Package 'dbscan'

November 13, 2018

Version 1.1-3

Date 2018-11-12

Title Density Based Clustering of Applications with Noise (DBSCAN) and Related Algorithms

Description A fast reimplementation of several density-based algorithms of the DBSCAN family for spatial data. Includes the DBSCAN (density-based spatial clustering of applications with noise) and OPTICS (ordering points to identify the clustering structure) clustering algorithms HDBSCAN (hierarchical DBSCAN) and the LOF (local outlier factor) algorithm. The implementations use the kd-tree data structure (from library ANN) for faster k-nearest neighbor search. An R interface to fast kNN and fixed-radius NN search is also provided.

Imports Rcpp (>= 0.12.12), graphics, stats, methods

LinkingTo Rcpp

Suggests fpc, microbenchmark, testthat, dendextend, igraph, knitr, DMwR

VignetteBuilder knitr

BugReports https://github.com/mhahsler/dbscan

License GPL (>= 2)

Copyright ANN library is copyright by University of Maryland, Sunil Arya and David Mount. All other code is copyright by Michael Hahsler and Matthew Piekenbrock.

SystemRequirements C++11

NeedsCompilation yes

Author Michael Hahsler [aut, cre, cph], Matthew Piekenbrock [aut, cph], Sunil Arya [ctb, cph], David Mount [ctb, cph]

Maintainer Michael Hahsler <mhahsler@lyle.smu.edu>

Repository CRAN

Date/Publication 2018-11-13 22:50:48 UTC

R topics documented:

`dbscan` *DBSCAN*

Description

Fast reimplementation of the DBSCAN (Density-based spatial clustering of applications with noise) clustering algorithm using a kd-tree. The implementation is significantly faster and can work with larger data sets then dbscan in **fpc**.

Usage

```
dbscan(x, eps, minPts = 5, weights = NULL, borderPoints = TRUE, ...)

## S3 method for class 'dbscan_fast'
predict(object, newdata = NULL, data, ...)
```

Arguments

`x`	a data matrix or a dist object. Alternatively, a frNN object with fixed-radius nearest neighbors can also be specified. In this case eps can be missing and will be taken form the frNN object.
`eps`	size of the epsilon neighborhood.
`minPts`	number of minimum points in the eps region (for core points). Default is 5 points.

weights	numeric; weights for the data points. Only needed to perform weighted clustering.
borderPoints	logical; should border points be assigned. The default is TRUE for regular DBSCAN. If FALSE then border points are considered noise (see DBSCAN* in Campello et al, 2013).
object	a DBSCAN clustering object.
data	the data set used to create the DBSCAN clustering object.
newdata	new data set for which cluster membership should be predicted.
...	additional arguments are passed on to fixed-radius nearest neighbor search algorithm. See frNN for details on how to control the search strategy.

Details

Note: use dbscan::dbscan to call this implementation when you also use package **fpc**.

This implementation of DBSCAN implements the original algorithm as described by Ester et al (1996). DBSCAN estimates the density around each data point by counting the number of points in a user-specified eps-neighborhood and applies a used-specified minPts thresholds to identify core, border and noise points. In a second step, core points are joined into a cluster if they are density-reachable (i.e., there is a chain of core points where one falls inside the eps-neighborhood of the next). Finally, border points are assigned to clusters. The algorithm only needs parameters eps and minPts.

Border points are arbitrarily assigned to clusters in the original algorithm. DBSCAN* (see Campello et al 2013) treats all border points as noise points. This is implemented with borderPoints = FALSE.

Setting parameters for DBSCAN: minPts is often set to be dimensionality of the data plus one or higher. The knee in kNNdistplot can be used to find suitable values for eps.

See frNN for more information on the parameters related to nearest neighbor search.

A precomputed frNN object can be supplied as x. In this case eps does not need to be specified. This option us useful for large data sets, where a sparse distance matrix is available. See frNN how to create frNN objects.

predict can be used to predict cluster memberships for new data points. A point is considered a member of a cluster if it is within the eps neighborhood of a member of the cluster. Points which cannot be assigned to a cluster will be reported as members of the noise cluster 0.

Value

An object of class 'dbscan_fast' with the following components:

eps	value of the eps parameter.
minPts	value of the minPts parameter.
cluster	A integer vector with cluster assignments. Zero indicates noise points.

Author(s)

Michael Hahsler

References

Martin Ester, Hans-Peter Kriegel, Joerg Sander, Xiaowei Xu (1996). A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. Institute for Computer Science, University of Munich. *Proceedings of 2nd International Conference on Knowledge Discovery and Data Mining (KDD-96).*

Campello, R. J. G. B.; Moulavi, D.; Sander, J. (2013). Density-Based Clustering Based on Hierarchical Density Estimates. *Proceedings of the 17th Pacific-Asia Conference on Knowledge Discovery in Databases, PAKDD 2013,* Lecture Notes in Computer Science 7819, p. 160.

See Also

kNNdistplot, frNN, dbscan in **fpc**.

Examples

```
data(iris)
iris <- as.matrix(iris[,1:4])

## find suitable eps parameter using a k-NN plot for k = dim + 1
## Look for the knee!
kNNdistplot(iris, k = 5)
abline(h=.5, col = "red", lty=2)

res <- dbscan(iris, eps = .5, minPts = 5)
res

pairs(iris, col = res$cluster + 1L)

## use precomputed frNN
fr <- frNN(iris, eps = .5)
dbscan(fr, minPts = 5)

## example data from fpc
set.seed(665544)
n <- 100
x <- cbind(
  x = runif(10, 0, 10) + rnorm(n, sd = 0.2),
  y = runif(10, 0, 10) + rnorm(n, sd = 0.2)
  )

res <- dbscan(x, eps = .3, minPts = 3)
res

## plot clusters and add noise (cluster 0) as crosses.
plot(x, col=res$cluster)
points(x[res$cluster==0,], pch = 3, col = "grey")

hullplot(x, res)

## predict cluster membership for new data points
## (Note: 0 means it is predicted as noise)
```

```
newdata <- x[1:5,] + rnorm(10, 0, .2)
predict(res, newdata, data = x)

## compare speed against fpc version (if microbenchmark is installed)
## Note: we use dbscan::dbscan to make sure that we do now run the
## implementation in fpc.
## Not run:
if (requireNamespace("fpc", quietly = TRUE) &&
    requireNamespace("microbenchmark", quietly = TRUE)) {
  t_dbscan <- microbenchmark::microbenchmark(
    dbscan::dbscan(x, .3, 3), times = 10, unit = "ms")
  t_dbscan_linear <- microbenchmark::microbenchmark(
    dbscan::dbscan(x, .3, 3, search = "linear"), times = 10, unit = "ms")
  t_dbscan_dist <- microbenchmark::microbenchmark(
    dbscan::dbscan(x, .3, 3, search = "dist"), times = 10, unit = "ms")
  t_fpc <- microbenchmark::microbenchmark(
    fpc::dbscan(x, .3, 3), times = 10, unit = "ms")

  r <- rbind(t_fpc, t_dbscan_dist, t_dbscan_linear, t_dbscan)
  r

  boxplot(r,
    names = c('fpc', 'dbscan (dist)', 'dbscan (linear)', 'dbscan (kdtree)'),
    main = "Runtime comparison in ms")

  ## speedup of the kd-tree-based version compared to the fpc implementation
  median(t_fpc$time) / median(t_dbscan$time)
}
## End(Not run)
```

DS3 *DS3: Spatial data with arbitrary shapes*

Description

Contains 8000 2-d points, with 6 "natural" looking shapes, all of which have an sinusoid-like shape that intersects with each cluster.

Usage

```
data("DS3")
```

Format

A data frame with 8000 observations on the following 2 variables.

`X` a numeric vector

`Y` a numeric vector

Details

Originally used as a benchmark data set for the Chameleon clustering algorithm[1] to illustrate the a data set containing arbitrarily shaped spatial data surrounded by both noise and artifacts.

Source

Obtained at https://cs.joensuu.fi/sipu/datasets/

References

Karypis, George, Eui-Hong Han, and Vipin Kumar (1999). "Chameleon: Hierarchical clustering using dynamic modeling." *Computer* 32(8): 68-75.

Examples

```
data(DS3)
plot(DS3, pch=20, cex=0.25)
```

---

extractFOSC *Framework for Optimal Selection of Clusters*

Description

Generic reimplementation of the Framework for Optimal Selection of Clusters (FOSC; Campello et al, 2013). Can be parameterized to perform unsupervised cluster extraction through a stability-based measure, or semisupervised cluster extraction through either a constraint-based extraction (with a stability-based tiebreaker) or a mixed (weighted) constraint and stability-based objective extraction.

Usage

```
extractFOSC(x, constraints = NA, alpha = 0, minPts = 2L,
            prune_unstable = FALSE,
            validate_constraints = FALSE)
```

Arguments

x	a valid hclust object.
constraints	Either a list or matrix of pairwise constraints. If not supplied, an unsupervised measure of stability is used to make local cuts and extract the optimal clusters. See details.
alpha	numeric; weight between [0, 1] for mixed-objective semi-supervised extraction. Defaults to 0.
minPts	numeric; Defaults to 2. Only needed if class-less noise is a valid label in the model.

`prune_unstable` logical; should significantly unstable subtrees be pruned? The default is `FALSE` for the original optimal extraction framework (see Campello et al, 2013). See details for what `TRUE` implies.

`validate_constraints` logical; should constraints be checked for validity? See details for what are considered valid constraints.

Details

Campello et al (2013) suggested a 'Framework for Optimal Selection of Clusters' (FOSC) as a framework to make local (non-horizontal) cuts to any cluster tree hierarchy. This function implements the original extraction algorithms as described by the framework for hclust objects. Traditional cluster extraction methods from hierarchical representations (such as 'hclust' objects) generally rely on global parameters or cutting values which are used to partition a cluster hierarchy into a set of disjoint, flat clusters. Although such methods are widespread, using global parameter settings are inherently limited in that they cannot capture patterns within the cluster hierarchy at varying *local* levels of granularity.

Rather than partitioning a hierarchy based on the number of the cluster one expects to find (k) or based on some linkage distance threshold (H), the FOSC proposes that the optimal clusters may exist at varying distance thresholds in the hierarchy. To enable this idea, FOSC requires one parameter (minPts) that represents *the minimum number of points that constitute a valid cluster.* The first step of the FOSC algorithm is to traverse the given cluster hierarchy divisely, recording new clusters at each split if both branches represent more than or equal to minPts. Branches that contain less than minPts points at one or both branches inherit the parent clusters identity. Note that using FOSC, due to the constraint that minPts must be greater than or equal to 2, it is possible that the optimal cluster solution chosen makes local cuts that render parent branches of sizes less than minPts as noise, which are denoted as 0 in the final solution.

Traversing the original cluster tree using minPts creates a new, simplified cluster tree that is then post-processed recursively to extract clusters that maximize for each cluster C_i the cost function

$$\max_{\delta_2,\ldots,\delta_k} J = \sum_{i=2}^{k} \delta_i S(C_i)$$

where $S(C_i)$ is the stability-based measure as

$$S(C_i) = \sum_{x_j \in C_i} \left(\frac{1}{h_{min}(x_j, C_i)} - \frac{1}{h_{max}(C_i)}\right)$$

δ_i represents an indicator function, which constrains the solution space such that clusters must be disjoint (cannot assign more than 1 label to each cluster). The measure $S(C_i)$ used by FOSC is an unsupervised validation measure based on the assumption that, if you vary the linkage/distance threshold across all possible values, more prominent clusters that survive over many threshold variations should be considered as stronger candidates of the optimal solution. For this reason, using this measure to detect clusters is referred to as an unsupervised, *stability-based* extraction approach. In some cases it may be useful to enact *instance-level* constraints that ensure the solution space conforms to linkage expectations known *a priori*. This general idea of using preliminary expectations to augment the clustering solution will be referred to as *semisupervised clustering*. If constraints are given in the call to `extractFOSC`, the following alternative objective function is maximized:

$$J = \frac{1}{2n_c} \sum_{j=1}^{n} \gamma(x_j)$$

n_c is the total number of constraints given and $\gamma(x_j)$ represents the number of constraints involving object x_j that are satisfied. In the case of ties (such as solutions where no constraints were given), the unsupervised solution is used as a tiebreaker. See Campello et al (2013) for more details.

As a third option, if one wishes to prioritize the degree at which the unsupervised and semisupervised solutions contribute to the overall optimal solution, the parameter α can be set to enable the extraction of clusters that maximize the *mixed* objective function

$$J = \alpha S(C_i) + (1 - \alpha)\gamma(C_i))$$

FOSC expects the pairwise constraints to be passed as either 1) an $n(n-1)/2$ vector of integers representing the constraints, where 1 represents should-link, -1 represents should-not-link, and 0 represents no preference using the unsupervised solution (see below for examples). Alternatively, if only a few constraints are needed, a named list representing the (symmetric) adjacency list can be used, where the names correspond to indices of the points in the original data, and the values correspond to integer vectors of constraints (positive indices for should-link, negative indices for should-not-link). Again, see the examples section for a demonstration of this.

The parameters to the input function correspond to the concepts discussed above. The `minPts` parameter to represent the minimum cluster size to extract. The optional `constraints` parameter contains the pairwise, instance-level constraints of the data. The optional `alpha` parameters controls whether the mixed objective function is used (if `alpha` is greater than 0). If the `validate_constraints` parameter is set to true, the constraints are checked (and fixed) for symmetry (if point A has a should-link constraint with point B, point B should also have the same constraint). Asymmetric constraints are not supported.

Unstable branch pruning was not discussed by Campello et al (2013), however in some data sets it may be the case that specific subbranches scores are significantly greater than sibling and parent branches, and thus sibling branches should be considered as noise if their scores are cumulatively lower than the parents. This can happen in extremely nonhomogeneous data sets, where there exists locally very stable branches surrounded by unstable branches that contain more than *minPts* points. `prune_unstable = TRUE` will remove the unstable branches.

Value

`cluster`	A integer vector with cluster assignments. Zero indicates noise points (if any).
`hc`	The original hclust object augmented with the n-1 cluster-wide objective scores from the extraction encoded in the 'stability', 'constraint', and 'total' named members.

Author(s)

Matt Piekenbrock

References

Campello, Ricardo JGB, Davoud Moulavi, Arthur Zimek, and Joerg Sander (2013). "A framework for semi-supervised and unsupervised optimal extraction of clusters from hierarchies." *Data Mining and Knowledge Discovery* 27(3): 344-371.

See Also

hdbscan, cutree

Examples

```
data("moons")

## Regular HDBSCAN using stability-based extraction (unsupervised)
cl <- hdbscan(moons, minPts = 5)
cl$cluster

## Constraint-based extraction from the HDBSCAN hierarchy
## (w/ stability-based tiebreaker (semisupervised))
cl_con <- extractFOSC(cl$hc, minPts = 5,
  constraints = list("12" = c(49, -47)))
cl_con$cluster

## Alternative formulation: Constraint-based extraction from the HDBSCAN hierarchy
## (w/ stability-based tiebreaker (semisupervised)) using distance thresholds
dist_moons <- dist(moons)
cl_con2 <- extractFOSC(cl$hc, minPts = 5,
  constraints = ifelse(dist_moons < 0.1, 1L,
                ifelse(dist_moons > 1, -1L, 0L)))

cl_con2$cluster # same as the second example
```

---

frNN *Find the Fixed Radius Nearest Neighbors*

Description

This function uses a kd-tree to find the fixed radius nearest neighbors (including distances) fast.

Usage

```
frNN(x, eps, sort = TRUE, search = "kdtree", bucketSize = 10,
  splitRule = "suggest", approx = 0)
```

Arguments

x	a data matrix, a dist object or a frNN object.
eps	neighbors radius.
search	nearest neighbor search strategy (one of "kdtree" or "linear", "dist").
sort	sort the neighbors by distance? This is expensive and can be done later using sort().
bucketSize	max size of the kd-tree leafs.
splitRule	rule to split the kd-tree. One of "STD", "MIDPT", "FAIR", "SL_MIDPT", "SL_FAIR" or "SUGGEST" (SL stands for sliding). "SUGGEST" uses ANNs best guess.
approx	use approximate nearest neighbors. All NN up to a distance of a factor of 1+approx eps may be used. Some actual NN may be omitted leading to spurious clusters and noise points. However, the algorithm will enjoy a significant speedup.

Details

For details on the parameters see kNN.

Note: self-matches are not returned!

To create a frNN object from scratch, you need to supply at least the elements id with a list of integer vectors with the nearest neighbor ids for each point and eps (see below).

Value

An object of class frNN (subclass of NN) containing a list with the following components:

id	a list of integer vectors. Each vector contains the ids of the fixed radius nearest neighbors.
dist	a list with distances (same structure as ids).
eps	eps used.

Author(s)

Michael Hahsler

References

David M. Mount and Sunil Arya (2010). ANN: A Library for Approximate Nearest Neighbor Searching, http://www.cs.umd.edu/~mount/ANN/.

See Also

NN and kNN for k nearest neighbor search.

Examples

```
data(iris)
x <- iris[, -5]

# Find fixed radius nearest neighbors for each point
nn <- frNN(x, eps=.5)

# Number of neighbors
hist(sapply(adjacencylist(nn), length),
  xlab = "k", main="Number of Neighbors",
  sub = paste("Neighborhood size eps =", nn$eps))

# Explore neighbors of point i = 10
i <- 10
nn$id[[i]]
nn$dist[[i]]
plot(x, col = ifelse(1:nrow(iris) %in% nn$id[[i]], "red", "black"))

# get an adjacency list
head(adjacencylist(nn))

# plot the fixed radius neighbors (and then reduced to a radius of .3)
plot(nn, x)
plot(frNN(nn, .3), x)

## manually create a frNN object for dbscan (dbscan only needs ids and eps)
nn <- list(ids = list(c(2,3), c(1,3), c(1,2,3), c(3,5), c(4,5)), eps = 1)
class(nn) <- c("NN", "frNN")
nn
dbscan(nn, minPts = 2)
```

---

glosh *Global-Local Outlier Score from Hierarchies*

Description

Calculate the Global-Local Outlier Score from Hierarchies (GLOSH) score for each data point using a kd-tree to speed up kNN search.

Usage

```
glosh(x, k = 4, ...)
```

Arguments

x	an hclust object, data matrix, or dist object.
k	size of the neighborhood.
...	further arguments are passed on to kNN.

Details

GLOSH compares the density of a point to densities of any points associated within current and child clusters (if any). Points that have a substantially lower density than the density mode (cluster) they most associate with are considered outliers. GLOSH is computed from a hierarchy a clusters.

Specifically, consider a point *x* and a density or distance threshold *lambda*. GLOSH is calculated by taking 1 minus the ratio of how long any of the child clusters of the cluster *x* belongs to "survives" changes in *lambda* to the highest *lambda* threshold of x, above which x becomes a noise point.

Scores close to 1 indicate outliers. For more details on the motivation for this calculation, see Campello et al (2015).

Value

A numeric vector of length equal to the size of the original data set containing GLOSH values for all data points.

Author(s)

Matt Piekenbrock

References

Campello, Ricardo JGB, Davoud Moulavi, Arthur Zimek, and Joerg Sander. "Hierarchical density estimates for data clustering, visualization, and outlier detection." ACM Transactions on Knowledge Discovery from Data (TKDD) 10, no. 1 (2015): 5.

See Also

kNN, pointdensity, lof.

Examples

```
set.seed(665544)
n <- 100
x <- cbind(
  x=runif(10, 0, 5) + rnorm(n, sd=0.4),
  y=runif(10, 0, 5) + rnorm(n, sd=0.4)
  )

### calculate LOF score
glosh <- glosh(x, k=3)

### distribution of outlier scores
summary(glosh)
hist(glosh, breaks=10)

### simple function to plot point size is proportional to GLOSH score
plot_glosh <- function(x, glosh){
  plot(x, pch = ".", main = "GLOSH (k=3)")
  points(x, cex = glosh*3, pch = 1, col="red")
  text(x[glosh > 0.80,], labels = round(glosh, 3)[glosh > 0.80], pos = 3)
```

```
}
plot_glosh(x, glosh)

### GLOSH with any hierarchy
x_dist <- dist(x)
x_sl <- hclust(x_dist, method = "single")
x_upgma <- hclust(x_dist, method = "average")
x_ward <- hclust(x_dist, method = "ward.D2")

## Compare what different linkage criterion consider as outliers
glosh_sl <- glosh(x_sl, k = 3)
plot_glosh(x, glosh_sl)

glosh_upgma <- glosh(x_upgma, k = 3)
plot_glosh(x, glosh_upgma)

glosh_ward <- glosh(x_ward, k = 3)
plot_glosh(x, glosh_ward)

## GLOSH is automatically computed with HDBSCAN
all(hdbscan(x, minPts = 3)$outlier_scores == glosh(x, k = 3))
```

---

hdbscan *HDBSCAN*

Description

Fast implementation of the HDBSCAN (Hierarchical DBSCAN) and its related algorithms using Rcpp.

Usage

```
hdbscan(x, minPts, xdist = NULL,
    gen_hdbscan_tree = FALSE,
    gen_simplified_tree = FALSE)

## S3 method for class 'hdbscan'
print(x, ...)
## S3 method for class 'hdbscan'
plot(x, scale="suggest",
    gradient=c("yellow", "red"), show_flat = FALSE, ...)
```

Arguments

x	a data matrix or a dist object.
minPts	integer; Minimum size of clusters. See details.
xdist	an optional precomputed dist object. May reduce computation time if supplied.

`gen_hdbscan_tree`
: logical; should the robust single linkage tree be explicitly computed. (see cluster tree in Chaudhuri et al, 2010).

`gen_simplified_tree`
: logical; should the simplified hierarchy be explicitly computed. (see Campello et al, 2013).

`...`
: additional arguments are passed on to the appropriate S3 methods (such as plotting parameters).

`scale`
: integer; used to scale condensed tree based on the graphics device. Lower scale results in wider trees.

`gradient`
: character vector; the colors to build the condensed tree coloring with.

`show_flat`
: logical; whether to draw boxes indicating the most stable clusters.

Details

Computes the hierarchical cluster tree representing density estimates along with the stability-based flat cluster extraction proposed by Campello et al. (2013). HDBSCAN essentially computes the hierarchy of all DBSCAN* clusterings, and then uses a stability-based extraction method to find optimal cuts in the hierarchy, thus producing a flat solution.

Additional, related algorithms including the "Global-Local Outlier Score from Hierarchies" (GLOSH) (see section 6 of Campello et al. 2015) outlier scores and ability to cluster based on instance-level constraints (see section 5.3 of Campello et al. 2015) are supported. The algorithms only need the parameter `minPts`.

Note that `minPts` not only acts as a minimum cluster size to detect, but also as a "smoothing" factor of the density estimates implicitly computed from HDBSCAN.

Value

A object of class 'hdbscan' with the following components:

`cluster`
: A integer vector with cluster assignments. Zero indicates noise points.

`minPts`
: value of the minPts parameter.

`cluster_scores`
: The sum of the stability scores for each salient ('flat') cluster. Corresponds to cluster ids given the in 'cluster' member.

`membership_prob`
: The 'probability' or individual stability of a point within its clusters. Between 0 and 1.

`outlier_scores`
: The outlier score (GLOSH) of each point.

`hc`
: An 'hclust' object of the HDBSCAN hierarchy.

Author(s)

Matt Piekenbrock

References

Martin Ester, Hans-Peter Kriegel, Joerg Sander, Xiaowei Xu (1996). A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. Institute for Computer Science, University of Munich. *Proceedings of 2nd International Conference on Knowledge Discovery and Data Mining (KDD-96).*

Campello, R. J. G. B.; Moulavi, D.; Sander, J. (2013). Density-Based Clustering Based on Hierarchical Density Estimates. *Proceedings of the 17th Pacific-Asia Conference on Knowledge Discovery in Databases, PAKDD 2013,* Lecture Notes in Computer Science 7819, p. 160.

Campello, Ricardo JGB, et al. "Hierarchical density estimates for data clustering, visualization, and outlier detection." ACM Transactions on Knowledge Discovery from Data (TKDD) 10.1 (2015): 5.

See Also

dbscan

Examples

```
## cluster the moons data set with HDBSCAN
data(moons)

res <- hdbscan(moons, minPts = 5)
res

plot(res)

plot(moons, col = res$cluster + 1L)

## DS3 from Chameleon
data("DS3")

res <- hdbscan(DS3, minPts = 50)
res

## Plot the simplified tree, highlight the most stable clusters
plot(res, show_flat = TRUE)

## Plot the actual clusters
plot(DS3, col=res$cluster+1L, cex = .5)
```

---

hullplot *Plot Convex Hulls of Clusters*

Description

This function produces a two-dimensional scatter plot with added convex hulls for clusters.

Usage

```
hullplot(x, cl, col = NULL, cex = 0.5, hull_lwd = 1, hull_lty = 1,
  solid = TRUE, alpha = .2, main = "Convex Cluster Hulls", ...)
```

Arguments

x	a data matrix. If more than 2 columns are provided, then the data is plotted using the first two principal components.
cl	a clustering. Either a numeric cluster assignment vector or a clustering object (a list with an element named `cluster`).
col	colors used for clusters. Defaults to the standard palette. The first color (default is black) is used for noise/unassigned points (cluster id 0).
cex	expansion factor for symbols.
hull_lwd, hull_lty	line width and line type used for the convex hull.
main	main title.
solid, alpha	draw filled polygons instead of just lines for the convex hulls? alpha controls the level of alpha shading.
...	additional arguments passed on to plot.

Author(s)

Michael Hahsler

Examples

```
set.seed(2)
n <- 400

x <- cbind(
  x = runif(4, 0, 1) + rnorm(n, sd=0.1),
  y = runif(4, 0, 1) + rnorm(n, sd=0.1)
  )
cl <- rep(1:4, time = 100)

### original data with true clustering
hullplot(x, cl, main = "True clusters")
### use differnt symbols
hullplot(x, cl, main = "True clusters", pch = cl)
### just the hulls
hullplot(x, cl, main = "True clusters", pch = NA)
### a version suitable for b/w printing)
hullplot(x, cl, main = "True clusters", solid = FALSE, col = "black", pch = cl)


### run some clustering algorithms and plot the resutls
db <- dbscan(x, eps = .07, minPts = 10)
hullplot(x, db, main = "DBSCAN")
```

```
op <- optics(x, eps = 10, minPts = 10)
opDBSCAN <- extractDBSCAN(op, eps_cl = .07)
hullplot(x, opDBSCAN, main = "OPTICS")

opXi <- extractXi(op, xi = 0.05)
hullplot(x, opXi, main = "OPTICSXi")

# Extract minimal 'flat' clusters only
opXi <- extractXi(op, xi = 0.05, minimum = TRUE)
hullplot(x, opXi, main = "OPTICSXi")

km <- kmeans(x, centers = 4)
hullplot(x, km, main = "k-means")

hc <- cutree(hclust(dist(x)), k = 4)
hullplot(x, hc, main = "Hierarchical Clustering")
```

`jpclust` *Jarvis-Patrick Clustering*

Description

Fast C++ implementation of the Jarvis-Patrick clustering which first builds a shared nearest neighbor graph (k nearest neighbor sparsification) and then places two points in the same cluster if they are in each other's nearest neighbor list and they share at least kt nearest neighbors.

Usage

```
jpclust(x, k, kt, ...)
```

Arguments

`x`	a data matrix/data.frame (Euclidean distance is used), a precomputed dist object or a kNN object created with `kNN()`.
`k`	Neighborhood size for nearest neighbor sparsification. If `x` is a kNN object then `k` may be missing.
`kt`	threshold on the number of shared nearest neighbors (including the points themselves) to form clusters.
`...`	additional arguments are passed on to the k nearest neighbor search algorithm. See `kNN` for details on how to control the search strategy.

Details

Note: Following the original paper, the shared nearest neighbor list is constructed as the k neighbors plus the point itself (as neighbor zero). Therefore, the threshold `kt` can be in the range [1, k].

Fast nearest neighbors search with `kNN()` is only used if `x` is a matrix. In this case Euclidean distance is used.

Value

A object of class 'general_clustering' with the following components:

`cluster`	A integer vector with cluster assignments. Zero indicates noise points.
`type`	name of used clustering algorithm.
`param`	list of used clustering parameters.

Author(s)

Michael Hahsler

References

R. A. Jarvis and E. A. Patrick. 1973. Clustering Using a Similarity Measure Based on Shared Near Neighbors. *IEEE Trans. Comput. 22,* 11 (November 1973), 1025-1034.

See Also

kNN

Examples

```
data("DS3")

# use a shared neighborhood of 20 points and require 12 shared neighbors
cl <- jpclust(DS3, k = 20, kt = 12)
cl

plot(DS3, col = cl$cluster+1L, cex = .5)
# Note: JP clustering does not consider noise and thus,
# the sine wave points chain clusters together.

# use a precomputed kNN object instead of the original data.
nn <- kNN(DS3, k = 30)
nn

cl <- jpclust(nn, k = 20, kt = 12)
cl

# cluster with noise removed (use low pointdensity to identify noise)
d <- pointdensity(DS3, eps = 25)
hist(d, breaks = 20)
DS3_noiseless <- DS3[d > 110,]

cl <- jpclust(DS3_noiseless, k = 20, kt = 10)
cl

plot(DS3_noiseless, col = cl$cluster+1L, cex = .5)
```

kNN *Find the k Nearest Neighbors*

Description

This function uses a kd-tree to find all k nearest neighbors in a data matrix (including distances) fast.

Usage

```
kNN(x, k, sort = TRUE, search = "kdtree", bucketSize = 10,
  splitRule = "suggest", approx = 0)
```

Arguments

x	a data matrix, a dist object or a kNN object.
k	number of neighbors to find.
search	nearest neighbor search strategy (one of "kdtree", "linear" or "dist").
sort	sort the neighbors by distance? Note that this is expensive and `sort = FALSE` is much faster. kNN objects can be sorted using `sort()`.
bucketSize	max size of the kd-tree leafs.
splitRule	rule to split the kd-tree. One of "STD", "MIDPT", "FAIR", "SL_MIDPT", "SL_FAIR" or "SUGGEST" (SL stands for sliding). "SUGGEST" uses ANNs best guess.
approx	use approximate nearest neighbors. All NN up to a distance of a factor of 1+approx eps may be used. Some actual NN may be omitted leading to spurious clusters and noise points. However, the algorithm will enjoy a significant speedup.

Details

`search` controls if a kd-tree or linear search (both implemented in the ANN library; see Mount and Arya, 2010). Note, that these implementations cannot handle NAs. `search="dist"` precomputes Euclidean distances first using R. NAs are handled, but the resulting distance matrix cannot contain NAs. To use other distance measures, a precomputed distance matrix can be provided as `x` (`search` is ignored).

`bucketSize` and `splitRule` influence how the kd-tree is built. `approx` uses the approximate nearest neighbor search implemented in ANN. All nearest neighbors up to a distance of `eps`/(1+`approx`) will be considered and all with a distance greater than `eps` will not be considered. The other points might be considered. Note that this results in some actual nearest neighbors being omitted leading to spurious clusters and noise points. However, the algorithm will enjoy a significant speedup. For more details see Mount and Arya (2010).

Note: self-matches are removed!

Value

An object of class kNN containing a list with the following components:

dist	a matrix with distances.
id	a matrix with ids.
k	number of k used.

Author(s)

Michael Hahsler

References

David M. Mount and Sunil Arya (2010). ANN: A Library for Approximate Nearest Neighbor Searching, http://www.cs.umd.edu/~mount/ANN/.

See Also

NN and frNN for fixed radius nearest neighbors.

Examples

```
data(iris)
x <- iris[, -5]

# finding kNN directly in data (using a kd-tree)
nn <- kNN(x, k=5)
nn

# explore neighborhood of point 10
i <- 10
nn$id[i,]
plot(x, col = ifelse(1:nrow(iris) %in% nn$id[i,], "red", "black"))

# Visualize the 5 nearest neighbors
plot(nn, x)

# Visualize a reduced 2-NN graph
plot(kNN(nn, k = 2), x)
```

---

kNNdist *Calculate and plot the k-Nearest Neighbor Distance*

Description

Fast calculation of the k-nearest neighbor distances in a matrix of points. The plot can be used to help find a suitable value for the eps neighborhood for DBSCAN. Look for the knee in the plot.

Usage

```
kNNdist(x, k, ...)
kNNdistplot(x, k = 4, ...)
```

Arguments

x	the data set as a matrix or a dist object.
k	number of nearest neighbors used (use minPoints).
...	further arguments are passed on to kNN.

Details

See kNN for a discussion of the kd-tree related parameters.

Value

kNNdist returns a numeric vector with the distance to its k nearest neighbor.

Author(s)

Michael Hahsler

See Also

kNN.

Examples

```
data(iris)
iris <- as.matrix(iris[,1:4])

kNNdist(iris, k=4, search="kd")
kNNdistplot(iris, k=4)
## the knee is around a distance of .5

cl <- dbscan(iris, eps = .5, minPts = 4)
pairs(iris, col = cl$cluster+1L)
## Note: black are noise points
```

---

lof *Local Outlier Factor Score*

Description

Calculate the Local Outlier Factor (LOF) score for each data point using a kd-tree to speed up kNN search.

Usage

```
lof(x, k = 4, ...)
```

Arguments

x	a data matrix or a dist object.
k	size of the neighborhood.
...	further arguments are passed on to kNN.

Details

LOF compares the local density of an point to the local densities of its neighbors. Points that have a substantially lower density than their neighbors are considered outliers. A LOF score of approximately 1 indicates that density around the point is comparable to its neighbors. Scores significantly larger than 1 indicate outliers.

Note: If there are more than k duplicate points in the data, then lof can become NaN caused by an infinite local density. In this case we set lof to 1. The paper by Breunig et al (2000) suggests a different method of removing all duplicate points first.

Value

A numeric vector of length ncol(x) containing LOF values for all data points.

Author(s)

Michael Hahsler

References

Breunig, M., Kriegel, H., Ng, R., and Sander, J. (2000). LOF: identifying density-based local outliers. In *ACM Int. Conf. on Management of Data,* pages 93-104.

See Also

kNN, pointdensity, glosh.

Examples

```
set.seed(665544)
n <- 100
x <- cbind(
  x=runif(10, 0, 5) + rnorm(n, sd=0.4),
  y=runif(10, 0, 5) + rnorm(n, sd=0.4)
  )

### calculate LOF score
lof <- lof(x, k=3)

### distribution of outlier factors
```

```
summary(lof)
hist(lof, breaks=10)

### point size is proportional to LOF
plot(x, pch = ".", main = "LOF (k=3)")
points(x, cex = (lof-1)*3, pch = 1, col="red")
text(x[lof>2,], labels = round(lof, 1)[lof>2], pos = 3)
```

`moons` *Moons Data*

Description

Contains 100 2-d points, half of which are contained in two moons or "blobs"" (25 points each blob), and the other half in asymmetric facing crescent shapes. The three shapes are all linearly separable.

Usage

```
data("moons")
```

Format

A data frame with 100 observations on the following 2 variables.

`X` a numeric vector

`Y` a numeric vector

Details

This data was generated with the following Python commands using the SciKit-Learn library. dontrun import sklearn.datasets as data moons, _ = data.make_moons(n_samples=50, noise=0.05) blobs, _ = data.make_blobs(n_samples=50, centers=[(-0.75,2.25), (1.0, 2.0)], cluster_std=0.25) test_data = np.vstack([moons, blobs])

Source

See the HDBSCAN notebook from github documentation: http://hdbscan.readthedocs.io/en/latest/how_hdbscan_works.html

References

1. Pedregosa, Fabian, Gael Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel et al. "Scikit-learn: Machine learning in Python." Journal of Machine Learning Research 12, no. Oct (2011): 2825-2830.

Examples

```
data(moons)
plot(moons, pch=20)
```

NN *Nearest Neighbors Auxiliary Functions*

Description

Helper functions for nearest neighbors (kNN and frNN).

Usage

```
adjacencylist(x, ...)
## S3 method for class 'NN'
plot(x, data, main = NULL, ...)
```

Arguments

x	a nearest neighbor object (of class kNN or frNN).
...	further arguments are currently ignored.
data	data with the coordinates for plotting.
main	main title for the plot.

Value

`adjacencylist` returns a list with one element for each original data point. Each element contains the row ids of the nearest neighbors. The adjacency list can be used to create for example a graph object.

Author(s)

Michael Hahsler

See Also

`frNN` and `kNN`.

Examples

```
data(iris)
x <- iris[, -5]

# finding kNN directly in data (using a kd-tree)
nn <- kNN(x, k=5)
nn

# plot the kNN where NN are shown as line conecting points.
plot(nn, x)

# show the first few elements of the adjacency list
head(adjacencylist(nn))
```

```
# create a graph and find connected components (if igraph is installed)
if("igraph" %in% installed.packages()){
  library("igraph")
  g <- graph_from_adj_list(adjacencylist(nn))
  comp <- components(g)
  plot(x, col = comp$membership)

  # detect clusters (communities) with the label propagation algorithm
  cl <- membership(cluster_label_prop(g))
  plot(x, col = cl)
}
```

---

optics *OPTICS*

Description

Implementation of the OPTICS (Ordering points to identify the clustering structure) clustering algorithm using a kd-tree.

Usage

```
optics(x, eps = NULL, minPts = 5, ...)

extractDBSCAN(object, eps_cl)
extractXi(object, xi, minimum = FALSE, correctPredecessors = TRUE)

## S3 method for class 'optics'
predict(object, newdata = NULL, data, ...)
```

Arguments

x	a data matrix or a dist object.
eps	upper limit of the size of the epsilon neighborhood. Limiting the neighborhood size improves performance and has no or very little impact on the ordering as long as it is not set too low. If not specified, the largest minPts-distance in the data set is used which gives the same result as infinity.
minPts	the parameter is used to identify dense neighborhoods and the reachability distance is calculated as the distance to the minPts nearest neighbor. Controls the smoothness of the reachability distribution. Default is 5 points.
eps_cl	Threshold to identify clusters (eps_cl <= eps).
xi	Steepness threshold to identify clusters hierarchically using the Xi method.
object	an object of class optics. For predict it needs to contain not just an ordering, but also an extracted clustering.
data	the data set used to create the OPTICS clustering object.

newdata	new data set for which cluster membership should be predicted.
minimum	boolean representing whether or not to extract the minimal (non-overlapping) clusters in the Xi clustering algorithm.
correctPredecessors	boolean Correct a common artifacting by pruning the steep up area for points that have predecessors not in the cluster–found by the ELKI framework, see details below.
...	additional arguments are passed on to fixed-radius nearest neighbor search algorithm. See frNN for details on how to control the search strategy.

Details

This implementation of OPTICS implements the original algorithm as described by Ankerst et al (1999). OPTICS is an ordering algorithm using similar concepts to DBSCAN. However, for OPTICS eps is only an upper limit for the neighborhood size used to reduce computational complexity. Note that minPts in OPTICS has a different effect then in DBSCAN. It is used to define dense neighborhoods, but since eps is typically set rather high, this does not effect the ordering much. However, it is also used to calculate the reachability distance and larger values will make the reachability distance plot smoother.

OPTICS linearly orders the data points such that points which are spatially closest become neighbors in the ordering. The closest analog to this ordering is dendrogram in single-link hierarchical clustering. The algorithm also calculates the reachability distance for each point. plot() produces a reachability-plot which shows each points reachability distance where the points are sorted by OPTICS. Valleys represent clusters (the deeper the valley, the more dense the cluster) and high points indicate points between clusters.

extractDBSCAN extracts a clustering from an OPTICS ordering that is similar to what DBSCAN would produce with an eps set to eps_cl (see Ankerst et al, 1999). The only difference to a DBSCAN clustering is that OPTICS is not able to assign some border points and reports them instead as noise.

extractXi extract clusters hiearchically specified in Ankerst et al (1999) based on the steepness of the reachability plot. One interpretation of the xi parameter is that it classifies clusters by change in relative cluster density. The used algorithm was originally contributed by the ELKI framework, but contains a set of fixes.

See frNN for more information on the parameters related to nearest neighbor search.

Value

An object of class 'optics' with components:

eps	value of eps parameter.
minPts	value of minPts parameter.
order	optics order for the data points in x.
reachdist	reachability distance for each data point in x.
coredist	core distance for each data point in x.

If extractDBSCAN was called, then in addition the following components are available:

eps_cl	reachability distance for each point in x.
cluster	assigned cluster labels in the order of the data points in x.

If extractXi was called, then in addition the following components are available:

xi	Steepness thresholdx.
cluster	assigned cluster labels in the order of the data points in x.
clusters_xi	data.frame containing the start and end of each cluster found in the OPTICS ordering.

Author(s)

Michael Hahsler and Matthew Piekenbrock

References

Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, Joerg Sander (1999). OPTICS: Ordering Points To Identify the Clustering Structure. ACM SIGMOD international conference on Management of data. ACM Press. pp. 49–60.

See Also

frNN, as.reachability.

Examples

```
set.seed(2)
n <- 400

x <- cbind(
  x = runif(4, 0, 1) + rnorm(n, sd=0.1),
  y = runif(4, 0, 1) + rnorm(n, sd=0.1)
  )

plot(x, col=rep(1:4, time = 100))

### run OPTICS (Note: we use the default eps calculation)
res <- optics(x, minPts = 10)
res

### get order
res$order

### plot produces a reachability plot
plot(res)

### plot the order of points in the reachability plot
plot(x, col = "grey")
polygon(x[res$order,])

### extract a DBSCAN clustering by cutting the reachability plot at eps_cl
```

```
res <- extractDBSCAN(res, eps_cl = .065)
res

plot(res)  ## black is noise
hullplot(x, res)

### re-cut at a higher eps threshold
res <- extractDBSCAN(res, eps_cl = .1)
res
plot(res)
hullplot(x, res)

### extract hierarchical clustering of varying density using the Xi method
res <- extractXi(res, xi = 0.05)
res

plot(res)
hullplot(x, res)

# Xi cluster structure
res$clusters_xi

### use OPTICS on a precomputed distance matrix
d <- dist(x)
res <- optics(d, minPts = 10)
plot(res)
```

`pointdensity` *Calculate Local Density at Each Data Point*

Description

Calculate the local density at each data point as either the number of points in the eps-neighborhood (as used in DBSCAN) or the kernel density estimate (kde) of a uniform kernel. The function uses a kd-tree for fast fixed-radius nearest neighbor search.

Usage

```
pointdensity(x, eps, type = "frequency",
  search = "kdtree", bucketSize = 10,
  splitRule = "suggest", approx = 0)
```

Arguments

`x`	a data matrix.
`eps`	radius of the eps-neighborhood, i.e., bandwidth of the uniform kernel).
`type`	`"frequency"` or `"density"`. should the raw count of points inside the eps-neighborhood or the kde be returned.
`search, bucketSize, splitRule, approx`	algorithmic parameters for `frNN`.

Details

DBSCAN estimates the density around a point as the number of points in the eps-neighborhood of the point (including the query point itself). The kde using a uniform kernel is just this count divided by $2epsn$, where n is the number of points in `x`.

Points with low local density often indicate noise (see e.g., Wishart (1969) and Hartigan (1975)).

Value

A vector of the same length as data points (rows) in `x` with the count or density values for each data point.

Author(s)

Michael Hahsler

References

WISHART, D. (1969), Mode Analysis: A Generalization of Nearest Neighbor which Reduces Chaining Effects, in Numerical Taxonomy, Ed., A.J. Cole, Academic Press, 282-311.

John A. Hartigan (1975), Clustering Algorithms, John Wiley \& Sons, Inc., New York, NY, USA.

See Also

frNN.

Examples

```
set.seed(665544)
n <- 100
x <- cbind(
  x=runif(10, 0, 5) + rnorm(n, sd=0.4),
  y=runif(10, 0, 5) + rnorm(n, sd=0.4)
  )
plot(x)

### calculate density
d <- pointdensity(x, eps = .5, type = "density")

### density distribution
summary(d)
hist(d, breaks = 10)

### point size is proportional to Density
plot(x, pch = 19, main = "Density (eps = .5)", cex = d*5)

### Wishart (1969) single link clustering method
# 1. remove noise with low density
f <- pointdensity(x, eps = .5, type = "frequency")
x_nonoise <- x[f >= 5,]
```

```
# 2. use single-linkage on the non-noise points
hc <- hclust(dist(x_nonoise), method = "single")
plot(x, pch = 19, cex = .5)
points(x_nonoise, pch = 19, col= cutree(hc, k = 4)+1L)
```

`reachability` *Density Reachability Structures*

Description

Class "reachability" provides general functions for representing various hierarchical representations as "reachability plots", as originally defined by Ankerst et al (1999). Methods include fast implementations of the conversion algorithms introduced by Sanders et al (2003) to convert between dendrograms and reachability plot objects.

Usage

```
as.reachability(object, ...)

## S3 method for class 'optics'
as.reachability(object, ...)

## S3 method for class 'dendrogram'
as.reachability(object, ...)

## S3 method for class 'reachability'
as.dendrogram(object, ...)

## S3 method for class 'reachability'
print(x, ...)

## S3 method for class 'reachability'
plot(x, order_labels = FALSE,
  xlab = "Order", ylab = "Reachability dist.",
  main = "Reachability Plot", ...)
```

Arguments

`object`	any R object that can be made into one of class "reachability", such as an object of class "optics" or "dendrogram".
`x`	object of class "reachability".
`order_labels`	whether to plot text labels for each points reachability distance.
`xlab`	x-axis label, defaults to "Order".
`ylab`	y-axis label, defaults to "Reachability dist.".
`main`	Title of the plot, defaults to "Reachability Plot".
`...`	graphical parameters, or arguments for other methods.

Details

Dendrograms are a popular visualization tool for representing hierarchical relationships. In agglomerative clustering, dendrograms can be constructed using a variety of linkage criterion (such as single or complete linkage), many of which are frequently used to 1) visualize the density-based relationships in the data or 2) extract cluster labels from the data the dendrogram represents.

The original ordering algorithm OPTICS as described by Ankerst et al (1999) introduced the notion of 2-dimensional representation of so-called "density-reachability" that was shown to be useful for data visualization. This representation was shown to essentially convey the same information as the more traditional dendrogram structure by Sanders et al (2003).

Different hierarchical representations, such as dendrograms or reachability plots, may be preferable depending on the context. In smaller datasets, cluster memberships may be more easily identifiable through a dendrogram representation, particularly is the user is already familiar with tree-like representations. For larger datasets however, a reachability plot may be preferred for visualizing macro-level density relationships.

The central idea behind a reachability plot is that the ordering in which points are plotted identifies underlying hierarchical density representation. OPTICS linearly orders the data points such that points which are spatially closest become neighbors in the ordering. Valleys represent clusters, which can be represented hierarchically. Although the ordering is crucial to the structure of the reachability plot, its important to note that OPTICS, like DBSCAN, is not entirely deterministic and, just like the dendrogram, isomorphisms may exist

A variety of cluster extraction methods have been proposed using reachability plots. Because both cluster extraction depend directly on the ordering OPTICS produces, they are part of the optics interface. Nonetheless, reachability plots can be created directly from other types of linkage trees, and vice versa.

See `optics` for more information on how OPTICS is formulated. `extractDBSCAN` and `extractXi` are the two cluster extraction methods presented in the original OPTICS publication.

Value

An object of class 'reachability' with components:

`order`	order to use for the data points in `x`.
`reachdist`	reachability distance for each data point in `x`.

Author(s)

Matthew Piekenbrock

References

Ankerst, M., M. M. Breunig, H.-P. Kriegel, J. Sander (1999). OPTICS: Ordering Points To Identify the Clustering Structure. *ACM SIGMOD international conference on Management of data.* ACM Press. pp. 49–60.

Sander, J., X. Qin, Z. Lu, N. Niu, and A. Kovarsky (2003). Automatic extraction of clusters from hierarchical clustering representations. *Pacific-Asia Conference on Knowledge Discovery and Data Mining.* Springer Berlin Heidelberg.

See Also

dendrogram, optics, extractDBSCAN, extractXi, hclust.

Examples

```
set.seed(2)
n <- 20

x <- cbind(
  x = runif(4, 0, 1) + rnorm(n, sd=0.1),
  y = runif(4, 0, 1) + rnorm(n, sd=0.1)
)

plot(x, xlim=range(x), ylim=c(min(x)-sd(x), max(x)+sd(x)), pch=20)
text(x = x, labels = 1:nrow(x), pos=3)

### run OPTICS
res <- optics(x, eps = 10,  minPts = 2)
res

### plot produces a reachability plot
plot(res)

### Extract reachability components from OPTICS
reach <- as.reachability(res)
reach

### plot still produces a reachability plot; points ids
### (rows in the original data) can be displayed with order_labels = TRUE
plot(reach, order_labels = TRUE)

### Reachability objects can be directly converted to dendrograms
dend <- as.dendrogram(reach)
dend
plot(dend)

### A dendrogram can be converted back into a reachability object
plot(as.reachability(dend))
```

---

sNN *Shared Nearest Neighbors*

Description

Calculates the number of shared nearest neighbors.

Usage

```
sNN(x, k, kt = NULL, sort = TRUE, search = "kdtree", bucketSize = 10,
  splitRule = "suggest", approx = 0)
```

Arguments

x	a data matrix, a dist object or a kNN object.
k	number of neighbors to consider to calculate the shared nearest neighbors.
kt	threshold on the number of shared nearest neighbors graph. Edges are only preserved if kt or more neighbors are shared.
search	nearest neighbor search strategy (one of "kdtree", "linear" or "dist").
sort	sort the neighbors by distance? Note that this is expensive and sort = FALSE is much faster. kNN objects can be sorted using sort().
bucketSize	max size of the kd-tree leafs.
splitRule	rule to split the kd-tree. One of "STD", "MIDPT", "FAIR", "SL_MIDPT", "SL_FAIR" or "SUGGEST" (SL stands for sliding). "SUGGEST" uses ANNs best guess.
approx	use approximate nearest neighbors. All NN up to a distance of a factor of 1+approx eps may be used. Some actual NN may be omitted leading to spurious clusters and noise points. However, the algorithm will enjoy a significant speedup.

Details

The number of shared nearest neighbors is the intersection of the kNN neighborhood of two points. Note: that each point is considered to be part of its own kNN neighborhood. The range for the shared nearest neighbors is [0,k].

Value

An object of class sNN (subclass of kNN and NN) containing a list with the following components:

id	a matrix with ids.
dist	a matrix with the distances.
shared	a matrix with the number of shared nearest neighbors.
k	number of k used.

Author(s)

Michael Hahsler

See Also

NN and kNN for k nearest neighbors.

Examples

```
data(iris)
x <- iris[, -5]

# finding kNN and add the number of shared nearest neighbors.
k <- 5
nn <- sNN(x, k = k)
nn

# shared nearest neighbor distribution
table(as.vector(nn$shared))

# explore neighborhood of point 10
i <- 10
nn$shared[i,]

plot(nn, x)

# apply a threshold to create a sNN graph with edges
# if more than 3 neighbors are shared.
plot(sNN(nn, kt = 3), x)
```

---

sNNclust *Shared Nearest Neighbor Clustering*

Description

Implements the shared nearest neighbor clustering algorithm by Ertoz, Steinbach and Kumar.

Usage

```
sNNclust(x, k, eps, minPts, borderPoints = TRUE, ...)
```

Arguments

x	a data matrix/data.frame (Euclidean distance is used), a precomputed dist object or a kNN object created with kNN().
k	Neighborhood size for nearest neighbor sparsification to create the shared NN graph.
eps	Two objects are only reachable from each other if they share at least eps nearest neighbors.
minPts	minimum number of points that share at least eps nearest neighbors for a point to be considered a core points.
borderPoints	should borderPoints be assigned to clusters like in DBSCAN?
...	additional arguments are passed on to the k nearest neighbor search algorithm. See kNN for details on how to control the search strategy.

Details

Algorithm:

1) Constructs a shared nearest neighbor graph for a given k. The edge weights are the number of shared k nearest neighbors (in the range of [0, k]).

2) Find each points SNN density, i.e., the number of points which have a similarity of eps or greater.

3) Find the core points, i.e., all points that have an SNN density greater than MinPts.

4) Form clusters from the core points and assign border points (i.e., non-core points which share at least eps neighbors with a core point).

Note that steps 2-4 are DBSCAN and that `eps` is used on a similarity (the number of shared neighbors) and not on a distance like in DBSCAN.

Value

A object of class 'general_clustering' with the following components:

`cluster`	A integer vector with cluster assignments. Zero indicates noise points.
`type`	name of used clustering algorithm.
`param`	list of used clustering parameters.

Author(s)

Michael Hahsler

References

Levent Ertoz, Michael Steinbach, Vipin Kumar, Finding Clusters of Different Sizes, Shapes, and Densities in Noisy, High Dimensional Data, *SIAM International Conference on Data Mining,* 2003, 47-59.

See Also

`jpclust`

Examples

```
data("DS3")

# Out of k = 20 NN 7 (eps) have to be shared to create a link in the sNN graph.
# A point needs a least 16 (minPts) links in the sNN graph to be a core point.
# Noise points have cluster id 0 and are shown in black.
cl <- sNNclust(DS3, k = 20, eps = 7, minPts = 16)
plot(DS3, col = cl$cluster + 1L, cex = .5)
```

Index