Package: joinery (via r-universe)

July 8, 2026

Type Package

Title Heuristic Index-Based Record Linkage

Version 1.0.0.9000

Description Links records that refer to the same entity across sources that share no common key, such as people, firms, or addresses with spelling variation, abbreviations, or reordered words. Linkage is described declaratively as a strategy that normalises, tokenises, phonetically encodes, weights, and blocks each field; candidate pairs are then scored by the rarity-weighted overlap of their tokens and every score is attributed back to individual tokens for explainability. Strategies compose into staged pipelines of exact, fuzzy, and optional embedding-based matching that carry unmatched records forward and resolve entities as connected components. The same strategy runs on an in-memory 'data.table' backend or an out-of-core 'DuckDB' backend, and diagnostic and calibration tools help tune a strategy and filter false positives. The token-retrieval heuristic follows Doherr (2023) <doi:10.2139/ssrn.4326848>.

URL https://edubruell.github.io/joinery/, https://github.com/edubruell/joinery

BugReports https://github.com/edubruell/joinery/issues

License MIT + file LICENSE

Encoding UTF-8

Language en-GB

Imports S7, rlang (>= 1.1.0), data.table, igraph, stringi, phonics, lubridate, cli, glue, tinyplot, graphics

Collate 'joinery-package.R' 'import-standalone-purrr.R' 'import-standalone-types-check.R' 'import-standalone-obj-type.R' 'internal_validation.R' 'internal_fanout.R' 'internal_progress.R' 'internal_chunking.R' 'internal_staging.R' 'strategy_step.R' 'strategy_smoothing.R'

'strategy_search.R' 'strategy_blocking.R'
'strategy_embedding.R' 'strategy_exact.R' 'generics_core.R'
'generics_calibration.R' 'generics_embedding.R'
'methods_datatable_prepare.R' 'methods_datatable_resolve.R'
'methods_datatable_materialize.R' 'methods_datatable_dedup.R'
'methods_datatable_search.R' 'methods_datatable_multistage.R'
'methods_datatable_inspect.R' 'embedding_cache.R'
'embedding_methods_datatable.R' 'exact_methods_datatable.R'
'methods_tibble.R' 'embedding_methods_tibble.R'
'preparer_word.R' 'preparer_tokens.R' 'duckdb_control.R'
'methods_duckdb_batch.R' 'methods_duckdb_prepare.R'
'methods_duckdb_resolve.R' 'methods_duckdb_materialize.R'
'methods_duckdb_dedup.R' 'methods_duckdb_search.R'
'methods_duckdb_multistage.R' 'methods_duckdb_inspect.R'
'preparer_stopwords.R' 'preparer_subword.R'
'embedding_methods_duckdb.R' 'exact_methods_duckdb.R'
'calibration_aip.R' 'generics_diagnostic.R'
'diagnostic_classes.R' 'calibration_classes.R'
'diagnostic_recommendations.R' 'diagnostic_summarise.R'
'diagnostic_audit.R' 'diagnostic_compare.R'
'diagnostic_explain.R' 'diagnostic_sample.R'
'diagnostic_rarity.R' 'plan_strategy.R'
'calibration_labelling.R' 'calibration_features.R'
'calibration_features_embedding.R' 'calibration_filter.R'
'calibration_tidymodels.R' 'calibration_dispatch.R'
'calibration_calibrate.R' 'calibration_recipe.R'
'diagnostic_plots.R' 'data.R'

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.3

Depends R (>= 4.1.0)

LazyData true

VignetteBuilder knitr

Suggests duckdb, DBI, dbplyr, dplyr, tidyllm, tibble, stringdist, sentencepiece, generics, parsnip, recipes, workflows, yardstick, probably, testthat (>= 3.0.0), withr, knitr, rmarkdown

Config/pak/sysreqs libglpk-dev libicu-dev libxml2-dev

Repository https://edubruell.r-universe.dev

Date/Publication 2026-07-07 10:13:59 UTC

RemoteUrl https://github.com/edubruell/joinery

RemoteRef HEAD

RemoteSha 2ee7c82c52a508d68aa4cae056c49ed673518c4a

Contents

ambiguity_plot *Bar chart of candidates-per-record distribution (candidates only)*

Description

Bar chart of candidates-per-record distribution (candidates only)

Usage

```
ambiguity_plot(x, ...)
```

Arguments

x	A Match_Overview object from summarise_matches() with match_type == "candidates".
...	Passed to tinyplot::tinyplot().

Value

Invisibly, the plotted data.table (ambiguity_dist).

apply_filter *Apply a fitted filter to match features*

Description

Score a Match_Features table with a fitted Filter_Model and return a Calibrated_Matches object. When matches is supplied, the original match table is enriched with tp_prob and predicted_tp columns and stored in the result's @matches slot; when matches is NULL, the features table itself is enriched and stored.

Usage

```
apply_filter(
  features,
  filter_model,
  threshold = NULL,
  threshold_rule = c("youden", "target_recall", "cost_weighted"),
  target_recall = 0.95,
  cost_ratio = 1,
  matches = NULL,
  ...
)
```

Arguments

features	A Match_Features object.
filter_model	A Filter_Model produced by fit_filter().
threshold	Numeric scalar in (0, 1) or NULL. When non-NULL it is used verbatim and overrides threshold_rule. When NULL, the threshold is chosen on the training labels per threshold_rule. Decision 13.7 default.
threshold_rule	The operating-point rule used when threshold is NULL: "youden" (default, maximise Youden's J, symmetric error costs), "target_recall" (the highest threshold still achieving target_recall), or "cost_weighted" (minimise cost_ratio * FN + FP). For a firm panel the recall-favouring rules are usually the right operating point, splitting one business across years is worse than admitting a few co-located firms a later collapse can still catch.
target_recall	Target recall in (0, 1] for threshold_rule = "target_recall". Default 0.95.
cost_ratio	cost(FN) / cost(FP) for threshold_rule = "cost_weighted"; > 1 favours recall. Default 1 (symmetric).
matches	Optional raw matches table to enrich. When supplied, tp_prob / predicted_tp are broadcast onto every row of the pair (candidates: both source == "base" and source == "target" rows of a match_id; duplicates: every row of a duplicate_group).
...	Reserved for future expansion.

Value

A Calibrated_Matches object.

Examples

```
strat <- search_strategy(
  workshop   ~ normalize_text() + word_tokens(min_nchar = 3),
  proprietor ~ normalize_text() + word_tokens(min_nchar = 2),
  block_by  = c("postcode_area", "trade"),
  threshold = 0.30
)
matches <- search_candidates(
  workshop_listings, workshop_register,
  base_id = "listing_id", target_id = "reg_no", strategy = strat
)
feats <- match_features(matches, strat,
                        base = workshop_listings, id = "listing_id",
                        target = workshop_register, target_id = "reg_no")
model <- fit_filter(feats, match_labels_example)
# Broadcast the true-positive probability back onto the match rows.
apply_filter(feats, model, matches = matches)
```

`approximate_date` *Approximate dates by rounding to coarser time units*

Description

`approximate_date()` rounds dates to the start of broader time periods (month, quarter, half-year, year, or decade). This is useful for fuzzy temporal matching when exact dates may differ slightly but represent the same general time period.

Usage

```
approximate_date(
  x,
  unit = c("month", "quarter", "half", "year", "decade"),
  format = NULL,
  orders = c("ymd", "dmy", "mdy")
)
```

Arguments

`x`	A character or Date vector containing dates to approximate.
`unit`	Character string specifying the rounding unit. One of: • `"month"` – round to first day of month (default) • `"quarter"` – round to first day of quarter (Jan 1, Apr 1, Jul 1, Oct 1) • `"half"` – round to first day of half-year (Jan 1 or Jul 1) • `"year"` – round to January 1 • `"decade"` – round to first year of decade (e.g., 2020-01-01)
`format`	Optional format string for parsing (passed to `as.Date()`). If `NULL` (default), attempts automatic parsing via lubridate.
`orders`	Optional character vector of lubridate order specifications. Used when `format = NULL`. Defaults to `c("ymd", "dmy", "mdy")`.

Details

Rounding always goes to the **start** of the period:

- `"month"`: 2023-03-15 -> 2023-03-01
- `"quarter"`: 2023-03-15 -> 2023-01-01 (Q1), 2023-05-20 -> 2023-04-01 (Q2)
- `"half"`: 2023-03-15 -> 2023-01-01 (H1), 2023-08-20 -> 2023-07-01 (H2)
- `"year"`: 2023-03-15 -> 2023-01-01
- `"decade"`: 2023-03-15 -> 2020-01-01

Value

A character vector of dates in ISO 8601 format (YYYY-MM-DD), rounded to the start of the specified time unit. Unparseable dates return `NA_character_` with a warning.

See Also

`normalize_date()` for exact dates, `date_tokens()` to split a date into part tokens.

Other date preparers: `date_tokens()`, `normalize_date()`

Examples

```
approximate_date("2023-03-15", unit = "month")
# "2023-03-01"

approximate_date("2023-03-15", unit = "quarter")
# "2023-01-01"

approximate_date("2023-08-20", unit = "half")
# "2023-07-01"

approximate_date("2023-03-15", unit = "year")
# "2023-01-01"

approximate_date("2023-03-15", unit = "decade")
# "2020-01-01"

approximate_date(c("2023-01-15", "2023-04-20", "2023-09-10"), unit = "quarter")
# c("2023-01-01", "2023-04-01", "2023-07-01")
```

---

as_cologne *Encode text phonetically with the Cologne procedure*

Description

The Cologne phonetic procedure (Koelner Phonetik) is the German-language counterpart to Soundex. It maps text to a digit string by German pronunciation rules, so variants like `"Meier"`, `"Maier"`, and `"Mayer"` share one key. Reach for this over `as_soundex()` or `as_metaphone()` when the data is German.

Usage

```
as_cologne(text)
```

Arguments

`text`	A character string or vector to encode, or a token list-column (one character vector of tokens per row) when the encoder is placed *after* a token generator – each token is then encoded in place.

Details

Returns text, so it slots ahead of a token generator, or use it directly on a one-word column. Like any phonetic key it favours recall over precision; pair it with a sharper field rather than matching on the key alone.

Value

A character vector of Cologne phonetic keys (digit strings), one per input element.

See Also

Other phonetic encoders: `as_metaphone()`, `as_soundex()`

Examples

```
as_cologne(c("Meier", "Maier", "Mayer"))  # same key
```

`as_metaphone` *Encode text phonetically with Metaphone*

Description

Names that sound alike are often spelled differently: `"Smith"` and `"Smyth"`, `"Meyer"` and `"Maier"`. Metaphone encodes text by how it sounds, so those variants share one key and match even though the letters differ. Best on single-word fields such as surnames or company names; it is tuned for English pronunciation (for German, see `as_cologne()`).

Usage

```
as_metaphone(text)
```

Arguments

`text`	A character string or vector to encode, or a token list-column (one character vector of tokens per row) when the encoder is placed *after* a token generator – each token is then encoded in place.

Details

Runs on either side of a token generator: ahead of one (on a text column), or after one (on a token column, encoding each token in place). Phonetic keys are deliberately coarse, so they trade precision for recall: pair them with a sharper field rather than matching on a phonetic key alone.

Value

A character vector of Metaphone keys, one per input element.

See Also

Other phonetic encoders: `as_cologne()`, `as_soundex()`

Examples

```
as_metaphone("Smith")
as_metaphone(c("Meyer", "Maier"))  # same key
```

---

as_soundex *Encode text phonetically with Soundex*

Description

Soundex is the classic phonetic code: it keeps the first letter and reduces the rest to a short digit string (for example `"Robert"` and `"Rupert"` both become `"R163"`), so spellings that sound alike share one key. It is coarser and older than Metaphone but widely understood and a good default for English surnames.

Usage

```
as_soundex(text)
```

Arguments

`text`	A character string or vector to encode, or a token list-column (one character vector of tokens per row) when the encoder is placed *after* a token generator – each token is then encoded in place.

Details

Runs on either side of a token generator: ahead of one (on a text column), or after one (on a token column, encoding each token in place). As with any phonetic key it favours recall over precision; pair it with a sharper field rather than matching on the key alone.

Value

A character vector of Soundex keys (letter followed by digits), one per input element.

See Also

Other phonetic encoders: `as_cologne()`, `as_metaphone()`

Examples

```
as_soundex("Robert")
as_soundex(c("Robert", "Rupert"))  # same key
```

audit_strategy *Audit a Search Strategy Against Data*

Description

Pre-match diagnostic (Q1). Runs preparation and rarity computation, reports per-column token / rarity statistics and (when `block_by` is set) block-size distribution and estimated comparison count. Surfaces recommendations linking pre-match symptoms to strategy levers.

Usage

```
audit_strategy(data, id, strategy, ...)
```

Arguments

`data`	A data.frame / tibble / data.table (or backend-specific table).
`id`	Character scalar naming the ID column in `data`.
`strategy`	A `Search_Strategy` object.
`...`	Additional backend-specific arguments. Notably: `target` (optional second table for cross-table vocabulary overlap) and `sample_n` (optional integer; if set, audit a random sample of rows).

Value

A `Strategy_Audit` object.

Examples

```
strat <- search_strategy(
  workshop ~ normalize_text() + word_tokens(min_nchar = 3),
  block_by  = c("postcode_area", "trade"),
  threshold = 0.7
)
audit_strategy(workshop_register, "reg_no", strat)
```

base_example *Base dataset for record linkage example*

Description

A dataset containing 3,300 person records with German, Turkish, and Polish names, including addresses across various German cities. Approximately 5% of records are intentional duplicates with small variations to simulate real-world data quality issues.

Usage

```
base_example
```

Format

A tibble with 3,300 rows and 7 variables:

id_base Character identifier for base records (B0001-B3150)

Vorname First name, weighted by ethnic group prevalence

Nachname Last name, weighted by ethnic group prevalence

Strasse Street name, including German street types

Hausnummer House number, some with letter suffixes

Ort City or town name

Kreis Administrative district (Kreis)

Source

Synthetically generated using weighted sampling from common German, Turkish, and Polish names (93%, 4%, and 3% respectively) and realistic German geography.

---

batch_map *Apply a function to DuckDB table batches*

Description

Streams a DuckDB table through a batch plan and applies a user-defined function to each batch. The function must accept a data.frame and return a data.frame. Results can be collected in memory or written back to DuckDB incrementally.

Usage

```
batch_map(plan, con, input_table, fn, persist = TRUE, output_table = NULL)
```

Arguments

`plan`	A batch plan produced by `duckdb_batch_plan()`. Must include columns `batch_id` and `row_count`, plus either row-number windows (`row_start`, `row_end`), block identifier columns, or a `blocks` list-column for consolidated batches.
`con`	A DuckDB connection.
`input_table`	Character. Name of the source table in DuckDB.
`fn`	A function applied to each batch. Receives a data.frame and must return a data.frame.
`persist`	Logical. If `TRUE`, results of each batch are appended to `output_table` inside DuckDB and a lazy table reference is returned. If `FALSE`, returns a list of batch results as data.frames.

`output_table` Optional DuckDB table name where results are stored when `persist = TRUE`. If omitted, a random temporary table name is generated. Ignored when `persist = FALSE`.

Details

Database work is performed batch-by-batch, allowing preprocessing of tables that exceed available RAM. For each batch, a SQL slice or block filter is executed, the function is applied, and (optionally) results are appended to a DuckDB table.

Value

- If `persist = TRUE`: A `tbl_duckdb_connection` pointing to the output table.
- If `persist = FALSE`: A list of data.frames, one per batch.

`block_on_tokens` *Block on a Column's Rare Tokens (region-free blocking)*

Description

Build a token-blocking key for use inside a strategy's `block_by`. Where a plain column name blocks two records only when they share a literal value, `block_on_tokens()` blocks them when they share **any** of a designated column's (rare) tokens. This is **region-free**: a record that drifts across a region boundary - a firm that moves to a new postcode, say - still co-blocks with its earlier self through a distinctive name token, and so becomes a candidate where a literal block would never compare them.

Hand it to `block_by` in place of (or mixed with) a column name:

```
# fully region-free - share a rare name token, regardless of place
search_strategy(name ~ normalize_text + word_tokens(min_nchar = 3),
                block_by = block_on_tokens("name", max_df = 50))

# region-bounded - share a rare name token AND sit in the same plz2
search_strategy(name ~ normalize_text + word_tokens(min_nchar = 3),
                block_by = list(block_on_tokens("name", max_df = 50), "plz2"))
```

`max_df` and `min_rarity` select which tokens are eligible block keys, using the **global** (corpus-wide) document frequency: a token appearing in more than `max_df` records, or whose global rarity falls below `min_rarity`, is dropped as a key. This is where "block on the distinctive words, not the common ones" lives - a franchise name ("ALDI") is globally common, fails the cap, and never becomes a block key, while a distinctive brand survives. A record with no surviving block key is **unreachable via token-blocking in this stage** (it contributes no token-block rows).

Token-blocking is the densest operation in the package: every pair sharing a surviving key is materialised. It is safe **only** behind a real `max_df` (or `min_rarity`) plus the always-on fan-out guard. Passing neither cap is a loud warning, not an error, but you almost always want one.

Usage

```
block_on_tokens(
  column,
  max_df = Inf,
  min_rarity = 0,
  preparer = NULL,
  min_nchar = 3L
)
```

Arguments

column	The column whose tokens become block keys (for example "name").
max_df	Numeric scalar. Global document-frequency cap: tokens appearing in more than max_df records corpus-wide are dropped as block keys. Default Inf (no cap - see the density warning above).
min_rarity	Numeric scalar. Global rarity floor: tokens whose corpus-wide rarity falls below this are dropped as block keys. Default 0.
preparer	Optional preprocessing pipeline for the blocking column, given as a one-sided or two-sided formula like the column ~ steps you pass to search_strategy() (for example ~ normalize_text + word_tokens(min_nchar = 4)). Default NULL reuses the column's own scored preparer when column is also a scored column, else falls back to normalize_text + word_tokens(min_nchar = min_nchar).
min_nchar	Integer scalar. Minimum token length for the default preparer. Default 3L.

Value

A Block_On_Tokens spec, to be placed in block_by.

See Also

search_strategy()

Examples

```
# Block on a rare word from the workshop name instead of a region, so a
# workshop still co-blocks with its relocated self. The max_df cap keeps
# common words ("joinery") from becoming block keys.
strat <- search_strategy(
  workshop ~ normalize_text() + word_tokens(min_nchar = 3),
  block_by     = list(block_on_tokens("workshop", max_df = 50, min_nchar = 4),
                      "trade"),
  rarity_scope = "global",
  threshold    = 0.6
)
strat
```

block_size_plot *Bar chart of block sizes (requires block_by on strategy)*

Description

Bar chart of block sizes (requires block_by on strategy)

Usage

```
block_size_plot(x, ...)
```

Arguments

x	A Strategy_Audit object from audit_strategy().
...	Passed to tinyplot::tinyplot().

Value

Invisibly, the plotted data.table (block_summary$distribution).

calibrate *Evaluate a fitted filter on labelled pairs*

Description

Compute calibration diagnostics for a fitted false-positive filter on a labelled evaluation set. Returns a Filter_Calibration carrying the reliability table, Brier score, log-loss, per-class confusion matrix, and a threshold sweep curve.

Two call shapes:

- calibrate(calibrated_matches, labels) - evaluate on labels held out from the training fit.
- calibrate(calibrated_matches) - evaluate on the training labels stored on the Filter_Model (sanity-check view; do not use for model selection).

Usage

```
calibrate(x, labels = NULL, bins = 10L, ...)
```

Arguments

x	A Calibrated_Matches object from apply_filter() / calibrate_matches().
labels	Optional labels data.table (typically from import_labels()) for held-out evaluation.
bins	Integer. Number of equal-width probability bins for the reliability table. Default 10.
...	Reserved for future expansion.

Value

A Filter_Calibration object.

---

calibrate_matches *Calibrate matches end-to-end (features -> filter -> apply)*

Description

High-level Q5 verb. Builds features via match_features(), fits a Filter_Model via fit_filter(), and applies it via apply_filter() to return a Calibrated_Matches object enriched with tp_prob / predicted_tp. Dispatches on the strategy class.

Usage

```
calibrate_matches(matches, strategy, ...)
```

Arguments

matches	Match output table (data.table / tibble / data.frame / DuckDB lazy tbl).
strategy	The search_strategy() or embedding_strategy() used to produce matches.
...	Method-specific arguments. Required: labels (manually labelled rows produced by import_labels()), base, and id. Optional: target, target_id (forwarded to match_features()), model, class_weighted, na_fill, threshold, plus all match_features() tuning knobs (top_n, include_string_sim, include_block_stats, method).

Value

A Calibrated_Matches object.

Examples

```
strat <- search_strategy(
  workshop   ~ normalize_text() + word_tokens(min_nchar = 3),
  proprietor ~ normalize_text() + word_tokens(min_nchar = 2),
  block_by  = c("postcode_area", "trade"),
  threshold = 0.30
)
matches <- search_candidates(
  workshop_listings, workshop_register,
  base_id = "listing_id", target_id = "reg_no", strategy = strat
)
# One call: build features, fit the filter, apply it. Uses the shipped
# labelled pairs, which line up with this exact search.
calibrate_matches(matches, strat, labels = match_labels_example,
                  base = workshop_listings, id = "listing_id",
                  target = workshop_register, target_id = "reg_no")
```

clear_embedding_cache *Clear the embedding reuse cache*

Description

Empties joinery's in-session embedding cache, and optionally the on-disk cache. The cache stores raw embedding vectors so that the data.table and tibble backends reuse them instead of re-embedding on every call. You rarely need to call this by hand; it is mainly useful to force a clean re-embed or to reclaim memory in a long-running session.

Usage

```
clear_embedding_cache(disk = FALSE)
```

Arguments

disk	Logical. If TRUE, also delete the on-disk cache files in the directory set by options(joinery.embedding_cache_dir = ...). Defaults to FALSE (clear the in-session cache only).

Value

Invisibly NULL.

See Also

compute_embeddings() for how the cache is filled, and joinery (package options) for joinery.embedding_reuse and joinery.embedding_cache_dir.

cluster_size_plot *Bar chart of cluster-size distribution (duplicates only)*

Description

Bar chart of cluster-size distribution (duplicates only)

Usage

```
cluster_size_plot(x, ...)
```

Arguments

x	A Match_Overview object from summarise_matches() with match_type == "duplicates".
...	Passed to tinyplot::tinyplot().

Value

Invisibly, the plotted `data.table` (cluster_dist).

`compare_stages` *Compare Stages of a Multi-Stage Match*

Description

Multi-stage diagnostic. Produces per-stage `Match_Overview` objects, marginal coverage per stage, and overlaid per-stage score distributions. Note that `summarise_matches()` does **not** auto-detect a `stage` column - users explicitly call this verb when they want per-stage analysis (see `notes/diagnostics_design.md`).

Usage

```
compare_stages(matches, ...)
```

Arguments

`matches`	Multi-stage match table with a `stage` column.
`...`	Method-specific arguments. The data.table method will accept `base` and `target` for coverage.

Value

A `Stage_Comparison` object.

Examples

```
exact <- exact_strategy(
  workshop ~ normalize_text() + word_tokens(min_nchar = 3),
  block_by = c("postcode_area", "trade")
)
fuzzy <- search_strategy(
  workshop ~ normalize_text() + word_tokens(min_nchar = 3),
  block_by  = c("postcode_area", "trade"),
  threshold = 0.55
)
g <- multi_stage_search(
  workshop_panel, workshop_panel,
  base_id = "record_id", target_id = "record_id",
  list(exact = exact, fuzzy = fuzzy),
  self = TRUE, source_by = "year", collapse = "rep"
)
# See how much each pass added that earlier passes had not reached.
compare_stages(g, base = workshop_panel, target = workshop_panel)
```

compute_embeddings *Compute Embeddings for Records*

Description

Compute embedding vectors for records using an `Embedding_Strategy`. This is a backend-specific generic that handles data retrieval, text assembly, and embedding computation via tidyllm.

Embedding is the expensive part of a vector match, so each record is embedded once and the vector is reused on later calls. The data.table and tibble backends keep a per-session cache keyed by model and record content; the DuckDB backend reuses through its persisted `embeddings` column. Reuse is controlled by the `joinery.embedding_reuse` and `joinery.embedding_cache_dir` options (see `joinery` package options) and can be cleared with `clear_embedding_cache()`.

Usage

```
compute_embeddings(data, id, strategy, ...)
```

Arguments

`data`	A data.frame / tibble / data.table (or db table in other backends).
`id`	Character scalar naming the ID column in `data`.
`strategy`	An `Embedding_Strategy` object specifying columns, embedding model, and normalization settings.
`...`	Additional arguments passed to backend-specific methods.

Value

A backend-specific table with columns: `id` and `embedding` (where `embedding` contains numeric vectors).

See Also

`clear_embedding_cache()`

compute_rarity *Compute Token Rarity for Record Linkage*

Description

`compute_rarity()` assigns a rarity score to each token produced by `prepare_search_data()`, using the rarity method defined in a `Search_Strategy`.

Usage

```
compute_rarity(tokens, strategy, ...)
```

Arguments

tokens	A token table created by prepare_search_data(), in any backend-specific representation. Must contain at least column, token, and row_id, plus any block_by columns.
strategy	A Search_Strategy defining the rarity method, blocking variables, and field structure.
...	Additional arguments passed to backend-specific methods.

Details

Rarity quantifies how informative a token is when comparing records. In **joinery**, rarity is always computed:

- using **one global rarity metric** specified in the strategy,
- **per column**, because each field has its own token distribution,
- **within each block** (if the strategy specifies block_by).

The input tokens must be the long-format token table returned by prepare_search_data(), containing at minimum:

- an ID column,
- a column field indicating the source variable,
- a token field,
- a row_id identifying the originating record,
- and any block_by variables required by the strategy.

Backends (e.g., data.frame, data.table, DuckDB relations) may implement their own methods for this generic, but all must return the same logical structure: the original token table with an added numeric rarity column.

Value

The same token table with an added rarity column.

---

contribution_plot *Horizontal bar chart of per-column score contributions*

Description

Horizontal bar chart of per-column score contributions

Usage

```
contribution_plot(x, ...)
```

Arguments

x	A Match_Explanation object from explain_match().
...	Passed to tinyplot::tinyplot().

Value

Invisibly, the plotted data.table (per_column_contrib).

---

coverage_plot *Bar chart of match coverage (base and/or target)*

Description

Bar chart of match coverage (base and/or target)

Usage

```
coverage_plot(x, ...)
```

Arguments

x	A Match_Overview object from summarise_matches().
...	Passed to tinyplot::tinyplot().

Value

Invisibly, the plotted data.table.

---

custom_tokens *Tokenize with your own tokenizer*

Description

The built-in tokenizers split text into words, n-grams, or dates. When none of them fits your data, write your own and place it in the strategy formula with custom_tokens(): name ~ normalize_text() + custom_tokens(my_tokenizer).

Usage

```
custom_tokens(text, tokenizer)
```

Arguments

text	A character vector to tokenize.
tokenizer	A plain function, or an object with a tokenize() method.

Details

The tokenizer can be a plain function that takes a character vector and returns a list of token vectors. It can also be a fitted tokenizer object with a `tokenize()` method, which is how a tokenizer with learned state, such as a trained subword vocabulary, enters a strategy.

Whichever form you pass, it must return a list with one element per input string, each element a character vector of that record's tokens (`character(0)` for a record with none). `custom_tokens()` checks this and stops with a clear error if the result does not fit.

Value

A list of character vectors, one per input element.

See Also

`tokenize()`, the method a fitted tokenizer provides; `word_tokens()` for the standard whitespace tokenizer.

Other token generators: `generate_ngrams()`, `numeric_tokens()`, `subword_tokens()`, `word_tokens()`

Examples

```
# A bare-function tokenizer: split on vowels
split_vowels <- function(text) strsplit(text, "[aeiou]+")
custom_tokens(c("plexiglas", "brandenburg"), split_vowels)
```

---

date_tokens *Extract date components as tokens*

Description

`date_tokens()` parses dates and extracts specified components (year, month, day) as separate tokens. This is useful for flexible date matching where you want to match on specific date parts rather than full dates.

Usage

```
date_tokens(
  x,
  components = c("year", "month", "day"),
  format = NULL,
  orders = c("ymd", "dmy", "mdy")
)
```

Arguments

x	A character or Date vector containing dates to tokenize.
components	Character vector specifying which date components to extract. Can include "year", "month", and/or "day". Defaults to all three.
format	Optional format string for parsing (passed to as.Date()). If NULL (default), attempts automatic parsing via lubridate.
orders	Optional character vector of lubridate order specifications (e.g., c("dmy", "mdy", "ymd")). Used when format = NULL. Defaults to c("ymd", "dmy", "mdy").

Details

Components are returned as zero-padded strings:

- "year" – 4-digit year (e.g., "2023")
- "month" – 2-digit month (e.g., "01", "12")
- "day" – 2-digit day (e.g., "05", "31")

The order of tokens in the output follows the order of components.

Value

A list of character vectors, one per input element. Each vector contains the requested date components as strings. Unparseable dates return an empty character vector with a warning.

See Also

normalize_date() to match whole dates, approximate_date() to match on coarser periods.

Other date preparers: approximate_date(), normalize_date()

Examples

```
date_tokens("2023-12-31")
# list(c("2023", "12", "31"))

date_tokens("31.12.2023", components = c("year", "month"))
# list(c("2023", "12"))

date_tokens("12/31/2023", components = "year")
# list("2023")

date_tokens(c("2023-01-15", "15.06.2023"))
# list(c("2023", "01", "15"), c("2023", "06", "15"))
```

deduplicate_table *Deduplicate a Table*

Description

Generic function that removes or merges duplicate records from a table based on duplicate pairs identified by `detect_duplicates()`.

Usage

```
deduplicate_table(base_table, duplicates, id, ...)
```

Arguments

`base_table`	A data.frame / tibble / data.table (or db table in other backends).
`duplicates`	A table of duplicate pairs generated by detect_duplicates
`id`	Character scalar naming the ID column in `base_table`.
`...`	Additional arguments passed to backend-specific methods.

Value

A deduplicated version of `base_table`.

Examples

```
ex <- exact_strategy(
  workshop ~ normalize_text() + word_tokens(min_nchar = 3),
  block_by = c("postcode_area", "trade")
)
dups <- detect_duplicates(workshop_register, id = "reg_no", strategy = ex)
deduped <- deduplicate_table(workshop_register, dups, id = "reg_no")
c(before = nrow(workshop_register), after = nrow(deduped))
```

detect_duplicates *Detect Duplicate Records*

Description

Find likely duplicate records inside a single table and group them. Records are compared by how much of their rare, informative token content they share (not by character-level edit distance), every pair is scored, and any pair scoring at or above the threshold is linked. Records that link directly or transitively form one duplicate group.

Pass a `search_strategy()` for fuzzy, scored matching, or an `exact_strategy()` to group only records whose token sets are identical.

Usage

```
detect_duplicates(base_table, id, strategy, ...)
```

Arguments

base_table	A data.frame, tibble, data.table, or backend table to deduplicate.
id	Character scalar naming the ID column in base_table.
strategy	A Search_Strategy (or Exact_Strategy) describing how to tokenize each column, how to block, and the matching threshold.
...	Additional arguments passed to backend-specific methods. The most useful are threshold (override the strategy's threshold) and weights (a named numeric vector overriding the strategy's column weights).

Value

A table with one row per record that belongs to a duplicate group:

duplicate_group Group label shared by all records that are duplicates of one another.

id The record ID.

score The record's match score within its group.

rank Rank within the group; rank 1 is the representative kept by deduplicate_table().

<original columns> Every other column from base_table.

See Also

deduplicate_table() to collapse the groups, search_candidates() for the cross-table version, multi_stage_dedup() for staged passes.

Examples

```
data(base_example)

strat <- search_strategy(
  Nachname ~ normalize_text() + word_tokens(min_nchar = 3),
  Vorname  ~ normalize_text() + word_tokens(min_nchar = 3),
  Ort      ~ normalize_text(),
  block_by = "Kreis",
  threshold = 0.8
)

dups <- detect_duplicates(base_example, id = "id_base", strategy = strat)
head(dups)
```

`drop_joinery_temp_tables` *Drop all temporary DuckDB tables created by joinery*

Description

The DuckDB backend writes ephemeral tables during batch preprocessing (for example the token tables built by `prepare_search_data()`). A clean run drops them when it finishes, but a run that is killed partway, or a machine that loses power mid-job, can leave them behind on disk. This sweeps them up.

Usage

```
drop_joinery_temp_tables(
  con,
  prefixes = c("_joinery_tokens_", "_joinery_tmp_", "_joinery_emb_")
)
```

Arguments

`con`	A DuckDB connection.
`prefixes`	Character vector of table name prefixes that identify joinery temporary tables. Defaults cover all current ephemeral table types.

Details

Each temporary table carries a reserved name prefix such as `"_joinery_tokens_"` or `"_joinery_tmp_"`. Only tables whose names begin with one of those prefixes are removed, so your own tables are never touched. Pass extra `prefixes` to cover temporary table types added in future.

Value

A character vector of removed table names, invisibly.

Examples

```
if (requireNamespace("duckdb", quietly = TRUE) &&
    requireNamespace("DBI", quietly = TRUE)) {
  con <- DBI::dbConnect(duckdb::duckdb(), ":memory:")
  # A stray joinery temp table left behind by an interrupted run:
  DBI::dbWriteTable(con, "_joinery_tmp_demo", data.frame(x = 1))

  drop_joinery_temp_tables(con)  # removes it, returns its name invisibly
  DBI::dbDisconnect(con, shutdown = TRUE)
}
```

`drop_numeric_tokens` *Drop numeric (house-number) tokens from token lists*

Description

Symmetric inverse of `numeric_tokens()`: removes pure-digit tokens (typically house numbers) from a token column. Operates on the list-of-character token vectors produced by earlier steps such as `word_tokens()`, mirroring `filter_stopwords()`.

Useful in address pipelines where the street name carries the matching signal but the house number is noise (and fans out blocks): tokenize the street, then `drop_numeric_tokens()` to keep only the name tokens.

Usage

```
drop_numeric_tokens(tokens, keep_letters = TRUE)
```

Arguments

`tokens`	A list of character vectors.
`keep_letters`	Logical. If TRUE (default), number-letter tokens such as "12A" are retained; only pure-digit tokens like "12" are dropped. If FALSE, any token containing a digit is dropped.

Value

A list of character vectors with numeric tokens removed.

See Also

`numeric_tokens()`, its inverse; `filter_stopwords()` for the same idea with a named word list.

Other token transformers: `drop_short_tokens()`, `extract_initials()`, `filter_stopwords()`, `fuzzy_tokens()`, `token_shapes()`, `use_dictionary()`

Examples

```
drop_numeric_tokens(list(c("MAIN", "12", "ST")))
# list(c("MAIN", "ST"))

drop_numeric_tokens(list(c("MAIN", "12A")), keep_letters = FALSE)
# list("MAIN")
```

drop_short_tokens *Drop short tokens from token lists*

Description

Removes tokens shorter than `min_nchar` characters from a token column. Where `word_tokens()`'s own `min_nchar` filters length at *tokenisation*, this filters length *after* a token transform - which is where it matters for the phonetic encoders (`as_cologne()`, `as_soundex()`, `as_metaphone()`) and `generate_ngrams()`: those produce short codes, and a 1-2 character code maps to a very large equivalence class (low distinctiveness), so it behaves as a false-match magnet. Chain `drop_short_tokens()` after the encoder to keep only the discriminative codes.

Operates on the list-of-character token vectors produced by earlier steps, mirroring `filter_stopwords()` / `drop_numeric_tokens()`.

Usage

```
drop_short_tokens(tokens, min_nchar = 2)
```

Arguments

`tokens`	A list of character vectors.
`min_nchar`	Whole number; tokens with fewer than this many characters are dropped. Default 2.

Value

A list of character vectors with short tokens removed.

See Also

`filter_stopwords()` and `drop_numeric_tokens()` for the same list-column idea with other drop rules; `word_tokens()` for the same length cut applied at tokenisation instead.

Other token transformers: `drop_numeric_tokens()`, `extract_initials()`, `filter_stopwords()`, `fuzzy_tokens()`, `token_shapes()`, `use_dictionary()`

Examples

```
drop_short_tokens(list(c("BAU", "AG", "X")))
# list(c("BAU", "AG"))  # the 1-char token is dropped at the default min_nchar = 2

# keep only Cologne codes of 4+ digits (drops the collision-prone short class)
drop_short_tokens(as_cologne(list(c("Bülau", "Mertens"))), min_nchar = 4)
# list("67268")
```

duckdb_batch_plan *Create a Batch Plan for DuckDB Table Processing*

Description

Analyses a DuckDB table and generates a batch plan (data.table) that defines how to split the table into atomic processing units. Each row of the plan represents one batch with row counts, optional row-number windows, and block identifiers (if blocking is used).

Usage

```
duckdb_batch_plan(
  db_tbl,
  id,
  target_batch_size = NULL,
  min_batch_size = NULL,
  chunk_strategy = "block_consolidated",
  block_by = NULL,
  atomic_blocks = FALSE
)
```

Arguments

db_tbl	A DuckDB table reference (result of dplyr::tbl(con, "table_name"))
id	Character. Column name(s) to use as record identifier(s). Not used for batching but validated to exist in the table.
target_batch_size	Positive integer. Target number of rows per batch. Default: 1e6 (1 million rows).
min_batch_size	Positive integer. Minimum table size to trigger batching. If total rows < min_batch_size, returns single batch. Default: 1e5 (100k rows).
chunk_strategy	Character. One of "even", "block_first", or "block_consolidated". Default: "block_consolidated".
block_by	Optional character vector. Column name(s) to use for semantic blocking. If specified, batches respect block boundaries. Supports multiple columns (e.g., c("region", "year")).
atomic_blocks	Logical. When FALSE (default) the planner may sub-split a block larger than target_batch_size into row-number windows - correct for *preprocess* batching, where token generation is per-row independent. When TRUE (the *scoring* path) a block is treated as **indivisible**: small blocks are consolidated under the budget but a large block is kept whole as a single chunk (flagged oversized = TRUE), never sub-split - because a match pair only forms *within* a block, so splitting one would silently drop cross-pairs. Requires block_by; rejects chunk_strategy = "even".

Details

The function supports three chunking strategies:

- `"even"`: Simple row-number chunking, ignores blocks
- `"block_first"`: Each batch = one block (or sub-chunks if block > target_batch_size)
- `"block_consolidated"`: Consolidates small blocks to minimize batch count (default)

Small tables: If total rows < `min_batch_size`, returns a single batch regardless of strategy. With blocking, still respects blocks.

Row-number windows: For unblocked or large-block sub-chunking, `row_start` and `row_end` define window boundaries (1-based, inclusive). For block-based batches (small blocks), these are NA.

Consolidation: `"block_consolidated"` (default) combines multiple small blocks into single batches up to `target_batch_size` to reduce overhead. Each batch may contain zero, one, or multiple blocks (depending on sizes and consolidation).

Row ordering: To ensure `row_start` and `row_end` windows are consecutive and can be reliably sliced from the DB, the function sorts by the `id` column before computing row numbers. This ensures reproducible, deterministic batch boundaries.

Value

A `data.table` with columns:

- `batch_id`: integer, sequential batch identifier (1, 2, 3, ...)
- `row_count`: integer, number of rows in this batch
- `row_start`: integer (or NA), window start for row-number-based batches; NA for block-based
- `row_end`: integer (or NA), window end for row-number-based batches; NA for block-based
- Additional columns (if `block_by` specified): one per blocking variable, containing block values

Examples

```
if (requireNamespace("duckdb", quietly = TRUE) &&
    requireNamespace("DBI", quietly = TRUE) &&
    requireNamespace("dplyr", quietly = TRUE)) {
  con <- DBI::dbConnect(duckdb::duckdb(), ":memory:")
  DBI::dbWriteTable(
    con, "data",
    data.frame(id = 1:1000, region = rep(LETTERS[1:5], length.out = 1000))
  )
  tbl_ref <- dplyr::tbl(con, "data")

  # Unblocked, even row-number chunking
  plan1 <- duckdb_batch_plan(
    tbl_ref, id = "id",
    target_batch_size = 200, chunk_strategy = "even"
  )
```

```
  # Blocked, consolidated strategy (default, respects regions)
  plan2 <- duckdb_batch_plan(
    tbl_ref, id = "id",
    target_batch_size = 200, block_by = "region"
  )
  DBI::dbDisconnect(con, shutdown = TRUE)
}
```

---

duckdb_control *DuckDB Execution Control*

Description

Build a Duckdb_Control bundling the DuckDB backend's execution knobs, and pass it as `control =` to `prepare_search_data()`, `detect_duplicates()`, or `search_candidates()` on DuckDB tables. It controls **how** a match runs (memory, batching, chunking, failure isolation), never **what** matches - matching semantics stay on the `search_strategy()`.

Two execution stages, two atomicity rules:

- **Preprocess batching** (tokenization) is per-row, governed by `target_batch_size` / `min_batch_size` / `chunk_strategy`. Any row split is safe.
- **Scoring chunking** (the overlap join) is *block-atomic* - a pair only forms within a block, so a block can never be split. `chunk_by` packs *whole* blocks under `target_batch_size`; `on_error` isolates a pathological block from the rest of the run.

Chunking is a DuckDB (out-of-core) concern; the in-memory data.table backend ignores it.

Usage

```
duckdb_control(
  target_batch_size = NULL,
  min_batch_size = NULL,
  chunk_strategy = c("block_consolidated", "block_first", "even"),
  chunk_by = NULL,
  on_error = c("skip", "retry", "stop"),
  progress = NULL
)
```

Arguments

`target_batch_size`	`NULL` (auto-tune from RAM / row size) or a positive number of rows per batch / scoring chunk.
`min_batch_size`	`NULL` (auto-tune) or a positive number - the minimum table size before preprocess batching engages.

chunk_strategy	Preprocess chunking strategy: "block_consolidated" (default), "block_first", or "even".
chunk_by	Scoring chunk key. NULL (default) auto-derives a coarse key when the input is large and leaves small inputs monolithic; FALSE forces the monolithic path; a character vector names an explicit key, which must be a subset of the strategy's block_by (else cross-chunk pairs would be silently dropped).
on_error	Per-scoring-chunk failure policy: "skip" (default - record and continue), "retry" (re-run once with conservative pragmas, then skip), or "stop" (re-raise).
progress	NULL (auto), TRUE, or FALSE to force or suppress progress output.

Value

A Duckdb_Control object.

See Also

prepare_search_data(), detect_duplicates(), search_candidates().

Examples

```
# The control object just bundles execution knobs; it carries no data.
ctrl <- duckdb_control(target_batch_size = 5e5, on_error = "skip")
ctrl


# Pass it to a verb running on a DuckDB table.
if (requireNamespace("duckdb", quietly = TRUE) &&
    requireNamespace("DBI", quietly = TRUE) &&
    requireNamespace("dplyr", quietly = TRUE)) {
  con <- DBI::dbConnect(duckdb::duckdb(), ":memory:")
  DBI::dbWriteTable(con, "reg", as.data.frame(workshop_register))
  strat <- search_strategy(
    workshop ~ normalize_text() + word_tokens(min_nchar = 3),
    block_by  = c("postcode_area", "trade"),
    threshold = 0.7
  )
  detect_duplicates(dplyr::tbl(con, "reg"), "reg_no", strat, control = ctrl)
  DBI::dbDisconnect(con, shutdown = TRUE)
}
```

---

embedding_strategy *Create an Embedding Strategy*

Description

Construct an Embedding_Strategy object for semantic matching using embeddings. This is a distinct strategy type from token-based strategies created with search_strategy().

Embedding strategies:

- Represent entire records as embedding vectors
- Use cosine similarity for scoring
- Support blocking variables to restrict comparisons
- Require the tidyllm package for embedding computation

Usage

```
embedding_strategy(
  columns = NULL,
  embedding_model,
  threshold,
  collapse_sep = " ",
  normalize = TRUE,
  batch_size = 1000,
  block_by = NULL
)
```

Arguments

columns	Character vector of column names to embed, or NULL (default) to use all non-id character-like columns.
embedding_model	A tidyllm provider object (e.g., ollama(.model = "mxbai-embed-large")). This is passed directly to tidyllm's embed() function.
threshold	Numeric scalar in (0, 1). Cosine similarity threshold for filtering matches.
collapse_sep	Character scalar. Separator used when joining multiple columns into a single text string. Default is " ".
normalize	Logical scalar. If TRUE (default), apply L2 normalization to embeddings before computing cosine similarity.
batch_size	Numeric scalar. Number of records to process per batch when computing embeddings. Default is 1000.
block_by	Character vector of blocking variable names, or NULL (default). When specified, comparisons are only made within matching blocks.

Value

An Embedding_Strategy S7 object.

Examples

```
## Not run:
library(tidyllm)

# Create an embedding strategy using Ollama
emb_strat <- embedding_strategy(
  columns = c("name", "address"),
  embedding_model = ollama(.model = "mxbai-embed-large"),
  threshold = 0.85
)

# Use in multi-stage workflow
results <- multi_stage_search(
  base_table = customers_a,
  target_table = customers_b,
  base_id = "id_a",
  target_id = "id_b",
  strategies = list(
    token_stage = search_strategy(name ~ normalize_text() + word_tokens()),
    semantic_stage = emb_strat
  )
)

## End(Not run)
```

`exact_strategy` *Define an Exact Matching Strategy*

Description

Creates an Exact_Strategy for exact, score-1.0 token-set matching. Hand it to the same verbs you would hand a `search_strategy()`: `detect_duplicates()` to group identical records within a table, `search_candidates()` to match them across tables. Both return the usual result with `score == 1.0`.

Two records link only when **every column's token set is equal** within the same block. This is the same as a fuzzy score of exactly 1.0, reached without any scoring or threshold, and it is **robust to empty columns**: two records with identical names and both streets blank will link, where a weighted threshold would silently reject them. (A blank column drags a weighted score below its threshold, since its weight stays in the denominator; exact matching has no such ceiling.)

Use it as the cheap first stage of a staged workflow (exact first, then fuzzy on whatever is left): the leftover records come from `extract_unmatched()`, and `multi_stage_dedup()` / `multi_stage_search()` thread them through for you when you pass `list(exact_strategy(...), search_strategy(...))`.

Usage

```
exact_strategy(
  ...,
```

```
  block_by = NULL,
  rarity = "inverse_freq",
  containment = c("off", "forward", "bidirectional"),
  min_base_rarity = 0,
  min_containment_tokens = 1
)
```

Arguments

`...`	Two-sided formulas `column ~ step1 + step2`, identical in form to `search_strategy()`.
`block_by`	Optional character vector of blocking columns.
`rarity`	Character scalar rarity metric, used only by the `min_base_rarity` containment guard to measure how much identifying weight a base record's tokens carry. One of `"inverse_freq"` (default), `"smoothed_inverse_freq"`, `"tfidf"`, or `"bm25"`; see `search_strategy()` for what each formula does. Plain equality and forward containment without a `min_base_rarity` floor never consult it, so the default is almost always fine.
`containment`	One of `"off"` (set-equality, default), `"forward"` (link when the base record's tokens are a subset of the target's), or `"bidirectional"` (either direction). Whether it helps depends on the data; it over-links on noisy corpora, so it is never the default.
`min_base_rarity`	Numeric containment guard: drop links whose base record carries summed rarity mass below this floor. Default `0`.
`min_containment_tokens`	Numeric containment guard, default 1 (no restriction). A *proper* containment link (one record's tokens a strict subset of the other's) requires the contained record to hold at least this many tokens; set-equality always links. Raise to 2 to stop a single generic token - a bare category or hub name (e.g. a shopping-centre name) - being a subset of every richer "Store + Centre" name and transitively chaining unrelated records into one entity. Ignored when `containment = "off"`.

Value

An Exact_Strategy object.

See Also

`search_strategy()`, `detect_duplicates()`, `search_candidates()`, `extract_unmatched()`.

Examples

```
# Link only workshops whose name tokens are identical within the same area
# and trade. No threshold to tune, and blank columns do not sink a match.
ex <- exact_strategy(
  workshop ~ normalize_text() + word_tokens(min_nchar = 3),
  block_by = c("postcode_area", "trade")
)
dups <- detect_duplicates(workshop_register, id = "reg_no", strategy = ex)
```

```
head(dups)
```

`explain_match` *Explain a Single Match*

Description

Attribution diagnostic (Q3). Reconstructs per-column and per-token contributions to a single match score. Dispatches on the second positional argument: a `Search_Strategy` triggers reconstruction from raw inputs; a tokens-shaped table is used directly.

Usage

```
explain_match(matches, x, ...)
```

Arguments

`matches`	Match output table.
`x`	Either a `Search_Strategy` (ergonomic form) or a tokens table with `rarity` (power-user form).
`...`	Backend-specific arguments. For the ergonomic form: `base`, `target`, `match_id`.

Value

A `Match_Explanation` object.

Examples

```
strat <- search_strategy(
  workshop ~ normalize_text() + word_tokens(min_nchar = 3),
  block_by  = c("postcode_area", "trade"),
  threshold = 0.7
)
matches <- search_candidates(
  workshop_listings, workshop_register,
  base_id = "listing_id", target_id = "reg_no", strategy = strat
)
# Break one pair's score down into its per-token contributions.
first_id <- matches$match_id[matches$source == "target"][1]
explain_match(matches, strat,
              base = workshop_listings, id = "listing_id",
              target = workshop_register, target_id = "reg_no",
              match_id = first_id)
```

export_for_labelling *Export a match sample to CSV for manual labelling*

Description

Write a sampled set of matches to a CSV pre-filled with an `equal` column on block-header rows. Users edit the CSV in any spreadsheet, marking only exceptions (e.g. false positives) and leaving the rest as defaults.

Block definition follows the matches schema: for candidate matches (from `search_candidates()`), the header is the base-side row and candidate rows inherit its default. For duplicate matches (from `detect_duplicates()`), the header is the rank-1 row and the remaining records in the duplicate group inherit its default.

Usage

```
export_for_labelling(sample, file, default_label = 1L)
```

Arguments

`sample`	A `Match_Sample` object or a `data.table` / `data.frame` with the matches schema.
`file`	Path to the CSV file to write.
`default_label`	Integer scalar (default `1L`) used as the block-default `equal` value on header rows. `0L` for the inverse workflow.

Value

Invisibly returns `file`.

See Also

`import_labels()`

extract_initials *Extract initials from tokens*

Description

Keeps only the first character of each token (`"ANNA"` becomes `"A"`). Use it to match on initials when full first names are recorded inconsistently, for example when one source has `"Anna Berta Schmidt"` and another `"A. B. Schmidt"`.

Usage

```
extract_initials(tokens)
```

Arguments

tokens	A list of character vectors.

Details

It transforms a token column, so it runs after a token generator such as word_tokens().

Value

A list of character vectors of single-character initials.

See Also

Other token transformers: drop_numeric_tokens(), drop_short_tokens(), filter_stopwords(), fuzzy_tokens(), token_shapes(), use_dictionary()

Examples

```
extract_initials(list(c("Anna", "Berta")))
# list(c("A", "B"))
```

---

extract_unmatched *Extract Unmatched Records*

Description

Identify and extract records from a table that were not matched in a record linkage operation.

Usage

```
extract_unmatched(data, id, matches, ...)
```

Arguments

data	A data.frame / tibble / data.table (or db table in other backends) containing the original records.
id	Character scalar naming the ID column in data.
matches	A table of matched record pairs, containing the ID column.
...	Additional arguments passed to backend-specific methods.

Value

A subset of data containing only records whose IDs do not appear in matches.

Examples

```
strat <- search_strategy(
  workshop ~ normalize_text() + word_tokens(min_nchar = 3),
  block_by  = c("postcode_area", "trade"),
  threshold = 0.7
)
matches <- search_candidates(
  workshop_listings, workshop_register,
  base_id = "listing_id", target_id = "reg_no", strategy = strat
)
# The listings that found no register match, ready for a looser next pass.
leftover <- extract_unmatched(workshop_listings, "listing_id", matches)
nrow(leftover)
```

---

filter_stopwords *Filter out stopwords from token lists*

Description

Some tokens carry no matching signal but appear everywhere: legal forms like `GMBH` or `LTD`, articles, generic words. Because they are common they create many spurious matches and fan out blocks. `filter_stopwords()` removes named tokens so matching rests on the distinctive ones. The comparison is case-insensitive.

Usage

```
filter_stopwords(tokens, stopwords)
```

Arguments

`tokens`	A list of character vectors, as produced by `word_tokens()`.
`stopwords`	A character vector of tokens to remove (case-insensitive).

Details

It transforms a token column, so it runs after a token generator such as `word_tokens()`.

Value

A list of character vectors with the stopwords removed.

See Also

`drop_numeric_tokens()` to remove house numbers the same way.

Other token transformers: `drop_numeric_tokens()`, `drop_short_tokens()`, `extract_initials()`, `fuzzy_tokens()`, `token_shapes()`, `use_dictionary()`

Examples

```
filter_stopwords(list(c("MUELLER", "GMBH")), stopwords = c("gmbh"))
# list("MUELLER")
```

`find_stopwords` *Discover candidate stopwords from a prepared token table*

Description

Scores every `(src_column, token)` by its document frequency - the share of records in that column whose value contains the token - and returns the tokens common enough to be poor discriminators. These are stopword candidates: feed them to `filter_stopwords()` in the preparer chain and re-run `prepare_search_data()`.

Usage

```
find_stopwords(
  tokens,
  max_prop = 0.3,
  top_n = NULL,
  by_block = FALSE,
  block_by = NULL
)
```

Arguments

`tokens`	A token table produced by `prepare_search_data()` (data.table or DuckDB backend). Must contain `src_column`, `token`, and `row_id`.
`max_prop`	Numeric in `(0, 1]`. Return tokens whose document-frequency share is at least this value. Default `0.3` (token appears in 30% or more of a column's records).
`top_n`	Optional integer. If supplied, instead of (or in addition to) the `max_prop` cut, keep at most the `top_n` most frequent tokens per column. When both are given, the union is returned.
`by_block`	Logical. Compute the share within each block rather than corpus-wide. Requires `block_by` to name the block columns (the token table also carries the id column, so they cannot be inferred safely). Default `FALSE`.
`block_by`	Character vector of block columns. Required when `by_block = TRUE`; pass the strategy's `block_by`. Ignored otherwise.

Details

Document frequency is computed corpus-wide by default (`by_block = FALSE`), i.e. across all blocks. This matches the intuition of a stopword as a globally common term. With `by_block = TRUE` the share is computed within each block and a token is returned if it crosses `max_prop` in *any* block, reported at its maximum block-level share - useful when a token is rare overall but saturates a single dense block.

Value

A data.table with one row per flagged (src_column, token): src_column, token, df (distinct records containing the token), n_records (records in the column / block), and prop = df / n_records. Sorted by src_column then descending prop. Empty (zero-row) when nothing crosses the threshold.

See Also

filter_stopwords() to apply the result in a preparer chain.

---

find_subwords *Learn a subword vocabulary from your text*

Description

Word tokens fail quietly on compound words and misspellings: "Maschinenbaugesellschaft" and "Maschinenbau GmbH" share no word, so they never meet in a word-token match. Subword tokens fix this by learning, from your own data, a vocabulary of frequent word pieces and splitting every record into those pieces. Two spellings of the same name then share most of their pieces even when they share no whole word.

find_subwords() is the learning step. Run it once on the text of a column (after the same cleaning you use in your strategy), then place the fitted model in the strategy formula with subword_tokens():

```
sw <- find_subwords(normalize_text(register$name))
strat <- search_strategy(
  name ~ normalize_text() + subword_tokens(sw)
)
```

The fitted model is self-contained: you can save it with saveRDS() and reuse it in a later session, and the same model always splits text the same way.

Usage

```
find_subwords(
  x,
  vocab_size = 500L,
  algorithm = c("bpe", "unigram"),
  keep_boundary = FALSE,
  sample_n = NULL,
  columns = NULL
)
```

Arguments

x	The training text: a character vector, or a table (data frame, tibble, data.table, DuckDB table) together with columns.
vocab_size	Number of subword pieces to learn. Default 500.
algorithm	"bpe" (default) or "unigram", the two learning algorithms SentencePiece offers. Start with "bpe"; try "unigram" if its splits look better on your data.
keep_boundary	Keep the word-boundary marker on each piece? The default FALSE strips it, so the same piece matches whether it starts a word or not; that is usually what record linkage wants.
sample_n	Optional cap on the number of rows used for training. On a corpus of millions of rows a uniform sample of a few hundred thousand fits an equivalent vocabulary in a fraction of the time.
columns	For table input: character vector naming the text column(s) to train on.

Details

Fit the model on the text as the strategy will see it: apply the same cleaning steps (for example normalize_text()) to the training text that precede subword_tokens() in the formula. Training on raw text and applying to cleaned text (or the other way round) degrades the splits.

vocab_size sets how fine the splits are. Small vocabularies split aggressively into short pieces; large ones keep frequent words whole and only split rare ones. The default of 500 suits mid-sized name corpora; large corpora often support 1000 to 8000. On a very small corpus the fitted vocabulary can come out smaller than requested; the model records the size actually fitted. rarity_distribution() on the prepared tokens is the read for judging the result.

Very frequent single-character pieces are normal and mostly harmless: they carry almost no rarity, so they contribute little to a score. Compose subword_tokens(sw) + drop_short_tokens(2) to drop them outright.

Value

A fitted Subword_Model, ready for subword_tokens().

See Also

subword_tokens() to use the model in a strategy; find_stopwords() for the same fit-then-apply pattern on stopwords.

Examples

```
if (requireNamespace("sentencepiece", quietly = TRUE)) {
  firms <- rep(c(
    "maschinenbau mueller", "tischlerei schmidt",
    "holzbau wagner", "metallbau krause"
  ), 40)
  sw <- find_subwords(firms, vocab_size = 60)
  sw
  subword_tokens("maschinenbaugesellschaft mueller", sw)
```

```
}
```

`fit_filter` *Fit a false-positive filter on labelled match pairs*

Description

Fit a baseline classifier to predict whether each scored pair is a true match (`equal == 1L`) or a false positive (`equal == 0L`). The baseline path uses `stats::glm` with the logit link and no external dependencies. The features object is the input from `match_features()`; labels carry the `equal` column produced by `import_labels()`.

Usage

```
fit_filter(
  features,
  labels,
  model = "logistic",
  class_weighted = FALSE,
  na_fill = 0,
  ...
)
```

Arguments

`features`	A `Match_Features` object.
`labels`	A `data.table` / `data.frame` with the matches schema plus an integer `equal` column (`0L` / `1L`). Typically produced by `import_labels()`.
`model`	Character scalar (default `"logistic"`) selecting the baseline `glm()` path. Future M6 work will accept a fitted parsnip / workflow object here.
`class_weighted`	Logical scalar. When `TRUE`, fit `glm` with inverse-class-frequency `weights =`, useful for imbalanced training sets. Default `FALSE`.
`na_fill`	Numeric scalar used to impute predictor NAs. Default `0` (sensible for aIP slot columns where NA means "no token").
`...`	Reserved for future expansion.

Value

A `Filter_Model` object.

Examples

```
strat <- search_strategy(
  workshop   ~ normalize_text() + word_tokens(min_nchar = 3),
  proprietor ~ normalize_text() + word_tokens(min_nchar = 2),
  block_by  = c("postcode_area", "trade"),
  threshold = 0.30
)
matches <- search_candidates(
  workshop_listings, workshop_register,
  base_id = "listing_id", target_id = "reg_no", strategy = strat
)
feats <- match_features(matches, strat,
                        base = workshop_listings, id = "listing_id",
                        target = workshop_register, target_id = "reg_no")
# match_labels_example carries the same pairs with a hand-checked `equal` flag.
model <- fit_filter(feats, match_labels_example)
model
```

---

frontier_plot *Cost/recall frontier scatter for a strategy plan*

Description

Plots each candidate block key at (candidate, exact_twin_survival) - the recall axis - with the brute-pair cost in the point labels. The knee is the cheapest candidate whose twin survival stays high.

Usage

```
frontier_plot(x, ...)
```

Arguments

x	A `Strategy_Plan` object from `plan_strategy()`.
...	Passed to `tinyplot::tinyplot()`.

Value

Invisibly, the plotted `data.table` (frontier).

`fuzzy_tokens` *Collapse near-duplicate tokens to a canonical form*

Description

Typos and minor spelling differences split one real token into many (`"Neumann"`, `"Neumann"` with a slip, `"Neuman"`). `fuzzy_tokens()` finds tokens within a string distance of each other, groups them, and rewrites every member of a group to one canonical spelling, so the variants match. Unlike `use_dictionary()`, which needs a known synonym list, this discovers the groups from the data.

Usage

```
fuzzy_tokens(x, max_dist = 2, method = "osa", min_nchar = 1)
```

Arguments

`x`	A character vector to tokenize and canonicalize.
`max_dist`	Maximum string distance for two tokens to be treated as the same. For `method = "jw"` this is a Jaro-Winkler distance in `[0, 1]` (smaller is stricter); for edit-distance methods it is a count of edits.
`method`	A `stringdist::stringdist()` method, e.g. `"osa"` (default), `"lv"`, or `"jw"`.
`min_nchar`	Minimum token length to consider; shorter tokens are dropped before grouping.

Details

Use it when a field has organic spelling noise and you do not have a dictionary. The canonical form per group is the longest token, breaking ties by the most central token, then alphabetically.

When not to use it:

- **High-cardinality columns.** It compares every distinct token against every other in one dense distance matrix, so cost and memory grow with the square of the number of distinct tokens. On a large vocabulary (tens of thousands of distinct tokens and up) it is slow and memory-hungry. Normalize aggressively first, and prefer `use_dictionary()` when the groups are already known.
- **When over-merging is costly.** Grouping is by connected components, so matches chain transitively: if `A` is close to `B` and `B` to `C`, all three collapse even when `A` and `C` are far apart. A loose `max_dist` or short tokens can fuse genuinely distinct values. Keep `max_dist` tight, raise `min_nchar` to drop noise-prone short tokens, and check the groups on a sample before trusting them.

Value

A list of character vectors, one per input element, with each token replaced by its group's canonical form.

See Also

use_dictionary() when the groups are known in advance.

Other token transformers: drop_numeric_tokens(), drop_short_tokens(), extract_initials(), filter_stopwords(), token_shapes(), use_dictionary()

Examples

```
fuzzy_tokens(c("Neumann", "Neumaxn", "Neuman"), max_dist = 2)
# every row's token becomes "NEUMANN"
```

---

generate_ngrams *Generate character n-grams from text*

Description

An n-gram is a sliding window of n consecutive characters. Matching on character n-grams instead of whole words tolerates typos, truncations, and joined-up spellings, because two strings that differ by a letter still share most of their windows ("meier" and "maier" share "ei", "er", and so on). Reach for it on short, noisy fields where word tokens are too brittle.

Usage

```
generate_ngrams(text, n)
```

Arguments

text	A character vector to break into n-grams.
n	The window length (number of characters per n-gram).

Details

It tokenizes text directly, so it replaces word_tokens() rather than following it. The trade-off is fan-out: every string yields many overlapping tokens, so n-grams cost more to match than words. Larger n is sharper and cheaper, smaller n is fuzzier and denser.

Value

A list of character vectors, one per input element. Strings shorter than n yield an empty vector.

See Also

word_tokens() for whole-word tokens.

Other token generators: custom_tokens(), numeric_tokens(), subword_tokens(), word_tokens()

Examples

```
generate_ngrams("hello", 2)
generate_ngrams("an example", 3)
```

---

import_labels *Import a labelled CSV back into a feature/label table*

Description

Read a CSV written by export_for_labelling() (optionally edited by a user), propagate the block-default equal value from each header row onto unmarked rows in that block, validate the schema, and return a data.table ready for fit_filter() / calibrate_matches().

Usage

```
import_labels(file)
```

Arguments

file	Path to the CSV file to read.

Value

A data.table with the same rows as the original sample plus a fully populated equal column (0L / 1L).

See Also

export_for_labelling()

---

inspect_tokens *Inspect Tokens for a Specific Column*

Description

Extract and examine the tokens generated for a specific column after applying the preprocessing steps defined in a Search_Strategy. Useful for debugging and understanding how text is tokenized.

Usage

```
inspect_tokens(data, id, strategy, column)
```

Arguments

data	A data.frame / tibble / data.table (or db table in other backends).
id	Character scalar naming the ID column in data.
strategy	A Search_Strategy object defining preprocessing steps.
column	<data-masked> The column to inspect.

Value

A backend-specific table showing the tokens generated for the specified column.

---

joinery_recipe *Build a tidymodels recipe for calibration features*

Description

Construct a pre-configured recipes::recipe() suitable for fitting a false-positive filter on the output of match_features(). Tags ID columns (searched, found, match_id) with role "id", sets equal as the outcome, and keeps every other numeric column as a predictor. Requires the suggested recipes package.

Usage

```
joinery_recipe(features, labels, ...)
```

Arguments

features	A Match_Features object.
labels	A labels data.table with equal (as for fit_filter()).
...	Reserved for future expansion.

Value

A recipes::recipe() object.

match_features *Build a per-pair feature table for calibration*

Description

Computes a wide, one-row-per-pair feature data.table from a joinery match result, suitable for downstream calibration / false-positive filtering. The schema is documented in notes/calibration_design.md and treated as the public API. Additions are allowed; reorders or renames are not.

Dispatches on (matches, strategy). A Search_Strategy returns the full token schema (core + token-side columns + string similarity). An Embedding_Strategy returns the reduced "embedding" schema (core columns + string similarity + cosine_sim + embedding norms).

Usage

```
match_features(matches, strategy, ...)
```

Arguments

matches	A match result table (data.table / tibble / data.frame / DuckDB lazy tbl) from detect_duplicates() or search_candidates().
strategy	The Search_Strategy or Embedding_Strategy used to produce matches.
...	Method-specific arguments. Both strategy methods accept: base (the base table used as input to matching), id (character scalar naming the ID column in base), target (optional target table for cross-table candidate matches), target_id (ID column in target, defaults to id), include_string_sim (logical; when TRUE (default) emits sim_sf_<col> / sim_fs_<col> per column via stringdist::stringsim() - requires the stringdist suggested package), method (stringdist method applied to every column, default "jw". Only a scalar is honoured today; the argument shape also reserves a named character vector for per-column methods, the additive path to the per-column comparators a future probabilistic strategy will use), and include_block_stats (logical; whether to compute cnt / icnt / ipos). The Search_Strategy method additionally accepts top_n (named integer / list controlling per-column top-N counts for the m_/f_/s_ columns; use a default entry as fallback; set a column to 0 to suppress its set). The Embedding_Strategy method emits cosine_sim (pass-through of score) and embedding_norm_s / embedding_norm_f (L2 norms of the **pre-normalization** embeddings, recomputed only over the matched record subset).

Value

A Match_Features object wrapping a wide feature data.table.

Examples

```
strat <- search_strategy(
  workshop   ~ normalize_text() + word_tokens(min_nchar = 3),
  proprietor ~ normalize_text() + word_tokens(min_nchar = 2),
```

```
  block_by  = c("postcode_area", "trade"),
  threshold = 0.30
)
matches <- search_candidates(
  workshop_listings, workshop_register,
  base_id = "listing_id", target_id = "reg_no", strategy = strat
)
# One row per pair, with the features a filter can learn from.
feats <- match_features(matches, strat,
                        base = workshop_listings, id = "listing_id",
                        target = workshop_register, target_id = "reg_no")
feats
```

`match_labels_example` *Labelled candidate pairs for calibration examples*

Description

A frozen set of candidate match pairs with a ground-truth `equal` label, for the calibration workflow (`fit_filter`, `apply_filter`, `calibrate`). It is a saved `search_candidates()` run linking `workshop_listings` to `workshop_register`, with `equal` filled from each listing's true `actual_link`. The false positives are concentrated in the deliberately hard tiers, common-surname homonyms and planted register duplicates, so a learned filter has real mistakes to catch. The schema matches what `import_labels` returns, so it feeds the calibration verbs with no manual labelling step.

Usage

```
match_labels_example
```

Format

A data frame with 1,862 rows (two rows per pair) and 12 variables:

match_id Integer pair id. Each id groups one `base` row and one `target` row.

score The candidate score from the frozen search

source `"base"` (the searched listing) or `"target"` (the register candidate). The candidate row carries the label.

id The record id: a `listing_id` on base rows, a `reg_no` on target rows

workshop Business name of the row

proprietor Proprietor name of the row

trade Trade

postcode_area UK outward-code area

gen_tier The generation tier of the row, useful for slicing the hard cases

actual_link The searched listing's true `reg_no` (present on base rows)

rank Candidate rank within the pair

equal Evaluation label. `1` when the target candidate is the listing's true match, `0` otherwise. On base header rows it is the block default.

Source

Synthetically generated by `data-raw/generate_match_labels.R` from the shipped `workshop_listings` / `workshop_register` pair.

See Also

`fit_filter`, `import_labels`, `workshop_listings`

`materialize_records` *Materialize Records by ID*

Description

Rehydrate a set of record IDs back into their **full records**. The positive (semi-join) complement of `extract_unmatched()`: where `extract_unmatched()` *produces* a residual set of IDs, `materialize_records()` pulls those IDs back into complete, scorable rows for the next stage.

Usage

```
materialize_records(data, id, ids, ...)
```

Arguments

`data`	A data.frame / tibble / data.table (or db table in other backends) - the corpus to pull records from.
`id`	Character scalar naming the ID column in `data`.
`ids`	Either an atomic vector of ID values, or a table carrying them (read from an `id` column, else a column named `id`'s value).
`...`	Additional arguments passed to backend-specific methods.

Details

`ids` is **polymorphic**. It may be either

- an atomic vector of ID values, or
- a table (data.frame / data.table / backend tbl) carrying the IDs. The lookup order for the ID column is: a column literally named `id` first (the `extract_unmatched()` / `resolve_entities()` output convention), otherwise a column named the same as `id`.

The return is a **semi-join**: IDs absent from `data` are silently dropped (there is nothing to rehydrate), never NULL-filled. IDs are coerced to a common type on both sides, so a BIGINT-corpus / character-id request still matches. Row order is not guaranteed; the caller sorts if needed.

On the DuckDB backend the IDs are **always** registered as a temp table and joined - never inlined as an `id IN (<literal list>)`, which binds in roughly O(n^2) and pins cores for minutes on large residual sets.

Value

The rows of `data` whose ID is in `ids`, all columns intact, one row per matching record, in no guaranteed order.

See Also

`extract_unmatched()`, the negative complement that produces the residual IDs this verb rehydrates.

`multi_stage_dedup` *Staged Duplicate Detection (within one table)*

Description

Deduplicate a single table in increasingly tolerant passes. A typical run starts with a cheap `exact_strategy()` pass that catches the clean duplicates, then applies looser `search_strategy()` passes (often with wider blocking) to the records still unmatched. All the links found across the passes are grouped into duplicate groups at the end, so a record linked to `B` in an early pass and `B` linked to `C` in a later one all land in the same group.

For linking across two tables or several sources, use `multi_stage_search()`.

Usage

```
multi_stage_dedup(table, id, strategies, ...)
```

Arguments

`table`	A data.frame, tibble, data.table, or backend table to deduplicate.
`id`	Character scalar naming the ID column in `table`.
`strategies`	Named, ordered list of strategies to apply in turn. Each element is an `exact_strategy()`, `search_strategy()`, or `embedding_strategy()`.
`...`	Further arguments to the staged run: • `rep_by`: optional character scalar naming a priority column on `table` used to choose each group's representative (passed to `resolve_entities()`: smallest `rep_by` wins, ties broken by smallest id). • `edge_filter`: optional callback `function(edges, stage_name)` applied to each pass's links before they are accumulated (for example a domain rule that drops implausible matches). The links carry `from`, `to`, `score`, and `stage`. Backend methods may accept additional arguments.

Value

The standard dedup result: `duplicate_group | id | score | rank` plus the original columns of `table`, and a `stage` column recording which pass first linked each record.

See Also

multi_stage_search() for the cross-table version, detect_duplicates() for a single pass, resolve_entities() for the grouping step.

Examples

```
# Two passes over one table: exact token-set first, then a looser fuzzy pass
# on whatever the exact pass left unmatched.
exact <- exact_strategy(
  workshop ~ normalize_text() + word_tokens(min_nchar = 3),
  block_by = c("postcode_area", "trade")
)
fuzzy <- search_strategy(
  workshop ~ normalize_text() + word_tokens(min_nchar = 3),
  block_by  = c("postcode_area", "trade"),
  threshold = 0.6
)
dups <- multi_stage_dedup(workshop_register, "reg_no",
                          list(exact = exact, fuzzy = fuzzy))
head(dups)
```

---

multi_stage_search *Staged Search Across Tables or Sources*

Description

Link the same real-world entity across two tables, or across several datasets or vintages of one dataset, by running an ordered list of strategies as successive search passes. Each pass adds the links it finds to a running record of every match (the ledger), and at the end all the links are grouped into entities, one row per record showing which entity it belongs to.

A typical run starts with a cheap exact_strategy() pass to catch the clean matches, then applies one or more looser search_strategy() passes to the records still unmatched. Use this when the two sides are not interchangeable: for example one record may carry only part of another's information, so it matters which side is searched against which. For finding duplicates within a single table, use multi_stage_dedup() instead.

Usage

```
multi_stage_search(
  base_table,
  target_table,
  base_id,
  target_id,
  strategies,
  ...
)
```

Arguments

base_table	The left table in the linkage.
target_table	The right table. Pass base_table again with self = TRUE to search a single pooled table against itself.
base_id	Character scalar naming the ID column in base_table.
target_id	Character scalar naming the ID column in target_table.
strategies	Named, ordered list of strategies to apply in turn. Each element is an exact_strategy(), search_strategy(), or embedding_strategy().
...	Further arguments controlling the staged run: • self: logical; TRUE searches base_table against itself (for example, pooling several years into one table and linking across them). • source_by: optional character vector naming the column(s) that record where each row came from (for example "year" or "register"). When set, every link is tagged as within-source or cross-source, and the result reports each entity's source and covered_sources. • collapse: what happens between stages. "none" only carries the still-unmatched records forward, while "rep" also collapses each group found so far to a single representative, shrinking the search space for the looser passes that follow. (A third mode that merges the token sets of a whole group is reserved for a future release and not yet available.) • rep_rule: rule for choosing each group's representative. Currently "canonical" is the only rule wired; to set the representative yourself, pass a priority column with rep_by. Other rules are reserved for a future release. • rebind: how the next stage's two sides are formed from the representatives and the residual: "explicit", "self", or "accumulate" (the path for incremental panel updates). • direction: which way each pass searches: "forward", "backward", or "bidirectional". • edge_filter: optional callback function(edges, stage_name) applied to each pass's links before they are accumulated (for example a domain rule that drops implausible matches). • rep_by: optional priority column for choosing representatives (passed to resolve_entities()). Backend methods may accept additional arguments.

Value

One row per pooled record describing its entity: entity | id | rep | rank | score | source | covered_sources | n_in_entity | stage. The full list of links found, with the stage and direction of each, is attached as the ledger attribute and read with attr(result, "ledger").

See Also

multi_stage_dedup() for the within-one-table version, resolve_entities() for the grouping step, exact_strategy() for the usual front stage.

Examples

```
# Follow each workshop across years: pool the panel, search it against itself,
# exact first then fuzzy, collapsing each group found so later passes see less.
exact <- exact_strategy(
  workshop ~ normalize_text() + word_tokens(min_nchar = 3),
  block_by = c("postcode_area", "trade")
)
fuzzy <- search_strategy(
  workshop ~ normalize_text() + word_tokens(min_nchar = 3),
  block_by  = c("postcode_area", "trade"),
  threshold = 0.55
)
g <- multi_stage_search(
  workshop_panel, workshop_panel,
  base_id = "record_id", target_id = "record_id",
  list(exact = exact, fuzzy = fuzzy),
  self = TRUE, source_by = "year", collapse = "rep"
)
head(g)
```

---

norm_plot *Bar chart of embedding norm quantiles*

Description

Plots p05/p25/p50/p75/p95 of the embedding vector norms. A norm of 1 is annotated; for an L2-normalised strategy all bars should sit on it.

Usage

```
norm_plot(x, ...)
```

Arguments

x	An `Embedding_Audit` object from `audit_strategy()`.
...	Passed to `tinyplot::tinyplot()`.

Value

Invisibly, the plotted `data.table` (quantile, norm).

normalize_date *Normalize dates to ISO 8601 format (YYYY-MM-DD)*

Description

The same day is written "31.12.2023", "12/31/2023", or "2023-12-31" depending on who typed it. normalize_date() parses these mixed formats and rewrites them to one ISO 8601 string (YYYY-MM-DD), so a date column matches on the day it names rather than on how it was formatted. It recognizes European (DD.MM.YYYY), American (MM/DD/YYYY), and ISO-style inputs.

Usage

```
normalize_date(x, format = NULL, orders = c("ymd", "dmy", "mdy"))
```

Arguments

x	A character or Date vector containing dates to normalize.
format	Optional format string for parsing (passed to as.Date()). If NULL (default), attempts automatic parsing via multiple common formats.
orders	Optional character vector of lubridate order specifications (e.g., c("dmy", "mdy", "ymd")). Used when format = NULL. Defaults to c("ymd", "dmy", "mdy").

Details

Returns text. For matching on individual date parts (year only, year and month) use date_tokens(); to deliberately blur near-dates together use approximate_date().

When format is provided, uses as.Date(x, format) directly. When format = NULL, tries lubridate::parse_date_time() with the specified orders to handle mixed formats flexibly.

Value

A character vector of dates in ISO 8601 format (YYYY-MM-DD). Unparseable dates return NA_character_ with a warning.

See Also

Other date preparers: approximate_date(), date_tokens()

Examples

```
normalize_date("31.12.2023")
# "2023-12-31"

normalize_date("12/31/2023")
# "2023-12-31"

normalize_date(c("2023-01-15", "15.01.2023", "01/15/2023"))
```

```
# c("2023-01-15", "2023-01-15", "2023-01-15")

normalize_date("31-12-2023", format = "%d-%m-%Y")
# "2023-12-31"
```

`normalize_street` *Normalize street names across languages*

Description

Street names are written many ways for the same place: `"Hauptstr."`, `"Hauptstrasse"`, `"Haupt Strasse"`. `normalize_street()` collapses those variants to one canonical spelling so an address column matches on the street name rather than on its abbreviation. It normalizes Unicode, folds to ASCII, upper-cases, and cleans whitespace, then rewrites known street-type tokens from a multilingual dictionary.

Usage

```
normalize_street(
  x,
  lang = NULL,
  drop_house_numbers = FALSE,
  drop_stopwords = FALSE,
  dict = joinery::street_types,
  stopwords = joinery::street_stopwords
)
```

Arguments

`x`	A character vector containing street names or address fragments.
`lang`	Optional language code (e.g., `"de"`, `"en"`, `"fr"`). When provided, the dictionary is filtered to that language and safe language-specific suffix matching is enabled. It also restricts `drop_stopwords` to that language's particle list.
`drop_house_numbers`	Logical (default `FALSE`). When `TRUE`, drops any token beginning with a digit (house numbers like `"12"`, `"12A"`, `"123B"`), keeping only the street name. Applied after street-type replacement.
`drop_stopwords`	Logical (default `FALSE`). When `TRUE`, removes locative particles and articles (e.g. German `AN DER`, French `DE LA`) listed in `stopwords`, collapsing `"An der Alster"` to `"ALSTER"`. When `lang` is given, only that language's particles are removed; otherwise the whole `stopwords` set is used.
`dict`	A dictionary of street-type definitions, typically street_types, containing the columns: • `canonical`: canonical uppercase form • `variant`: lowercased normalized variant form

- type: "exact" or "suffix"
- lang: ISO language code

stopwords — A street-stopword table, typically street_stopwords, with columns stopword (uppercase ASCII) and lang. Only consulted when drop_stopwords = TRUE.

Details

Returns text, so it sits where normalize_text() would in a pipeline, ahead of a token generator: street ~ normalize_street(lang = "de") + word_tokens().

Exact matches (e.g., "st", "rd.", "via") are always replaced. Suffix matches (e.g., German "strasse" endings or Dutch "straat") are applied **only when** lang **is explicitly specified**, which prevents unsafe substitutions such as rewriting the ending of "LINCOLN LANE".

Normalization steps include:

- Unicode -> Latin transliteration and ASCII folding (stri_trans_general)
- Conversion to uppercase
- Removal of non-alphanumeric characters
- Tokenization on spaces and per-token replacement

Exact variants are replaced verbatim with their canonical form. Suffix variants are replaced only when:

- lang is specified, and
- the token ends with a known variant suffix for that language.

Value

A character vector of normalized street names. NA inputs are preserved as NA. Rows reduced to nothing (e.g. a bare house number with drop_house_numbers = TRUE) become "".

See Also

Other text normalizers: normalize_text(), strip_vowels()

Examples

```
normalize_street("Muellerstrasse", lang = "de")
# "MUELLERSTRASSE"

normalize_street("123 Main St.")
# "123 MAIN STREET"

normalize_street("Calle Mayor 3", lang = "es")
# "CALLE MAYOR 3"

normalize_street("Hauptstr. 123A", lang = "de", drop_house_numbers = TRUE)
# "HAUPTSTRASSE"

normalize_street("An der Alster 5", lang = "de",
```

```
                  drop_house_numbers = TRUE, drop_stopwords = TRUE)
# "ALSTER"
```

`normalize_text` *Normalize text for matching*

Description

The usual first step in a preparer pipeline. Folds text to upper case, transliterates accented and non-Latin characters to ASCII, drops anything that is not a letter, digit, or space, and collapses runs of whitespace. The point is to make superficial differences in case, accents, and punctuation disappear so that `"Cafe-Conac"` and `"cafe conac"` reduce to the same text before it is split into tokens.

Usage

```
normalize_text(text, transliteration = "De-ASCII")
```

Arguments

`text`	A character string or vector to normalize.
`transliteration`	A transliteration scheme passed to `stringi::stri_trans_general()`, defaulting to `"De-ASCII"` (German-aware folding, which expands umlauts to digraphs such as ue and oe). Use `"Latin-ASCII"` for plain accent stripping, which drops the diacritic instead of expanding it.

Details

Returns text, so it goes ahead of a token generator such as `word_tokens()` in a strategy: `name ~ normalize_text() + word_tokens()`.

Value

A character vector the same length as `text`: upper-cased, ASCII, alphanumeric-and-space only, with surrounding and repeated spaces removed.

See Also

`word_tokens()`, the token generator that usually follows.

Other text normalizers: `normalize_street()`, `strip_vowels()`

Examples

```
normalize_text("Cafe Conac")
normalize_text("Strasse", transliteration = "Latin-ASCII")
```

numeric_tokens *Tokenize numeric fields, expanding ranges into individual numbers*

Description

Turns numeric/house-number-like text into a list of tokens. Expands ranges such as "12-14" or "7-9" into c("12","13","14"). Uses original spacing/separators to detect ranges, while normalization cleans text for tokenization.

Usage

```
numeric_tokens(text, keep_letters = TRUE, destructive = FALSE)
```

Arguments

text	Character vector of numeric or address fields.
keep_letters	Logical. If TRUE, retains letter suffixes like "12A". Only applies when destructive = FALSE.
destructive	Logical. If TRUE, removes all non-digit characters except whitespace. If FALSE (default), preserves letters alongside digits.

Value

A list of character vectors, one per input element. Each vector contains numeric tokens, with ranges expanded into sequences.

See Also

drop_numeric_tokens(), its inverse, to discard numbers from a token column instead.

Other token generators: custom_tokens(), generate_ngrams(), subword_tokens(), word_tokens()

Examples

```
numeric_tokens("12-14")
# list(c("12", "13", "14"))

numeric_tokens("7A 9B", keep_letters = TRUE)
# list(c("7A", "9B"))

numeric_tokens("House 5", destructive = TRUE)
# list("5")
```

plan_strategy *Plan a Search Strategy from Raw Inputs*

Description

Helps you choose a blocking before you run anything. Where `audit_strategy()` grades a strategy you have already settled on, and `rarity_distribution()` reads one column's token distribution, `plan_strategy()` compares several candidate blockings side by side and shows the trade-off between how many comparisons each one costs and how many true matches it would keep together.

It never builds the pair set, so it is safe to run on a full corpus. For each candidate blocking it reports: how many blocks it makes and how big they are, an estimate of how many record comparisons it implies, and the share of identical-token records that stay in the same block (the recall it would cost you). It also reports how much an `exact_strategy()` front stage would absorb, the shape of the leftover records, and how discriminative each column is, including a warning when a column that is often empty puts a ceiling on achievable scores.

The strategy you pass supplies only the column preparation steps; its own `block_by` is ignored, since the blocking is exactly what you are choosing here.

Usage

```
plan_strategy(
  base,
  strategy,
  target = NULL,
  block_candidates = list(),
  base_id = NULL,
  target_id = NULL,
  n_offenders = 20L,
  min_rarity_grid = NULL,
  containment = FALSE,
  ...
)
```

Arguments

`base`	A data.frame / tibble / data.table (or backend table).
`strategy`	A `Search_Strategy` supplying the tokenization to plan against.
`target`	Optional second table. `NULL` (default) plans a dedup; non-`NULL` plans a cross-table search.
`block_candidates`	Named list of candidate `block_by` specs to compare (e.g. `list(plz2 = "plz2", plz5_wz = c("plz5", "wz08_3"))`).
`base_id`	Character scalar naming the id column in `base` (required).
`target_id`	Character scalar naming the id column in `target` (defaults to `base_id`).

n_offenders	Number of top-df "offender" tokens (the fan-out drivers) to report per column. Defaults to 20.
min_rarity_grid	Optional numeric vector of min_rarity cut points for the cost curve. NULL (default) picks a grid from the rarity distribution.
containment	Logical. When TRUE, adds the per-column containment share, the one read that performs a bounded structural join. Defaults to FALSE, which keeps plan_strategy() scoring-free.
...	Backend-specific arguments, such as sample_n (DuckDB).

Value

A Strategy_Plan object.

See Also

audit_strategy() to grade a chosen strategy, rarity_distribution() for one column's distribution, exact_strategy() for the front stage it sizes.

Examples

```
strat <- search_strategy(
  workshop ~ normalize_text() + word_tokens(min_nchar = 3)
)
# Compare two candidate blockings side by side before committing to one.
plan_strategy(
  workshop_register, strat,
  block_candidates = list(area       = "postcode_area",
                          area_trade = c("postcode_area", "trade")),
  base_id = "reg_no"
)
```

---

prepare_search_data *Prepare Data for Record Linkage Search*

Description

Turn a table into the long-format token table the matching verbs work on: it applies each column's preparation steps, splits the text into tokens, and attaches the id and any blocking columns. The other verbs (detect_duplicates(), search_candidates()) call this for you, so you rarely need it directly; reach for it when you want to see or post-process the tokens yourself.

Usage

```
prepare_search_data(data, id, strategy, ...)
```

Arguments

data	A data.frame / tibble / data.table (or db table in other backends).
id	Character scalar naming the ID column in data.
strategy	A Search_Strategy object.
...	Additional arguments passed to backend-specific methods.

Value

A long-format token table with one row per token, carrying the id, the source column, the token, a row_id, and any blocking columns.

See Also

inspect_tokens() for a quick per-column look at the tokens.

---

rarity_distribution *Read the Token Rarity Distribution*

Description

A pre-match read of how token rarity is distributed in your data. For each column (and block, when the strategy blocks) it reports the spread of token document frequency and rarity, plus an offender list: the most common tokens, the ones that drive a match to balloon. Use it to set min_rarity and max_token_df from what is actually in the data instead of guessing.

It never builds the pair set: it only tokenizes and measures rarity, so it is cheap enough to run on a full corpus before committing to a strategy.

Usage

```
rarity_distribution(data, id, strategy, ...)
```

Arguments

data	A data.frame / tibble / data.table (or backend-specific table).
id	Character scalar naming the ID column in data.
strategy	A Search_Strategy object.
...	Additional backend-specific arguments. Notably n_offenders (integer; how many top-df tokens to list, default 20) and sample_n (DuckDB: rows to pull before delegating; default all).

Value

A Rarity_Distribution object.

See Also

search_strategy() for the min_rarity / max_token_df levers this verb informs; audit_strategy() for the broader pre-match audit.

Examples

```
strat <- search_strategy(
  workshop ~ normalize_text() + word_tokens(min_nchar = 3),
  block_by = c("postcode_area", "trade")
)
# Read the token distribution and the most common tokens before matching.
rarity_distribution(workshop_register, "reg_no", strat)
```

---

rarity_histogram *Bar chart of median token rarity per column*

Description

Bar chart of median token rarity per column

Usage

```
rarity_histogram(x, ...)
```

Arguments

x	A Strategy_Audit object from audit_strategy().
...	Passed to tinyplot::tinyplot().

Value

Invisibly, the plotted data.table (column_rarity_stats).

---

recommendations *Recommendations from a Diagnostic Object*

Description

Accessor returning the recommendations strings stored on a diagnostic result object. Returns character(0) when no recommendations fired. The same strings are surfaced inline by the object's print() method.

Methods for individual classes live alongside those classes - diagnostic classes (Match_Overview, Strategy_Audit) in diagnostic_classes.R; calibration classes (Calibrated_Matches, Filter_Calibration) in calibration_classes.R.

Usage

```
recommendations(x, ...)
```

Arguments

x	A diagnostic result object (Strategy_Audit, Match_Overview, Calibrated_Matches, Filter_Calibration).
...	Reserved for future methods.

Value

A character vector.

---

resolve_entities *Group Matched Pairs into Entities*

Description

Take a list of matched record pairs (an edge list) and turn it into entities: records that link directly or through a chain of links are grouped together, each group gets an entity number, and one record in each group is marked as its representative.

This is the grouping step detect_duplicates() performs internally, exposed on its own so you can resolve any pair list into entities, for example the output of search_candidates() or a set of links you assembled yourself.

Usage

```
resolve_entities(
  edges,
  id_a,
  id_b,
  score = NULL,
  vertices = NULL,
  rep_by = NULL,
  block_by = NULL,
  ...
)
```

Arguments

edges	A backend table of record-pair edges (one row per edge).
id_a, id_b	Character scalars naming the two endpoint columns in edges.
score	Optional character scalar naming a per-edge score column in edges. When supplied, within-entity rank is ordered by descending best score (the maximum over a vertex's incident edges) and that best score is returned as an extra score column. When NULL, ranking falls back to the rep_by/id rule.

vertices	Optional. All vertex ids to include, so that ids absent from every edge come back as their own singleton entity (rank 1, rep = self). Either an atomic vector of ids, or a table with an id column (plus any rep_by column). When NULL, only ids appearing in edges are returned.
rep_by	Optional character scalar naming a priority column (on the vertices table) used to pick the canonical representative: the member with the smallest rep_by wins, ties broken by smallest id.
block_by	Optional character vector of columns in edges used to run connected components per block (DuckDB backend).
...	Additional arguments passed to backend-specific methods.

Details

The result does not depend on the order of rows in edges: the same pairs always produce the same entity, rep, and rank. Entity numbers are assigned by the smallest member id in each group. The representative (the rank-1 member) is chosen by highest best score when a score column is given, then by smallest rep_by when given, then by smallest id.

Value

One row per resolved vertex:

id The vertex id.

entity Integer entity (connected-component) label.

rep The canonical representative id of the entity.

rank Rank within the entity; rank 1 is the representative.

score Best incident-edge score per vertex (only when score is supplied).

Examples

```
# r1-r2 and r2-r3 chain into one entity; r4-r5 form another
edges <- data.table::data.table(
  a = c("r1", "r2", "r4"),
  b = c("r2", "r3", "r5")
)
resolve_entities(edges, id_a = "a", id_b = "b")
```

---

sample_matches *Sample Matches for Review*

Description

Sampling diagnostic (Q4). Modes: "high", "low", "borderline", "ambiguous", "top_gap", "random".

Usage

```
sample_matches(matches, ...)
```

Arguments

matches	Match output table.
...	Method-specific arguments. Standard arguments: mode (one of the sampling modes above), n (number of rows to sample), and mode-specific extras (e.g. threshold for "borderline").

Value

A Match_Sample object.

Examples

```
strat <- search_strategy(
  workshop ~ normalize_text() + word_tokens(min_nchar = 3),
  block_by  = c("postcode_area", "trade"),
  threshold = 0.7
)
matches <- search_candidates(
  workshop_listings, workshop_register,
  base_id = "listing_id", target_id = "reg_no", strategy = strat
)
# Pull the borderline pairs near the threshold, the ones worth eyeballing.
sample_matches(matches, mode = "borderline", n = 5, threshold = 0.7)
```

---

score_density *Kernel density of the score distribution*

Description

Expands the pre-binned histogram to approximate raw scores before passing to the density estimator.

Usage

```
score_density(x, threshold = x@score_dist$threshold %||% NA_real_, ...)
```

Arguments

x	A Match_Overview object from summarise_matches().
threshold	Numeric. Draws a dashed vertical line. Defaults to the threshold stored in x@score_dist$threshold when available.
...	Passed to tinyplot::tinyplot().

Value

Invisibly, the `data.table` of expanded scores.

`score_embeddings` *Score Embedding Pairs Using Cosine Similarity*

Description

Compute cosine similarity scores between base and target embeddings. This is a pure scoring function that operates on pre-computed embeddings.

Usage

```
score_embeddings(base_embeddings, target_embeddings, strategy, ...)
```

Arguments

`base_embeddings`	A table with columns: `id` and `embedding`.
`target_embeddings`	A table with columns: `id` and `embedding`.
`strategy`	An `Embedding_Strategy` object (used for normalization settings).
`...`	Additional arguments passed to backend-specific methods.

Value

A backend-specific table with columns: `base_id`, `target_id`, `score`.

`score_histogram` *Bar chart of the pre-binned score distribution*

Description

Bar chart of the pre-binned score distribution

Usage

```
score_histogram(x, threshold = x@score_dist$threshold %||% NA_real_, ...)
```

Arguments

`x`	A `Match_Overview` object from `summarise_matches()`.
`threshold`	Numeric. Draws a dashed vertical line. Defaults to the threshold stored in `x@score_dist$threshold` when available.
`...`	Passed to `tinyplot::tinyplot()`.

Value

Invisibly, the plotted `data.table` (histogram with bin_mid column).

`search_candidates` *Search for Candidate Matches Between Tables*

Description

Find candidate matches between two tables: for each record on one side, the records on the other side that share enough rare, informative token content to score at or above the threshold. This is the cross-table counterpart of `detect_duplicates()`.

Pass a `search_strategy()` for fuzzy, scored matching, or an `exact_strategy()` to keep only pairs whose token sets are identical.

Usage

```
search_candidates(base_table, target_table, base_id, target_id, strategy, ...)
```

Arguments

`base_table`	A data.frame, tibble, data.table, or backend table.
`target_table`	The table to search against.
`base_id`	Character scalar naming the ID column in `base_table`.
`target_id`	Character scalar naming the ID column in `target_table`.
`strategy`	A `Search_Strategy` (or `Exact_Strategy`) describing how to tokenize each column, how to block, and the matching threshold.
`...`	Additional arguments passed to backend-specific methods, such as `threshold` and `weights`.

Value

A table with two rows per matched pair (one for the base record, one for the target record), sharing a `match_id`:

match_id Identifier shared by the two rows of a matched pair.

score The pair's match score.

source `"base"` or `"target"`.

id The record ID.

`<original columns>` Every other column from the source table.

rank Rank of this candidate among a record's matches.

See Also

`detect_duplicates()` for the within-table version, `extract_unmatched()` for the residual, `multi_stage_search()` for staged passes.

Examples

```
data(base_example)
data(target_example)

strat <- search_strategy(
  Nachname ~ normalize_text() + word_tokens(min_nchar = 3),
  Vorname  ~ normalize_text() + word_tokens(min_nchar = 3),
  Ort      ~ normalize_text(),
  block_by = "Kreis",
  threshold = 0.8
)

matches <- search_candidates(
  base_example, target_example,
  base_id = "id_base", target_id = "id_target",
  strategy = strat
)
head(matches)
```

`search_strategy` *Define a Search Strategy for Record Linkage*

Description

Creates a Search_Strategy object that specifies how columns should be preprocessed for token index based record linkage, along with optional weights, blocking variables, rarity computation method, rIP smoothing, and similarity threshold.

Usage

```
search_strategy(
  ...,
  block_by = NULL,
  weights = numeric(),
  rarity = "inverse_freq",
  rarity_scope = c("block", "global"),
  min_rarity = 0,
  max_token_df = Inf,
  threshold = 0.9,
  smoothing = smooth_rip_identity(),
  max_candidates = Inf,
  max_fanout = 5e+07,
  on_fanout = c("cap", "abort", "off"),
  feedback_strength = 0,
  on_missing = c("penalise", "renormalise")
)
```

Arguments

`...`	Two sided formulas of the form `column ~ preprocessing_steps`. The left hand side names the column; the right hand side contains one or more function calls to apply in sequence (for example `name ~ normalize_text() + word_tokens(min_nchar = 3)`).
`block_by`	Optional character vector of column names to use for blocking. Candidate searches will be restricted to records sharing the same blocking key values. Default is `NULL` (no blocking).
`weights`	Optional named numeric vector of weights for similarity scoring. Names should correspond to columns. Default is `numeric()` (uniform weights).
`rarity`	Character scalar choosing how a token's rarity (its informativeness, the weight it carries in scoring) is computed from token counts. A shared rare token is strong evidence two records match; a shared common one is weak. The four methods differ in how hard they push common tokens down. Let `f` be the token's frequency, `df` its document frequency (how many records contain it), and `N` the record count, all measured over the scope set by `rarity_scope`. One of: `"inverse_freq"` (default) `rarity = 1 / f`. Simple and robust: a token seen once is maximally rare, one seen often counts for little. A good first choice and what every other article uses. `"smoothed_inverse_freq"` `rarity = 1 / (f + 1)`. The same shape, damped so the very rarest tokens do not dominate quite as sharply. Reach for it when single-occurrence tokens (often typos) are swinging scores too much. `"tfidf"` `rarity = tf * idf` with `tf = f / sum(f)` and `idf = log(1 + N / df)`. Weighs a token by how few records carry it, not just its raw count. Use when the same token recurs at very different rates across columns or blocks and you want document spread to matter. `"bm25"` `rarity = log((N - df + 0.5) / (df + 0.5))`, the Okapi BM25 inverse-document-frequency term. The most aggressive: it drives common tokens toward zero and turns *negative* for a token in more than half the records, which actively penalises boilerplate. Use on corpora thick with shared legal-form or trade words ("Ltd", "Joinery") that you want suppressed hard. Default is `"inverse_freq"`; most linkages never need to change it. Inspect the token distribution with `rarity_distribution()` before switching, and note `rarity_scope` decides whether these counts are block-local or corpus-wide.
`rarity_scope`	Character scalar, `"block"` (default) or `"global"`, selecting the scope over which a token's rarity (informativeness) is measured. `"block"` measures rarity within each block (the historical and only previous behaviour). `"global"` measures it across the whole corpus, so a token's informativeness no longer depends on which block it lands in. This is the chain defence for region-free linking: a globally common name (think a franchise) gets low global rarity and is dropped by `min_rarity`, while a distinctive brand reads as a strong link signal regardless of where it appears. Only the rarity metric and the `min_rarity` gate follow this argument; the cost axis (block-local `df`, `max_token_df`, the fan-out guard) stays block-local under both scopes.

`min_rarity` Numeric scalar specifying the minimum rarity value required for a token to be included in similarity scoring. Tokens with rarity below this threshold are filtered out. Default is `0`.

`max_token_df` Numeric scalar specifying the maximum raw document frequency a token may have within its `(block, column)` to be kept. Tokens appearing in more than `max_token_df` records are dropped *before* the token-overlap join, so a single hyper-common token (a house number, `STRASSE`) can't fan out a block even at `min_rarity = 0`. The blunt document-frequency companion to the rarity-metric `min_rarity`; the two cut on different axes and compose. Default is `Inf` (off). See `rarity_distribution()` to choose a value from the token distribution.

`threshold` Numeric scalar specifying the minimum relative identification potential required for two records to be considered matches. Default is `0.9`.

`smoothing` A `Smoothing` object created by one of the smooth_rip helpers that controls how rIP values are smoothed before scoring. Default is `smooth_rip_identity()`.

`max_candidates` Numeric scalar specifying the maximum number of candidate matches to retain per record. Default is `Inf` (no limit). When finite, only the top `max_candidates` highest scoring matches are kept per record.

`max_fanout` Numeric scalar. The always-on guard against a single hot or boilerplate token (think a directory publisher's name, or a stopword that slipped through) fanning one block into a huge number of pairwise comparisons. This is the same failure `min_rarity` / `max_token_df` address, but on by default. It caps the estimated number of record pairs the token-overlap join will form, predicted cheaply from token frequencies before the join runs (no pairs are built to measure it). Default `5e7`; set `Inf` (or `on_fanout = "off"`) to disable. Use `rarity_distribution()` or `plan_strategy()` to pick a value for your data.

`on_fanout` What to do when the estimated fan-out exceeds `max_fanout`: `"cap"` (default) auto-drops the smallest set of hyper-common tokens needed to get under budget (they carry near-zero rarity, so scores barely move) and emits a loud warning naming what was dropped; `"abort"` stops with an actionable error instead; `"off"` disables the guard entirely.

`feedback_strength` Numeric scalar controlling feedback weighted scoring. Default is `0` (disabled). Positive values adjust scores based on the proportion of matched tokens.

`on_missing` How to score a pair when a weighted column is **empty on both records**. With `"penalise"` (default) the column still counts against the score. For example, if `Strasse` has weight 0.3 and a record's street is blank, that record's score can never rise above 0.7, so a threshold of 0.8 will never match it even on a perfect name. `"renormalise"` removes that ceiling: it spreads the weight of any column blank on *both* sides across the columns that are present (a column present on only one side still counts as a genuine mismatch). This is powerful but aggressive, since it turns a record with no street into a name-only matcher, so it is opt-in and never the default. If you mainly want to handle empty columns, the safer route is to run an `exact_strategy()` stage first, whose matches do not depend on weights or thresholds.

Value

A Search_Strategy object.

Examples

```
# Tokenize two name columns, block on region, keep pairs scoring at least 0.8.
strat <- search_strategy(
  Nachname ~ normalize_text() + word_tokens(min_nchar = 3),
  Vorname  ~ normalize_text() + word_tokens(min_nchar = 3),
  block_by  = "Kreis",
  threshold = 0.8
)
strat
```

---

similarity_histogram *Histogram of sampled pairwise cosine similarities*

Description

Histogram of sampled pairwise cosine similarities

Usage

```
similarity_histogram(x, threshold = attr(x, "threshold"), bins = 30L, ...)
```

Arguments

x	An Embedding_Audit object from audit_strategy().
threshold	Numeric. Draws a dashed vertical line at the strategy threshold (default: attr(x, "threshold")).
bins	Integer. Number of histogram bins.
...	Passed to tinyplot::tinyplot().

Value

Invisibly, the histogram data.table with columns bin_lower, bin_upper, bin_mid, count.

smooth_rip *Configure rIP smoothing for a search strategy*

Description

Helper functions that construct S7 Smoothing objects used by search_strategy() to control how relative identification potential (rIP) is smoothed before scoring.

All helpers are pure configuration; they do not perform any computation by themselves. Backend methods for detect_duplicates() and search_candidates() interpret the resulting Smoothing object.

Usage

```
smooth_rip_identity()

smooth_rip_log()

smooth_rip_offset(alpha = 0.5)

smooth_rip_softmax(temperature = 1)
```

Arguments

alpha	Numeric scalar; offset that is added to rIP values prior to normalization. Must be non negative.
temperature	Numeric scalar; softmax temperature parameter. Must be strictly positive.

Details

rIP Smoothing Helpers

Value

An object inheriting from Smoothing that can be passed to the smoothing argument of search_strategy().

Functions

- smooth_rip_identity(): Identity rIP smoothing (no transformation beyond standard per record normalization). This is the default.
- smooth_rip_log(): Logarithmic rIP smoothing. Backends typically apply log1p(rIP) and then renormalize within each record and column.
- smooth_rip_offset(): Offset based rIP smoothing with a constant offset alpha that is added to all rIP values before renormalization.
- smooth_rip_softmax(): Softmax style rIP smoothing with a temperature parameter that controls how sharp or flat the transformed distribution is.

See Also

search_strategy()

`stage_coverage_plot` *Line plot of cumulative base coverage by stage*

Description

Uses percentage coverage when base was supplied to compare_stages(), raw record counts otherwise.

Usage

```
stage_coverage_plot(x, ...)
```

Arguments

x	A Stage_Comparison object from compare_stages().
...	Passed to tinyplot::tinyplot().

Value

Invisibly, the plotted data.table (marginal_coverage with stage_idx).

`stage_score_plot` *Grouped bar chart of score distributions by stage*

Description

Grouped bar chart of score distributions by stage

Usage

```
stage_score_plot(x, ...)
```

Arguments

x	A Stage_Comparison object from compare_stages().
...	Passed to tinyplot::tinyplot().

Value

Invisibly, the plotted data.table (score_dist_by_stage with bin_mid).

`street_stopwords` *Multilingual Street-Name Stopwords*

Description

Locative particles and articles that recur inside multi-word street names but carry no discriminative signal for matching - German `AN DER`, French `DE LA`, Italian `DELLA`, and so on. Used by `normalize_street()` when `drop_stopwords = TRUE` to collapse e.g. `"An der Alster"` to `"ALSTER"`.

Usage

```
street_stopwords
```

Format

An object of class `tbl_df` (inherits from `tbl`, `data.frame`) with 58 rows and 2 columns.

Details

The list is deliberately tight: only true prepositions and articles, never adjectives (`"NEUE"`, `"GROSSE"`) or directionals that can themselves be the distinguishing part of a name. Entries are uppercase ASCII so they join directly against `normalize_street()`'s already-uppercased, transliterated tokens. When a `lang` is supplied to `normalize_street()`, only that language's particles are removed.

Format:

A tibble with two columns:

stopword Character string. The particle in uppercase ASCII (e.g. `"AN"`, `"DER"`, `"DE"`, `"DELLA"`).

lang ISO 639-1 language code (`"de"`, `"en"`, `"fr"`, `"es"`, `"it"`, `"pt"`, `"nl"`).

Source

Manually curated from common multi-word street-name patterns across languages. Expandable as new particles are encountered.

See Also

`normalize_street()`, street_types

`street_types` *Multilingual Street-Type Normalization Dictionary*

Description

A curated cross-linguistic dictionary of street-type forms used for robust address standardization and record linkage. Each entry maps a *variant* - including abbreviations, orthographic alternatives, morphological forms, and transliterated spellings - to a *canonical* street-type label.

Usage

```
street_types
```

Format

An object of class `tbl_df` (inherits from `tbl`, `data.frame`) with 143 rows and 4 columns.

Details

Unlike simple suffix lists, this dictionary encodes language-specific normalization rules. Each variant is marked as either:

- **"exact"** - the variant should only match a token when it appears *exactly*, e.g. `"st."` -> `"STREET"` (English), `"pl"` -> `"PLAZA"` (Spanish)
- **"suffix"** - the variant may safely match a token *ending with* that sequence, e.g. `"gatan"` -> `"GATA"` (Swedish), `"strasse"` -> `"STRASSE"` (German)

By separating exact vs. suffix behaviour and tagging each entry with an ISO language code, **joinery** can normalize addresses *without* incorrect transformations (e.g. preventing `"LINCOLN"` -> `"LANE"`, or `"VICTOR"` -> `"RUE"`). This structure enables high-precision multilingual address cleaning.

Languages Covered:

The dictionary currently includes major street-type systems from:

- **German** - Straße, Gasse, Weg, Platz, Allee, etc.
- **English** - Street, Road, Avenue, Boulevard, Lane, etc.
- **French** - Rue, Avenue, Boulevard, Impasse, Quai, Chemin, etc.
- **Spanish** - Calle, Avenida, Paseo, Plaza, Camino, etc.
- **Italian** - Via, Piazza, Corso, Viale, etc.
- **Portuguese** - Rua, Avenida, Praça, Alameda, Travessa, etc.
- **Polish** - Ulica, Aleja, Plac, Osiedle, etc.
- **Dutch** - Straat, Laan, Weg, Plein, etc.
- **Turkish** - Sokak, Cadde, Bulvar, Meydan, etc.
- **Swedish** - Gata, Gatan, Vägen, Torg, etc.
- **Danish/Norwegian** - Gade, Vej, Plads, etc.
- **Greek (transliterated)** - Odos, Leoforos, Plateia
- **Russian (transliterated)** - Ulitsa, Prospekt, Pereulok, etc.

Additional languages and street-type systems can be incorporated as needed.

Use in `normalize_street()`**:**

`street_types` is used by `normalize_street()` to:

- standardize street-type tokens to a canonical form,
- optionally apply language-specific suffix rules (`lang = "de"`, `"sv"`, etc.),
- avoid over-normalization by matching only valid variants for the specified language,
- support multilingual cleaning workflows in data preprocessing and record linkage.

Format:

A tibble with four columns:

canonical Character string. The standardized street-type label in uppercase ASCII (e.g. `"STRASSE"`, `"AVENUE"`, `"PLAC"`).

variant Character string. A lowercase spelling, abbreviation, transliteration, or inflected form seen in raw address data (e.g. `"str."`, `"straße"`, `"avda"`, `"gatan"`).

type Either `"exact"` or `"suffix"`, indicating whether the variant should match only whole tokens or may safely match as a word-final suffix.

lang ISO 639-1 language code (e.g. `"de"`, `"en"`, `"fr"`, `"sv"`), used to restrict normalization to the appropriate street-type system.

Source

Manually curated based on postal conventions, open datasets, and commonly observed street-name variations across languages. The dictionary is periodically expanded as new variants are encountered in real-world data.

---

strip_vowels *Strip vowels from text (consonant skeleton)*

Description

Reduces text to its consonant skeleton by removing vowels (A, E, I, O, U, including accented variants). Two spellings that differ only in their vowels, such as `"MEYER"` and `"MAYER"` or `"MUELLER"` and `"MULLER"`, collapse to the same skeleton, so they match despite the difference. It is a lighter-weight alternative to the phonetic encoders (`as_soundex()`, `as_metaphone()`) when you only want to ignore vowel variation.

Usage

```
strip_vowels(text)
```

Arguments

`text`	A character vector.

Details

Returns text, so it goes ahead of a token generator in a pipeline.

Value

A character vector with vowels removed, upper-cased and ASCII-folded.

See Also

`as_soundex()` and `as_metaphone()` for full phonetic encoding.

Other text normalizers: `normalize_street()`, `normalize_text()`

Examples

```
strip_vowels("Mueller")   # "MLLR"
strip_vowels("Cafe Noir") # "CF NR"
strip_vowels(c("Anna", "Peter"))
```

---

subword_tokens *Split text into learned subword tokens*

Description

The apply half of the subword pair: place a model fitted by `find_subwords()` in a strategy formula and every record is split into the learned word pieces: `name ~ normalize_text() + subword_tokens(sw)`.

Usage

```
subword_tokens(text, model)
```

Arguments

`text`	A character vector to tokenize.
`model`	A fitted `Subword_Model` from `find_subwords()`.

Details

Use it where word tokens are too brittle: compound words (`"Maschinenbaugesellschaft"` never word-matches `"Maschinenbau"`), frequent misspellings, or OCR noise. Two spellings of the same name share most of their pieces even when they share no whole word, so they can still meet and score. Scoring is unchanged: subword pieces are ordinary tokens, each weighted by its own rarity.

The model decides how text is split, so pass the text through the same cleaning steps the model was fitted on (see `find_subwords()`).

Value

A list of character vectors, one per input element, each holding that element's subword tokens.

See Also

find_subwords() to fit the model; custom_tokens(), which this is a named shortcut for; drop_short_tokens() to drop single-letter pieces.

Other token generators: custom_tokens(), generate_ngrams(), numeric_tokens(), word_tokens()

Examples

```
if (requireNamespace("sentencepiece", quietly = TRUE)) {
  firms <- rep(c(
    "maschinenbau mueller", "tischlerei schmidt",
    "holzbau wagner", "metallbau krause"
  ), 40)
  sw <- find_subwords(firms, vocab_size = 60)
  subword_tokens(c("maschinenbaugesellschaft", "holzbau wagner kg"), sw)
}
```

---

summarise_matches *Summarise a Match Result*

Description

Post-match overview (Q2). Auto-detects whether the input is a duplicate table (presence of duplicate_group column) or a candidate table (presence of match_id and source columns), and reports score distribution, coverage (when base / target are supplied), cluster-size or candidates-per-record distribution, and top-1-vs-top-2 score-gap distribution for candidates. Recommendations link symptoms to strategy levers.

Usage

```
summarise_matches(matches, ...)
```

Arguments

matches	Match output table from detect_duplicates() or search_candidates().
...	Method-specific arguments. The data.table method accepts: base (optional base input table for coverage), target (optional target input table for candidate coverage), and bins (integer number of histogram bins for the score distribution; default 50).

Value

A Match_Overview object.

Examples

```
s <- search_strategy(
  Nachname ~ normalize_text() + word_tokens(min_nchar = 3),
  block_by = "Kreis",
  threshold = 0.8
)
dups <- detect_duplicates(base_example, "id_base", s)
summarise_matches(dups, base = base_example)
```

---

target_example *Target dataset for record linkage example*

Description

A dataset containing 3,000 person records designed to match with `base_example`. Approximately 80% of records correspond to records in the base dataset but with realistic errors and variations (typos, abbreviated names, title additions, street name variations). The remaining 20% are new records with no match.

Usage

```
target_example
```

Format

A tibble with 3,000 rows and 7 variables:

actual_link The actual base_id in the simulation process

id_target Character identifier for target records (T0001-T3600)

Vorname First name, may include titles, initials, or middle names

Nachname Last name, may contain typos or token swaps

Strasse Street name with possible abbreviations or typos

Hausnummer House number with possible letter suffixes

Ort City or town name, may contain typos

Kreis Administrative district (Kreis)

Source

Synthetically generated by distorting 80% of `base_example` records and adding 20% new unmatched records.

token_contribution_plot

Horizontal bar chart of per-token score contributions, coloured by column

Description

Horizontal bar chart of per-token score contributions, coloured by column

Usage

```
token_contribution_plot(x, ...)
```

Arguments

x	A `Match_Explanation` object from `explain_match()`.
...	Passed to `tinyplot::tinyplot()`.

Value

Invisibly, the plotted `data.table` (shared_tokens with token_label).

token_frequency_plot

Bar chart of average tokens per record per column

Description

Bar chart of average tokens per record per column

Usage

```
token_frequency_plot(x, ...)
```

Arguments

x	A `Strategy_Audit` object from `audit_strategy()`.
...	Passed to `tinyplot::tinyplot()`.

Value

Invisibly, the plotted `data.table` (column_token_stats).

token_shapes *Convert tokens to shape signatures*

Description

Reduces each token to its letter/digit pattern: every letter becomes `"A"`, every digit `"N"`, anything else `"X"`. The signature ignores the actual characters and keeps only the layout, which is useful for matching on the format of a code or identifier (postal codes, licence plates, product codes) rather than its exact value, or as a coarse blocking key.

Usage

```
token_shapes(tokens)
```

Arguments

`tokens`	A list of character vectors.

Details

It transforms a token column, so it runs after a token generator such as `word_tokens()`.

Value

A list of character vectors of shape signatures, one signature per input token.

See Also

Other token transformers: `drop_numeric_tokens()`, `drop_short_tokens()`, `extract_initials()`, `filter_stopwords()`, `fuzzy_tokens()`, `use_dictionary()`

Examples

```
token_shapes(list(c("MUELLER", "A12B")))
# list(c("AAAAAAA", "ANNA"))
```

tokenize *Tokenize Text with a Fitted Tokenizer*

Description

Most tokenizers in joinery are plain functions, like `word_tokens()`. Some tokenizers instead carry fitted state, for example a subword vocabulary learned from your corpus. Such a tokenizer is an object, and `tokenize()` is how that object turns text into tokens. You rarely call it yourself: place the object in a strategy formula with `custom_tokens()` and joinery calls `tokenize()` for you.

To bring your own tokenizer, write a `tokenize()` method for its class with `S7::method()`. The method must return a list with one element per element of `text`, each element a character vector of that record's tokens (`character(0)` for a record with none).

Usage

```
tokenize(tokenizer, text, ...)
```

Arguments

`tokenizer`	A fitted tokenizer object.
`text`	A character vector, one element per record.
`...`	Additional arguments passed to methods.

Value

A list of character vectors, one per element of `text`.

See Also

`custom_tokens()`, which puts a tokenizer into a strategy formula.

top_gap_density *Bar chart of top-1 vs top-2 score gap distribution (candidates only)*

Description

Bar chart of top-1 vs top-2 score gap distribution (candidates only)

Usage

```
top_gap_density(x, ...)
```

Arguments

x	A Match_Overview object from summarise_matches() with match_type == "candidates".
...	Passed to tinyplot::tinyplot().

Value

Invisibly, the plotted data.table (top_gap_dist with bin_mid).

---

use_dictionary *Map tokens to canonical groups with a lookup table*

Description

When you already know which tokens mean the same thing (a curated synonym list, brand-name variants, a code-to-label table), use_dictionary() rewrites each token to its group label so the variants collapse to one token and match. Use it when the mapping is known in advance; when you instead want joinery to discover near-duplicates from the data, use fuzzy_tokens().

Usage

```
use_dictionary(text, dict)
```

Arguments

text	A character vector of tokens to look up.
dict	A data.table::data.table with a tokens column and a token_group column. Rows whose tokens value matches an input token supply that token's group label.

Details

Tokens absent from the dictionary return no group, so chain this after a token generator and keep a sharper field alongside it.

Value

A list of character vectors, one per input element, holding the matched group labels (empty when the token is not in dict).

See Also

fuzzy_tokens() to discover groups from the data instead.

Other token transformers: drop_numeric_tokens(), drop_short_tokens(), extract_initials(), filter_stopwords(), fuzzy_tokens(), token_shapes()

Examples

```
dict <- data.table::data.table(
  tokens = c("example", "sample"),
  token_group = c("example/sample", "example/sample")
)
use_dictionary("example", dict)
use_dictionary("nonexistent", dict)
```

---

vocab_overlap_plot *Bar chart of vocabulary overlap between base and target per column*

Description

Bar chart of vocabulary overlap between base and target per column

Usage

```
vocab_overlap_plot(x, ...)
```

Arguments

x	A `Strategy_Audit` object from `audit_strategy()` called with `target` supplied.
...	Passed to `tinyplot::tinyplot()`.

Value

Invisibly, the plotted `data.table`.

---

word_tokens *Split text into word tokens*

Description

The workhorse tokenizer. It splits each string on whitespace into a vector of words, the tokens joinery matches on. It almost always follows `normalize_text()`, which strips punctuation and case first so the split is clean: `name ~ normalize_text() + word_tokens()`.

Usage

```
word_tokens(text, min_nchar = 0)
```

Arguments

text	A character vector to split into words.
min_nchar	Minimum token length to keep. Tokens shorter than this are dropped. Defaults to 0 (keep everything).

Details

Set min_nchar to drop very short tokens (single initials, stray letters) that match too easily and add noise.

Value

A list of character vectors, one per input element, each holding that element's word tokens.

See Also

normalize_text(), the usual preceding step; filter_stopwords() to drop common words by name.

Other token generators: custom_tokens(), generate_ngrams(), numeric_tokens(), subword_tokens()

Examples

```
word_tokens("this is an example")
word_tokens("this is an example", min_nchar = 3)  # drops "is", "an"
```

---

workshop_listings *Workshop external directory (target) for record linkage examples*

Description

A messier external directory of the same UK workshops as workshop_register, the target table for cross-source linkage. Listings carry the realistic distortions a scrappy directory introduces, planted as labelled tiers (via gen_tier) so each exercises a specific joinery feature: slogan-stuffed supersets (exact containment), movers that changed postcode area (token blocking with global rarity), phonetic name variants (Cologne/Soundex encoders), shared-venue and bare-category rows that bait over-linking (the containment guards), and common-surname homonyms (ambiguity and calibration). The matchable columns share names with workshop_register so one formula serves both tables.

Usage

```
workshop_listings
```

Format

A tibble with 894 rows and 8 variables:

listing_id Character identifier for listings (e.g. "L00042")

workshop Directory rendering of the business name, often messy

proprietor Directory rendering of the proprietor name; may be missing

trade Trade; half the blocking key

postcode_area UK outward-code area; half the blocking key

town Town for the postcode area

actual_link Evaluation only. The `reg_no` this listing refers to, or `NA` for a workshop absent from the register.

gen_tier Evaluation only. The feature-exercise tier (clean, slogan, variant, mover, phonetic, hub_member, hub_trap, category_trap, homonym_area, homonym_block, homonym_total, new).

Source

Synthetically generated by `data-raw/generate_workshop_example.R` from `workshop_register` and a frozen LLM seed of messy renderings.

See Also

`workshop_register`

`workshop_panel` *Multi-year workshop panel for cross-year linkage examples*

Description

A small pooled-long panel of the same UK woodworking workshops as `workshop_register`, observed across five years (2019 to 2023). It is the runnable example for cross-year entity resolution: each row is one workshop in one year, names drift over time, a minority of workshops relocate to a different postcode area, and workshops enter and exit so trajectories have gaps. The task is to recover the stable identity (`true_entity`) behind the year-by-year rows. A reappearance window of five years matches the design of the real Yellow Pages panel the package was built against.

Usage

```
workshop_panel
```

Format

A tibble with 847 rows and 10 variables:

record_id Character row id, unique per workshop-year (e.g. "YR-00042")

year Observation year, 2019 to 2023

workshop Business name as recorded that year, with light per-year noise

proprietor Proprietor name, in a directory rendering that varies by year

trade One of eight woodworking trades

postcode_area UK outward-code area; changes mid-trajectory for movers

town Town for the postcode area

established Year the workshop was established (stable within an entity)

true_entity Evaluation only. The stable entity key (the `workshop_register` `reg_no`); every year-row of one workshop shares it.

change_tier Evaluation only. The cross-year challenge the entity carries: `stable` (per-year noise only), `name_drift` (a structural name change part-way through), `mover` (a postcode-area relocation), or `phonetic` (a code-preserving stem twin).

Source

Synthetically generated by `data-raw/generate_workshop_panel.R`, which draws core workshops from `workshop_register` and gives each a year span with drift. Seeded and offline; ships no real business.

See Also

`workshop_register`, `workshop_listings`

`workshop_register` *Workshop guild register (base) for record linkage examples*

Description

A synthetic register of UK joinery and carpentry workshops, styled like an excerpt from a Guild of Master Craftsmen trade roll. It is the clean base table for the linkage examples: distinctive workshop names paired with boilerplate trade and legal-form terms, so the rarity-versus-boilerplate behaviour of the matcher is visible. Pairs with `workshop_listings`, the messier external directory. Block on `(postcode_area, trade)`.

Usage

```
workshop_register
```

Format

A tibble with 1,052 rows and 15 variables:

reg_no Character registration number, the base id. Most are `"GMC-#####"`; planted duplicates, homonyms, and shared-venue rows carry `"GMC-D####"`, `"GMC-H####"`, and `"GMC-V####"` prefixes respectively.

workshop Canonical business name (distinctive stem plus trade and legal-form boilerplate)

proprietor Proprietor name

trade One of eight woodworking trades; half the blocking key

legal_form Ltd, LLP, Partnership, or Sole Trader

postcode_area UK outward-code area (e.g. "LS"); half the blocking key

town Town for the postcode area

address Street address

established Year the workshop was established

employees Headcount, varying with legal form

apprentices Number of apprentices

guild_member Logical, whether a current guild member

sic UK SIC 2007 industry code for the trade

true_entity Evaluation only. Same-entity key: planted duplicate rows share it, homonym workshops get distinct keys.

gen_tier Evaluation only. Which generation tier the row belongs to. Three rows are `hub_trap`: short-named shared venues ("Trinity Workshops", "The Forge", "Riverside Works") that are themselves guild registered. Their two-token names are a forward-containment subset of every `<workshop>, <venue>` listing, so they bait an exact containment strategy into merging unrelated workshops; the `min_containment_tokens` guard blocks them.

Source

Synthetically generated. Distinct workshop identities come from a frozen LLM seed (`data-raw/llm_workshop_seed.R`); all geography, colour columns, planted duplicates, and homonyms are added by the seeded, offline `data-raw/generate_workshop_example.R`. Ships no real business.

See Also

`workshop_listings`

Index