Package ‘simplifyEnrichment’

April 28, 2025

Type Package

Title Simplify Functional Enrichment Results

Version 2.2.0

Date 2024-09-13

Depends R (>= 4.0.0)

Imports simona, ComplexHeatmap (>= 2.7.4), grid, circlize, GetoptLong, digest, tm, GO.db, AnnotationDbi, slam, methods, clue, grDevices, stats, utils, cluster (>= 1.14.2), colorspace, GlobalOptions (>= 0.1.0)

Suggests knitr, ggplot2, cowplot, mclust, apcluster, MCL, dbscan, igraph, gridExtra, dynamicTreeCut, testthat, gridGraphics, flexclust, BiocManager, InteractiveComplexHeatmap (>= 0.99.11), shiny, shinydashboard, cola, hu6800.db, rmarkdown, genefilter, gridtext, fpc

Description
A new clustering algorithm, ``binary cut", for clustering similarity matrices of functional terms is implemeted in this package. It also provides functions for visualizing, summarizing and comparing the clusterings.

biocViews Software, Visualization, GO, Clustering, GeneSetEnrichment

URL `https://github.com/jokergoo/simplifyEnrichment`,
`https://simplifyEnrichment.github.io`

VignetteBuilder knitr

License MIT + file LICENSE

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.1

git_url https://git.bioconductor.org/packages/simplifyEnrichment

git_branch RELEASE_3_21

git_last_commit 04bc7cd

git_last_commit_date 2025-04-15

Repository Bioconductor 3.21

Date/Publication 2025-04-27

Author Zuguang Gu [aut, cre] (ORCID: <`https://orcid.org/0000-0002-7395-8709`>)

Maintainer Zuguang Gu <`z.gu@dkfz.de`>

Contents

anno_word_cloud 2
anno_word_cloud_from_GO 4
area_above_ecdf 5
cluster_terms 6
cmp_make_clusters 8
count_words 9
dend_node_apply 11
difference_score 12
export_to_shiny_app 13
GO_similarity 14
ht_clusters 15
keyword_enrichment_from_GO 17
partition_by_kmeans 18
plot_binary_cut 19
register_clustering_methods 20
scale_fontsize 22
select_cutoff 22
se_opt 23
simplifyGO 24
simplifyGOFromMultipleLists 25
summarizeGO 28
word_cloud_grob 29

Index **32**

`anno_word_cloud` *Word cloud annotations*

Description

Word cloud annotations

Usage

```
anno_word_cloud(
  align_to,
  term,
  exclude_words = NULL,
  max_words = 10,
```

```
  word_cloud_grob_param = list(),
  fontsize_range = c(4, 16),
  value_range = NULL,
  bg_gp = gpar(fill = "#DDDDDD", col = "#AAAAAA"),
  side = c("right", "left"),
  add_new_line = FALSE,
  count_words_param = list(),
  ...,
  return_gbl = FALSE
)
```

Arguments

`align_to`	How to align the annotations to the heatmap. Similar as in `ComplexHeatmap::anno_link`, the value of `align_to` can be a list of row indices or a categorical vector where each vector in the list corresponds to a word cloud. If it is a categorical vector, rows with the same level correspond to a same word cloud. If `align_to` is a categorical vector and `term` is a list, names of `term` should have overlap to the levels in `align_to`. When `align_to` is set as a categorical vector, normally the same value is set to `row_split` in the main heatmap so that each row slice can correspond to a word cloud.
`term`	The description text used for constructing the word clouds. The value should have the same format as `align_to`. If `align_to` is a list, `term` should also be a list. In this case, the length of vectors in `term` is not necessarily the same as in `align_to`. E.g. `length(term[[1]])` is not necessarily equal to `length(align_to[[1]]`. If `align_to` is a categorical vector, `term` should also be a character vector with the same length as `align_to`. To make it more genrall, when `align_to` is a list, `term` can also be a list of data frames where the first column contains keywords and the second column contains numeric values that will be mapped to font sizes in the word clouds.
`exclude_words`	The words excluced for construcing word cloud.
`max_words`	Maximal number of words visualized in the word cloud.
`word_cloud_grob_param`	A list of graphics parameters passed to `word_cloud_grob`.
`fontsize_range`	The range of the font size. The value should be a numeric vector with length two. The font size interpolation is linear.
`value_range`	The range of values to map to font sizes.
`bg_gp`	Graphics parameters for controlling the background.
`side`	Side of the annotation relative to the heatmap.
`add_new_line`	Whether to add new line after every word? If `TRUE`, each word will be in a separated line.
`count_words_param`	A list of parameters passed to `count_words`.
`...`	Other parameters.
`return_gbl`	Internally used.

Details

The word cloud annotation is constructed by `ComplexHeatmap::anno_link`.

If the annotation is failed to construct or no keyword is found, the function returns a `ComplexHeatmap::anno_empty` with 1px width.

English stop words, punctuation and numbers are removed by default when counting words. As specific stop words might coincide with gene or pathway names, and numbers in genes names might be meaningful it is recommended to adjust this behaviour by passing appropriate arguments to the `count_words` function using `count_words_param`.

Examples

```
gm = readRDS(system.file("extdata", "random_GO_BP_sim_mat.rds", package = "simplifyEnrichment"))
go_id = rownames(gm)
go_term = AnnotationDbi::select(GO.db::GO.db, keys = go_id, columns = "TERM")$TERM

split = sample(letters[1:4], 100, replace = TRUE)
align_to = split(1:100, split)
term = lapply(letters[1:4], function(x) sample(go_term, sample(100:400, 1)))
names(term) = letters[1:4]

require(ComplexHeatmap)
mat = matrix(rnorm(100*10), nrow = 100)
Heatmap(mat, cluster_rows = FALSE, row_split = split,
 right_annotation = rowAnnotation(foo = anno_word_cloud(align_to, term)))
```

`anno_word_cloud_from_GO`

Word cloud annotations from GO

Description

Word cloud annotations from GO

Usage

```
anno_word_cloud_from_GO(
  align_to,
  go_id,
  stat = c("pvalue", "count"),
  min_stat = ifelse(stat == "count", 5, 0.05),
  term = NULL,
  exclude_words = NULL,
  ...
)
```

Arguments

`align_to`	The same format as in `anno_word_cloud`.
`go_id`	The value should be in the same format as `align_to`. If `go_id` is a vector, it should have the same length as `align_to`, and if `go_id` is a list, note, e.g. `length(go_id[[1]])` is not necessarily equal to `length(align_to[[1]]`. If `align_to` is a categorical vector and `go_id` is a list, names of `go_id` should have overlap to the levels in `align_to`.
`stat`	What type of value to map to font sizes of the keywords. There are two possible values. "pvalue": enrichment is applied to keywords and -log10(p-value) is used to map to font size; "count": simply word frequency of keywords.
`min_stat`	Minimal value for `stat` for selecting keywords.
`term`	Alternatively the GO description can be set via the `term` argument. The same format as in `anno_word_cloud`.
`exclude_words`	The words excluced for construcing word cloud. Some words are internally exclucded: `c("via", "protein", "factor", "side", "type", "specific")`.
`...`	All other arguments passed to `anno_word_cloud`.

`area_above_ecdf` *Area above the eCDF curve*

Description

Area above the eCDF curve

Usage

```
area_above_ecdf(x)
```

Arguments

`x`	A vector of similarity values.

Details

Denote `F(x)` as the eCDF (empirical Cumulative Distribution Function) of the similarity vector `x`, this function calculates the area above the eCDF curve, which is `1 - \int_0^1 F(x)dx`.

Value

A numeric value.

`cluster_terms` *Cluster terms based on their similarity matrix*

Description

Cluster terms based on their similarity matrix

Usage

```
cluster_terms(
  mat,
  method = "binary_cut",
  control = list(),
  verbose = se_opt$verbose
)

cluster_by_kmeans(mat, max_k = max(2, min(round(nrow(mat)/5), 100)), ...)

cluster_by_pam(mat, max_k = max(2, min(round(nrow(mat)/10), 100)), ...)

cluster_by_dynamicTreeCut(mat, minClusterSize = 5, ...)

cluster_by_fast_greedy(mat, ...)

cluster_by_leading_eigen(mat, ...)

cluster_by_louvain(mat, ...)

cluster_by_walktrap(mat, ...)

cluster_by_mclust(mat, G = seq_len(max(2, min(round(nrow(mat)/5), 100))), ...)

cluster_by_apcluster(mat, s = apcluster::negDistMat(r = 2), ...)

cluster_by_hdbscan(mat, minPts = 5, ...)

cluster_by_MCL(mat, addLoops = TRUE, ...)
```

Arguments

`mat`	A similarity matrix.
`method`	The clustering methods. Value should be in `all_clustering_methods()`.
`control`	A list of parameters passed to the corresponding clustering function.
`verbose`	Whether to print messages.
`max_k`	Maximal *k* for k-means/PAM clustering. K-means/PAM clustering is applied from k = 2 to k = max_k.

`...`	Other arguments.
`minClusterSize`	Minimal number of objects in a cluster. Pass to `dynamicTreeCut::cutreeDynamic()`.
`G`	Passed to the `G` argument in `mclust::Mclust()` which is the number of clusters.
`s`	Passed to the `s` argument in `apcluster::apcluster()`.
`minPts`	Passed to the `minPts` argument in `dbscan::hdbscan()`.
`addLoops`	Passed to the `addLoops` argument in `MCL::mcl()`.

Details

New clustering methods can be registered by `register_clustering_methods()`.

Please note it is better to directly use `cluster_terms()` for clustering while not the individual `cluster_by_*` functions because `cluster_terms()` does additional cluster label adjustment.

By default, there are the following clustering methods and corresponding clustering functions:

- `kmeans` see `cluster_by_kmeans()`.
- `dynamicTreeCut` see `cluster_by_dynamicTreeCut()`.
- `mclust` see `cluster_by_mclust()`.
- `apcluster` see `cluster_by_apcluster()`.
- `hdbscan` see `cluster_by_hdbscan()`.
- `fast_greedy` see `cluster_by_fast_greedy()`.
- `louvain` see `cluster_by_louvain()`.
- `walktrap` see `cluster_by_walktrap()`.
- `MCL` see `cluster_by_MCL()`.
- `binary_cut` see `binary_cut()`.

The additional argument in individual clustering functions can be set with the `control` argument in `cluster_terms()`.

`cluster_by_kmeans()`: The best k for k-means clustering is determined according to the "elbow" or "knee" method on the distribution of within-cluster sum of squares (WSS) on each k. All other arguments are passed from `...` to `stats::kmeans()`.

`cluster_by_pam()`: PAM is applied by `fpc::pamk()` which can automatically select the best k. All other arguments are passed from `...` to `fpc::pamk()`.

`cluster_by_dynamicTreeCut()`: All other arguments are passed from `...` to `dynamicTreeCut::cutreeDynamic()`.

`cluster_by_fast_greedy()`: All other arguments are passed from `...` to `igraph::cluster_fast_greedy()`.

`cluster_by_leading_eigen()`: All other arguments are passed from `...` to `igraph::cluster_leading_eigen()`.

`cluster_by_louvain()`: All other arguments are passed from `...` to `igraph::cluster_louvain()`.

`cluster_by_walktrap()`: All other arguments are passed from `...` to `igraph::cluster_walktrap()`.

`cluster_by_mclust()`: All other arguments are passed from `...` to `mclust::Mclust()`.

`cluster_by_apcluster()`: All other arguments are passed from `...` to `apcluster::apcluster()`.

`cluster_by_hdbscan()`: All other arguments are passed from `...` to `dbscan::hdbscan()`.

`cluster_by_MCL()`: All other arguments are passed from `...` to `MCL::mcl()`.

Value

A vector of numeric cluster labels.

`cmp_make_clusters` *Compare clustering methods*

Description

Compare clustering methods

Usage

```
cmp_make_clusters(
  mat,
  method = setdiff(all_clustering_methods(), "mclust"),
  verbose = TRUE
)

cmp_make_plot(mat, clt, plot_type = c("mixed", "heatmap"), nrow = 3)

compare_clustering_methods(
  mat,
  method = setdiff(all_clustering_methods(), "mclust"),
  plot_type = c("mixed", "heatmap"),
  nrow = 3,
  verbose = TRUE
)
```

Arguments

`mat`	The similarity matrix.
`method`	Which methods to compare. All available methods are in `all_clustering_methods()`. A value of `"all"` takes all available methods. By default "mclust" is excluded because its long runtime.
`verbose`	Whether to print messages. Ddetails The function compares following default clustering methods by default: -`kmeans` see `cluster_by_kmeans`. -`pam` see `cluster_by_pam`. -`dynamicTreeCut` see `cluster_by_dynamicTreeCut`. -`mclust` see `cluster_by_mclust`. By default it is not included. -`apcluster` see `cluster_by_apcluster`. -`hdbscan` see `cluster_by_hdbscan`. -`fast_greedy` see `cluster_by_fast_greedy`. -`louvain` see `cluster_by_louvain`. -`walktrap` see `cluster_by_walktrap`. -`MCL` see `cluster_by_MCL`. -`binary_cut` see `binary_cut`. Also the user-defined methods in `all_clustering_methods` are also compared.
`clt`	A list of clusterings from `cmp_make_clusters()`.
`plot_type`	What type of plots to make. See **Details**.
`nrow`	Number of rows of the layout when `plot_type` is set to `"heatmap"`.

Details

For `cmp_make_plot()`, if `plot_type` is the default value `"mixed"`, a figure with three panels will be generated:

- A heatmap of the similarity matrix with different classifications as row annotations.
- A heatmap of the pair-wise concordance of the classifications of every two clustering methods.
- Barplots of the difference scores for each method (calculated by `difference_score`), the number of clusters (total clusters and the clusters with size >= 5) and the mean similarity of the terms that are in the same clusters.

If `plot_type` is `"heatmap"`. There are heatmaps for the similarity matrix under clusterings from different methods. The last panel is a table with the number of clusters under different clusterings.

`compare_clustering_methods()` is basically a wrapper function of `cmp_make_clusters()` and `cmp_make_plot()`.

Value

`cmp_make_clusters()` returns a list of cluster label vectors from different clustering methods.

`cmp_make_plot()` returns no value.

`compare_clustering_methods()` returns no value.

Examples

```
mat = readRDS(system.file("extdata", "random_GO_BP_sim_mat.rds",
    package = "simplifyEnrichment"))
compare_clustering_methods(mat)
compare_clustering_methods(mat, plot_type = "heatmap")
```

`count_words` *Calculate word frequency*

Description

Calculate word frequency

Usage

```
count_words(
  term,
  exclude_words = NULL,
  stop_words = stopwords(),
  min_word_length = 1,
  tokenizer = "words",
  transform_case = tolower,
  remove_numbers = TRUE,
```

```
  remove_punctuation = TRUE,
  custom_transformer = NULL,
  stemming = FALSE,
  dictionary = NULL
)
```

Arguments

`term`	A vector of description texts.
`exclude_words`	The words that should be excluded.
`stop_words`	The stop words that should be be removed.
`min_word_length`	Minimum length of the word to be counted.
`tokenizer`	The tokenizer function, one of the values accepted by `tm::termFreq`.
`transform_case`	The function normalizing lettercase of the words.
`remove_numbers`	Whether to remove numbers.
`remove_punctuation`	Whether to remove punctuation.
`custom_transformer`	Custom function that transforms words.
`stemming`	Whether to only keep the roots of inflected words.
`dictionary`	A vector of words to be counted (if given all other words will be excluded).

Details

The text preprocessing followings the instruction from `http://www.sthda.com/english/wiki/word-cloud-generator-in-r-one-killer-function-to-do-everything-you-need`.

Value

A data frame with words and frequencies.

Examples

```
gm = readRDS(system.file("extdata", "random_GO_BP_sim_mat.rds", package = "simplifyEnrichment"))
go_id = rownames(gm)
go_term = AnnotationDbi::select(GO.db::GO.db, keys = go_id, columns = "TERM")$TERM
count_words(go_term) |> head()
```

`dend_node_apply` *Apply functions on every node in a dendrogram*

Description

Apply functions on every node in a dendrogram

Usage

```
dend_node_apply(dend, fun)

edit_node(dend, fun = function(d, index) d)
```

Arguments

`dend`	A `dendrogram` object.
`fun`	A self-defined function.

Details

`dend_node_apply()` returns a vector or a list as the same length as the number of nodes in the dendrogram.

The self-defined function can have one single argument which is the sub-dendrogram at a certain node. E.g. to get the number of members at every node:

```
dend_node_apply(dend, function(d) attr(d, "members"))
```

The self-defined function can have a second argument, which is the index of current sub-dendrogram in the complete dendrogram. E.g. `dend[[1]]` is the first child node of the complete dendrogram and `dend[[c(1, 2)]]` is the second child node of `dend[[1]]`, et al. This makes that at a certain node, it is possible to get information of its child nodes and parent nodes.

```
dend_node_apply(dend, function(d, index) {
    dend[[c(index, 1)]] # is the first child node of d, or simply d[[1]]
    dend[[index[-length(index)]]] # is the parent node of d
    ...
})
```

Note for the top node, the value of `index` is `NULL`.

In `edit_node()`, if `fun` only has one argument, it is basically the same as `stats::dendrapply()`, but it can have a second argument which is the index of the node in the dendrogram, which makes it possible to get information of child nodes and parent nodes for a specific node.

As an example, we first assign random values to every node in the dendrogram:

```
mat = matrix(rnorm(100), 10)
dend = as.dendrogram(hclust(dist(mat)))
dend = edit_node(dend, function(d) {attr(d, 'score') = runif(1); d})
```

Then for every node, we take the maximal absolute difference to all its child nodes and parent node as the attribute `abs_diff`.

```
dend = edit_node(dend, function(d, index) {
    n = length(index)
    s = attr(d, "score")
    if(is.null(index)) {  # d is the top node
        s_children = sapply(d, function(x) attr(x, "score"))
        s_parent = NULL
    } else if(is.leaf(d)) { # d is the leaf
        s_children = NULL
        s_parent = attr(dend[[index[-n]]], "score")
    } else {
        s_children = sapply(d, function(x) attr(x, "score"))
        s_parent = attr(dend[[index[-n]]], "score")
    }
    abs_diff = max(abs(s - c(s_children, s_parent)))
    attr(d, "abs_diff") = abs_diff
    return(d)
})
```

Value

`dend_node_apply()` returns a vector or a list, depends on whether `fun` returns a scalar or more complex values.

`edit_node()` returns a `dendrogram` object.

Examples

```
mat = matrix(rnorm(100), 10)
dend = as.dendrogram(hclust(dist(mat)))
# number of members on every node
dend_node_apply(dend, function(d) attr(d, "members"))
# the depth on every node
dend_node_apply(dend, function(d, index) length(index))
```

`difference_score` *Difference score*

Description

Difference score

Usage

```
difference_score(mat, cl)
```

Arguments

`mat`	The similarity matrix.
`cl`	Cluster labels.

Details

This function measures the different between the similarity values for the terms that belong to the same clusters and in different clusters. The difference score is the Kolmogorov-Smirnov statistic between the two distributions.

Value

A numeric scalar.

Examples

```
mat = readRDS(system.file("extdata", "random_GO_BP_sim_mat.rds",
    package = "simplifyEnrichment"))
cl = binary_cut(mat)
difference_score(mat, cl)
```

`export_to_shiny_app` *Interactively visualize the similarity heatmap*

Description

Interactively visualize the similarity heatmap

Usage

```
export_to_shiny_app(mat, cl = binary_cut(mat))
```

Arguments

`mat`	A similarity matrix.
`cl`	Cluster labels inferred from the similarity matrix, e.g. from `cluster_terms()` or `binary_cut()`.

Value

A shiny application.

Examples

```
if(interactive()) {
    mat = readRDS(system.file("extdata", "random_GO_BP_sim_mat.rds",
        package = "simplifyEnrichment"))
    cl = binary_cut(mat)
    export_to_shiny_app(mat, cl)
}
```

`GO_similarity` *Calculate Gene Ontology (GO) semantic similarity matrix*

Description

Calculate Gene Ontology (GO) semantic similarity matrix

Usage

```
GO_similarity(
  go_id,
  ont = NULL,
  db = "org.Hs.eg.db",
  measure = "Sim_XGraSM_2013"
)

guess_ont(go_id, db = "org.Hs.eg.db")

random_GO(n, ont = c("BP", "CC", "MF"), db = "org.Hs.eg.db")
```

Arguments

`go_id`	A vector of GO IDs.
`ont`	Sub-ontology of GO. Value should be one of "BP", "CC" or "MF". If it is not specified, the function automatically identifies it by random sampling 10 IDs from go_id (see `guess_ont()`).
`db`	Annotation database. It should be an OrgDb package name from `https://bioconductor.org/packages/release/BiocViews.html#___OrgDb`. The value can also directly be an `OrgDb` object.
`measure`	Semantic measure for the GO similarity, pass to `simona::term_sim()`. All valid values are in `simona::all_term_sim_methods()`.
`n`	Number of GO IDs.

Details

The default similarity method is "Sim_XGraSM_2013". Since the semantic similarities are calculated based on gene annotations to GO terms, I suggest users also try the following methods:

- `"Sim_Lin_1998"`

- `"Sim_Resnik_1999"`
- `"Sim_Relevance_2006"`
- `"Sim_SimIC_2010"`
- `"Sim_XGraSM_2013"`
- `"Sim_EISI_2015"`
- `"Sim_AIC_2014"`
- `"Sim_Wang_2007"`
- `"Sim_GOGO_2018"`

In `guess_ont()`, only 10 random GO IDs are checked.

In `random_GO()`, only GO terms with gene annotations are sampled.

Value

`GO_similarity()` returns a symmetric matrix.

`guess_ont()` returns a single character scalar of "BP", "CC" or "MF". If there are more than one ontologies detected. It returns `NULL`.

`random_GO()` returns a vector of GO IDs.

Examples

```
go_id = random_GO(100)
mat = GO_similarity(go_id)


go_id = random_GO(100)
guess_ont(go_id)
```

`ht_clusters` *Visualize the similarity matrix and the clustering*

Description

Visualize the similarity matrix and the clustering

Usage

```
ht_clusters(
  mat,
  cl,
  dend = NULL,
  col = c("white", "red"),
  draw_word_cloud = TRUE,
  min_term = round(nrow(mat) * 0.01),
```

```
  order_by_size = FALSE,
  stat = "pvalue",
  min_stat = ifelse(stat == "count", 5, 0.05),
  exclude_words = character(0),
  max_words = 10,
  word_cloud_grob_param = list(),
  fontsize_range = c(4, 16),
  bg_gp = gpar(fill = "#DDDDDD", col = "#AAAAAA"),
  column_title = NULL,
  ht_list = NULL,
  use_raster = TRUE,
  run_draw = TRUE,
  ...
)
```

Arguments

`mat`	A similarity matrix.
`cl`	Cluster labels inferred from the similarity matrix, e.g. from `cluster_terms()` or `binary_cut()`.
`dend`	Used internally.
`col`	A vector of colors that map from 0 to the 97.5^th percentile of the similarity values. The value can also be a color mapping function generated by `circlize::colorRamp2()`.
`draw_word_cloud`	Whether to draw the word clouds.
`min_term`	Minimal number of functional terms in a cluster. All the clusters with size less than `min_term` are all merged into one separated cluster in the heatmap.
`order_by_size`	Whether to reorder clusters by their sizes. The cluster that is merged from small clusters (size < `min_term`) is always put to the bottom of the heatmap.
`stat`	Type of value for mapping to the font size of keywords in the word clouds. There are two options: "count": simply number of keywords; "pvalue": enrichment on keywords is performed (by fisher's exact test) and -log10(pvalue) is used to map to font sizes.
`min_stat`	Minimal value for `stat` for selecting keywords.
`exclude_words`	Words that are excluded in the word cloud.
`max_words`	Maximal number of words visualized in the word cloud.
`word_cloud_grob_param`	A list of graphic parameters passed to `word_cloud_grob()`.
`fontsize_range`	The range of the font size. The value should be a numeric vector with length two. The font size interpolation is linear.
`bg_gp`	Graphics parameters for controlling word cloud annotation background.
`column_title`	Column title for the heatmap.
`ht_list`	A list of additional heatmaps added to the left of the similarity heatmap.
`use_raster`	Whether to write the heatmap as a raster image.
`run_draw`	Internally used.
`...`	Other arguments passed to `ComplexHeatmap::draw,HeatmapList-method`.

Value

A `ComplexHeatmap::HeatmapList` object.

Examples

```
mat = readRDS(system.file("extdata", "random_GO_BP_sim_mat.rds",
    package = "simplifyEnrichment"))
cl = binary_cut(mat)
ht_clusters(mat, cl, word_cloud_grob_param = list(max_width = 80))
ht_clusters(mat, cl, word_cloud_grob_param = list(max_width = 80),
    order_by_size = TRUE)
```

`keyword_enrichment_from_GO` *Keyword enrichment for GO terms*

Description

Keyword enrichment for GO terms

Usage

```
keyword_enrichment_from_GO(go_id, min_bg = 5, min_term = 2)
```

Arguments

`go_id`	A vector of GO IDs.
`min_bg`	Minimal number of GO terms (in the background, i.e. all GO temrs in the GO database) that contain a specific keyword.
`min_term`	Minimal number of GO terms (GO terms in `go_id`) that contain a specific keyword.

Details

The enrichment is applied by Fisher's exact test. For a keyword, there is the following 2x2 contigency table:

```
                 | contains the keyword | does not contain the keyword
    In the GO set     |         s11          |          s12
    Not in the GO set |         s21          |          s22
```

where `s11`, `s12`, `s21` and `s22` are the counts of GO terms in the four categories.

Value

A data frame with keyword enrichment results.

Examples

```
go_id = random_GO(100)
keyword_enrichment_from_GO(go_id)
```

`partition_by_kmeans` *Partition the matrix*

Description

Partition the matrix

Usage

```
partition_by_kmeans(mat, n_repeats = 10)

partition_by_pam(mat)

partition_by_hclust(mat)

partition_by_kmeanspp(mat)
```

Arguments

`mat`	The submatrix in the binary cut clustering process.
`n_repeats`	Number of repeated runs of k-means clustering.

Details

These functions can be set to the `partition_fun` argument in `binary_cut()`.

`partition_by_kmeans()`: Since k-means clustering brings randomness, this function performs k-means clustering several times (controlled by `n_repeats`) and uses the final consensus partitioning results.

`partition_by_pam()`: The clustering is performed by `cluster::pam()` with the pamonce argument set to 5.

`partition_by_hclust()`: The "ward.D2" clusering method was used.

`partition_by_kmeanspp()`: It uses the kmeanspp method from the **flexclust** package.

Value

All partitioning functions split the matrix into two groups and return a categorical vector of labels of 1 and 2.

`plot_binary_cut` *Cluster functional terms by recursively binary cutting the similarity matrix*

Description

Cluster functional terms by recursively binary cutting the similarity matrix

Usage

```
plot_binary_cut(
  mat,
  value_fun = area_above_ecdf,
  cutoff = 0.85,
  partition_fun = partition_by_pam,
  dend = NULL,
  dend_width = unit(3, "cm"),
  depth = NULL,
  show_heatmap_legend = TRUE,
  ...
)

binary_cut(
  mat,
  value_fun = area_above_ecdf,
  partition_fun = partition_by_hclust,
  cutoff = 0.85,
  try_all_partition_fun = TRUE,
  partial = nrow(mat) > 1500
)
```

Arguments

`mat`	A similarity matrix.
`value_fun`	A function that calculates the scores for the four submatrices on a node.
`cutoff`	The cutoff for splitting the dendrogram.
`partition_fun`	A function to split each node into two groups. Pre-defined functions in this package are `partition_by_kmeanspp()`, `partition_by_pam()` and `partition_by_hclust()`.
`dend`	A dendrogram object, used internally.
`dend_width`	Width of the dendrogram on the plot.
`depth`	Depth of the recursive binary cut process.
`show_heatmap_legend`	Whether to show the heatmap legend.
`...`	Other arguments.

`try_all_partition_fun`	Different `partition_fun` may give different clusterings. If the vaule of `try_all_partition_fun` is set to `TRUE`, the similarity matrix is clustered by three partitioning method: `partition_by_pam()`, `partition_by_kmeanspp()` and `partition_by_hclust()`. The clustering with the highest difference score is finally selected as the final clustering.
`partial`	Whether to generate the complete clustering or the clustering stops when submatrices cannot be split anymore.

Details

After the functions which perform clustering are executed, such as `simplifyGO()` or `binary_cut()`, the dendrogram is temporarily saved and `plot_binary_cut()` directly uses this dendrogram.

Value

`binary_cut()` returns a vector of numeric cluster labels.

Examples

```
mat = readRDS(system.file("extdata", "random_GO_BP_sim_mat.rds",
    package = "simplifyEnrichment"))
plot_binary_cut(mat, depth = 1)
plot_binary_cut(mat, depth = 2)
plot_binary_cut(mat)

mat = readRDS(system.file("extdata", "random_GO_BP_sim_mat.rds",
    package = "simplifyEnrichment"))
binary_cut(mat)
```

`register_clustering_methods`

Configure clustering methods

Description

Configure clustering methods

Usage

```
register_clustering_methods(...)

all_clustering_methods()

remove_clustering_methods(method)

reset_clustering_methods()
```

Arguments

`...`	A named list of clustering functions, see in **Details**.
`method`	A vector of method names.

Details

The user-defined functions should accept at least one argument which is the input matrix. The second optional argument should always be `...` so that parameters for the clustering function can be passed by the `control` argument from `cluster_terms()`, `simplifyGO()` or `simplifyEnrichment()`. If users forget to add `...`, it is added internally.

Please note, the user-defined function should automatically identify the optimized number of clusters.

The function should return a vector of cluster labels. Internally it is converted to numeric labels.

The default clustering methods are:

- `kmeans` see `cluster_by_kmeans()`.
- `dynamicTreeCut` see `cluster_by_dynamicTreeCut()`.
- `mclust` see `cluster_by_mclust()`.
- `apcluster` see `cluster_by_apcluster()`.
- `hdbscan` see `cluster_by_hdbscan()`.
- `fast_greedy` see `cluster_by_fast_greedy()`.
- `louvain` see `cluster_by_louvain()`.
- `walktrap` see `cluster_by_walktrap()`.
- `MCL` see `cluster_by_MCL()`.
- `binary_cut` see `binary_cut()`.

Value

`all_clustering_methods()` returns a vector of clustering method names.

Examples

```
register_clustering_methods(
    # assume there are 5 groups
    random = function(mat, ...) sample(5, nrow(mat), replace = TRUE)
)
all_clustering_methods()
remove_clustering_methods("random")
all_clustering_methods()
remove_clustering_methods(c("kmeans", "mclust"))
all_clustering_methods()
reset_clustering_methods()
all_clustering_methods()
```

`scale_fontsize` *Scale font size*

Description

Scale font size

Usage

```
scale_fontsize(x, rg = c(1, 30), fs = c(4, 16))
```

Arguments

`x`	A numeric vector.
`rg`	The range.
`fs`	Range of the font size.

Details

It is a linear interpolation.

Value

A numeric vector.

Examples

```
x = runif(10, min = 1, max = 20)
# scale x to fontsize 4 to 16.
scale_fontsize(x)
```

`select_cutoff` *Select the cutoff for binary cut*

Description

Select the cutoff for binary cut

Usage

```
select_cutoff(
  mat,
  cutoff = seq(0.6, 0.98, by = 0.01),
  verbose = se_opt$verbose,
  ...
)
```

Arguments

`mat`	A similarity matrix.
`cutoff`	A list of cutoffs to test. Note the range of the cutoff values should be inside `[0.5, 1]`.
`verbose`	Whether to print messages.
`...`	Pass to `binary_cut()`.

Details

Binary cut is applied to each cutoff and the clustering results are evaluated by following metrics:

- difference score, calculated by `difference_score()`.
- number of clusters.
- block mean, which is the mean similarity in the blocks in the diagonal of the heatmap.

Examples

```
mat = readRDS(system.file("extdata", "random_GO_BP_sim_mat.rds",
    package = "simplifyEnrichment"))
select_cutoff(mat)
```

`se_opt` *Global parameters*

Description

Global parameters

Usage

```
se_opt(..., RESET = FALSE, READ.ONLY = NULL, LOCAL = FALSE, ADD = FALSE)
```

Arguments

`...`	Arguments for the parameters, see "details" section.
`RESET`	Whether to reset to default values.
`READ.ONLY`	Please ignore.
`LOCAL`	Please ignore.
`ADD`	Please ignore.

Details

There are the following global options:

- `verobse`: Whether to print messages.

Value

A `GlobalOptionsFun` object.

`simplifyGO` *Simplify Gene Ontology (GO) enrichment results*

Description

Simplify Gene Ontology (GO) enrichment results

Usage

```
simplifyGO(
  mat,
  method = "binary_cut",
  control = list(),
  plot = TRUE,
  verbose = TRUE,
  column_title = qq("@{nrow(mat)} GO terms clustered by '@{method}'"),
  ht_list = NULL,
  ...
)

simplifyEnrichment(...)
```

Arguments

`mat`	A GO similarity matrix. You can also provide a vector of GO IDs to this argument.
`method`	Method for clustering the matrix. See `cluster_terms()`.
`control`	A list of parameters for controlling the clustering method, passed to `cluster_terms()`.
`plot`	Whether to make the heatmap.
`verbose`	Whether to print messages.
`column_title`	Column title for the heatmap.
`ht_list`	A list of additional heatmaps added to the left of the similarity heatmap.
`...`	Arguments passed to `ht_clusters()`.

Details

This is basically a wrapper function that it first runs `cluster_terms()` to cluster GO terms and then runs `ht_clusters()` to visualize the clustering.

The arguments in `simplifyGO()` passed to `ht_clusters()` are:

- `draw_word_cloud`: Whether to draw the word clouds.

- `min_term`: Minimal number of GO terms in a cluster. All the clusters with size less than `min_term` are all merged into one single cluster in the heatmap.
- `order_by_size`: Whether to reorder GO clusters by their sizes. The cluster that is merged from small clusters (size < `min_term`) is always put to the bottom of the heatmap.
- `stat`: What values of keywords are used to map to font sizes in the word clouds.
- `exclude_words`: Words that are excluded in the word cloud.
- `max_words`: Maximal number of words visualized in the word cloud.
- `word_cloud_grob_param`: A list of graphic parameters passed to `word_cloud_grob()`.
- `fontsize_range` The range of the font size. The value should be a numeric vector with length two. The minimal font size is mapped to word frequency value of 1 and the maximal font size is mapped to the maximal word frequency. The font size interlopation is linear.
- `bg_gp`: Graphic parameters for controlling the background of word cloud annotations.

Value

A data frame with two columns: GO IDs and cluster labels.

Examples

```
set.seed(123)
go_id = random_GO(500)
mat = GO_similarity(go_id)
df = simplifyGO(mat, word_cloud_grob_param = list(max_width = 80))
head(df)
```

`simplifyGOFromMultipleLists`

Perform simplifyGO analysis with multiple lists of GO IDs

Description

Perform simplifyGO analysis with multiple lists of GO IDs

Usage

```
simplifyGOFromMultipleLists(
  lt,
  go_id_column = NULL,
  padj_column = NULL,
  padj_cutoff = 0.01,
  filter = function(x) any(x < padj_cutoff),
  default = 1,
  ont = NULL,
  db = "org.Hs.eg.db",
  measure = "Sim_XGraSM_2013",
```

```
    heatmap_param = list(NULL),
    show_barplot = TRUE,
    method = "binary_cut",
    control = list(),
    min_term = NULL,
    verbose = TRUE,
    column_title = NULL,
    ...
)
```

Arguments

`lt`	A data frame, a list of numeric vectors (e.g. adjusted p-values) where each numeric vector has GO IDs as names, or a list of GO IDs.
`go_id_column`	Column index of GO ID if `lt` contains a list of data frames.
`padj_column`	Column index of adjusted p-values if `lt` contains a list of data frames.
`padj_cutoff`	Cut off for adjusted p-values.
`filter`	A self-defined function for filtering GO IDs. By default it requires GO IDs should be significant in at least one list.
`default`	The default value for the adjusted p-values. See **Details**.
`ont`	Pass to `GO_similarity()`.
`db`	Pass to `GO_similarity()`.
`measure`	Pass to `GO_similarity()`.
`heatmap_param`	Parameters for controlling the heatmap, see **Details**.
`show_barplot`	Whether draw barplots which shows numbers of significant GO terms in clusters.
`method`	Pass to `simplifyGO()`.
`control`	Pass to `simplifyGO()`.
`min_term`	Pass to `simplifyGO()`.
`verbose`	Pass to `simplifyGO()`.
`column_title`	Pass to `simplifyGO()`.
`...`	Pass to `simplifyGO()`.

Details

The input data can have three types of formats:

- A list of numeric vectors of adjusted p-values where each vector has the GO IDs as names.
- A data frame. The column of the GO IDs can be specified with `go_id_column` argument and the column of the adjusted p-values can be specified with `padj_column` argument. If these columns are not specified, they are automatically identified. The GO ID column is found by checking whether a column contains all GO IDs. The adjusted p-value column is found by comparing the column names of the data frame to see whether it might be a column for adjusted p-values. These two columns are used to construct a numeric vector with GO IDs as names.

- A list of character vectors of GO IDs. In this case, each character vector is changed to a numeric vector where all values take 1 and the original GO IDs are used as names of the vector.

Now let's assume there are `n` GO lists, we first construct a global matrix where columns correspond to the `n` GO lists and rows correspond to the "union" of all GO IDs in the lists. The value for the ith GO ID and in the jth list are taken from the corresponding numeric vector in `lt`. If the jth vector in `lt` does not contain the ith GO ID, the value defined by `default` argument is taken there (e.g. in most cases the numeric values are adjusted p-values, `default` is set to 1). Let's call this matrix as `M0`.

Next step is to filter `M0` so that we only take a subset of GO IDs of interest. We define a proper function via argument `filter` to remove GO IDs that are not important for the analysis. Functions for `filter` is applied to every row in `M0` and `filter` function needs to return a logical value to decide whether to remove the current GO ID. For example, if the values in `lt` are adjusted p-values, the `filter` function can be set as `function(x) any(x < padj_cutoff)` so that the GO ID is kept as long as it is signfiicant in at least one list. After the filter, let's call the filtered matrix `M1`.

GO IDs in `M1` (row names of `M1`) are used for clustering. A heatmap of `M1` is attached to the left of the GO similarity heatmap so that the group-specific (or list-specific) patterns can be easily observed and to corresponded to GO functions.

Argument `heatmap_param` controls several parameters for heatmap `M1`:

- `transform`: A self-defined function to transform the data for heatmap visualization. The most typical case is to transform adjusted p-values by `-log10(x)`.
- `breaks`: break values for color interpolation.
- `col`: The corresponding values for `breaks`.
- `labels`: The corresponding labels.
- `name`: Legend title.

Examples

```
# perform functional enrichment on the signatures genes from cola anlaysis
require(cola)
data(golub_cola)
res = golub_cola["ATC:skmeans"]
require(hu6800.db)
x = hu6800ENTREZID
mapped_probes = mappedkeys(x)
id_mapping = unlist(as.list(x[mapped_probes]))
lt = functional_enrichment(res, k = 3, id_mapping = id_mapping) # you can check the value of `lt`

# a list of data frames
simplifyGOFromMultipleLists(lt, padj_cutoff = 0.001)

# a list of numeric values
lt2 = lapply(lt, function(x) structure(x$p.adjust, names = x$ID))
simplifyGOFromMultipleLists(lt2, padj_cutoff = 0.001)

# a list of GO IDS
lt3 = lapply(lt, function(x) x$ID[x$p.adjust < 0.001])
```

```
simplifyGOFromMultipleLists(lt3)
```

`summarizeGO` *A simplified way to visualize enrichment in GO clusters*

Description

A simplified way to visualize enrichment in GO clusters

Usage

```
summarizeGO(
  go_id,
  value = NULL,
  aggregate = mean,
  method = "binary_cut",
  control = list(),
  verbose = TRUE,
  axis_label = "Value",
  title = "",
  legend_title = axis_label,
  min_term = round(nrow(mat) * 0.01),
  stat = "pvalue",
  min_stat = ifelse(stat == "count", 5, 0.05),
  exclude_words = character(0),
  max_words = 6,
  word_cloud_grob_param = list(),
  fontsize_range = c(4, 16),
  bg_gp = gpar(fill = "#DDDDDD", col = "#AAAAAA")
)
```

Arguments

`go_id`	A vector of GO IDs.
`value`	A list of numeric value associate with `go_id`. We suggest to use `-log10(p.adjust)` or `-log2(fold enrichment)` as the values.
`aggregate`	Function to aggregate values in each GO cluster.
`method`	Method for clustering the matrix. See `cluster_terms()`.
`control`	A list of parameters for controlling the clustering method, passed to `cluster_terms()`.
`verbose`	Whether to print messages.
`axis_label`	X-axis label.
`title`	Title for the whole plot.
`legend_title`	Title for the legend.

`min_term`	Minimal number of functional terms in a cluster. All the clusters with size less than `min_term` are all merged into one separated cluster in the heatmap.
`stat`	Type of value for mapping to the font size of keywords in the word clouds. There are two options: "count": simply number of keywords; "pvalue": enrichment on keywords is performed (by fisher's exact test) and -log10(pvalue) is used to map to font sizes.
`min_stat`	Minimal value for `stat` for selecting keywords.
`exclude_words`	Words that are excluded in the word cloud.
`max_words`	Maximal number of words visualized in the word cloud.
`word_cloud_grob_param`	A list of graphic parameters passed to `word_cloud_grob`.
`fontsize_range`	The range of the font size. The value should be a numeric vector with length two. The font size interpolation is linear.
`bg_gp`	Graphics parameters for controlling word cloud annotation background.

Details

There are several other ways to specify GO IDs and the associated values.

1. specify `value` as a named vector where GO IDs are the names.
2. specify `value` as a list of numeric named vectors. In this case, `value` contains multiple enrichment results.

Please refer to `https://jokergoo.github.io/2023/10/02/simplified-simplifyenrichment-plot/` for more examples of this function.

`word_cloud_grob` *A simple grob for the word cloud*

Description

A simple grob for the word cloud

Usage

```
word_cloud_grob(
  text,
  fontsize,
  line_space = unit(4, "pt"),
  word_space = unit(4, "pt"),
  max_width = unit(80, "mm"),
  col = function(fs) circlize::rand_color(length(fs), luminosity = "dark"),
  add_new_line = FALSE,
  test = FALSE
)
```

```
## S3 method for class 'word_cloud'
widthDetails(x)

## S3 method for class 'word_cloud'
heightDetails(x)
```

Arguments

`text`	A vector of words.
`fontsize`	The corresponding font size. With the frequency of the words known, `scale_fontsize` can be used to linearly interpolate frequencies to font sizes.
`line_space`	Space between lines. The value can be a `grid::unit` object or a numeric scalar which is measured in mm.
`word_space`	Space between words. The value can be a `grid::unit` object or a numeric scalar which is measured in mm.
`max_width`	The maximal width of the viewport to put the word cloud. The value can be a `grid::unit` object or a numeric scalar which is measured in mm. Note this might be larger than the final width of the returned grob object.
`col`	Colors for the words. The value can be a vector, in numeric or character, which should have the same length as `text`. Or it is a self-defined function that takes the font size vector as the only argument. The function should return a color vector. See Examples.
`add_new_line`	Whether to add new line after every word? If `TRUE`, each word will be in a separated line.
`test`	Internally used. It basically adds borders to the words and the viewport.
`x`	The `word_cloud` grob returned by `word_cloud_grob`.

Value

A `grid::grob` object. The width and height of the grob can be get by `grid::grobWidth` and `grid::grobHeight`.

Examples

```
# very old R versions do not have strrep() function
if(!exists("strrep")) {
    strrep = function(x, i) paste(rep(x, i), collapse = "")
}
words = sapply(1:30, function(x) strrep(sample(letters, 1), sample(3:10, 1)))
require(grid)
gb = word_cloud_grob(words, fontsize = runif(30, min = 5, max = 30),
    max_width = 100)
grid.newpage(); grid.draw(gb)

# color as a single scalar
gb = word_cloud_grob(words, fontsize = runif(30, min = 5, max = 30),
    max_width = 100, col = 1)
grid.newpage(); grid.draw(gb)
```

```
# color as a vector
gb = word_cloud_grob(words, fontsize = runif(30, min = 5, max = 30),
    max_width = 100, col = 1:30)
grid.newpage(); grid.draw(gb)

# color as a function
require(circlize)
col_fun = colorRamp2(c(5, 17, 30), c("blue", "black", "red"))
gb = word_cloud_grob(words, fontsize = runif(30, min = 5, max = 30),
    max_width = 100, col = function(fs) col_fun(fs))
grid.newpage(); grid.draw(gb)
```

Index

all_clustering_methods (register_clustering_methods), 20
all_clustering_methods(), *6*, *8*
anno_word_cloud, 2
anno_word_cloud_from_GO, 4
apcluster::apcluster(), *7*
area_above_ecdf, 5

binary_cut (plot_binary_cut), 19
binary_cut(), *7*, *13*, *16*, *18*, *21*, *23*

circlize::colorRamp2(), *16*
cluster::pam(), *18*
cluster_by_apcluster (cluster_terms), 6
cluster_by_apcluster(), *7*, *21*
cluster_by_dynamicTreeCut (cluster_terms), 6
cluster_by_dynamicTreeCut(), *7*, *21*
cluster_by_fast_greedy (cluster_terms), 6
cluster_by_fast_greedy(), *7*, *21*
cluster_by_hdbscan (cluster_terms), 6
cluster_by_hdbscan(), *7*, *21*
cluster_by_kmeans (cluster_terms), 6
cluster_by_kmeans(), *7*, *21*
cluster_by_leading_eigen (cluster_terms), 6
cluster_by_louvain (cluster_terms), 6
cluster_by_louvain(), *7*, *21*
cluster_by_MCL (cluster_terms), 6
cluster_by_MCL(), *7*, *21*
cluster_by_mclust (cluster_terms), 6
cluster_by_mclust(), *7*, *21*
cluster_by_pam (cluster_terms), 6
cluster_by_walktrap (cluster_terms), 6
cluster_by_walktrap(), *7*, *21*
cluster_terms, 6
cluster_terms(), *13*, *16*, *21*, *24*, *28*
cmp_make_clusters, 8
cmp_make_plot (cmp_make_clusters), 8
compare_clustering_methods (cmp_make_clusters), 8
ComplexHeatmap::HeatmapList, *17*
count_words, 9

dbscan::hdbscan(), *7*
dend_node_apply, 11
difference_score, 12
difference_score(), *23*
dynamicTreeCut::cutreeDynamic(), *7*

edit_node (dend_node_apply), 11
export_to_shiny_app, 13

fpc::pamk(), *7*

GO_similarity, 14
GO_similarity(), *26*
guess_ont (GO_similarity), 14

heightDetails.word_cloud (word_cloud_grob), 29
ht_clusters, 15
ht_clusters(), *24*

igraph::cluster_fast_greedy(), *7*
igraph::cluster_leading_eigen(), *7*
igraph::cluster_louvain(), *7*
igraph::cluster_walktrap(), *7*

keyword_enrichment_from_GO, 17

MCL::mcl(), *7*
mclust::Mclust(), *7*

partition_by_hclust (partition_by_kmeans), 18
partition_by_hclust(), *19*
partition_by_kmeans, 18
partition_by_kmeanspp (partition_by_kmeans), 18

partition_by_kmeanspp(), *19*
partition_by_pam (partition_by_kmeans), 18
partition_by_pam(), *19*
plot_binary_cut, 19

random_GO (GO_similarity), 14
register_clustering_methods, 20
register_clustering_methods(), *7*
remove_clustering_methods (register_clustering_methods), 20
reset_clustering_methods (register_clustering_methods), 20

scale_fontsize, 22
se_opt, 23
select_cutoff, 22
simona::all_term_sim_methods(), *14*
simona::term_sim(), *14*
simplifyEnrichment (simplifyGO), 24
simplifyEnrichment(), *21*
simplifyGO, 24
simplifyGO(), *21*, *26*
simplifyGOFromMultipleLists, 25
stats::dendrapply(), *11*
stats::kmeans(), *7*
summarizeGO, 28

widthDetails.word_cloud (word_cloud_grob), 29
word_cloud_grob, 29
word_cloud_grob(), *16*, *25*