Package ‘supraHex’
December 22, 2016

Type Package

Title supraHex: a supra-hexagonal map for analysing tabular omics data

Version 1.12.0

Date 2016-9-24

Author Hai Fang and Julian Gough

Maintainer Hai Fang <hfang@well.ox.ac.uk>

Depends R (>= 3.0.2), hexbin

Imports ape, MASS

Description A supra-hexagonal map is a giant hexagon on a 2-dimensional grid seamlessly consisting of smaller hexagons. It is supposed to train, analyse and visualise a high-dimensional omics input data. The supraHex is able to carry out gene clustering/meta-clustering and sample correlation, plus intuitive visualisations to facilitate exploratory analysis. More importantly, it allows for overlaying additional data onto the trained map to explore relations between input and additional data. So with supraHex, it is also possible to carry out multi-layer omics data comparisons. Newly added utilities are advanced heatmap visualisation and tree-based analysis of sample relationships. Uniquely to this package, users can ultra-fastly understand any tabular omics data, both scientifically and artistically, especially in a sample-specific fashion but without loss of information on large genes.

URL http://suprahex.r-forge.r-project.org

Collate sPipeline.r sHexGrid.r sTopology.r sInitial.r sTrainology.r sTrainSeq.r sTrainBatch.r sBMH.r sNeighDirect.r sNeighAny.r sHexDist.r sDistance.r sDmat.r sDmatMinima.r sDmatCluster.r sCompReorder.r sWriteData.r sMapOverlay.r visHexPattern.r visHexGrid.r visHexMapping.r visHexComp.r visColormap.r visColorbar.r visVp.r visHexMulComp.r visHexAnimate.r visCompReorder.r visDmatCluster.r visKernels.r visColoralpha.r visHeatmap.r visHeatmapAdv.r visTreeBootstrap.r visTreeBSclust.r visDmatHeatmap.r visHexBarplot.r

License GPL-2

biocViews Software, Clustering, Visualization, GeneExpression

NeedsCompilation no

R topics documented:

  Fang .................................................. 2
Human embryo gene expression dataset from Fang et al. (2010)

Description

Human embryo dataset contains gene expression levels (5441 genes and 18 embryo samples) from Fang et al. (2010).
Usage
data(Fang)

Value

• Fang: a gene expression matrix of 5441 genes x 18 samples, involving six successive stages, each with three replicates.

• Fang.sampleinfo: a matrix containing the information of the 18 samples for the expression matrix Fang. The three columns correspond to the sample information: "Name", "Stage" and "Replicate".

• Fang.geneinfo: a matrix containing the information of the 5441 genes for the expression matrix Fang. The three columns correspond to the gene information: "AffyID", "EntrezGene" and "Symbol".

References


Golub

Leukemia gene expression dataset from Golub et al. (1999)

Description

Leukemia dataset (learning set) contains gene expression levels (3051 genes and 38 patient samples) from Golub et al. (1999). This dataset has been pre-processed: capping into floor of 100 and ceiling of 16000; filtering by exclusion of genes with max/min <= 5 or max − min <= 500, where max and min refer respectively to the maximum and minimum intensities for a particular gene across mRNA samples; 2-base logarithmic transformation.

Usage
data(Golub)

Value

• Golub: a gene expression matrix of 3051 genes x 38 samples. These samples include 11 acute myeloid leukemia (AML) and 27 acute lymphoblastic leukemia (ALL) which can be further subtyped into 19 B-cell ALL and 8 T-cell ALL.

References

sBMH

Function to identify the best-matching hexagons/rectangles for the input data

Description

sBMH is supposed to identify the best-matching hexagons/rectangles (BMH) for the input data.

Usage

sBMH(sMap, data, which_bmh = c("best", "worst", "all"))

Arguments

sMap an object of class "sMap" or a codebook matrix
data a data frame or matrix of input data
which_bmh which BMH is requested. It can be a vector consisting of any integer values from [1, nHex]. Alternatively, it can also be one of "best", "worst" and "all" choices. Here, "best" is equivalent to 1, "worst" for nHex, and "all" for seq(1, nHex)

Value

a list with following components:

• bmh: the requested BMH matrix of dlen x length(which_bmh), where dlen is the total number of rows of the input data
• qerr: the corresponding matrix of quantization errors (i.e., the distance between the input data and their BMH), with the same dimensions as "bmh" above
• mqe: the mean quantization error for the "best" BMH
• call: the call that produced this result

Note

"which_bmh" upon request can be a vector consisting of any integer values from [1, nHex]

See Also

sPipeline

Examples

# 1) generate an iid normal random matrix of 100x10
data <- matrix(rnorm(100*10,mean=0, sd=1), nrow=100, ncol=10)

# 2) from this input matrix, determine nHex=5*sqrt(nrow(data))=50,
# but it returns nHex=61, via "SHexGrid(nHex=50)", to make sure a supra-hexagonal grid
sTopol <- sTopology(data=data, lattice="hexa", shape="supahex")

# 3) initialise the codebook matrix using "uniform" method
sI <- sInitial(data=data, sTopol=sTopol, init="uniform")
# 4) define trainology at "rough" stage
sT_rough <- sTrainology(sMap=sI, data=data, stage="rough")

# 5) training at "rough" stage
sM_rough <- sTrainBatch(sMap=sI, data=data, sTrain=sT_rough)

# 6) define trainology at "finetune" stage
sT_finetune <- sTrainology(sMap=sI, data=data, stage="finetune")

# 7) training at "finetune" stage
sM_finetune <- sTrainBatch(sMap=sM_rough, data=data, sTrain=sT_rough)

# 8) find the best-matching hexagons/rectangles for the input data
response <- sBMH(sMap=sM_finetune, data=data, which_bmh="best")

### Description

`sCompReorder` is supposed to reorder component planes for the input map/data. It returns an object of class "sReorder". It is realized by using a new map grid (with sheet shape consisting of a rectangular lattice) to train component plane vectors (either column-wise vectors of codebook/data matrix or the covariance matrix thereof). As a result, similar component planes are placed closer to each other. It is highly recommend to use trained map (i.e. codebook matrix) as input if data matrix is hugely big to save computational costs.

### Usage

`sCompReorder(sMap, xdim = NULL, ydim = NULL, amplifier = NULL, metric = c("none", "pearson", "spearman", "kendall", "euclidean", "manhattan", "cos", "mi"), init = c("linear", "uniform", "sample"), algorithm = c("sequential", "batch"), alphaType = c("invert", "linear", "power"), neighKernel = c("gaussian", "bubble", "cutgaussian", "ep", "gamma"))`

### Arguments

- **sMap**: an object of class "sMap" or input data frame/matrix
- **xdim**: an integer specifying x-dimension of the grid
- **ydim**: an integer specifying y-dimension of the grid
- **amplifier**: an integer specifying the amplifier (3 by default) of the number of component planes. The product of the component number and the amplifier constitutes the number of rectangles in the sheet grid
- **metric**: distance metric used to define the similarity between component planes. It can be "none", which means directly using column-wise vectors of codebook/data matrix. Otherwise, first calculate the covariance matrix from the codebook/data matrix. The distance metric used for calculating the covariance matrix between component planes can be: "pearson" for pearson correlation, "spearman" for spearman rank correlation, "kendall" for kendall tau rank correlation, "euclidean" for euclidean distance, "manhattan" for cityblock distance, "cos" for cosine similarity, "mi" for mutual information. See `sDistance` for details
**init**  
an initialisation method. It can be one of "uniform", "sample" and "linear" initialisation methods

**algorithm**  
the training algorithm. It can be one of "sequential" and "batch" algorithm. By default, it uses 'sequential' algorithm. If the input data contains a large number of samples but not a great amount of zero entries, then it is reasonable to use 'batch' algorithm for its fast computations (probably also without the compromise of accuracy)

**alphaType**  
the alpha type. It can be one of "invert", "linear" and "power" alpha types

**neighKernel**  
the training neighbor kernel. It can be one of "gaussian", "bubble", "cutgaussian", "ep" and "gamma" kernels

**Value**  
an object of class "sReorder", a list with following components:

- **nHex**: the total number of rectangles in the grid
- **xdim**: x-dimension of the grid
- **ydim**: y-dimension of the grid
- **uOrder**: the unique order/placement for each component plane that is reordered to the "sheet"-shape grid with rectangular lattice
- **coord**: a matrix of nHex x 2, with each row corresponding to the coordinates of each "uOrder" rectangle in the 2D map grid
- **call**: the call that produced this result

**Note**  
All component planes are uniquely placed within a "sheet"-shape rectangle grid:

- Each component plane mapped to the "sheet"-shape grid with rectangular lattice is determined iteratively in an order from the best matched to the next compromised one.
- If multiple components are hit in the same rectangular lattice, the worse one is always sacrificed by moving to the next best one till all components are placed somewhere exclusively on their own.

The size of "sheet"-shape rectangle grid depends on the input arguments:

- How the input parameters are used to determine nHex is taken priority in the following order: "xdim & ydim" > "nHex" > "data".
- If both of xdim and ydim are given, $nHex = xdim \times ydim$.
- If only data is input, $nHex = 5 \times \sqrt{dlen}$, where dlen is the number of rows of the input data.
- After nHex is determined, xy-dimensions of rectangle grid are then determined according to the square root of the two biggest eigenvalues of the input data.

**See Also**

`sTopology, sPipeline, sBMH, sDistance, visCompReorder`
**Examples**

# 1) generate an iid normal random matrix of 100x10
```r
data <- matrix(rnorm(100*10, mean=0, sd=1), nrow=100, ncol=10)
colnames(data) <- paste(rep("Var",10), seq(1:10), sep="")
```

# 2) get trained using by default setup
```r
sMap <- sPipeline(data=data)
```

# 3) reorder component planes in different ways
# 3a) directly using column-wise vectors of codebook matrix
```r
sReorder <- sCompReorder(sMap=sMap, amplifier=2, metric="none")
```

# 3b) according to covariance matrix of pearson correlation of codebook matrix
```r
sReorder <- sCompReorder(sMap=sMap, amplifier=2, metric="pearson")
```

# 3c) according to covariance matrix of pearson correlation of input matrix
```r
sReorder <- sCompReorder(sMap=data, amplifier=2, metric="pearson")
```

---

**sDistance**

*Function to compute the pairwise distance for a given data matrix*

**Description**

`sDistance` is supposed to compute and return the distance matrix between the rows of a data matrix using a specified distance metric.

**Usage**

```r
sDistance(data, metric = c("pearson", "spearman", "kendall", "euclidean", "manhattan", "cos", "mi"))
```

**Arguments**

- `data`: a data frame or matrix of input data
- `metric`: distance metric used to calculate a symmetric distance matrix. See 'Note' below for options available

**Value**

- `dist`: a symmetric distance matrix of nRow x nRow, where nRow is the number of rows of input data matrix

**Note**

The distance metrics are supported:

- "pearson": Pearson correlation. Note that two curves that have identical shape, but different magnitude will still have a correlation of 1
- "spearman": Spearman rank correlation. As a nonparametric version of the pearson correlation, it calculates the correlation between the ranks of the data values in the two vectors (more robust against outliers)
"kendall": Kendall tau rank correlation. Compared to spearman rank correlation, it goes a step further by using only the relative ordering to calculate the correlation. For all pairs of data points \((x_i, y_i)\) and \((x_j, y_j)\), it calls a pair of points either as concordant \((N_c\) in total) if \((x_i - x_j) \times (y_i - y_j) > 0\), or as discordant \((N_d\) in total) if \((x_i - x_j) \times (y_i - y_j) < 0\). Finally, it calculates gamma coefficient \((N_c - N_d) / (N_c + N_d)\) as a measure of association which is highly resistant to tied data.

"euclidean": Euclidean distance. Unlike the correlation-based distance measures, it takes the magnitude into account (input data should be suitably normalized).

"manhattan": Cityblock distance. The distance between two vectors is the sum of absolute value of their differences along any coordinate dimension.

"cos": Cosine similarity. As an uncentered version of pearson correlation, it is a measure of similarity between two vectors of an inner product space, i.e., measuring the cosine of the angle between them (using a dot product and magnitude).

"mi": Mutual information (MI). MI provides a general measure of dependencies between variables, in particular, positive, negative and nonlinear correlations. The calculation of MI is implemented via applying adaptive partitioning method for deriving equal-probability bins (i.e., each bin contains approximately the same number of data points). The number of bins is heuristically determined (the lower bound): \(1 + \log_2(n)\), where \(n\) is the length of the vector. Because MI increases with entropy, we normalize it to allow comparison of different pairwise clone similarities: \(2 \times MI / [H(x) + H(y)]\), where \(H(x)\) and \(H(y)\) stand for the entropy for the vector \(x\) and \(y\), respectively.

**See Also**

`sDmatCluster`

**Examples**

```r
# 1) generate an iid normal random matrix of 100x10
data <- matrix(rnorm(100*10, mean=0, sd=1), nrow=100, ncol=10)

# 2) calculate distance matrix using different metric
sMap <- sPipeline(data=data)
# 2a) using "pearson" metric
dist <- sDistance(data=data, metric="pearson")
# 2b) using "cos" metric
# dist <- sDistance(data=data, metric="cos")
# 2c) using "spearman" metric
# dist <- sDistance(data=data, metric="spearman")
# 2d) using "kendall" metric
# dist <- sDistance(data=data, metric="kendall")
# 2e) using "euclidean" metric
# dist <- sDistance(data=data, metric="euclidean")
# 2f) using "manhattan" metric
# dist <- sDistance(data=data, metric="manhattan")
# 2g) using "mi" metric
# dist <- sDistance(data=data, metric="mi")
```
**sDmat**

*Function to calculate distance matrix in high-dimensional input space but according to neighborhood relationships in 2D output space*

---

### Description

`sDmat` is supposed to calculate distance (measured in high-dimensional input space) to neighbors (defined by based on 2D output space) for each of hexagons/rectangles.

### Usage

```r
sDmat(sMap, which_neigh = 1, distMeasure = c("median", "mean", "min", "max"))
```

### Arguments

- **sMap**: an object of class "sMap"
- **which_neigh**: which neighbors in 2D output space are used for the calculation. By default, it sets to "1" for direct neighbors, and "2" for neighbors within neighbors no more than 2, and so on.
- **distMeasure**: distance measure used to calculate distances in high-dimensional input space.

### Value

- **dMat**: a vector with the length of nHex. It stores the distance a hexagon/rectangle is away from its output-space-defined neighbors in high-dimensional input space.

### Note

- "which_neigh" is defined in output 2D space, but "distMeasure" is defined in high-dimensional input space.

### See Also

- `sNeighAny`

### Examples

```r
# 1) generate an iid normal random matrix of 100x10
data <- matrix( rnorm(100*10, mean=0, sd=1), nrow=100, ncol=10)
# 2) get trained using by default setup
sMap <- sPipeline(data=data)
# 3) calculate "median" distances in INPUT space to different neighbors in 2D OUTPUT space
# 3a) using direct neighbors in 2D OUTPUT space
dMat <- sDmat(sMap=sMap, which_neigh=1, distMeasure="median")
# 3b) using no more than 2-topological neighbors in 2D OUTPUT space
# dMat <- sDmat(sMap=sMap, which_neigh=2, distMeasure="median")
```
sDmatCluster

Function to partition a grid map into clusters

Description

sDmatCluster is supposed to obtain clusters from a grid map. It returns an object of class "sBase".

Usage

sDmatCluster(sMap, which_neigh = 1, distMeasure = c("median", "mean", "min", "max"), clusterLinkage = c("average", "complete", "single", "bmh"), reindexSeed = c("hclust", "svd", "none"))

Arguments

sMap
an object of class "sMap"

which_neigh
which neighbors in 2D output space are used for the calculation. By default, it sets to "1" for direct neighbors, and "2" for neighbors within neighbors no more than 2, and so on

distMeasure
distance measure used to calculate distances in high-dimensional input space. It can be one of "median", "mean", "min" and "max" measures

clusterLinkage
cluster linkage used to derive clusters. It can be "bmh", which accumulates a cluster just based on best-matching hexagons/rectangles but can not ensure each cluster is continuous. Instead, each cluster is continuous when using region-growing algorithm with one of "average", "complete" and "single" linkages

reindexSeed
the way to index seed. It can be "hclust" for reindexing seeds according to hierarchical clustering of patterns seen in seeds, "svd" for reindexing seeds according to svd of patterns seen in seeds, or "none" for seeds being simply increased by the hexagon indexes (i.e. always in an increasing order as hexagons radiate outwards)

Value

an object of class "sBase", a list with following components:

- seeds: the vector to store cluster seeds, i.e., a list of local minima (in 2D output space) of distance matrix (in input space). They are represented by the indexes of hexagons/rectangles
- bases: the vector with the length of nHex to store the cluster memberships/bases, where nHex is the total number of hexagons/rectangles in the grid
- call: the call that produced this result

Note

The first item in the return "seeds" is the first cluster, whose memberships are those in the return "bases" that equals 1. The same relationship is held for the second item, and so on

See Also

sPipeline, sDmatMinima, sBMH, sNeighDirect, sDistance, visDmatCluster
Examples

# 1) generate an iid normal random matrix of 100x10
data <- matrix( rnorm(100*10,mean=0,sd=1), nrow=100, ncol=10)

# 2) get trained using by default setup
sMap <- sPipeline(data=data)

# 3) partition the grid map into clusters based on different criteria
# 3a) based on "bmh" criterion
sBase <- sDmatCluster(sMap=sMap, which_neigh=1, distMeasure="median", clusterLinkage="bmh")
# 3b) using region-growing algorithm with linkage "average"
sBase <- sDmatCluster(sMap=sMap, which_neigh=1, distMeasure="median", clusterLinkage="average")

# 4) visualise clusters/bases partitioned from the sMap
visDmatCluster(sMap,sBase)

sDmatMinima

Function to identify local minima (in 2D output space) of distance matrix (in high-dimensional input space)

Description

sDmatMinima is supposed to identify local minima of distance matrix (resulting from sDmat). The criterion of being local minima is that the distance associated with a hexagon/rectangle is always smaller than its direct neighbors (i.e., 1-neighborhood)

Usage

sDmatMinima(sMap, which_neigh = 1, distMeasure = c("median", "mean", "min", "max"))

Arguments

sMap an object of class "sMap"
which_neigh which neighbors in 2D output space are used for the calculation. By default, it sets to "1" for direct neighbors, and "2" for neighbors within neighbors no more than 2, and so on
distMeasure distance measure used to calculate distances in high-dimensional input space. It can be one of "median", "mean", "min" and "max" measures

Value

- minima: a vector to store a list of local minima (represented by the indexes of hexagons/rectangles

Note

Do not get confused by "which_neigh" and the criteria of being local minima. Both of them deal with 2D output space. However, "which_neigh" is used to assist in the calculation of distance matrix (so can be 1-neighborhood or more); instead, the criterion of being local minima is only 1-neighborhood in the strictest sense
**sHexDist**

Function to calculate distances between hexagons/rectangles in a 2D grid

### Description

`sHexDist` is supposed to calculate euclidian distances between each pair of hexagons/rectangles in a 2D grid of input "sTopol" or "sMap" object. It returns a symmetric matrix containing pairwise distances.

### Usage

`sHexDist(sObj)`

### Arguments

- **sObj**: an object of class "sTopol" or "sInit" or "sMap"

### Value

- **dist**: a symmetric matrix of nHex x nHex, containing pairwise distances, where nHex is the total number of hexagons/rectangles in the grid

### Note

The return matrix has rows/columns ordered in the same order as the "coord" matrix of the input object does.

### See Also

`sTopology`, `sInitial`
Examples

# 1) generate an iid normal random matrix of 100x10
data <- matrix( rnorm(100*10,mean=0,sd=1), nrow=100, ncol=10)

# 2) from this input matrix, determine nHex=5*sqrt(nrow(data))=50,
# but it returns nHex=61, via “sHexGrid(nHex=50)”, to make sure a supra-hexagonal grid
sTopol <- sTopology(data=data, lattice="hexa", shape="suprahex")

# 3) initialise the codebook matrix using "uniform" method
sI <- sInitial(data=data, sTopol=sTopol, init="uniform")

# 4) calculate distances between hexagons/rectangles in a 2D grid based on different objects
# 4a) based on an object of class "sTopol"
dist <- sHexDist(sObj=sTopol)
# 4b) based on an object of class "sMap"
dist <- sHexDist(sObj=sI)

sHexGrid

Function to define a supra-hexagonal grid

Description

sHexGrid is supposed to define a supra-hexagonal map grid. A supra-hexagon is a giant hexagon, which seamlessly consists of smaller hexagons. Due to the symmetric nature, it can be uniquely determined by specifying the radius away from the grid centroid. This function takes input the grid radius (or the number of hexagons in the grid, but will be adjusted to meet the definition of supra-hexagon), and returns a list (see ‘Value’ below) containing: the grid radius, the total number of hexagons in the grid, the 2D coordinates of the grid centroid, the step for each hexagon away from the grid centroid, and the 2D coordinates of all hexagons in the grid.

Usage

sHexGrid(r = NULL, nHex = NULL)

Arguments

r

an integer specifying the radius in a supra-hexagonal grid

nHex

the number of input hexagons in the grid

Value

a list with following components:

• r: the grid radius
• nHex: the total number of hexagons in the grid. It may differ from the input value; actually it is always no less than the input one to ensure a supra-hexagonal grid exactly formed
• centroid: the 2D coordinates of the grid centroid
• stepCentroid: a vector with the length of nHex. It stores how many steps a hexagon is away from the grid centroid (‘1’ for the centroid itself). Starting with the centroid, it orders outward. Also, for those hexagons of the same step, it orders from the rightmost in an anti-clock wise
• coord: a matrix of nHex x 2 with each row specifying the 2D coordinates of a hexagon in the grid. The order of rows is the same as ‘centroid’ above
• call: the call that produced this result
Note

The relationships among return values:

- \( n_{Hex} = 1 + 6 \times r \times (r - 1)/2 \)
- \( centroid = coord[1,] \)
- \( stepCentroid[1] = 1 \)
- \( stepCentroid[2:n_{Hex}] = unlist(sapply(2:r, function(x)c((1 + 6 \times x \times (x - 1)/2 - 6 \times (x - 1) + 1): (1 + 6 \times x \times (x - 1)/2)) >= 1 \times x)) \)

See Also

sTopology

Examples

# The supra-hexagonal grid is exactly determined by specifying the radius.
res <- sHexGrid(r=2)

# The grid is determined according to the number of input hexagons (after being adjusted).
# The return res$nHex is always no less than the input one.
# It ensures a supra-hexagonal grid is exactly formed.
res <- sHexGrid(nHex=12)

# Ignore input nHex if r is also given
res <- sHexGrid(r=3, nHex=100)

# By default, r=3 if no parameters are specified
res <- sHexGrid()
sInitial

Value

an object of class "sInit", a list with following components:

- `nHex`: the total number of hexagons/rectangles in the grid
- `xdim`: x-dimension of the grid
- `ydim`: y-dimension of the grid
- `lattice`: the grid lattice
- `shape`: the grid shape
- `coord`: a matrix of nHex x 2, with each row corresponding to the coordinates of a hexagon/rectangle in the 2D map grid
- `init`: an initialisation method
- `codebook`: a codebook matrix of nHex x ncol(data), with each row corresponding to a prototype vector in input high-dimensional space
- `call`: the call that produced this result

Note

The initialisation methods include:

- "uniform": the codebook matrix is uniformly initialised via randomly taking any values within the interval [min, max] of each column of input data
- "sample": the codebook matrix is initialised via randomly sampling/selecting input data
- "linear": the codebook matrix is linearly initialised along the first two greatest eigenvectors of input data

See Also

sTopology

Examples

# 1) generate an iid normal random matrix of 100x10
data <- matrix( rnorm(100*10,mean=0,sd=1), nrow=100, ncol=10)

# 2) from this input matrix, determine nHex=5*sqrt(nrow(data))=50, # but it returns nHex=61, via "sHexGrid(nHex=50)", to make sure a supra-hexagonal gridsTopol <- sTopology(data=data, lattice="hexa", shape="suprahex")

# 3) initialise the codebook matrix using different methods
# 3a) using "uniform" method
sI_uniform <- sInitial(data=data, sTopol=sTopol, init="uniform")
# 3b) using "sample" method
# sI_sample <- sInitial(data=data, sTopol=sTopol, init="sample")
# 3c) using "linear" method
# sI_linear <- sInitial(data=data, sTopol=sTopol, init="linear")
sMapOverlay

*Function to overlay additional data onto the trained map for viewing the distribution of that additional data*

**Description**

sMapOverlay is supposed to overlay additional data onto the trained map for viewing the distribution of that additional data. It returns an object of class "sMap". It is realised by first estimating the hit histogram weighted by the neighborhood kernel, and then calculating the distribution of the additional data over the map (similarly weighted by the neighborhood kernel). The final overlaid distribution of additional data is normalised by the hit histogram.

**Usage**

`sMapOverlay(sMap, data, additional)`

**Arguments**

- **sMap**: an object of class "sMap"
- **data**: a data frame or matrix of input data
- **additional**: a numeric vector or numeric matrix used to overlay onto the trained map. It must have the length (if being vector) or row number (if matrix) being equal to the number of rows in input data

**Value**

an object of class "sMap", a list with following components:

- `nHex`: the total number of hexagons/rectangles in the grid
- `xdim`: x-dimension of the grid
- `ydim`: y-dimension of the grid
- `lattice`: the grid lattice
- `shape`: the grid shape
- `coord`: a matrix of nHex x 2, with rows corresponding to the coordinates of all hexagons/rectangles in the 2D map grid
- `init`: an initialisation method
- `neighKernel`: the training neighborhood kernel
- `codebook`: a codebook matrix of nHex x ncol(additional), with rows corresponding to overlaid vectors
- `hits`: a vector of nHex, each element meaning that a hexagon/rectangle contains the number of input data vectors being hit wherein
- `mqe`: the mean quantization error for the "best" BMH
- `call`: the call that produced this result

**Note**

Weighting by neighbor kernel is to avoid rigid overlaying by only focusing on the best-matching map nodes as there may exist several closest best-matching nodes for an input data vector.
sNeighAny

Function to calculate any neighbors for each hexagon/rectangle in a grid

Description

sNeighAny is supposed to calculate any neighbors for each hexagon/rectangle in a regular 2D grid. It returns a matrix with rows for the self, and columns for its any neighbors.

Usage

sNeighAny(sObj)

Arguments

sObj an object of class "sTopol" or "sInit" or "sMap"

Value

- aNeigh: a matrix of nHex x nHex, containing distance info in terms of any neighbors, where nHex is the total number of hexagons/rectangles in the grid

Note

The return matrix has rows for the self, and columns for its neighbors. The non-zeros mean the distance away from its neighbors, and the zeros for the self-self. It has rows/columns ordered in the same order as the "coord" matrix of the input object does.

See Also

sNeighDirect
Examples

# 1) generate an iid normal random matrix of 100x10
data <- matrix( rnorm(100*10, mean=0, sd=1), nrow=100, ncol=10)

# 2) from this input matrix, determine nHex=5*sqrt(nrow(data))=50,
# but it returns nHex=61, via "sHexGrid(nHex=50)"), to make sure a supra-hexagonal grid
sTopol <- sTopology(data=data, lattice="hexa", shape="suprahex")

# 3) initialise the codebook matrix using "uniform" method
sI <- sInitial(data=data, sTopol=sTopol, init="uniform")

# 4) calculate any neighbors based on different objects
# 4a) based on an object of class "sTopol"
aNeigh <- sNeighAny(sObj=sTopol)
# 4b) based on an object of class "sMap"
aNeigh <- sNeighAny(sObj=sI)

sNeighDirect

Function to calculate direct neighbors for each hexagon/rectangle in a grid

Description

sNeighDirect is supposed to calculate direct neighbors for each hexagon/rectangle in a regular 2D grid. It returns a matrix with rows for the self, and columns for its direct neighbors.

Usage

sNeighDirect(sObj)

Arguments

sObj: an object of class "sTopol" or "sInit" or "sMap"

Value

- dNeigh: a matrix of nHex x nHex, containing presence/absence info in terms of direct neighbors, where nHex is the total number of hexagons/rectangles in the grid

Note

The return matrix has rows for the self, and columns for its direct neighbors. The "1" means the presence of direct neighbors, "0" for the absence. It has rows/columns ordered in the same order as the "coord" matrix of the input object does.

See Also

sHexDist
Examples

# 1) generate an iid normal random matrix of 100x10
data <- matrix( rnorm(100*10,mean=0,sd=1), nrow=100, ncol=10)

# 2) from this input matrix, determine nHex=5*sqrt(nrow(data))=50,
# but it returns nHex=61, via "sHexGrid(nHex=50)", to make sure a supra-hexagonal grid
sTopol <- sTopology(data=data, lattice="hexa", shape="suprahex")

# 3) initialise the codebook matrix using "uniform" method
sI <- sInitial(data=data, sTopol=sTopol, init="uniform")

# 4) calculate direct neighbors based on different objects
# 4a) based on an object of class "sTopol"
dNeigh <- sNeighDirect(sObj=sTopol)
# 4b) based on an object of class "sMap"
# dNeigh <- sNeighDirect(sObj=sI)

sPipeline

Function to setup the pipeline for completing ab initio training given
the input data

Description

sPipeline is supposed to finish ab inito training for the input data. It returns an object of class
"sMap".

Usage

sPipeline(data = NULL, xdim = NULL, ydim = NULL, nHex = NULL, lattice = c("hexa", "rect"), shape = c("suprahex", "sheet"), init = c("linear", "uniform", "sample"), algorithm = c("batch", "sequential"), alphaType = c("invert", "linear", "power"), neighKernel = c("gaussian", "bubble", "cutgaussian", "ep", "gamma"), finetuneSustain = F, verbose = T)

Arguments

data a data frame or matrix of input data
xdim an integer specifying x-dimension of the grid
ydim an integer specifying y-dimension of the grid
nHex the number of hexagons/rectangles in the grid
lattice the grid lattice, either "hexa" for a hexagon or "rect" for a rectangle
shape the grid shape, either "suprahex" for a supra-hexagonal grid or "sheet" for a hexagonal/rectangle sheet
init an initialisation method. It can be one of "uniform", "sample" and "linear" initialisation methods
algorithm the training algorithm. It can be one of "sequential" and "batch" algorithm. By default, it uses 'batch' algorithm purely because of its fast computations (probably also without the compromise of accuracy). However, it is highly recommended not to use 'batch' algorithm if the input data contain lots of zeros; it is
because matrix multiplication used in the 'batch' algorithm can be problematic in this context. If much computation resource is at hand, it is always safe to use the 'sequential' algorithm

alphaType the alpha type. It can be one of "invert", "linear" and "power" alpha types

neighKernel the training neighborhood kernel. It can be one of "gaussian", "bubble", "cut-gaussian", "ep" and "gamma" kernels

finetuneSustain logical to indicate whether sustain the "finetune" training. If true, it will repeat the "finetune" stage until the mean quantization error does get worse. By default, it sets to true

verbose logical to indicate whether the messages will be displayed in the screen. By default, it sets to false for no display

Value an object of class "sMap", a list with following components:

• nHex: the total number of hexagons/rectangles in the grid
• xdim: x-dimension of the grid
• ydim: y-dimension of the grid
• lattice: the grid lattice
• shape: the grid shape
• coord: a matrix of nHex x 2, with rows corresponding to the coordinates of all hexagons/rectangles in the 2D map grid
• init: an initialisation method
• neighKernel: the training neighborhood kernel
• codebook: a codebook matrix of nHex x ncol(data), with rows corresponding to prototype vectors in input high-dimensional space
• hits: a vector of nHex, each element meaning that a hexagon/rectangle contains the number of input data vectors being hit wherein
• mqe: the mean quantization error for the "best" BMH
• call: the call that produced this result

Note The pipeline sequentially consists of:

• i) sTopology used to define the topology of a grid (with "suprahex" shape by default ) according to the input data;
• ii) sInitial used to initialise the codebook matrix given the pre-defined topology and the input data (by default using "uniform" initialisation method);
• iii) sTrainology and sTrainSeq used to get the grid map trained at both "rough" and "fine-tune" stages. If instructed, sustain the "finetune" training until the mean quantization error does get worse;
• iv) sBMH used to identify the best-matching hexagons/rectangles (BMH) for the input data, and these response data are appended to the resulting object of "sMap" class.
sTopology

References


See Also

sTopology, sInitial, sTrainology, sTrainSeq, sTrainBatch, sBMH, visHexMulComp

Examples

# 1) generate an iid normal random matrix of 100x10
data <- matrix(rnorm(100*10, mean=0, sd=1), nrow=100, ncol=10)
colnames(data) <- paste(rep(’Var’, 10), seq(1:10), sep=’’)

# 2) get trained using by default setup but with different neighborhood kernels
# 2a) with ”gaussian” kernel
sMap <- sPipeline(data=data, neighKernel=”gaussian”)
# 2b) with ”bubble” kernel
sMap <- sPipeline(data=data, neighKernel=”bubble”)
# 2c) with ”cutgaussian” kernel
sMap <- sPipeline(data=data, neighKernel=”cutgaussian”)
# 2d) with ”ep” kernel
sMap <- sPipeline(data=data, neighKernel=”ep”)
# 2e) with ”gamma” kernel
sMap <- sPipeline(data=data, neighKernel=”gamma”)

# 3) visualise multiple component planes of a supra-hexagonal grid
visHexMulComp(sMap, colormap=”jet”, ncolors=20, zlim=c(-1,1),
gp=grid::gpar(cex=0.8))

sTopology

Function to define the topology of a map grid

Description

sTopology is supposed to define the topology of a 2D map grid. The topological shape can be either a supra-hexagonal grid or a hexagonal/rectangle sheet. It returns an object of ”sTopol” class, containing: the total number of hexagons/rectangles in the grid, the grid xy-dimensions, the grid lattice, the grid shape, and the 2D coordinates of all hexagons/rectangles in the grid. The 2D coordinates can be directly used to measure distances between any pair of lattice hexagons/rectangles.

Usage

sTopology(data = NULL, xdim = NULL, ydim = NULL, nHex = NULL, lattice = c(”hexa”, ”rect”), shape = c(”suprahex”, ”sheet”))
Arguments

- **data**: a data frame or matrix of input data
- **xdim**: an integer specifying x-dimension of the grid
- **ydim**: an integer specifying y-dimension of the grid
- **nHex**: the number of hexagons/rectangles in the grid
- **lattice**: the grid lattice, either "hexa" for a hexagon or "rect" for a rectangle
- **shape**: the grid shape, either "suprahex" for a supra-hexagonal grid or "sheet" for a hexagonal/rectangle sheet

Value

An object of class "sTopol", a list with following components:

- **nHex**: the total number of hexagons/rectangles in the grid. It is not always the same as the input nHex (if any); see "Note" below for the explanation
- **xdim**: x-dimension of the grid
- **ydim**: y-dimension of the grid
- **lattice**: the grid lattice
- **shape**: the grid shape
- **coord**: a matrix of nHex x 2, with each row corresponding to the coordinates of a hexagon/rectangle in the 2D map grid
- **call**: the call that produced this result

Note

The output of nHex depends on the input arguments and grid shape:

- How the input parameters are used to determine nHex is taken priority in the following order: "xdim & ydim" > "nHex" > "data"
- If both of xdim and ydim are given, \( n_{Hex} = xdim \times ydim \) for the "sheet" shape, \( r = (\text{min}(xdim, ydim) + 1)/2 \) for the "suprahex" shape
- If only data is input, \( n_{Hex} = 5 \times \sqrt{\text{dlen}} \), where dlen is the number of rows of the input data
- With nHex in hand, it depends on the grid shape:
  - For "sheet" shape, xy-dimensions of sheet grid is determined according to the square root of the two biggest eigenvalues of the input data
  - For "suprahex" shape, see \texttt{sHexGrid} for calculating the grid radius \( r \). The xdim (and ydim) is related to \( r \) via \( xdim = 2 \times r - 1 \)

See Also

\texttt{sHexGrid, visHexMapping}
Examples

# For "suprahex" shape
sTopol <- sTopology(xdim=3, ydim=3, lattice="hexa", shape="suprahex")

# Error: "The suprahex shape grid only allows for hexagonal lattice"
sTopol <- sTopology(xdim=3, ydim=3, lattice="rect", shape="suprahex")

# For "sheet" shape with hexagonal lattice
sTopol <- sTopology(xdim=3, ydim=3, lattice="hexa", shape="sheet")

# For "sheet" shape with rectangle lattice
sTopol <- sTopology(xdim=3, ydim=3, lattice="rect", shape="sheet")

# By default, nHex=19 (i.e., r=3; xdim=ydim=5) for "suprahex" shape
sTopol <- sTopology(shape="suprahex")

# By default, xdim=ydim=5 (i.e., nHex=25) for "sheet" shape
sTopol <- sTopology(shape="sheet")

# Determine the topology of a supra-hexagonal grid based on input data
# 1) generate an iid normal random matrix of 100x10
data <- matrix(rnorm(100*10, mean=0, sd=1), nrow=100, ncol=10)
# 2) from this input matrix, determine nHex=5*sqrt(nrow(data))=50,
# but it returns nHex=61, via "sHexGrid(nHex=50)", to make sure a supra-hexagonal grid
sTopol <- sTopology(data=data, lattice="hexa", shape="suprahex")

# visualise a supre-hexagonal grid
visHexMapping(sTopol, mappingType="indexes")

---

**sTrainBatch**  
*Function to implement training via batch algorithm*

**Description**

*sTrainBatch* is supposed to perform batch training algorithm. It requires three inputs: a "sMap" or "sInit" object, input data, and a "sTrain" object specifying training environment. The training is implemented iteratively, but instead of choosing a single input vector, the whole input matrix is used. In each training cycle, the whole input matrix first land in the map through identifying the corresponding winner hexagon/rectangle (BMH), and then the codebook matrix is updated via updating formula (see "Note" below for details). It returns an object of class "sMap".

**Usage**

sTrainBatch(sMap, data, sTrain, verbose = T)

**Arguments**

- **sMap**: an object of class "sMap" or "sInit"
- **data**: a data frame or matrix of input data
- **sTrain**: an object of class "sTrain"
- **verbose**: logical to indicate whether the messages will be displayed in the screen. By default, it sets to TRUE for display
Value

an object of class "sMap", a list with following components:

• nHex: the total number of hexagons/rectangles in the grid
• xdim: x-dimension of the grid
• ydim: y-dimension of the grid
• lattice: the grid lattice
• shape: the grid shape
• coord: a matrix of nHex x 2, with each row corresponding to the coordinates of a hexagon/rectangle in the 2D map grid
• init: an initialisation method
• neighKernel: the training neighborhood kernel
• codebook: a codebook matrix of nHex x ncol(data), with each row corresponding to a prototype vector in input high-dimensional space
• call: the call that produced this result

Note

Updating formula is: \( m_i(t + 1) = \frac{\sum_{dlen}^{dlen} h_{wi}(t)x_j}{\sum_{dlen}^{dlen} h_{wi}(t)} \), where

• \( t \) denotes the training time/step
• \( x_j \) is an input vector \( j \) from the input data matrix (with \( dlen \) rows in total)
• \( i \) and \( w \) stand for the hexagon/rectangle \( i \) and the winner BMH \( w \), respectively
• \( m_i(t + 1) \) is the prototype vector of the hexagon \( i \) at time \( t + 1 \)
• \( h_{wi}(t) \) is the neighborhood kernel, a non-increasing function of i) the distance \( d_{wi} \) between the hexagon/rectangle \( i \) and the winner BMH \( w \), and ii) the radius \( \delta_t \) at time \( t \). There are five kernels available:
  – For "gaussian" kernel, \( h_{wi}(t) = e^{-d_{wi}^2/(2*\delta_t^2)} \)
  – For "cutgaussian" kernel, \( h_{wi}(t) = e^{-d_{wi}^2/(2*\delta_t^2)} * (d_{wi} \leq \delta_t) \)
  – For "bubble" kernel, \( h_{wi}(t) = (d_{wi} \leq \delta_t) \)
  – For "ep" kernel, \( h_{wi}(t) = (1 - d_{wi}^2/\delta_t^2) * (d_{wi} \leq \delta_t) \)
  – For "gamma" kernel, \( h_{wi}(t) = 1/\Gamma(d_{wi}^2/(4*\delta_t^2) + 2) \)

See Also

sTrainology, visKernels

Examples

# 1) generate an iid normal random matrix of 100x10
data <- matrix( rnorm(100*10,mean=0,sd=1), nrow=100, ncol=10)

# 2) from this input matrix, determine nHex=5*sqrt(nrow(data))=50,
# but it returns nHex=61, via "sHexGrid(nHex=50)", to make sure a supra-hexagonal grid
sTopol <- sTopology(data=data, lattice="hexa", shape="suprahex")

# 3) initialise the codebook matrix using "uniform" method
sI <- sInitial(data=data, sTopol=sTopol, init="uniform")
# 4) define trainology at "rough" stage
sT_rough <- sTrainology(sMap=sI, data=data, stage="rough")

# 5) training at "rough" stage
sM_rough <- sTrainBatch(sMap=sI, data=data, sTrain=sT_rough)

# 6) define trainology at "finetune" stage
sT_finetune <- sTrainology(sMap=sI, data=data, stage="finetune")

# 7) training at "finetune" stage
sM_finetune <- sTrainBatch(sMap=sM_rough, data=data, sTrain=sT_rough)

### sTrainology

**Function to define trainology (training environment)**

**Description**

sTrainology is supposed to define the train-ology (i.e., the training environment/parameters). The trainology here refers to the training algorithm, the training stage, the stage-specific parameters (alpha type, initial alpha, initial radius, final radius and train length), and the training neighbor kernel used. It returns an object of class "sTrain".

**Usage**

sTrainology(sMap, data, algorithm = c("batch", "sequential"),
stage = c("rough", "finetune", "complete"), alphaType = c("invert",
"linear", "power"), neighKernel = c("gaussian", "bubble",
"cutgaussian",
"ep", "gamma"))

**Arguments**

- **sMap**: an object of class "sMap" or "sInit"
- **data**: a data frame or matrix of input data
- **algorithm**: the training algorithm. It can be one of "sequential" and "batch" algorithm
- **stage**: the training stage. The training can be achieved using two stages (i.e., "rough" and "finetune") or one stage only (i.e., "complete")
- **alphaType**: the alpha type. It can be one of "invert", "linear" and "power" alpha types
- **neighKernel**: the training neighbor kernel. It can be one of "gaussian", "bubble", "cutgaussian", "ep" and "gamma" kernels

**Value**

an object of class "sTrain", a list with following components:

- **algorithm**: the training algorithm
- **stage**: the training stage
- **alphaType**: the alpha type
- **alphaInitial**: the initial alpha
- radiusInitial: the initial radius
- radiusFinal: the final radius
- neighKernel: the neighbor kernel
- call: the call that produced this result

Note

Training stage-specific parameters:

- "radiusInitial": it depends on the grid shape and training stage
  - For "sheet" shape: it equals \( \max(1, \text{ceiling}(\max(xdim, ydim)/8)) \) at "rough" or "complete" stage, and \( \max(1, \text{ceiling}(\max(xdim, ydim)/32)) \) at "finetune" stage
  - For "suprahex" shape: it equals \( \max(1, \text{ceiling}(r/2)) \) at "rough" or "complete" stage, and \( \max(1, \text{ceiling}(r/8)) \) at "finetune" stage

- "radiusFinal": it depends on the training stage
  - At "rough" stage, it equals \( \frac{\text{radiusInitial}}{4} \)
  - At "finetune" or "complete" stage, it equals 1

- "trainLength": how many times the whole input data are set for training. It depends on the training stage and training algorithm
  - At "rough" stage, it equals \( \max(1, 10 \times \text{trainDepth}) \)
  - At "finetune" stage, it equals \( \max(1, 40 \times \text{trainDepth}) \)
  - At "complete" stage, it equals \( \max(1, 50 \times \text{trainDepth}) \)
  - When using "batch" algorithm and the trainLength equals 1 according to the above equation, the trainLength is forced to be 2 unless radiusInitial equals radiusFinal
  - Where \( \text{trainDepth} \) is the training depth, defined as \( nHex/dlen \), i.e., how many hexagons/rectangles are used per the input data length (here \( dlen \) refers to the number of rows)

See Also

sInitial

Examples

# 1) generate an iid normal random matrix of 100x10
data <- matrix( rnorm(100*10,mean=0,sd=1), nrow=100, ncol=10)

# 2) from this input matrix, determine \( nHex=5\times\sqrt{\text{nrow}(\text{data})}=50 \)
# but it returns \( nHex=61 \), via \( \text{sHexGrid}(nHex=50) \), to make sure a supra-hexagonal grid
sTopol <- sTopology(data=data, lattice="hexa", shape="suprahex")

# 3) initialise the codebook matrix using "uniform" method
sI <- sInitial(data=data, sTopol=sTopol, init="uniform")

# 4) define trainology at different stages
# 4a) define trainology at "rough" stage
sT_rough <- sTrainology(sMap=sI, data=data, stage="rough")

# 4b) define trainology at "finetune" stage
sT_finetune <- sTrainology(sMap=sI, data=data, stage="finetune")

# 4c) define trainology using "complete" stage
sT_complete <- sTrainology(sMap=sI, data=data, stage="complete")
### Description

`sTrainSeq` is supposed to perform sequential training algorithm. It requires three inputs: a "sMap" or "sInit" object, input data, and a "sTrain" object specifying training environment. The training is implemented iteratively, each training cycle consisting of: i) randomly choose one input vector; ii) determine the winner hexagon/rectangle (BMH) according to minimum distance of codebook matrix to the input vector; ii) update the codebook matrix of the BMH and its neighbors via updating formula (see "Note" below for details). It also returns an object of class "sMap".

### Usage

```r
sTrainSeq(sMap, data, sTrain, verbose = T)
```

### Arguments

- **sMap**: an object of class "sMap" or "sInit"
- **data**: a data frame or matrix of input data
- **sTrain**: an object of class "sTrain"
- **verbose**: logical to indicate whether the messages will be displayed in the screen. By default, it sets to TRUE for display

### Value

An object of class "sMap", a list with following components:

- `nHex`: the total number of hexagons/rectangles in the grid
- `xdim`: x-dimension of the grid
- `ydim`: y-dimension of the grid
- `lattice`: the grid lattice
- `shape`: the grid shape
- `coord`: a matrix of nHex x 2, with each row corresponding to the coordinates of a hexagon/rectangle in the 2D map grid
- `init`: an initialisation method
- `neighKernel`: the training neighborhood kernel
- `codebook`: a codebook matrix of nHex x ncol(data), with each row corresponding to a prototype vector in input high-dimensional space
- `call`: the call that produced this result

### Note

Updating formula is: $m_i(t + 1) = m_i(t) + \alpha(t) \ast h_{w_i}(t) \ast [x(t) - m_i(t)]$, where

- $t$ denotes the training time/step
- $i$ and $w$ stand for the hexagon/rectangle $i$ and the winner BMH $w$, respectively
- $x(t)$ is an input vector randomly choosen (from the input data) at time $t$
• \( m_i(t) \) and \( m_i(t+1) \) are respectively the prototype vectors of the hexagon \( i \) at time \( t \) and \( t+1 \)

• \( \alpha(t) \) is the learning rate at time \( t \). There are three types of learning rate functions:
  
  - For "linear" function, \( \alpha(t) = \alpha_0 * (1 - t/T) \)
  - For "power" function, \( \alpha(t) = \alpha_0 * (0.005/\alpha_0)^{t/T} \)
  - For "invert" function, \( \alpha(t) = \alpha_0/(1 + 100 * t/T) \)
  
  Where \( \alpha_0 \) is the initial learning rate (typically, \( \alpha_0 = 0.5 \) at "rough" stage, \( \alpha_0 = 0.05 \) at "finetune" stage), \( T \) is the length of training time/step (often being set to input data length, i.e., the total number of rows)

• \( h_{wi}(t) \) is the neighborhood kernel, a non-increasing function of i) the distance \( d_{wi} \) between the hexagon/rectangle \( i \) and the winner BMH \( w \), and ii) the radius \( \delta_t \) at time \( t \). There are five kernels available:
  
  - For "gaussian" kernel, \( h_{wi}(t) = e^{-d_{wi}^2/(2*\delta_t^2)} \)
  - For "cutguassian" kernel, \( h_{wi}(t) = e^{-d_{wi}^2/(2*\delta_t^2)} * (d_{wi} \leq \delta_t) \)
  - For "bubble" kernel, \( h_{wi}(t) = (d_{wi} \leq \delta_t) \)
  - For "ep" kernel, \( h_{wi}(t) = (1 - d_{wi}^2/\delta_t^2) * (d_{wi} \leq \delta_t) \)
  - For "gamma" kernel, \( h_{wi}(t) = 1/\Gamma(d_{wi}^2/(4*\delta_t^2) + 2) \)

See Also

sTrainology, visKernels

Examples

# 1) generate an iid normal random matrix of 100x10
data <- matrix( rnorm(100*10,mean=0,sd=1), nrow=100, ncol=10)

# 2) from this input matrix, determine nHex=5*sqrt(nrow(data))=50,
# but it returns nHex=61, via "sHexGrid(nHex=50)", to make sure a supra-hexagonal grid
sTopol <- sTopology(data=data, lattice="hexa", shape="suprahex")

# 3) initialise the codebook matrix using "uniform" method
sI <- sInitial(data=data, sTopol=sTopol, init="uniform")

# 4) define trainology at "rough" stage
sT_rough <- sTrainology(sMap=sI, data=data, algorithm="sequential", stage="rough")

# 5) training at "rough" stage
sM_rough <- sTrainSeq(sMap=sI, data=data, sTrain=sT_rough)

# 6) define trainology at "finetune" stage
sT_finetune <- sTrainology(sMap=sI, data=data, algorithm="sequential", stage="finetune")

# 7) training at "finetune" stage
sM_finetune <- sTrainSeq(sMap=sM_rough, data=data, sTrain=sT_rough)
sWriteData **Function to write out the best-matching hexagons and/or cluster bases in terms of data**

**Description**

sWriteData is supposed to write out the best-matching hexagons and/or cluster bases in terms of data.

**Usage**

sWriteData(sMap, data, sBase = NULL, filename = NULL, keep.data = F)

**Arguments**

- **sMap**: an object of class "sMap" or a codebook matrix
- **data**: a data frame or matrix of input data
- **sBase**: an object of class "sBase"
- **filename**: a character string naming a filename
- **keep.data**: logical to indicate whether or not to also write out the input data. By default, it sets to false for not keeping it. It is highly expensive to keep the large data sets

**Value**

A data frame with following components:

- **ID**: ID for data. It inherits the rownames of data (if exists). Otherwise, it is sequential integer values starting with 1 and ending with dlen, the total number of rows of the input data
- **Hexagon_index**: the index for best-matching hexagons
- **Cluster_base**: optional, it is only appended when sBase is given. It stores the cluster memberships/bases
- **data**: optional, it is only appended when keep.data is true

**Note**

If "filename" is not NULL, a tab-delimited text file will be also written out. If "sBase" is not NULL and comes from the "sMap" partition, then cluster bases are also appended. If "keep.data" is true, the data will be part of output.

**See Also**

sBMH
Examples

# 1) generate an iid normal random matrix of 100x10
data <- matrix(rnorm(100*10,mean=0,sd=1), nrow=100, ncol=10)

# 2) get trained using by default setup
sMap <- sPipeline(data=data)

# 3) write data's BMH hitting the trained map
output <- sWriteData(sMap=sMap, data=data, filename="sData_output.txt")

# 4) partition the grid map into cluster bases
sBase <- sDmatCluster(sMap=sMap, which_neigh=1, distMeasure="median", clusterLinkage="average")

# 5) write data's BMH and cluster bases
output <- sWriteData(sMap=sMap, data=data, sBase=sBase, filename="sData_base_output.txt")

visColoralpha

Function to add transparent (alpha) into colors

Description

visColoralpha is supposed to add transparent (alpha) into colors.

Usage

visColoralpha(col, alpha)

Arguments

col input colors. It can be vector of R color specifications, such as a color name (as
listed by `colors()), a hexadecimal string of the form "#rrggbb" or "#rrggbbaa"

alpha numeric vector of values in the range [0, 1] for alpha transparency channel (0
means transparent and 1 means opaque)

Value

a vector of colors (after transparent being added)

Note

none

See Also

visColormap
Examples

# 1) define "blue-white-red" colormap
palette.name <- visColormap(colormap="bwr")

# 2) use the return function "palette.name" to generate 10 colors spanning "bwr"
col <- palette.name(10)

# 3) add transparent (alpha=0.5)
cols <- visColoralpha(col, alpha=0.5)

visColorbar is supposed to define a colorbar

Usage

visColorbar(colormap = c("bwr", "jet", "gbr", "wyr", "br", "yr", "rainbow", "wb"), ncolors = 40, zlim = c(0, 1), gp = grid::gpar())

Arguments

colormap short name for the colormap. It can be one of "jet" (jet colormap), "bwr" (blue-white-red colormap), "gbr" (green-black-red colormap), "wyr" (white-yellow-red colormap), "br" (black-red colormap), "yr" (yellow-red colormap), "wb" (white-black colormap), and "rainbow" (rainbow colormap, that is, red-yellow-green-cyan-blue-magenta). Alternatively, any hyphen-separated HTML color names, e.g. "blue-black-yellow", "royalblue-white-sandybrown", "darkgreen-white-darkviolet". A list of standard color names can be found in http://html-color-codes.info/color-names

ncolors the number of colors specified

zlim the minimum and maximum z values for which colors should be plotted, defaulting to the range of the finite values of z. Each of the given colors will be used to color an equispaced interval of this range. The midpoints of the intervals cover the range, so that values just outside the range will be plotted

gp an object of class gpar, typically the output from a call to the function gpar (i.e., a list of graphical parameter settings)

Value

invisibly

Note

none

See Also

visColormap, visHexMulComp, visCompReorder
Examples

```r
# draw "blue-white-red" colorbar
visColorbar(colormap="bwr")
```

**visColormap**

*Function to define a colormap*

**Description**

`visColormap` is supposed to define a colormap. It returns a function, which will take an integer argument specifying how many colors interpolate the given colormap.

**Usage**

```r
visColormap(colormap = c("bwr", "jet", "gbr", "wyr", "br", "yr",
"rainbow",
"wb"))
```

**Arguments**

- **colormap**: short name for the colormap

**Value**

- `palette.name`: a function that takes an integer argument for generating that number of colors interpolating the given sequence

**Note**

The input colormap includes:

- "jet": jet colormap
- "bwr": blue-white-red
- "gbr": green-black-red
- "wyr": white-yellow-red
- "br": black-red
- "yr": yellow-red
- "wb": white-black
- "rainbow": rainbow colormap, that is, red-yellow-green-cyan-blue-magenta
- Alternatively, any hyphen-separated HTML color names, e.g. "blue-black-yellow", "royalblue-white-sandybrown", "darkblue-lightblue-lightyellow-darkorange", "darkgreen-white-darkviolet", "darkgreen-lightgreen-lightpink-darkred". A list of standard color names can be found in [http://html-color-codes.info/color-names](http://html-color-codes.info/color-names)

**See Also**

`visColoralpha`
Examples

```r
# 1) define "blue-white-red" colormap
palette.name <- visColormap(colormap="bwr")

# 2) use the return function "palette.name" to generate 10 colors spanning "bwr"
palette.name(10)
```

**visCompReorder**

*Function to visualise multiple component planes reordered within a sheet-shape rectangle grid*

**Description**

`visCompReorder` is supposed to visualise multiple component planes reordered within a sheet-shape rectangle grid.

**Usage**

```r
visCompReorder(sMap, sReorder, margin = rep(0.1, 4), height = 7,
    title.rotate = 0, title.xy = c(0.45, 1), colormap = c("bwr", "jet",
    "gbr", "wyr", "br", "yr", "rainbow", "wb"), ncolors = 40, zlim = NULL,
    border.color = "transparent", gp = grid::gpar(), newpage = T)
```

**Arguments**

- `sMap`: an object of class "sMap"
- `sReorder`: an object of class "sReorder"
- `margin`: margins as units of length 4 or 1
- `height`: a numeric value specifying the height of device
- `title.rotate`: the rotation of the title
- `title.xy`: the coordinates of the title
- `colormap`: short name for the colormap. It can be one of "jet" (jet colormap), "bwr" (blue-white-red colormap), "gbr" (green-black-red colormap), "wyr" (white-yellow-red colormap), "br" (black-red colormap), "yr" (yellow-red colormap), "wb" (white-black colormap), and "rainbow" (rainbow colormap, that is, red-yellow-green-cyan-blue-magenta). Alternatively, any hyphen-separated HTML color names, e.g. "blue-black-yellow", "royalblue-white-sandybrown", "darkgreen-white-darkviolet". A list of standard color names can be found in [http://html-color-codes.info/color-names](http://html-color-codes.info/color-names)
- `ncolors`: the number of colors specified
- `zlim`: the minimum and maximum z values for which colors should be plotted, defaulting to the range of the finite values of z. Each of the given colors will be used to color an equispaced interval of this range. The midpoints of the intervals cover the range, so that values just outside the range will be plotted
- `border.color`: the border color for each hexagon
- `gp`: an object of class "gpar". It is the output from a call to the function "gpar" (i.e., a list of graphical parameter settings)
- `newpage`: logical to indicate whether to open a new page. By default, it sets to true for opening a new page
Value

invisible

Note

none

See Also

visVp, visHexComp, visColorbar, sCompReorder

Examples

```r
# 1) generate data with an iid matrix of 1000 x 9
data <- cbind(matrix(rnorm(1000*3,mean=0, sd=1), nrow=1000, ncol=3),
          matrix(rnorm(1000*3,mean=0.5, sd=1), nrow=1000, ncol=3),
          matrix(rnorm(1000*3,mean=-0.5, sd=1), nrow=1000, ncol=3))
colnames(data) <- c("S1", "S1", "S1", "S2", "S2", "S2", "S3", "S3", "S3")
# 2) sMap resulted from using by default setup
sMap <- sPipeline(data=data)
# 3) reorder component planes
sReorder <- sCompReorder(sMap=sMap, amplifier=2, metric="none")
# 4) visualise multiple component planes reordered within a sheet-shape rectangle grid
visCompReorder(sMap=sMap, sReorder=sReorder, margin=rep(0.1,4), height=7,
    title.rotate=0, title.xy=c(0.45, 1), colormap="gbr", ncolors=10,
    zlim=c(-1,1), border.color="transparent")
```

visDmatCluster  
Function to visualise clusters/bases partitioned from a supra-hexagonal grid

Description

visDmatCluster is supposed to visualise clusters/bases partitioned from a supra-hexagonal grid

Usage

```r
visDmatCluster(sMap, sBase, height = 7, margin = rep(0.1, 4),
area.size = 1, gp = grid::gpar(cex = 0.8, font = 2, col = "black"),
border.color = "transparent", fill.color = NULL, lty = 1, lwd = 1,
lineend = "round", linejoin = "round", colormap = c("rainbow", "jet",
"bwr", "gbr", "wyr", "br", "yr", "wb"), clip = c("on", "inherit",
"off"),
newpage = T)
```
Arguments

- **sMap**: an object of class "sMap"
- **sBase**: an object of class "sBase"
- **height**: a numeric value specifying the height of device
- **margin**: margins as units of length 4 or 1
- **area.size**: an integer or a vector specifying the area size of each hexagon
- **gp**: an object of class "gpar". It is the output from a call to the function "gpar" (i.e., a list of graphical parameter settings)
- **border.color**: the border color for each hexagon
- **fill.color**: the filled color for each hexagon
- **lty**: the line type for each hexagon. 0 for 'blank', 1 for 'solid', 2 for 'dashed', 3 for 'dotted', 4 for 'dotdash', 5 for 'longdash', 6 for 'twodash'
- **lwd**: the line width for each hexagon
- **lineend**: the line end style for each hexagon. It can be one of 'round', 'butt' and 'square'
- **linejoin**: the line join style for each hexagon. It can be one of 'round', 'mitre' and 'bevel'
- **colormap**: short name for the colormap. It can be one of "jet" (jet colormap), "bwr" (blue-white-red colormap), "gbr" (green-black-red colormap), "wyr" (white-yellow-red colormap), "br" (black-red colormap), "yr" (yellow-red colormap), "wb" (white-black colormap), and "rainbow" (rainbow colormap, that is, red-yellow-green-cyan-blue-magenta). Alternatively, any hyphen-separated HTML color names, e.g., "blue-black-yellow", "royalblue-white-sandybrown", "darkgreen-white-darkviolet". A list of standard color names can be found in [http://html-color-codes.info/color-names](http://html-color-codes.info/color-names)
- **clip**: either "on" for clipping to the extent of this viewport, "inherit" for inheriting the clipping region from the parent viewport, or "off" to turn clipping off altogether
- **newpage**: logical to indicate whether to open a new page. By default, it sets to true for opening a new page

Value

- invisible

Note

- none

See Also

- `sDmatCluster`, `sDmat`, `visColormap`, `visHexGrid`

Examples

```r
# 1) generate an iid normal random matrix of 100x10
data <- matrix( rnorm(100*10,mean=0,sd=1), nrow=100, ncol=10)

# 2) get trained using by default setup
sMap <- sPipeline(data=data)

# 3) partition the grid map into clusters using region-growing algorithm
```
visDmatHeatmap

Function to visualise gene clusters/bases partitioned from a supra-hexagonal grid using heatmap

Description

visDmatHeatmap is supposed to visualise gene clusters/bases partitioned from a supra-hexagonal grid using heatmap

Usage

visDmatHeatmap(sMap, data, sBase, base.color = "rainbow", base.separated.arg = NULL, base.legend.location = c("none", "bottomleft", "bottomright", "bottom", "left", "topleft", "top", "topright", "right", "center"), reorderRow = c("none", "hclust", "svd"), keep.data = F, ...)

Arguments

- sMap: an object of class "sMap" or a codebook matrix
- data: a data frame or matrix of input data
- sBase: an object of class "sBase"
- base.color: short name for the colormap used to encode bases (in row side bar). It can be one of "jet" (jet colormap), "bwr" (blue-white-red colormap), "gbr" (green-black-red colormap), "wyr" (white-yellow-red colormap), "br" (black-red colormap), "yr" (yellow-red colormap), "wb" (white-black colormap), and "rainbow" (rainbow colormap, that is, red-yellow-green-cyan-blue-magenta). Alternatively, any hyphen-separated HTML color names, e.g. "blue-black-yellow", "royalblue-white-sandybrown", "darkgreen-white-darkviolet". A list of standard color names can be found in http://html-color-codes.info/color-names
base.separated.arg

a list of main parameters used for styling bar separated lines. See ‘Note’ below for details on the parameters

base.legend.location

location of legend to describe bases. If "none", this legend will not be displayed

reorderRow

the way to reorder the rows within a base. It can be "none" for rows within a base being reordered by the hexagon indexes, "hclust" for rows within a base being reordered according to hierarchical clustering of patterns seen, "svd" for rows within a base being reordered according to svd of patterns seen

keep.data

logical to indicate whether or not to also write out the input data. By default, it sets to false for not keeping it. It is highly expensive to keep the large data sets

... additional graphic parameters used in "visHeatmapAdv". For most parameters, please refer to [http://www.inside-r.org/packages/cran/gplots/docs/heatmap](http://www.inside-r.org/packages/cran/gplots/docs/heatmap).

Value

da data frame with following components:

- **ID**: ID for data. It inherits the rownames of data (if exists). Otherwise, it is sequential integer values starting with 1 and ending with dlen, the total number of rows of the input data
- **Hexagon_index**: the index for best-matching hexagons
- **Cluster_base**: optional, it is only appended when sBase is given. It stores the cluster memberships/bases
- **data**: optional, it is only appended when keep.data is true

Note: the returned data has rows in the same order as visualised in the heatmap

Note

A list of parameters in "base.separated.arg”:

- **"lty"**: the line type. Line types can either be specified as an integer (0=blank, 1=solid (default), 2=dashed, 3=dotted, 4=dotdash, 5=longdash, 6=twodash) or as one of the character strings "blank", "solid", "dashed", "dotted", "dotdash", "longdash", "twodash". where "blank" uses ‘invisible lines’ (i.e., does not draw them)
- **"lwd"**: the line width
- **"col"**: the line color

See Also

sDmatCluster, visHeatmapAdv

Examples

```r
# 1) generate an iid normal random matrix of 100x10
data <- matrix(rnorm(100*10, mean=0, sd=1), nrow=100, ncol=10)

# 2) get trained using by default setup
sMap <- sPipeline(data=data)

# 3) partition the grid map into clusters using region-growing algorithm
sBase <- sDmatCluster(sMap=sMap, which_neigh=1,
```
distMeasure="median", clusterLinkage="average")

# 4) heatmap visualisation
output <- visDmatHeatmap(sMap, data, sBase, base.legend.location="bottomleft", labRow=NA)

visHeatmap

Function to visualise input data matrix using heatmap

Description

visHeatmap is supposed to visualise input data matrix using heatmap. Note: this heatmap displays matrix in a bottom-to-top direction

Usage


Arguments

data an input gene-sample data matrix used for heatmap
scale a character indicating when the input matrix should be centered and scaled. It can be one of "none" (no scaling), "row" (being scaled in the row direction), "column" (being scaled in the column direction)
row.metric distance metric used to calculate the distance metric between rows. It can be one of "none" (i.e. no dendrogram between rows), "pearson", "spearman", "kendall", "euclidean", "manhattan", "cos" and "mi". See details at http://suprahex.r-forge.r-project.org/sDistance.html
row.method the agglomeration method used to cluster rows. This should be one of "ward", "single", "complete", "average", "mcquitty", "median" or "centroid". See 'Note' below for details
column.metric distance metric used to calculate the distance metric between columns. It can be one of "none" (i.e. no dendrogram between rows), "pearson", "spearman", "kendall", "euclidean", "manhattan", "cos" and "mi". See details at http://suprahex.r-forge.r-project.org/sDistance.html
column.method the agglomeration method used to cluster columns. This should be one of "ward", "single", "complete", "average", "mcquitty", "median" or "centroid". See 'Note' below for details
visHeatmap

colormap short name for the colormap. It can be one of "jet" (jet colormap), "bwr" (blue-white-red colormap), "gbr" (green-black-red colormap), "wyr" (white-yellow-red colormap), "br" (black-red colormap), "yr" (yellow-red colormap), "wb" (white-black colormap), and "rainbow" (rainbow colormap, that is, red-yellow-green-cyan-blue-magenta). Alternatively, any hyphen-separated HTML color names, e.g. "blue-black-yellow", "royalblue-white-sandybrown", "darkgreen-white-darkviolet". A list of standard color names can be found in http://html-color-codes.info/color-names

colors the number of colors specified over the colormap

zlim the minimum and maximum z/pattern values for which colors should be plotted, defaulting to the range of the finite values of z. Each of the given colors will be used to color an equispaced interval of this range. The midpoints of the intervals cover the range, so that values just outside the range will be plotted

row.cutree an integer scalar specifying the desired number of groups being cut from the row dendrogram. Note, this optional is only enabled when the row dendrogram is built

row.colormap short name for the colormap to color-code the row groups (i.e. sidebar colors used to annotate the rows)

column.cutree an integer scalar specifying the desired number of groups being cut from the column dendrogram. Note, this optional is only enabled when the column dendrogram is built

column.colormap short name for the colormap to color-code the column groups (i.e. sidebar colors used to annotate the columns)

... additional graphic parameters. Type ?heatmap for the complete list.

Value

invisible

Note

The clustering methods are provided:

- "ward": Ward’s minimum variance method aims at finding compact, spherical clusters
- "single": The single linkage method (which is closely related to the minimal spanning tree) adopts a ‘friends of friends’ clustering strategy
- "complete": The complete linkage method finds similar clusters
- "average","mcquitty","median","centroid": These methods can be regarded as aiming for clusters with characteristics somewhere between the single and complete link methods. Two methods "median" and "centroid" are not leading to a monotone distance measure, or equivalently the resulting dendrograms can have so called inversions (which are hard to interpret)

See Also

visHeatmap
Examples

# 1) generate data with an iid matrix of 100 x 9
data <- cbind(matrix(rnorm(100*3,mean=0,sd=1), nrow=100, ncol=3),
               matrix(rnorm(100*3,mean=0.5,sd=1), nrow=100, ncol=3),
               matrix(rnorm(100*3,mean=-0.5,sd=1), nrow=100, ncol=3))
colnames(data) <- c("S1","S1","S1","S2","S2","S2","S3","S3","S3")

# 2) prepare colors for the column sidebar
lvs <- unique(colnames(data))
lvs_color <- visColormap(colormap="rainbow")(length(lvs))
my_ColSideColors <- sapply(colnames(data), function(x)
  lvs_color[x==lvs])

# 3) heatmap with row dendrogram (with 10 color-coded groups)
visHeatmap(data, row.metric="euclidean", row.method="average",
           colormap="gbr", zlim=c(-2,2),
           ColSideColors=my_ColSideColors, row.cutree=10, row.colormap="jet",
           labRow=NA)

visHeatmapAdv

Function to visualise input data matrix using advanced heatmap

Description

visHeatmapAdv is supposed to visualise input data matrix using advanced heatmap. It allows for adding multiple sidecolors in both columns and rows. Besides, the sidecolor can be automatically added via cutting histogram into groups. Note: this heatmap displays matrix in a top-to-bottom direction

Usage

visHeatmapAdv(data, scale = c("none", "row", "column"), Rowv = T,
               Colv = T, dendrogram = c("both", "row", "column", "none"),
               dist.metric = c("euclidean", "pearson", "spearman", "kendall",
                              "manhattan", "cos", "mi"), linkage.method = c("complete",
                                                               "ward","single",
                                                               "average",
                                                               "mcquitty", "median", "centroid"),
               colormap = c("bwr", "jet", "gbr", "wyr", "br", "yr", "rainbow", "wb"),
               ncolors = 64, zlim = NULL, RowSideColors = NULL, row.cutree = NULL,
               row.colormap = c("jet"),
               ColSideColors = NULL, column.cutree = NULL,
               column.colormap = c("jet"), ...
)

Arguments

data an input gene-sample data matrix used for heatmap

scale a character indicating when the input matrix should be centered and scaled. It can be one of "none" (no scaling), "row" (being scaled in the row direction), "column" (being scaled in the column direction)
Rowv determines if and how the row dendrogram should be reordered. By default, it is TRUE, which implies dendrogram is computed and reordered based on row means. If NULL or FALSE, then no dendrogram is computed and no reordering is done. If a dendrogram, then it is used "as-is", i.e. without any reordering. If a vector of integers, then dendrogram is computed and reordered based on the order of the vector.

Colv determines if and how the column dendrogram should be reordered. Has the options as the Rowv argument above and additionally when x is a square matrix, Colv = "Rowv" means that columns should be treated identically to the rows.

dendrogram character string indicating whether to draw 'none', 'row', 'column' or 'both' dendrograms. Defaults to 'both'. However, if Rowv (or Colv) is FALSE or NULL and dendrogram is 'both', then a warning is issued and Rowv (or Colv) arguments are honoured.

dist.metric distance metric used to calculate the distance metric between columns (or rows). It can be one of "none" (i.e. no dendrogram between rows), "pearson", "spearman", "kendall", "euclidean", "manhattan", "cos" and "mi". See details at http://suprahex.r-forge.r-project.org/sDistance.html

linkage.method the agglomeration method used to cluster/linkages columns (or rows). This should be one of "ward", "single", "complete", "average", "mcquitty", "median" or "centroid". See 'Note' below for details

colormap short name for the colormap. It can be one of "jet" (jet colormap), "bwr" (blue-white-red colormap), "gbr" (green-black-red colormap), "wyr" (white-yellow-red colormap), "br" (black-red colormap), "yt" (yellow-red colormap), "wb" (white-black colormap), and "rainbow" (rainbow colormap, that is, red-yellow-green-cyan-blue-magenta). Alternatively, any hyphen-separated HTML color names, e.g. "blue-black-yellow", "royalblue-white-sandybrown", "darkgreen-white-darkviolet". A list of standard color names can be found in http://html-color-codes.info/color-names

ncolors the number of colors specified over the colormap

zlim the minimum and maximum z/pattern values for which colors should be plotted, defaulting to the range of the finite values of z. Each of the given colors will be used to color an equispaced interval of this range. The midpoints of the intervals cover the range, so that values just outside the range will be plotted

RowSideColors NULL or a matrix of "numRowsidebars" X nrow(x), where "numRowsidebars" stands for the number of sidebars annotating rows of x. This matrix contains the color names for vertical sidebars. By default, it sets to NULL. In this case, sidebars in rows can still be enabled by cutting the row dendrogram into several clusters (see the next two parameters)

row.cutree an integer scalar specifying the desired number of groups being cut from the row dendrogram. Note, this optional is only enabled when the ColSideColors is NULL

row.colormap short name for the colormap to color-code the row groups (i.e. sidebar colors used to annotate the rows)

ColSideColors NULL or a matrix of ncol(x) X "numColsidebars", where "numColsidebars" stands for the number of sidebars annotating the columns of x. This matrix contains the color names for horizontal sidebars. By default, it sets to NULL. In this case, sidebars in columns can still be enabled by cutting the column dendrogram into several clusters (see the next two parameters)
column.cutree  an integer scalar specifying the desired number of groups being cut from the column dendrogram. Note, this optional is only enabled when the column dendrogram is built.

column.colormap  short name for the colormap to color-code the column groups (i.e. sidebar colors used to annotate the columns)

...  additional graphic parameters. For most parameters, please refer to http://www.inside-r.org/packages/cran/gplots/docs/heatmap.2. For example, the parameters "srtRow" and "srtCol" to control the angle of row/column labels (in degrees from horizontal: 45 degrees for the column, 0 degrees for the row, by default), i.e. string rotation. The parameters "offsetRow" and "offsetCol" to indicate the number of character-width spaces to place between row/column labels and the edge of the plotting region. Unique to this function, there are two parameters "RowSideWidth" and RowSideLabelLocation, to respectively indicate the fraction of the row side width and the location (either bottom or top) of the row side labelling; the other two parameters "ColSideHeight" and "ColSideLabelLocation" for the column side height and the location (either left or right) of the column side labelling; and two parameters "RowSideBox" and "ColSideBox" to indicate whether there are boxes outside.

Value  invisible

Note  The clustering/linkage methods are provided:

- "ward": Ward's minimum variance method aims at finding compact, spherical clusters
- "single": The single linkage method (which is closely related to the minimal spanning tree) adopts a 'friends of friends' clustering strategy
- "complete": The complete linkage method finds similar clusters
- "average", "mcquitty", "median", "centroid": These methods can be regarded as aiming for clusters with characteristics somewhere between the single and complete link methods. Two methods "median" and "centroid" are not leading to a monotone distance measure, or equivalently the resulting dendrograms can have so called inversions (which are hard to interpret)

See Also  visHeatmapAdv

Examples

# 1) generate data with an iid matrix of 100 x 9
data <- cbind(matrix(rnorm(100*3,mean=0,sd=1), nrow=100, ncol=3),
   matrix(rnorm(100*3,mean=0.5,sd=1), nrow=100, ncol=3),
   matrix(rnorm(100*3,mean=-0.5,sd=1), nrow=100, ncol=3))

# 2a) shown with row and column dendrograms
visHeatmapAdv(data, dendrogram="both", colormap="gbr", zlim=c(-2,2),
   colnames(data) <- c("S1_R1","S1_R2","S1_R3","S2_R1","S2_R2","S2_R3","S3_R1","S3_R2","S3_R3")

# 2) heatmap after clustering both rows and columns
visHeatmapAdv(data, dendrogram="both", colormap="gbr", zlim=c(-2,2),
   # 2a) shown with row and column dendrograms
   visHeatmapAdv(data, dendrogram="both", colormap="gbr", zlim=c(-2,2),
   colnames(data) <-
   c("S1_R1","S1_R2","S1_R3","S2_R1","S2_R2","S2_R3","S3_R1","S3_R2","S3_R3")

   # 2) heatmap after clustering both rows and columns
   visHeatmapAdv(data, dendrogram="both", colormap="gbr", zlim=c(-2,2),
visHexAnimate function to animate multiple component planes of a supra-hexagonal grid

**Description**

visHexAnimate is supposed to animate multiple component planes of a supra-hexagonal grid. The output can be a pdf file containing a list of frames/images, a mp4 video file or a gif file. To support video output file, the software 'ffmpeg' must be first installed (also put its path into the system PATH variable; see Note). To support gif output file, the software 'ImageMagick' must be first installed (also put its path into the system PATH variable; see Note).

**Usage**

visHexAnimate(sMap, which.components = NULL, filename =
"visHexAnimate",
filetype = c("pdf", "mp4", "gif"), image.type = c("jpg", "png"),
sec_per_frame = 1, margin = rep(0.1, 4), height = 7, title.rotate = 0,
title.xy = c(0.45, 1), colormap = c("bwr", "jet", "gbr", "wyr", "br",
"yr", "rainbow", "wb"), ncolors = 40, zlim = NULL,
border.color = "transparent", gp = grid::gpar())

Arguments

sMap an object of class "sMap"
which.components
an integer vector specifying which components will be visualised. By default, it
is NULL meaning all components will be visualised
filename the without-extension part of the name of the output file. By default, it is
'visHexAnimate'
filetype the type of the output file, i.e. the extension of the output file name. It can be
one of either 'pdf' for the pdf file, 'mp4' for the mp4 video file, 'gif' for the gif
file
image.type the type of the image files temporarily generated. It can be one of either 'jpg' or
'png'. These temporary image files are used for producing mp4/gif output file.
The reason doing so is to accommodate that sometimes only one of image types
is supported so that you can choose the right one
sec_per_frame a numeric value specifying how long (seconds) it takes to stream a frame/image.
This argument only works when producing mp4 video or gif file.
margin margins as units of length 4 or 1
height a numeric value specifying the height of device
title.rotate the rotation of the title
title.xy the coordinates of the title
colormap short name for the colormap. It can be one of "jet" (jet colormap), "bwr" (blue-
white-red colormap), "gbr" (green-black-red colormap), "wyr" (white-yellow-
red colormap), "br" (black-red colormap), "yr" (yellow-red colormap), "wb" (white-black colormap), and "rainbow" (rainbow colormap, that is, red-yellow-
green-cyan-blue-magenta). Alternatively, any hyphen-separated HTML color
names, e.g. "blue-black-yellow", "royalblue-white-sandybrown", "darkgreen-
white-darkviolet". A list of standard color names can be found in http://
html-color-codes.info/color-names
ncolors the number of colors specified
zlim the minimum and maximum z values for which colors should be plotted, de-
faulting to the range of the finite values of z. Each of the given colors will be
used to color an equispaced interval of this range. The midpoints of the intervals
cover the range, so that values just outside the range will be plotted
border.color the border color for each hexagon
gp an object of class gpar, typically the output from a call to the function gpar (i.e.,
a list of graphical parameter settings)

Value

If specifying the output file name (see argument ‘filename’ above), the output file is either 'file-
name.pdf' or 'filename.mp4' or 'filename.gif' in the current working directory. If no output file
name specified, by default the output file is either 'visHexAnimate.pdf' or 'visHexAnimate.mp4' or
'visHexAnimate.gif'
Note

When producing mp4 video, this function requires the installation of the software 'ffmpeg' at
https://www.ffmpeg.org. Shell command lines for ffmpeg installation in Terminal (for both
Linux and Mac) are:

1) wget -O ffmpeg.tar.gz http://www.ffmpeg.org/releases/ffmpeg-2.7.1.tar.gz
2) mkdir ~/ffmpeg | tar xvfz ffmpeg.tar.gz -C ~/ffmpeg --strip-components=1
3) cd ffmpeg
4a) # Assuming you want installation with a ROOT (sudo) privilege:
    ./configure --disable-yasm
4b) # Assuming you want local installation without ROOT (sudo) privilege:
    ./configure --disable-yasm --prefix=$HOME/ffmpeg
5) make
6) make install
7) # add the system PATH variable to your ~/.bash_profile file if you follow 4b) route:
   export PATH=$HOME/ffmpeg:$PATH
8) # make sure ffmpeg has been installed successfully:
    ffmpeg -h

When producing gif file, this function requires the installation of the software 'ImageMagick' at
http://www.imagemagick.org. Shell command lines for ImageMagick installation in Terminal
are:

1) wget http://www.imagemagick.org/download/ImageMagick.tar.gz
2) mkdir ~/ImageMagick | tar xvzf ImageMagick.tar.gz -C ~/ImageMagick --strip-components=1
3) cd ImageMagick
4) ./configure --prefix=$HOME/ImageMagick
5) make
6) make install
7) # add the system PATH variable to your ~/.bash_profile file.
   For Linux:
   export MAGICK_HOME=$HOME/ImageMagick
   export PATH=$MAGICK_HOME/bin:$PATH
   export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$MAGICK_HOME/lib
   For Mac:
   export MAGICK_HOME=$HOME/ImageMagick
   export PATH=$MAGICK_HOME/bin:$PATH
   export DYLD_LIBRARY_PATH=$MAGICK_HOME/lib/
8a) # check configuration:
    convert -list configure
8b) # check image format supported:
    identify -list format

Tips:
Prior to 4), please make sure libjpeg and libpng are installed. If NOT, for Mac try this:
brew install libjpeg libpng
To check whether ImageMagick does work, please get additional information from:
identify -list format
convert -list configure
On details, please refer to http://www.imagemagick.org/script/advanced-unix-installation.php
See Also

visHexMulComp

Examples

## Not run:
# 1) generate data with an iid matrix of 1000 x 3
data <- cbind(matrix(rnorm(1000*3, mean=0, sd=1), nrow=1000, ncol=3),
              matrix(rnorm(1000*3, mean=0.5, sd=1), nrow=1000, ncol=3),
              matrix(rnorm(1000*3, mean=-0.5, sd=1), nrow=1000, ncol=3))
colnames(data) <- c("S1","S1","S1","S2","S2","S2","S3","S3","S3")

# 2) sMap resulted from using by default setup
sMap <- sPipeline(data=data)

# 3) animate sMap
# output as a <a href="visHexAnimate.pdf">pdf</a> file
visHexAnimate(sMap, filename="visHexAnimate", filetype="pdf")
# output as a <a href="visHexAnimate.mp4">mp4</a> file
visHexAnimate(sMap, filename="visHexAnimate", filetype="mp4")
# output as a <a href="visHexAnimate.gif">gif</a> file
visHexAnimate(sMap, filename="visHexAnimate", filetype="gif")

## End(Not run)

visHexBarplot

Function to visualise codebook matrix using barplot for all hexagons or a specific one

Description

visHexBarplot is supposed to visualise codebook matrix using barplot for all hexagons or a specific one

Usage

visHexBarplot(sObj, which.hexagon = NULL, height = 7, margin = rep(0.1, 4),
              colormap = c("customized", "bwr", "jet", "gbr", "wy", "br",
              "rainbow", "wb"), customized.color = "red", zeropattern.color = "gray",
              gp = grid::gpar(cex = 0.7, font = 1, col = "black"), bar.text.cex = 0.8,
              bar.text.srt = 90, newpage = T)

Arguments

- `sObj`: an object of class "sMap" or "sTopol" or "sInit"
- `which.hexagon`: the integer specifying which hexagon to display. If NULL, all hexagons will be visualised
- `height`: a numeric value specifying the height of device
- `margin`: margins as units of length 4 or 1
visHexBarplot

**colormap**
short name for the predefined colormap, and "customized" for custom input (see the next 'customized.color'). The predefined colormap can be one of "jet" (jet colormap), "bwr" (blue-white-red colormap), "gbr" (green-black-red colormap), "wyr" (white-yellow-red colormap), "br" (black-red colormap), "yr" (yellow-red colormap), "wb" (white-black colormap), and "rainbow" (rainbow colormap, that is, red-yellow-green-cyan-blue-magenta). Alternatively, any hyphen-separated HTML color names, e.g. "blue-black-yellow", "royalblue-white-sandybrown", "darkgreen-white-darkviolet". A list of standard color names can be found in http://html-color-codes.info/color-names

customized.color
the customized color for pattern visualisation

zeropattern.color
the color for zero horizontal line

gp
an object of class "gpar". It is the output from a call to the function "gpar" (i.e., a list of graphical parameter settings)

bar.text.cex
a numerical value giving the amount by which bar text should be magnified relative to the default (i.e., 1)

bar.text.srt
a numerical value giving the angle by which bar text should be orientated

newpage
logical to indicate whether to open a new page. By default, it sets to true for opening a new page

**Value**

invisible

**Note**

none

**See Also**

sPipeline, visColormap

**Examples**

# 1) generate data with an iid matrix of 1000 x 9
data <- cbind(matrix(rnorm(1000*3,mean=0,sd=1), nrow=1000, ncol=3),
matrix(rnorm(1000*3,mean=0.5,sd=1), nrow=1000, ncol=3),
matrix(rnorm(1000*3,mean=-0.5,sd=1), nrow=1000, ncol=3))
colnames(data) <- c("S1","S1","S1","S2","S2","S2","S3","S3","S3")

# 2) sMap resulted from using by default setup
sMap <- sPipeline(data=data)

# 3) plot codebook patterns using different types
# 3a) for all hexagons
visHexBarplot(sMap)
# 3b) only for the first hexagon
visHexBarplot(sMap, which.hexagon=1)
visHexComp

Function to visualise a component plane of a supra-hexagonal grid

Description

visHexComp is supposed to visualise a supra-hexagonal grid in the context of viewport

Usage

```r
visHexComp(sMap, comp, margin = rep(0.6, 4), area.size = 1, 
colormap = c("bwr", "jet", "gbr", "wyr", "br", "yr", "rainbow", "wb"), 
colors = 40, zlim = c(0, 1), border.color = "transparent", 
newpage = T)
```

Arguments

- **sMap**: an object of class "sMap"
- **comp**: a component/column of codebook matrix from an object "sMap"
- **margin**: margins as units of length 4 or 1
- **area.size**: an integer or a vector specifying the area size of each hexagon
- **colormap**: short name for the colormap. It can be one of "jet" (jet colormap), "bwr" (blue-white-red colormap), "gbr" (green-black-red colormap), "wyr" (white-yellow-red colormap), "br" (black-red colormap), "yr" (yellow-red colormap), "wb" (white-black colormap), and "rainbow" (rainbow colormap, that is, red-yellow-green-cyan-blue-magenta). Alternatively, any hyphen-separated HTML color names, e.g. "blue-black-yellow", "royalblue-white-sandybrown", "darkgreen-white-darkviolet". A list of standard color names can be found in [http://html-color-codes.info/color-names](http://html-color-codes.info/color-names)
- **ncolors**: the number of colors specified
- **zlim**: the minimum and maximum z values for which colors should be plotted, defaulting to the range of the finite values of z. Each of the given colors will be used to color an equispaced interval of this range. The midpoints of the intervals cover the range, so that values just outside the range will be plotted
- **border.color**: the border color for each hexagon
- **newpage**: a logical to indicate whether or not to open a new page

Value

- **invisible**

Note

- **none**

See Also

- `visColormap`, `visHexGrid`
Examples

# 1) generate an iid normal random matrix of 100x10
data <- matrix( rnorm(100*10,mean=0, sd=1), nrow=100, ncol=10)
colnames(data) <- paste(rep('Var',10), seq(1:10), sep='')

# 2) sMap resulted from using by default setup
sMap <- sPipeline(data=data)

# 3) visualise the first component plane with a supra-hexagonal grid
visHexComp(sMap, comp=sMap$codebook[,1], colormap="jet", ncolors=100, zlim=c(-1,1))

visHexGrid

Function to visualise a supra-hexagonal grid

Description

visHexGrid is supposed to visualise a supra-hexagonal grid

Usage

visHexGrid(hbin, area.size = 1, border.color = NULL, fill.color = NULL,
  lty = 1, lwd = 1, lineend = "round", linejoin = "round")

Arguments

  hbin  an object of class "hexbin"
  area.size  an integer or a vector specifying the area size of each hexagon
  border.color  the border color for each hexagon
  fill.color  the filled color for each hexagon
  lty  the line type for each hexagon. 0 for 'blank', 1 for 'solid', 2 for 'dashed', 3 for 'dotted', 4 for 'dotdash', 5 for 'longdash', 6 for 'twodash'
  lwd  the line width for each hexagon
  lineend  the line end style for each hexagon. It can be one of 'round', 'butt' and 'square'
  linejoin  the line join style for each hexagon. It can be one of 'round', 'mitre' and 'bevel'

Value

  invisible

Note

  none

See Also

  visHexComp
Examples

# 1) generate an iid normal random matrix of 100x10
data <- matrix(rnorm(100*10, mean=0, sd=1), nrow=100, ncol=10)
colnames(data) <- paste(rep('Var', 10), seq(1:10), sep="")

# 2) sMap resulted from using by default setup
sMap <- sPipeline(data=data)

# 3) create an object of "hexbin" class from sMap
dat <- data.frame(sMap$coord)
xdim <- sMap$xdim
ydim <- sMap$ydim
hbin <- hexbin::hexbin(dat$x, dat$y, xbins=xdim-1,
shape=sqrt(0.75)*ydim/xdim)

# 4) visualise hbin object
vp <- hexbin::hexViewport(hbin)
visHexGrid(hbin)

---

visHexMapping

Function to visualise various mapping items within a supra-hexagonal grid

Description

visHexMapping is supposed to visualise various mapping items within a supra-hexagonal grid

Usage

visHexMapping(sObj, mappingType = c("indexes", "hits", "dist",
"antidist", 
"bases", "customized"), labels = NULL, height = 7, margin = rep(0.1,
4),
area.size = 1, gp = grid::gpar(cex = 0.7, font = 1, col = "black"),
border.color = NULL, fill.color = "transparent", lty = 1, lwd = 1,
lineend = "round", linejoin = "round", clip = c("on", "inherit", 
"off"),
newpage = T)

Arguments

sObj
an object of class "sMap" or "sInit" or "sTopol"
mappingType the mapping type, can be "indexes", "hits", "dist", "antidist", "bases", and "customized"
labels NULL or a vector with the length of nHex
height a numeric value specifying the height of device
margin margins as units of length 4 or 1
area.size an inteter or a vector specifying the area size of each hexagon
gp an object of class "gpar". It is the output from a call to the function "gpar" (i.e.,
a list of graphical parameter settings)
visHexMapping

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>border.color</td>
<td>the border color for each hexagon</td>
</tr>
<tr>
<td>fill.color</td>
<td>the filled color for each hexagon</td>
</tr>
<tr>
<td>lty</td>
<td>the line type for each hexagon. 0 for 'blank', 1 for 'solid', 2 for 'dashed', 3 for 'dotted', 4 for 'dotdash', 5 for 'longdash', 6 for 'twodash'</td>
</tr>
<tr>
<td>lwd</td>
<td>the line width for each hexagon</td>
</tr>
<tr>
<td>lineend</td>
<td>the line end style for each hexagon. It can be one of 'round', 'butt' and 'square'</td>
</tr>
<tr>
<td>linejoin</td>
<td>the line join style for each hexagon. It can be one of 'round', 'mitre' and 'bevel'</td>
</tr>
<tr>
<td>clip</td>
<td>either &quot;on&quot; for clipping to the extent of this viewport, &quot;inherit&quot; for inheriting the clipping region from the parent viewport, or &quot;off&quot; to turn clipping off altogether</td>
</tr>
<tr>
<td>newpage</td>
<td>logical to indicate whether to open a new page. By default, it sets to true for opening a new page</td>
</tr>
</tbody>
</table>

**Value**

invisible

**Note**

The mappingType includes:

- "indexes": the index of hexagons in a supra-hexagonal grid
- "hits": the number of input data vectors hitting the hexagons
- "dist": distance (in high-dimensional input space) to neighbors (defined in 2D output space)
- "antidist": the oppose version of "dist"
- "bases": clusters partitioned from the sMap
- "customized": displaying input "labels"

**See Also**

sDmat, sDmatCluster, visHexGrid

**Examples**

```R
# 1) generate data with an iid matrix of 1000 x 9
data <- cbind(matrix(rnorm(1000*3, mean=0, sd=1), nrow=1000, ncol=3),
               matrix(rnorm(1000*3, mean=0.5, sd=1), nrow=1000, ncol=3),
               matrix(rnorm(1000*3, mean=-0.5, sd=1), nrow=1000, ncol=3))
colnames(data) <- c("S1","S1","S1","S2","S2","S2","S3","S3","S3")

# 2) sMap resulted from using by default setup
sMap <- sPipeline(data=data)

# 3) visualise supported mapping items within a supra-hexagonal grid
# 3a) for indexes of hexagons
visHexMapping(sMap, mappingType="indexes")
# 3b) for the number of input data vectors hitting the hexagons
visHexMapping(sMap, mappingType="hits")
# 3c) for distance (in high-dimensional input space) to neighbors (defined in 2D output space)
visHexMapping(sMap, mappingType="dist")
# 3d) for clusters/bases partitioned from the sMap
visHexMapping(sMap, mappingType="bases")
```
**visHexMulComp**

*Function to visualise multiple component planes of a supra-hexagonal grid*

**Description**

`visHexMulComp` is supposed to visualise multiple component planes of a supra-hexagonal grid.

**Usage**

```r
g = visHexMulComp(sMap, which.components = NULL, rect.grid = NULL, margin = rep(0.1, 4), height = 7, title.rotate = 0, title.xy = c(0.45, 1), colormap = c("bwr", "jet", "gbr", "wyr", "br", "yr", "rainbow", "wb"), ncolors = 40, zlim = NULL, border.color = "transparent", gp = grid::gpar(), newpage = T)
```

**Arguments**

- `sMap`: an object of class "sMap"
- `which.components`: an integer vector specifying which components will be visualised. By default, it is `NULL` meaning all components will be visualised.
- `rect.grid`: a vector specifying the number of rows and columns for a rectangle grid wherein the component planes are placed. By default, it is `NULL` (decided on according to the number of component planes that will be visualised).
- `margin`: margins as units of length 4 or 1.
- `height`: a numeric value specifying the height of device.
- `title.rotate`: the rotation of the title.
- `colormap`: short name for the colormap. It can be one of "jet" (jet colormap), "bwr" (blue-white-red colormap), "gbr" (green-black-red colormap), "wyr" (white-yellow-red colormap), "br" (black-red colormap), "yr" (yellow-red colormap), "wb" (white-black colormap), and "rainbow" (rainbow colormap, that is, red-yellow-green-cyan-blue-magenta). Alternatively, any hyphen-separated HTML color names, e.g. "blue-black-yellow", "royalblue-white-sandybrown", "darkgreen-white-darkviolet". A list of standard color names can be found in [http://html-color-codes.info/color-names](http://html-color-codes.info/color-names).
- `ncolors`: the number of colors specified.
- `zlim`: the minimum and maximum z values for which colors should be plotted, defaulting to the range of the finite values of z. Each of the given colors will be used to color an equispaced interval of this range. The midpoints of the intervals cover the range, so that values just outside the range will be plotted.
- `border.color`: the border color for each hexagon.
- `gp`: an object of class gpar, typically the output from a call to the function gpar (i.e., a list of graphical parameter settings).
- `newpage`: logical to indicate whether to open a new page. By default, it sets to true for opening a new page.
Value

invisible

Note

none

See Also

visVp, visHexComp, visColorbar

Examples

# 1) generate data with an iid matrix of 1000 x 3
data <- cbind(matrix(rnorm(1000*3, mean=0, sd=1), nrow=1000, ncol=3),
                matrix(rnorm(1000*3, mean=0.5, sd=1), nrow=1000, ncol=3),
                matrix(rnorm(1000*3, mean=-0.5, sd=1), nrow=1000, ncol=3))
colnames(data) <- c("S1", "S2", "S3")

# 2) sMap resulted from using by default setup
sMap <- sPipeline(data=data)

# 3) visualise multiple component planes of a supra-hexagonal grid
visHexMulComp(sMap, colormap="jet", ncolors=20, zlim=c(-1,1),
                gp=grid::gpar(cex=0.8))

# 3a) visualise only the first 6 component planes
visHexMulComp(sMap, which.components=1:6, colormap="jet", ncolors=20,
                zlim=c(-1,1), gp=grid::gpar(cex=0.8))

# 3b) visualise only the first 6 component planes within the rectangle grid of 3 X 2
visHexMulComp(sMap, which.components=1:6, rect.grid=c(3,2),
                colormap="jet", ncolors=20, zlim=c(-1,1), gp=grid::gpar(cex=0.8))

visHexPattern

Function to visualise codebook matrix or input patterns within a supra-hexagonal grid

Description

visHexPattern is supposed to codebook matrix or input patterns within a supra-hexagonal grid.

Usage

visHexPattern(sObj, plotType = c("lines", "bars", "radars"), pattern = NULL,
              height = 7, margin = rep(0.1, 4), colormap = c("customized", "bwr",
                          "jet", "gbr", "wyr", "br", "yr", "rainbow", "wb"),
              customized.color = "red",
              alterntive.color = c("transparent", "gray"), zeropattern.color = "gray",
              legend = T, legend.cex = 0.8, legend.label = NULL, newpage = T)
Arguments

sObj
an object of class "sMap" or "sTopol" or "sInit"

plotType
the plot type, can be "lines" for line/point graph, "bars" for bar graph, "radars" for radar graph

pattern
By default, it sets to "NULL" for the codebook matrix. It is intended for the user-input patterns, i.e., a matrix with the dimension of nHex x nPattern, where nHex is the number of hexagons and nPattern is the number of elements for each pattern

height
a numeric value specifying the height of device

margin
margins as units of length 4 or 1

colormap
short name for the predefined colormap, and "customized" for custom input (see the next 'customized.color'). The predefined colormap can be one of "jet" (jet colormap), "bwr" (blue-white-red colormap), "gbr" (green-black-red colormap), "wyr" (white-yellow-red colormap), "br" (black-red colormap), "yr" (yellow-red colormap), "wb" (white-black colormap), and "rainbow" (rainbow colormap, that is, red-yellow-green-cyan-blue-magenta). Alternatively, any hyphen-separated HTML color names, e.g. "blue-black-yellow", "royalblue-white-sandybrown", "darkgreen-white-darkviolet". A list of standard color names can be found in http://html-color-codes.info/color-names

customized.color
the customized color for pattern visualisation

alternative.color
the alternative color used to indicate the hexagon layout

zeropattern.color
the color for zero horizontal line

legend
logical to indicate whether to add the legend

legend.cex
a numerical value giving the amount by which legend text should be magnified relative to the default (i.e., 1)

legend.label
a vector specifying the legend label. By default, it is NULL for using column names of the codebook matrix (or the matrix given by the parameter 'pattern')

newpage
logical to indicate whether to open a new page. By default, it sets to true for opening a new page

Value

invisible

Note

The "plotType" includes:

• "lines": line plot. If multiple colors are given, the points are also plotted. When the pattern involves both positive and negative values, zero horizontal line is also shown

• "bars": bar plot. When the pattern involves both positive and negative values, the zero horizontal line is in the middle of the hexagon; otherwise at the top of the hexagon for all negative values, and at the bottom for all positive values

• "radars": radar plot. Each radar diagram represents one pattern, wherein each element value is proportional to the distance from the center. Note, it starts on the right and wind counterclockwise around the circle
See Also

sPipeline, visColormap

Examples

# 1) generate data with an iid matrix of 1000 x 9
data <- cbind(matrix(rnorm(1000*3,mean=0,sd=1), nrow=1000, ncol=3),
matrix(rnorm(1000*3,mean=0.5,sd=1), nrow=1000, ncol=3),
matrix(rnorm(1000*3,mean=-0.5,sd=1), nrow=1000, ncol=3))
colnames(data) <- c("S1","S1","S1","S2","S2","S2","S3","S3","S3")

# 2) sMap resulted from using by default setup
sMap <- sPipeline(data=data)

# 3) plot codebook patterns using different types
# 3a) line plot
visHexPattern(sMap, plotType=\"lines\")
# 3b) bar plot
visHexPattern(sMap, plotType=\"bars\")
# 3c) radar plot
visHexPattern(sMap, plotType=\"radars\")
# 4) plot user-input patterns using different types
# 4a) generate pattern data with two different groups "S" and "T"
nHex <- sMap$nHex
pattern <- cbind(matrix(runif(nHex*3,min=0,max=1), nrow=nHex, ncol=3),
matrix(runif(nHex*3,min=1,max=2), nrow=nHex, ncol=3))
colnames(pattern) <- c("S1","S2","S3","T1","T2","T3")
# 4b) for line plot
visHexPattern(sMap, plotType=\"lines\", pattern=pattern,
customized.color=\"red\", zeropattern.color=\"gray\")
# 4c) for bar plot
visHexPattern(sMap, plotType=\"bars\", pattern=pattern,
customized.color=rep(c(\"red\",\"green\"),each=3))
visHexPattern(sMap, plotType=\"bars\", pattern=pattern,
customized.color=rep(c(\"red\",\"green\"),each=3), legend.label=c(\"S\",\"T\")
# 4d) for radar plot
visHexPattern(sMap, plotType=\"radars\", pattern=pattern,
customized.color=rep(c(\"red\",\"green\"),each=3))
visHexPattern(sMap, plotType=\"radars\", pattern=pattern,
customized.color=rep(c(\"red\",\"green\"),each=3), legend.label=c(\"S\",\"T\")

visKernels

Function to visualize neighborhood kernels

Description

visKernels is supposed to visualize a series of neighborhood kernels, each of which is a non-increasing functions of: i) the distance $d_{wi}$ between the hexagon/rectangle $i$ and the winner $w$, and ii) the radius $\delta_t$ at time $t$.

Usage

visKernels(newpage = T)
Arguments

newpage logical to indicate whether to open a new page. By default, it sets to true for opening a new page.

Value

invisible

Note

There are five kernels that are currently supported:

• For "gaussian" kernel, \( h_{wi}(t) = e^{-d_{wi}^2/(2\delta_t^2)} \)
• For "cutguassian" kernel, \( h_{wi}(t) = e^{-d_{wi}^2/(2\delta_t^2)} \ast (d_{wi} \leq \delta_t) \)
• For "bubble" kernel, \( h_{wi}(t) = (d_{wi} \leq \delta_t) \)
• For "ep" kernel, \( h_{wi}(t) = (1 - d_{wi}^2/\delta_t^2) \ast (d_{wi} \leq \delta_t) \)
• For "gamma" kernel, \( h_{wi}(t) = 1/\Gamma(d_{wi}^2/(4*\delta_t^2) + 2) \)

These kernels above are displayed within a plot for each fixed radius. Three different radii (i.e., 1 and 2) are illustrated.

See Also

sTrainSeq, sTrainBatch

Examples

# visualise currently supported five kernels
visKernels()

visTreeBootstrap

Function to build and visualise the bootstrapped tree

Description

visTreeBootstrap is supposed to build the tree, perform bootstrap analysis and visualise the bootstrapped tree. It returns an object of class "phylo". For easy downstream analysis, the bootstrapped tree is rerooted either at the internal node with the miminum bootstrap/confidence value or at any customised internal node.

Usage

visTreeBootstrap(data, algorithm = c("nj", "fastme.ols", "fastme.bal"), metric = c("euclidean", "pearson", "spearman", "cos", "manhattan", "kendall", "mi"), num.bootstrap = 100, consensus = FALSE, consensus.majority = 0.5, reroot = "min.bootstrap", plot.phylo.arg = NULL, nodelabels.arg = NULL, visTree = T, verbose = T, ...)
Arguments

data an input data matrix used to build the tree. The built tree describes the relationships between rows of input matrix.

algorithm the tree-building algorithm. It can be one of "nj" for the neighbor-joining tree estimation, "fastme.ols" for the minimum evolution algorithm with ordinary least-squares (OLS) fitting of a metric to a tree structure, and "fastme.bal" for the minimum evolution algorithm under a balanced (BAL) weighting scheme.

metric distance metric used to calculate a distance matrix between rows of input matrix. It can be: "pearson" for pearson correlation, "spearman" for spearman rank correlation, "kendall" for kendall tau rank correlation, "euclidean" for euclidean distance, "manhattan" for cityblock distance, "cos" for cosine similarity, "mi" for mutual information.

tip.label the labels for tip nodes. Tip labels come from the row names of the input matrix, but are not necessarily the same order as they appear in the input matrix.

Value

an object of class "phylo". It can return a bootstrapped tree or a consensus tree (if enabled): When a bootstrapped tree is returned (also visualised by default), the "phylo" object has a list with following components:

- Nnode: the number of internal nodes
- node.label: the labels for internal nodes. Here, each internal node is associated with the bootstrap value
- tip.label: the labels for tip nodes. Tip labels come from the row names of the input matrix, but are not necessarily the same order as they appear in the input matrix.
visTreeBootstrap

- **edge**: a two-column matrix describing the links between tree nodes (including internal and tip nodes)
- **edge\.length**: a vector indicating the edge length in the 'edge'
- **Note**: the tree structure is indexed with 1:Ntip for tip nodes, and (Ntip+1):(Ntip+Nnode) for internal nodes, where Ntip is the number of tip nodes and Nnode for the number of internal nodes. Moreover, nrow(data) = Ntip = Nnode - 2.

When a consensus tree is returned (no visualisation), the "phylo" object has a list with following components:

- **Nnode**: the number of internal nodes
- **tip\.label**: the lables for tip nodes. Tip labels come from the row names of the input matrix, but are not necessarily the same order as they appear in the input matrix
- **edge**: a two-column matrix describing the links between tree nodes (including internal and tip nodes)

Note

A list of main parameters used in the function "ape::plot.phylo":

- **"type"**: a character string specifying the type of phylogeny to be drawn; it must be one of "phylogram" (the default), "cladogram", "fan", "unrooted", "radial" or any unambiguous abbreviation of these
- **"direction"**: a character string specifying the direction of the tree. Four values are possible: "rightwards" (the default), "leftwards", "upwards", and "downwards"
- **"lab4ut"**: (= labels for unrooted trees) a character string specifying the display of tip labels for unrooted trees: either "horizontal" where all labels are horizontal (the default), or "axial" where the labels are displayed in the axis of the corresponding terminal branches. This option has an effect only if type = "unrooted"
- **"edge\.color"**: a vector of mode character giving the colours used to draw the branches of the plotted phylogeny. These are taken to be in the same order than the component edge of phy. If fewer colours are given than the length of edge, then the colours are recycled
- **"edge\.width"**: a numeric vector giving the width of the branches of the plotted phylogeny. These are taken to be in the same order than the component edge of phy. If fewer widths are given than the length of edge, then these are recycled
- **"edge\.lty"**: same than the previous argument but for line types; 1: plain, 2: dashed, 3: dotted, 4: dotdash, 5: longdash, 6: twodash
- **"font"**: an integer specifying the type of font for the labels: 1 (plain text), 2 (bold), 3 (italic, the default), or 4 (bold italic)
- **"cex"**: a numeric value giving the factor scaling of the tip and node labels (Character EXPansion). The default is to take the current value from the graphical parameters
- **"adj"**: a numeric specifying the justification of the text strings of the labels: 0 (left-justification), 0.5 (centering), or 1 (right-justification). This option has no effect if type="unrooted". If NULL (the default) the value is set with respect of direction (see details)
- **"srt"**: a numeric giving how much the labels are rotated in degrees (negative values are allowed resulting in clock-like rotation); the value has an effect respectively to the value of direction (see Examples). This option has no effect if type="unrooted"
- **"no.margin"**: a logical. If TRUE, the margins are set to zero and the plot uses all the space of the device
• "label.offset": a numeric giving the space between the nodes and the tips of the phylogeny and their corresponding labels. This option has no effect if type="unrooted"

• "rotate.tree": for "fan", "unrooted", or "radial" trees: the rotation of the whole tree in degrees (negative values are accepted)

A list of main parameters used in the function "ape::nodelabels":

• "text": a vector of mode character giving the text to be printed. By default, the labels for internal nodes (see "node.label"), that is, the bootstrap values associated with internal nodes

• "node": a vector of mode numeric giving the numbers of the nodes where the text or the symbols are to be printed. By default, indexes for internal nodes, that is, \((N_{\text{tip}}+1):(N_{\text{tip}}+N_{\text{node}})\), where \(N_{\text{tip}}\) is the number of tip nodes and \(N_{\text{node}}\) for the number of internal nodes

• "adj": one or two numeric values specifying the horizontal and vertical, respectively, justification of the text or symbols. By default, the text is centered horizontally and vertically. If a single value is given, this alters only the horizontal position of the text

• "frame": a character string specifying the kind of frame to be printed around the text. This must be one of "rect" (the default), "circle", "none", or any unambiguous abbreviation of these

• "cex": a numeric value giving the factor scaling of the tip and node labels (Character EXpansion). The default is to take the current value from the graphical parameters

• "font": an integer specifying the type of font for the labels: 1 (plain text), 2 (bold), 3 (italic, the default), or 4 (bold italic)

• "col": a character string giving the color to be used for the text or the plotting symbols; this is eventually recycled

• "bg": a character string giving the color to be used for the background of the text frames or of the plotting symbols if it applies; this is eventually recycled. It can be one of "jet" (jet colormap), "bwr" (blue-white-red colormap), "gbr" (green-black-red colormap), "wyr" (white-yellow-red colormap), "br" (black-red colormap), "yr" (yellow-red colormap), "wb" (white-black colormap), and "rainbow" (rainbow colormap, that is, red-yellow-green-cyan-blue-magenta). Alternatively, any hyphen-separated HTML color names, e.g. "blue-black-yellow", "royalblue-white-sandybrown", "darkgreen-white-darkviolet". A list of standard color names can be found in http://html-color-codes.info/color-names

See Also

visTreeBootstrap

Examples

# 1) generate an iid normal random matrix of 100x10
data <- matrix( rnorm(100*10,mean=0,sd=1), nrow=100, ncol=10)
colnames(data) <- paste(rep("\'Var","S'|10), seq(1:10), sep="")
data <- t(data)

# 2) build neighbor-joining tree with bootstrap values and visualise it by default
visTreeBootstrap(data)

# 3) only display those internal nodes with bootstrap values > 30
# 3a) generate the bootstrapped tree (without visualisation)
tree.bs <- visTreeBootstrap(data, visTree=FALSE)
# 3b) look at the bootstrap values and ordered row names of input matrix
# the bootstrap values
tree_bs$node.label
# ordered row names of input matrix

tree_bs$tip.label
# 3c) determine internal nodes that should be displayed

Ntip <- length(tree_bs$tip.label) # number of tip nodes
Nnode <- length(tree_bs$node.label) # number of internal nodes

flag <- as.numeric(tree_bs$node.label) > 30

text <- tree_bs$node.label[flag]

node <- Ntip + 1:Nnode[flag]

visTreeBootstrap(data, nodelabels.arg=list(text=text, node=node))

# 4) obtain the consensus tree
tree_cons <- visTreeBootstrap(data, consensus=TRUE, num.bootstrap=10)

---

*visTreeBSclust*  
*Function to obtain clusters from a bootstrapped tree*

**Description**

*visTreeBSclust* is supposed to obtain clusters from a bootstrapped tree.

**Usage**

visTreeBSclust(tree_bs, bootstrap.cutoff = 80, max.fraction = 1, min.size = 3, visTree = T, plot.phylo.arg = NULL, nodelabels.arg = NULL, verbose = T, ...)

**Arguments**

- **tree bs**: an "phylo" object storing a bootstrapped tree
- **bootstrap.cutoff**: an integer specifying bootstrap-derived clusters
- **max.fraction**: the maximum fraction of leaves contained in a cluster
- **min.size**: the minimum number of leaves contained in a cluster
- **visTree**: logical to indicate whether the tree will be visualised. By default, it sets to true for display
- **plot.phylo.arg**: a list of main parameters used in the function "ape::plot.phylo"  
  [http://www.inside-r.org/packages/cran/ape/docs/plot.phylo](http://www.inside-r.org/packages/cran/ape/docs/plot.phylo). See 'Note' below for details on the parameters
- **nodelabels.arg**: a list of main parameters used in the function "ape::nodelabels"  
  [http://www.inside-r.org/packages/cran/ape/docs/nodelabels](http://www.inside-r.org/packages/cran/ape/docs/nodelabels). See 'Note' below for details on the parameters
- **verbose**: logical to indicate whether the messages will be displayed in the screen. By default, it sets to true for display
- **...**: additional "ape::plot.phylo" parameters
Value

a data frame following components:

- **Samples**: the labels for tip nodes (samples)
- **Clusters**: the clusters each tip node belongs to; unassigned tip nodes will be the cluster called 'C0'
- **Clans**: the internal node id for each cluster

Note

A list of main parameters used in the function "ape::plot.phylo":

- "type": a character string specifying the type of phylogeny to be drawn; it must be one of "phylogram" (the default), "cladogram", "fan", "unrooted", "radial" or any unambiguous abbreviation of these
- "direction": a character string specifying the direction of the tree. Four values are possible: "rightwards" (the default), "leftwards", "upwards", and "downwards"
- "lab4ut": (= labels for unrooted trees) a character string specifying the display of tip labels for unrooted trees: either "horizontal" where all labels are horizontal (the default), or "axial" where the labels are displayed in the axis of the corresponding terminal branches. This option has an effect only if type = "unrooted"
- "edge.color": a vector of mode character giving the colours used to draw the branches of the plotted phylogeny. These are taken to be in the same order than the component edge of phy. If fewer colours are given than the length of edge, then the colours are recycled
- "edge.width": a numeric vector giving the width of the branches of the plotted phylogeny. These are taken to be in the same order than the component edge of phy. If fewer widths are given than the length of edge, then these are recycled
- "edge.lty": same than the previous argument but for line types; 1: plain, 2: dashed, 3: dotted, 4: dotdash, 5: longdash, 6: twodash
- "font": an integer specifying the type of font for the labels: 1 (plain text), 2 (bold), 3 (italic, the default), or 4 (bold italic)
- "cex": a numeric value giving the factor scaling of the tip and node labels (Character EXpansion). The default is to take the current value from the graphical parameters
- "adj": a numeric specifying the justification of the text strings of the labels: 0 (left-justification), 0.5 (centering), or 1 (right-justification). This option has no effect if type="unrooted". If NULL (the default) the value is set with respect of direction (see details)
- "srt": a numeric giving how much the labels are rotated in degrees (negative values are allowed resulting in clock-like rotation); the value has an effect respectively to the value of direction (see Examples). This option has no effect if type="unrooted"
- "no.margin": a logical. If TRUE, the margins are set to zero and the plot uses all the space of the device
- "label.offset": a numeric giving the space between the nodes and the tips of the phylogeny and their corresponding labels. This option has no effect if type="unrooted"
- "rotate.tree": for "fan", "unrooted", or "radial" trees: the rotation of the whole tree in degrees (negative values are accepted)

A list of main parameters used in the function "ape::nodelabels":

- "text": a vector of mode character giving the text to be printed. By default, the labels for internal nodes (see "node.label"), that is, the bootstrap values associated with internal nodes
visVp

Function to create viewports for multiple supra-hexagonal grids

visVp is supposed to create viewports, which describe rectangular regions on a graphics device and define a number of coordinate systems for each of supra-hexagonal grids.

Description

visVp is supposed to create viewports, which describe rectangular regions on a graphics device and define a number of coordinate systems for each of supra-hexagonal grids.

Examples

# 1) generate an iid normal random matrix of 100x10
data <- matrix(rnorm(100*10, mean=0, sd=1), nrow=100, ncol=10)
colnames(data) <- paste(rep("Var",10), seq(1:10), sep="")
data <- t(data)

# 2) build neighbor-joining tree with bootstrap values and visualise it by default
tree_bs <- visTreeBootstrap(data)

# 3) obtain clusters from a bootstrapped tree
res <- visTreeBSclust(tree_bs, bootstrap.cutoff=80)
## hide tip labels and modify the font of internal node labels
res <- visTreeBSclust(tree_bs, bootstrap.cutoff=80,
nodelabels.arg=list(cex=0.4), show.tip.label=FALSE)

See Also

visTreeBootstrap
Usage

visVp(height = 7, xdim = 1, ydim = 1, colNum = 1, rowNum = 1, gp = grid::gpar(), newpage = T)

Arguments

height a numeric value specifying the height of device
xdim an integer specifying x-dimension of the grid
ydim an integer specifying y-dimension of the grid
colNum an integer specifying the number of columns
rowNum an integer specifying the number of rows
gp an object of class gpar, typically the output from a call to the function gpar (i.e., a list of graphical parameter settings)
newpage logical to indicate whether to open a new page. By default, it sets to true for opening a new page

Value

vpnames an R object of "viewport" class

Note

none

See Also

visHexMulComp, visCompReorder

Examples

# 1) create 5x5 viewports
vpnames <- visVp(colNum=5, rowNum=5)

# 2) look at names of these viewports
vpnames

Xiang Arabidopsis embryo gene expression dataset from Xiang et al. (2011)

Description

Arabidopsis embryo dataset contains gene expression levels (3625 genes and 7 embryo samples) from Xiang et al. (2011). This dataset has been pre-processed: capping into floor of intensity 777.6; 2-base logarithmic transformation; row/gene centering; and keeping genes with at least 2-fold changes (in any stage) as compared to the average over embryo stages.

Usage

data(Xiang)
Value

- Xiang: a gene expression matrix of 3625 genes x 7 stage samples. These embryo stages are: zygote, quadrant, globular, heart, torpedo, bent, and mature.

References

Index

*Topic datasets
  Fang, 2
  Golub, 3
  Xiang, 63

Fang, 2

Golub, 3

sBMH, 4, 6, 10, 17, 20, 21, 29
sCompReorder, 5, 34
sDistance, 5, 6, 7, 10
sDmat, 9, 11, 12, 35, 51
sDmatCluster, 8, 10, 35, 37, 51
sDmatMinima, 10, 11
sHexDist, 12, 17, 18
sHexGrid, 13, 22
sInitial, 12, 14, 20, 21, 26
sMapOverlay, 16
sNeighAny, 9, 12, 17
sNeighDirect, 10, 17, 18
sPipeline, 4, 6, 10, 17, 19, 47, 55
sTopology, 6, 12, 14, 15, 20, 21, 21
sTrainBatch, 21, 23, 56
sTrainology, 20, 21, 24, 25, 28
sTrainSeq, 20, 21, 27, 56
sWriteData, 29

visColoralpha, 30, 32
visColorbar, 31, 34, 53
visColormap, 30, 31, 32, 35, 47, 48, 55
visCompReorder, 6, 31, 33, 63
visDmatCluster, 10, 34
visDmatHeatmap, 36
visHeatmap, 38, 39
visHeatmapAdv, 37, 40, 42
visHexAnimate, 43
visHexBarplot, 46
visHexComp, 34, 48, 49, 53
visHexGrid, 35, 48, 49, 51
visHexMapping, 22, 50
visHexMulComp, 17, 21, 31, 46, 52, 63
visHexPattern, 53
visKernels, 24, 28, 55
visTreeBootstrap, 56, 59, 62
visTreeBSclust, 60
visVp, 34, 53, 62

Xiang, 63

65