseRand.R PamParam.R RcppExports.R SomParam.R TwoStepParam.R utils.R

License GPL-3

NeedsCompilation yes

VignetteBuilder knitr

SystemRequirements C++11

RoxygenNote 7.2.3

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

git_branch RELEASE_3_17

git_last_commit 3234042

git_last_commit_date 2023-04-25

Date/Publication 2023-10-15

Author Aaron Lun [aut, cre],
Stephanie Hicks [ctb]

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

R topics documented:

`bluster-package` *bluster: 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.

Author(s)

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

Other contributors:

- Stephanie Hicks [contributor]

`.defaultScalarArguments` *Define the default arguments*

Description

Provide a consistent mechanism to handle specification of default arguments to the underlying clustering functions.

Usage

```
.defaultScalarArguments(x)

.showScalarArguments(object)

.extractScalarArguments(x)
```

Arguments

`x, object`	A BlusterParam object.

Details

The idea is to simplify the derivation of new BlusterParam objects, by allowing developers to indicate that the underlying function default should be used for particular arguments. This avoids duplication of the default arguments in the object constructor; instead, default arguments can be indicated as such by setting them to `NULL`, in which case they will not be explicitly passed to the underlying clustering function.

Value

For `.defaultScalarArguments`, a named character vector is returned. Each entry corresponds to an argument to the clustering function - the name is the argument name, and the value is the argument type.

For `.extractScalarArguments`, a named list of non-default scalar arguments is returned. Any arguments set to their default values are omitted from the list.

For `.showScalarArguments`, the values of the arguments are printed to screen. Default values are marked with `[default]`.

Author(s)

Aaron Lun

Examples

```
.defaultScalarArguments(PamParam(10))
.extractScalarArguments(PamParam(10))
.extractScalarArguments(PamParam(10, variant="faster"))
```

`AffinityParam-class` *Affinity propogation*

Description

Use affinity propagation from the **apcluster** package to cluster observations. Note that this requires the installation of the **apcluster** package.

Usage

```
AffinityParam(
  s = NULL,
  p = NA,
  q = NA,
  maxits = 1000,
  convits = 100,
  lam = 0.9,
  nonoise = FALSE
)

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

Arguments

s	A function that accepts a matrix of observations by dimensions and returns a similarity matrix. If NULL, defaults to the output of negDistMat with r=2.
p, q	Numeric scalars controlling the input preference, i.e., the resolution of the clustering. These are passed to the apcluster function, where values of NA are the default.
maxits, convits, lam, nonoise	Further arguments to pass to the apcluster function.
x	A numeric matrix-like object where rows represent observations and columns represent variables.
BLUSPARAM	A AffinityParam object.
full	Logical scalar indicating whether the full affinity propagation statistics should be returned.

Details

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

Setting q (and less typically, p) allows us to tune the resolution of the clustering. In particular, when p=NA, it is computed based on the setting of q:

- If the specified q lies in [0, 1], p is defined as the q-quantile of the finite similarities across all pairs of observations. When q=NA, it defaults to 0.5.
- If q is negative, p is defined as the M + abs(M) * q where M is the smallest finite similarity across all pairs. This yields smaller p values while still responding to the scale of the similarities.

The resulting value is used as the self-preference, i.e., the diagonal of the availability matrix. Larger values yield more clusters as each data point is more inclined to form its own cluster.

Value

The AffinityParam constructor will return a AffinityParam 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 (a list containing similarity, the similarity matrix; and apcluster, the direct output of apcluster).

Author(s)

Aaron Lun

See Also

apcluster from the **apcluster** package, which does all of the heavy lifting.

Examples

```
clusterRows(iris[,1:4], AffinityParam())
clusterRows(iris[,1:4], AffinityParam(q=0.9))
clusterRows(iris[,1:4], AffinityParam(s=apcluster::expSimMat()))
```

AgnesParam-class *Agglomerative nesting*

Description

Run the agnes function on a distance matrix within clusterRows.

Usage

```
AgnesParam(
  metric = NULL,
  stand = NULL,
  method = NULL,
  par.method = NULL,
  cut.fun = NULL,
  cut.dynamic = FALSE,
  cut.params = list()
)

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

Arguments

metric, stand, method, par.method	Further arguments to pass to agnes.
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.params	Further arguments to pass to cut.fun.
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 AgnesParam object `x`, users can simply call `x[[i]]` or `x[[i]] <- value` where `i` is any argument used in the constructor.

If `cut.fun=NULL`, `cut.dynamic=FALSE` and `cut.params` does not have `h` or `k`, `clusterRows` will automatically set `h` to half the tree height when calling `cutree`.

Value

The `AgnesParam` constructor will return a AgnesParam 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` (a list containing `agnes`, the function output; `dist`, the dissimilarity matrix; and `hclust`, a hclust object created from `agnes`).

Author(s)

Aaron Lun

See Also

`agnes`, which actually does all the heavy lifting.

HclustParam, for the more commonly used implementation of hierarchical clustering.

Examples

```
clusterRows(iris[,1:4], AgnesParam())
clusterRows(iris[,1:4], AgnesParam(method="ward"))
```

---

approxSilhouette *Approximate silhouette width*

Description

Given a clustering, quickly compute an approximate silhouette width for each observation.

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 distance with the root-mean-squared-distance, which can be computed very efficiently for large datasets. The approximated averages are then used to compute the silhouette width using the usual definition.

Value

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

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

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)
```

ClaraParam-class *Clustering Large Applications*

Description

Run the CLARA algorithm, an extension of the PAM method for large datasets.

Usage

```
ClaraParam(
  centers,
  metric = NULL,
  stand = NULL,
  samples = NULL,
  sampsize = NULL
)

## S4 method for signature 'ANY,ClaraParam'
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.
metric, stand, samples, sampsize	Further arguments to pass to clara. Set to the function defaults if not supplied.
x	A numeric matrix-like object where rows represent observations and columns represent variables.
BLUSPARAM	A ClaraParam object.
full	Logical scalar indicating whether the full PAM 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 ClaraParam object x, users can simply call x[[i]] or x[[i]] <- value where i is any argument used in the constructor.

Note that clusterRows will always use rngR=TRUE, for greater consistency with other algorithms of the FixedNumberParam class; and pamLike=TRUE, for consistency with the PAM implementation from which it was derived.

Value

The `ClaraParam` constructor will return a ClaraParam 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` (a list containing `clara`, the direct output of `clara`).

Author(s)

Aaron Lun

See Also

`clara`, which actually does all the heavy lifting.

PamParam, for the original PAM algorithm.

Examples

```
clusterRows(iris[,1:4], ClaraParam(centers=4))
clusterRows(iris[,1:4], ClaraParam(centers=4, sampsize=50))
clusterRows(iris[,1:4], ClaraParam(centers=sqrt))
```

---

clusterRMSD *Compute the RMSD per cluster*

Description

Compute the root mean-squared deviation (RMSD) for each cluster.

Usage

```
clusterRMSD(x, clusters, sum = FALSE)
```

Arguments

`x`	Numeric matrix containing observations in rows and variables in columns.
`clusters`	Vector containing the assigned cluster for each observation.
`sum`	Logical scalar indicating whether to compute the sum of squares.

Details

The RMSD for each cluster is a measure of its dispersion; clusters with large internal heterogeneity will have high RMSDs and are good candidates for further subclustering.

Value

Numeric vector of RMSD values per cluster. If `sum=TRUE`, a numeric vector of the sum of squares per cluster is returned instead.

Author(s)

Aaron Lun

Examples

```
x <- matrix(rnorm(10000), ncol=10)
kout <- kmeans(x, 5)
clusterRMSD(x, kout$cluster)
```

`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())
```

---

clusterSweep *Clustering parameter sweeps*

Description

Perform a sweep across combinations of parameters to obtain different clusterings from the same algorithm.

Usage

```
clusterSweep(
  x,
  BLUSPARAM,
  ...,
  full = FALSE,
  BPPARAM = SerialParam(),
  args = list()
)
```

Arguments

x	A numeric matrix-like object where rows represent observations and columns represent variables.
BLUSPARAM	A BlusterParam object specifying the algorithm to use.
...	Named vectors or lists specifying the parameters to sweep over.
full	Logical scalar indicating whether the full clustering statistics should be returned for each method.
BPPARAM	A BiocParallelParam specifying how the sweep should be parallelized.
args	A named list of additional arguments to use with .... This is provided in case there is a name conflict with the existing arguments in this function signature.

Details

This function allows users to conveniently test out a range of clustering parameters in a single call. The name of each argument in ... should be a legitimate argument to x[[i]], and will be used to modify any existing values in BLUSPARAM to obtain a new set of parameters. (For all other parameters, the existing values in BLUSPARAM are used.) If multiple arguments are provided, all combinations are tested.

We attempt to create a unique name for each column based on its parameter combination. This has the format of `<NAME1>.<VALUE1>_<NAME2>.<VALUE2>_...` based on the parameter names and values. Note that any non-atomic values are simply represented by the name of their class; no attempt is made to convert these into a compact string.

If an entry of `...` is a *named* list of vectors, we expand those to generate all possible combinations of values. For example, if we passed:

```
blah.args = list(a = 1:5, b = LETTERS[1:3])
```

This would be equivalent to manually specifying:

```
blah.args = list(list(a = 1, b = "A"), list(a = 1, b = "B"), ...)
```

The auto-expansion mechanism allows us to conveniently test parameter combinations when those parameters are stored inside `x` as a list. The algorithm is recursive so any internal named lists containing vectors are similarly expanded. Expansion can be disabled by wrapping vectors in `I`, in which case they are passed verbatim. No expansion is performed for non-vector arguments.

Value

A List containing:

- `clusters`, a DataFrame with number of rows equal to that of `x`, where each column corresponds to (and is named after) a specific combination of clustering parameters.
- `parameters`, another DataFrame with number of rows equal to the number of columns in the previous `clusters` DataFrame. Each row contains the specific parameter combination for each column of `clusters`.
- If `full=TRUE`, `objects` is an additional list of length equal to the number of rows in `clusters`. This contains the `objects` produced by each run.

Author(s)

Aaron Lun

See Also

`clusterRows`, which manages the dispatch to specific methods based on `BLUSPARAM`.

BlusterParam, which determines which algorithm is actually used.

Examples

```
out <- clusterSweep(iris[,1:4], KmeansParam(10),
    centers=4:10, algorithm=c("Lloyd", "Hartigan-Wong"))
out$clusters[,1:5]
out$parameters

out <- clusterSweep(iris[,1:4], NNGraphParam(), k=c(5L, 10L, 15L, 20L),
    cluster.fun=c("louvain", "walktrap"))
out$clusters[,1:5]
```

```
out$parameters

# Combinations are automatically expanded inside named lists:
out <- clusterSweep(iris[,1:4], NNGraphParam(), k=c(5L, 10L, 15L, 20L),
    cluster.args=list(steps=3:4))
out$clusters[,1:5]
out$parameters
```

`compareClusterings` *Compare pairs of clusterings*

Description

Compute the adjusted Rand index between all pairs of clusterings, where larger values indicate a greater similarity between clusterings.

Usage

```
compareClusterings(clusters, adjusted = TRUE)
```

Arguments

`clusters`	A list of factors or vectors where each entry corresponds to a clustering. All vectors should be of the same length. The list itself should usually be named with a suitable label for each clustering.
`adjusted`	Logical scalar indicating whether the adjusted Rand index should be returned.

Details

The aim of this function is to allow us to easily determine the relationships between clusterings. For example, we might use this to determine which parameter settings have the greatest effect in a sweep by `clusterSweep`. Alternatively, we could use this to obtain an "ordering" of clusterings for visualization, e.g., with **clustree**.

This function does not provide any insight into the relationships between individual clusters. A large Rand index only means that two clusterings are similar but does not specify the corresponding set of clusters across clusterings. For that task, we suggest using the `linkClusters` function instead.

Value

A symmetric square matrix of pairwise (adjusted) Rand indices between all pairs of clusters.

Aaron Lun

See Also

`linkClusters`, which identifies relationships between individual clusters across clusterings.

`pairwiseRand`, for calculation of the pairwise Rand index.

Examples

```
clusters <- list(
    nngraph = clusterRows(iris[,1:4], NNGraphParam()),
    hclust = clusterRows(iris[,1:4], HclustParam(cut.dynamic=TRUE)),
    kmeans = clusterRows(iris[,1:4], KmeansParam(20))
)

aris <- compareClusterings(clusters)

# Visualizing the relationships between clusterings.
# Here, k-means is forced to be least similar to the others.
ari.as.graph <- igraph::graph.adjacency(aris, mode="undirected", weighted=TRUE)
plot(ari.as.graph)

# Obtain an ordering of clusterings, using the eigenvector
# as a 1-dimensional summary of the matrix:
ev1 <- eigen(aris)$vectors[,1]
o <- order(ev1)
rownames(aris)[o]
```

DbscanParam-class *Density-based clustering with DBSCAN*

Description

Perform density-based clustering with a fast re-implementation of the DBSCAN algorithm.

Usage

```
DbscanParam(
  eps = NULL,
  min.pts = 5,
  core.prop = 0.5,
  chunk.size = 1000,
  BNPARAM = KmknnParam(),
  BPPARAM = SerialParam()
)

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

Arguments

eps	Numeric scalar specifying the distance to use to define neighborhoods. If NULL, this is determined from min.pts and core.prop.
min.pts	Integer scalar specifying the minimum number of neighboring observations required for an observation to be a core point.

core.prop	Numeric scalar specifying the proportion of observations to treat as core points. This is only used when eps=NULL, see Details.
chunk.size	Integer scalar specifying the number of points to process per chunk.
BNPARAM	A BiocNeighborParam object specifying the algorithm to use for the neighbor searches. This should be able to support both nearest-neighbor and range queries.
BPPARAM	A BiocParallelParam object specifying the parallelization to use for the neighbor searches.
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 additional statistics should be returned.

Details

DBSCAN operates by identifying core points, i.e., observations with at least min.pts neighbors within a distance of eps. It then identifies which core points are neighbors of each other, forming components of connected core points. All non-core points are then connected to the closest core point within eps. All groups of points that are connected in this manner are considered to be part of the same cluster. Any unconnected non-core points are treated as noise and reported as NA.

As a suitable value of eps may not be known beforehand, we can automatically determine it from the data. For all observations, we compute the distance to the kth neighbor where k is defined as round(min.pts * core.prop). We then define eps as the core.prop quantile of the distances across all observations. The default of core.prop=0.5 means that around half of the observations will be treated as core points.

Larger values of eps will generally result in fewer observations classified as noise, as they are more likely to connect to a core point. It may also promote agglomeration of existing clusters into larger entities if they are connected by regions of (relatively) low density. Conversely, larger values of min.pts will generally increase the number of noise points and may fragment larger clusters into subclusters.

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

Value

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

The clusterRows method will return a factor of length equal to nrow(x) containing the cluster assignments. Note that this may contain NA values corresponding to noise points. If full=TRUE, a list is returned with clusters (the factor, as above) and objects (a list containing the eps and min.pts used in the analysis).

Author(s)

Aaron Lun

References

Ester M et al. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, 226-231.

Examples

```
clusterRows(iris[,1:4], DbscanParam())
clusterRows(iris[,1:4], DbscanParam(core.prop=0.8))
```

DianaParam-class *Divisive analysis clustering*

Description

Use the diana function to perform divisive analysis clustering.

Usage

```
DianaParam(
  metric = NULL,
  stand = NULL,
  cut.fun = NULL,
  cut.dynamic = FALSE,
  cut.params = list()
)

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

Arguments

metric	String specifying the distance metric to use in dist.
stand	Further arguments to pass to diana.
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.params	Further arguments to pass to cut.fun.
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 DianaParam object `x`, users can simply call `x[[i]]` or `x[[i]] <- value` where `i` is any argument used in the constructor.

If `cut.fun=NULL`, `cut.dynamic=FALSE` and `cut.params` does not have `h` or `k`, `clusterRows` will automatically set `h` to half the tree height when calling `cutree`.

Value

The `DianaParam` constructor will return a DianaParam 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` (a list containing `diana`, the function output; `dist`, the dissimilarity matrix; and `hclust`, a hclust object created from `diana`).

Author(s)

Aaron Lun

See Also

`diana`, which actually does all the heavy lifting.

HclustParam, for the more commonly used implementation of hierarchical clustering.

Examples

```
clusterRows(iris[,1:4], DianaParam())
clusterRows(iris[,1:4], DianaParam(metric="manhattan"))
```

FixedNumberParam-class

The FixedNumberParam class

Description

The FixedNumberParam is a virtual subclass of the BlusterParam class. It causes `clusterRows` to dispatch to clustering algorithms that rely on a pre-specified number of clusters, e.g., KmeansParam.

Available methods

`centers(x, n=NULL)` will return the specified number of centers in a FixedNumberParam `x`. This can be an positive integer, or a function that accepts the number of observations and returns a positive number. If a function and `n` is supplied, the function is called on `n` and the result is rounded to obtain an integer.

`centers(x) <- value` will replace the specified number of centers in `x` with an integer scalar or function `value`. The function should accept a single argument and return a positive integer.

Author(s)

Aaron Lun

See Also

KmeansParam, for the archetypal example of a concrete subclass.

`HclustParam-class` *Hierarchical clustering*

Description

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

Usage

```
HclustParam(
  metric = NULL,
  method = NULL,
  cut.fun = NULL,
  cut.dynamic = FALSE,
  cut.height = NULL,
  cut.number = NULL,
  cut.params = list(),
  ...
)

## 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, cut.number`	Deprecated, use `h` and `k` in `cut.params` instead.
`cut.params`	Further arguments to pass to `cut.fun`.
`...`	Deprecated, more arguments to add to `cut.params`.

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.

If cut.fun=NULL, cut.dynamic=FALSE and cut.params does not have h or k, clusterRows will automatically set h to half the tree height when calling cutree.

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 (a list containing dist, the distance matrix; and hclust, the output of hclust).

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"))
```

HierarchicalParam-class

The HierarchicalParam class

Description

The HierarchicalParam is a virtual subclass of the BlusterParam class. It causes clusterRows to dispatch to clustering algorithms that produce a dissimilarity matrix and a dendrogram.

Available slots

The virtual class provides `metric`, the choice of distance metric. This is conventionally passed to `dist` and defaults to a Euclidean distance in most subclasses.

It also provides a number of slots to manage the final tree cut:

- `cut.fun`, a function that takes a hclust object as its first argument and returns a vector of cluster assignments. If `NULL`, the choice of function is determined from `cut.dynamic`.
- `cut.dynamic`, a logical scalar indicating whether a dynamic tree cut should be performed by `cutreeDynamic`. Otherwise `cutree` is used. Ignored if `cut.fun` is not `NULL`.
- `cut.params`, further arguments to pass to the tree cut function specified by the previous arguments.

Return value

The contract is that, when `full=TRUE`, the `objects` field of the `clusterRows` return value will always contain at least the following elements:

- `dist`, a dist object containing a dissimilarity matrix, usually a distance matrix.
- `hclust`, a hclust object containing a dendrogram.

See Also

HclustParam, for the archetypal example of a concrete subclass.

KmeansParam-class *K-means clustering*

Description

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

Usage

```
KmeansParam(centers, iter.max = NULL, nstart = NULL, algorithm = NULL)

## 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.
`iter.max, nstart, algorithm`	Further arguments to pass to `kmeans`. Set to the `kmeans` defaults if not supplied.
`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` (a list containing `kmeans`, the direct output of `kmeans`).

Author(s)

Aaron Lun

See Also

`kmeans`, which actually does all the heavy lifting.

MbkmeansParam, for a faster but more approximate version of the k-means algorithm.

Examples

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

`linkClusters` *Create a graph between different clusterings*

Description

Create a graph that links together clusters from different clusterings, e.g., generated using different parameter settings or algorithms. This is useful for identifying corresponding clusters between clusterings and to create meta-clusters from multiple clusterings.

Usage

```
linkClusters(clusters, prefix = TRUE, denominator = c("union", "min", "max"))

linkClustersMatrix(x, y, denominator = c("union", "min", "max"))
```

Arguments

clusters	A list of factors or vectors where each entry corresponds to a clustering. All vectors should be of the same length. The list itself should usually be named with a suitable label for each clustering.
prefix	Logical scalar indicating whether the cluster levels should be prefixed with its clustering. If clusters is not named, numeric prefixes are used instead.
denominator	String specifying how the strength of the correspondence between clusters should be computed.
x, y	Factor or vector specifying a clustering of the same cells.

Details

Links are only formed between clusters from different clusterings, e.g., between clusters X in clustering 1 and Y in clustering 2. The edge weight of each link is set to the strength of the correspondence between the two clusters; this is defined from the number of cells with those two labels in their respective clusterings. A larger number of cells indicates that X and Y are corresponding clusters.

Of course, the number of cells also depends on the total number of cells in each cluster. To account for this, we normalize the strength by a function of the total number of cells in the two clusters. The choice of function is determined by denominator and determines how the strength is adjusted for dissimilar cluster sizes.

- For "min", the number of shared cells is divided by the smaller of the totals between the two clusters.
- For "max", the number of shared cells is divided by the larger of the totals.
- For "union", the number of shared cells is divided by the size of the union of cells in the two clusters. The result is equivalent to the Jaccard index.

In situations where X splits into multiple smaller clusters $Y1$, $Y2$, etc. in another clustering, denominator="min" will report strong links between X and its constituent subclusters while "max" and "union" will report weak links. Conversely, denominator="max" and "union" can only form strong links when there is a 1:1 mapping between clusters in different clusterings. This usually yields simpler correspondences between clusterings at the cost of orphaning some of the smaller subclusters. denominator="union" is most stringent as it will penalize the presence of non-shared cells in both clusters, whereas "max" only does so for the larger cluster.

The general idea is to use the graph returned by this function in visualization routines or for community-based clustering, to identify "clusters of clusters" that can inform about the relationships between clusterings.

Value

For linkClusters, a graph object where each node is a cluster level in one of the clusterings in clusters. Edges are weighted by the strength of the correspondence between two clusters in different clusterings.

For linkClustersMatrix, a matrix is returned where each row and column corresponds to a cluster in x and y, respectively. Entries represent the strength of the correspondence between the associated clusters; this is equivalent to a submatrix of the adjacency matrix from the graph returned by linkClusters.

Author(s)

Aaron Lun

See Also

The **clustree** package, which provides another method for visualizing relationships between clusterings.

compareClusterings, which computes similarities between the clusterings themselves.

Examples

```
clusters <- list(
    nngraph = clusterRows(iris[,1:4], NNGraphParam()),
    hclust = clusterRows(iris[,1:4], HclustParam(cut.dynamic=TRUE)),
    kmeans = clusterRows(iris[,1:4], KmeansParam(5))
)

g <- linkClusters(clusters)
plot(g)

igraph::cluster_walktrap(g)

# Results as a matrix, for two clusterings:
linkClustersMatrix(clusters[[1]], clusters[[2]], denominator="union")
```

---

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 function 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 - see "Weighting Schemes" below.

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.

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.

Shared neighbor weighting schemes

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**.

Tehcnically, edges with zero weights are assigned a nominal small positive weight of the order of 1e-6. This is done only to satisfy the requirements for positive weights in many **igraph** clustering algorithms. We do not just remove these edges as that might lead to the situation where some observations have no edges at all and thus form single-observation clusters.

Note that the behavior of k for type="rank" is slightly different from that used in the original SNNCliq implementation by Xu and Su. The original implementation considers each observation to be its first nearest neighbor that contributes to k. Here, the k nearest neighbours refers to the number of *other* observations.

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)
```

MbkmeansParam-class *Mini-batch k-means clustering*

Description

Run the mini-batch k-means mbkmeans function with the specified number of centers within clusterRows. This sacrifices some accuracy for speed compared to the standard k-means algorithm. Note that this requires installation of the **mbkmeans** package.

Usage

```
MbkmeansParam(
  centers,
  batch_size = NULL,
  max_iters = 100,
  num_init = 1,
  init_fraction = NULL,
  initializer = "kmeans++",
  calc_wcss = FALSE,
  early_stop_iter = 10,
  tol = 1e-04,
  BPPARAM = SerialParam()
)
```

```
## S4 method for signature 'ANY,MbkmeansParam'
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.
batch_size, max_iters, num_init, init_fraction, initializer, calc_wcss, early_stop_iter, tol, BPPARAM	Further arguments to pass to mbkmeans.
x	A numeric matrix-like object where rows represent observations and columns represent variables.
BLUSPARAM	A MbkmeansParam object.
full	Logical scalar indicating whether the full mini-batch 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 MbkmeansParam object x, users can simply call x[[i]] or x[[i]] <- value where i is any argument used in the constructor.

For batch_size and init_fraction, a value of NULL means that the default arguments in the mbkmeans function signature are used. These defaults are data-dependent and so cannot be specified during construction of the MbkmeansParam object, but instead are defined within the clusterRows method.

Value

The MbkmeansParam constructor will return a MbkmeansParam 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 (a list containing mbkmeans, the direct output of mbkmeans).

Author(s)

Stephanie Hicks

See Also

mbkmeans from the **mbkmeans** package, which actually does all the heavy lifting.

KmeansParam, for dispatch to the standard k-means algorithm.

Examples

```
clusterRows(iris[,1:4], MbkmeansParam(centers=3))
clusterRows(iris[,1:4], MbkmeansParam(centers=3, batch_size=10))
clusterRows(iris[,1:4], MbkmeansParam(centers=3, init_fraction=0.5))
```

---

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))
```

---

nestedClusters *Map nested clusterings*

Description

Map an alternative clustering to a reference clustering, where the latter is expected to be nested within the former.

Usage

```
nestedClusters(ref, alt)
```

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.

Details

This function identifies mappings between two clusterings on the same set of cells where alt is potentially nested within ref (e.g., as it is computed at higher resolution). To do so, we take each alt cluster and compute the the proportion of its cells that are derived from each ref cluster. The corresponding ref cluster is identified as that with the highest proportion, as reported by the which field in the mapping DataFrame.

The quality of the mapping is determined by max in the output mapping DataFrame. A low value indicates that alt does not have a clear counterpart in ref, representing loss of heterogeneity. Note that this is not a symmetrical inference; multiple alt clusters can map to the same ref cluster without manifesting as a low max. This implicitly assumes that an increase in resolution in alt is not problematic.

The ref.score value for each cluster ref is formally defined as the probability of randomly picking a cell that belongs to ref, conditional on the event that the chosen cell belongs to the same alt cluster as a randomly chosen cell from ref. This probability is equal to unity when ref is an exact superset of all alt clusters that contain its cells, corresponding to perfect 1:many nesting. In contrast, if the alt clusters contain a mix of cells from different ref, this probability will be low and can be used as a diagnostic for imperfect nesting.

Value

A list containing:

- proportions, a matrix where each row corresponds to one of the alt clusters and each column corresponds to one of the ref clusters. Each matrix entry represents the proportion of cells in alt that are assigned to each cluster in ref. (That is, the proportions across all ref clusters should sum to unity for each alt cluster.)
- alt.mapping, a DataFrame with one row per cluster in alt. This contains the columns max, a numeric vector specifying the maximum value of statistic for that alt cluster; and which, a character vector specifying the ref cluster in which the maximum value occurs.
- ref.score, a numeric vector of length equal to the number of ref clusters. This represents the degree of nesting of alt clusters within each ref cluster, see Details.

See Also

linkClusters, to do this in a symmetric manner (i.e., without nesting).

pairwiseRand, for another way of comparing two sets of clusterings.

Examples

```
m <- matrix(runif(10000), ncol=10)
clust1 <- kmeans(m,10)$cluster
clust2 <- kmeans(m,20)$cluster
nestedClusters(clust1, clust2)

# The ref.score is 1 in cases of perfect nesting.
nestedClusters(clust1, clust1)$ref.score

nest.clust <- paste0(clust1, sample(letters, length(clust1), replace=TRUE))
nestedClusters(clust1, nest.clust)$ref.score

# In contrast, it is much lower when nesting is bad.
nestedClusters(clust1, sample(clust1))$ref.score
```

NNGraphParam-class *Graph-based clustering*

Description

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

Usage

```
NNGraphParam(
  shared = TRUE,
  k = 10,
  ...,
  BNPARAM = KmknnParam(),
  BPPARAM = SerialParam(),
  cluster.fun = "walktrap",
  cluster.args = list()
)

SNNGraphParam(
  k = 10,
  type = "rank",
  BNPARAM = KmknnParam(),
  BPPARAM = SerialParam(),
  cluster.fun = "walktrap",
  cluster.args = list()
)

KNNGraphParam(
  k = 10,
  directed = FALSE,
  BNPARAM = KmknnParam(),
  BPPARAM = SerialParam(),
  cluster.fun = "walktrap",
  cluster.args = list()
)

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

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

Arguments

shared	Logical scalar indicating whether a shared NN graph should be constructed.

k	An integer scalar specifying the number of nearest neighbors to consider during graph construction.
...	Further arguments to pass to SNNGraphParam (if shared=TRUE) or KNNGraphParam.
BNPARAM	A BiocNeighborParam object specifying the nearest neighbor algorithm.
BPPARAM	A BiocParallelParam object to use for parallel processing.
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.
type	A string specifying the type of weighting scheme to use for shared neighbors.
directed	A logical scalar indicating whether the output of buildKNNGraph should be a directed graph.
x	A matrix-like object containing expression values for each observation (row) and dimension (column).
BLUSPARAM	A NNGraphParam object.
full	Logical scalar indicating whether the graph-based clustering objects should be returned.

Details

The SNNGraphParam and KNNGraphParam classes are both derived from the NNGraphParam virtual class. This former will perform clustering with a shared nearest-neighbor (SNN) graph while the latter will use a simpler k-nearest neighbor (KNN) graph - see ?makeSNNGraph for 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. The exception is that of shared, which is not a valid i as it is implicit in the identity of the class.

Value

The constructors will return a NNGraphParam object with the specified parameters. If shared=TRUE, this is a SNNGraphParam object; otherwise it is a KNNGraphParam object.

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))

# Note: cluster_louvain is randomized as of igraph 1.3.0.
set.seed(100)
clusterRows(iris[,1:4], NNGraphParam(cluster.fun="louvain"))

# On the plus side, we can finally pass a resolution parameter.
set.seed(100)
clusterRows(iris[,1:4], NNGraphParam(cluster.fun="louvain",
    cluster.args=list(resolution=0.5)))
```

---

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. Such events can be diagnosed by looking for small (i.e., near-zero or negative) entries in the ratio matrix; on the other hand, large values (i.e., 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.

`compareClusterings`, which does this for multiple clusterings.

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")
```

PamParam-class *Partitioning around medoids*

Description

Partition observations into k-medoids as a more robust version of k-means.

Usage

```
PamParam(
  centers,
  metric = NULL,
  medoids = NULL,
  nstart = NULL,
  stand = NULL,
  do.swap = NULL,
  variant = NULL
)

## S4 method for signature 'ANY,PamParam'
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.
metric, medoids, nstart, stand, do.swap, variant	Further arguments to pass to pam. Set to the function defaults if not supplied.
x	A numeric matrix-like object where rows represent observations and columns represent variables.
BLUSPARAM	A PamParam object.
full	Logical scalar indicating whether the full PAM 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 PamParam object `x`, users can simply call `x[[i]]` or `x[[i]] <- value` where `i` is any argument used in the constructor.

Value

The PamParam constructor will return a PamParam 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 (a list containing pam, the direct output of pam).

Author(s)

Aaron Lun

See Also

pam, which actually does all the heavy lifting.

KmeansParam, for the more commonly used k-means algorithm.

ClaraParam, for a scalable extension to the PAM approach.

Examples

```
clusterRows(iris[,1:4], PamParam(centers=4))
clusterRows(iris[,1:4], PamParam(centers=4, variant="faster", do.swap=FALSE))
clusterRows(iris[,1:4], PamParam(centers=sqrt))
```

SomParam-class *Clustering with self-organizing maps*

Description

Use the self-organizing map implementation in the **kohonen** package to cluster observations into the specified number of nodes. Note that this requires the installation of the **kohonen** package.

Usage

```
SomParam(
  centers,
  dim.ratio = 1,
  topo = "rectangular",
  neighbourhood.fct = "bubble",
  toroidal = FALSE,
  rlen = 100,
  alpha = c(0.05, 0.01),
  radius = NULL,
  dist.fct = "sumofsquares"
)

## S4 method for signature 'ANY,SomParam'
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.
dim.ratio	A positive numeric scalar in specifying how centers should be distributed between the x and y dimensions. Defaults to equal distribution, i.e., both dimensions will be of length equal to the square root of centers. Values above 1 will distribute more nodes to x while values below 1 will distribute mode nodes to y.
topo, neighbourhood.fct, toroidal	Further arguments to pass to the somgrid function in the **kohonen** package.
rlen, alpha, radius, dist.fct	Further arguments to pass to the som function in the **kohonen** package.
x	A numeric matrix-like object where rows represent observations and columns represent variables.
BLUSPARAM	A SomParam object.
full	Logical scalar indicating whether the full SOM 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.

Note that the final number of clusters may not be exactly equal to centers, depending on how dim.ratio is specified. For example, if centers is a perfect square and dim.ratio=1, we will get exactly the requested number of points.

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

For radius, a value of NULL means that the default argument in the som function signature is used. This is are data-dependent and so cannot be specified during construction of the SomParam object.

For dist.fct, users can specify any string that can be used in the dist.fcts arguments in som. In practice, the only real alternative is "manhattan".

Value

The SomParam constructor will return a SomParam 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 (a list containing som, the direct output of som).

Author(s)

Aaron Lun

See Also

som from the **kohonen** package, which does all of the heavy lifting.

FixedNumberParam, the parent of the SomParam class.

Examples

```
clusterRows(iris[,1:4], SomParam(centers=16))
clusterRows(iris[,1:4], SomParam(centers=12, dim.ratio=3/4))
```

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