Package ‘scclust’

May 22, 2017

Type Package

Title Size-Constrained Clustering

Version 0.1.1

Date 2017-05-21

Description Provides wrappers for 'scclust', a C library for computationally efficient size-constrained clustering with near-optimal performance. See <https://github.com/fsavje/scclust> for more information.

Depends R (>= 3.2.3), distances

Imports

Suggests testthat

NeedsCompilation yes

License GPL (>= 3)

LicenseNote The scclust package includes the scclust C library (distributed under the LGPLv2.1 license).

URL `https://github.com/fsavje/scclust-R`

BugReports `https://github.com/fsavje/scclust-R/issues`

Encoding UTF-8

RoxygenNote 6.0.1

Author Fredrik Savje [aut, cre],
Michael Higgins [aut],
Jasjeet Sekhon [aut]

Maintainer Fredrik Savje <`fredrik.savje@berkeley.edu`>

Repository CRAN

Date/Publication 2017-05-22 05:14:33 UTC

R topics documented:

scclust-package *scclust: Size-Constrained Clustering*

Description

The scclust package is an R wrapper for the scclust library. The package provides functions to construct near-optimal size-constrained clusterings. Subject to user-specified constraints on the size and composition of the clusters, scclust constructs a clustering so that within-cluster pair-wise distances are minimized.

Details

The main clustering function is sc_clustering. Statistics about clusters can be derived with the get_clustering_stats function. To check if a clustering satisfies some set of constraints, use check_clustering. Use scclust to construct a scclust object from an existing clustering.

Clusters can also be constructed with hierarchical_clustering. However, this function does not support type constraints and does not provide optimality guarantees. Its main use is to refine clusterings constructed with the sc_clustering function.

scclust was made with large data sets in mind, and it can cluster tens of millions of data points within minutes on an ordinary desktop computer.

See the package's website for more information: https://github.com/fsavje/scclust-R.

More information about the scclust library is found here: https://github.com/fsavje/scclust.

Bug reports and suggestions are greatly appreciated. They are best reported here: https://github.com/fsavje/scclust-R/issues.

References

Higgins, Michael J., Fredrik Sävje and Jasjeet S. Sekhon (2016), 'Improving massive experiments with threshold blocking', *Proceedings of the National Academy of Sciences*, **113:27**, 7369–7376. http://www.pnas.org/lookup/doi/10.1073/pnas.1510504113

Sävje, Fredrik and Michael J. Higgins and Jasjeet S. Sekhon (2017), 'Generalized Full Matching', arXiv 1703.03882. https://arxiv.org/abs/1703.03882

check_clustering *Check clustering constraints*

Description

check_clustering checks whether a clustering satisfies constraints on the size and composition of the clusters.

Usage

```
check_clustering(clustering, size_constraint = NULL, type_labels = NULL,
  type_constraints = NULL, primary_data_points = NULL)
```

Arguments

clustering a scclust object containing a non-empty clustering.

size_constraint
an integer with the required minimum cluster size. If NULL, only the type constraints will be checked.

type_labels a vector containing the type of each data point. May be NULL when type_constraints is NULL.

type_constraints
a named integer vector containing type-specific size constraints. If NULL, only the overall constraint will be checked.

primary_data_points
a vector specifying primary data points, either by point indices or with a logical vector of length equal to the number of points. check_clustering checks so all primary data points are assigned to a cluster. NULL indicates that no such check should be done.

Value

Returns TRUE if clustering satisfies the constraints, and FALSE if it does not. Throws an error if clustering is an invalid instance of the scclust class.

See Also

See sc_clustering for details on how to specify the type_labels and type_constraints parameters.

Examples

```
# Example scclust clustering
my_scclust <- scclust(c("A", "A", "B", "C", "B",
                        "C", "C", "A", "B", "B"))
```

```
# Check so each cluster contains at least two data points
check_clustering(my_scclust, 2)
# > TRUE


# Check so each cluster contains at least four data points
check_clustering(my_scclust, 4)
# > FALSE


# Data point types
my_types <- factor(c("x", "y", "y", "z", "z",
                     "x", "y", "z", "x", "x"))


# Check so each cluster contains at least one point of each type
check_clustering(my_scclust,
                 NULL,
                 my_types,
                 c("x" = 1, "y" = 1, "z" = 1))
# > TRUE


# Check so each cluster contains one data point of both "x" and "z"
# and at least three points in total
check_clustering(my_scclust,
                 3,
                 my_types,
                 c("x" = 1, "z" = 1))
# > TRUE


# Check so each cluster contains five data points of type "y"
check_clustering(my_scclust,
                 NULL,
                 my_types,
                 c("y" = 5))
# > FALSE
```

cluster_count *Count the number of clusters*

Description

cluster_count returns the number of clusters in a clustering.

Usage

```
cluster_count(clustering)
```

Arguments

`clustering` a `scclust` object containing a non-empty clustering.

Value

Returns an integer with the number of clusters in `clustering`.

Examples

```
# Example scclust clustering
my_scclust <- scclust(c("A", "A", "B", "C", "B",
                        "C", "C", "A", "B", "B"))

cluster_count(my_scclust)
# > 3
```

`get_clustering_stats` *Get clustering statistics*

Description

`get_clustering_stats` calculates statistics of a clustering.

Usage

```
get_clustering_stats(distances, clustering)
```

Arguments

`distances` a `distances` object describing the distances between the data points in `clustering`.

`clustering` a `scclust` object containing a non-empty clustering.

Details

The function reports the following measures:

`num_data_points`	total number of data points
`num_assigned`	number of points assigned to a cluster
`num_clusters`	number of clusters
`min_cluster_size`	size of the smallest cluster
`max_cluster_size`	size of the largest cluster
`avg_cluster_size`	average cluster size
`sum_dists`	sum of all within-cluster distances
`min_dist`	smallest within-cluster distance
`max_dist`	largest within-cluster distance
`avg_min_dist`	average of the clusters' smallest distances
`avg_max_dist`	average of the clusters' largest distances

`avg_dist_weighted` average of the clusters' average distances weighed by cluster size
`avg_dist_unweighted` average of the clusters' average distances (unweighed)

Let $d(i, j)$ denote the distance between data points i and j. Let c be a cluster containing the indices of points assigned to the cluster. Let

$$D(c) = \{d(i, j) : i, j \in c \land i > j\}$$

be a function returning all within-cluster distances in c. Let C be a set containing all clusters.

`sum_dists` is defined as:

$$\sum_{c \in C} sum(D(c))$$

`min_dist` is defined as:

$$\min_{c \in C} \min(D(c))$$

`max_dist` is defined as:

$$\max_{c \in C} \max(D(c))$$

`avg_min_dist` is defined as:

$$\sum_{c \in C} \frac{\min(D(c))}{|C|}$$

`avg_max_dist` is defined as:

$$\sum_{c \in C} \frac{\max(D(c))}{|C|}$$

Let:

$$AD(c) = \frac{sum(D(c))}{|D(c)|}$$

be the average within-cluster distance in cluster c.

`avg_dist_weighted` is defined as:

$$\sum_{c \in C} \frac{|c| AD(c)}{num_a ssigned}$$

where $num_a ssigned$ is the number of assigned data points (see above).

`avg_dist_unweighted` is defined as:

$$\sum_{c \in C} \frac{AD(c)}{|C|}$$

Value

Returns a list of class `clustering_stats` containing the statistics.

Examples

```
my_data_points <- data.frame(x = c(0.1, 0.2, 0.3, 0.4, 0.5,
                                   0.6, 0.7, 0.8, 0.9, 1.0),
                             y = c(10, 9, 8, 7, 6,
                                   10, 9, 8, 7, 6))

my_distances <- distances(my_data_points)

my_scclust <- scclust(c("A", "A", "B", "C", "B",
                        "C", "C", "A", "B", "B"))

get_clustering_stats(my_distances, my_scclust)

# >                     Value
# > num_data_points     10.0000000
# > num_assigned        10.0000000
# > num_clusters         3.0000000
# > min_cluster_size     3.0000000
# > max_cluster_size     4.0000000
# > avg_cluster_size     3.3333333
# > sum_dists           18.2013097
# > min_dist             0.5000000
# > max_dist             3.0066593
# > avg_min_dist         0.8366584
# > avg_max_dist         2.4148611
# > avg_dist_weighted    1.5575594
# > avg_dist_unweighted  1.5847484
```

hierarchical_clustering *Hierarchical size-constrained clustering*

Description

hierarchical_clustering serves two purposes. Its primary use is to refine existing clusters subject to clustering constraints. That is, given a (non-optimal) clustering satisfying some constraints, the function splits clusters so to decrease within-cluster distances without violating the constraints. The function can also be used to derive size-constrained clusterings from scratch. In both cases, it uses a hierarchical clustering algorithm.

Usage

```
hierarchical_clustering(distances, size_constraint, batch_assign = TRUE,
  existing_clustering = NULL)
```

Arguments

distances	a distances object with distances between the data points.
size_constraint	an integer with the required minimum cluster size.
batch_assign	a logical indicating whether data points should be assigned in batches when spliting clusters (see below for details).
existing_clustering	a scclust object containing a non-empty clustering to refine. If NULL, the function derives a clustering from scratch.

Details

While hierarchical_clustering can be used to derive size-constrained clusters from scratch, its main purpose is to be used together with sc_clustering. The clusters produced by the main clustering function are guaranteed to be close to optimal (in particular, within a constant factor of the optimal solution). However, it is occasionally possible to refine the clustering. In particular, sc_clustering tends produce large clusters in regions of the metric space with many data points. In some cases, it is beneficial to divide these clusters into smaller groups. hierarchical_clustering splits these larger clusters so that all within-cluster distances weakly decrease while respecting the overall the size constraint.

hierarchical_clustering implements a divisive hierarchical clustering algorithm that respect size constraints. Starting from any clustering satisfying the size constraints (which may be a single cluster containing all data points), the function searches for clusters that can be broken into two or more new clusters without violating the constraints. When such a cluster is found, it breaks the cluster into two new clusters. It continues in this fashion until all remaining clusters are unbreakable.

Breakable clusters are broken in three stages. First, it tries to find two data points as far as possible from each other. The two points are called *centers*, and they are the starting points for the new clusters. The remaining data points in the old cluster will be assigned to one of the centers. In the second stage, each center picks the closest data points so that the new two clusters exactly satisfy the size constraint. In the last stage, data points that still are in the old cluster are assigned to the cluster containing the closest center. The final stage is done either for each point one-by-one or in batches (see below). When all data points are assigned to a cluster, the old cluster is removed and the two new are added to the clustering. If one or both of the new clusters are breakable, they will go through the same procedure again.

In some applications, it is desirable to avoid clusters that contain a number of data points that are not multiples of size_constraint. After the second stage described in the previous paragraph, both partial clusters are exact multiples of the size constraint. By assigning remaining data points in the third stage in batches of size_constraint, the function ensures, to the greatest extent possible, that the size of the final clusters are multiples of the size constraint.

Value

Returns a scclust object with the derived clustering.

See Also

sc_clustering is the main clustering function in the package.

Examples

```
# Make example data
my_data <- data.frame(x1 = rnorm(10000),
                      x2 = rnorm(10000),
                      x3 = rnorm(10000))

# Construct distance metric
my_dist <- distances(my_data)

# Make clustering with `sc_clustering`
my_clustering <- sc_clustering(my_dist, 3)

# Refine clustering with `hierarchical_clustering`
my_refined_clustering <- hierarchical_clustering(my_dist,
                                                 size_constraint = 3,
                                                 existing_clustering = my_clustering)

# Make clustering from scratch with `hierarchical_clustering`
my_other_clustering <- hierarchical_clustering(my_dist, 3)
```

is.scclust *Check scclust object*

Description

`is.scclust` checks whether the provided object is a valid instance of the `scclust` class.

Usage

```
is.scclust(x)
```

Arguments

x	object to check.

Details

`is.scclust` does not check whether the clustering itself is sensible or whether the clustering satisfies some set of constraints. See `check_clustering` for that functionality.

Value

Returns `TRUE` if `x` is a valid `scclust` object, otherwise `FALSE`.

`scclust` *Constructor for scclust objects*

Description

The `scclust` function constructs a `scclust` object from existing cluster labels.

Usage

```
scclust(cluster_labels, unassigned_labels = NULL, ids = NULL)
```

Arguments

`cluster_labels` a vector containing each data point's cluster label.

`unassigned_labels`
labels that denote unassigned data points. If `NULL`, `NA` values in `cluster_labels` are used to denote unassigned points.

`ids` IDs of the data points. Should be a vector of the same length as `cluster_labels` or `NULL`. If `NULL`, the IDs are set to `1:length(cluster_labels)`.

Details

`scclust` does not derive clusters from sets of data points; see `sc_clustering` and `hierarchical_clustering` for that functionality.

Value

Returns a `scclust` object with the clustering described by the provided labels.

Examples

```
# 10 data points in 3 clusters
my_scclust1 <- scclust(c("A", "A", "B", "C", "B",
                         "C", "C", "A", "B", "B"))

# 8 data points in 3 clusters, 2 points unassigned
my_scclust2 <- scclust(c(1, 1, 2, 3, 2,
                         NA, 3, 1, NA, 2))

# Custom labels indicating unassiged points
my_scclust3 <- scclust(c("A", "A", "B", "C", "NONE",
                         "C", "C", "NONE", "B", "B"),
                       unassigned_labels = "NONE")

# Two different labels indicating unassiged points
my_scclust4 <- scclust(c("A", "A", "B", "C", "NONE",
                         "C", "C", "0", "B", "B"),
                       unassigned_labels = c("NONE", "0"))
```

```
# Custom data point IDs
my_labels5 <- scclust(c("A", "A", "B", "C", "B",
                        "C", "C", "A", "B", "B"),
                      ids = letters[1:10])
```

sc_clustering *Size-constrained clustering*

Description

sc_clustering constructs near-optimal size-constrained clusterings. Subject to user-specified constraints on the size and composition of the clusters, a clustering is constructed so that within-cluster pair-wise distances are minimized. The function does not restrict the number of clusters.

Usage

```
sc_clustering(distances, size_constraint = NULL, type_labels = NULL,
  type_constraints = NULL, seed_method = "exclusion_updating",
  primary_data_points = NULL, primary_unassigned_method = "closest_seed",
  secondary_unassigned_method = "ignore", seed_radius = NULL,
  primary_radius = "seed_radius", secondary_radius = "estimated_radius",
  batch_size = 100L)
```

Arguments

distances
: a distances object with distances between the data points.

size_constraint
: an integer with the required minimum cluster size.

type_labels
: a vector containing the type of each data point. May be NULL when type_constraints is NULL.

type_constraints
: a named integer vector containing type-specific size constraints. If NULL, only the overall constraint given by size_constraint will be imposed on the clustering.

seed_method
: a character scalar indicating how seeds should be selected.

primary_data_points
: a vector specifying primary data points, either with point indices or with a logical vector of length equal to the number of points. NULL indicates that all data points are "primary".

primary_unassigned_method
: a character scalar indicating how unassigned (primary) points should be assigned to clusters.

secondary_unassigned_method
: a character scalar indicating how unassigned secondary points should be assigned to clusters.

`seed_radius` a positive numeric scalar restricting the maximum length of an edge in the graph used to construct the clustering. `NULL` indicates no restriction. This parameter (together with `primary_radius` and `secondary_radius`) can be used to control the maximum distance between points assigned to the same cluster (see below for details).

`primary_radius` a positive numeric scalar, a character scalar or NULL restricting the match distance for unassigned primary points. If numeric, the value is used to restrict the distances. If character, it must be one of "no_radius", "seed_radius" or "estimated_radius" (see below for details). `NULL` indicates no radius restriction.

`secondary_radius`
a positive numeric scalar, a character scalar or NULL restricting the match distance for unassigned secondary points. If numeric, the value is used to restrict the distances. If character, it must be one of "no_radius", "seed_radius" or "estimated_radius" (see below for details). `NULL` indicates no radius restriction.

`batch_size` an integer scalar specifying batch size when `seed_method` is set to "batches".

Details

`sc_clustering` constructs a clustering so to minimize within-cluster dissimilarities while ensuring that constraints on the size and composition of the clusters are satisfied. It is possible to impose an overall size constraint so that each cluster must contain at least a certain number of points in total. It is also possible to impose constraints on the composition of the clusters so that each cluster must contain a certain number of points of different types. For example, in a sample with "red" and "blue" data points, one can constrain the clustering so that each cluster must contain at least 10 points in total of which at least 3 must be "red" and at least 2 must be "blue".

The function implements an algorithm that first summaries the distances between data points in a sparse graph and then constructs the clustering based on the graph. This admits fast execution while ensuring near-optimal performance. In particular, the maximum within-cluster distance is garantueed to be at most four times the maximum distance in the optimal clustering. The average performance is much closer to optimal than the worst case bound.

In more detail, the clustering algorithm has four steps:

1. Construct sparse graph encoding clustering constraints.
2. Select a set of vertices in the graph to act as "seeds".
3. Construct initial clusters from the seeds' neighborhoods in the graph.
4. Assign remaining data points to clusters.

Each data point is represented by a vertex in the sparse graph, and arcs are weighted by the distance between the data points they connect. The graph is constructed so that each vertex's neighborhood satisfies the clustering constraints supplied by the user. For example, if the clusters must contain at least five points, each closed neighborhood in the graph will contain five vertices. `sc_clustering` constructs the graph that minimize the arc weights subject to the neighborhood constraints; this ensures near-optimal performance. The function selects "seeds" that have non-overlapping neighborhood in the graph. By constructing clusters as supersets of the neighborhoods, it ensures that the clustering will satisfy the constraints imposed by the user.

The `seed_method` option decides how the seeds are chosen. Any set of seeds yields a near-optimal clustering, but, heuristically, performance can be improved by picking the seeds more carefully.

In most cases, smaller clusters are desirable since they tend to minimize within-cluster distances. As the number of data points is fixed, we minimize the cluster size by maximizing the number of clusters (and, thus, the number of seeds). When `seed_method` is set to "lexical", seeds are chosen in lexical order. This admits a fast solution, but the function might pick points that are central in the graph. Central points tend to exclude many other points from being seeds and, thus, lead to larger clusters.

The "exclusion_order" and "exclusion_updating" options calculates, for each vertex, how many other vertices are excluded if a vertex is picked as a seed. By picking seeds that exclude few other vertices, the function avoids central points and increases the number of seeds. "exclusion_updating" updates the count after each picked seed so that already excluded vertices are not counted twice; "exclusion_order" derives the count once. The former option is, thus, better-performing but slower.

Deriving the exclusion count is an expensive operation, and it might not be feasible to use the option in very large samples. The "inwards_order" and "inwards_updating" options count the number of inwards-pointing arcs in the graph, which approximates the exclusion count. "inwards_updating" updates the count after each picked seed, while "inwards_order" derives the count once. The "batches" option is identical to "lexical" but it derives the graph in batches. This limits the memory complexity to a constant value proportional to `batch_size`. This can be useful when one imposes large size constraints. The "batches" option is still experimental and can currently only be used when one does not impose type constraints.

Once the function has selected seeds so that no additional points can be selected without creating overlap in the graph, it constructs the initial clusters. A unique cluster label is assigned to each seed, and all points in the seed's neighborhood is assigned the same label. Some data points might not be in a neighborhood of a seed and will, therefore, not be assigned to a cluster after the initial clusters have been formed. `primary_unassigned_method` specifies how the unassigned points are assigned. With the "ignore" option, the points are left unassigned. With the "any_neighbor", the function re-uses the sparse graph and assigns the unassigned points to any adjacent cluster in the graph. The "closest_assigned" option assigns the points to the clusters that contain their closest assigned vertex, and the "closest_seed" option assigns to the cluster that contain the closest seed. All these options ensure that the clustering is near-optimal.

Occasionally, some data points are allowed to be left unassigned. Consider the following prediction problem as an example. We have a set of data points with a known outcome value ("training points") and another set of points for which the outcome is unknown ("prediction points"). We want to use the training points to predict the outcome for the prediction points. By clustering the data, we can use the cluster mean of the training points to predict the values of the prediction points in the same cluster. To make this viable, we need to ensure that each cluster contains at least one training point (the cluster mean would otherwise be undefined). We can impose this constraint using the `type_labels` and `type_constraints` options. We also need to make sure that each prediction point is assigned to a cluster. We do, however, not need that all training points are assigned a cluster; some training points might not provide useful information (e.g., if they are very dissimilar to all prediction points). In this case, by specifying only the prediction points in `primary_data_points`, we ensure that all those points are assigned to clusters. The points not specified in `primary_data_points` (i.e., the training points) will be assigned to clustering only insofar that it is needed to satisfy the clustering constraints. This can lead to large improvements in the clustering if the types of points are unevenly distributed in the metric space. Points not specified in `primary_data_points` are called "secondary".

Generally, one does not want to discard all unassigned secondary points – some of them will, occasionally, be close to a cluster and contain useful information. Similar to `primary_unassigned_method`, we can use the secondary_unassigned_method to specify how the leftover secondary points should be assigned. The three possible options are "ignore", "closest_assigned" and "closest_seed". In nearly all cases, it is beneficial to impose a radius constraint when assigning secondary points (see below for details).

sc_clustering tries to minimize within-cluster distances subject to that all (primary) data points are assigned to clusters. In some cases, it might be beneficial to leave some points unassigned to avoid clusters with very dissimilar points. The function has options that can be used to indirectly constrain the maximum within-cluster distance. Specifically, by restricting the maximum distance in the four steps of the function, we can bound the maximum within-cluster distance. The bound is, however, a blunt tool. A too small bound might lead to that only a few data points are assigned to clusters. If a bound is needed, it is recommended to use a very liberal bound. This will avoid the very dissimilar clusters, but let the function optimize the remaining cluster assignments.

The seed_radius option limits the maximum distance between adjacent vertices in the sparse graph. If the distance to a neighbor in a point's neighborhood is greater than seed_radius, the neighborhood will be deleted and the point is not allowed be a seed (it can, however, still be in other points' neighborhoods). The primary_radius option limits the maximum distance when a primary point is assigned to in the fourth step. When primary_radius is set to a positive numeric value, this will be used to as the radius restriction. "no_radius" and NULL indicate no restriction. "seed_radius" indicates that the same restriction as seed_radius should be used, and "estimated_radius" sets the restriction to the estimated average distance between the seeds and their neighbors. When primary_unassigned_method is set to "any_neighbor", primary_radius must be set to "seed_radius".

The way the radius constraints restrict the maximum within-cluster distance depends on the primary_unassigned_method option. When primary_unassigned_method is "ignore", the maximum distance is bounded by 2 * seed_radius. When primary_unassigned_method is "any_neighbor", it is bounded by 4 * seed_radius. When primary_unassigned_method is "closest_assigned", it is bounded by 2 * seed_radius + 2 * primary_radius. When primary_unassigned_method is "closest_seed", it is bounded by the maximum of 2 * seed_radius and 2 * primary_radius.

The secondary_radius option restricts how secondary points are assigned, but is otherwise identical to the primary_radius option.

Value

Returns a scclust object with the derived clustering.

References

Higgins, Michael J., Fredrik Sävje and Jasjeet S. Sekhon (2016), 'Improving massive experiments with threshold blocking', *Proceedings of the National Academy of Sciences*, **113:27**, 7369–7376. http://www.pnas.org/lookup/doi/10.1073/pnas.1510504113

Sävje, Fredrik and Michael J. Higgins and Jasjeet S. Sekhon (2017), 'Generalized Full Matching', arXiv 1703.03882. https://arxiv.org/abs/1703.03882

See Also

hierarchical_clustering can be used to refine the clustering constructed by sc_clustering.

Examples

```
# Make example data
my_data <- data.frame(id = 1:50000,
                      type = factor(rbinom(50000, 3, 0.3),
                                    labels = c("A", "B", "C", "D")),
                      x1 = rnorm(50000),
                      x2 = rnorm(50000),
                      x3 = rnorm(50000))

# Construct distance metric
my_dist <- distances(my_data,
                     id_variable = "id",
                     dist_variables = c("x1", "x2", "x3"))

# Make clustering with at least 3 data points in each cluster
my_clustering <- sc_clustering(my_dist, 3)

# Check so clustering satisfies constraints
check_clustering(my_clustering, 3)
# > TRUE

# Get statistics about the clustering
get_clustering_stats(my_dist, my_clustering)
# > num_data_points        5.000000e+04
# > ...

# Make clustering with at least one point of each type in each cluster
my_clustering <- sc_clustering(my_dist,
                               type_labels = my_data$type,
                               type_constraints = c("A" = 1, "B" = 1,
                                                    "C" = 1, "D" = 1))

# Check so clustering satisfies constraints
check_clustering(my_clustering,
                 type_labels = my_data$type,
                 type_constraints = c("A" = 1, "B" = 1,
                                      "C" = 1, "D" = 1))
# > TRUE

# Make clustering with at least 8 points in total of which at least
# one must be "A", two must be "B" and five can be any type
my_clustering <- sc_clustering(my_dist,
                               size_constraint = 8,
                               type_labels = my_data$type,
                               type_constraints = c("A" = 1, "B" = 2))
```

Index