

Package ‘clue’

November 22, 2009

Version 0.3-33

Date 2009-11-22

Title Cluster ensembles

Description CLUster Ensembles

Author Kurt Hornik, with contributions from Walter Boehm

Maintainer Kurt Hornik <Kurt.Hornik@R-project.org>

License GPL-2

Depends R (>= 2.7.0)

Imports stats, cluster, graphics, methods

Suggests e1071, lpSolve (>= 5.5.7), quadprog (>= 1.4-8), relations

Enhances RWeka, ape, cba, cclust, flexmix, flexclust, kernlab, mclust, modeltools

Repository CRAN

Date/Publication 2009-11-22 19:06:40

R topics documented:

addtree	2
Cassini	4
CKME	5
cl_agreement	6
cl_bag	9
cl_boot	10
cl_classes	11
cl_consensus	12
cl_dissimilarity	16
cl_ensemble	21
cl_fuzziness	22

cl_margin	24
cl_medoid	24
cl_membership	25
cl_object_names	27
cl_pam	28
cl_pclust	29
cl_predict	31
cl_prototypes	32
cl_tabulate	33
cl_ultrametric	33
cl_validity	35
fit_ultrametric_target	36
GVME	37
GVME_Consensus	38
hierarchy	39
Kinship82	41
Kinship82_Consensus	42
kmedoids	43
l1_fit_ultrametric	44
lattice	45
ls_fit_sum_of_ultrametrics	47
ls_fit_ultrametric	48
n_of_classes	51
n_of_objects	52
partition	53
pclust	54
Phonemes	57
solve_LSAP	58
sumt	59
Index	61

addtree

Least Squares Fit of Additive Tree Distances to Dissimilarities

Description

Find the additive tree distance or centroid distance minimizing least squares distance (Euclidean dissimilarity) to a given dissimilarity object.

Usage

```
ls_fit_addtree(x, method = c("SUMT", "IP", "IR"), weights = 1,
              control = list())
ls_fit_centroid(x)
```

Arguments

<code>x</code>	a dissimilarity object inheriting from class " <code>dist</code> ".
<code>method</code>	a character string indicating the fitting method to be employed. Must be one of " <code>SUMT</code> " (default), " <code>IP</code> ", or " <code>IR</code> ", or a unique abbreviation thereof.
<code>weights</code>	a numeric vector or matrix with non-negative weights for obtaining a weighted least squares fit. If a matrix, its numbers of rows and columns must be the same as the number of objects in <code>x</code> , and the lower diagonal part is used. Otherwise, it is recycled to the number of elements in <code>x</code> .
<code>control</code>	a list of control parameters. See Details .

Details

Additive tree distances are object dissimilarities d satisfying the so-called *additive tree conditions*, also known as *four-point conditions* $d_{ij} + d_{kl} \leq \max(d_{ik} + d_{jl}, d_{il} + d_{jk})$ for all quadruples i, j, k, l . Equivalently, for each such quadruple, the largest two values of the sums $d_{ij} + d_{kl}$, $d_{ik} + d_{jl}$, and $d_{il} + d_{jk}$ must be equal. Centroid distances are additive tree distances where the inequalities in the four-point conditions are strengthened to equalities (such that all three sums are equal), and can be represented as $d_{ij} = g_i + g_j$, i.e., as sums of distances from a “centroid”. See, e.g., Barthélémy and Guénoche (1991) for more details on additive tree distances.

With $L(d) = \sum w_{ij}(x_{ij} - d_{ij})^2$, the problem to be solved by `ls_fit_addtree` is minimizing L over all additive tree distances d . This problem is known to be NP hard.

We provide three heuristics for solving this problem.

Method "`SUMT`" implements the SUMT (Sequential Unconstrained Minimization Technique, Fiacco and McCormick, 1968) approach of de Soete (1983). Incomplete dissimilarities are currently not supported.

Methods "`IP`" and "`IR`" implement the Iterative Projection and Iterative Reduction approaches of Hubert and Arabie (1995) and Roux (1988), respectively. Non-identical weights and incomplete dissimilarities are currently not supported.

See `ls_fit_ultrametric` for details on these methods and available control parameters.

It should be noted that all methods are heuristics which can not be guaranteed to find the global minimum. Standard practice would recommend to use the best solution found in “sufficiently many” replications of the base algorithm.

`ls_fit_centroid` finds the centroid distance d minimizing $L(d)$ (currently, only for the case of identical weights). This optimization problem has a closed-form solution.

There is a `plot` method for the fitted additive tree distance.

Value

An object of class "`cl_addtree`" containing the optimal additive tree distances.

References

J.-P. Barthélémy and A. Guénoche (1991). *Trees and proximity representations*. Chichester: John Wiley & Sons. ISBN 0-471-92263-3.

A. V. Fiacco and G. P. McCormick (1968). *Nonlinear programming: Sequential unconstrained minimization techniques*. New York: John Wiley & Sons.

L. Hubert and P. Arabie (1995). Iterative projection strategies for the least squares fitting of tree structures to proximity data. *British Journal of Mathematical and Statistical Psychology*, **48**, 281–317.

M. Roux (1988). Techniques of approximation for building two tree structures. In C. Hayashi and E. Diday and M. Jambu and N. Ohsumi (Eds.), *Recent Developments in Clustering and Data Analysis*, pages 151–170. New York: Academic Press.

G. de Soete (1983). A least squares algorithm for fitting additive trees to proximity data. *Psychometrika*, **48**, 621–626.

Cassini

Cassini Data

Description

A Cassini data set with 1000 points in 2-dimensional space which are drawn from the uniform distribution on 3 structures. The two outer structures are banana-shaped; the “middle” structure in between them is a circle.

Usage

```
data("Cassini")
```

Format

A classed list with components

`x` a matrix with 1000 rows and 2 columns giving the coordinates of the points.

`classes` a factor indicating which structure the respective points belong to.

Details

Instances of Cassini data sets can be created using function `mlbench.cassini` in package **mlbench**. The data set at hand was obtained using

```
library("mlbench")
set.seed(1234)
Cassini <- mlbench.cassini(1000)
```

Examples

```
data("Cassini")
op <- par(mfcol = c(1, 2))
## Plot the data set:
plot(Cassini$x, col = as.integer(Cassini$classes),
      xlab = "", ylab = "")
## Create a "random" k-means partition of the data:
set.seed(1234)
party <- kmeans(Cassini$x, 3)
## And plot that.
plot(Cassini$x, col = cl_class_ids(party),
      xlab = "", ylab = "")
## (We can see the problem ...)
par(op)
```

CKME

Cassini Data Partitions Obtained by K-Means

Description

A cluster ensemble of 50 k -means partitions of the Cassini data into three classes.

Usage

```
data("CKME")
```

Format

A cluster ensemble of 50 (k -means) partitions.

Details

The ensemble was generated via

```
require("clue")
data("Cassini")
set.seed(1234)
CKME <- cl_boot(Cassini$x, 50, 3)
```

cl_agreement *Agreement Between Partitions or Hierarchies*

Description

Compute the agreement between (ensembles) of partitions or hierarchies.

Usage

```
cl_agreement(x, y = NULL, method = "euclidean", ...)
```

Arguments

x	an ensemble of partitions or hierarchies and dissimilarities, or something coercible to that (see cl_ensemble).
y	NULL (default), or as for x.
method	a character string specifying one of the built-in methods for computing agreement, or a function to be taken as a user-defined method. If a character string, its lower-cased version is matched against the lower-cased names of the available built-in methods using pmatch . See Details for available built-in methods.
...	further arguments to be passed to methods.

Details

If *y* is given, its components must be of the same kind as those of *x* (i.e., components must either all be partitions, or all be hierarchies or dissimilarities).

If all components are partitions, the following built-in methods for measuring agreement between two partitions with respective membership matrices *u* and *v* (brought to a common number of columns) are available:

"euclidean" $1 - d/m$, where *d* is the Euclidean dissimilarity of the memberships, i.e., the square root of the minimal sum of the squared differences of *u* and all column permutations of *v*, and *m* is an upper bound for the maximal Euclidean dissimilarity. See Dimitriadou, Weingessel and Hornik (2002).

"manhattan" $1 - d/m$, where *d* is the Manhattan dissimilarity of the memberships, i.e., the minimal sum of the absolute differences of *u* and all column permutations of *v*, and *m* is an upper bound for the maximal Manhattan dissimilarity.

"Rand" the Rand index (the rate of distinct pairs of objects both in the same class or both in different classes in both partitions), see Rand (1971) or Gordon (1999), page 198. For soft partitions, (currently) the Rand index of the corresponding nearest hard partitions is used.

"cRand" the Rand index corrected for agreement by chance, see Hubert and Arabie (1985) or Gordon (1999), page 198. Can only be used for hard partitions.

"NMI" Normalized Mutual Information, see Strehl and Ghosh (2002). For soft partitions, (currently) the NMI of the corresponding nearest hard partitions is used.

- "KP" the Katz-Powell index, i.e., the product-moment correlation coefficient between the elements of the co-membership matrices $C(u) = uu'$ and $C(v)$, respectively, see Katz and Powell (1953). For soft partitions, (currently) the Katz-Powell index of the corresponding nearest hard partitions is used. (Note that for hard partitions, the (i, j) entry of $C(u)$ is one iff objects i and j are in the same class.)
- "angle" the maximal cosine of the angle between the elements of u and all column permutations of v .
- "diag" the maximal co-classification rate, i.e., the maximal rate of objects with the same class ids in both partitions after arbitrarily permuting the ids.
- "Jaccard" the Jaccard index, i.e., the ratio of the numbers of distinct pairs of objects in the same class in both partitions and in at least one partition, respectively. For soft partitions, (currently) the Jaccard index of the corresponding nearest hard partitions is used.
- "FM" the index of Fowlkes and Mallows (1983), i.e., the ratio $N_{xy} / \sqrt{N_x N_y}$ of the number N_{xy} of distinct pairs of objects in the same class in both partitions and the geometric mean of the numbers N_x and N_y of distinct pairs of objects in the same class in partition x and partition y , respectively. For soft partitions, (currently) the Fowlkes-Mallows index of the corresponding nearest hard partitions is used.

If all components are hierarchies, available built-in methods for measuring agreement between two hierarchies with respective ultrametrics u and v are as follows.

- "euclidean" $1/(1+d)$, where d is the Euclidean dissimilarity of the ultrametrics (i.e., the square root of the sum of the squared differences of u and v).
- "manhattan" $1/(1+d)$, where d is the Manhattan dissimilarity of the ultrametrics (i.e., the sum of the absolute differences of u and v).
- "cophenetic" The cophenetic correlation coefficient. (I.e., the product-moment correlation of the ultrametrics.)
- "angle" the cosine of the angle between the ultrametrics.
- "gamma" $1-d$, where d is the rate of inversions between the associated ultrametrics (i.e., the rate of pairs (i, j) and (k, l) for which $u_{ij} < u_{kl}$ and $v_{ij} > v_{kl}$). (This agreement measure is a linear transformation of Kruskal's γ .)

The measures based on ultrametrics also allow computing agreement with "raw" dissimilarities on the underlying objects (R objects inheriting from class "dist").

If a user-defined agreement method is to be employed, it must be a function taking two clusterings as its arguments.

Symmetric agreement objects of class "cl_agreement" are implemented as symmetric proximity objects with self-proximities identical to one, and inherit from class "cl_proximity". They can be coerced to dense square matrices using `as.matrix`. It is possible to use 2-index matrix-style subscripting for such objects; unless this uses identical row and column indices, this results in a (non-symmetric agreement) object of class "cl_cross_agreement".

Value

If y is NULL, an object of class "cl_agreement" containing the agreements between the all pairs of components of x . Otherwise, an object of class "cl_cross_agreement" with the agreements between the components of x and the components of y .

References

- E. Dimitriadou, A. Weingessel and K. Hornik (2002). A combination scheme for fuzzy clustering. *International Journal of Pattern Recognition and Artificial Intelligence*, **16**, 901–912.
- E. B. Fowlkes and C. L. Mallows (1983). A method for comparing two hierarchical clusterings. *Journal of the American Statistical Association*, **78**, 553–569.
- A. D. Gordon (1999). *Classification* (2nd edition). Boca Raton, FL: Chapman & Hall/CRC.
- L. Hubert and P. Arabie (1985). Comparing partitions. *Journal of Classification*, **2**, 193–218.
- W. M. Rand (1971). Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical Association*, **66**, 846–850.
- L. Katz and J. H. Powell (1953). A proposed index of the conformity of one sociometric measurement to another. *Psychometrika*, **18**, 249–256.
- A. Strehl and J. Ghosh (2002). Cluster ensembles — A knowledge reuse framework for combining multiple partitions. *Journal of Machine Learning Research*, **3**, 583–617.

See Also

[cl_dissimilarity](#); [classAgreement](#) in package **e1071**.

Examples

```
## An ensemble of partitions.
data("CKME")
pens <- CKME[1 : 20] # for saving precious time ...
summary(c(cl_agreement(pens)))
summary(c(cl_agreement(pens, method = "Rand")))
summary(c(cl_agreement(pens, method = "diag")))
cl_agreement(pens[1:5], pens[6:7], method = "NMI")
## Equivalently, using subscripting.
cl_agreement(pens, method = "NMI")[1:5, 6:7]

## An ensemble of hierarchies.
d <- dist(USArrests)
hclust_methods <- c("ward", "single", "complete", "average",
                  "mcquitty", "median", "centroid")
hclust_results <- lapply(hclust_methods, function(m) hclust(d, m))
names(hclust_results) <- hclust_methods
hens <- cl_ensemble(list = hclust_results)
summary(c(cl_agreement(hens)))
## Note that the Euclidean agreements are *very* small.
## This is because the ultrametrics differ substantially in height:
u <- lapply(hens, cl_ultrametric)
round(sapply(u, max), 3)
## Rescaling the ultrametrics to [0, 1] gives:
u <- lapply(u, function(x) (x - min(x)) / (max(x) - min(x)))
shens <- cl_ensemble(list = lapply(u, as.cl_dendrogram))
summary(c(cl_agreement(shens)))
## Au contraire ...
summary(c(cl_agreement(hens, method = "cophenetic")))
cl_agreement(hens[1:3], hens[4:5], method = "gamma")
```

Description

Construct partitions of objects by running a base clustering algorithm on bootstrap samples from a given data set, and “suitably” aggregating these primary partitions.

Usage

```
cl_bag(x, B, k = NULL, algorithm = "kmeans", parameters = NULL,
       method = "DFBC1", control = NULL)
```

Arguments

x	the data set of objects to be clustered, as appropriate for the base clustering algorithm.
B	an integer giving the number of bootstrap replicates.
k	NULL (default), or an integer giving the number of classes to be used for a partitioning base algorithm.
algorithm	a character string or function specifying the base clustering algorithm.
parameters	a named list of additional arguments to be passed to the base algorithm.
method	a character string indicating the bagging method to use. Currently, only method "DFBC1" is available, which implements algorithm <i>BagClust1</i> in Dudoit & Fridlyand (2003).
control	a list of control parameters for the aggregation. Currently, not used.

Details

Bagging for clustering is really a rather general conceptual framework than a specific algorithm. If the primary partitions generated in the bootstrap stage form a cluster ensemble (so that class memberships of the objects in x can be obtained), consensus methods for cluster ensembles (as implemented, e.g., in [cl_consensus](#) and [cl_medoid](#)) can be employed for the aggregation stage. In particular, (possibly new) bagging algorithms can easily be realized by directly running [cl_consensus](#) on the results of [cl_boot](#).

In *BagClust1*, aggregation proceeds by generating a reference partition by running the base clustering algorithm on the whole given data set, and averaging the ensemble memberships after optimally matching them to the reference partition (in fact, by minimizing Euclidean dissimilarity, see [cl_dissimilarity](#)).

If the base clustering algorithm yields prototypes, aggregation can be based on clustering these. This is the idea underlying the “Bagged Clustering” algorithm introduced in Leisch (1999) and implemented by function [bclust](#) in package **e1071**.

Value

An R object representing a partition of the objects given in x .

References

S. Dudoit and J. Fridlyand (2003). Bagging to improve the accuracy of a clustering procedure. *Bioinformatics*, **19**/9, 1090–1099.

F. Leisch (1999). *Bagged Clustering*. Working Paper 51, SFB “Adaptive Information Systems and Modeling in Economics and Management Science”. <http://www.ci.tuwien.ac.at/~leisch/papers/wp51.ps>.

Examples

```
set.seed(1234)
## Run BagClust1 on the Cassini data.
data("Cassini")
party <- cl_bag(Cassini$x, 50, 3)
plot(Cassini$x, col = cl_class_ids(party), xlab = "", ylab = "")
## Actually, using fuzzy c-means as a base learner works much better:
if(require("e1071", quiet = TRUE)) {
  party <- cl_bag(Cassini$x, 20, 3, algorithm = "cmeans")
  plot(Cassini$x, col = cl_class_ids(party), xlab = "", ylab = "")
}
```

cl_boot

Bootstrap Resampling of Clustering Algorithms

Description

Generate bootstrap replicates of the results of applying a “base” clustering algorithm to a given data set.

Usage

```
cl_boot(x, B, k = NULL,
        algorithm = if (is.null(k)) "hclust" else "kmeans",
        parameters = list(), resample = FALSE)
```

Arguments

x	the data set of objects to be clustered, as appropriate for the base clustering algorithm.
B	an integer giving the number of bootstrap replicates.
k	NULL (default), or an integer giving the number of classes to be used for a partitioning base algorithm.
algorithm	a character string or function specifying the base clustering algorithm.
parameters	a named list of additional arguments to be passed to the base algorithm.
resample	a logical indicating whether the data should be resampled in addition to “sampling from the algorithm”. If resampling is used, the class memberships of the objects given in x are predicted from the results of running the base algorithm on bootstrap samples of x.

Details

This is a rather simple-minded function with limited applicability, and mostly useful for studying the effect of (uncontrolled) random initializations of fixed-point partitioning algorithms such as `kmeans` or `cmeans`, see the examples. To study the effect of varying control parameters or explicitly providing random starting values, the respective cluster ensemble has to be generated explicitly (most conveniently by using `replicate` to create a list `lst` of suitable instances of clusterings obtained by the base algorithm, and using `cl_ensemble(list = lst)` to create the ensemble).

Value

A cluster ensemble of length B , with either (if resampling is not used, default) the results of running the base algorithm on the given data set, or (if resampling is used) the memberships for the given data predicted from the results of running the base algorithm on bootstrap samples of the data.

Examples

```
## Study e.g. the effect of random kmeans() initializations.
data("Cassini")
pens <- cl_boot(Cassini$x, 15, 3)
diss <- cl_dissimilarity(pens)
summary(c(diss))
plot(hclust(diss))
```

cl_classes

Cluster Classes

Description

Extract the classes in a partition or hierarchy.

Usage

```
cl_classes(x)
```

Arguments

`x` an R object representing a partition or hierarchy of objects.

Details

For partitions, the classes are the equivalence classes (“clusters”) of the partition; for soft partitions, the classes of the nearest hard partition are used.

For hierarchies represented by trees, the classes are the sets of objects corresponding to (joined at or split by) the nodes of the tree.

Value

A list inheriting from "`cl_classes_of_objects`" of vectors indicating the classes.

Description

Compute the consensus clustering of an ensemble of partitions or hierarchies.

Usage

```
cl_consensus(x, method = NULL, weights = 1, control = list())
```

Arguments

<code>x</code>	an ensemble of partitions or hierarchies, or something coercible to that (see cl_ensemble).
<code>method</code>	a character string specifying one of the built-in methods for computing consensus clusterings, or a function to be taken as a user-defined method, or <code>NULL</code> (default value). If a character string, its lower-cased version is matched against the lower-cased names of the available built-in methods using pmatch . See Details for available built-in methods and defaults.
<code>weights</code>	a numeric vector with non-negative case weights. Recycled to the number of elements in the ensemble given by <code>x</code> if necessary.
<code>control</code>	a list of control parameters. See Details .

Details

Consensus clusterings “synthesize” the information in the elements of a cluster ensemble into a single clustering, often by minimizing a criterion function measuring how dissimilar consensus candidates are from the (elements of) the ensemble (the so-called “optimization approach” to consensus clustering).

The most popular criterion functions are of the form $L(x) = \sum w_b d(x_b, x)^p$, where d is a suitable dissimilarity measure (see [cl_dissimilarity](#)), w_b is the case weight given to element x_b of the ensemble, and $p \geq 1$. If $p = 1$ and minimization is over all possible base clusterings, a consensus solution is called a *median* of the ensemble; if minimization is restricted to the elements of the ensemble, a consensus solution is called a *medoid* (see [cl_medoid](#)). For $p = 2$, we obtain *least squares* consensus partitions and hierarchies (generalized means). See also Gordon (1999) for more information.

If all elements of the ensemble are partitions, the built-in consensus methods compute consensus partitions by minimizing a criterion of the form $L(x) = \sum w_b d(x_b, x)^p$ over all hard or soft partitions x with a given (maximal) number k of classes. Available built-in methods are as follows.

"SE" a fixed-point algorithm for obtaining *soft* least squares Euclidean consensus partitions (i.e., for minimizing L with Euclidean dissimilarity d and $p = 2$ over all soft partitions with a given maximal number of classes).

This iterates between individually matching all partitions to the current approximation to the consensus partition, and computing the next approximation as the membership matrix closest to a suitable weighted average of the memberships of all partitions after permuting their columns for the optimal matchings of class ids.

The following control parameters are available for this method.

`k` an integer giving the number of classes to be used for the least squares consensus partition. By default, the maximal number of classes in the ensemble is used.

`maxiter` an integer giving the maximal number of iterations to be performed. Defaults to 100.

`nruns` an integer giving the number of runs to be performed. Defaults to 1.

`reltol` the relative convergence tolerance. Defaults to `sqrt(.Machine$double.eps)`.

`start` a matrix with number of rows equal to the number of objects of the cluster ensemble, and k columns, to be used as a starting value, or a list of such matrices. By default, suitable random membership matrices are used.

`verbose` a logical indicating whether to provide some output on minimization progress. Defaults to `getOption("verbose")`.

In the case of multiple runs, the first optimum found is returned.

This method can also be referred to as "soft/euclidean".

"GV1" the fixed-point algorithm for the "first model" in Gordon and Vichi (2001) for minimizing L with d being GV1 dissimilarity and $p = 2$ over all soft partitions with a given maximal number of classes.

This is similar to "SE", but uses GV1 rather than Euclidean dissimilarity.

Available control parameters are the same as for "SE".

"DWH" an extension of the greedy algorithm in Dimitriadou, Weingessel and Hornik (2002) for (approximately) obtaining soft least squares Euclidean consensus partitions. The reference provides some structure theory relating finding the consensus partition to an instance of the multiple assignment problem, which is known to be NP-hard, and suggests a simple heuristic based on successively matching an individual partition x_b to the current approximation to the consensus partition, and compute the memberships of the next approximation as a weighted average of those of the current one and of x_b after permuting its columns for the optimal matching of class ids.

The following control parameters are available for this method.

`k` an integer giving the number of classes to be used for the least squares consensus partition. By default, the maximal number of classes in the ensemble is used.

`order` a permutation of the integers from 1 to the size of the ensemble, specifying the order in which the partitions in the ensemble should be aggregated. Defaults to using a random permutation (unlike the reference, which does not permute at all).

"HE" a fixed-point algorithm for obtaining *hard* least squares Euclidean consensus partitions (i.e., for minimizing L with Euclidean dissimilarity d and $p = 2$ over all hard partitions with a given maximal number of classes.)

Available control parameters are the same as for "SE".

This method can also be referred to as "hard/euclidean".

"SM" a fixed-point algorithm for obtaining *soft* median Manhattan consensus partitions (i.e., for minimizing L with Manhattan dissimilarity d and $p = 1$ over all soft partitions with a given maximal number of classes).

Available control parameters are the same as for "SE".

This method can also be referred to as "soft/manhattan".

"SM" a fixed-point algorithm for obtaining *hard* median Manhattan consensus partitions (i.e., for minimizing L with Manhattan dissimilarity d and $p = 1$ over all hard partitions with a given maximal number of classes).

Available control parameters are the same as for "SE".

This method can also be referred to as "hard/manhattan".

"GV3" a SUMT algorithm for the "third model" in Gordon and Vichi (2001) for minimizing L with d being co-membership dissimilarity and $p = 2$. (See `sumt` for more information on the SUMT approach.) This optimization problem is equivalent to finding the membership matrix m for which the sum of the squared differences between $C(m) = mm'$ and the weighted average co-membership matrix $\sum_b w_b C(m_b)$ of the partitions is minimal.

Available control parameters are `method`, `control`, `eps`, `q`, and `verbose`, which have the same roles as for `sumt`, and the following.

`k` an integer giving the number of classes to be used for the least squares consensus partition.

By default, the maximal number of classes in the ensemble is used.

`nruns` an integer giving the number of runs to be performed. Defaults to 1.

`start` a matrix with number of rows equal to the size of the cluster ensemble, and k columns, to be used as a starting value, or a list of such matrices. By default, a membership based on a rank k approximation to the weighted average co-membership matrix is used.

In the case of multiple runs, the first optimum found is returned.

"soft/symdiff" a SUMT approach for minimizing $L = \sum w_b d(x_b, x)$ over all soft partitions with a given maximal number of classes, where d is the Manhattan dissimilarity of the co-membership matrices (coinciding with `symdiff` partition dissimilarity in the case of hard partitions).

Available control parameters are the same as for "GV3".

"hard/symdiff" an exact solver for minimizing $L = \sum w_b d(x_b, x)$ over all hard partitions (possibly with a given maximal number of classes as specified by the control parameter `k`), where d is `symdiff` partition dissimilarity (so that soft partitions in the ensemble are replaced by their closest hard partitions), or equivalently, Rand distance or pair-bonds (Boorman-Arabie D) distance. The consensus solution is found via mixed linear or quadratic programming.

By default, method "SE" is used for ensembles of partitions.

If all elements of the ensemble are hierarchies, the following built-in methods for computing consensus hierarchies are available.

"euclidean" an algorithm for minimizing $L(x) = \sum w_b d(x_b, x)^2$ over all dendrograms, where d is Euclidean dissimilarity. This is equivalent to finding the best least squares ultrametric approximation of the weighted average $d = \sum w_b u_b$ of the ultrametrics u_b of the hierarchies x_b , which is attempted by calling `ls_fit_ultrametric` on d with appropriate control parameters.

This method can also be referred to as "cophenetic".

"manhattan" a SUMT for minimizing $L = \sum w_b d(x_b, x)$ over all dendrograms, where d is Manhattan dissimilarity.

Available control parameters are the same as for "euclidean".

"majority" a hierarchy obtained from an extension of the majority consensus tree of Margush and McMorris (1981), which minimizes $L(x) = \sum w_b d(x_b, x)$ over all dendrograms, where d is the symmetric difference dissimilarity. The unweighted p -majority tree is the n -tree (hierarchy in the strict sense) consisting of all subsets of objects contained in more than $100p$ percent of the n -trees T_b induced by the dendrograms, where $1/2 \leq p < 1$ and $p = 1/2$ (default) corresponds to the standard majority tree. In the weighted case, it consists of all subsets A for which $\sum_{b:A \in T_b} w_b > Wp$, where $W = \sum_b w_b$. We also allow for $p = 1$, which gives the *strict consensus tree* consisting of all subsets contained in each of the n -trees. The majority dendrogram returned is a representation of the majority tree where all splits have height one. The fraction p can be specified via the control parameter `p`.

By default, method "euclidean" is used for ensembles of hierarchies.

If a user-defined consensus method is to be employed, it must be a function taking the cluster ensemble, the case weights, and a list of control parameters as its arguments, with formals named `x`, `weights`, and `control`, respectively.

Most built-in methods use heuristics for solving hard optimization problems, and cannot be guaranteed to find a global minimum. Standard practice would recommend to use the best solution found in "sufficiently many" replications of the methods.

Value

The consensus partition or hierarchy.

References

- E. Dimitriadou, A. Weingessel and K. Hornik (2002). A combination scheme for fuzzy clustering. *International Journal of Pattern Recognition and Artificial Intelligence*, **16**, 901–912.
- A. D. Gordon and M. Vichi (2001). Fuzzy partition models for fitting a set of partitions. *Psychometrika*, **66**, 229–248.
- A. D. Gordon (1999). *Classification* (2nd edition). Boca Raton, FL: Chapman & Hall/CRC.
- T. Margush and F. R. McMorris (1981). Consensus n -trees. *Bulletin of Mathematical Biology*, **43**, 239–244.

See Also

[cl_medoid](#), [consensus](#)

Examples

```
## Consensus partition for the Rosenberg-Kim kinship terms partition
## data based on co-membership dissimilarities.
data("Kinship82")
m1 <- cl_consensus(Kinship82, method = "GV3",
                  control = list(k = 3, verbose = TRUE))
## (Note that one should really use several replicates of this.)
## Value for criterion function to be minimized:
sum(cl_dissimilarity(Kinship82, m1, "comem") ^ 2)
## Compare to the consensus solution given in Gordon & Vichi (2001).
data("Kinship82_Consensus")
```

```

m2 <- Kinship82_Consensus[["JMF"]]
sum(cl_dissimilarity(Kinship82, m2, "comem") ^ 2)
## Seems we get a better solution ...
## How dissimilar are these solutions?
cl_dissimilarity(m1, m2, "comem")
## How "fuzzy" are they?
cl_fuzziness(cl_ensemble(m1, m2))
## Do the "nearest" hard partitions fully agree?
cl_dissimilarity(as.cl_hard_partition(m1),
                 as.cl_hard_partition(m2))

## Consensus partition for the Gordon and Vichi (2001) macroeconomic
## partition data based on Euclidean dissimilarities.
data("GVME")
set.seed(1)
## First, using k = 2 classes.
m1 <- cl_consensus(GVME, method = "GV1",
                  control = list(k = 2, verbose = TRUE))
## (Note that one should really use several replicates of this.)
## Value of criterion function to be minimized:
sum(cl_dissimilarity(GVME, m1, "GV1") ^ 2)
## Compare to the consensus solution given in Gordon & Vichi (2001).
data("GVME_Consensus")
m2 <- GVME_Consensus[["MF1/2"]]
sum(cl_dissimilarity(GVME, m2, "GV1") ^ 2)
## Seems we get a slightly better solution ...
## But note that
cl_dissimilarity(m1, m2, "GV1")
## and that the maximal deviation of the memberships is
max(abs(cl_membership(m1) - cl_membership(m2)))
## so the differences seem to be due to rounding.
## Do the "nearest" hard partitions fully agree?
table(cl_class_ids(m1), cl_class_ids(m2))

## And now for k = 3 classes.
m1 <- cl_consensus(GVME, method = "GV1",
                  control = list(k = 3, verbose = TRUE))
sum(cl_dissimilarity(GVME, m1, "GV1") ^ 2)
## Compare to the consensus solution given in Gordon & Vichi (2001).
m2 <- GVME_Consensus[["MF1/3"]]
sum(cl_dissimilarity(GVME, m2, "GV1") ^ 2)
## This time we look much better ...
## How dissimilar are these solutions?
cl_dissimilarity(m1, m2, "GV1")
## Do the "nearest" hard partitions fully agree?
table(cl_class_ids(m1), cl_class_ids(m2))

```

Description

Compute the dissimilarity between (ensembles) of partitions or hierarchies.

Usage

```
cl_dissimilarity(x, y = NULL, method = "euclidean", ...)
```

Arguments

x	an ensemble of partitions or hierarchies and dissimilarities, or something coercible to that (see cl_ensemble).
y	NULL (default), or as for x.
method	a character string specifying one of the built-in methods for computing dissimilarity, or a function to be taken as a user-defined method. If a character string, its lower-cased version is matched against the lower-cased names of the available built-in methods using pmatch . See Details for available built-in methods.
...	further arguments to be passed to methods.

Details

If *y* is given, its components must be of the same kind as those of *x* (i.e., components must either all be partitions, or all be hierarchies or dissimilarities).

If all components are partitions, the following built-in methods for measuring dissimilarity between two partitions with respective membership matrices *u* and *v* (brought to a common number of columns) are available:

"euclidean" the Euclidean dissimilarity of the memberships, i.e., the square root of the minimal sum of the squared differences of *u* and all column permutations of *v*. See Dimitriadou, Weingessel and Hornik (2002).

"manhattan" the Manhattan dissimilarity of the memberships, i.e., the minimal sum of the absolute differences of *u* and all column permutations of *v*.

"comemberships" the Euclidean dissimilarity of the elements of the co-membership matrices $C(u) = uu'$ and $C(v)$, i.e., the square root of the sum of the squared differences of $C(u)$ and $C(v)$.

"symdiff" the cardinality of the symmetric set difference of the sets of co-classified pairs of distinct objects in the partitions. I.e., the number of distinct pairs of objects in the same class in exactly one of the partitions. (Alternatively, the cardinality of the symmetric set difference between the (binary) equivalence relations corresponding to the partitions.) For soft partitions, (currently) the symmetric set difference of the corresponding nearest hard partitions is used.

"Rand" the Rand distance, i.e., the rate of distinct pairs of objects in the same class in exactly one of the partitions. (Related to the Rand index *a* via the linear transformation $d = (1 - a)/2$.) For soft partitions, (currently) the Rand distance of the corresponding nearest hard partitions is used.

"GV1" the square root of the dissimilarity Δ_1 used for the first model in Gordon and Vichi (2001), i.e., the square root of the minimal sum of the squared differences of the *matched* non-zero columns of *u* and *v*.

"BA/*d*" distance measures for hard partitions discussed in Boorman and Arabie (1972), with *d* one of 'A', 'C', 'D', or 'E'. For soft partitions, the distances of the corresponding nearest hard partitions are used.

"BA/A" is the minimum number of single element moves (move from one class to another or a new one) needed to transform one partition into the other. Introduced in Rubin (1967).

"BA/C" is the minimum number of lattice moves for transforming one partition into the other, where partitions are said to be connected by a lattice move if one is *just* finer than the other (i.e., there is no other partition between them) in the partition lattice (see [cl_meet](#)). Equivalently, with *z* the join of *x* and *y* and *S* giving the number of classes, this can be written as $S(x) + S(y) - 2S(z)$. Attributed to David Pavy.

"BA/D" is the "pair-bonds" distance, which can be defined as $S(x) + S(y) - 2S(z)$, with *z* the meet of *x* and *y* and *S* the *supervaluation* (i.e., non-decreasing with respect to the partial order on the partition lattice) function $\sum_i (n_i(n_i - 1)) / (n(n - 1))$, where the n_i are the numbers of objects in the respective classes of the partition (such that $n_i(n_i - 1)/2$ are the numbers of pair bonds in the classes), and *n* the total number of objects.

"BA/E" is the normalized information distance, defined as $1 - I/H$, where *I* is the average mutual information between the partitions, and *H* is the average entropy of the meet *z* of the partitions. Introduced in Rajsiki (1961).

(Boorman and Arabie also discuss a distance measure (*B*) based on the minimum number of set moves needed to transform one partition into the other, which, differently from the *A* and *C* distance measures is hard to compute (Day, 1981) and (currently) not provided.)

"VI" Variation of Information, see Meila (2003). If . . . has an argument named `weights`, it is taken to specify case weights.

"Mallows" the Mallows-type distance by Zhou, Li and Zha (2005), which is related to the Monge-Kantorovich mass transfer problem, and given as the *p*-th root of the minimal value of the transportation problem $\sum w_{jk} \sum_i |u_{ij} - v_{ik}|^p$ with constraints $w_{jk} \geq 0$, $\sum_j w_{jk} = \alpha_j$, $\sum_k w_{jk} = \beta_k$, where $\sum_j \alpha_j = \sum_k \beta_k$. The parameters *p*, α and β all default to one (in this case, the Mallows distance coincides with the Manhattan dissimilarity), and can be specified via additional arguments named `p`, `alpha`, and `beta`, respectively.

"CSSD" the Cluster Similarity Sensitive Distance of Zhou, Li and Zha (2005), which is given as the minimal value of $\sum_{k,l} (1 - 2w_{kl} / (\alpha_k + \beta_l)) L_{kl}$, where $L_{kl} = \sum_i u_{ik} v_{il} d(p_{x;k}, p_{y;l})$ with $p_{x;k}$ and $p_{y;l}$ the prototype of the *k*-th class of *x* and the *l*-th class of *y*, respectively, *d* is the distance between these, and the w_{kl} as for Mallows distance. If prototypes are matrices, the Euclidean distance between these is used as default. Using the additional argument `L`, one can give a matrix of L_{kl} values, or the function *d*. Parameters α and β all default to one, and can be specified via additional arguments named `alpha` and `beta`, respectively.

For hard partitions, both Manhattan and squared Euclidean dissimilarity give twice the *transfer distance* (Charon et al., 2005), which is the minimum number of objects that must be removed so that the implied partitions (restrictions to the remaining objects) are identical. This is also known as the *R-metric* in Day (1981), i.e., the number of augmentations and removals of single objects needed to transform one partition into the other, and the *partition-distance* in Gusfield (2002), and equals twice the number of single element moves distance of Boorman and Arabie.

For hard partitions, the pair-bonds (Boorman-Arabie *D*) distance is identical to the Rand distance, and can also be written as the Manhattan distance between the co-membership matrices corresponding to the partitions, or equivalently, their symdiff distance, normalized by $n(n - 1)$.

If all components are hierarchies, available built-in methods for measuring dissimilarity between two hierarchies with respective ultrametrics u and v are as follows.

- "euclidean" the Euclidean dissimilarity of the ultrametrics (i.e., the square root of the sum of the squared differences of u and v).
- "manhattan" the Manhattan dissimilarity of the ultrametrics (i.e., the sum of the absolute differences of u and v).
- "cophenetic" $1 - c^2$, where c is the cophenetic correlation coefficient (i.e., the product-moment correlation of the ultrametrics).
- "gamma" the rate of inversions between the ultrametrics (i.e., the rate of pairs (i, j) and (k, l) for which $u_{ij} < u_{kl}$ and $v_{ij} > v_{kl}$).
- "symdiff" the cardinality of the symmetric set difference of the sets of classes (hierarchies in the strict sense) induced by the dendrograms. I.e., the number of sets of objects obtained by a split in exactly one of the hierarchies.
- "Chebyshev" the Chebyshev (maximal) dissimilarity of the ultrametrics (i.e., the maximum of the absolute differences of u and v).
- "Lyapunov" the logarithm of the product of the maximal and minimal ratios of the ultrametrics. This is also known as the "Hilbert projective metric" on the cone represented by the ultrametrics (e.g., Jardine & Sibson (1971), page 107), and only defined for *strict* ultrametrics (which are strictly positive for distinct objects).
- "BO" the m_δ family of tree metrics by Boorman and Olivier (1973), which are of the form $m_\delta = \int_0^\infty \delta(p(h), q(h)) dh$, where $p(h)$ and $q(h)$ are the hard partitions obtaining by cutting the trees (dendrograms) at height h , and δ is a suitably dissimilarity measure for partitions. In particular, when taking δ as symdiff or Rand dissimilarity, m_δ is the Manhattan dissimilarity of the hierarchies.
If . . . has an argument named `delta` it is taken to specify the partition dissimilarity δ to be employed.

The measures based on ultrametrics also allow computing dissimilarity with "raw" dissimilarities on the underlying objects (R objects inheriting from class "dist").

If a user-defined dissimilarity method is to be employed, it must be a function taking two clusterings as its arguments.

Symmetric dissimilarity objects of class "cl_dissimilarity" are implemented as symmetric proximity objects with self-proximities identical to zero, and inherit from class "cl_proximity". They can be coerced to dense square matrices using `as.matrix`. It is possible to use 2-index matrix-style subscripting for such objects; unless this uses identical row and column indices, this results in a (non-symmetric dissimilarity) object of class "cl_cross_dissimilarity".

Symmetric dissimilarity objects also inherit from class "dist" (although they currently do not "strictly" extend this class), thus making it possible to use them directly for clustering algorithms based on dissimilarity matrices of this class, see the examples.

Value

If y is NULL, an object of class "cl_dissimilarity" containing the dissimilarities between all pairs of components of x . Otherwise, an object of class "cl_cross_dissimilarity" with the dissimilarities between the components of x and the components of y .

References

- S. A. Boorman and P. Arabie (1972). Structural measures and the method of sorting. In R. N. Shepard, A. K. Romney, & S. B. Nerlove (eds.), *Multidimensional Scaling: Theory and Applications in the Behavioral Sciences, 1: Theory* (pages 225–249). New York: Seminar Press.
- S. A. Boorman and D. C. Olivier (1973). Metrics on spaces of finite trees. *Journal of Mathematical Psychology*, **10**, 26–59.
- I. Charon, L. Denoeud, A. Guénoche and O. Hudry (2005). *Maximum Transfer Distance Between Partitions*. Technical Report 2005D003, Ecole Nationale Supérieure des Télécommunications — Paris. http://www.enst.fr/_data/files/docs/id_515_1128675112_271.pdf
- W. E. H. Day (1981). The complexity of computing metric distances between partitions. *Mathematical Social Sciences*, **1**, 269–287.
- E. Dimitriadou, A. Weingessel and K. Hornik (2002). A combination scheme for fuzzy clustering. *International Journal of Pattern Recognition and Artificial Intelligence*, **16**, 901–912.
- A. D. Gordon and M. Vichi (2001). Fuzzy partition models for fitting a set of partitions. *Psychometrika*, **66**, 229–248.
- D. Gusfield (2002). Partition-distance: A problem and class of perfect graphs arising in clustering. *Information Processing Letters*, **82**, 159–164.
- N. Jardine and E. Sibson (1971). *Mathematical Taxonomy*. London: Wiley.
- M. Meila (2003). Comparing clusterings by the variation of information. In B. Schölkopf and M. K. Warmuth (eds.), *Learning Theory and Kernel Machines*, pages 173–187. Springer-Verlag: Lecture Notes in Computer Science 2777.
- C. Rajski (1961). A metric space of discrete probability distributions, *Information and Control*, **4**, 371–377.
- J. Rubin (1967). Optimal classification into groups: An approach for solving the taxonomy problem. *Journal of Theoretical Biology*, **15**, 103–144.
- D. Zhou, J. Li and H. Zha (2005). A new Mallows distance based metric for comparing clusterings. In *Proceedings of the 22nd international Conference on Machine Learning* (Bonn, Germany, August 07–11, 2005), pages 1028–1035. ICML '05, volume 119. ACM Press, New York, NY. DOI: <http://doi.acm.org/10.1145/1102351.1102481>

See Also

[cl_agreement](#)

Examples

```
## An ensemble of partitions.
data("CKME")
pens <- CKME[1 : 30]
diss <- cl_dissimilarity(pens)
summary(c(diss))
cl_dissimilarity(pens[1:5], pens[6:7])
## Equivalently, using subscripting.
diss[1:5, 6:7]
## Can use the dissimilarities for "secondary" clustering
## (e.g. obtaining hierarchies of partitions):
```

```

hc <- hclust(diss)
plot(hc)

## Example from Boorman and Arabie (1972).
P1 <- as.cl_partition(c(1, 2, 2, 2, 3, 3, 2, 2))
P2 <- as.cl_partition(c(1, 1, 2, 2, 3, 3, 4, 4))
cl_dissimilarity(P1, P2, "BA/A")
cl_dissimilarity(P1, P2, "BA/C")

## Hierarchical clustering.
d <- dist(USArrests)
x <- hclust(d)
cl_dissimilarity(x, d, "cophenetic")
cl_dissimilarity(x, d, "gamma")

```

cl_ensemble

Cluster Ensembles

Description

Creation and manipulation of cluster ensembles.

Usage

```

cl_ensemble(..., list = NULL)
as.cl_ensemble(x)
is.cl_ensemble(x)

```

Arguments

...	R objects representing clusterings of or dissimilarities between the same objects.
list	a list of R objects as in ...
x	for <code>as.cl_ensemble</code> , an R object as in ...; for <code>is.cl_ensemble</code> , an arbitrary R object.

Details

`cl_ensemble` creates “cluster ensembles”, which are realized as lists of clusterings (or dissimilarities) with additional class information, always inheriting from “`cl_ensemble`”. All elements of the ensemble must have the same number of objects.

If all elements are partitions, the ensemble has class “`cl_partition_ensemble`”; if all elements are dendrograms, it has class “`cl_dendrogram_ensemble`” and inherits from “`cl_hierarchy_ensemble`”; if all elements are hierarchies (but not always dendrograms), it has class “`cl_hierarchy_ensemble`”. Note that empty or “mixed” ensembles cannot be categorized according to the kind of elements they contain, and hence only have class “`cl_ensemble`”.

The list representation makes it possible to use `lapply` for computations on the individual clusterings in (i.e., the components of) a cluster ensemble.

Available methods for cluster ensembles include those for `subscripting`, `c`, `rep`, and `print`. There is also a `plot` method for ensembles for which all elements can be plotted (currently, additive trees, dendrograms and ultrametrics).

Value

`cl_ensemble` returns a list of the given clusterings or dissimilarities, with additional class information (see **Details**).

Examples

```
d <- dist(USArrests)
hclust_methods <- c("ward", "single", "complete", "average",
                   "mcquitty", "median", "centroid")
hclust_results <- lapply(hclust_methods, function(m) hclust(d, m))
names(hclust_results) <- hclust_methods
## Now create an ensemble from the results.
hens <- cl_ensemble(list = hclust_results)
hens
## Subscripting.
hens[1 : 3]
## Replication.
rep(hens, 3)
## Plotting.
plot(hens, main = names(hens))
## And continue to analyze the ensemble, e.g.
round(cl_dissimilarity(hens, method = "gamma"), 4)
```

cl_fuzziness

Partition Fuzziness

Description

Compute the fuzziness of partitions.

Usage

```
cl_fuzziness(x, method = NULL, normalize = TRUE)
```

Arguments

<code>x</code>	a cluster ensemble of partitions, or an R object coercible to such.
<code>method</code>	a character string indicating the fuzziness measure to be employed, or <code>NULL</code> (default), or a function to be taken as a user-defined method. Currently available built-in methods are "PC" (Partition Coefficient) and "PE" (Partition Entropy), with the default corresponding to the first one. If <code>method</code> is a character string, its lower-cased version is matched against the lower-cased names of the available built-in methods using <code>pmatch</code> .

`normalize` a logical indicating whether the fuzziness measure should be normalized in a way that hard partitions have value 0, and “completely fuzzy” partitions (where for all objects, all classes get the same membership) have value 1.

Details

If m contains the membership values of a partition, the (unnormalized) Partition Coefficient and Partition Entropy are given by $\sum_{n,i} m_{n,i}^2$ and $\sum_{n,i} H(m_{n,i})$, respectively, where $H(u) = u \log u - (1 - u) \log(1 - u)$.

Note that the normalization used here is different from the normalizations typically found in the literature.

If a user-defined fuzziness method is to be employed, it must be a function taking a matrix of membership values and a logical to indicate whether normalization is to be performed as its arguments (in that order; argument names are not used).

Value

An object of class "cl_fuzziness" giving the fuzziness values.

References

J. C. Bezdek (1981). *Pattern Recognition with Fuzzy Objective Function Algorithms*. New York: Plenum.

See Also

Function `fclustIndex` in package **e1071**, which also computes several other “fuzzy cluster indexes” (typically based on more information than just the membership values).

Examples

```
if(require("e1071", quiet = TRUE)) {
  ## Use an on-line version of fuzzy c-means from package e1071 if
  ## available.
  data("Cassini")
  pens <- cl_boot(Cassini$x, B = 15, k = 3, algorithm = "cmeans",
                 parameters = list(method = "ufcl"))
  pens
  summary(cl_fuzziness(pens, "PC"))
  summary(cl_fuzziness(pens, "PE"))
}
```

cl_margin	<i>Membership Margins</i>
-----------	---------------------------

Description

Compute the *margin* of the memberships of a partition, i.e., the difference between the largest and second largest membership values of the respective objects.

Usage

```
cl_margin(x)
```

Arguments

`x` an R object representing a partition of objects.

Details

For hard partitions, the margins are always 1.

For soft partitions, the margins may be taken as an indication of the “sureness” of classifying an object to the class with maximum membership value.

Examples

```
data("GVME")
## Look at the classes obtained for 1980:
split(cl_object_names(GVME[["1980"]]), cl_class_ids(GVME[["1980"]]))
## Margins:
x <- cl_margin(GVME[["1980"]])
## Add names, and sort:
names(x) <- cl_object_names(GVME[["1980"]])
sort(x)
## Note the "uncertainty" of assigning Egypt to the "intermediate" class
## of nations.
```

cl_medoid	<i>Medoid Partitions and Hierarchies</i>
-----------	------------------------------------------

Description

Compute the medoid of an ensemble of partitions or hierarchies, i.e., the element of the ensemble minimizing the sum of dissimilarities to all other elements.

Usage

```
cl_medoid(x, method = "euclidean")
```

Arguments

- `x` an ensemble of partitions or hierarchies, or something coercible to that (see [cl_ensemble](#)).
- `method` a character string or a function, as for argument `method` of function [cl_dissimilarity](#).

Details

Medoid clusterings are special cases of “consensus” clusterings characterized as the solutions of an optimization problem. See Gordon (2001) for more information.

The dissimilarities `d` for determining the medoid are obtained by calling [cl_dissimilarity](#) with arguments `x` and `method`. The medoid can then be found as the (first) row index for which the row sum of `as.matrix(d)` is minimal. Modulo possible differences in the case of ties, this gives the same results as (the medoid obtained by) [pam](#) in package **cluster**.

Value

The medoid partition or hierarchy.

References

A. D. Gordon (1999). *Classification* (2nd edition). Boca Raton, FL: Chapman & Hall/CRC.

See Also

[cl_consensus](#)

Examples

```
## An ensemble of partitions.
data("CKME")
pens <- CKME[1 : 20]
m1 <- cl_medoid(pens)
diss <- cl_dissimilarity(pens)
require("cluster")
m2 <- pens[[pam(diss, 1)$medoids]]
## Agreement of medoid consensus partitions.
cl_agreement(m1, m2)
## Or, more straightforwardly:
table(cl_class_ids(m1), cl_class_ids(m2))
```

cl_membership

Memberships of Partitions

Description

Compute the memberships values for objects representing partitions.

Usage

```
cl_membership(x, k = n_of_classes(x))
as.cl_membership(x)
```

Arguments

x an R object representing a partition of objects (for `cl_membership`) or raw memberships or class ids (for `as.cl_membership`).

k an integer giving the number of columns (corresponding to class ids) to be used in the membership matrix. Must not be less, and default to, the number of classes in the partition.

Details

`cl_membership` is a generic function.

The methods provided in package **clue** handle the partitions obtained from clustering functions in the base R distribution, as well as packages **RWeka**, **cba**, **cclust**, **cluster**, **e1071**, **flexclust**, **flexmix**, **kernlab**, and **mclust** (and of course, **clue** itself).

`as.cl_membership` can be used for coercing “raw” class ids (given as atomic vectors) or membership values (given as numeric matrices) to membership objects.

Value

An object of class "cl_membership" with the matrix of membership values.

See Also

[is.cl_partition](#)

Examples

```
## Getting the memberships of a single soft partition.
d <- dist(USArrests)
hclust_methods <- c("ward", "single", "complete", "average",
                  "mcquitty", "median", "centroid")
hclust_results <- lapply(hclust_methods, function(m) hclust(d, m))
names(hclust_results) <- hclust_methods
## Now create an ensemble from the results.
hens <- cl_ensemble(list = hclust_results)
## Create a dissimilarity object from this.
dl <- cl_dissimilarity(hens)
## And compute a soft partition.
require("cluster")
party <- fanny(dl, 2)
round(cl_membership(party), 5)
## The "nearest" hard partition to this:
as.cl_hard_partition(party)
## (which has the same class ids as cl_class_ids(party)).

## Extracting the memberships from the elements of an ensemble of
```

```
## partitions.
pens <- cl_boot(USArrests, 30, 3)
pens
mems <- lapply(pens, cl_membership)
## And turning these raw memberships into an ensemble of partitions.
pens <- cl_ensemble(list = lapply(mems, as.cl_partition))
pens
pens[[length(pens)]]
```

cl_object_names *Find Object Names*

Description

Find the names of the objects from which a taxonomy (partition or hierarchy) or proximity was obtained.

Usage

```
cl_object_names(x)
```

Arguments

x an R object representing a taxonomy or proximity.

Details

This is a generic function.

The methods provided in package **clue** handle the partitions and hierarchies obtained from clustering functions in the base R distribution, as well as packages **RWeka**, **ape**, **cba**, **cclust**, **cluster**, **e1071**, **flexclust**, **flexmix**, **kernlab**, and **mclust** (and of course, **clue** itself), in as much as possible.

There is also a method for object dissimilarities which inherit from class "[dist](#)".

Value

A character vector of length `n_of_objects(x)` in case the names of the objects could be determined, or `NULL`.

 cl_pam

K-Medoids Partitions of Clusterings

Description

Compute k -medoids partitions of clusterings.

Usage

```
cl_pam(x, k, method = "euclidean", solver = c("pam", "kmedoids"))
```

Arguments

x	an ensemble of partitions or hierarchies, or something coercible to that (see cl_ensemble).
k	an integer giving the number of classes to be used in the partition.
method	a character string or a function, as for argument <code>method</code> of function cl_dissimilarity .
solver	a character string indicating the k -medoids solver to be employed. May be abbreviated. If "pam" (default), the Partitioning Around Medoids (Kaufman & Rousseeuw (1990), Chapter 2) heuristic pam of package cluster is used. Otherwise, the exact algorithm of kmedoids is employed.

Details

An optimal k -medoids partition of the given cluster ensemble is defined as a partition of the objects x_i (the elements of the ensemble) into k classes C_1, \dots, C_k such that the criterion function $L = \sum_{l=1}^k \min_{j \in C_l} \sum_{i \in C_l} d(x_i, x_j)$ is minimized.

Such secondary partitions (e.g., Gordon & Vichi, 1998) are obtained by computing the dissimilarities d of the objects in the ensemble for the given dissimilarity method, and applying a dissimilarity-based k -medoids solver to d .

Value

An object of class "cl_pam" representing the obtained "secondary" partition, which is a list with the following components.

cluster	the class ids of the partition.
medoid_ids	the indices of the medoids.
prototypes	a cluster ensemble with the k prototypes (medoids).
criterion	the value of the criterion function of the partition.
description	a character string indicating the dissimilarity method employed.

References

- L. Kaufman and P. J. Rousseeuw (1990). *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley, New York.
- A. D. Gordon and M. Vichi (1998). Partitions of partitions. *Journal of Classification*, **15**, 265–285.

See Also

[cl_pclust](#) for more general prototype-based partitions of clusterings.

Examples

```
data("Kinship82")
party <- cl_pam(Kinship82, 3, "symdiff")
## Compare results with tables 5 and 6 in Gordon & Vichi (1998).
party
lapply(cl_prototypes(party), cl_classes)
table(cl_class_ids(party))
```

cl_pclust

Prototype-Based Partitions of Clusterings

Description

Compute prototype-based partitions of a cluster ensemble by minimizing $\sum w_b u_{bj}^m d(x_b, p_j)^e$, the sum of the case-weighted and membership-weighted e -th powers of the dissimilarities between the elements x_b of the ensemble and the prototypes p_j , for suitable dissimilarities d and exponents e .

Usage

```
cl_pclust(x, k, method = NULL, m = 1, weights = 1, control = list())
```

Arguments

- | | |
|---------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| x | an ensemble of partitions or hierarchies, or something coercible to that (see cl_ensemble). |
| k | an integer giving the number of classes to be used in the partition. |
| method | the consensus method to be employed, see cl_consensus . |
| m | a number not less than 1 controlling the softness of the partition (as the “fuzzification parameter” of the fuzzy c -means algorithm). The default value of 1 corresponds to hard partitions obtained from a generalized k -means problem; values greater than one give partitions of increasing softness obtained from a generalized fuzzy c -means problem. |
| weights | a numeric vector of non-negative case weights. Recycled to the number of elements in the ensemble given by x if necessary. |
| control | a list of control parameters. See Details . |

Details

Partitioning is performed using `pclust` via a family constructed from `method`. The dissimilarities d and exponent e are implied by the consensus method employed, and inferred via a registration mechanism currently only made available to built-in consensus methods. The default methods compute Least Squares Euclidean consensus clusterings, i.e., use Euclidean dissimilarity d and $e = 2$.

For $m = 1$, the partitioning procedure was introduced by Gaul and Schader (1988) for “Clusterwise Aggregation of Relations” (with the same domains), containing equivalence relations, i.e., hard partitions, as a special case.

Available control parameters are as for `pclust`.

The fixed point approach employed is a heuristic which cannot be guaranteed to find the global minimum (as this is already true for the computation of consensus clusterings). Standard practice would recommend to use the best solution found in “sufficiently many” replications of the base algorithm.

Value

An object of class "cl_partition" representing the obtained “secondary” partition by an object of class "cl_pclust", which is a list containing at least the following components.

<code>prototypes</code>	a cluster ensemble with the k prototypes.
<code>membership</code>	an object of class "cl_membership" with the membership values u_{bj} .
<code>cluster</code>	the class ids of the nearest hard partition.
<code>silhouette</code>	Silhouette information for the partition, see <code>silhouette</code> .
<code>validity</code>	precomputed validity measures for the partition.
<code>m</code>	the softness control argument.
<code>call</code>	the matched call.
<code>d</code>	the dissimilarity function $d = d(x, p)$ employed.
<code>e</code>	the exponent e employed.

References

- J. C. Bezdek (1981). *Pattern recognition with fuzzy objective function algorithms*. New York: Plenum.
- W. Gaul and M. Schader (1988). Clusterwise aggregation of relations. *Applied Stochastic Models and Data Analysis*, 4:273–282.

Examples

```
## Use a precomputed ensemble of 50 k-means partitions of the
## Cassini data.
data("CKME")
CKME <- CKME[1 : 30] # for saving precious time ...
diss <- cl_dissimilarity(CKME)
hc <- hclust(diss)
plot(hc)
## This suggests using a partition with three classes, which can be
```

```
## obtained using cutree(hc, 3). Could use cl_consensus() to compute
## prototypes as the least squares consensus clusterings of the classes,
## or alternatively:
set.seed(123)
x1 <- cl_pclust(CKME, 3, m = 1)
x2 <- cl_pclust(CKME, 3, m = 2)
## Agreement of solutions.
cl_dissimilarity(x1, x2)
table(cl_class_ids(x1), cl_class_ids(x2))
```

cl_predict

Predict Memberships

Description

Predict class ids or memberships from R objects representing partitions.

Usage

```
cl_predict(object, newdata = NULL,
           type = c("class_ids", "memberships"), ...)
```

Arguments

object	an R object representing a partition of objects.
newdata	an optional data set giving the objects to make predictions for. This must be of the same “kind” as the data set employed for obtaining the partition. If omitted, the original data are used.
type	a character string indicating whether class ids or memberships should be returned. May be abbreviated.
...	arguments to be passed to and from methods.

Details

Many algorithms resulting in partitions of a given set of objects can be taken to induce a partition of the underlying feature space for the measurements on the objects, so that class memberships for “new” objects can be obtained from the induced partition. Examples include partitions based on assigning objects to their “closest” prototypes, or providing mixture models for the distribution of objects in feature space.

This is a generic function. The methods provided in package **clue** handle the partitions obtained from clustering functions in the base R distribution, as well as packages **RWeka**, **cba**, **cclust**, **cluster**, **e1071**, **flexclust**, **flexmix**, **kernlab**, and **mclust** (and of course, **clue** itself).

Value

Depending on `type`, an object of class `"cl_class_ids"` with the predicted class ids, or of class `"cl_membership"` with the matrix of predicted membership values.

Examples

```
## Run kmeans on a random subset of the Cassini data, and predict the
## memberships for the "test" data set.
data("Cassini")
nr <- NROW(Cassini$x)
ind <- sample(nr, 0.9 * nr, replace = FALSE)
party <- kmeans(Cassini$x[ind, ], 3)
table(cl_predict(party, Cassini$x[-ind, ]),
      Cassini$classes[-ind])
```

cl_prototypes

Partition Prototypes

Description

Determine prototypes for the classes of an R object representing a partition.

Usage

```
cl_prototypes(x)
```

Arguments

`x` an R object representing a partition of objects.

Details

Many partitioning methods are based on prototypes (“centers”, “centroids”, “medoids”, ...). In typical cases, these are points in the feature space for the measurements on the objects to be partitioned, such that one can quantify the distance between the objects and the prototypes, and, e.g., classify objects to their closest prototype.

This is a generic function. The methods provided in package **clue** handle the partitions obtained from clustering functions in the base R distribution, as well as packages **cba**, **cclust**, **cluster**, **e1071**, **flexclust**, **kernlab**, and **mclust** (and of course, **clue** itself).

Examples

```
## Show how prototypes ("centers") vary across k-means runs on
## bootstrap samples from the Cassini data.
data("Cassini")
nr <- NROW(Cassini$x)
out <- replicate(50,
                { kmeans(Cassini$x[sample(nr, replace = TRUE), ], 3) },
                simplify = FALSE)
## Plot the data points in light gray, and the prototypes found.
plot(Cassini$x, col = gray(0.8))
points(do.call("rbind", lapply(out, cl_prototypes)), pch = 19)
```

cl_tabulate *Tabulate Vector Objects*

Description

Tabulate the unique values in vector objects.

Usage

```
cl_tabulate(x)
```

Arguments

x a vector.

Value

A data frame with components:

values the unique values.
counts an integer vector with the number of times each of the unique values occurs in x.

Examples

```
data("Kinship82")  
tab <- cl_tabulate(Kinship82)  
## The counts:  
tab$counts  
## The most frequent partition:  
tab$values[[which.max(tab$counts)]]
```

cl_ultrametric *Ultrametrics of Hierarchies*

Description

Compute the ultrametric distances for objects representing (total indexed) hierarchies.

Usage

```
cl_ultrametric(x, size = NULL, labels = NULL)  
as.cl_ultrametric(x)
```

Arguments

x	an R object representing a (total indexed) hierarchy of objects.
size	an integer giving the number of objects in the hierarchy.
labels	a character vector giving the names of the objects in the hierarchy.

Details

If `x` is not an ultrametric or a hierarchy with an ultrametric representation, `cl_ultrametric` uses `cophenetic` to obtain the ultrametric (also known as cophenetic) distances from the hierarchy, which in turn by default calls the S3 generic `as.hclust` on the hierarchy. Support for a class which represents hierarchies can thus be added by providing `as.hclust` methods for this class. In R 2.1.0 or better, `cophenetic` is an S3 generic as well, and one can also more directly provide methods for this if necessary.

`as.cl_ultrametric` is a generic function which can be used for coercing *raw* (non-classed) ultrametries, represented as numeric vectors (of the lower-half entries) or numeric matrices, to ultrametric objects.

Ultrametric objects are implemented as symmetric proximity objects with a dissimilarity interpretation so that self-proximities are zero, and inherit from classes "`cl_dissimilarity`" and "`cl_proximity`". See section **Details** in the documentation for `cl_dissimilarity` for implications.

Ultrametric objects can also be coerced to classes "`dendrogram`" and "`hclust`", and hence in particular use the `plot` methods for these classes. By default, plotting an ultrametric object uses the `plot` method for dendrograms.

Value

An object of class "`cl_ultrametric`" containing the ultrametric distances.

See Also

[is.cl_hierarchy](#)

Examples

```
hc <- hclust(dist(USArrests))
u <- cl_ultrametric(hc)
## Subscripting.
u[1 : 5, 1 : 5]
u[1 : 5, 6 : 7]
## Plotting.
plot(u)
```

Description

Compute validity measures for partitions and hierarchies, attempting to measure how well these clusterings capture the underlying structure in the data they were obtained from.

Usage

```
cl_validity(x, ...)
## Default S3 method:
cl\_validity(x, d, ...)
```

Arguments

`x` an object representing a partition or hierarchy.
`d` a dissimilarity object from which `x` was obtained.
`...` arguments to be passed to or from methods.

Details

`cl_validity` is a generic function.

For partitions, its default method gives the “dissimilarity accounted for”, defined as $1 - a_w/a_t$, where a_t is the average total dissimilarity, and the “average within dissimilarity” a_w is given by

$$\frac{\sum_{i,j} \sum_k m_{ik} m_{jk} d_{ij}}{\sum_{i,j} \sum_k m_{ik} m_{jk}}$$

where d and m are the dissimilarities and memberships, respectively, and the sums are over all pairs of objects and all classes.

For hierarchies, the validity measures computed by default are “variance accounted for” (VAF, e.g., Hubert, Arabie & Meulman, 2006) and “deviance accounted for” (DEV, e.g., Smith, 2001). If u is the ultrametric corresponding to the hierarchy x and d the dissimilarity x was obtained from, these validity measures are given by

$$\text{VAF} = \max \left(0, 1 - \frac{\sum_{i,j} (d_{ij} - u_{ij})^2}{\sum_{i,j} (d_{ij} - \text{mean}(d))^2} \right)$$

and

$$\text{DEV} = \max \left(0, 1 - \frac{\sum_{i,j} |d_{ij} - u_{ij}|}{\sum_{i,j} |d_{ij} - \text{median}(d)|} \right)$$

respectively. Note that VAF and DEV are not invariant under rescaling u , and may be “arbitrarily small” (i.e., 0 using the above definitions) even though u and d are “structurally close” in some sense.

For the results of using [agnes](#) and [diana](#), the agglomerative and divisive coefficients are provided in addition to the default ones.

Value

A list of class "cl_validity" with the computed validity measures.

References

L. Hubert, P. Arabie and J. Meulman (2006). *The structural representation of proximity matrices with MATLAB*. Philadelphia, PA: SIAM.

T. J. Smith (2001). Constructing ultrametric and additive trees based on the L_1 norm. *Journal of Classification*, **18/2**, 185–207.

See Also

`cluster.stats` in package **fpc** for a variety of cluster validation statistics; `fclustIndex` in package **e1071** for several fuzzy cluster indexes; `clustIndex` in package **cclust**; `silhouette` in package **cluster**.

```
fit_ultrametric_target
```

Fit Dissimilarities to a Hierarchy

Description

Find the ultrametric from a target equivalence class of hierarchies which minimizes weighted Euclidean or Manhattan dissimilarity to a given dissimilarity object.

Usage

```
ls_fit_ultrametric_target(x, y, weights = 1)
ll_fit_ultrametric_target(x, y, weights = 1)
```

Arguments

<code>x</code>	a dissimilarity object inheriting from class "dist".
<code>y</code>	a target hierarchy.
<code>weights</code>	a numeric vector or matrix with non-negative weights for obtaining a weighted fit. If a matrix, its numbers of rows and columns must be the same as the number of objects in <code>x</code> . Otherwise, it is recycled to the number of elements in <code>x</code> .

Details

The target equivalence class consists of all dendrograms for which the corresponding n -trees are the same as the one corresponding to `y`. I.e., all splits are the same as for `y`, and optimization is over the height of the splits.

The criterion function to be optimized over all ultrametries from the equivalence class is $\sum w_{ij} |x_{ij} - u_{ij}|^p$, where $p = 2$ in the Euclidean and $p = 1$ in the Manhattan case, respectively.

The optimum can be computed as follows. Suppose split s joins object classes A and B . As the ultrametric dissimilarities of all objects in A to all objects in B must be the same value, say, $u_{A,B} = u_s$, the contribution from the split to the criterion function is of the form $f_s(u_s) = \sum_{i \in A, j \in B} w_{ij} |x_{ij} - u_s|^p$. We need to minimize $\sum_s f_s(u_s)$ under the constraint that the u_s form a non-decreasing sequence, which is accomplished by using the Pool Adjacent Violator Algorithm (PAVA) using the weighted mean ($p = 2$) or weighted median ($p = 1$) for solving the blockwise optimization problems.

Value

An object of class "`cl_ultrametric`" containing the optimal ultrametric distances.

See Also

`ls_fit_ultrametric` for finding the ultrametric minimizing Euclidean dissimilarity (without fixing the splits).

Examples

```
data("Phonemes")
## Note that the Phonemes data set has the consonant misclassification
## probabilities, i.e., the similarities between the phonemes.
d <- as.dist(1 - Phonemes)
## Find the maximal dominated and minimal dominating ultrametries by
## hclust() with single and complete linkage:
y1 <- hclust(d, "single")
y2 <- hclust(d, "complete")
## Note that these are quite different:
cl_dissimilarity(y1, y2, "gamma")
## Now find the L2 optimal members of the respective dendrogram
## equivalence classes.
u1 <- ls_fit_ultrametric_target(d, y1)
u2 <- ls_fit_ultrametric_target(d, y2)
## Compute the L2 optimal ultrametric approximation to d.
u <- ls_fit_ultrametric(d)
## And compare ...
cl_dissimilarity(cl_ensemble(Opt = u, Single = u1, Complete = u2), d)
## The solution obtained via complete linkage is quite close:
cl_agreement(u2, u, "cophenetic")
```

Description

Soft partitions of 21 countries based on macroeconomic data for the years 1975, 1980, 1985, 1990, and 1995.

Usage

```
data("GVME")
```

Format

A named cluster ensemble of 5 soft partitions of 21 countries into 2 or 3 classes. The names are the years to which the partitions correspond.

Details

The partitions were obtained using fuzzy *c*-means on measurements of the following variables: the annual per capita gross domestic product (GDP) in USD (converted to 1987 prices); the percentage of GDP provided by agriculture; the percentage of employees who worked in agriculture; and gross domestic investment, expressed as a percentage of the GDP. See Gordon and Vichi (2001), page 230, for more details.

Source

Table 1 in Gordon and Vichi (2001).

References

A. D. Gordon and M. Vichi (2001). Fuzzy partition models for fitting a set of partitions. *Psychometrika*, **66**, 229–248.

GVME_Consensus

Gordon-Vichi Macroeconomic Consensus Partition Data

Description

The soft (“fuzzy”) consensus partitions for the macroeconomic partition data given in Gordon and Vichi (2001).

Usage

```
data("GVME_Consensus")
```

Format

A named cluster ensemble of eight soft partitions of 21 countries terms into two or three classes.

Details

The elements of the ensemble are consensus partitions for the macroeconomic partition data in Gordon and Vichi (2001), which are available as data set [GVME](#). Element names are of the form "*m/k*", where *m* indicates the consensus method employed (one of ‘MF1’, ‘MF2’, ‘JMF’, and ‘S&S’, corresponding to the application of models 1, 2, and 3 in Gordon and Vichi (2001) and the approach in Sato and Sato (1994), respectively), and *k* denotes the number classes (2 or 3).

Source

Tables 4 and 5 in Gordon and Vichi (2001).

References

A. D. Gordon and M. Vichi (2001). Fuzzy partition models for fitting a set of partitions. *Psychometrika*, **66**, 229–248.

M. Sato and Y. Sato (1994). On a multicriteria fuzzy clustering method for 3-way data. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, **2**, 127–142.

Examples

```
## Load the consensus partitions.
data("GVME_Consensus")
## Pick the partitions into 2 classes.
GVME_Consensus_2 <- GVME_Consensus[1 : 4]
## Fuzziness using the Partition Coefficient.
cl_fuzziness(GVME_Consensus_2)
## (Corresponds to 1 - F in the source.)
## Dissimilarities:
cl_dissimilarity(GVME_Consensus_2)
cl_dissimilarity(GVME_Consensus_2, method = "comem")
```

hierarchy

Hierarchies

Description

Determine whether an R object represents a hierarchy of objects, or coerce to an R object representing such.

Usage

```
is.cl_hierarchy(x)
is.cl_dendrogram(x)
```

```
as.cl_hierarchy(x)
as.cl_dendrogram(x)
```

Arguments

x an R object.

Details

These functions are generic functions.

The methods provided in package **clue** handle the partitions and hierarchies obtained from clustering functions in the base R distribution, as well as packages **RWeka**, **ape**, **cba**, **cclust**, **cluster**, **e1071**, **flexclust**, **flexmix**, **kernlab**, and **mclust** (and of course, **clue** itself).

The hierarchies considered by **clue** are *n-trees* (hierarchies in the strict sense) and *dendrograms* (also known as valued *n-trees* or total indexed hierarchies), which are represented by the virtual classes "cl_hierarchy" and "cl_dendrogram" (which inherits from the former), respectively.

n-trees on a set X of objects correspond to collections H of subsets of X , usually called *classes* of the hierarchy, which satisfy the following properties:

- H contains all singletons with objects of X , X itself, but not the empty set;
- The intersection of two sets A and B in H is either empty or one of the sets.

The classes of a hierarchy can be obtained by `cl_classes`.

Dendrograms are *n-trees* where additionally a height h is associated with each of the classes, so that for two classes A and B with non-empty intersection we have $h(A) \leq h(B)$ iff A is a subset of B . For each pair of objects one can then define u_{ij} as the height of the smallest class containing both i and j : this results in a dissimilarity on X which satisfies the ultrametric (3-point) conditions $u_{ij} \leq \max(u_{ik}, u_{jk})$ for all triples (i, j, k) of objects. Conversely, an ultrametric dissimilarity induces a unique dendrogram.

The ultrametric dissimilarities of a dendrogram can be obtained by `cl_ultrametric`.

`as.cl_hierarchy` returns an object of class "cl_hierarchy" "containing" the given object x if this already represents a hierarchy (i.e., `is.cl_hierarchy(x)` is true), or the ultrametric obtained from x via `as.cl_ultrametric`.

`as.cl_dendrogram` returns an object which has class "cl_dendrogram" and inherits from "cl_hierarchy", and contains x if it represents a dendrogram (i.e., `is.cl_dendrogram(x)` is true), or the ultrametric obtained from x .

Conceptually, hierarchies and dendrograms are *virtual* classes, allowing for a variety of representations.

There are group methods for comparing dendrograms and computing their minimum, maximum, and range based on the meet and join operations, see `cl_meet`. There is also a `plot` method.

Value

For the testing functions, a logical indicating whether the given object represents a clustering of objects of the respective kind.

For the coercion functions, a container object inheriting from "cl_hierarchy", with a suitable representation of the hierarchy given by x .

Examples

```
hcl <- hclust(dist(USArrests))
is.cl_dendrogram(hcl)
is.cl_hierarchy(hcl)
```

Description

Partitions of 15 kinship terms given by 85 female undergraduates at Rutgers University who were asked to sort the terms into classes “on the basis of some aspect of meaning”.

Usage

```
data("Kinship82")
```

Format

A cluster ensemble of 85 hard partitions of the 15 kinship terms.

Details

Rosenberg and Kim (1975) describe an experiment where perceived similarities of the kinship terms were obtained from six different “sorting” experiments. These “original” Rosenberg-Kim kinship terms data were published in Arabie, Carroll and de Sarbo (1987), and are also contained in file ‘indclus.data’ in the shell archive <http://www.netlib.org/mds/indclus.shar>.

For one of the experiments, partitions of the terms were printed in Rosenberg (1982). Comparison with the original data indicates that the partition data have the “nephew” and “niece” columns interchanged, which is corrected in the data set at hand.

Source

Table 7.1 in Rosenberg (1982), with the “nephew” and “niece” columns interchanged.

References

- P. Arabie, J. D. Carroll and W. S. de Sarbo (1987). *Three-way scaling and clustering*. Newbury Park, CA: Sage.
- S. Rosenberg and M. P. Kim (1975). The method of sorting as a data-gathering procedure in multivariate research. *Multivariate Behavioral Research*, **10**, 489–502.
- S. Rosenberg (1982). The method of sorting in multivariate research with applications selected from cognitive psychology and person perception. In N. Hirschberg and L. G. Humphreys (eds.), *Multivariate Applications in the Social Sciences*, 117–142. Hillsdale, NJ: Erlbaum.

Kinship82_Consensus

Gordon-Vichi Kinship82 Consensus Partition Data

Description

The soft (“fuzzy”) consensus partitions for the Rosenberg-Kim kinship terms partition data given in Gordon and Vichi (2001).

Usage

```
data("Kinship82_Consensus")
```

Format

A named cluster ensemble of three soft partitions of the 15 kinship terms into three classes.

Details

The elements of the ensemble are named "MF1", "MF2", and "JMF", and correspond to the consensus partitions obtained by applying models 1, 2, and 3 in Gordon and Vichi (2001) to the kinship terms partition data in Rosenberg (1982), which are available as data set [Kinship82](#).

Source

Table 6 in Gordon and Vichi (2001).

References

A. D. Gordon and M. Vichi (2001). Fuzzy partition models for fitting a set of partitions. *Psychometrika*, **66**, 229–248.

S. Rosenberg (1982). The method of sorting in multivariate research with applications selected from cognitive psychology and person perception. In N. Hirschberg and L. G. Humphreys (eds.), *Multivariate Applications in the Social Sciences*, 117–142. Hillsdale, NJ: Erlbaum.

Examples

```
## Load the consensus partitions.
data("Kinship82_Consensus")
## Fuzziness using the Partition Coefficient.
cl_fuzziness(Kinship82_Consensus)
## (Corresponds to 1 - F in the source.)
## Dissimilarities:
cl_dissimilarity(Kinship82_Consensus)
cl_dissimilarity(Kinship82_Consensus, method = "comem")
```

`kmedoids`*K-Medoids Clustering*

Description

Compute a k -medoids partition of a dissimilarity object.

Usage

```
kmedoids(x, k)
```

Arguments

<code>x</code>	a dissimilarity object inheriting from class " <code>dist</code> ", or a square matrix of pairwise object-to-object dissimilarity values.
<code>k</code>	an integer giving the number of classes to be used in the partition.

Details

Let d denote the pairwise object-to-object dissimilarity matrix corresponding to x . A k -medoids partition of x is defined as a partition of the numbers from 1 to n , the number of objects in x , into k classes C_1, \dots, C_k such that the criterion function $L = \sum_l \min_{j \in C_l} \sum_{i \in C_l} d_{ij}$ is minimized.

This is an NP-hard optimization problem. PAM (Partitioning Around Medoids, see Kaufman & Rousseeuw (1990), Chapter 2) is a very popular heuristic for obtaining optimal k -medoids partitions, and provided by `pam` in package `cluster`.

`kmedoids` is an exact algorithm based on a binary linear programming formulation of the optimization problem (e.g., Gordon & Vichi (1998), [P4']), using `lp` from package `lpSolve` as solver. Depending on available hardware resources (the number of constraints of the program is of the order n^2), it may not be possible to obtain a solution.

Value

An object of class "`kmedoids`" representing the obtained partition, which is a list with the following components.

<code>cluster</code>	the class ids of the partition.
<code>medoid_ids</code>	the indices of the medoids.
<code>criterion</code>	the value of the criterion function of the partition.

References

- L. Kaufman and P. J. Rousseeuw (1990). *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley, New York.
- A. D. Gordon and M. Vichi (1998). Partitions of partitions. *Journal of Classification*, **15**, 265–285.

l1_fit_ultrametric *Least Absolute Deviation Fit of Ultrametrics to Dissimilarities*

Description

Find the ultrametric with minimal absolute distance (Manhattan dissimilarity) to a given dissimilarity object.

Usage

```
l1_fit_ultrametric(x, method = c("SUMT", "IRIP"), weights = 1,
                  control = list())
```

Arguments

x	a dissimilarity object inheriting from or coercible to class "dist".
method	a character string indicating the fitting method to be employed. Must be one of "SUMT" (default) or "IRIP", or a unique abbreviation thereof.
weights	a numeric vector or matrix with non-negative weights for obtaining a weighted least squares fit. If a matrix, its numbers of rows and columns must be the same as the number of objects in x, and the lower diagonal part is used. Otherwise, it is recycled to the number of elements in x.
control	a list of control parameters. See Details .

Details

The problem to be solved is minimizing

$$L(u) = \sum_{i,j} w_{ij} |x_{ij} - u_{ij}|$$

over all u satisfying the ultrametric constraints (i.e., for all i, j, k , $u_{ij} \leq \max(u_{ik}, u_{jk})$). This problem is known to be NP hard (Krivanek and Moravek, 1986).

We provide two heuristics for solving this problem.

Method "SUMT" implements a SUMT (Sequential Unconstrained Minimization Technique, see [sumt](#)) approach using the sign function for the gradients of the absolute value function.

Available control parameters are `method`, `control`, `eps`, `q`, and `verbose`, which have the same roles as for [sumt](#), and the following.

`nruns` an integer giving the number of runs to be performed. Defaults to 1.

`start` a single dissimilarity, or a list of dissimilarities to be employed as starting values.

Method "IRIP" implements a variant of the Iteratively Reweighted Iterative Projection approach of Smith (2001), which attempts to solve the L_1 problem via a sequence of weighted L_2 problems, determining $u(t+1)$ by minimizing the criterion function

$$\sum_{i,j} w_{ij} (x_{ij} - u_{ij})^2 / \max(|x_{ij} - u_{ij}(t)|, m)$$

with m a “small” non-zero value to avoid zero divisors. We use the SUMT method of `ls_fit_ultrametric` for solving the weighted least squares problems.

Available control parameters are as follows.

`maxiter` an integer giving the maximal number of iteration steps to be performed. Defaults to 100.

`eps` a nonnegative number controlling the iteration, which stops when the maximal change in u is less than `eps`. Defaults to 10^{-6} .

`reltol` the relative convergence tolerance. Iteration stops when the relative change in the L_1 criterion is less than `reltol`. Defaults to 10^{-6} .

`MIN` the cutoff m . Defaults to 10^{-3} .

`start` a dissimilarity object to be used as the starting value for u .

`control` a list of control parameters to be used by the method of `ls_fit_ultrametric` employed for solving the weighted L_2 problems.

One may need to adjust the default control parameters to achieve convergence.

It should be noted that all methods are heuristics which can not be guaranteed to find the global minimum.

Value

An object of class "`cl_ultrametric`" containing the fitted ultrametric distances.

References

M. Krivanek and J. Moravek (1986). NP-hard problems in hierarchical tree clustering. *Acta Informatica*, **23**, 311–323.

T. J. Smith (2001). Constructing ultrametric and additive trees based on the L_1 norm. *Journal of Classification*, **18**, 185–207.

See Also

`cl_consensus` for computing least absolute deviation (Manhattan) consensus hierarchies; `ls_fit_ultrametric`.

lattice

Cluster Lattices

Description

Computations on the lattice of all (hard) partitions, or the lattice of all dendrograms, or the meet semilattice of all hierarchies (n -trees) of/on a set of objects: meet, join, and comparisons.

Usage

`cl_meet(x, y)`

`cl_join(x, y)`

Arguments

<code>x</code>	an ensemble of partitions or dendrograms or hierarchies, or an R object representing a partition or dendrogram or hierarchy.
<code>y</code>	an R object representing a partition or dendrogram or hierarchy. Ignored if <code>x</code> is an ensemble.

Details

For a given finite set of objects X , the set $H(X)$ of all (hard) partitions of X can be partially ordered by defining a partition P to be “finer” than a partition Q , i.e., $P \leq Q$, if each class of P is contained in some class of Q . With this partial order, $H(X)$ becomes a bounded *lattice*, with intersection and union of two elements given by their greatest lower bound (*meet*) and their least upper bound (*join*), respectively.

Specifically, the meet of two partitions computed by `cl_meet` is the partition obtained by intersecting the classes of the partitions; the classes of the join computed by `cl_join` are obtained by joining all elements in the same class in at least one of the partitions. Obviously, the least and greatest elements of the partition lattice are the partitions where each object is in a single class (sometimes referred to as the “splitter” partition) or in the same class (the “lumper” partition), respectively. Meet and join of an arbitrary number of partitions can be defined recursively.

In addition to computing the meet and join, the comparison operations corresponding to the above partial order as well as `min`, `max`, and `range` are available at least for R objects representing partitions inheriting from "`cl_partition`". The summary methods give the meet and join of the given partitions (for `min` and `max`), or a partition ensemble with the meet and join (for `range`).

If the partitions specified by `x` and `y` are soft partitions, the corresponding nearest hard partitions are used. Future versions may optionally provide suitable “soft” (fuzzy) extensions for computing meets and joins.

The set of all dendrograms on X can be ordered using pointwise inequality of the associated ultrametric dissimilarities: i.e., if D and E are the dendrograms with ultrametrics u and v , respectively, then $D \leq E$ if $u_{ij} \leq v_{ij}$ for all pairs (i, j) of objects. This again yields a lattice (of dendrograms). The join of D and E is the dendrogram with ultrametrics given by $\max(u_{ij}, v_{ij})$ (as this gives an ultrametric); the meet is the dendrogram with the maximal ultrametric dominated by $\min(u_{ij}, v_{ij})$, and can be obtained by applying single linkage hierarchical clustering to the minima.

The set of all hierarchies on X can be ordered by set-wise inclusion of the classes: i.e., if H and G are two hierarchies, then $H \leq G$ if all classes of H are also classes of G . This yields a meet semilattice, with meet given by the classes contained in both hierarchies. The join only exists if the union of the classes is a hierarchy.

In each case, a modular semilattice is obtained, which allows for a natural metrization via least element (semi)lattice move distances, see Barthélmy, Leclerc and Monjardet (1981). These latticial metrics are given by the BA/C (partitions), Manhattan (dendrograms), and symdiff (hierarchies) dissimilarities, respectively (see `cl_dissimilarity`).

Value

For `cl_meet` and `cl_join`, an object of class "`cl_partition`" or "`cl_dendrogram`" with the class ids or ultrametric dissimilarities of the meet and join of the partitions or dendrograms, respectively.

References

J.-P. Barthélemy, B. Leclerc and B. Monjardet (1981). On the use of ordered sets in problems of comparison and consensus of classification. *Journal of Classification*, **3**, 187–224.

Examples

```
## Two simple partitions of 7 objects.
A <- as.cl_partition(c(1, 1, 2, 3, 3, 5, 5))
B <- as.cl_partition(c(1, 2, 2, 3, 4, 5, 5))
## These disagree on objects 1-3, A splits objects 4 and 5 into
## separate classes. Objects 6 and 7 are always in the same class.
(A <= B) || (B <= A)
## (Neither partition is finer than the other.)
cl_meet(A, B)
cl_join(A, B)
## Meeting with the lumpner (greatest) or joining with the splitter
## (least) partition does not make a difference:
C_lumper <- as.cl_partition(rep(1, n_of_objects(A)))
cl_meet(cl_ensemble(A, B, C_lumper))
C_splitter <- as.cl_partition(seq(length = n_of_objects(A)))
cl_join(cl_ensemble(A, B, C_splitter))
## Another way of computing the join:
range(A, B, C_splitter)$max
```

```
ls_fit_sum_of_ultrametrics
```

Least Squares Fit of Sums of Ultrametrics to Dissimilarities

Description

Find a sequence of ultrametrics with sum minimizing square distance (Euclidean dissimilarity) to a given dissimilarity object.

Usage

```
ls_fit_sum_of_ultrametrics(x, nterms = 1, weights = 1, control = list())
```

Arguments

<code>x</code>	a dissimilarity object inheriting from or coercible to class " <code>dist</code> ".
<code>nterms</code>	an integer giving the number of ultrametrics to be fitted.
<code>weights</code>	a numeric vector or matrix with non-negative weights for obtaining a weighted least squares fit. If a matrix, its numbers of rows and columns must be the same as the number of objects in <code>x</code> , and the lower diagonal part is used. Otherwise, it is recycled to the number of elements in <code>x</code> .
<code>control</code>	a list of control parameters. See Details .

Details

The problem to be solved is minimizing the criterion function

$$L(u(1), \dots, u(n)) = \sum_{i,j} w_{ij} \left(x_{ij} - \sum_{k=1}^n u_{ij}(k) \right)^2$$

over all $u(1), \dots, u(n)$ satisfying the ultrametric constraints.

We provide an implementation of the iterative heuristic suggested in Carroll & Pruzansky (1980) which in each step t sequentially refits the $u(k)$ as the least squares ultrametric fit to the “residuals” $x - \sum_{l \neq k} u(l)$ using `ls_fit_ultrametric`.

Available control parameters include

`maxiter` an integer giving the maximal number of iteration steps to be performed. Defaults to 100.

`eps` a nonnegative number controlling the iteration, which stops when the maximal change in all $u(k)$ is less than `eps`. Defaults to 10^{-6} .

`reltol` the relative convergence tolerance. Iteration stops when the relative change in the criterion function is less than `reltol`. Defaults to 10^{-6} .

`method` a character string indicating the fitting method to be employed by the individual least squares fits.

`control` a list of control parameters to be used by the method of `ls_fit_ultrametric` employed. By default, if the SUMT method `method` is used, 10 inner SUMT runs are performed for each refitting.

It should be noted that the method used is a heuristic which can not be guaranteed to find the global minimum.

Value

A list of objects of class "`cl_ultrametric`" containing the fitted ultrametric distances.

References

J. D. Carroll and S. Pruzansky (1980). Discrete and hybrid scaling models. In E. D. Lantermann and H. Feger (eds.), *Similarity and Choice*. Bern (Switzerland): Huber.

`ls_fit_ultrametric` *Least Squares Fit of Ultrametrics to Dissimilarities*

Description

Find the ultrametric with minimal square distance (Euclidean dissimilarity) to given dissimilarity objects.

Usage

```
ls_fit_ultrametric(x, method = c("SUMT", "IP", "IR"), weights = 1,
                  control = list())
```

Arguments

<code>x</code>	a dissimilarity object inheriting from or coercible to class " <code>dist</code> ", or an ensemble of such objects.
<code>method</code>	a character string indicating the fitting method to be employed. Must be one of "SUMT" (default), "IP", or "IR", or a unique abbreviation thereof.
<code>weights</code>	a numeric vector or matrix with non-negative weights for obtaining a weighted least squares fit. If a matrix, its numbers of rows and columns must be the same as the number of objects in <code>x</code> , and the lower diagonal part is used. Otherwise, it is recycled to the number of elements in <code>x</code> .
<code>control</code>	a list of control parameters. See Details .

Details

For a single dissimilarity object `x`, the problem to be solved is minimizing

$$L(u) = \sum_{i,j} w_{ij} (x_{ij} - u_{ij})^2$$

over all u satisfying the ultrametric constraints (i.e., for all i, j, k , $u_{ij} \leq \max(u_{ik}, u_{jk})$). This problem is known to be NP hard (Krivanek and Moravek, 1986).

For an ensemble of dissimilarity objects, the criterion function is

$$L(u) = \sum_b w_b \sum_{i,j} w_{ij} (x_{ij}(b) - u_{ij})^2,$$

where w_b is the weight given to element x_b of the ensemble and can be specified via control parameter `weights` (default: all ones). This problem reduces to the above basic problem with x as the w_b -weighted mean of the x_b .

We provide three heuristics for solving the basic problem.

Method "SUMT" implements the SUMT (Sequential Unconstrained Minimization Technique, Fiacco and McCormick, 1968) approach of de Soete (1986) which in turn simplifies the suggestions in Carroll and Pruzansky (1980). (See `sumt` for more information on the SUMT approach.) We then use a final single linkage hierarchical clustering step to ensure that the returned object exactly satisfies the ultrametric constraints. The starting value u_0 is obtained by "random shaking" of the given dissimilarity object (if not given). If there are missing values in `x`, i.e., the given dissimilarities are *incomplete*, we follow a suggestion of de Soete (1984), imputing the missing values by the weighted mean of the non-missing ones, and setting the corresponding weights to zero.

Available control parameters are `method`, `control`, `eps`, `q`, and `verbose`, which have the same roles as for `sumt`, and the following.

`nruns` an integer giving the number of runs to be performed. Defaults to 1.

`start` a single dissimilarity, or a list of dissimilarities to be employed as starting values.

The default optimization using conjugate gradients should work reasonably well for medium to large size problems. For “small” ones, using `nlm` is usually faster. Note that the number of ultrametric constraints is of the order n^3 , where n is the number of objects in the dissimilarity object, suggesting to use the SUMT approach in favor of `constrOptim`.

If starting values for the SUMT are provided via `start`, the number of starting values gives the number of runs to be performed, and control option `nruns` is ignored. Otherwise, `nruns` starting values are obtained by random shaking of the dissimilarity to be fitted. In the case of multiple SUMT runs, the (first) best solution found is returned.

Method "IP" implements the Iterative Projection approach of Hubert and Arabie (1995). This iteratively projects the current dissimilarities to the closed convex set given by the ultrametric constraints (3-point conditions) for a single index triple (i, j, k) , in fact replacing the two largest values among d_{ij}, d_{ik}, d_{jk} by their mean. The following control parameters can be provided via the `control` argument.

`nruns` an integer giving the number of runs to be performed. Defaults to 1.

`order` a permutation of the numbers from 1 to the number of objects in `x`, specifying the order in which the ultrametric constraints are considered, or a list of such permutations.

`maxiter` an integer giving the maximal number of iterations to be employed.

`tol` a double indicating the maximal convergence tolerance. The algorithm stops if the total absolute change in the dissimilarities in an iteration is less than `tol`.

`verbose` a logical indicating whether to provide some output on minimization progress. Defaults to `getOption("verbose")`.

If permutations are provided via `order`, the number of these gives the number of runs to be performed, and control option `nruns` is ignored. Otherwise, `nruns` randomly generated orders are tried. In the case of multiple runs, the (first) best solution found is returned.

Non-identical weights and incomplete dissimilarities are currently not supported.

Method "IR" implements the Iterative Reduction approach suggested by Roux (1988), see also Barthélémy and Guénoche (1991). This is similar to the Iterative Projection method, but modifies the dissimilarities between objects proportionally to the aggregated change incurred from the ultrametric projections. Available control parameters are identical to those of method "IP".

Non-identical weights and incomplete dissimilarities are currently not supported.

It should be noted that all methods are heuristics which can not be guaranteed to find the global minimum. Standard practice would recommend to use the best solution found in “sufficiently many” replications of the base algorithm.

Value

An object of class "`cl_ultrametric`" containing the fitted ultrametric distances.

References

- J.-P. Barthélémy and A. Guénoche (1991). *Trees and proximity representations*. Chichester: John Wiley & Sons. ISBN 0-471-92263-3.
- J. D. Carroll and S. Pruzansky (1980). Discrete and hybrid scaling models. In E. D. Lantermann and H. Feger (eds.), *Similarity and Choice*. Bern (Switzerland): Huber.

- L. Hubert and P. Arabie (1995). Iterative projection strategies for the least squares fitting of tree structures to proximity data. *British Journal of Mathematical and Statistical Psychology*, **48**, 281–317.
- M. Krivanek and J. Moravek (1986). NP-hard problems in hierarchical tree clustering. *Acta Informatica*, **23**, 311–323.
- M. Roux (1988). Techniques of approximation for building two tree structures. In C. Hayashi and E. Diday and M. Jambu and N. Ohsumi (Eds.), *Recent Developments in Clustering and Data Analysis*, pages 151–170. New York: Academic Press.
- G. de Soete (1984). Ultrametric tree representations of incomplete dissimilarity data. *Journal of Classification*, **1**, 235–242.
- G. de Soete (1986). A least squares algorithm for fitting an ultrametric tree to a dissimilarity matrix. *Pattern Recognition Letters*, **2**, 133–137.

See Also

[cl_consensus](#) for computing least squares (Euclidean) consensus hierarchies by least squares fitting of average ultrametric distances; [l1_fit_ultrametric](#).

Examples

```
## Least squares fit of an ultrametric to the Miller-Nicely consonant
## phoneme confusion data.
data("Phonemes")
## Note that the Phonemes data set has the consonant misclassification
## probabilities, i.e., the similarities between the phonemes.
d <- as.dist(1 - Phonemes)
u <- ls_fit_ultrametric(d, control = list(verbose = TRUE))
## Cophenetic correlation:
cor(d, u)
## Plot:
plot(u)
## ("Basically" the same as Figure 1 in de Soete (1986).)
```

n_of_classes

Classes in a Partition

Description

Determine the number of classes and the class ids in a partition of objects.

Usage

```
n_of_classes(x)
cl_class_ids(x)
as.cl_class_ids(x)
```

Arguments

`x` an object representing a (hard or soft) partition (for `n_of_classes` and `cl_class_ids`), or raw class ids (for `as.cl_class_ids`).

Details

These function are generic functions.

The methods provided in package **clue** handle the partitions obtained from clustering functions in the base R distribution, as well as packages **RWeka**, **cba**, **cclust**, **cluster**, **e1071**, **flexclust**, **flexmix**, **kernlab**, and **mclust** (and of course, **clue** itself).

Note that the number of classes is taken as the number of distinct class ids actually used in the partition, and may differ from the number of columns in a membership matrix representing the partition.

`as.cl_class_ids` can be used for coercing “raw” class ids (given as atomic vectors) to class id objects.

Value

For `n_of_classes`, an integer giving the number of classes in the partition.

For `cl_class_ids`, a vector of integers with the corresponding class ids. For soft partitions, the class ids returned are those of the *nearest hard partition* obtained by taking the class ids of the (first) maximal membership values.

See Also

[is.cl_partition](#)

Examples

```
data("Cassini")
party <- kmeans(Cassini$x, 3)
n_of_classes(party)
## A simple confusion matrix:
table(cl_class_ids(party), Cassini$classes)
## For an "oversize" membership matrix representation:
n_of_classes(cl_membership(party, 6))
```

<code>n_of_objects</code>	<i>Number of Objects in a Partition or Hierarchy</i>
---------------------------	------------------------------------------------------

Description

Determine the number of objects from which a partition or hierarchy was obtained.

Usage

```
n_of_objects(x)
```

Arguments

`x` an R object representing a (hard or soft) partition or a hierarchy of objects, or dissimilarities between objects.

Details

This is a generic function.

The methods provided in package **clue** handle the partitions and hierarchies obtained from clustering functions in the base R distribution, as well as packages **RWeka**, **ape**, **cba**, **cclust**, **cluster**, **e1071**, **flexclust**, **flexmix**, **kernlab**, and **mclust** (and of course, **clue** itself).

There is also a method for object dissimilarities which inherit from class "`dist`".

Value

An integer giving the number of objects.

See Also

[is.cl_partition](#), [is.cl_hierarchy](#)

Examples

```
data("Cassini")
pcl <- kmeans(Cassini$x, 3)
n_of_objects(pcl)
hcl <- hclust(dist(USArrests))
n_of_objects(hcl)
```

partition

Partitions

Description

Determine whether an R object represents a partition of objects, or coerce to an R object representing such.

Usage

```
is.cl_partition(x)
is.cl_hard_partition(x)
is.cl_soft_partition(x)

as.cl_partition(x)
as.cl_hard_partition(x)
```

Arguments

`x` an R object.

Details

`is.cl_partition` and `is.cl_hard_partition` are generic functions.

The methods provided in package **clue** handle the partitions obtained from clustering functions in the base R distribution, as well as packages **RWeka**, **cba**, **cclust**, **cluster**, **e1071**, **flexclust**, **flexmix**, **kernlab**, and **mclust** (and of course, **clue** itself).

`is.cl_soft_partition` gives true iff `is.cl_partition` is true and `is.cl_hard_partition` is false.

`as.cl_partition` returns an object of class "cl_partition" "containing" the given object `x` if this already represents a partition (i.e., `is.cl_partition(x)` is true), or the memberships obtained from `x` via `as.cl_membership`.

`as.cl_hard_partition(x)` returns an object which has class "cl_hard_partition" and inherits from "cl_partition", and contains `x` if it already represents a hard partition (i.e., `is.cl_hard_partition(x)` is true), or the class ids obtained from `x`, using `x` if this is an atomic vector of raw class ids, or, if `x` represents a soft partition or is a raw matrix of membership values, using the class ids of the *nearest hard partition*, defined by taking the class ids of the (first) maximal membership values.

Conceptually, partitions and hard partitions are *virtual* classes, allowing for a variety of representations.

There are group methods for comparing partitions and computing their minimum, maximum, and range based on the meet and join operations, see `cl_meet`.

Value

For the testing functions, a logical indicating whether the given object represents a clustering of objects of the respective kind.

For the coercion functions, a container object inheriting from "cl_partition", with a suitable representation of the partition given by `x`.

Examples

```
data("Cassini")
pcl <- kmeans(Cassini$x, 3)
is.cl_partition(pcl)
is.cl_hard_partition(pcl)
is.cl_soft_partition(pcl)
```

pclust

Prototype-Based Partitioning

Description

Obtain prototype-based partitions of objects by minimizing the criterion $\sum w_b u_{bj}^m d(x_b, p_j)^e$, the sum of the case-weighted and membership-weighted e -th powers of the dissimilarities between the objects x_b and the prototypes p_j , for suitable dissimilarities d and exponents e .

Usage

```
pclus(x, k, family, m = 1, weights = 1, control = list())
pclus_family(D, C, init = NULL, description = NULL, e = 1,
             .modify = NULL, .subset = NULL)
pclus_object(prototypes, membership, cluster, family, m = 1, value, ...,
             classes = NULL, attributes = NULL)
```

Arguments

<code>x</code>	the object to be partitioned.
<code>k</code>	an integer giving the number of classes to be used in the partition.
<code>family</code>	an object of class "pclus_family" as generated by <code>pclus_family</code> , containing the information about d and e .
<code>m</code>	a number not less than 1 controlling the softness of the partition (as the “fuzzi-fication parameter” of the fuzzy c -means algorithm). The default value of 1 corresponds to hard partitions obtained from a generalized k -means problem; values greater than one give partitions of increasing softness obtained from a generalized fuzzy c -means problem.
<code>weights</code>	a numeric vector of non-negative case weights. Recycled to the number of elements given by <code>x</code> if necessary.
<code>control</code>	a list of control parameters. See Details .
<code>D</code>	a function for computing dissimilarities d between objects and prototypes.
<code>C</code>	a ‘consensus’ function with formals <code>x</code> , <code>weights</code> and <code>control</code> for computing a consensus prototype p minimizing $\sum_b w_b d(x_b, p)^e$.
<code>init</code>	a function with formals <code>x</code> and <code>k</code> initializing an object with k prototypes from the object <code>x</code> to be partitioned.
<code>description</code>	a character string describing the family.
<code>e</code>	a number giving the exponent e of the criterion.
<code>.modify</code>	a function with formals <code>x</code> , <code>i</code> and <code>value</code> for modifying a single prototype, or NULL (default).
<code>.subset</code>	a function with formals <code>x</code> and <code>i</code> for subsetting prototypes, or NULL (default).
<code>prototypes</code>	an object representing the prototypes of the partition.
<code>membership</code>	an object of class " cl_membership " with the membership values u_{bj} .
<code>cluster</code>	the class ids of the nearest hard partition.
<code>value</code>	the value of the criterion to be minimized.
<code>...</code>	further elements to be included in the generated <code>pclus</code> object.
<code>classes</code>	a character vector giving further classes to be given to the generated <code>pclus</code> object in addition to "pclus", or NULL (default).
<code>attributes</code>	a list of attributes, or NULL (default).

Details

For $m = 1$, a generalization of the Lloyd-Forgy variant of the k -means algorithm is used, which iterates between reclassifying objects to their closest prototypes (according to the dissimilarities given by `D`), and computing new prototypes as the consensus for the classes (using `C`).

For $m > 1$, a generalization of the fuzzy c -means recipe (e.g., Bezdek (1981)) is used, which alternates between computing optimal memberships for fixed prototypes, and computing new prototypes as the suitably weighted consensus clusterings for the classes.

This procedure is repeated until convergence occurs, or the maximal number of iterations is reached.

Currently, no local improvement heuristics are provided.

It is possible to perform several runs of the procedure via control arguments `nruns` or `start` (the default is to perform a single run), in which case the first partition with the smallest value of the criterion is returned.

The dissimilarity and consensus functions as well as the exponent e are specified via `family`. In principle, arbitrary representations of objects to be partitioned and prototypes (which do not necessarily have to be “of the same kind”) can be employed. In addition to `D` and `C`, what is needed are means to obtain an initial collection of k prototypes (`init`), to modify a single prototype (`.modify`), and subset the prototypes (`.subset`). By default, list and (currently, only dense) matrix (with the usual convention that the rows correspond to the objects) are supported. Otherwise, the family has to provide the functions needed.

Available control parameters are as follows.

`maxiter` an integer giving the maximal number of iterations to be performed. Defaults to 100.

`nruns` an integer giving the number of runs to be performed. Defaults to 1.

`reltol` the relative convergence tolerance. Defaults to `sqrt(.Machine$double.eps)`.

`start` a list of prototype objects to be used as starting values.

`verbose` a logical indicating whether to provide some output on minimization progress. Defaults to `getOption("verbose")`.

`control` control parameters to be used in the consensus function.

The fixed point approach employed is a heuristic which cannot be guaranteed to find the global minimum, in particular if `C` is not an exact minimizer. Standard practice would recommend to use the best solution found in “sufficiently many” replications of the base algorithm.

Value

`pclus` returns the partition found as an object of class `"pclus"` (as obtained by calling `pclus_object`) which in addition to the *default* components contains `call` (the matched call) and a `converged` attribute indicating convergence status (i.e., whether the maximal number of iterations was reached).

`pclus_family` returns an object of class `"pclus_family"`, which is a list with components corresponding to the formals of `pclus_family`.

`pclus_object` returns an object inheriting from class `"pclus"`, which is a list with components corresponding to the formals (up to and including `...`) of `pclus_object`, and additional classes and attributes specified by `classes` and `attributes`, respectively.

References

J. C. Bezdek (1981). *Pattern recognition with fuzzy objective function algorithms*. New York: Plenum.

See Also

[kmeans](#), [cmeans](#).

Phonemes

Miller-Nicely Consonant Phoneme Confusion Data

Description

Miller-Nicely data on the auditory confusion of 16 consonant phonemes.

Usage

```
data("Phonemes")
```

Format

A symmetric matrix of the misclassification probabilities of 16 English consonant phonemes.

Details

Miller and Nicely (1955) obtained the confusions by exposing female subjects to a series of syllables consisting of one of the 16 consonants followed by the vowel ‘a’ under 17 different experimental conditions. The data provided are obtained from aggregating the six so-called flat-noise conditions in which only the speech-to-noise ratio was varied into a single matrix of misclassification frequencies.

Source

The data set is also contained in file ‘mapclus.data’ in the shell archive <http://www.netlib.org/mds/mapclus.shar>.

References

G. A. Miller and P. E. Nicely (1955). An analysis of perceptual confusions among some English consonants. *Journal of the Acoustical Society of America*, **27**, 338–352.

 solve_LSAP

 Solve Linear Sum Assignment Problem

Description

Solve the linear sum assignment problem using the Hungarian method.

Usage

```
solve_LSAP(x, maximum = FALSE)
```

Arguments

x	a matrix with nonnegative entries and at least as many columns as rows.
maximum	a logical indicating whether to minimize or maximize the sum of assigned costs.

Details

If nr and nc are the numbers of rows and columns of x , `solve_LSAP` finds an optimal *assignment* of rows to columns, i.e., a one-to-one map p of the numbers from 1 to nr to the numbers from 1 to nc (a permutation of these numbers in case x is a square matrix) such that $\sum_{i=1}^{nr} x[i, p[i]]$ is minimized or maximized.

This assignment can be found using a linear program (and package **lpSolve** provides a function `lp.assign` for doing so), but typically more efficiently and provably in polynomial time $O(n^3)$ using primal-dual methods such as the so-called Hungarian method (see the references).

Value

An object of class "solve_LSAP" with the optimal assignment of rows to columns.

Author(s)

Walter Böhm <Walter.Boehm@wu-wien.ac.at> kindly provided C code implementing the Hungarian method.

References

C. Papadimitriou and K. Steiglitz (1982), *Combinatorial Optimization: Algorithms and Complexity*. Englewood Cliffs: Prentice Hall.

Examples

```
x <- matrix(c(5, 1, 4, 3, 5, 2, 2, 4, 4), nrow = 3)
solve_LSAP(x)
solve_LSAP(x, maximum = TRUE)
## To get the optimal value (for now):
y <- solve_LSAP(x)
sum(x[cbind(seq_along(y), y)])
```

sumt

*Sequential Unconstrained Minimization Technique***Description**

Solve constrained optimization problems via the Sequential Unconstrained Minimization Technique (SUMT).

Usage

```
sumt(x0, L, P, grad_L = NULL, grad_P = NULL, method = NULL,
     eps = NULL, q = NULL, verbose = NULL, control = list())
```

Arguments

<code>x0</code>	a list of starting values, or a single starting value.
<code>L</code>	a function to minimize.
<code>P</code>	a non-negative penalty function such that $P(x)$ is zero iff the constraints are satisfied.
<code>grad_L</code>	a function giving the gradient of <code>L</code> , or <code>NULL</code> (default).
<code>grad_P</code>	a function giving the gradient of <code>P</code> , or <code>NULL</code> (default).
<code>method</code>	a character string, or <code>NULL</code> . If not given, "CG" is used. If equal to "nlm", minimization is carried out using <code>nlm</code> . Otherwise, <code>optim</code> is used with <code>method</code> as the given method.
<code>eps</code>	the absolute convergence tolerance. The algorithm stops if the (maximum) distance between successive <code>x</code> values is less than <code>eps</code> . Defaults to <code>sqrt(.Machine\$double.eps)</code> .
<code>q</code>	a double greater than one controlling the growth of the ρ_k as described in Details . Defaults to 10.
<code>verbose</code>	a logical indicating whether to provide some output on minimization progress. Defaults to <code>getOption("verbose")</code> .
<code>control</code>	a list of control parameters to be passed to the minimization routine in case <code>optim</code> is used.

Details

The Sequential Unconstrained Minimization Technique is a heuristic for constrained optimization. To minimize a function L subject to constraints, one employs a non-negative function P penalizing violations of the constraints, such that $P(x)$ is zero iff x satisfies the constraints. One iteratively minimizes $L(x) + \rho_k P(x)$, where the ρ values are increased according to the rule $\rho_{k+1} = q\rho_k$ for some constant $q > 1$, until convergence is obtained in the sense that the Euclidean distance between successive solutions x_k and x_{k+1} is small enough. Note that the “solution” x obtained does not necessarily satisfy the constraints, i.e., has zero $P(x)$. Note also that there is no guarantee that global (approximately) constrained optima are found. Standard practice would recommend to use the best solution found in “sufficiently many” replications of the algorithm.

The unconstrained minimizations are carried out by either `optim` or `nlm`, using analytic gradients if both `grad_L` and `grad_P` are given, and numeric ones otherwise.

If more than one starting value is given, the solution with the minimal augmented criterion function value is returned.

Value

A list inheriting from class "sumt", with components `x`, `L`, `P`, and `rho` giving the solution obtained, the value of the criterion and penalty function at `x`, and the final ρ value used in the augmented criterion function.

References

A. V. Fiacco and G. P. McCormick (1968). *Nonlinear programming: Sequential unconstrained minimization techniques*. New York: John Wiley & Sons.

Index

*Topic **cluster**

- addtree, 2
- cl_agreement, 5
- cl_bag, 8
- cl_boot, 9
- cl_classes, 10
- cl_consensus, 11
- cl_dissimilarity, 15
- cl_ensemble, 20
- cl_fuzziness, 21
- cl_margin, 23
- cl_medoid, 23
- cl_membership, 24
- cl_object_names, 26
- cl_pam, 27
- cl_pclust, 28
- cl_predict, 30
- cl_prototypes, 31
- cl_ultrametric, 32
- cl_validity, 34
- fit_ultrametric_target, 35
- hierarchy, 38
- kmedoids, 42
- ll_fit_ultrametric, 43
- lattice, 44
- ls_fit_sum_of_ultrametrics, 46
- ls_fit_ultrametric, 47
- n_of_classes, 50
- n_of_objects, 51
- partition, 52

*Topic **datasets**

- Cassini, 3
- CKME, 4
- GVME, 36
- GVME_Consensus, 37
- Kinship82, 40
- Kinship82_Consensus, 41
- Phonemes, 56

*Topic **optimize**

- addtree, 2
- fit_ultrametric_target, 35
- kmedoids, 42
- ll_fit_ultrametric, 43
- ls_fit_sum_of_ultrametrics, 46
- ls_fit_ultrametric, 47
- solve_LSAP, 57
- sumt, 58

*Topic **utilities**

- cl_tabulate, 32

- addtree, 2
- agnes, 34
- as.cl_class_ids(*n_of_classes*), 50
- as.cl_dendrogram(*hierarchy*), 38
- as.cl_ensemble(*cl_ensemble*), 20
- as.cl_hard_partition(*partition*), 52
- as.cl_hierarchy(*hierarchy*), 38
- as.cl_membership, 53
- as.cl_membership(*cl_membership*), 24
- as.cl_partition(*partition*), 52
- as.cl_ultrametric, 39
- as.cl_ultrametric(*cl_ultrametric*), 32
- as.hclust, 33
- bclust, 8
- Cassini, 3
- CKME, 4
- cl_agreement, 5, 19
- cl_bag, 8
- cl_boot, 8, 9
- cl_class_ids(*n_of_classes*), 50
- cl_classes, 10, 39
- cl_consensus, 8, 11, 24, 28, 44, 50

- cl_dendrogram, 45
- cl_dendrogram (*hierarchy*), 38
- cl_dissimilarity, 7, 8, 11, 15, 24, 27, 33, 45
- cl_ensemble, 5, 11, 16, 20, 24, 27, 28
- cl_fuzziness, 21
- cl_hard_partition (*partition*), 52
- cl_hierarchy (*hierarchy*), 38
- cl_join (*lattice*), 44
- cl_margin, 23
- cl_medoid, 8, 11, 14, 23
- cl_meet, 17, 39, 53
- cl_meet (*lattice*), 44
- cl_membership, 24, 29, 54
- cl_object_names, 26
- cl_pam, 27
- cl_partition, 45
- cl_partition (*partition*), 52
- cl_pclust, 28, 28
- cl_predict, 30
- cl_prototypes, 31
- cl_tabulate, 32
- cl_ultrametric, 32, 36, 39, 44, 47, 49
- cl_validity, 34
- classAgreement, 7
- cluster.stats, 35
- clustIndex, 35
- cmeans, 10, 56
- consensus, 14
- constrOptim, 49
- cophenetic, 33

- dendrogram, 33
- diana, 34
- dist, 2, 18, 26, 35, 42, 43, 46, 48, 52

- fclustIndex, 22, 35
- fit_ultrametric_target, 35

- GVME, 36, 37
- GVME_Consensus, 37

- hclust, 33
- hierarchy, 38

- is.cl_dendrogram (*hierarchy*), 38
- is.cl_ensemble (*cl_ensemble*), 20
- is.cl_hard_partition (*partition*), 52

- is.cl_hierarchy, 33, 52
- is.cl_hierarchy (*hierarchy*), 38
- is.cl_partition, 25, 51, 52
- is.cl_partition (*partition*), 52
- is.cl_soft_partition (*partition*), 52

- Kinship82, 40, 41
- Kinship82_Consensus, 41
- kmeans, 10, 56
- kmedoids, 27, 42

- l1_fit_ultrametric, 43, 50
- l1_fit_ultrametric_target (*fit_ultrametric_target*), 35
- lattice, 44
- lp, 42
- ls_fit_addtree (*addtree*), 2
- ls_fit_centroid (*addtree*), 2
- ls_fit_sum_of_ultrametrics, 46
- ls_fit_ultrametric, 2, 13, 36, 44, 47, 47
- ls_fit_ultrametric_target (*fit_ultrametric_target*), 35

- mlbench.cassini, 4

- n_of_classes, 50
- n_of_objects, 26, 51
- nlm, 58, 59

- Ops.cl_dendrogram (*lattice*), 44
- Ops.cl_hierarchy (*lattice*), 44
- Ops.cl_partition (*lattice*), 44
- optim, 58, 59

- pam, 24, 27, 42
- partition, 52
- pclust, 29, 53
- pclust_family (*pclust*), 53
- pclust_object (*pclust*), 53
- Phonemes, 56
- plot, 3
- plot.cl_dendrogram (*hierarchy*), 38
- pmatch, 5, 11, 16, 21

- replicate, 10

silhouette, [29](#), [35](#)
solve_LSAP, [57](#)
Summary.cl_hierarchy(*lattice*), [44](#)
Summary.cl_partition(*lattice*), [44](#)
sumt, [13](#), [43](#), [48](#), [58](#)