Package 'themescopeR'

August 20, 2026

Title Social Representation Analysis via Semantic Network Mapping

Version 0.1.1

Description Implements the ThemeScope framework for detecting and visualising social representations in large-scale digital text corpora. From raw documents it builds, via 'udpipe' annotation, sentence-level word co-occurrence networks and derives two community-level indicators grounded in Social Representation Theory: the Prototypical Salience Index (PSI) for anchoring and the Concreteness Score (CS) for objectification. Communities are located in a two-dimensional, theoretically grounded representational map. The whole pipeline is usable from the R console; an optional 'shiny' graphical interface calls the same exported functions. The method is described in Misuraca, Spano and D'Aniello (2026) <doi:10.1177/01655515261454276>.

License MIT + file LICENSE

URL https://github.com/lucadaniello/themescopeR

BugReports https://github.com/lucadaniello/themescopeR/issues

Encoding UTF-8

Language en-GB

LazyData true

Depends R (>= 4.1.0)

Imports cli (>= 3.6.0), dplyr (>= 1.1.0), ggplot2 (>= 3.4.0), ggrepel (>= 0.9.0), igraph (>= 2.1.0), Matrix (>= 1.6.0), methods, readr, readxl, rlang, stats, tools, udpipe (>= 0.8.11), utils

Suggests bslib, DT, ggraph (>= 2.1.0), htmltools, knitr, plotly, rmarkdown, shiny, shinycssloaders, testthat (>= 3.0.0), visNetwork, writexl

Config/testthat/edition 3

Config/roxygen2/version 8.0.0

NeedsCompilation no

Author Luca D'Aniello [aut, cre] (ORCID:
<https://orcid.org/0000-0003-1019-9212>),
Michelangelo Misuraca [aut] (ORCID:
<https://orcid.org/0000-0002-8794-966X>),
Maria Spano [aut] (ORCID: <https://orcid.org/0000-0002-3103-2342>)

Maintainer Luca D'Aniello <luca.daniello@unina.it>

Repository CRAN

Date/Publication 2026-08-20 14:10:02 UTC

Contents

assign_quadrant *Assign quadrant labels from z-scored PSI and CS*

Description

Maps communities to the four quadrants of the ThemeScope representational space based on their z-scored Prototypical Salience Index (PSI, anchoring) and Concreteness Score (CS, objectification).

Usage

```
assign_quadrant(psi_z, cs_z)
```

Arguments

psi_z	Numeric vector of z-scored PSI values.
cs_z	Numeric vector of z-scored CS values (same length as psi_z).

Value

A factor with levels:

- "Stable Core": high PSI, high CS.
- "Ideological Core": high PSI, low CS.
- "Emerging Practices": low PSI, high CS.
- "Latent Representations": low PSI, low CS.

Examples

```
assign_quadrant(c(1, 1, -1, -1), c(1, -1, 1, -1))
```

brysbaert *Brysbaert et al. (2014) concreteness norms*

Description

A lexicon of concreteness ratings for English words, from crowd-sourced human judgements on a 1–5 scale (1 = highly abstract, 5 = highly concrete). Used by compute_cs() and themescope() to compute the Concreteness Score (CS), operationalising objectification in Social Representation Theory.

Usage

```
brysbaert
```

Format

A data frame with two columns:

word Character. The English word (lower-case).

conc.m Numeric. Mean concreteness rating on a 1–5 scale.

Source

Brysbaert, M., Warriner, A. B., & Kuperman, V. (2014). Concreteness ratings for 40 thousand generally known English word lemmas. *Behavior Research Methods*, 46(3), 904–911. doi:10.3758/s1342801304035

Examples

```
data(brysbaert)
head(brysbaert)
```

---

build_cooccurrence_matrix

Build a co-occurrence (similarity) matrix

Description

Counts how many text units (sentences or documents) each pair of vocabulary terms co-occur in (each pair counted at most once per unit), then optionally re-weights the counts with a similarity measure (see `normalize_cooccurrence()`). Also returns the **presence** a_t of each term, the number of units containing it, which is the correct marginal for the normalisation.

Usage

```
build_cooccurrence_matrix(
  words_df,
  vocab = NULL,
  unit = c("lemma", "token"),
  vocab_size = NULL,
  pos_filter = c("NOUN", "ADJ", "PROPN"),
  window = c("sentence", "document"),
  normalization = NULL
)
```

Arguments

`words_df`	An annotated words data frame (e.g. from `preprocess_texts()`), containing `doc_id`, `sentence_id`, `upos` and the `unit` column.
`vocab`	Optional vocabulary. If `NULL` (default) it is built from `words_df` with `build_vocab()` using `unit`, `vocab_size` and `pos_filter`. You may instead pass a `build_vocab()` data frame to reuse a fixed vocabulary; the analysis unit is then taken from its term column.

`unit`	Character. Word column to use: `"lemma"` (default) or `"token"`. Ignored when `vocab` is a `build_vocab()` data frame (the unit is inferred from it).
`vocab_size`	Integer or `NULL`. Passed to `build_vocab()` when `vocab` is `NULL` (`NULL` = keep all terms).
`pos_filter`	Character vector of POS tags. Passed to `build_vocab()` when `vocab` is `NULL`.
`window`	Co-occurrence unit: `"sentence"` (default) counts terms that share a sentence; `"document"` counts terms that share a document.
`normalization`	Similarity measure applied to the counts: one of `"association"`, `"equivalence"`, `"jaccard"`, `"salton"`, `"inclusion"`, or `"frequency"`. `NULL` (default) is equivalent to `"frequency"` (raw counts).

Value

A named list with:

`cooc_matrix` Symmetric sparse matrix after `normalization` (raw counts if `NULL`/`"frequency"`).

`counts` The raw sentence/document co-occurrence counts.

`presence` Named integer vector a_t (units containing each term).

`vocab` The vocabulary data frame used.

`unit`, `window`, `normalization` The settings used.

Examples

```
df <- data.frame(
  doc_id = c(1, 1, 1, 1, 1), sentence_id = c(1, 1, 1, 2, 2),
  lemma = c("dog", "cat", "dog", "bird", "cat"), upos = "NOUN"
)
res <- build_cooccurrence_matrix(df, normalization = "association")
res$cooc_matrix
```

---

build_cooccurrence_network

Build a thresholded semantic co-occurrence network

Description

Constructs an undirected weighted `igraph` graph from a similarity matrix, retaining only edges whose weight exceeds the `threshold_percentile` quantile of all non-zero weights.

Usage

```
build_cooccurrence_network(
  as_matrix,
  threshold_percentile = 0.98,
  verbose = TRUE
)
```

Arguments

as_matrix	Symmetric sparse Matrix of edge weights (e.g. the cooc_matrix from build_cooccurrence_matrix() or the output of normalize_cooccurrence()).
threshold_percentile	Numeric in $(0, 1)$. Quantile of non-zero weights used as the minimum edge weight (default 0.98). This is an implementation choice for network sparsification, not part of the metric definitions; tune it for your corpus.
verbose	Logical. Print progress messages (default TRUE).

Value

An undirected weighted igraph object; E(graph)$weight holds the edge weights and V(graph)$name the term labels. Only terms with at least one retained edge appear as vertices.

Examples

```
words <- readRDS(system.file("extdata", "demo_annotated.rds",
                             package = "themescopeR"))
vocab <- build_vocab(words, vocab_size = 300)
cooc  <- build_cooccurrence_matrix(words, vocab = vocab)
net   <- build_cooccurrence_network(cooc$cooc_matrix, threshold_percentile = 0.98,
                                    verbose = FALSE)
net
```

---

build_vocab *Build a vocabulary data frame from an annotated words data frame*

Description

Counts the most frequent terms in a POS-filtered, annotated data frame and returns them as a tidy vocabulary. The analysis unit, the "word", can be the surface token or the lemma.

Usage

```
build_vocab(
  words_df,
  unit = c("lemma", "token"),
  vocab_size = NULL,
  pos_filter = c("NOUN", "ADJ", "PROPN")
)
```

Arguments

words_df	An annotated data frame (e.g. the output of preprocess_texts()). The "word" used to build the vocabulary is the column named by unit (token or lemma); the data frame must also contain doc_id, sentence_id and upos. See validate_words_df().
unit	Character. Which column to use as the word: "lemma" (default) or "token". When "lemma", missing lemmas fall back to the token.
vocab_size	Integer or NULL. If NULL (default) the vocabulary contains **all** terms; otherwise only the top vocab_size by frequency.
pos_filter	Character vector of Universal POS tags to retain (default c("NOUN", "ADJ", "PROPN")).

Value

A data frame ordered by decreasing frequency with columns:

token **or** lemma The word (lower-cased); the column is named after unit.

freq Integer term frequency (number of occurrences).

upos The dominant (most frequent) Universal POS tag of the term.

Examples

```
df <- data.frame(
  doc_id = c(1, 1, 1, 1), sentence_id = c(1, 1, 1, 2),
  token = c("dogs", "cats", "dog", "birds"),
  lemma = c("dog", "cat", "dog", "bird"), upos = "NOUN"
)
build_vocab(df, unit = "lemma")
```

---

compute_association_strength

Compute Association Strength from a co-occurrence matrix

Description

Convenience wrapper around normalize_cooccurrence() with method = "association": $AS(t, t') = c_{tt'}/(a_t a_{t'})$ (ThemeScope papers, Eq. 1). Values lie in $[0, 1]$; higher values indicate stronger association, reducing the bias of highly frequent terms.

Usage

```
compute_association_strength(cooc_matrix, presence)
```

Arguments

cooc_matrix	Symmetric sparse Matrix of co-occurrence counts.
presence	Named numeric vector of term presences a_t.

Value

A sparse dgCMatrix of Association Strength values.

See Also

normalize_cooccurrence() for other similarity measures.

Examples

```
m <- Matrix::sparseMatrix(
  i = c(1, 2), j = c(2, 1), x = c(3, 3), dims = c(3, 3),
  dimnames = list(c("a", "b", "c"), c("a", "b", "c"))
)
compute_association_strength(m, c(a = 5, b = 4, c = 2))
```

---

compute_coherence *Compute edge-density coherence per community*

Description

Coherence is the ratio of actual to maximum possible edges within a community (its edge density): $|E_i|/(|G_i|(|G_i|-1)/2)$.

Usage

```
compute_coherence(graph, communities)
```

Arguments

graph	An igraph object.
communities	Named list of character vectors of community members.

Value

Named numeric vector of coherence values in $[0, 1]$. Communities with fewer than 2 members return NA.

Examples

```
words  <- readRDS(system.file("extdata", "demo_annotated.rds",
                              package = "themescopeR"))
result <- themescope(words, vocab_size = 300, seed = 1, verbose = FALSE)
compute_coherence(result$graph, result$communities)
```

compute_cs *Compute the Concreteness Score (CS) per community*

Description

The CS operationalises *objectification*: the degree to which a community is grounded in concrete, perceptually accessible content. It is the association-strength-weighted mean concreteness of edge endpoints

$$C_w(g_i) = \frac{\sum_{(t,t')\in E_i} w_{t,t'} \cdot \frac{c(t)+c(t')}{2}}{\sum_{(t,t')\in E_i} w_{t,t'}},$$

where $w_{t,t'}$ is the AS edge weight and $c(t)$ the concreteness rating. Only edges where **both** endpoints have a rating contribute.

Usage

```
compute_cs(graph, communities, concreteness_lexicon = brysbaert)
```

Arguments

`graph`	An undirected weighted `igraph` object with a `weight` edge attribute.
`communities`	Named list of character vectors of community members.
`concreteness_lexicon`	Data frame with columns `"word"` and `"conc.m"` (1–5 scale). Defaults to the bundled brysbaert norms.

Value

A named numeric vector of CS values. Communities where no edge has both endpoints in the lexicon return `NA`.

Examples

```
words  <- readRDS(system.file("extdata", "demo_annotated.rds",
                              package = "themescopeR"))
result <- themescope(words, vocab_size = 300, seed = 1, verbose = FALSE)
compute_cs(result$graph, result$communities)
```

compute_network_stats *Compute network-level and community-level statistics*

Description

Compute network-level and community-level statistics

Usage

```
compute_network_stats(graph, communities)
```

Arguments

graph	An igraph object.
communities	Named list of character vectors of community members.

Value

A list with community_stats (data frame: community, size, density, mean_degree, n_edges) and global_stats (named list: n_nodes, n_edges, mean_degree, modularity, n_communities).

Examples

```
words  <- readRDS(system.file("extdata", "demo_annotated.rds",
                              package = "themescopeR"))
result <- themescope(words, vocab_size = 300, seed = 1, verbose = FALSE)
compute_network_stats(result$graph, result$communities)$global_stats
```

compute_psi *Compute the Prototypical Salience Index (PSI) per community*

Description

The PSI operationalises *anchoring* in Social Representation Theory: the degree to which a thematic community is structurally embedded in the wider discourse. Following the ThemeScope papers it is defined as

$$\Psi(g_i) = \frac{\sum_{t \in g_i} a_t \cdot s_t}{\max_j \delta(g_j)},$$

where a_t is the *presence* of term t (the number of sentences containing it), $s_t = \frac{\sum_{t' \in N_i(t)} AS_{t,t'}}{|N_i(t)|}$ is the *mean association strength* of t's neighbours within the community subgraph, and $\delta(g_i) = \frac{2W_i}{|V_i|(|V_i|-1)}$ is the weighted internal density of community i (W_i = sum of internal edge weights).

Usage

```
compute_psi(graph, communities, presence)
```

Arguments

graph	An undirected weighted igraph object (the full network). Edge weights are the Association Strength values.
communities	Named list of character vectors of community members, as returned by detect_communities().
presence	Named numeric vector of term presence a_t (sentence counts), as returned in the presence element of build_cooccurrence_matrix(). Names must match vertex names in graph.

Details

This implementation follows the equations in the ThemeScope manuscript (Eqs. 2–4) and case study (Eq. 2): the per-term contribution uses the **mean neighbour association strength** s_t (not the raw degree) and the normalisation uses the **maximum weighted community density** (not the maximum numerator). Because the denominator is a single global constant, the subsequent z-scoring used for the ThemeScope map is unaffected by it.

Value

A named numeric vector of PSI values, one per community.

Examples

```
words  <- readRDS(system.file("extdata", "demo_annotated.rds",
                              package = "themescopeR"))
result <- themescope(words, vocab_size = 300, seed = 1, verbose = FALSE)
compute_psi(result$graph, result$communities, result$presence)
```

---

detect_communities *Detect communities in a semantic network*

Description

Runs walktrap or Louvain community detection on a weighted undirected graph and filters out small communities.

Usage

```
detect_communities(
  graph,
  algorithm = "walktrap",
  steps = 4,
  resolution = 1,
  min_size = 10,
  seed = NULL,
  verbose = TRUE
)
```

Arguments

graph	An undirected weighted igraph object, e.g. from build_cooccurrence_network().
algorithm	Character. One of "walktrap" (default), "louvain" or "leiden".
steps	Integer. Number of steps for the walktrap random walk (default 4). Used only when algorithm = "walktrap".
resolution	Numeric resolution parameter for "louvain" and "leiden" (default 1); higher values yield more, smaller communities. Ignored by walktrap.
min_size	Integer. Minimum number of members for a community to be retained; smaller communities are assigned NA membership (default 10).
seed	Optional integer. If supplied, sets the random seed before community detection for reproducibility (Louvain and Leiden are stochastic).
verbose	Logical. Print a progress message (default TRUE).

Value

A named list with:

membership Named integer vector mapping each vertex to its community ID (NA for vertices in removed small communities).

communities Named list of character vectors of member names, one per retained community ("C1", "C2", ...).

algorithm_result The raw communities object from igraph.

Examples

```
words <- readRDS(system.file("extdata", "demo_annotated.rds",
                             package = "themescopeR"))
vocab <- build_vocab(words, vocab_size = 300)
cooc  <- build_cooccurrence_matrix(words, vocab = vocab)
net   <- build_cooccurrence_network(cooc$cooc_matrix, verbose = FALSE)
comm <- detect_communities(net, algorithm = "walktrap", min_size = 5, seed = 1)
lengths(comm$communities)

comm <- detect_communities(net, algorithm = "louvain", resolution = 1,
                           min_size = 5, seed = 1)
lengths(comm$communities)
```

---

get_community_subgraphs

Extract community subgraphs

Description

For each retained community, induces the subgraph of its vertices and their interconnecting edges.

Usage

```
get_community_subgraphs(graph, membership)
```

Arguments

graph	An igraph object (the full network).
membership	Named integer vector of community assignments from detect_communities(). NA values are ignored.

Value

A named list of igraph objects, one per community ("C1", "C2", ...).

Examples

```
words <- readRDS(system.file("extdata", "demo_annotated.rds",
                             package = "themescopeR"))
vocab <- build_vocab(words, vocab_size = 300)
cooc  <- build_cooccurrence_matrix(words, vocab = vocab)
net   <- build_cooccurrence_network(cooc$cooc_matrix, verbose = FALSE)
comm <- detect_communities(net, min_size = 5, seed = 1)
subgraphs <- get_community_subgraphs(net, comm$membership)
vapply(subgraphs, igraph::vcount, numeric(1))
```

---

lexicon_coverage *Concreteness lexicon coverage diagnostics*

Description

Reports how many terms have a rating in a concreteness lexicon, overall and for a themescope() result, per community. A low coverage means the Concreteness Score (CS) is computed on few edges and may be unreliable; this typically happens when the corpus language does not match the lexicon (the bundled brysbaert norms cover **English** words only).

Usage

```
lexicon_coverage(x, lexicon = brysbaert)
```

Arguments

x	Either a themescope object (coverage is reported per community and for the whole network) or a character vector of terms.
lexicon	Data frame with columns "word" and "conc.m". Defaults to the bundled brysbaert norms.

Value

A data frame with columns `community` (community id, or `"(all)"` for the overall row), `n_terms`, `n_matched` and `coverage` (proportion in $[0, 1]$).

See Also

`compute_cs()`, `match_concreteness()`.

Examples

```
lex <- data.frame(word = c("dog", "cat"), conc.m = c(4.8, 4.7))
lexicon_coverage(c("dog", "cat", "freedom"), lex)
```

`match_concreteness` *Match terms to a concreteness lexicon*

Description

Looks up concreteness ratings for a set of terms. Matching is case-insensitive.

Usage

```
match_concreteness(terms, lexicon = brysbaert)
```

Arguments

`terms`	Character vector of terms to look up.
`lexicon`	Data frame with columns `"word"` (character) and `"conc.m"` (numeric mean concreteness on a 1–5 scale). Defaults to the bundled brysbaert norms.

Value

Named numeric vector of concreteness values aligned to `terms`. Terms not found in the lexicon receive `NA`.

Examples

```
lex <- data.frame(word = c("dog", "cat", "freedom"), conc.m = c(4.8, 4.7, 1.5))
match_concreteness(c("dog", "freedom", "unknown"), lex)
```

`normalize_cooccurrence` *Normalise a co-occurrence matrix into a similarity matrix*

Description

Re-weights raw co-occurrence counts with one of several similarity measures commonly used in co-word analysis. Given the co-occurrence count c_{ij} and the term presences a_i, a_j:

`"association"` $c_{ij}/(a_i a_j)$, Association Strength (ThemeScope default; van Eck & Waltman).

`"equivalence"` $c_{ij}^2/(a_i a_j)$, equivalence index.

`"jaccard"` $c_{ij}/(a_i + a_j - c_{ij})$.

`"salton"` $c_{ij}/\sqrt{a_i a_j}$, cosine or Salton's measure.

`"inclusion"` $c_{ij}/\min(a_i, a_j)$.

`"frequency"` the raw counts, unchanged.

Usage

```
normalize_cooccurrence(
  cooc_matrix,
  presence,
 method = c("association", "equivalence", "jaccard", "salton", "inclusion", "frequency")
)
```

Arguments

`cooc_matrix`	Symmetric sparse `Matrix` of co-occurrence counts.
`presence`	Named numeric vector of term presences a_t; names must match the rows/columns of `cooc_matrix`.
`method`	One of `"association"` (default), `"equivalence"`, `"jaccard"`, `"salton"`, `"inclusion"`, `"frequency"`.

Value

A sparse `dgCMatrix` of the same dimensions as `cooc_matrix`.

Examples

```
m <- Matrix::sparseMatrix(
  i = c(1, 2), j = c(2, 1), x = c(3, 3), dims = c(3, 3),
  dimnames = list(c("a", "b", "c"), c("a", "b", "c"))
)
normalize_cooccurrence(m, c(a = 5, b = 4, c = 2), method = "jaccard")
```

plot_network *Plot a community-coloured semantic network (igraph)*

Description

Draws the co-occurrence network with **igraph**: nodes are coloured by community using the shared pastel palette (matching `plot_themescope()`), edges are light grey, node size scales with degree, and the highest-degree terms of each community are labelled.

Usage

```
plot_network(
  graph,
  membership,
  palette = NULL,
  layout = "fr",
  top_n_labels = 8,
  edge_color = "grey85",
  repulsion = 0,
  seed = NULL,
  ...
)
```

Arguments

`graph`	An `igraph` object.
`membership`	Named integer vector of community assignments from `detect_communities()` (`NA` for unassigned vertices).
`palette`	Optional vector of community colours (community `i` uses `palette[i]`). Defaults to `themescope_colours()`.
`layout`	Character layout: `"fr"` (default), `"kk"`, `"dh"`, `"lgl"`.
`top_n_labels`	Integer. Highest-degree nodes per community to label (default `8`).
`edge_color`	Colour of the edges (default light grey).
`repulsion`	Numeric in `[0, 1]` (default `0`). Pushes communities apart: after the base layout is computed, each community's nodes are displaced outward from the global centre in proportion to `repulsion`, so higher values separate clusters more (`0` leaves the layout unchanged).
`seed`	Optional integer seed for a reproducible layout.
`...`	Passed to `igraph::plot.igraph()`.

Value

Invisibly `NULL`; called for the plot it draws.

Examples

```
words  <- readRDS(system.file("extdata", "demo_annotated.rds",
                              package = "themescopeR"))
result <- themescope(words, vocab_size = 300, seed = 1, verbose = FALSE)
plot_network(result$graph, result$membership, top_n_labels = 3, seed = 1)
```

`plot_themescope` *Create a ThemeScope representational map*

Description

Produces the two-dimensional strategic diagram locating each community in the space defined by z-scored PSI (x-axis, anchoring) and z-scored CS (y-axis, objectification). Points are **coloured by community** with the shared pastel palette (so colours match `plot_network()`); the four SRT quadrants are annotated in the corners. The only legend is community size (number of terms), placed at the bottom.

Usage

```
plot_themescope(
  psi,
  cs,
  community_labels = NULL,
  community_sizes = NULL,
  title = "ThemeScope Map",
  palette = NULL,
  quadrant_fill = FALSE,
  ...
)
```

Arguments

`psi`	Named numeric vector of PSI values (one per community).
`cs`	Named numeric vector of CS values (one per community); may contain `NA`.
`community_labels`	Optional named character vector of point labels (e.g. top terms per community). If `NULL`, community names from `psi` are used.
`community_sizes`	Optional named numeric vector of community sizes (the number of terms in each community) used to scale point area.
`title`	Character. Plot title (default `"ThemeScope Map"`).
`palette`	Optional vector of community colours (one per community, in the order of `psi`). Defaults to `themescope_colours()`.
`quadrant_fill`	Logical (default `FALSE`). If `TRUE`, the four SRT quadrants get a soft background tint matching their corner annotations.
`...`	Currently unused.

Value

A ggplot object.

Examples

```
plot_themescope(
  psi = c(C1 = 1.2, C2 = -0.5, C3 = 0.3),
  cs  = c(C1 = 0.8, C2 = -1.1, C3 = 0.2)
)
```

`preprocess_texts` *Annotate a document collection with udpipe*

Description

Tokenises, lemmatises and POS-tags a collection of raw documents using a **udpipe** language model, returning a tokens data frame ready for `themescope()` and the rest of the backend.

Usage

```
preprocess_texts(
  collection,
  model,
  text_col = "text",
  doc_id_col = "doc_id",
  batch_size = 500,
  parallel_cores = 1,
  verbose = TRUE
)
```

Arguments

`collection`	Data frame with a text column and a document-id column, typically the output of `read_collection()`.
`model`	A **udpipe** model object, a path to a `.udpipe` file, or a language name (e.g. `"english"`) which is downloaded and cached via `ts_download_model()`.
`text_col`	Name of the text column (default `"text"`).
`doc_id_col`	Name of the document-id column (default `"doc_id"`).
`batch_size`	Integer. Documents processed per batch (default `500`). Ignored when `parallel_cores` > 1 (udpipe chunks the work itself).
`parallel_cores`	Integer (default 1). If greater than 1, annotation is parallelised across that many CPU cores via `udpipe::udpipe()`, which speeds up large corpora considerably.
`verbose`	Logical. Print progress messages (default `TRUE`).

Value

The **complete udpipe** annotation as a data frame (one row per token), with all columns returned by `udpipe::udpipe_annotate()` preserved (`doc_id`, `paragraph_id`, `sentence_id`, `sentence`, `token_id`, `token`, `lemma`, `upos`, `xpos`, `feats`, `head_token_id`, `dep_rel`, ...). Nothing is dropped, so the full linguistic annotation is available for inspection. Downstream functions (`build_vocab()`, `build_cooccurrence_matrix()`) select and case-fold the relevant word column (`token` or `lemma`) themselves, and use `doc_id` + `sentence_id` as the sentence key.

Examples

```
# The shape of the annotation this returns; the bundled demo corpus ships
# pre-annotated, so no model download is needed to inspect it
words <- readRDS(system.file("extdata", "demo_annotated.rds",
                             package = "themescopeR"))
head(words[, c("doc_id", "sentence_id", "token", "lemma", "upos")])


# Annotating your own corpus. The language model is downloaded once and
# cached; here it goes to the session temporary directory instead.
coll  <- read_collection(system.file("extdata", "sample_collection.csv",
                                     package = "themescopeR"))
model <- try(ts_download_model("english", model_dir = tempdir()), silent = TRUE)
if (!inherits(model, "try-error")) {
  tokens <- preprocess_texts(utils::head(coll, 5), model = model)
  head(tokens[, c("doc_id", "token", "lemma", "upos")])
}
```

`read_collection` *Read a document collection into a tidy data frame*

Description

Imports a corpus of raw documents from a single file or a zip archive of many files, returning a tidy data frame with a `doc_id` and a `text` column (plus any metadata columns present in the source).

Usage

```
read_collection(path, text_col = NULL, id_col = NULL, sequential_ids = FALSE)
```

Arguments

`path`	Path to a single file or a `.zip` archive.
`text_col`	Optional name of the text column. If `NULL`, detected from common names (`text`, `body`, `content`, `comment`, `message`) or, failing that, the sole character column.

id_col	Optional name of the document-id column. If NULL, detected from common names (doc_id, id, document, ...); if none is found, a sequential doc_id is generated (doc_1, doc_2, ...).
sequential_ids	Logical (default FALSE). If TRUE, documents are always numbered doc_1, doc_2, ... The identifier that came with the file is not discarded: it is kept as a source_id column. Useful when the source ids are long hashes that clutter the tables (this is what the Shiny GUI uses).

Details

Supported single-file formats: .csv, .tsv, .txt, .xlsx/.xls, and .RData/.rda. A .zip may contain any number of these (e.g. many .csv or many .txt); all are read and row-bound. For .txt files, **each file is treated as one document** (its doc_id is the file name); this makes a zip of .txt files a natural multi-document collection.

Value

A data frame with doc_id (character) and text (character) as the first two columns, followed by any remaining source columns. Duplicated ids are made unique with a warning.

Examples

```
# The bundled 1000-document sample
path <- system.file("extdata", "sample_collection.csv", package = "themescopeR")
coll <- read_collection(path)
head(coll$doc_id, 3)

# Readable ids; the identifier from the file survives as `source_id`
coll <- read_collection(path, sequential_ids = TRUE)
head(coll[, c("doc_id", "source_id")], 3)

# A zip holding many .txt files, one document each
d <- file.path(tempdir(), "texts")
dir.create(d, showWarnings = FALSE)
writeLines("the cat sat on the mat", file.path(d, "doc1.txt"))
writeLines("the dog ran in the park", file.path(d, "doc2.txt"))
z <- file.path(tempdir(), "texts.zip")
utils::zip(z, files = list.files(d, full.names = TRUE), flags = "-qj")
read_collection(z)
unlink(c(d, z), recursive = TRUE)
```

---

read_themescope *Read a .themescope archive*

Description

Loads an analysis written by save_themescope(). Use summary() on the result for a table describing what the file contains, and archive$result for the themescope object itself (which plots, prints and coerces to a data frame as usual).

Usage

```
read_themescope(file)

## S3 method for class 'themescope_archive'
print(x, ...)

## S3 method for class 'themescope_archive'
summary(object, ...)
```

Arguments

file	Path to a .themescope file.
x	A themescope_archive object.
...	Ignored.
object	A themescope_archive object.

Value

An object of class themescope_archive: a list with result (the themescope object), words, collection, terms, communities, meta, created, package_version and version.

Functions

- print(themescope_archive): Print a one-screen overview of the archive.
- summary(themescope_archive): Describe the archive as a two-column data frame (field, value), covering the corpus, the parameters of the run, the vocabulary, the network and the lexicon. This is what the Shiny import window displays.

See Also

save_themescope().

Examples

```
words  <- readRDS(system.file("extdata", "demo_annotated.rds",
                              package = "themescopeR"))
result <- themescope(words, vocab_size = 300, seed = 1, verbose = FALSE)
f <- file.path(tempdir(), "demo.themescope")
save_themescope(result, f)

archive <- read_themescope(f)
summary(archive)
archive$result
unlink(f)
```

`save_themescope` *Save an analysis as a* `.themescope` *archive*

Description

Writes a finished analysis, together with the data it was computed from, to a single `.themescope` file. The archive is a compressed RDS holding the `themescope` object, the annotated words it was built on, the raw collection (optional), the term-level table with the scores used to draw the maps, the community-level table, and the parameters of the run.

Reopening the archive with `read_themescope()` restores maps, communities and top terms without re-annotating the corpus or re-running the pipeline.

Usage

```
save_themescope(
  x,
  file,
  words = NULL,
  collection = NULL,
  meta = list(),
  compress = "xz"
)
```

Arguments

`x`	A `themescope` object, as returned by `themescope()`.
`file`	Destination path. The extension `.themescope` is added when `file` has none.
`words`	Optional annotated words data frame (the `preprocess_texts()` output the analysis was run on). Storing it lets the archive be re-analysed with different parameters.
`collection`	Optional raw document collection (the `read_collection()` output), stored so the original texts travel with the results.
`meta`	Optional named list of extra information to record, for example `list(language = "english", treebank = "GUM", lexicon = "Brysbaert")`. Whatever is supplied is shown by `summary()` and by the import window of the Shiny GUI.
`compress`	Compression passed to `saveRDS()` (default `"xz"`).

Value

The path of the written file, invisibly.

See Also

`read_themescope()` to load an archive back.

Examples

```
words  <- readRDS(system.file("extdata", "demo_annotated.rds",
                              package = "themescopeR"))
result <- themescope(words, vocab_size = 300, seed = 1, verbose = FALSE)
f <- file.path(tempdir(), "demo.themescope")
save_themescope(result, f, meta = list(language = "english",
                                       lexicon = "Brysbaert"))
summary(read_themescope(f))
unlink(f)


# Store the annotated corpus alongside the results, so the archive can be
# re-analysed later with different parameters (slower: it compresses 60k rows)
g <- file.path(tempdir(), "demo-with-corpus.themescope")
save_themescope(result, g, words = words)
summary(read_themescope(g))
unlink(g)
```

`term_relevance` *Compute term relevance within communities*

Description

Implements the ThemeScope case-study term-relevance measure (Eq. 4), which combines lexical salience with the balance between a term's internal and external connectivity:

$$R_t(g_i) = \log(1 + a_t) \cdot \frac{s_t^{in}(g_i)}{s_t^{in}(g_i) + s_t^{out}(g_i)},$$

where a_t is term presence, s_t^{in} the total association strength linking t to terms within its own community, and s_t^{out} the total linking it to terms outside. Higher values identify terms that are both frequent and predominantly embedded within their community, downweighting generic terms shared across clusters.

Usage

```
term_relevance(graph, membership, presence)
```

Arguments

`graph`	An undirected weighted `igraph` object.
`membership`	Named integer vector of community assignments (`NA` for unassigned vertices), as returned by `detect_communities()`.
`presence`	Named numeric vector of term presence a_t.

Value

A data frame with columns `term`, `community` (e.g. `"C1"`), `relevance`, `degree`, `presence`, ordered by community then decreasing relevance. Unassigned vertices are dropped.

Examples

```
words  <- readRDS(system.file("extdata", "demo_annotated.rds",
                              package = "themescopeR"))
result <- themescope(words, vocab_size = 300, seed = 1, verbose = FALSE)
rel <- term_relevance(result$graph, result$membership, result$presence)
head(rel)
```

`themescope` *Run the full ThemeScope analysis pipeline*

Description

Executes the complete ThemeScope workflow in a single call. Accepts **either** an annotated words data frame **or** a raw document collection (in which case it is annotated with **udpipe** via `preprocess_texts()` first). The steps:

1. (Optional) annotate raw texts into words.
2. Build the vocabulary from POS-filtered words (`build_vocab()`).
3. Build the co-occurrence matrix + term presence, with the chosen `normalization` (`build_cooccurrence_matrix()`).
4. Construct a thresholded network (`build_cooccurrence_network()`).
5. Detect communities (walktrap / louvain / leiden).
6. Compute PSI (anchoring) and CS (objectification) per community.
7. Collect network statistics.

Usage

```
themescope(
  data,
  model = NULL,
  text_col = "text",
  doc_id_col = "doc_id",
  unit = c("lemma", "token"),
  concreteness_lexicon = brysbaert,
  vocab_size = 1500,
  pos_filter = c("NOUN", "ADJ", "PROPN"),
  window = c("sentence", "document"),
  normalization = "association",
  threshold_percentile = 0.98,
  community_algorithm = "walktrap",
  walktrap_steps = 4,
```

```
  resolution = 1,
  min_community_size = 10,
  seed = NULL,
  verbose = TRUE
)

## S3 method for class 'themescope'
print(x, ...)

## S3 method for class 'themescope'
summary(object, ...)

## S3 method for class 'themescope'
plot(
  x,
  type = c("map", "network"),
  label = c("id", "terms"),
  label_by = c("relevance", "frequency", "degree"),
  n_label_terms = 3,
  ...
)

## S3 method for class 'themescope'
as.data.frame(x, ...)
```

Arguments

`data`	Either an annotated words data frame (columns `doc_id`, `sentence_id`, `upos`, and `token`/`lemma`) or a raw collection (a `text` and `doc_id` column, e.g. from `read_collection()`).
`model`	Required only when `data` is a raw collection: a **udpipe** model object, a path to a `.udpipe` file, or a language name (see `preprocess_texts()`).
`text_col`, `doc_id_col`	Column names used when annotating a raw collection.
`unit`	Word unit for the vocabulary: `"lemma"` (default) or `"token"`.
`concreteness_lexicon`	Data frame with columns `"word"` and `"conc.m"`. Defaults to the bundled brysbaert lexicon. Pass `NULL` to skip CS.
`vocab_size`	Integer or `NULL`. Maximum vocabulary size (`NULL` = all).
`pos_filter`	Character vector of Universal POS tags (default `c("NOUN", "ADJ", "PROPN")`).
`window`	Co-occurrence unit: `"sentence"` (default) or `"document"`.
`normalization`	Similarity measure for the co-occurrence matrix: `"association"` (default), `"equivalence"`, `"jaccard"`, `"salton"`, `"inclusion"` or `"frequency"`. See `normalize_cooccurrence()`.
`threshold_percentile`	Numeric in $(0, 1)$; network edge threshold (default `0.98`).
`community_algorithm`	`"walktrap"` (default), `"louvain"` or `"leiden"`.

walktrap_steps	Integer (default 4); used by walktrap only.
resolution	Numeric resolution for louvain/leiden (default 1).
min_community_size	Integer (default 10).
seed	Optional integer seed for reproducible community detection.
verbose	Logical. Print progress messages (default TRUE).
x	A themescope object.
...	Ignored.
object	A themescope object.
type	Character. "map" (default) or "network".
label	Map point labels: "id" (community id, default) or "terms" (the top terms of each community).
label_by	Ranking used to pick the terms when label = "terms": "relevance" (default, R_t), "frequency" (term presence) or "degree". Passed to top_terms().
n_label_terms	Integer. Number of terms per label when label = "terms" (default 3).

Value

An S3 object of class "themescope" with elements graph, communities, membership, psi, cs, presence, network_stats, vocab (a build_vocab() data frame), params, and call.

Methods (by generic)

- print(themescope): Print a concise summary of a themescope object.
- summary(themescope): Detailed summary with a community statistics table.
- plot(themescope): Plot a themescope object (type = "map" or "network"). With label = "terms", communities on the map are labelled with their most representative terms (up to n_label_terms), chosen with the label_by method, instead of the community id; communities keep the same colour in both views.
- as.data.frame(themescope): Coerce to a tidy community-level data frame (raw and z-scored metrics plus quadrant).

Examples

```
# The bundled demo corpus is shipped already tokenised and POS tagged
words  <- readRDS(system.file("extdata", "demo_annotated.rds",
                              package = "themescopeR"))
result <- themescope(words, vocab_size = 300, seed = 1, verbose = FALSE)
result
as.data.frame(result)


# Straight from raw texts: annotated internally, which downloads a udpipe
# language model on first use (kept in the session temporary directory here)
coll  <- read_collection(system.file("extdata", "sample_collection.csv",
                                     package = "themescopeR"))
```

```
model <- try(ts_download_model("english", model_dir = tempdir()), silent = TRUE)
if (!inherits(model, "try-error")) {
  res <- themescope(utils::head(coll, 100), model = model, vocab_size = 100,
                    threshold_percentile = 0.9, min_community_size = 3,
                    seed = 1, verbose = FALSE)
  plot(res, type = "map")
}
```

`themescope_app` *Launch the themescopeR Shiny application*

Description

Opens the optional graphical interface to the ThemeScope pipeline. The app is a thin front-end: it delegates all computation to the exported package functions (`read_collection()`, `preprocess_texts()`, `themescope()`, `top_terms()`, `save_themescope()`, `read_themescope()`), so the GUI reproduces the console results exactly.

Usage

```
themescope_app(display_mode = "normal", max_upload_size_mb = NULL, ...)
```

Arguments

`display_mode`	Passed to `shiny::runApp()` (`"normal"` by default; `"showcase"` displays the app code alongside it).
`max_upload_size_mb`	Optional numeric. Baseline maximum file-upload size (in megabytes) for the GUI. Shiny's own default is only 5 MB, far too small for typical ThemeScope collections, so the app already raises this to 50 MB. Set a larger value here to lift the baseline further from R (e.g. `2048` for 2 GB). The app also raises it interactively: picking a file larger than the current limit opens a window that suggests a new limit, warns about the memory implications, and imports the file once confirmed.
`...`	Further arguments passed to `shiny::runApp()` (e.g. `port`, `launch.browser`).

Details

The interface follows the pipeline step by step: import (raw text, a saved `.themescope` archive, or the bundled demo corpus, which ships already annotated), preprocess, run, and refine the look of the network without recomputing anything. Results are shown as an interactive map and network, a filterable community table and per-cluster term lists, and the whole study can be exported as a `.themescope` archive.

The GUI requires a few additional packages declared in `Suggests`: **shiny**, **bslib**, **DT**, **plotly**, **visNetwork** and **shinycssloaders**. Install any that are missing with `install.packages()`.

Value

Called for its side effect of launching the app; does not return a useful value.

Examples

```
if (interactive()) {
  themescope_app()
}
```

`themescope_cache_dir` *Location of the themescopeR model cache*

Description

Returns the directory where downloaded **udpipe** language models are cached, so a model is fetched only once and reused across analyses. Defaults to `tools::R_user_dir("themescopeR", "cache")`, the location R reserves for package caches; override it with `options(themescopeR.model_dir = "/path")`.

Usage

```
themescope_cache_dir(create = FALSE)
```

Arguments

`create`	Logical. Create the directory if it does not exist (default `FALSE`).

Details

Asking where the cache is does **not** create anything: the directory is created only when a model is actually downloaded, that is when you call `ts_download_model()` or hand a language name to `preprocess_texts()`. Nothing else in the package writes outside the session temporary directory. Use `ts_clear_cache()` to delete what has been cached.

Value

A character path to the cache directory.

See Also

`ts_clear_cache()` to remove cached models, `ts_download_model()` to populate the cache.

Examples

```
# Where models would be cached; this call creates nothing
themescope_cache_dir()
```

themescope_colours *ThemeScope pastel colour palette*

Description

Returns the built-in palette of 40 pastel colours used to colour communities consistently across `plot_themescope()` and `plot_network()`.

Usage

```
themescope_colours(n = NULL)
```

Arguments

`n`	Optional number of colours to return (recycled if `n > 40`).

Value

A character vector of hex colours.

Examples

```
themescope_colours(5)
```

top_terms *Top terms per community*

Description

Returns the highest-ranked terms of each community, used to label and interpret the thematic clusters. Terms can be ranked by three criteria:

`"relevance"` **(default)** the ThemeScope case-study relevance measure $R_t(g_i)$ (`term_relevance()`), which combines lexical salience with a term's internal-versus-external connectivity. This is the labelling method described in the ThemeScope case study: it downweights generic, high-frequency terms that co-occur across several clusters and favours terms that are both frequent and structurally embedded in their own community.

`"frequency"` the term presence a_t (the number of sentences, or documents, in which the term occurs), the salience component of R_t on its own, i.e. plain frequency-based labelling.

`"degree"` the network degree of the term (number of co-occurrence links).

Usage

```
top_terms(x, n = 10, by = c("relevance", "frequency", "degree"))
```

Arguments

x	A themescope object (from themescope()).
n	Integer. Number of terms to return per community (default 10).
by	Ranking criterion: "relevance" (default, R_t), "frequency" (term presence a_t) or "degree".

Value

A data frame with columns community, rank, term, relevance, frequency (term presence a_t) and degree, ordered by community and then by the chosen ranking criterion (decreasing).

See Also

term_relevance() for the relevance measure R_t.

Examples

```
words  <- readRDS(system.file("extdata", "demo_annotated.rds",
                              package = "themescopeR"))
result <- themescope(words, vocab_size = 300, seed = 1, verbose = FALSE)
top_terms(result, n = 5)                    # by relevance R_t (default)
top_terms(result, n = 5, by = "frequency")  # by term presence a_t
```

---

ts_clear_cache *Delete cached language models*

Description

Removes the models and the registry downloaded by ts_download_model() and ts_list_models(), and the cache directory itself. Nothing else in the package writes persistently, so this clears everything themescopeR has ever stored on disk.

Usage

```
ts_clear_cache(model_dir = NULL, ask = interactive())
```

Arguments

model_dir	Directory to clear. Defaults to themescope_cache_dir().
ask	Logical. Ask for confirmation before deleting. Defaults to interactive().

Value

The paths that were removed, invisibly (character(0) if the cache was empty or the user declined).

See Also

themescope_cache_dir(), ts_download_model().

Examples

```
# Point the cache at a throwaway directory and clear it again
d <- file.path(tempdir(), "themescope-cache-example")
dir.create(d, showWarnings = FALSE)
file.create(file.path(d, "example-ud-2.15.udpipe"))
ts_clear_cache(model_dir = d, ask = FALSE)
dir.exists(d)
```

---

ts_download_model *Download and cache an updated udpipe language model*

Description

Downloads a Universal Dependencies 2.15 **udpipe** model from the TALL language-model collection into themescope_cache_dir(). If the model is already cached it is **not** re-downloaded, so models are fetched only once and reused across analyses.

Usage

```
ts_download_model(
  language,
  treebank = NULL,
  model_dir = NULL,
  overwrite = FALSE
)
```

Arguments

language	Character language name (e.g. "english", "italian"). See ts_list_models().
treebank	Optional treebank name (e.g. "EWT", "ISDT"). If NULL, the first model listed for the language is used.
model_dir	Directory to download into. If NULL (default) the model is cached in themescope_cache_dir(), which is created at that point and only then. Pass a path of your own (for instance tempdir()) to keep the download inside the current session.
overwrite	Logical. Re-download even if cached (default FALSE).

Details

The updated models and lexicons are curated and maintained by **Massimo Aria** in the `tall.language.models` repository, part of the TALL project. We gratefully acknowledge that work and redistribute nothing here: models are fetched on demand from the source repository.

To explore which models are available, together with their treebanks, contributors, descriptions, corpus sizes and Universal Dependencies hub pages, use `ts_list_models()` (whose returned data frame includes `description` and `hub_page_link` columns), or browse the repository directly: https://github.com/massimoaria/tall.language.models.

Value

The path to the cached `.udpipe` model file (invisibly).

References

Aria, M. *tall.language.models*: updated UDPipe models and lexicons for TALL. https://github.com/massimoaria/tall.language.models

See Also

`ts_list_models()` for the full catalogue (descriptions and links), `ts_clear_cache()` to delete the downloaded models.

Examples

```
# Downloaded into the session temporary directory, so nothing persists
path <- try(ts_download_model("english", model_dir = tempdir()), silent = TRUE)
if (!inherits(path, "try-error")) basename(path)
```

---

ts_list_models *List the available updated language models*

Description

Retrieves the registry of updated **udpipe** models (Universal Dependencies 2.15) maintained for TALL. The registry is downloaded once, cached in `themescope_cache_dir()` and reused on later calls; the cache directory is created only at that point.

Usage

```
ts_list_models(refresh = FALSE, cache_dir = NULL)
```

Arguments

`refresh`	Logical. Force a re-download of the registry (default `FALSE`).
`cache_dir`	Directory to keep the registry in. If `NULL` (default) it goes to `themescope_cache_dir()`, created at that point and only then.

Value

A data frame with one row per model, including `language_name`, `treebank` and `file` (the model code used to build the file name).

Examples

```
# Needs an internet connection the first time; cached afterwards. Kept in the
# session temporary directory here, so nothing persists.
models <- try(ts_list_models(cache_dir = tempdir()), silent = TRUE)
if (!inherits(models, "try-error")) {
  head(models[, c("language_name", "treebank")])
}
```

`ts_model_path` *Resolve the cached path of an updated udpipe model*

Description

Returns the path to a cached model without downloading anything and without creating any directory.

Usage

```
ts_model_path(language, treebank = NULL, model_dir = NULL)
```

Arguments

`language`	Character language name (e.g. `"english"`, `"italian"`). See `ts_list_models()`.
`treebank`	Optional treebank name (e.g. `"EWT"`, `"ISDT"`). If `NULL`, the first model listed for the language is used.
`model_dir`	Directory to download into. If `NULL` (default) the model is cached in `themescope_cache_dir()`, which is created at that point and only then. Pass a path of your own (for instance `tempdir()`) to keep the download inside the current session.

Value

The path to the cached `.udpipe` file, or `NA` if it is not cached.

Examples

```
# NA until the model has been downloaded (needs the catalogue, hence a
# connection on first use)
p <- try(ts_model_path("english"), silent = TRUE)
if (!inherits(p, "try-error")) p
```

validate_words_df *Validate an annotated words data frame*

Description

Checks that an annotated data frame has all columns required by the ThemeScope backend. The "word" used downstream is either the `token` or the `lemma` column (chosen via the `unit` argument of `build_vocab()` / `build_cooccurrence_matrix()`), so at least one of them must be present. Emits informative errors via **cli** when columns are missing.

Usage

```
validate_words_df(words_df)
```

Arguments

`words_df`	Data frame to validate. Must contain `doc_id`, `sentence_id`, `upos`, and at least one of `token` or `lemma`.

Value

Invisibly returns `TRUE` if validation passes.

Examples

```
df <- data.frame(doc_id = 1, sentence_id = 1, token = "dog", upos = "NOUN")
validate_words_df(df)
```

zscore *Compute z-scores*

Description

Computes standardised (z-scored) values, handling `NA`s gracefully.

Usage

```
zscore(x)
```

Arguments

`x`	Numeric vector to standardise.

Value

Numeric vector of z-scores. `NA` values in the input remain `NA`. Returns a vector of `NA`s (with a warning) if the standard deviation is zero.

Examples

```
zscore(c(1, 2, 3, 4, 5))
zscore(c(1, NA, 3))
```

Index