Package 'anticlust'

June 29, 2020

Type Package

Title Subset Partitioning via Anticlustering

Version 0.5.0

Author Martin Papenberg [aut, cre] (<https://orcid.org/0000-0002-9900-4268>),
Meik Michalke [ctb] (centroid based clustering algorithm),
Gunnar W. Klau [ths],
Juliane V. Tkotz [ctb] (package logo)

Maintainer Martin Papenberg <martin.papenberg@hhu.de>

Description The method of anticlustering partitions a pool of elements into groups (i.e., anticlusters) in such a way that the between-group similarity is maximized and -- at the same time -- the within-group heterogeneity is maximized. This reverses the logic of cluster analysis that strives for high within-group homogeneity and low similarity of the different groups. Computationally, anticlustering is accomplished by maximizing instead of minimizing a clustering objective function, such as the intra-cluster variance (used in k-means clustering) or the sum of pairwise distances within clusters. The function anticlustering() implements exact and heuristic anticlustering algorithms as described in Papenberg and Klau (2020; <doi:10.1037/met0000301>). The exact approach requires that the GNU linear programming kit (<https://www.gnu.org/software/glpk/glpk.html>) is available and the R package 'Rglpk' (<https://cran.R-project.org/package=Rglpk>) is installed. Some other functions are available to solve classical clustering problems. The function balanced_clustering() applies a cluster analysis under size constraints, i.e., creates equal-sized clusters. The function matching() can be used for (unrestricted, bipartite, or K-partite) matching. The function wce() can be used optimally solve the (weighted) cluster editing problem, also known as correlation clustering, clique partitioning problem or transitivity clustering.

License MIT + file LICENSE

URL https://github.com/m-Py/anticlust

BugReports https://github.com/m-Py/anticlust/issues

Depends R (>= 3.6.0)

Imports Matrix, RANN (>= 2.6.0)

Suggests Rglpk, testthat

Encoding UTF-8

LazyData true

RoxygenNote 7.0.2

SystemRequirements The exact (anti)clustering algorithms require that
the GNU linear programming kit (GLPK library) is installed
(<http://www.gnu.org/software/glpk/>).

NeedsCompilation no

Repository CRAN

Date/Publication 2020-06-29 10:20:03 UTC

R topics documented:

`anticlustering` *Anticlustering*

Description

Create groups of elements (anticlusters) that are as similar as possible to each other, by maximizing the heterogeneity within groups. Implements anticlustering algorithms as described in Papenberg and Klau (2020; <doi:10.1037/met0000301>).

Usage

```
anticlustering(
  x,
  K,
  objective = "diversity",
  method = "exchange",
  preclustering = FALSE,
  categories = NULL
)
```

Arguments

`x`	The data input. Can be one of two structures: (1) A feature matrix where rows correspond to elements and columns correspond to variables (a single numeric variable can be passed as a vector). (2) An N x N matrix dissimilarity matrix; can be an object of class `dist` (e.g., returned by `dist` or `as.dist`) or a `matrix` where the entries of the upper and lower triangular matrix represent pairwise dissimilarities.
`K`	How many anticlusters should be created. Alternatively: A vector of length `nrow(x)` describing how elements are assigned to anticlusters before the optimization starts.
`objective`	The objective to be maximized. The option "diversity" (default) maximizes the cluster editing objective function; the option "variance" maximizes the k-means objective function. See Details.
`method`	One of "exchange" (default) or "ilp". See Details.
`preclustering`	Boolean. Should a preclustering be conducted before anticlusters are created? Defaults to `FALSE`. See Details.
`categories`	A vector, data.frame or matrix representing one or several categorical constraints. See Details.

Details

This function is used to solve anticlustering. That is, `K` groups are created in such a way that all groups are as similar as possible. This is accomplished by maximizing instead of minimizing a clustering objective function. This function natively supports the maximization of two clustering objective functions:

- cluster editing ‘diversity‘ objective, setting `objective = "diversity"` (default)
- k-means ‘variance‘ objective, setting `objective = "variance"`

The k-means objective is the variance within groups—that is, the sum of the squared distances between each element and its cluster center (see `variance_objective`). The cluster editing objective is the sum of pairwise distances within groups (see `diversity_objective`). Maximizing either of these clustering objectives (i.e., anticlustering) will partition the data set into similar groups, whereas traditional cluster analysis is used to obtain a low between-group similarity.

Anticluster editing is also known as the »maximum diverse grouping problem« because it maximizes group diversity as measured by the sum of pairwise distances. Hence, anticlustering maximizes between-group similarity by maximizing within-group heterogeneity. In previous versions of this package, `method = "distance"` was used (and is still supported) to request anticluster editing, but now `method = "diversity"` is preferred because there are several clustering objectives based on pairwise distances (e.g., see `dispersion_objective`).

If the data input `x` is a feature matrix (that is: each row is a "case" and each column is a "variable") and the option `objective = "diversity"` is used, the Euclidean distance is computed as the basic unit of the anticluster editing objective. If a different measure of dissimilarity is preferred, you may pass a self-generated dissimiliarity matrix via the argument `x`.

In the standard case, groups of equal size are generated. Adjust the argument `K` to create groups of different size.

Heuristic anticlustering

By default, a heuristic method is employed for anticlustering: the exchange method (`method = "exchange"`). Building on an initial assignment of elements to anticlusters, elements are sequentially swapped between anticlusters in such a way that each swap improves set similarity by the largest amount that is possible. In the default case, elements are randomly assigned to anticlusters before the exchange procedure starts; however, it is also possible to explicitly specify the initial assignment using the argument `K` (in this case, `K` has length `nrow(x)`). The exchange procedure is repeated for each element. Because each possible swap is investigated for each element, the total number of exchanges grows quadratically with input size, rendering the exchange method unsuitable for large N.

When setting `preclustering = TRUE`, only the `K -1` most similar elements serve as exchange partners, which can dramatically speed up the optimization (more information on the preclustering option is included below). This option is recommended for larger N. For very large N, check out the function `fast_anticlustering` that was specifically implemented to process very large data sets.

Exact anticlustering

An optimal anticluster editing objective can be found via integer linear programming (the integer linear program implemented here can be found in Papenberg & Klau, 2020, (8) - (12)). To this end, set `method = "ilp"`. To obtain an optimal solution, the open source GNU linear programming kit (available from https://www.gnu.org/software/glpk/glpk.html) and the R package `Rglpk` must be installed. The optimal solution is retrieved by setting `objective = "diversity"`, `method = "ilp"` and `preclustering = FALSE`. Use this combination of arguments only for small problem sizes.

To relax the optimality requirement, it is possible to set the argument `preclustering = TRUE`. In this case, the anticluster editing objective is still optimized using integer linear programming, but a preprocessing forbids very similar elements to be assigned to the same anticluster. The preclustering reduces the size of the solution space, making the integer linear programming approach applicable for larger problem instances. With preclustering, optimality is no longer guaranteed, but the solution is usually optimal or very close to optimal.

The variance criterion cannot be optimized to optimality using integer linear programming because the k-means objective function is not linear. However, it is possible to employ the function `generate_partitions` to obtain optimal solutions for small problem instances.

Preclustering

A useful heuristic for anticlustering is to form small groups of very similar elements and assign these to different groups. This logic is used as a preprocessing when setting `preclustering =`

`TRUE`. That is, before the anticlustering objective is optimized, a cluster analysis identifies small groups of similar elements (pairs if `K = 2`, triplets if `K = 3`, and so forth). The optimization of the anticlustering objective is then conducted under the constraint that these matched elements cannot be assigned to the same group. When using the exchange algorithm, preclustering is conducted using a call to `matching`. When using `method = "ilp"`, the preclustering optimally finds groups of minimum pairwise distance by solving the integer linear program described in Papenberg and Klau (2020; (8) - (10), (12) - (13)).

Categorical constraints

The argument `categories` may induce categorical constraints. The grouping variables indicated by `categories` will be balanced out across anticlusters. Currently, this functionality is only available in combination with the exchange method, but not with the integer linear programming approach.

Optimize a custom objective function

It is possible to pass a `function` to the argument `objective`. See `dispersion_objective` for an example. If `objective` is a function, the exchange method assigns elements to anticlusters in such a way that the return value of the custom function is maximized (hence, the function should return larger values when the between-group similarity is higher). The custom function has to take two arguments: the first is the data argument, the second is the clustering assignment. That is, the argument `x` will be passed down to the user-defined function as first argument. **However, only after** `as.matrix` has been called on `x`. This implies that in the function body, columns of the data set cannot be accessed using `data.frame` operations such as `$`. Objects of class `dist` will be converted to matrix as well.

Value

A vector of length N that assigns a group (i.e, a number between 1 and `K`) to each input element.

Author(s)

Martin Papenberg <`martin.papenberg@hhu.de`>

References

Grötschel, M., & Wakabayashi, Y. (1989). A cutting plane algorithm for a clustering problem. Mathematical Programming, 45, 59-96.

Papenberg, M., & Klau, G. W. (2020). Using anticlustering to partition data sets into equivalent parts. Psychological Methods. Advance Online Publication. https://doi.org/10.1037/met0000301.

Späth, H. (1986). Anticlustering: Maximizing the variance criterion. Control and Cybernetics, 15, 213-218.

See Also

`fast_anticlustering`

`variance_objective`

`diversity_objective`

Examples

```
# Optimize the cluster editing (diversity) criterion
anticlusters <- anticlustering(
  schaper2019[, 3:6],
  K = 3,
  categories = schaper2019$room
)
# Compare feature means by anticluster
by(schaper2019[, 3:6], anticlusters, function(x) round(colMeans(x), 2))
# Compare standard deviations by anticluster
by(schaper2019[, 3:6], anticlusters, function(x) round(apply(x, 2, sd), 2))
# check that the "room" is balanced across anticlusters:
table(anticlusters, schaper2019$room)

# You can try to improve the solution using the old output as
# the new K argument:
new_anticlusters <- anticlustering(
  schaper2019[, 3:6],
  K = anticlusters,
  categories = schaper2019$room
)


## Use preclustering and variance (k-means) criterion on large data set
N <- 1000
K = 2
lds <- data.frame(f1 = rnorm(N), f2 = rnorm(N))
ac <- anticlustering(
  lds,
  K = K,
  objective = "variance",
  preclustering = TRUE
)

# The following is equivalent to setting `preclustering = TRUE`:
cl <- balanced_clustering(lds, K = N / K)
ac <- anticlustering(
  lds,
  K = K,
  objective = "variance",
  categories = cl
)
```

`balanced_clustering` *Create balanced clusters of equal size*

Description

Create balanced clusters of equal size

Usage

```
balanced_clustering(x, K, method = "centroid")
```

Arguments

`x`	The data input. Can be one of two structures: (1) A feature matrix where rows correspond to elements and columns correspond to variables (a single numeric variable can be passed as a vector). (2) An N x N matrix dissimilarity matrix; can be an object of class `dist` (e.g., returned by `dist` or `as.dist`) or a `matrix` where the entries of the upper and lower triangular matrix represent pairwise dissimilarities.
`K`	How many clusters should be created.
`method`	One of "centroid" or "ilp". See Details.

Details

This function partitions a set of elements into `K` equal-sized clusters. The function offers two methods: a heuristic and an exact method. The heuristic (`method = "centroid"`) first computes the centroid of all data points. If the input is a feature matrix, the centroid is defined as the mean vector of all columns. If the input is a dissimilarity matrix, the most central element acts as the centroid; the most central element is defined as the element having the minimum maximal distance to all other elements. After identifying the centroid, the algorithm proceeds as follows: The element having the highest distance from the centroid is clustered with its `(N/K) -1` nearest neighbours (neighbourhood is defined according to the Euclidean distance if the data input is a feature matrix). From the remaining elements, again the element farthest to the centroid is selected and clustered with its `(N/K) -1` neighbours; the procedure is repeated until all elements are part of a cluster.

An exact method (`method = "ilp"`) can be used to solve equal-sized weighted cluster editing optimally (implements the integer linear program described in Papenberg and Klau, 2020; (8) - (10), (12) - (13)). The cluster editing objective is the sum of pairwise distances within clusters; clustering is accomplished by minimizing this objective. If the argument `x` is a features matrix, the Euclidean distance is computed as the basic unit of the cluster editing objective. If another distance measure is preferred, users may pass a self-computed dissimiliarity matrix via the argument `x`. The optimal cluster editing objective can be found via integer linear programming. To obtain an optimal solution, the open source GNU linear programming kit (available from https://www.gnu.org/software/glpk/glpk.html) and the R package `Rglpk` must be installed.

Value

An integer vector representing the cluster affiliation of each data point

Author(s)

Martin Papenberg <`martin.papenberg@hhu.de`>

Meik Michalke <`meik.michalke@hhu.de`>

Source

The centroid method was originally developed and contributed by Meik Michalke. It was later rewritten by Martin Papenberg, who also implemented the integer linear programming method.

References

Grötschel, M., & Wakabayashi, Y. (1989). A cutting plane algorithm for a clustering problem. Mathematical Programming, 45, 59–96.

Papenberg, M., & Klau, G. W. (2020). Using anticlustering to partition data sets into equivalent parts. Psychological Methods. Advance Online Publication. https://doi.org/10.1037/met0000301.

Examples

```
# Cluster a data set and visualize results
N <- 1000
lds <- data.frame(f1 = rnorm(N), f2 = rnorm(N))
cl <- balanced_clustering(lds, K = 10)
plot_clusters(lds, clusters = cl)

# Repeat using a distance matrix as input
cl2 <- balanced_clustering(dist(lds), K = 10)
plot_clusters(lds, clusters = cl2)
```

`categorical_sampling` *Random sampling employing a categorical constraint*

Description

This function can be used to obtain a stratified split of a data set.

Usage

```
categorical_sampling(categories, K)
```

Arguments

`categories`	A matrix or vector of one or more categorical variables.
`K`	The number of groups that are returned.

Details

This function can be used to obtain a stratified split of a data set. Using this function is like calling a`anticlustering`‘ with argument ‘categories‘ where no optimization is conducted; the categories are just evenly split between samples. Apart from the restriction that categories are balanced between samples, the split is random.

Value

A vector representing the sample each element was assigned to.

Examples

```
data(schaper2019)
categories <- schaper2019$room
groups <- categorical_sampling(categories, K = 6)
table(groups, categories)
```

`dispersion_objective` *Cluster dispersion*

Description

Compute the dispersion objective for a given clustering (i.e., the minimum distance between two elements within the same cluster).

Usage

```
dispersion_objective(x, clusters)
```

Arguments

`x`	The data input. Can be one of two structures: (1) A feature matrix where rows correspond to elements and columns correspond to variables (a single numeric variable can be passed as a vector). (2) An N x N matrix dissimilarity matrix; can be an object of class `dist` (e.g., returned by `dist` or `as.dist`) or a `matrix` where the entries of the upper and lower triangular matrix represent pairwise dissimilarities.
`clusters`	A vector representing (anti)clusters (e.g., returned by `anticlustering`).

Details

The dispersion is the minimum distance between two elements within the same cluster. When the input `x` is a feature matrix, the Euclidean distance is used as the distance unit. Maximizing the dispersion maximizes the minimum heterogeneity within clusters and is an anticlustering task.

References

Brusco, M. J., Cradit, J. D., & Steinley, D. (in press). Combining diversity and dispersion criteria for anticlustering: A bicriterion approach. British Journal of Mathematical and Statistical Psychology. https://doi.org/10.1111/bmsp.12186

Examples

```
N <- 50 # number of elements
M <- 2  # number of variables per element
K <- 5  # number of clusters
random_data <- matrix(rnorm(N * M), ncol = M)
random_clusters <- sample(rep_len(1:K, N))
dispersion_objective(random_data, random_clusters)

# Maximize the dispersion
optimized_clusters <- anticlustering(
  random_data,
  K = random_clusters,
  objective = dispersion_objective
)
dispersion_objective(random_data, optimized_clusters)
```

`diversity_objective` *(Anti)cluster editing "diversity" objective*

Description

Compute the diversity for a given clustering.

Usage

```
diversity_objective(x, clusters)
```

Arguments

`x`	The data input. Can be one of two structures: (1) A data matrix where rows correspond to elements and columns correspond to features (a single numeric feature can be passed as a vector). (2) An N x N matrix dissimilarity matrix; can be an object of class `dist` (e.g., returned by `dist` or `as.dist`) or a `matrix` where the entries of the upper and lower triangular matrix represent the pairwise dissimilarities.
`clusters`	A vector representing (anti)clusters (e.g., returned by `anticlustering`).

Details

The objective function used in (anti)cluster editing is the diversity, i.e., the sum of the pairwise distances between elements within the same groups. When the input `x` is a feature matrix, the Euclidean distance is computed as the basic distance unit of this objective.

Value

The cluster editing objective

Author(s)

Martin Papenberg <martin.papenberg@hhu.de>

References

Brusco, M. J., Cradit, J. D., & Steinley, D. (in press). Combining diversity and dispersion criteria for anticlustering: A bicriterion approach. British Journal of Mathematical and Statistical Psychology. https://doi.org/10.1111/bmsp.12186

Papenberg, M., & Klau, G. W. (2020). Using anticlustering to partition data sets into equivalent parts. Psychological Methods. Advance Online Publication. https://doi.org/10.1037/met0000301.

Examples

```
data(iris)
distances <- dist(iris[1:60, -5])
## Clustering
clusters <- balanced_clustering(distances, K = 3)
# This is low:
diversity_objective(distances, clusters)
## Anticlustering
anticlusters <- anticlustering(distances, K = 3)
# This is higher:
diversity_objective(distances, anticlusters)
```

`fast_anticlustering` *Fast anticlustering*

Description

The most efficient way to solve anticlustering optimizing the k-means variance criterion with an exchange method. Can be used for very large data sets.

Usage

```
fast_anticlustering(x, K, k_neighbours = Inf, categories = NULL)
```

Arguments

`x`	A numeric vector, matrix or data.frame of data points. Rows correspond to elements and columns correspond to features. A vector represents a single numeric feature.
`K`	How many anticlusters should be created.
`k_neighbours`	The number of neighbours that serve as exchange partner for each element. Defaults to Inf, i.e., each element is exchanged with each element in other groups.
`categories`	A vector, data.frame or matrix representing one or several categorical constraints.

Details

This function was created to make anticlustering applicable to large data sets (e.g., 100,000 elements). It optimizes the k-means variance objective because computing all pairwise distances is not feasible for many elements. Additionally, this function employs a speed-optimized exchange method. For each element, the potential exchange partners are generated using a nearest neighbor search with the function `nn2` from the `RANN` package. The nearest neighbors then serve as exchange partners. This approach is inspired by the preclustering heuristic according to which good solutions are found when similar elements are in different sets—by swapping nearest neighbors, this will often be the case. The number of exchange partners per element has to be set using the argument `k_neighbours`; by default, it is set to `Inf`, meaning that all possible swaps are tested. This default must be changed by the user for large data sets. More exchange partners generally improve the output, but also increase run time.

When setting the `categories` argument, exchange partners will be generated from the same category. Note that when `categories` has multiple columns (i.e., each element is assigned to multiple columns), each combination of categories is treated as a distinct category by the exchange method.

Author(s)

Martin Papenberg <martin.papenberg@hhu.de>

See Also

`anticlustering`

`variance_objective`

Examples

```
features <- iris[, - 5]

start <- Sys.time()
ac_exchange <- fast_anticlustering(features, K = 3)
Sys.time() - start

## The following call is equivalent to the call above:
start <- Sys.time()
ac_exchange <- anticlustering(features, K = 3, objective = "variance")
Sys.time() - start

## Improve run time by using fewer exchange partners:
start <- Sys.time()
ac_fast <- fast_anticlustering(features, K = 3, k_neighbours = 10)
Sys.time() - start

by(features, ac_exchange, function(x) round(colMeans(x), 2))
by(features, ac_fast, function(x) round(colMeans(x), 2))
```

`generate_partitions` *Generate all partitions of same cardinality*

Description

Generate all partitions of same cardinality

Usage

```
generate_partitions(N, K, generate_permutations = FALSE)
```

Arguments

`N`	The total N. `K` has to be dividble by `N`.
`K`	How many partitions
`generate_permutations`	If TRUE, all permutations are returned, resulting in duplicate partitions.

Details

In principle, anticlustering can be solved to optimality by generating all possible partitions of N items into K groups. The example code below illustrates how to do this. However, this approach only works for small N because the number of partitions grows exponentially with N.

The partition c(1, 2, 2, 1) is the same as the partition c(2, 1, 1, 2) but they correspond to different permutations of the elements [1, 1, 2, 2]. If the argument `generate_permutations` is `TRUE`, all permutations are returned. To solve balanced anticlustering exactly, it is sufficient to inspect all partitions while ignoring duplicated permutations.

Value

A list of all partitions (or permutations if `generate_permutations` is `TRUE`).

Author(s)

Martin Papenberg <`martin.papenberg@hhu.de`>

References

Papenberg, M., & Klau, G. W. (2020). Using anticlustering to partition data sets into equivalent parts. Psychological Methods. Advance Online Publication. https://doi.org/10.1037/met0000301.

Examples

```
## Generate all partitions to solve k-means anticlustering
## to optimality.

N <- 14
K <- 2
features <- matrix(sample(N * 2, replace = TRUE), ncol = 2)
partitions <- generate_partitions(N, K)
length(partitions) # number of possible partitions

## Create an objective function that takes the partition
## as first argument (then, we can use sapply to compute
## the objective for each partition)
var_obj <- function(clusters, features) {
  variance_objective(features, clusters)
}

all_objectives <- sapply(
  partitions,
  FUN = var_obj,
  features = features
)

## Check out distribution of the objective over all partitions:
hist(all_objectives) # many large, few low objectives
## Get best k-means anticlustering objective:
best_obj <- max(all_objectives)
## It is possible that there are multiple best solutions:
sum(all_objectives == best_obj)
## Select one best partition:
best_anticlustering <- partitions[all_objectives == best_obj][[1]]
## Look at mean for each partition:
by(features, best_anticlustering, function(x) round(colMeans(x), 2))


## Get best k-means clustering objective:
min_obj <- min(all_objectives)
sum(all_objectives == min_obj)
## Select one best partition:
best_clustering <- partitions[all_objectives == min_obj][[1]]

## Plot minimum and maximum objectives:
user_par <- par("mfrow")
par(mfrow = c(1, 2))
plot_clusters(
  features,
  best_anticlustering,
  illustrate_variance = TRUE,
  main = "Maximum variance"
)
plot_clusters(
  features,
  best_clustering,
  illustrate_variance = TRUE,
  main = "Minimum variance"
)
par(mfrow = user_par)
```

`matching` *Matching*

Description

Conduct K-partite or unrestricted (minimum distance) matching to find pairs or groups of similar elements. By default, finding matches is based on the Euclidean distance between data points, but a custom dissimilarity measure can also be employed.

Usage

```
matching(
  x,
  p = 2,
  match_between = NULL,
  match_within = NULL,
  match_extreme_first = TRUE,
  target_group = NULL,
  sort_output = TRUE
)
```

Arguments

`x`	The data input. Can be one of two structures: (1) A feature matrix where rows correspond to elements and columns correspond to variables (a single numeric variable can be passed as a vector). (2) An N x N matrix dissimilarity matrix; can be an object of class `dist` (e.g., returned by `dist` or `as.dist`) or a `matrix` where the entries of the upper and lower triangular matrix represent pairwise dissimilarities.
`p`	The size of the groups; the default is 2, in which case the function returns pairs.
`match_between`	An optional vector, `data.frame` or matrix representing one or several categorical constraints. If passed, the argument `p` is ignored and matches are sought between elements of different categories.
`match_within`	An optional vector, `data.frame` or matrix representing one or several categorical constraints. If passed, matches are sought between elements of the same category.
`match_extreme_first`	Logical: Determines if matches are first sought for extreme elements first or for central elements. Defaults to `TRUE`.

`target_group`	Currently, the options "none", smallest" and "diverse" are supported. See Details.
`sort_output`	Boolean. If `TRUE` (default), the output clusters are sorted by similarity. See Details.

Details

If the data input `x` is a feature matrix, matching is based on the Euclidean distance between data points. If the argument `x` is a dissimilarity matrix, matching is based on the user-specified dissimilarities. To find matches, the algorithm proceeds by selecting a target element and then searching its nearest neighbours. Critical to the behaviour or the algorithm is the order in which target elements are selected. By default, the most extreme elements are selected first, i.e., elements with the highest distance to the centroid of the data set (see `balanced_clustering` that relies on the same algorithm). Set the argument `match_extreme_first` to `FALSE`, to enforce that elements close to the centroid are first selected as targets.

If the argument `match_between` is passed and the groups specified via this argument are of different size, target elements are selected from the smallest group by default (because in this group, all elements can be matched). However, it is also possible to specify how matches are selected through the option `target_group`. When specifying `"none"`, matches are always selected from extreme elements, irregardless of the group sizes (or from central elements first if `match_extreme_first = FALSE`). With option `"smallest"`, matches are selected from the smallest group. With option `"diverse"`, matches are selected from the most heterogenous group according to the sum of pairwise distances within groups.

The output is an integer vector encoding which elements have been matched. The grouping numbers are sorted by similarity. That is, elements with the grouping number »1« have the highest intragroup similarity, followed by 2 etc (groups having the same similarity index are still assigned a different grouping number, though). Similarity is measured as the sum of pairwise (Euclidean) distances within groups (see `diversity_objective`). To prevent sorting by similarity (this is some extra computational burden), set `sort_output = FALSE`. Some unmatched elements may be `NA`. This happens if it is not possible to evenly split the item pool evenly into groups of size `p` or if the categories described by the argument `match_between` are of different size.

Value

An integer vector encoding the matches. See Details for more information.

Note

It is possible to specify grouping restrictions via `match_between` and `match_within` at the same time.

Author(s)

Martin Papenberg <`martin.papenberg@hhu.de`>

Examples

```
# Find triplets
```

```
N <- 120
lds <- data.frame(f1 = rnorm(N), f2 = rnorm(N))
triplets <- matching(lds, p = 3)
plot_clusters(
  lds,
  clusters = triplets,
  within_connection = TRUE
)

# Bipartite matching with unequal-sized groups:
# Only selects matches for some elements
N <- 100
data <- matrix(rnorm(N), ncol = 1)
groups <- sample(1:2, size = N, replace = TRUE, prob = c(0.8, 0.2))
matched <- matching(data[, 1], match_between = groups)
plot_clusters(
  cbind(groups, data),
  clusters = matched,
  within_connection = TRUE
)

# Match objects from the same category only
matched <- matching(
  schaper2019[, 3:6],
  p = 3,
  match_within = schaper2019$room
)
head(table(matched, schaper2019$room))

# Match between different plant species in the »iris« data set
species <- iris$Species != "versicolor"
matched <- matching(
  iris[species, 1],
  match_between = iris[species, 5]
)
# Adjust `match_extreme_first` argument
matched2 <- matching(
  iris[species, 1],
  match_between = iris[species, 5],
  match_extreme_first = FALSE
)
# Plot the matching results
user_par <- par("mfrow")
par(mfrow = c(1, 2))
data <- data.frame(
  Species = as.numeric(iris[species, 5]),
  Sepal.Length = iris[species, 1]
)
plot_clusters(
  data,
  clusters = matched,
  within_connection = TRUE,
  main = "Extreme elements matched first"
)
plot_clusters(
  data,
  clusters = matched2,
  within_connection = TRUE,
  main = "Central elements matched first"
)
par(mfrow = user_par)
```

`mean_sd_obj` *An objective function measuring similarity of sets*

Description

Compute the discrepancy in means and standard deviations between clusters.

Usage

```
mean_sd_obj(features, clusters)
```

Arguments

`features`	A matrix or data.frame of data points. Rows correspond to elements and columns correspond to features.
`clusters`	A clustering vector

Details

This function can be passed as the argument `objective` to the function `anticlustering` to minimize differences in means and standard deviations between anticlusters.

Value

A value quantifying similarity in means and standard deviations. Higher values indicate that means and standard deviations are more similar.

Author(s)

Martin Papenberg <`martin.papenberg@hhu.de`>

Examples

```
data(schaper2019)
features <- schaper2019[1:48, 3:6]
ac <- anticlustering(
  features,
  K = 3,
  categories = schaper2019$room[1:48],
  objective = mean_sd_obj
)
by(features, ac, function(x) round(colMeans(x), 2))
by(features, ac, function(x) round(apply(x, 2, sd), 2))
```

`mean_sd_tab` *Means and standard deviations by group variable formatted in table*

Description

Means and standard deviations by group variable formatted in table

Usage

```
mean_sd_tab(features, groups, decimals = 2, na.rm = FALSE, return_diff = FALSE)
```

Arguments

`features`	A data frame of features
`groups`	A grouping vector
`decimals`	The number of decimals
`na.rm`	Should NAs be removed prior to computing stats (Default = FALSE)
`return_diff`	Boolean. Should an additional row be printed that contains the difference between minimum and maximum

Value

A table that illustrates means and standard deviations (in brackets)

Author(s)

Martin Papenberg <`martin.papenberg@hhu.de`>

Examples

```
data(iris)
mean_sd_tab(iris[, -5], iris[, 5])
```

n_partitions *Number of equal sized partitions*

Description

Number of equal sized partitions

Usage

```
n_partitions(N, K)
```

Arguments

N	How many elements
K	How many partitions

Value

The number of partitions

Examples

```
n_partitions(20, 2)
```

plot_clusters *Visualize a cluster analysis*

Description

Visualize a cluster analysis

Usage

```
plot_clusters(
  features,
  clusters,
  within_connection = FALSE,
  between_connection = FALSE,
  illustrate_variance = FALSE,
  show_axes = FALSE,
  xlab = NULL,
  ylab = NULL,
  xlim = NULL,
  ylim = NULL,
  main = "",
  cex = 1.2,
  cex.axis = 1.2,
  cex.lab = 1.2,
  lwd = 1.5,
  lty = 2,
  frame.plot = FALSE,
  cex_centroid = 2
)
```

Arguments

`features`	A data.frame or matrix representing the features that are plotted. Must have two columns.
`clusters`	A vector representing the clustering
`within_connection`	Boolean. Connect the elements within each clusters through lines? Useful to illustrate a graph structure.
`between_connection`	Boolean. Connect the elements between each clusters through lines? Useful to illustrate a graph structure. (This argument only works for two clusters).
`illustrate_variance`	Boolean. Illustrate the variance criterion in the plot?
`show_axes`	Boolean, display values on the x and y-axis? Defaults to ‘FALSE‘.
`xlab`	The label for the x-axis
`ylab`	The label for the y-axis
`xlim`	The limits for the x-axis
`ylim`	The limits for the y-axis
`main`	The title of the plot
`cex`	The size of the plotting symbols, see `par`
`cex.axis`	The size of the values on the axes
`cex.lab`	The size of the labels of the axes
`lwd`	The width of the lines connecting elements.
`lty`	The line type of the lines connecting elements (see `par`).
`frame.plot`	a logical indicating whether a box should be drawn around the plot.
`cex_centroid`	The size of the cluster center symbol (has an effect only if `illustrate_variance` is `TRUE`)

Details

In most cases, the argument `clusters` is a vector returned by one of the functions `anticlustering`, `balanced_clustering` or `matching`. However, the plotting function can also be used to plot the results of other cluster functions such as `kmeans`. This function is usually just used to get a fast impression of the results of an (anti)clustering assignment, but limited in its functionality. It is useful for depicting the intra-cluster connections using argument `within_connection`.

Author(s)

Martin Papenberg <martin.papenberg@hhu.de>

Examples

```
N <- 15
features <- matrix(runif(N * 2), ncol = 2)
K <- 3
clusters <- balanced_clustering(features, K = K)
anticlusters <- anticlustering(features, K = K)
user_par <- par("mfrow")
par(mfrow = c(1, 2))
plot_clusters(features, clusters, main = "Cluster editing", within_connection = TRUE)
plot_clusters(features, anticlusters, main = "Anticluster editing", within_connection = TRUE)
par(mfrow = user_par)
```

---

plot_similarity *Plot similarity objective by cluster*

Description

Plot similarity objective by cluster

Usage

```
plot_similarity(x, groups)
```

Arguments

`x`	The data input. Can be one of two structures: (1) A data matrix where rows correspond to elements and columns correspond to features (a single numeric feature can be passed as a vector). (2) An N x N matrix dissimilarity matrix; can be an object of class `dist` (e.g., returned by `dist` or `as.dist`) or a `matrix` where the entries of the upper and lower triangular matrix represent the pairwise dissimilarities.
`groups`	A grouping vector of length N, usually the output of `matching`.

Details

Plots the sum of pairwise distances by group.

Value

The diversity (sum of distances) by group.

Author(s)

Martin Papenberg <`martin.papenberg@hhu.de`>

See Also

`diversity_objective`

Examples

```
# Match elements and plot similarity by match
N <- 100
lds <- data.frame(f1 = rnorm(N), f2 = rnorm(N))
pairs <- matching(lds, p = 2)
plot_similarity(lds, pairs)
```

`schaper2019` *Ratings for 96 words*

Description

A stimulus set that was used in experiments by Schaper, Kuhlmann and Bayen (2019a; 2019b). The item pool consists of 96 German words. Each word represents an object that is either typically found in a bathroom or in a kitchen.

Usage

`schaper2019`

Format

A data frame with 96 rows and 7 variables

item The name of an object (in German)

room The room in which the item is typically found; can be 'kitchen' or 'bathroom'

rating_consistent How expected would it be to find the `item` in the typical `room`

rating_inconsistent How expected would it be to find the `item` in the atypical `room`

syllables The number of syllables in the object name

frequency A value indicating the relative frequency of the object name in German language (lower values indicate higher frequency)

list Represents the set affiliation of the `item` as realized in experiments by Schaper et al.

Source

Courteously provided by Marie Lusia Schaper and Ute Bayen.

References

Schaper, M. L., Kuhlmann, B. G., & Bayen, U. J. (2019a). Metacognitive expectancy effects in source monitoring: Beliefs, in-the-moment experiences, or both? Journal of Memory and Language, 107, 95–110. https://doi.org/10.1016/j.jml.2019.03.009

Schaper, M. L., Kuhlmann, B. G., & Bayen, U. J. (2019b). Metamemory expectancy illusion and schema-consistent guessing in source monitoring. Journal of Experimental Psychology: Learning, Memory, and Cognition, 45, 470. https://doi.org/10.1037/xlm0000602

Examples

```
head(schaper2019)
features <- schaper2019[, 3:6]

# Optimize the variance criterion
# (tends to maximize similarity in feature means)
anticlusters <- anticlustering(
  features,
  K = 3,
  objective = "variance",
  categories = schaper2019$room,
  method = "exchange"
)

# Means are quite similar across sets:
by(features, anticlusters, function(x) round(colMeans(x), 2))
# Check differences in standard deviations:
by(features, anticlusters, function(x) round(apply(x, 2, sd), 2))
# Room is balanced between the three sets:
table(Room = schaper2019$room, Set = anticlusters)

# Maximize the diversity criterion
ac_dist <- anticlustering(
  features,
  K = 3,
  objective = "diversity",
  categories = schaper2019$room,
  method = "exchange"
)
# With the distance criterion, means tend to be less similar,
# but standard deviations tend to be more similar:
by(features, ac_dist, function(x) round(colMeans(x), 2))
by(features, ac_dist, function(x) round(apply(x, 2, sd), 2))
```

`variance_objective` *Objective value for the variance criterion*

Description

Compute the k-means variance objective for a given clustering.

Usage

```
variance_objective(x, clusters)
```

Arguments

`x`	A vector, matrix or data.frame of data points. Rows correspond to elements and columns correspond to features. A vector represents a single feature.
`clusters`	A vector representing (anti)clusters (e.g., returned by `anticlustering` or `balanced_clustering`)

Details

The variance objective is given by the sum of the squared errors between cluster centers and individual data points. It is the objective function used in k-means clustering, see `kmeans`.

Value

The total within-cluster variance

Author(s)

Martin Papenberg <`martin.papenberg@hhu.de`>

References

Jain, A. K. (2010). Data clustering: 50 years beyond k-means. Pattern Recognition Letters, 31, 651–666.

Papenberg, M., & Klau, G. W. (2020). Using anticlustering to partition data sets into equivalent parts. Psychological Methods. Advance Online Publication. https://doi.org/10.1037/met0000301.

Späth, H. (1986). Anticlustering: Maximizing the variance criterion. Control and Cybernetics, 15, 213–218.

Examples

```
data(iris)
## Clustering
clusters <- balanced_clustering(
  iris[, -5],
  K = 3
)
# This is low:
variance_objective(
  iris[, -5],
  clusters
)
## Anticlustering
```

```
anticlusters <- anticlustering(
  iris[, -5],
  K = 3,
  objective = "variance"
)
# This is higher:
variance_objective(
  iris[, -5],
  anticlusters
)

# Illustrate variance objective
N <- 18
data <- matrix(rnorm(N * 2), ncol = 2)
cl <- balanced_clustering(data, K = 3)
plot_clusters(data, cl, illustrate_variance = TRUE)
```

`wce` *Exact weighted cluster editing*

Description

Optimally solves weighted cluster editing (also known as »correlation clustering« or »clique partitioning problem«).

Usage

```
wce(x)
```

Arguments

`x`	A N x N similarity matrix. Larger values indicate stronger agreement / similarity between a pair of data points

Details

Finds the clustering that maximizes the sum of pairwise similarities within clusters. In the input some similarities should be negative (indicating dissimilarity) because otherwise the maximum sum of similarities is obtained by simply joining all elements within a single big cluster.

Value

An integer vector representing the cluster affiliation of each data point

Note

This function requires the R package `Rglpk` and the GNU linear programming kit.

Author(s)

Martin Papenberg <martin.papenberg@hhu.de>

References

Bansal, N., Blum, A., & Chawla, S. (2004). Correlation clustering. Machine Learning, 56, 89–113.

Böcker, S., & Baumbach, J. (2013). Cluster editing. In Conference on Computability in Europe (pp. 33–44).

Grötschel, M., & Wakabayashi, Y. (1989). A cutting plane algorithm for a clustering problem. Mathematical Programming, 45, 59-96.

Wittkop, T., Emig, D., Lange, S., Rahmann, S., Albrecht, M., Morris, J. H., . . . Baumbach, J. (2010). Partitioning biological data with transitivity clustering. Nature Methods, 7, 419–420.

Examples

```
features <- swiss
distances <- dist(scale(swiss))
hist(distances)
# Define agreement as being close enough to each other.
# By defining low agreement as -1 and high agreement as +1, we
# solve *unweighted* cluster editing
agreements <- ifelse(as.matrix(distances) < 3, 1, -1)
clusters <- wce(agreements)
plot(swiss, col = clusters, pch = 19)
```

Index