Package 'anticlust'

March 14, 2025

Type Package

Title Subset Partitioning via Anticlustering

Version 0.8.10

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. Nagel [ctb] (package logo),
Martin Breuer [ctb] (Bicriterion algorithm by Brusco et al.),
Marie L. Schaper [ctb] (Example data set),
Max Diekhoff [ctb] (Optimal maximum dispersion algorithm),
Hannah Hengelbrock [ctb] (TPSDP heuristic by Yang et al.)

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

Description The method of anticlustering partitions a pool of elements into groups (i.e., anticlusters) with the goal of maximizing between-group similarity or within-group heterogeneity. The anticlustering approach thereby reverses the logic of cluster analysis that strives for high within-group homogeneity and clear separation between 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 main function anticlustering() gives access to optimal and heuristic anticlustering methods described in Papenberg and Klau (2021; <doi:10.1037/met0000301>), Brusco et al. (2020; <doi:10.1111/bmsp.12186>), Papenberg (2024; <doi:10.1111/bmsp.12315>), and Papenberg et al. (2025; <doi:10.1101/2025.03.03.641320>). The optimal algorithms require that an integer linear programming solver is installed. This package will install 'lpSolve' (<https://cran.r-project.org/package=lpSolve>) as a default solver, but it is also possible to use the package 'Rglpk' (<https://cran.r-project.org/package=Rglpk>), which requires the GNU linear programming kit (<https://www.gnu.org/software/glpk/glpk.html>), the package 'Rsymphony' (<https://cran.r-project.org/package=Rsymphony>), which requires the SYMPHONY ILP solver (<https://github.com/coin-or/SYMPHONY>), or the commercial solver Gurobi, which provides its own R package that is not available via CRAN (<https://www.gurobi.com/downloads/>). 'Rglpk', 'Rsymphony', 'gurobi' and their system dependencies have to be manually installed by the user because they are only suggested dependencies. Full access to the bicriterion anticlustering method proposed by Brusco et al. (2020) is given via the function bicrite-

rion_anticlustering(), while kplus_anticlustering() implements the full functionality of the k-plus anticlustering approach proposed by Papenberg (2024). 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, https://m-py.github.io/anticlust/

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

Depends R (>= 3.6.0)

Imports Matrix, RANN (>= 2.6.0), lpSolve

Suggests knitr, palmerpenguins, Rglpk, rmarkdown, Rsymphony, tinytest, gurobi

VignetteBuilder knitr, rmarkdown

Encoding UTF-8

LazyData true

NeedsCompilation yes

RoxygenNote 7.3.2

SystemRequirements Rendering the vignette requires pandoc (<https://pandoc.org/>).

Repository CRAN

Date/Publication 2025-03-13 23:20:05 UTC

Contents

`anticlustering` *Anticlustering*

Description

Partition a pool of elements into groups (i.e., anticlusters) with the aim of creating high within-group heterogeneity and high between-group similarity. Anticlustering is accomplished by maximizing instead of minimizing a clustering objective function. Implements anticlustering methods as described in Papenberg and Klau (2021; <doi:10.1037/met0000301>), Brusco et al. (2020; <doi:10.1111/bmsp.12186>), Papenberg (2024; <doi:10.1111/bmsp.12315>), and Papenberg et al. (2025; <doi:10.1101/2025.03.03.641320>).

Usage

```
anticlustering(
  x,
  K,
  objective = "diversity",
  method = "exchange",
  preclustering = FALSE,
  categories = NULL,
  repetitions = NULL,
  standardize = FALSE,
  cannot_link = NULL,
  must_link = 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) A vector describing the size of each group, or (b) a vector of length nrow(x) describing how elements are assigned to anticlusters before the optimization starts.
objective	The objective to be maximized. The options "diversity" (default; previously called "distance", which is still supported), "average-diversity", "variance", "kplus" and "dispersion" are natively supported. May also be a user-defined function. See Details.
method	One of "exchange" (default) , "local-maximum", "brusco", "ilp", or "2PML". 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 variables whose distribution should be similar between groups. See Details.
repetitions	The number of times a search heuristic is initiated when using method = "exchange", method = "local-maximum", method = "brusco", or method = "2PML". In the end, the best objective found across the repetitions is returned.
standardize	Boolean. If TRUE and x is a feature matrix, the data is standardized through a call to scale before the optimization starts. This argument is silently ignored if x is a distance matrix.
cannot_link	A 2 column matrix where each row has the indices of two elements that must not be assigned to the same anticluster.
must_link	A numeric vector of length nrow(x). Elements having the same value in this vector are assigned to the same anticluster.

Details

This function is used to solve anticlustering. That is, the data input is divided into K groups in such a way that elements within groups are heterogeneous and the different groups are similar. Anticlustering is accomplished by maximizing instead of minimizing a clustering objective function. The maximization of five objectives is natively supported (other functions can also defined by the user as described below):

- the diversity, setting objective = "diversity" (this is the default objective)
- the average diversity, which normalizes the diversity by cluster size, setting objective = "average-diversity"
- the k-means (or "variance") objective, setting objective = "variance"
- the k-plus objective, an extension of the k-means objective, setting objective = "kplus"
- the dispersion, which is the minimum distance between any two elements within the same cluster (setting objective = "dispersion")

The k-means objective is the within-group variance—that is, the sum of the squared distances between each element and its cluster center (see variance_objective). K-means anticlustering focuses on minimizing differences with regard to the means of the input variables (that is, the columns in x), but it ignores any other distribution characterstics such as the variance / standard deviation. K-plus anticlustering (using objective = "kplus") is an extension of the k-means criterion that

also minimizes differences with regard to the standard deviations between groups (for details see kplus_anticlustering). K-plus anticlustering can also be extended towards higher order moments such as skew and kurtosis; to consider these additional distribution characteristics, use the function kplus_anticlustering. Setting objective = "kplus" in anticlustering function will only consider means and standard deviations (in my experience, this is what users usually want). It is strongly recommended to set the argument standardize = TRUE when using the k-plus objective.

The "diversity" objective is the sum of pairwise distances of elements within the same groups (see diversity_objective). Hence, anticlustering using the diversity criterion maximizes between-group similarity by maximizing within-group heterogeneity (represented as the sum of all pairwise distances). If it is computed on the basis of the Euclidean distance (which is the default behaviour if x is a feature matrix), the diversity is an all rounder objective that tends to equate all distribution characteristics between groups (such as means, variances, ...). Note that the equivalence of within-group heterogeneity and between-group similarity only holds for equal-sized groups. For unequal-sized groups, it is recommended to use a different objective when striving for overall between-group similarity, e.g., the k-plus objective or the "average-diversity". The average diversity was introduced in version 0.8.6, and it is more useful if groups are not equal-sized. The average diversity normalizes the sum of intra-cluster distances by group size. If all groups are equal-sized, it is equivalent to the regular diversity. In the publication that introduces the anticlust package (Papenberg & Klau, 2021), we used the term "anticluster editing" to refer to the maximization of the diversity, because the reversed procedure - minimizing the diversity - is also known as "cluster editing".

The "dispersion" is the minimum distance between any two elements that are part of the same cluster; maximization of this objective ensures that any two elements within the same group are as dissimilar from each other as possible. Applications that require high within-group heterogeneity often require to maximize the dispersion. Oftentimes, it is useful to also consider the diversity and not only the dispersion; to optimize both objectives at the same time, see the function bicriterion_anticlustering.

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" or objective = "dispersion" is used, the Euclidean distance is computed as the basic unit of the objectives. If a different measure of dissimilarity is preferred, you may pass a self-generated dissimilarity 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 (see Examples).

Algorithms for anticlustering

By default, a heuristic method is employed for anticlustering: the exchange method (method = "exchange"). First, elements are randomly assigned to anticlusters (It is also possible to explicitly specify the initial assignment using the argument K; in this case, K has length nrow(x).) Based on the initial assignment, elements are systematically swapped between anticlusters in such a way that each swap improves the objective value. For an element, each possible swap with elements in other clusters is simulated; then, the one swap is performed that improves the objective the most, but a swap is only conducted if there is an improvement at all. This swapping procedure is repeated for each element. When using method = "local-maximum", the exchange method does not terminate after the first iteration over all elements; instead, the swapping continues until a local maximum is reached. This method corresponds to the algorithm "LCW" by Weitz and Lakshminarayanan (1998). This means that after the exchange process has been conducted once for each data point, the algorithm restarts with the first element and proceeds to conduct exchanges until the objective cannot be improved.

When setting `preclustering = TRUE`, only the `K - 1` most similar elements serve as exchange partners for each element, which can speed up the optimization (more information on the preclustering heuristic follows below). If the `categories` argument is used, only elements having the same value in `categories` serve as exchange partners.

Using `method = "brusco"` implements the local bicriterion iterated local search (BILS) heuristic by Brusco et al. (2020) and returns the partition that best optimized either the diversity or the dispersion during the optimization process. The function `bicriterion_anticlustering` can also be used to run the algorithm by Brusco et al., but it returns multiple partitions that approximate the optimal pareto efficient set according to both objectives (diversity and dispersion). Thus, to fully utilize the BILS algorithm, use the function `bicriterion_anticlustering`.

Optimal anticlustering

Usually, heuristics are employed to tackle anticlustering problems, and their performance is generally very satisfying. However, heuristics do not investigate all possible group assignments and therefore do not (necessarily) find the "globally optimal solution", i.e., a partitioning that has the best possible value with regard to the objective that is optimized. Enumerating all possible partitions in order to find the best solution, however, quickly becomes impossible with increasing N, and therefore it is not possible to find a global optimum this way. Because all anticlustering problems considered here are also NP-hard, there is also no (known) clever algorithm that might identify the best solution without considering all possibilities - at least in the worst case. Integer linear programming (ILP) is an approach for tackling NP hard problems that nevertheless tries to be clever when finding optimal solutions: It does not necessarily enumerate all possibilities but is still guaranteed to return the optimal solution. Still, for NP hard problems such as anticlustering, ILP methods will also fail at some point (i.e., when N increases).

`anticlust` implements optimal solution algorithms via integer linear programming. In order to use the ILP methods, set `method = "ilp"`. The integer linear program optimizing the diversity was described in Papenberg & Klau, (2021; (8) - (12)). It can also be used to optimize the k-means and k-plus objectives, but you actually have to use the function `optimal_anticlustering` for these objectives. The documentation of the function `optimal_dispersion` and `optimal_anticlustering` contain more information on the optimal anticlustering algorithms.

Categorical variables

There are two ways to balance categorical variables among anticlusters (also see the package vignette "Using categorical variables with anticlustering"). The first way is to treat them as "hard constraints" via the argument `categories` (see Papenberg & Klau, 2021). If done so, balancing the categorical variable is accomplished via `categorical_sampling` through a stratified split before the anticlustering optimization. After that, the balance is never changed when the algorithm runs (hence, it is a "hard constraint"). When `categories` has multiple columns (i.e., there are multiple categorical variables), each combination of categories is treated as a distinct category by the exchange method (i.e., the multiple columns are "merged" into a single column). This behaviour may lead to less than optimal results on the level of each single categorical variable. In this case, it may be useful to treat the categorical variables as part of the numeric data, i.e., the first argument `x` via binary coding (e.g. using `categories_to_binary`). The examples show how to do this when using the bicriterion algorithm by Bruso et al. Using the argument `categories` is only available for the classical exchange procedures, that is, for `method = "exchange"` and `method = "local-maximum"`.

Anticlustering with constraints

Versions 0.8.6 and 0.8.7 of anticlust introduced the possibility to induce cannot-link and must-link constraints with anticlustering with the arguments `cannot_link` and `must_link`, respectively.

Cannot-link constraints ensure that pairs of items are assigned to different clusters. They are given as a 2-column matrix, where each row has the indices of two elements, which must not be assigned to the same cluster. It is possible that a set of cannot-link constraints cannot be fulfilled. To verify whether the constraints cannot be fulfilled (and to actually assign elements while respecting the constraints), a graph coloring algorithm algorithm is used. This algorithm is is actually the same method as used in `optimal_dispersion`. The graph coloring algorithm uses an ILP solver and it greatly profits (that is, it may be much faster) from the Rsymphony package, which is not installed as a necessary dependency with anticlust. It is therefore recommended to manually install the Rsymphony package, which is then automatically selected as solver when using the `must_link` argument. If you have access to the gurobi solver and have the gurobi R package installed, it will be selected as solver (which is even faster than Symphony).

Must-link constraints are passed as a single vector of length `nrow(x)`. Positions that have the same numeric index are assigned to the same anticluster (if the constraints can be fulfilled). When including must-link constraints, `method = "2PML"` performs a specialized search heuristic that potentially yields better results than `method = "local-maximum"`. The must-link functionality and the 2PML algorithm was introduced in Papenberg et al. (2025).

The examples illustrate the usage of the `must_link` and `cannot_link` arguments. Currently, the different kinds of constraints (arguments `must_link`, `cannot_link`, and `categories`) cannot be used together, but this may change in future versions.

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 (2021; (8) - (10), (12) - (13)). Note that when combining preclustering restrictions with `method = "ilp"`, the anticlustering result is no longer guaranteed to be globally optimal, but only optimal given the preclustering restrictions.

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

Brusco, M. J., Cradit, J. D., & Steinley, D. (2020). Combining diversity and dispersion criteria for anticlustering: A bicriterion approach. British Journal of Mathematical and Statistical Psychology, 73, 275-396. https://doi.org/10.1111/bmsp.12186

Papenberg, M., & Klau, G. W. (2021). Using anticlustering to partition data sets into equivalent parts. Psychological Methods, 26(2), 161–174. https://doi.org/10.1037/met0000301.

Papenberg, M. (2024). K-plus Anticlustering: An Improved k-means Criterion for Maximizing Between-Group Similarity. British Journal of Mathematical and Statistical Psychology, 77(1), 80-102. https://doi.org/10.1111/bmsp.12315

Papenberg, M., Wang, C., Diop, M., Bukhari, S. H., Oskotsky, B., Davidson, B. R., Vo, K. C., Liu, B., Irwin, J. C., Combes, A., Gaudilliere, B., Li, J., Stevenson, D. K., Klau, G. W., Giudice, L. C., Sirota, M., & Oskotsky, T. T. (2025). Anticlustering for sample allocation to minimize batch effects. bioRxiv. https://doi.org/10.1101/2025.03.03.641320

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

Weitz, R. R., & Lakshminarayanan, S. (1998). An empirical comparison of heuristic methods for creating maximally diverse groups. Journal of the Operational Research Society, 49(6), 635-646. https://doi.org/10.1057/palgrave.jors.2600510

Examples

```
# Use default method ("exchange") and the default diversity criterion, also include
# a categorical variable via argument `categories`:
anticlusters <- anticlustering(
  schaper2019[, 3:6],
  K = 3,
  categories = schaper2019$room
)
# Compare feature means and standard deviations by anticluster
mean_sd_tab(schaper2019[, 3:6], anticlusters)
# Verify that the "room" is balanced across anticlusters:
table(anticlusters, schaper2019$room)

# Use multiple starts of the algorithm to improve the objective and
# optimize the k-means criterion ("variance")
anticlusters <- anticlustering(
  schaper2019[, 3:6],
  objective = "variance",
  K = 3,
  categories = schaper2019$room,
  method = "local-maximum", # better search algorithm
  repetitions = 20 # multiple restarts of the algorithm
)
# Compare means and standard deviations by anticluster
mean_sd_tab(schaper2019[, 3:6], anticlusters)
```

```
# Use different group sizes and optimize the extended k-means
# criterion ("kplus")
anticlusters <- anticlustering(
  schaper2019[, 3:6],
  objective = "kplus",
  K = c(24, 24, 48),
  categories = schaper2019$room,
  repetitions = 20,
  method = "local-maximum",
  standardize = TRUE # ususally recommended
)

# Use cannot_link constraints: Element 1 must not be linked with elements 2 to 10:
cl_matrix <- matrix(c(rep(1, 9), 2:10), ncol = 2)
cl <- anticlustering(
  schaper2019[, 3:6],
  K = 10,
  cannot_link = cl_matrix
)
all(cl[1] != cl[2:10])

# Use cannot_link constraints: Element 1 must be linked with elements 2 to 10.
# Element 11 must be linked with elements 12-20.
must_link <- rep(NA, nrow(schaper2019))
must_link[1:10] <- 1
must_link[11:20] <- 2
cl <- anticlustering(
  schaper2019[, 3:6],
  K = 3,
  must_link = must_link
)
cl[1:10]
cl[11:20]

# Use the heuristic by Brusco et al. (2020) for k-plus anticlustering
# Include categorical variable as part of the optimization criterion rather
# than the argument categories!
anticlusters <- anticlustering(
  cbind(
    kplus_moment_variables(schaper2019[, 3:6], 2),
    categories_to_binary(schaper2019$room)
  ),
  objective = "variance", # k-plus anticlustering because of the input above!
  K = 3,
  repetitions = 20,
  method = "brusco"
)

mean_sd_tab(schaper2019[, 3:6], anticlusters)
table(anticlusters, schaper2019$room)
```

`balanced_clustering` *Create balanced clusters of equal size*

Description

Create balanced clusters of equal size

Usage

```
balanced_clustering(x, K, method = "centroid", solver = 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 clusters should be created.
`method`	One of "centroid" or "ilp". See Details.
`solver`	Optional. The solver used to obtain the optimal method if `method = "ilp"`. Currently supports "glpk" and "symphony". Is ignored for `method = "centroid"`.

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 `method = "ilp"` uses a "solver" to optimize the clustering objective. See `optimal_anticlustering` for an overview of the solvers that are available.

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. (2021). Using anticlustering to partition data sets into equivalent parts. Psychological Methods, 26(2), 161–174. 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)
```

---

bicriterion_anticlustering

Bicriterion iterated local search heuristic

Description

This function implements the bicriterion algorithm BILS for anticlustering by Brusco et al. (2020; <doi:10.1111/bmsp.12186>). The description of their algorithm is given in Section 3 of their paper (in particular, see the Pseudocode in their Figure 2). As of anticlust version 0.8.6, this function also includes some extensions to the BILS algorithm that are implemented through the optional arguments `dispersion_distances`, `average_diversity`, `init_partitions`, and `return`. If these arguments are not changed, the function performs the "vanilla" BILS as described in Brusco et al.

Usage

```
bicriterion_anticlustering(
  x,
  K,
  R = NULL,
  W = c(1e-06, 1e-05, 1e-04, 0.001, 0.01, 0.1, 0.5, 0.99, 0.999, 0.999999),
  Xi = c(0.05, 0.1),
  dispersion_distances = NULL,
  average_diversity = FALSE,
  init_partitions = NULL,
  return = "paretoset"
)
```

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) A vector describing the size of each group, or (b) a vector of length `nrow(x)` describing how elements are assigned to anticlusters before the optimization starts.
R	The desired number of restarts for the algorithm. By default, both phases (MBPI + ILS) of the algorithm are performed once. See details.
W	Optional argument, a vector of weights defining the relative importance of dispersion and diversity (0 <= W <= 1). See details.
Xi	Optional argument, specifies probability of swapping elements during the iterated local search. See examples.
dispersion_distances	A distance matrix used to compute the dispersion if the dispersion should not be computed on the basis of argument `x`.
average_diversity	Boolean. Compute the diversity not as a global sum across all pairwise within-group distances, but as the sum of the average of within-group distances.
init_partitions	A matrix of initial partitions (rows = partitions; columns = elements) that serve as starting partitions during the iterations of the first phase of the BILS (i.e., the MBPI). If not passed, a new random partition is generated at the start of each iteration (which is the default behaviour).
return	Either "paretoset" (default), "best-diversity", "best-average-diversity", "best-dispersion". See below.

Details

The bicriterion algorithm by Brusco et al. (2020) aims to simultaneously optimize two anticlustering criteria: the `diversity_objective` and the `dispersion_objective`. It returns a list of partitions that approximate the pareto set of efficient solutions across both criteria. By considering both the diversity and dispersion, this algorithm is well-suited for maximizing overall within-group heterogeneity. To select a partition among the approximated pareto set, it is reasonable to plot the objectives for each partition (see Examples).

The arguments `R`, `W` and `Xi` are named for consistency with Brusco et al. (2020). The argument `K` is used for consistency with other functions in anticlust; Brusco et al. used 'G' to denote the number of groups. However, note that `K` can not only be used to denote the number of equal-sized groups, but also to specify group sizes, as in `anticlustering`.

This function implements the combined bicriterion algorithm BILS, which consists of two phases: The multistart bicriterion pairwise interchange heuristic (MBPI, which is a local maximum search heuristic similar to `method = "local-maximum"` in `anticlustering`), and the iterated local search (ILS), which is an improvement procedure that tries to overcome local optima. The argument `R` denotes the number of restarts of the two phases of the algorithm. If `R` has length 1, half of the repetitions perform the first phase MBPI), the other half perform the ILS. If `R` has length 2, the first entry indicates the number of restarts of MBPI the second entry indicates the number of restarts of ILS. The argument `W` denotes the relative weight given to the diversity and dispersion criterion in a given run of the search heuristic. In each run, the a weight is randomly selected from the vector `W`. As default values, we use the weights that Brusco et al. used in their analyses. All values in `W` have to be in [0, 1]; larger values indicate that diversity is more important, whereas smaller values indicate that dispersion is more important; `w = .5` implies the same weight for both criteria. The argument `Xi` is the probability that an element is swapped during the iterated local search (specifically, Xi has to be a vector of length 2, denoting the range of a uniform distribution from which the probability of swapping is selected). For `Xi`, the default is selected consistent with the analyses by Brusco et al.

If the data input `x` is a feature matrix (that is: each row is a "case" and each column is a "variable"), a matrix of the Euclidean distances is computed as input to the algorithm. If a different measure of dissimilarity is preferred, you may pass a self-generated dissimilarity matrix via the argument `x`. The argument `dispersion_distances` can additionally be used if the dispersion should be computed on the basis of a different distance matrix.

If multiple `init_partitions` are given, ensure that each partition (i.e., each row of`init_partitions`) has the exact same output of `table`.

Value

By default, a `matrix` of anticlustering partitions (i.e., the approximated pareto set). Each row corresponds to a partition, each column corresponds to an input element. If the argument `return` is set to either "best-diversity", "best-average-diversity", or "best-dispersion", it only returns one partition (as a vector), that maximizes the respective objective.

Note

For technical reasons, the pareto set returned by this function has a limit of 500 partitions. Usually however, the algorithm usually finds much fewer partitions. There is one following exception: We do not recommend to use this method when the input data is one-dimensional where the algorithm

may identify too many equivalent partitions causing it to run very slowly (see section 5.6 in Breuer, 2020).

Author(s)

Martin Breuer <M.Breuer@hhu.de>, Martin Papenberg <martin.papenberg@hhu.de>

References

Brusco, M. J., Cradit, J. D., & Steinley, D. (2020). Combining diversity and dispersion criteria for anticlustering: A bicriterion approach. British Journal of Mathematical and Statistical Psychology, 73, 275-396. https://doi.org/10.1111/bmsp.12186

Breuer (2020). Using anticlustering to maximize diversity and dispersion: Comparing exact and heuristic approaches. Bachelor thesis.

Examples

```
# Generate some random data
M <- 3
N <- 80
K <- 4
data <- matrix(rnorm(N * M), ncol = M)

# Perform bicriterion algorithm, use 200 repetitions
pareto_set <- bicriterion_anticlustering(data, K = K, R = 200)

# Compute objectives for all solutions
diversities_pareto <- apply(pareto_set, 1, diversity_objective, x = data)
dispersions_pareto <- apply(pareto_set, 1, dispersion_objective, x = data)

# Plot the pareto set
plot(
  diversities_pareto,
  dispersions_pareto,
  col = "blue",
  cex = 2,
  pch = as.character(1:NROW(pareto_set))
)

# Get some random solutions for comparison
rnd_solutions <- t(replicate(n = 200, sample(pareto_set[1, ])))

# Compute objectives for all random solutions
diversities_rnd <- apply(rnd_solutions, 1, diversity_objective, x = data)
dispersions_rnd <- apply(rnd_solutions, 1, dispersion_objective, x = data)

# Plot random solutions and pareto set. Random solutions are far away
# from the good solutions in pareto set
plot(
  diversities_rnd, dispersions_rnd,
  col = "red",
  xlab = "Diversity",
```

```
  ylab = "Dispersion",
  ylim = c(
    min(dispersions_rnd, dispersions_pareto),
    max(dispersions_rnd, dispersions_pareto)
  ),
  xlim = c(
    min(diversities_rnd, diversities_pareto),
    max(diversities_rnd, diversities_pareto)
  )
)

# Add approximated pareto set from bicriterion algorithm:
points(diversities_pareto, dispersions_pareto, col = "blue", cex = 2, pch = 19)
```

`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 `anticlustering` with argument 'categories', but without optimizing a clustering objective. 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)
```

```
# Unequal sized groups
groups <- categorical_sampling(categories, K = c(24, 24, 48))
table(groups, categories)

# Heavily unequal sized groups, is harder to balance the groups
groups <- categorical_sampling(categories, K = c(51, 19, 26))
table(groups, categories)
```

`categories_to_binary` *Get binary representation of categorical variables*

Description

Get binary representation of categorical variables

Usage

```
categories_to_binary(categories, use_combinations = FALSE)
```

Arguments

`categories`	A vector, data.frame or matrix representing one or several categorical variables
`use_combinations`	Logical, should the output also include columns representing the combination / interaction of the categories (defaults to `FALSE`).

Details

The conversion of categorical variables to binary variables is done via `model.matrix`. Since version 0.8.9, each category of a categorical variable is coded by a separate variable. So this is not 'dummy' coding, which is often used to encode predictors in statistical analysis. Dummy coding uses a reference category that has only zeros for each variable, while all other categories consist of a 1 and otherwise zeros. This implies that there is a different distance to the reference category than among the other categories, which is unwarranted in anticlustering.

This function can be used to include categorical variables as part of the optimization criterion in anticlustering, rather than including them as hard constraints as done when using the argument `categories` in `anticlustering` (or `fast_anticlustering`). This way, categorical variables are treated as numeric variables, which can be useful when there are several categorical variables or when the group sizes are unequal (or both). See examples. Please see the vignette 'Using categorical variables with anticlustering' for more information on this approach.

Value

A matrix encoding the categorical variable(s) in binary form.

Author(s)

Martin Papenberg <martin.papenberg@hhu.de>

References

Papenberg, M. (2024). K-plus Anticlustering: An Improved k-means Criterion for Maximizing Between-Group Similarity. British Journal of Mathematical and Statistical Psychology, 77(1), 80–102. https://doi.org/10.1111/bmsp.12315

Examples

```
# How to encode a categorical variable with three levels:
unique(iris$Species)
categories_to_binary(iris$Species)[c(1, 51, 101), ]

# Use Schaper data set for anticlustering example
data(schaper2019)
features <- schaper2019[, 3:6]
K <- 3
N <- nrow(features)

# - Generate data input for k-means anticlustering -
# We conduct k-plus anticlustering by first generating k-plus variables,
# and also include the categorical variable as "numeric" input for the
# k-means optimization (rather than as input for the argument \code{categories})

input_data <- cbind(
  kplus_moment_variables(features, T = 2),
  categories_to_binary(schaper2019$room)
)

kplus_groups <- anticlustering(
  input_data,
  K = K,
  objective = "variance",
  method = "local-maximum",
  repetitions = 10
)
mean_sd_tab(features, kplus_groups)
table(kplus_groups, schaper2019$room) # argument categories was not used!
```

---

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. (2020). Combining diversity and dispersion criteria for anticlustering: A bicriterion approach. British Journal of Mathematical and Statistical Psychology, 73, 275-396. 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. (2020). Combining diversity and dispersion criteria for anticlustering: A bicriterion approach. British Journal of Mathematical and Statistical Psychology, 73, 275-396. https://doi.org/10.1111/bmsp.12186

Papenberg, M., & Klau, G. W. (2021). Using anticlustering to partition data sets into equivalent parts. Psychological Methods, 26(2), 161–174. 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

Increasing the speed of (k-means / k-plus) anticlustering by (1) conducting fewer exchanges during the optimization and (2) using an alternative formulation of the k-means objective. Makes anticlustering applicable to quite large data sets.

Usage

```
fast_anticlustering(
  x,
  K,
  k_neighbours = Inf,
  categories = NULL,
  exchange_partners = 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. Alternatively: (a) A vector describing the size of each group, or (b) a vector of length `nrow(x)` describing how elements are assigned to anticlusters before the optimization starts.
`k_neighbours`	The number of nearest neighbours that serve as exchange partner for each element. See details.
`categories`	A vector, data.frame or matrix representing one or several categorical constraints.
`exchange_partners`	Optional argument. A list of length `NROW(x)` specifying for each element the indices of the elements that serve as exchange partners. If used, this argument overrides the `k_neighbours` argument. See examples.

Details

This function was created to make anticlustering applicable to large data sets (e.g., several 100,000 elements). It optimizes the k-means objective because computing all pairwise distances as is done when optimizing the "diversity" (i.e., the default in `anticlustering`) is not feasible for very large data sets (for about N > 20000 on my personal computer). Using `fast_anticlustering` for k-plus anticlustering is also possible by applying `kplus_moment_variables` on the input (and possibly by using the argument `exchange_partners`, see Examples).

The function `fast_anticlustering` employs a speed-optimized exchange method, which is basically equivalent to `method = "exchange"` in `anticlustering`, but may reduce the number of

exchanges that are investigated for each input element. 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. If `k_neighbours` is set differently (which is usually recommended when running this function), the default behaviour is to generate exchange partners using a nearest neighbour search (using the function `nn2` from the `RANN` package). Using more exchange partners can improve the quality of the results, but also increase run time. Note that for very large data sets, anticlustering generally becomes "easier" (even a random split may yield satisfactory results), so using few exchange partners is usually not a problem.

It is possible to specify custom exchange partners using the argument `exchange_partners` instead of relying on the default nearest neighbour search. When using `exchange_partners`, it is not necessary that each element has the same number of exchange partners; this is why the argument `exchange_partners` has to be a `list` instead of a `matrix` or `data.frame`. Exchange partners can for example be generated by `generate_exchange_partners` (see Examples), but a custom list may also be used. Note that categorical constraints induced via `categories` may not be respected during the optimization if the `exchange_partners` argument allows exchanges between members of different categories, so care must be taken when combining the arguments `exchange_partners` and `categories`.

In `anticlustering(..., objective = "variance")`, the run time of computing the k-means objective is in O(M N), where N is the total number of elements and M is the number of variables. This is because the variance is computed as the sum of squared distances between all data points and their cluster centers. The function `fast_anticlustering` uses a different - but equivalent - formulation of the k-means objective, where the re-computation of the objective only depends and M but no longer on N. In particular, this variant of k-means anticlustering minimizes the weighted sum of squared distances between cluster centroids and the overall data centroid; the distances between all individual data points and their cluster center are not computed (Späth, 1986). Using the different objective formulation reduces the run time by an order of magnitude and makes k-means anticlustering applicable to very large data sets (even in the millions). For a fixed number of exchange partners (specified using the argument `k_neighbours`), the approximate run time of `fast_anticlustering` is in O(M N). The algorithm `method = "exchange"` in `anticlustering` with `objective = "variance"` has a run time of O(M N^3). Thus, `fast_anticlustering` can improve the run time by two orders of magnitude as compared to the standard exchange algorithm. The nearest neighbour search, which is done in the beginning usually does not strongly contribute to the overall run time. It is nevertheless possible to suppress the nearest neighbour search by using the `exchange_partners` argument.

When setting the `categories` argument, exchange partners (i.e., nearest neighbours) will be generated from the same category. Note that when `categories` has multiple columns, each combination of these categories is treated as a distinct category by the exchange method. You can also use `categories_to_binary` to potentially improve results for several categorical variables, instead of using the argument `categories`.

Author(s)

Martin Papenberg <martin.papenberg@hhu.de>

References

Papenberg, M., & Klau, G. W. (2021). Using anticlustering to partition data sets into equivalent parts. Psychological Methods, 26(2), 161–174. https://doi.org/10.1037/met0000301.

Papenberg, M. (2024). K-plus Anticlustering: An Improved k-means Criterion for Maximizing Between-Group Similarity. British Journal of Mathematical and Statistical Psychology, 77(1), 80–102. https://doi.org/10.1111/bmsp.12315

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

See Also

anticlustering

kplus_moment_variables

categories_to_binary

variance_objective

generate_exchange_partners

Examples

```
## Use fewer or more exchange partners to adjust speed (vs. quality tradeoff)
features <- iris[, - 5]
N <- nrow(features)
init <- sample(rep_len(1:3, N)) # same starting point for all calls:
groups1 <- fast_anticlustering(features, K = init) # default: all exchanges
groups2 <- fast_anticlustering(features, K = init, k_neighbours = 20)
groups3 <- fast_anticlustering(features, K = init, k_neighbours = 2)

variance_objective(features, groups1)
variance_objective(features, groups2)
variance_objective(features, groups3)

# K-plus anticlustering is straight forward when sticking with the default
# for k_neighbours
kplus_anticlusters <- fast_anticlustering(
  kplus_moment_variables(features, T = 2),
  K = 3
)
mean_sd_tab(features, kplus_anticlusters)

# Some care is needed when applying k-plus using with this function
# while using a reduced number of exchange partners generated in the
# nearest neighbour search. Then we:
# 1) Use kplus_moment_variables() on the numeric input
# 2) Generate custom exchange_partners because otherwise nearest
#    neighbours are internally selected based on the extended k-plus
#    variables returned by kplus_moment_variables()
#    (which does not really make sense)
kplus_anticlusters <- fast_anticlustering(
  kplus_moment_variables(features, T = 2),
  K = 3,
 exchange_partners = generate_exchange_partners(120, features = features, method = "RANN")
 )
mean_sd_tab(features, kplus_anticlusters)
```

```
# Or we use random exchange partners:
kplus_anticlusters <- fast_anticlustering(
  kplus_moment_variables(features, T = 2),
  K = 3,
 exchange_partners = generate_exchange_partners(120, N = nrow(features), method = "random")
)
mean_sd_tab(features, kplus_anticlusters)


# Working on several 1000 elements is very fast (Here n = 10000, m = 2)
data <- matrix(rnorm(10000 * 2), ncol = 2)
start <- Sys.time()
groups <- fast_anticlustering(data, K = 5, k_neighbours = 5)
Sys.time() - start
```

`generate_exchange_partners` *Get exchange partners for fast_anticlustering()*

Description

Get exchange partners for fast_anticlustering()

Usage

```
generate_exchange_partners(
  n_exchange_partners,
  N = NULL,
  features = NULL,
  method = "random",
  categories = NULL
)
```

Arguments

`n_exchange_partners`	The number of exchange partners per element
`N`	The number of elements for which exchange partners; can be `NULL` if `features` is passed (it is ignored if `features` is passed).
`features`	The features for which nearest neighbours are sought if `method = "RANN"`. May be NULL if random exchange partners are generated.
`method`	Currently supports "random" (default), "RANN" and "restricted_random". See details.
`categories`	A vector, data.frame or matrix representing one or several categorical constraints.

Details

The method = "RANN" generates exchange partners using a nearest neighbour search via nn2 from the RANN package; methode = "restricted_random" generates random exchange partners but ensures that for each element, no duplicates are generated and that the element itself does not occur as exchange partner (this is the slowest method, and I would not recommend it for large N); method = "random" (default) does not impose these restrictions and generates unrescricted random partners (it may therefore generate duplicates and the element itself as exchange partner).

When setting the categories argument and using method = "RANN", exchange partners (i.e., nearest neighbours) will be generated from the same category; methode = "restricted_random" will also adhere to categorical constraints induced via categories (i.e. each element only receives exchange partners from the same category as itself); methode = "random" cannot incoorporate categorical restrictions.

Value

A list of length N. Is usually used as input to the argument exchange_partners in fast_anticlustering. Then, the i'th element of the list contains the indices of the exchange partners that are used for the i'th element.

Examples

```
# Restricted random method generates no duplicates per element and cannot return
# the element itself as exchange partner
generate_exchange_partners(5, N = 10, method = "restricted_random")
# "random" simply randomizes with replacement and without restrictions
# (categorical restrictions are also not possible; is much faster for large data sets)
generate_exchange_partners(5, N = 10, method = "random")
# May return less than 5 exchange partners if there are not enough members
# of the same category:
generate_exchange_partners(
  5, N = 10,
  method = "restricted_random",
  categories = cbind(schaper2019$room, schaper2019$frequency)
)
# using nearest neighbour search (unlike RANN::nn2, this does not
# return the ID of the element itself as neighbour)
generate_exchange_partners(5, features = schaper2019[, 3:5], method = "RANN")[1:3]
# compare with RANN directly:
RANN::nn2(schaper2019[, 3:5], k = 6)$nn.idx[1:3, ] # note k = 6
```

---

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. (2021). Using anticlustering to partition data sets into equivalent parts. Psychological Methods, 26(2), 161–174. 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)
```

`kplus_anticlustering` *K-plus anticlustering*

Description

Perform anticlustering using the k-plus objective to maximize between-group similarity. This function implements the k-plus anticlustering method described in Papenberg (2024; <doi:10.1111/bmsp.12315>).

Usage

```
kplus_anticlustering(
  x,
  K,
  variance = TRUE,
  skew = FALSE,
  kurtosis = FALSE,
  covariances = FALSE,
  T = NULL,
  standardize = TRUE,
  ...
)
```

Arguments

x	A feature matrix where rows correspond to elements and columns correspond to variables (a single numeric variable can be passed as a vector).
K	How many anticlusters should be created. Alternatively: (a) A vector describing the size of each group, or (b) a vector of length `nrow(x)` describing how elements are assigned to anticlusters before the optimization starts.
variance	Boolean: Should the k-plus objective include a term to maximize between-group similarity with regard to the variance? (Default = TRUE)
skew	Boolean: Should the k-plus objective include a term to maximize between-group similarity with regard to skewness? (Default = FALSE)
kurtosis	Boolean: Should the k-plus objective include a term to maximize between-group similarity with regard to kurtosis? (Default = FALSE)
covariances	Boolean: Should the k-plus objective include a term to maximize between-group similarity with regard to covariance structure? (Default = FALSE)
T	Optional argument: An integer specifying how many distribution moments should be equalized between groups.
standardize	Boolean. If `TRUE`, the data is standardized through a call to `scale` before the optimization starts. Defaults to TRUE. See details.
...	Arguments passed down to `anticlustering`. All of the arguments are supported except for `objective`.

Details

This function implements the unweighted sum approach for k-plus anticlustering. Details are given in Papenberg (2024).

The optional argument `T` denotes the number of distribution moments that are considered in the anticlustering process. For example, `T = 4` will lead to similar means, variances, skew and kurtosis. For the first four moments, it is also possible to use the boolean convenience arguments `variance`, `skew` and `kurtosis`; the mean (the first moment) is always included and cannot be "turned off". If the argument `T` is used, it overrides the arguments `variance`, `skew` and `kurtosis` (corresponding to the second, third and fourth moment), ignoring their values.

The `standardization` is applied to all original features and the additional k-plus features that are appended to the data set in order to optimize the k-plus criterion. When using standardization, all criteria such as means, variances and skewness receive a comparable weight during the optimization. It is usually recommended not to change the default setting `standardization = TRUE`.

This function can use any arguments that are also possible in `anticlustering` (except for 'objective' because the objective optimized here is the k-plus objective; to use a different objective, call `anticlustering` directly). Any arguments that are not explicitly changed here (i.e., `standardize = TRUE`) receive the default given in `anticlustering` (e.g., `method = "exchange"`.)

References

Papenberg, M. (2024). K-plus Anticlustering: An Improved k-means Criterion for Maximizing Between-Group Similarity. British Journal of Mathematical and Statistical Psychology, 77(1), 80–102. https://doi.org/10.1111/bmsp.12315

Examples

```
# Generate some data
N <- 180
M <- 4
features <- matrix(rnorm(N * M), ncol = M)
# standard k-plus anticlustering: optimize similarity with regard to mean and variance:
cl <- kplus_anticlustering(features, K = 3, method = "local-maximum")
mean_sd_tab(features, cl)
# Visualize an anticlustering solution:
plot(features, col = palette()[2:4][cl], pch = c(16:18)[cl])

# Also optimize with regard to skewness and kurtosis
cl2 <- kplus_anticlustering(
  features,
  K = 3,
  method = "local-maximum",
  skew = TRUE,
  kurtosis = TRUE
)

# The following two calls are equivalent:
init_clusters <- sample(rep_len(1:3, nrow(features)))
# 1.
x1 <- kplus_anticlustering(
  features,
  K = init_clusters,
  variance = TRUE,
  skew = TRUE
)
# 2.
x2 <- kplus_anticlustering(
  features,
  K = init_clusters,
  T = 3
)
```

```
# Verify:
all(x1 == x2)
```

`kplus_moment_variables` *Compute k-plus variables*

Description

Compute k-plus variables

Usage

```
kplus_moment_variables(x, T, standardize = TRUE)
```

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.
`T`	The number of distribution moments for which variables are generated.
`standardize`	Logical, should all columns of the output be standardized (defaults to TRUE).

Details

The k-plus criterion is an extension of the k-means criterion (i.e., the "variance", see `variance_objective`). In `kplus_anticlustering`, equalizing means and variances simultaneously (and possibly additional distribution moments) is accomplished by internally appending new variables to the data input `x`. When using only the variance as additional criterion, the new variables represent the squared difference of each data point to the mean of the respective column. All columns are then included—in addition to the original data—in standard k-means anticlustering. The logic is readily extended towards higher order moments, see Papenberg (2024). This function gives users the possibility to generate k-plus variables themselves, which offers some additional flexibility when conducting k-plus anticlustering.

Value

A matrix containing all columns of `x` and all additional columns of k-plus variables. If `x` has M columns, the output matrix has M * T columns.

Author(s)

Martin Papenberg <martin.papenberg@hhu.de>

References

Papenberg, M. (2024). K-plus Anticlustering: An Improved k-means Criterion for Maximizing Between-Group Similarity. British Journal of Mathematical and Statistical Psychology, 77(1), 80–102. https://doi.org/10.1111/bmsp.12315

Examples

```
# Use Schaper data set for example
data(schaper2019)
features <- schaper2019[, 3:6]
K <- 3
N <- nrow(features)

# Some equivalent ways of doing k-plus anticlustering:

init_groups <- sample(rep_len(1:3, N))
table(init_groups)

kplus_groups1 <- anticlustering(
  features,
  K = init_groups,
  objective = "kplus",
  standardize = TRUE,
  method = "local-maximum"
)

kplus_groups2 <- anticlustering(
  kplus_moment_variables(features, T = 2), # standardization included by default
  K = init_groups,
  objective = "variance", # (!)
  method = "local-maximum"
)

# this function uses standardization by default unlike anticlustering():
kplus_groups3 <- kplus_anticlustering(
  features,
  K = init_groups,
  method = "local-maximum"
)

all(kplus_groups1 == kplus_groups2)
all(kplus_groups1 == kplus_groups3)
all(kplus_groups2 == kplus_groups3)
```

`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 intra-group 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_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)
```

`optimal_anticlustering` *Optimal ("exact") algorithms for anticlustering*

Description

Wrapper function that gives access to all optimal algorithms for anticlustering that are available in anticlust.

Usage

```
optimal_anticlustering(x, K, objective, solver = NULL, time_limit = 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 or alternatively: (a) A vector describing the size of each group (the latter currently only works for `objective = "dispersion"`).
`objective`	The anticlustering objective, can be "diversity", "variance", "kplus" or "dispersion".
`solver`	Optional. The solver used to obtain the optimal method. Currently supports "glpk", "symphony", "lpSolve" and "gurobi". See details.
`time_limit`	Time limit in seconds, given to the solver. Default is there is no time limit.

Details

This is a wrapper for all optimal methods supported in anticlust (currently and in the future). As compared to `anticlustering`, it allows to specify the solver to obtain an optimal solution and it can be used to obtain optimal solutions for all supported anticlustering objectives (variance, diversity, k-plus, dispersion). For the objectives "variance", "diversity" and "kplus", the optimal ILP method in Papenberg and Klau (2021) is used, which maximizes the sum of all pairwise intra-cluster distances (given user specified number of clusters, for equal-sized clusters). To employ k-means anticlustering (i.e. set `objective = "variance"`), the squared Euclidean distance is used. For k-plus anticlustering, the squared Euclidean distance based on the extended k-plus data matrix is used (see `kplus_moment_variables`). For the diversity (and the dispersion), the Euclidean distance is used by default, but any user-defined dissimilarity matrix is possible.

The dispersion is solved optimal using the approach described in optimal_dispersion.

The optimal methods make use of "solvers" that actually implement the algorithm for finding optimal solutions. The package anticlust supports three solvers:

- The default solver lpSolve (<https://sourceforge.net/projects/lpsolve/>).
- GNU linear programming kit (<http://www.gnu.org/software/glpk/>), available from the package Rglpk and requested using `solver = "glpk"`. The R package Rglpk has to be installed manually if this solver should be used.
- The Symphony solver (<https://github.com/coin-or/SYMPHONY>), available from the package Rsymphony and requested using `solver = "symphony"`. (The package Rsymphony has to be installed manually if this solver should be used).
- The commercial gurobi solver, see https://www.gurobi.com/downloads/.

For the maximum dispersion problem, it seems that the Symphony solver is fastest, while the lpSolve solver seems to be good for maximum diversity. However, note that in general the dispersion can be solved optimally for much larger data sets than the diversity.

If a `time_limit` is set and the function cannot find in the optimal objective in the given time, it will throw an error.

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>

Examples

```
data <- matrix(rnorm(24), ncol = 2)

# These calls are equivalent for k-means anticlustering:
optimal_anticlustering(data, K = 2, objective = "variance")
optimal_anticlustering(dist(data)^2, K = 2, objective = "diversity")

# These calls are equivalent for k-plus anticlustering:
optimal_anticlustering(data, K = 2, objective = "kplus")
optimal_anticlustering(dist(kplus_moment_variables(data, 2))^2, K = 2, objective = "diversity")
```

`optimal_dispersion` *Maximize dispersion for K groups*

Description

Maximize dispersion for K groups

Usage

```
optimal_dispersion(
  x,
  K,
  solver = NULL,
  max_dispersion_considered = NULL,
  min_dispersion_considered = NULL,
  npartitions = 1,
  time_limit = 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`	The number of groups or a vector describing the size of each group.
`solver`	Optional argument; currently supports "lpSolve", "glpk", "symphony", and "gurobi". See `optimal_anticlustering`.
`max_dispersion_considered`	Optional argument used for early stopping. If the dispersion found is equal to or exceeds this value, a solution having the previous best dispersion is returned.
`min_dispersion_considered`	Optional argument used for speeding up the algorithm computation. If passed, the dispersion is optimized starting from this value instead the global minimum distance.
`npartitions`	The number of groupings that are returned, each having an optimal dispersion value (defaults to 1).
`time_limit`	Time limit in seconds, given to the solver. Default is there is no time limit.

Details

The dispersion is the minimum distance between two elements within the same group. This function implements an optimal method to maximize the dispersion. If the data input `x` is a feature matrix and

not a dissimilarity matrix, the pairwise Euclidean distance is used. It uses the algorithm presented in Max Diekhoff's Bachelor thesis at the Computer Science Department at Heinrich Heine University Düsseldorf.

To find out which items are not allowed to be grouped in the same cluster for maximum dispersion, the algorithm sequentially builds instances of a graph coloring problem, using an integer linear programming (ILP) representation (also see Fernandez et al., 2013). It is possible to specify the ILP solver via the argument `solver` (See `optimal_anticlustering` for more information on this argument). Optimally solving the maximum dispersion problem is NP-hard for K > 2 and therefore computationally infeasible for larger data sets. For K = 3 and K = 4, it seems that this approach scales up to several 100 elements, or even > 1000 for K = 3 (at least when using the Symphony solver). For larger data sets, use the heuristic approaches in `anticlustering` or `bicriterion_anticlustering`. However, note that for K = 2, the optimal approach is usually much faster than the heuristics.

In the output, the element `edges` defines which elements must be in separate clusters in order to achieve maximum dispersion. All elements not listed here can be changed arbitrarily between clusters without reducing the dispersion. If the maximum possible dispersion corresponds to the minimum dispersion in the data set, the output elements `edges` and `groups` are set to `NULL` because all possible groupings have the same value of dispersion. In this case the output element `dispersions_considered` has length 1.

If a `time_limit` is set and the function cannot find in the optimal dispersion in the given time, it will throw an error.

Value

A list with four elements:

`dispersion`	The optimal dispersion
`groups`	An assignment of elements to groups (vector)
`edges`	A matrix of 2 columns. Each row contains the indices of elements that had to be investigated to find the dispersion (i.e., each pair of elements cannot be part of the same group in order to achieve maximum dispersion).
`dispersions_considered`	All distances that were tested until the dispersion was found.

Note

If the SYMPHONY solver is used, an unfortunate "message" is printed to the console when this function terminates:

sym_get_col_solution(): No solution has been stored!

This message is no reason to worry and instead is a direct result of the algorithm finding the optimal value for the dispersion. Unfortunately, this message is generated in the C code underlying the SYMPHONY library (via the printing function `printf`), which cannot be prevented in R.

Author(s)

Max Diekhoff

Martin Papenberg <martin.papenberg@hhu.de>

References

Diekhoff (2023). Maximizing dispersion for anticlustering. Retrieved from https://www.cs.hhu.de/fileadmin/redaktion/Fakult
Naturwissenschaftliche_Fakultaet/Informatik/Algorithmische_Bioinformatik/Bachelor-_Masterarbeiten/2831963_ba_ifo_A

Fernández, E., Kalcsics, J., & Nickel, S. (2013). The maximum dispersion problem. Omega, 41(4), 721–730. https://doi.org/10.1016/j.omega.2012.09.005

See Also

dispersion_objective anticlustering

Examples

```
N <- 30
M <- 5
K <- 3
data <- matrix(rnorm(N*M), ncol = M)
distances <- dist(data)

opt <- optimal_dispersion(distances, K = K)
opt

# Compare to bicriterion heuristic:
groups_heuristic <- anticlustering(
  distances,
  K = K,
  method = "brusco",
  objective = "dispersion",
  repetitions = 100
)
c(
  OPT = dispersion_objective(distances, opt$groups),
  HEURISTIC = dispersion_objective(distances, groups_heuristic)
)

# Different group sizes are possible:
table(optimal_dispersion(distances, K = c(15, 10, 5))$groups)

# Induce cannot-link constraints by maximizing the dispersion:
solvable <- matrix(1, ncol = 6, nrow = 6)
solvable[2, 1] <- -1
solvable[3, 1] <- -1
solvable[4, 1] <- -1
solvable <- as.dist(solvable)
solvable

# An optimal solution has to put item 1 in a different group than
# items 2, 3 and 4 -> this is possible for K = 2
optimal_dispersion(solvable, K = 2)$groups

# It no longer works when item 1 can also not be linked with item 5:
# (check out output!)
```

```
unsolvable <- as.matrix(solvable)
unsolvable[5, 1] <- -1
unsolvable <- as.dist(unsolvable)
unsolvable
optimal_dispersion(unsolvable, K = 2)
# But:
optimal_dispersion(unsolvable, K = c(2, 4)) # group sizes, not number of groups
```

`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))
```

`three_phase_search_anticlustering` *Three phase search with dynamic population size heuristic*

Description

This function implements the three phase search algorithm TPSPD for anticlustering by Yang et al. (2022; <doi.org/10.1016/j.ejor.2022.02.003>). The description of their algorithm is given in Section 2 of their paper (in particular, see the Pseudocode in Algorithm 1).

Usage

```
three_phase_search_anticlustering(
  x,
  K,
  N,
  objective = "diversity",
  number_iterations = 50,
  clusters = NULL,
```

```
  upper_bound = NULL,
  lower_bound = NULL,
  beta_max = 15,
  theta_max = NULL,
  theta_min = NULL,
  beta_min = NULL,
  eta_max = 3,
  alpha = 0.05
)
```

Arguments

x	The data input, as in anticlustering.
K	Number of anticlusters to be formed.
N	Number of elements.
objective	The anticlustering objective, can be "diversity" or "dispersion".
number_iterations	A number that defines how many times the steps in the search algorithm are repeated.
clusters	A vector of length K that specifies the number of elements each cluster can contain. If this vector is not NULL, the lower and upper bounds will be disregarded.
upper_bound	Maximum number of elements in each anticluster. By default, anticlusters are of equal size, calculated as the total number of items divided by the number of clusters.
lower_bound	Minimum number of elements in each anticluster. By default, anticlusters are of equal size, calculated as the total number of items divided by the number of clusters.
beta_max	The algorithm begins with a pool of random initial solutions of size beta_max. Over time, the size of the solution pool decreases linearly until it reaches beta_min.
theta_max	Parameter for the strength of undirected perturbation, which decreases linearly over time from theta_max to theta_min.
theta_min	Parameter for the strength of undirected perturbation, which decreases linearly over time from theta_max to theta_min.
beta_min	The minimum solution pool size the algorithm should reach before making a determination.
eta_max	Parameter that specifies how many times the steps in the direct perturbation are executed.
alpha	Parameter for weighing the discrimination of a slightly worse local optimal child solution.

Details

Details of the implementation of the algorithm can be found in the pseudocode of the paper Yang et al. (2022). However, we performed one change as compared to the original description of the algorithm: Instead of setting a time limit, we define the number of iterations the algorithm performs before terminating (via argument number_iterations).

Value

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

Author(s)

Hannah Hengelbrock <Hannah.Hengelbrock@hhu.de>, Martin Papenberg <martin.papenberg@hhu.de>

References

Xiao Yang et al. "A three-phase search approach with dynamic population size for solving the maximally diverse grouping problem". In: European Journal of Operational Research 302.3 (2022) <doi:10.1016/j.ejor.2022.02.003>

Examples

```
# Generate some random data
N <- 120
M <- 5
K <- 4
dat <- matrix(rnorm(N * M), ncol = M)
distances <- dist(dat)

# Perform three hase serach algorithm
result1 <- three_phase_search_anticlustering(dat, K, N)

# Compute objectives function
diversity_objective(distances, result1)

# Standard algorithm:
result2 <- anticlustering(distances, K=K, method="local-maximum", repetitions = 10)
diversity_objective(distances, result2)
```

---

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. (2021). Using anticlustering to partition data sets into equivalent parts. Psychological Methods, 26(2), 161–174. 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, solver = NULL)
```

Arguments

x	A N x N similarity matrix. Larger values indicate stronger agreement / similarity between a pair of data points
solver	Optional argument; if passed, has to be either "glpk" or "symphony". See details.

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. The function uses a "solver" to optimize the clustering objective. See optimal_anticlustering for an overview of the solvers that are available.

Value

An integer vector representing the cluster affiliation of each data point

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