Package 'couplr'

November 7, 2026

Title Optimal Pairing and Matching via Linear Assignment

Version 1.9.0

Description Solves optimal pairing and matching problems using linear assignment algorithms. Provides implementations of the Hungarian method (Kuhn 1955) <doi:10.1002/nav.3800020109>, Jonker-Volgenant shortest path algorithm (Jonker and Volgenant 1987) <doi:10.1007/BF02278710>, Auction algorithm (Bertsekas 1988) <doi:10.1007/BF02186476>, cost-scaling (Goldberg and Kennedy 1995) <doi:10.1007/BF01585996>, scaling algorithms (Gabow and Tarjan 1989) <doi:10.1137/0218069>, push-relabel (Goldberg and Tarjan 1988) <doi:10.1145/48014.61051>, and Sinkhorn entropy-regularized transport (Cuturi 2013) <doi:10.48550/arxiv.1306.0895>. Designed for matching plots, sites, samples, or any pairwise optimization problem. Supports rectangular matrices, forbidden assignments, data frame inputs, batch solving, k-best solutions, and pixel-level image morphing for visualization. Includes automatic preprocessing with variable health checks, multiple scaling methods (standardized, range, robust), greedy matching algorithms, and comprehensive balance diagnostics for assessing match quality using standardized differences and distribution comparisons.

License MIT + file LICENSE

Language en-US

Encoding UTF-8

Depends R (>= 4.1.0)

Imports Rcpp (>= 1.0.0), tibble (>= 3.0.0), dplyr (>= 1.0.0), rlang (>= 0.4.0), generics (>= 0.1.0), purrr (>= 0.3.0), parallel, parallelly (>= 1.30.0)

Suggests testthat (>= 3.0.0), htmlwidgets, e1071, R.utils, microbenchmark, withr, knitr, rmarkdown, bench, future (>= 1.20.0), future.apply (>= 1.8.0), ggplot2, ggraph, tidygraph, magick, av, png, combinat, cobalt, MatchIt, marginaleffects, optmatch, clue, lpSolve

LinkingTo Rcpp

SystemRequirements C++17

LazyData true

VignetteBuilder knitr

URL https://gillescolling.com/couplr/, https://github.com/gcol33/couplr

BugReports https://github.com/gcol33/couplr/issues

Config/testthat/edition 3

Config/roxygen2/version 8.1.0

Config/Needs/coverage xml2

NeedsCompilation yes

Author Gilles Colling [aut, cre, cph] (ORCID: <https://orcid.org/0000-0003-3070-6066>)

Maintainer Gilles Colling <gilles.colling051@gmail.com>

Repository CRAN

Date/Publication 2026-11-07 16:50:26 UTC

Contents

animated_methods . . . 3
assignment . . . 3
assignment_duals . . . 9
as_assignment_matrix . . . 11
as_matchit . . . 11
autoplot.sensitivity_analysis . . . 13
bal.tab.couplr_result . . . 15
balance_diagnostics . . . 15
balance_table . . . 18
bottleneck_assignment . . . 19
cardinality_match . . . 20
cem_match . . . 25
compute_distances . . . 26
diagnose_distance_matrix . . . 29
estimate_dense_matrix_mb . . . 30
estimate_dense_solve_mb . . . 30
example_costs . . . 31
example_df . . . 33
explain_dispatch . . . 34
full_match . . . 36
hospital_staff . . . 38
join_matched . . . 41
lap_animate . . . 44
lap_solve . . . 45
lap_solvers . . . 47
lap_solve_batch . . . 48
lap_solve_kbest . . . 50
lap_solve_line_metric . . . 52

matchmaker 53
match_couples 55
match_data 59
match_path 61
pixel_morph 63
pixel_morph_animate 65
plot.matching_result 68
preprocess_matching_vars 69
ps_match 71
result_accessors 72
sinkhorn 73
sinkhorn_to_assignment 76
solver_status_values 77
subclass_match 78
update_constraints 79
verify_assignment 80
verify_flow 84

Index **88**

`animated_methods` *List Methods That Support Animation*

Description

Returns the character vector of method strings for which a trace function has been registered. Use any of these with `lap_animate()`.

Usage

```
animated_methods()
```

Value

Character vector of registered method names.

`assignment` *Solve a Linear Assignment Problem*

Description

Solve the linear assignment problem (minimum- or maximum-cost matching) using several algorithms. Forbidden edges can be marked as `NA` or `Inf`.

Usage

```
assignment(
  cost,
  maximize = FALSE,
  method = "auto",
  auction_eps = NULL,
  eps = NULL,
  memory_mode = "auto",
  certify = NULL,
  cardinality = c("complete", "maximum", "fixed"),
  n_matches = NULL,
  unmatched_penalty = NULL
)

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

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

is_lap_solve_result(x)

get_total_cost(x)

get_method_used(x)
```

Arguments

`cost`	Numeric matrix; rows = tasks, columns = agents. `NA` or `Inf` entries are treated as forbidden assignments.
`maximize`	Logical; if `TRUE`, maximizes the total cost instead of minimizing.
`method`	Character string indicating the algorithm to use. Options:

General-purpose solvers:

- `"auto"` — Automatic selection based on problem characteristics (default)
- `"jv"` — 'Jonker-Volgenant', fast general-purpose O(n^3) with warm-start
- `"hungarian"` — Classic 'Hungarian' (shortest augmenting path) O(n^3)
- `"munkres"` — Matrix-form 'Kuhn-Munkres' O(n^4), reference implementation

Auction-based solvers:

- `"auction"` — 'Bertsekas' auction with adaptive epsilon
- `"auction_gs"` — 'Gauss-Seidel' variant, good for spatial structure
- `"auction_scaled"` — 'Epsilon-scaling' auction, bidding at a decreasing sequence of epsilon values. Not chosen by `"auto"`; a rule sending heavily tied costs to it was tested and did not meet its criterion, and it remains available by name for tied costs

Specialized solvers:

- "sap" — Shortest augmenting path over the shared flow model's network. "ssp" is accepted as a second spelling of this method and resolves to "sap".
- "sap_dense" — Shortest augmenting path with a linear scan in place of a heap, O(n * m^2), suited to a dense cost matrix
- "lapmod" — 'Jonker-Volgenant' over the finite entries alone, stored row by row (CSR). Written for sparse input; see below for why "auto" does not divert sparse matrices to it
- "hk01" — 'Hopcroft-Karp' for binary (0/1) or constant costs. Constant costs make every perfect matching optimal. On a {0,1} matrix the search runs over the zero-cost edges alone, where a perfect matching totals zero and is therefore optimal; if none exists the problem is passed to the weighted solver on the original costs.
- "ssap_bucket" — 'Dial' algorithm for integer costs
- "bruteforce" — Exact enumeration for tiny problems (n <= 8)

Advanced solvers:

- "csa" — 'Goldberg-Kennedy' cost-scaling assignment (CSA-Q): epsilon starts at the span of the costs and is divided by 10 each refine, rows are discharged from a stack by double-push, and each row keeps its three cheapest arcs between scans (the fourth-best heuristic). Real-valued costs are read as supplied; the final epsilon-optimal assignment is repaired to an optimal one
- "gabow_tarjan" — 'Gabow-Tarjan' bit-scaling with complementary slackness. On a graph of V vertices and E edges the bound is O(sqrt(V) * E * log(V * C)), which for an n by n cost matrix is O(n^2.5 * log(n * C)). Its optimality bound holds for a matching that saturates both sides, so a rectangular problem gains a dummy side of zero cost. The dummies are copies of one node and are carried as a single unit holding as many partners as there are dummies, so the problem is solved at its own n by m shape.
- "cycle_cancel" — Cycle-canceling with 'Karp' algorithm
- "csflow" — Successive shortest paths with 'Johnson' potentials
- "network_simplex" — 'Network simplex' with spanning tree representation
- "push_relabel" — 'Goldberg-Tarjan' cost-scaling push-relabel
- "ramshaw_tarjan" — 'Ramshaw-Tarjan', optimized for rectangular matrices (n != m)

One-dimensional problems have their own entry point, lap_solve_line_metric(), which takes two point vectors rather than a cost matrix and runs in O(n log n).

Under "auto", a single pass over cost supplies the facts the following rules need, and the first matching rule wins:

1. at most 8 rows and 8 columns: "bruteforce", exact and faster than setting up a general solver;
2. finite entries all equal, or all either 0 or 1: "hk01", which exploits the absence of a real cost scale;
3. everything else: "jv".

Sparsity and aspect ratio used to divert the choice to "lapmod" and "sap". Neither beat "jv" on the regime grid, at any of the four admissibility densities or three aspect ratios measured, so both now fall through to it; each stays reachable by naming it.

Naming a method skips the pass. Rectangular problems are transposed internally so the solver always sees at least as many columns as rows, and the assignment is mapped back afterwards.

auction_eps — Optional numeric epsilon for the 'Auction'/'Auction-GS' methods. If NULL, an internal default (e.g., 1e-9) is used.

eps — Deprecated. Use auction_eps. If provided and auction_eps is NULL, its value is used for auction_eps.

memory_mode — One of "auto" (default), "dense", "lazy" or "implicit". cost is already a materialized matrix by the time it reaches assignment(), so "auto" here is diagnostic only: it warns if the matrix is large relative to free system RAM (nothing else can be done post-hoc once the matrix already exists – build it via compute_distances(memory_mode = ...) instead to avoid materializing it in the first place). "lazy" and "implicit" describe how a cost source is read rather than how a matrix is stored, so they apply to a lazy cost specification; "implicit" also accepts a matrix, where it solves the same problem by generating the pairs it needs and saves nothing, which is what makes it a check on the complete solve rather than a faster one. Against "lazy" on a lazy cost specification, "implicit" leads from 5,000 units upward on the eight-covariate problem the benchmarks use, by 1.1x at 5,000 rising to 3.1x at 50,000, and loses below that where the loop's fixed costs are still visible. What it buys at every size is the certificate over the complete problem.

certify — Logical; whether to attach a checked assignment_certificate as certificate. NULL, the default, takes the path's own answer: TRUE under memory_mode = "implicit", where the certificate is what distinguishes the answer from an approximate one and the loop has already done most of the scan, and FALSE elsewhere. The check reads the duals the solve returned where it returned some (see u and v below), so it costs one pass over the admissible pairs; elsewhere it costs the solve verify_assignment() runs to get them. A solve that did not reach a complete optimal matching has nothing to certify and gets no certificate.

cardinality — How many pairs to produce.

- "complete" (default) — every row is matched; an input admitting no complete matching is an error.
- "maximum" — as many pairs as the admissible edges allow, and the cheapest total among matchings of that size.
- "fixed" — exactly n_matches pairs, chosen to minimize total cost.

All three are solved exactly by the same solver: the two non-complete modes append dummy columns priced so that the solver's own optimum is the requested objective.

n_matches — Integer; the number of pairs to produce. Required when cardinality = "fixed", and not accepted otherwise.

unmatched_penalty
: Numeric; the cost charged for leaving one row unmatched, under `cardinality = "maximum"`. Supplying it replaces the lexicographic objective with a single one: a pair costing more than the penalty is worth dropping. Left `NULL`, no pair is ever traded away for a cost saving.

x
: An assignment result object

...
: Additional arguments passed to `print()`. Currently ignored.

object
: A `lap_solve_result` object.

Details

`method = "auto"` selects an algorithm based on problem size and the costs:

- Very small (n <= 8 and m <= 8): `"bruteforce"` — exact enumeration
- Binary/constant costs: `"hk01"` — specialized for 0/1 costs
- Otherwise: `"jv"`

`explain_dispatch()` reports which rule fired and why. The other solvers are available by naming them explicitly.

Value

An object of class `lap_solve_result`, a list with elements:

- `match` — integer vector of length `min(nrow(cost), ncol(cost))` giving the assigned column for each row (0 if unassigned).
- `total_cost` — numeric scalar, the objective value.
- `status` — character scalar drawn from `solver_status_values()`, computed from what the solver terminated on. `"optimal"` means the solver reached its own optimality condition with every row matched; it is not a checked proof. Use `verify_assignment()` for that.
- `method_used` — character scalar, the algorithm actually used.
- `dispatch` — list recording how `method` was chosen: the rule that fired under `"auto"`, the condition that triggered it, and whether the method was named explicitly. See `explain_dispatch()`.
- `certificate` — an `assignment_certificate`, present when one was checked. See `certify`.
- `u`, `v` — row and column duals in the orientation of `cost`, present when the solver produced optimal duals for `cost`: `"jv"` (and so `"auto"` on every problem it sends there), `"hungarian"`, the lazy `"jv"` path, and `memory_mode = "implicit"`, where they are the duals the last restricted master produced. `verify_assignment()` reads them instead of solving again. Absent under `cardinality = "maximum"` and `"fixed"`, whose solver duals belong to the padded problem rather than to `cost`, and for `"gabow_tarjan"`, whose duals are those of its scaled integer instance.

Under `memory_mode = "implicit"` the result also carries `search`: the columns the first round gave each row (`seed_width`), the pairs the candidate set ended up holding (`candidate_edges`) out of `possible_edges`, the pairs a cost was computed for (`edges_evaluated`), the round count, and `rounds`, one row per round of what the master held, what priced out and what each step cost.

Integer conversion for bit-scaling

`"gabow_tarjan"` is a bit-scaling algorithm and runs on integer costs. The conversion is part of the method, so the rule it follows and what it claims afterwards are both properties of the solve rather than of the caller's preparation.

Finite costs are shifted so the smallest is zero, multiplied by a scale factor `s`, and rounded to the nearest integer. `s` is set so that `K * s * (max - min)` stays at or below `10^13`, where `K` is `n + 1` on a square problem and `2 * min(n, m) + 1` on a rectangular one. `K` is the separation the algorithm needs between the optimum and a 1-optimal matching, and the bound holds every scaled cost clear of the sentinel that marks a forbidden pair while keeping the path sums the solver forms inside 64-bit arithmetic. Forbidden pairs take the sentinel and take part in neither the range nor the conversion.

A matrix whose finite entries are each within `1e-9` of an integer is taken as an integer matrix: `s` is one, the shift is an integer, and every cost reaches the solver as the integer nearest to it. Where those entries are integers the conversion is exact and the optimum is the optimum of the instance as supplied; where they are merely near one, each cost moves by at most `1e-9`, so the matching returned costs at most `2 * min(n, m) * 1e-9` more than that optimum. Such a matrix is refused when its range exceeds `1.25 * 10^14 / K`, since a scale of one is the only one available and no choice brings the instance inside the bound; the error names the limit and points at `"jv"` or `"auction"`.

Costs that are not integers are solved on the rounded instance. Rounding moves each cost by at most `1 / (2 * s)`, so the matching returned costs at most `min(n, m) * K * (max - min) / 10^13` more than the optimum of the matrix as supplied. On a square problem that is about `n^2 * (max - min) / 10^13`, which is `10^-7` of the range at `n = 1000` and `10^-5` of it at `n = 10000`. Duals are divided by `s` and shifted back, so they belong to the original matrix; whether the matching is optimal for that matrix and not only for the rounded one is a question `verify_assignment()` answers rather than one this bound settles.

See Also

- `lap_solve()` — Tidy interface returning tibbles
- `lap_solve_kbest()` — Find k-best assignments ('Murty' algorithm)
- `assignment_duals()` — Extract dual variables for sensitivity analysis
- `bottleneck_assignment()` — Minimize maximum edge cost (minimax)
- `sinkhorn()` — Entropy-regularized optimal transport

Examples

```
cost <- matrix(c(4,2,5, 3,3,6, 7,5,4), nrow = 3, byrow = TRUE)
res  <- assignment(cost)
res$match; res$total_cost
```

`assignment_duals` *Solve an Assignment Problem and Return Dual Potentials*

Description

Solves the linear assignment problem and returns dual potentials (u, v) in addition to the optimal matching. The dual variables provide an optimality certificate and enable sensitivity analysis.

Usage

```
assignment_duals(cost, maximize = FALSE, certify = FALSE)

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

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

## S3 method for class 'assignment_duals_result'
as.data.frame(x, row.names = NULL, optional = FALSE, ...)
```

Arguments

`cost`	Numeric matrix; rows = tasks, columns = agents. `NA` or `Inf` entries are treated as forbidden assignments. A lazy cost specification from `compute_distances()` is also accepted, and is solved without materializing the matrix.
`maximize`	Logical; if `TRUE`, maximizes the total cost instead of minimizing.
`certify`	Logical; if `TRUE`, the duals are checked against `cost` with `verify_assignment()` and the resulting `assignment_certificate` is attached as `certificate`. The check is one pass over the admissible pairs and reuses the duals computed here, so it costs no second solve.
`x`	An `assignment_duals_result`.
`...`	Ignored.
`object`	An `assignment_duals_result`.
`row.names, optional`	Ignored.

Details

The dual variables satisfy the complementary slackness conditions:

- For minimization: `u[i] + v[j] <= cost[i,j]` for all (i,j)
- For any assigned pair (i,j): `u[i] + v[j] = cost[i,j]`

This implies that `sum(u) + sum(v) = total_cost` (strong duality).

Applications of dual variables:

- **Optimality verification**: Check that duals satisfy constraints
- **Sensitivity analysis**: Reduced cost `c[i,j] - u[i] - v[j]` shows how much an edge cost must decrease before it enters the solution
- **Pricing in column generation**: Use duals to price new columns
- **Warm starting**: Reuse duals when costs change slightly

Value

A list with class `"assignment_duals_result"` containing:

- `match` - integer vector of column assignments (1-based)
- `total_cost` - optimal objective value
- `u` - numeric vector of row dual variables (length n)
- `v` - numeric vector of column dual variables (length m)
- `status` - character, e.g. "optimal"
- `certificate` - an `assignment_certificate`, present only under `certify = TRUE`

See Also

`assignment()` for standard assignment without duals, `verify_assignment()` for the check `certify = TRUE` runs

Examples

```
cost <- matrix(c(4, 2, 5, 3, 3, 6, 7, 5, 4), nrow = 3, byrow = TRUE)
result <- assignment_duals(cost)

# Check optimality: u + v should equal cost for assigned pairs
for (i in 1:3) {
  j <- result$match[i]
  cat(sprintf("Row %d -> Col %d: u + v = %.2f, cost = %.2f\n",
              i, j, result$u[i] + result$v[j], cost[i, j]))
}

# Verify strong duality
cat("sum(u) + sum(v) =", sum(result$u) + sum(result$v), "\n")
cat("total_cost =", result$total_cost, "\n")

# Reduced costs (how much must cost decrease to enter solution)
reduced <- outer(result$u, result$v, "+")
reduced_cost <- cost - reduced
print(round(reduced_cost, 2))
```

as_assignment_matrix *Convert an Assignment Result to a Binary Matrix*

Description

Turns a list or tidy assignment result into a 0/1 assignment matrix. Original dimensions and identifiers are retained where recorded.

Usage

```
as_assignment_matrix(x, n_sources = NULL, n_targets = NULL)
```

Arguments

x	An assignment result object of class lap_solve_result
n_sources	Number of source nodes, optional
n_targets	Number of target nodes, optional

Value

Integer matrix with 0 and 1 entries

as_matchit *Convert a Matching Result to a matchit Object*

Description

Constructs a matchit-class S3 object from a couplr result, enabling use with any function that accepts **MatchIt** objects (e.g., **cobalt**, **marginaleffects**).

Usage

```
as_matchit(
  result,
  left,
  right,
  formula = NULL,
  left_id = "id",
  right_id = "id",
  estimand = NULL,
  ...
)
```

Arguments

result	A couplr result object (matching_result, full_matching_result, cem_result, or subclass_result)
left	Data frame of left (treated) units
right	Data frame of right (control) units
formula	Optional formula used for matching. If not provided, a default formula is constructed from result$info$vars.
left_id	Name of ID column in left (default: "id")
right_id	Name of ID column in right (default: "id")
estimand	Target estimand stamped on the matchit object: one of "ATT", "ATC" or "ATE". NULL (default) reads it from the design, which every couplr front door records as info$estimand. MatchIt and marginaleffects read this field to pick the target population and the weighting of the effect estimate, so give it explicitly whenever the design does not determine it – in particular when left holds the controls, which makes the design's left-focal weighting an ATC.
...	Additional arguments (ignored)

Value

An S3 object of class "matchit" with fields:

match.matrix Match matrix (treated x controls)

treat Named treatment vector (1/0)

weights Matching weights

X Covariate matrix

call Original call

info Metadata from couplr

Examples

```
if (requireNamespace("MatchIt", quietly = TRUE)) {
  set.seed(1)
  left <- data.frame(id = 1:5, age = c(25, 35, 45, 55, 65))
  right <- data.frame(id = 6:15, age = runif(10, 20, 70))
  result <- match_couples(left, right, vars = "age")
  mi <- as_matchit(result, left, right)
  mi
  if (requireNamespace("cobalt", quietly = TRUE)) cobalt::bal.tab(mi)
}
```

`autoplot.sensitivity_analysis` *Rosenbaum Sensitivity Analysis*

Description

Assesses how sensitive a matched comparison is to hidden bias using Rosenbaum bounds on the Wilcoxon signed-rank statistic.

Usage

```
## S3 method for class 'sensitivity_analysis'
autoplot(object, alpha = 0.05, ...)

sensitivity_analysis(
  result,
  left,
  right,
  outcome_var,
  gamma = seq(1, 3, by = 0.25),
  alternative = c("greater", "less", "two.sided"),
  left_id = "id",
  right_id = "id"
)

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

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

## S3 method for class 'sensitivity_analysis'
as.data.frame(x, row.names = NULL, optional = FALSE, ...)

## S3 method for class 'sensitivity_analysis'
plot(x, alpha = 0.05, ...)
```

Arguments

`object`	A `sensitivity_analysis` object.
`alpha`	Significance level (default: 0.05)
`...`	Additional arguments passed to plot
`result`	A matching_result object from `match_couples()`
`left`	Original left (treated) dataset
`right`	Original right (control) dataset

`outcome_var`	Name of the outcome column in `left` and `right`
`gamma`	Numeric vector of sensitivity parameters (default: seq(1, 3, by = 0.25)). Gamma = 1 means no hidden bias.
`alternative`	Direction of the test: "greater" (default), "less", or "two.sided"
`left_id`	Name of ID column in left (default: "id")
`right_id`	Name of ID column in right (default: "id")
`x`	A sensitivity_analysis object
`row.names, optional`	Ignored.

Details

Rosenbaum (2002, Chapter 4) bounds quantify how much hidden bias (an unobserved confounder) would be needed to explain away the observed treatment effect. The sensitivity parameter Gamma represents the maximum ratio of treatment odds between two matched units:

- Gamma = 1: No hidden bias (standard Wilcoxon test)
- Gamma = 2: One unit could be twice as likely to receive treatment due to an unobserved factor

The function computes upper and lower bounds on the p-value of the Wilcoxon signed-rank test under each level of hidden bias. A finding is "insensitive to bias" if p_upper remains below 0.05 even at large Gamma.

Value

An S3 object of class `sensitivity_analysis` containing:

results Tibble with columns: gamma, t_stat, p_upper, p_lower

n_pairs Number of matched pairs analyzed

critical_gamma Smallest gamma at which p_upper > 0.05

alternative Direction of test

References

Rosenbaum, P.R. (2002). Observational Studies, 2nd edition. Springer.

Examples

```
set.seed(42)
left <- data.frame(id = 1:20, x = rnorm(20), outcome = rnorm(20, 1, 1))
right <- data.frame(id = 21:40, x = rnorm(20), outcome = rnorm(20, 0, 1))
result <- match_couples(left, right, vars = "x")
sens <- sensitivity_analysis(result, left, right, outcome_var = "outcome")
print(sens)
```

bal.tab.couplr_result *Balance Tables from cobalt for Matching Results*

Description

S3 methods enabling `cobalt::bal.tab()` on couplr result objects. Requires the **cobalt** package to be installed.

Usage

```
## S3 method for class 'couplr_result'
bal.tab(x, left, right, ...)

## S3 method for class 'subclass_result'
bal.tab(x, data = NULL, ...)
```

Arguments

`x`	A couplr result object
`left`	Data frame of left (treated) units
`right`	Data frame of right (control) units
`...`	Additional arguments. Arguments named in `as_matchit()`'s signature go to the conversion; the rest go to `cobalt::bal.tab()`.
`data`	Data frame used for subclassification (for subclass_result only)

Details

These methods convert couplr results to the format cobalt expects (a matchit-class object) and then delegate to cobalt's own `bal.tab.matchit()` method. The **cobalt** package must be installed but is not required for couplr to function.

Value

A cobalt balance table object

balance_diagnostics *Balance Diagnostics for Matched Samples*

Description

Computes comprehensive balance statistics comparing the distribution of matching variables between left and right units in the matched sample.

Usage

```
balance_diagnostics(result, ...)

## S3 method for class 'matching_result'
balance_diagnostics(
  result,
  left,
  right,
  vars = NULL,
  left_id = "id",
  right_id = "id",
  ...
)

## S3 method for class 'full_matching_result'
balance_diagnostics(
  result,
  left,
  right,
  vars = NULL,
  left_id = "id",
  right_id = "id",
  ...
)

## S3 method for class 'cem_result'
balance_diagnostics(
  result,
  left,
  right,
  vars = NULL,
  left_id = "id",
  right_id = "id",
  ...
)

## S3 method for class 'subclass_result'
balance_diagnostics(result, data = NULL, vars = NULL, ...)

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

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

## S3 method for class 'balance_diagnostics'
plot(x, type = c("love", "histogram", "variance"), threshold = 0.1, ...)
```

```
## S3 method for class 'balance_diagnostics'
autoplot(
  object,
  type = c("love", "histogram", "variance"),
  threshold = 0.1,
  ...
)
```

Arguments

result	A matching result object from match_couples(), full_match(), cem_match(), or subclass_match()
...	Additional arguments (ignored)
left	Data frame of left units
right	Data frame of right units
vars	Character vector of variable names to check balance for. Defaults to the variables used in matching (if available in result).
left_id	Character, name of ID column in left data (default: "id")
right_id	Character, name of ID column in right data (default: "id")
data	Data frame used for subclassification (when result is a subclass_result)
x, object	A balance_diagnostics object.
type	Type of plot: "love" (default), "histogram", or "variance"
threshold	Threshold for standardized differences (default: 0.1)

Details

This function computes several balance metrics:

Standardized Difference: The difference in means divided by the pooled standard deviation. Values less than 0.1 indicate excellent balance, 0.1-0.25 good balance.

Variance Ratio: The ratio of variances (left/right). Values close to 1 are ideal; the conventional acceptable range is 0.5 to 2. For weighted estimators (full matching, CEM, subclassification) the means and variances are computed using the stratum weights.

KS Statistic: Kolmogorov-Smirnov test statistic comparing distributions. Lower values indicate more similar distributions.

Overall Metrics include mean absolute standardized difference across all variables, proportion of variables with large imbalance (|std diff| > 0.25), and maximum standardized difference.

Value

An S3 object of class balance_diagnostics containing:

var_stats Tibble with per-variable balance statistics

overall List with overall balance metrics

pairs Tibble of matched pairs with variables

n_matched Number of matched pairs

n_unmatched_left Number of unmatched left units

n_unmatched_right Number of unmatched right units

method Matching method used

has_blocks Whether blocking was used

block_stats Per-block statistics (if blocking used)

Examples

```
# Create sample data
set.seed(123)
left <- data.frame(
  id = 1:10,
  age = rnorm(10, 45, 10),
  income = rnorm(10, 50000, 15000)
)
right <- data.frame(
  id = 11:30,
  age = rnorm(20, 47, 10),
  income = rnorm(20, 52000, 15000)
)

# Match
result <- match_couples(left, right, vars = c("age", "income"))

# Get balance diagnostics
balance <- balance_diagnostics(result, left, right, vars = c("age", "income"))
print(balance)

# Get balance table
balance_table(balance)
```

---

balance_table *Balance Table of a Balance Diagnostics Object*

Description

Formats balance diagnostics into a clean table for display or export.

Usage

```
balance_table(balance, digits = 3)
```

Arguments

balance	A balance_diagnostics object from balance_diagnostics()
digits	Number of decimal places for rounding (default: 3)

Value

A tibble with formatted balance statistics

`bottleneck_assignment` *Solve the Bottleneck Assignment Problem*

Description

Finds an assignment that minimizes (or maximizes) the maximum edge cost in a perfect matching. Unlike standard LAP which minimizes the sum of costs, BAP minimizes the maximum (bottleneck) cost.

Usage

```
bottleneck_assignment(cost, maximize = FALSE)

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

Arguments

`cost`	Numeric matrix; rows = tasks, columns = agents. `NA` or `Inf` entries are treated as forbidden assignments.
`maximize`	Logical; if `TRUE`, maximizes the minimum edge cost instead of minimizing the maximum (maximin objective). Default is `FALSE` (minimax).
`object`	A result returned by this function.
`...`	Ignored by the summary method.

Details

The Bottleneck Assignment Problem (BAP) is a variant of the Linear Assignment Problem where instead of minimizing the sum of assignment costs, we minimize the maximum cost among all assignments (minimax objective).

Algorithm: Uses binary search on the sorted unique costs combined with 'Hopcroft-Karp' bipartite matching to find the minimum threshold that allows a perfect matching.

Complexity: O(E * sqrt(V) * log(unique costs)) where E = edges, V = vertices.

Applications:

- Task scheduling with deadline constraints (minimize latest completion)
- Resource allocation (minimize maximum load/distance)
- Network routing (minimize maximum link utilization)
- Fair division problems (minimize maximum disparity)

Value

A list with class "bottleneck_result" containing:

- match - integer vector of length nrow(cost) giving the assigned column for each row (1-based indexing)
- bottleneck - numeric scalar, the bottleneck (max/min edge) value
- status - character scalar, e.g. "optimal"

summary() returns a summary.assignment_result with the bottleneck value, status, and complete row-to-column assignment table.

See Also

assignment() for standard LAP (sum objective), lap_solve() for tidy LAP interface

Examples

```
# Simple example: minimize max cost
cost <- matrix(c(1, 5, 3,
                 2, 4, 6,
                 7, 1, 2), nrow = 3, byrow = TRUE)
result <- bottleneck_assignment(cost)
result$bottleneck  # Maximum edge cost in optimal assignment

# Maximize minimum (fair allocation)
profits <- matrix(c(10, 5, 8,
                    6, 12, 4,
                    3, 7, 11), nrow = 3, byrow = TRUE)
result <- bottleneck_assignment(profits, maximize = TRUE)
result$bottleneck  # Minimum profit among all assignments

# With forbidden assignments
cost <- matrix(c(1, NA, 3,
                 2, 4, Inf,
                 5, 1, 2), nrow = 3, byrow = TRUE)
result <- bottleneck_assignment(cost)
```

---

cardinality_match *Cardinality Matching*

Description

Maximizes the number of matched pairs subject to balance constraints, and reports how far the matched sample sits from the largest one the constraints admit.

Usage

```
cardinality_match(
  left,
  right,
  vars,
  fine = NULL,
  refined = NULL,
  refined_exact = 1L,
  moments = NULL,
  max_std_diff = Inf,
  engine = c("auto", "flow", "branch_bound", "heuristic"),
  distance = "euclidean",
  weights = NULL,
  scale = FALSE,
  auto_scale = FALSE,
  sigma = NULL,
  max_distance = Inf,
  calipers = NULL,
  left_id = NULL,
  right_id = NULL,
  time_limit = 30,
  node_limit = 500L,
  method = "auto",
  max_iter = 100L,
  batch_fraction = 0.1,
  memory_mode = "auto"
)
```

Arguments

left	Data frame of "left" units, one row each.
right	Data frame of "right" units, one row each.
vars	Character vector of matching variable names.
fine	Exact fine balance on one partition: a character vector of column names, cross-classified, or a one-sided formula. In a formula + and : both mean interaction, so ~ region + sex and ~ region:sex both ask for balance on every region-by-sex cell. fine = X is the same request as refined = list(X), refined_exact = 1.
refined	A nested hierarchy of groupings, coarsest first: a list, each element a character vector or one-sided formula, or a character vector as shorthand for its own sequence of prefixes, so c("region", "site") means region, then region crossed with site. Each level must sit inside the one before it.
refined_exact	How many of the coarsest levels of refined are enforced exactly (default: 1). The remaining levels are balanced as closely as the exact levels allow.
moments	Moment constraints beyond max_std_diff: a named numeric of standardized-difference bounds, as in c(x = 0.05, y = 0.1), or a list of entries with fields var, stat ("std_diff" or "mean_diff"), transform, max and min. A variable bounded here must not also be bounded by max_std_diff.

`max_std_diff` Maximum absolute standardized difference on every variable in `vars` (default: `Inf`, stating no such constraint). A finite value states one moment row per variable and per direction, which is what sends a call to the search over unit selections; see Details.

`engine` Which solver answers the problem: `"flow"` for a single min-cost flow solve, available when no moment row is stated; `"branch_bound"` for the search over unit selections that moment rows need; `"auto"` (default) for whichever of the two the stated constraints call for; and `"heuristic"` for the pruning loop.

`distance` Distance metric (default: "euclidean").

`weights` Optional named vector of variable weights.

`scale` Scaling method (default: FALSE).

`auto_scale` If TRUE, check variable health and select scaling automatically (default: FALSE).

`sigma` Bandwidth for the distance metrics that take one.

`max_distance` Pairs farther apart than this are not available.

`calipers` Named vector of caliper widths, each on a column of both data frames: a matching variable or another one, such as a propensity score.

`left_id`, `right_id` Name of the id column on each side. When absent, ids come from an `id` column, from row names, or are synthesized.

`time_limit` Seconds the search may run (default: 30). A search that stops on the clock returns a result that depends on the speed of the machine. Set `time_limit = Inf` and bound the search with `node_limit` to control the amount of search independently of machine speed. Floating-point ties can change the search order across platforms.

`node_limit` Branch-and-bound nodes the search may open, counted over all of its stages (default: 500).

`method` LAP solver method for the heuristic's initial match (default: "auto"). Reaches `engine = "heuristic"` only.

`max_iter` Maximum pruning iterations (default: 100). Reaches `engine = "heuristic"` only.

`batch_fraction` Fraction of the remaining pairs the heuristic deletes per iteration (default: 0.1). Reaches `engine = "heuristic"` only.

`memory_mode` One of "auto" (default), "dense" or "implicit". "implicit" solves the flow and branch and bound engines without building the pair set: every network solve is taken over the pairs generated so far, the pairs it omits are priced against its potentials, and the ones pricing below zero are added and the solve repeated, so each solve is optimal over every pair. The distance range the tier weights are built on is read in one pass over the pairs. The result carries a `search` record. "auto" never selects it, and the heuristic engine does not take it.

Details

Cardinality matching (Zubizarreta 2012) asks for the largest matched sample that satisfies stated balance constraints, rather than the closest one. The constraints come in two kinds, and which kinds are stated decides how the problem is solved.

Fine and refined covariate balance are representable in the matching network itself (Rosenbaum, Ross and Silber 2007; Pimentel, Kelz, Silber and Rosenbaum 2015): every category becomes a node, and a single min-cost flow solve returns the largest balanced sample together with a dual certificate of its optimality, at polynomial cost. `max_std_diff` defaults to `Inf`, stating no moment constraint, so a call asking for fine or refined balance alone takes this path and comes back certified:

```
cardinality_match(left, right, vars, fine = "region")
```

Linear moment constraints, bounds on a standardized or mean difference, do not fit in the network. Summed over the matched pairs, such a bound is a linear constraint on which units are matched, never on how they are paired, so `engine = "branch_bound"` searches over unit selections: an integer programme with one 0/1 variable per unit and a few rows, solved by branch and bound on its linear relaxation. A finite `max_std_diff` states one such row per variable and per direction, so it is what moves a call onto this path.

The objective is lexicographic, and the search settles it one stage at a time. It first finds the most matched units the constraints admit, then the least imbalance at each refined level, coarsest first, and last the least total distance. A pair is never added at the cost of a match and never dropped to shorten one. The distance stage reads the cheapest pairing of a selection off the balance network and adds a cut from the potentials of that solve (Benders decomposition), and when calipers or `max_distance` forbid pairs, a selection whose units cannot all be paired adds a Hall-condition row instead.

The report says how the search ended. `certified` is `TRUE` only when every stage closed, and `gap` says how many matched units separate the answer from the bound when the first stage did not. Every stopping path returns a matched set that satisfies every stated constraint, together with a bound valid for the whole problem. When the distance stage stops on the budget, the matched sample is a largest balanced one, with `gap` zero, whose total distance is the smallest the search reached.

The certificate carries no tolerance. Each bound is a weak-duality bound read from the relaxation's duals in exact arithmetic or with a rigorous rounding margin, so it holds whatever the simplex returned, and every selection is checked against the constraints exactly before it is accepted.

How long the search runs depends on how far the relaxation sits from the best integer selection. With continuous covariates the relaxation's bound on the number of matched units is usually within one unit of the optimum, and the search closes at its first node when it finds a selection that reaches it. When the optimum lies one unit below that bound, proving it takes many more nodes, and the report shows the incumbent beside a bound one unit above it. Set `node_limit` and `time_limit` to a budget you are willing to spend, and read `stopped_on` and `gap` to see what the budget bought.

`engine = "heuristic"` runs a pruning loop instead: a full match, then repeated deletion of the pairs carrying the worst variable's imbalance. Its properties follow from the loop. It starts from a full match and deletes pairs, and never re-adds one, so it cannot recover from an over-aggressive deletion. It never maximizes cardinality, it only shrinks a starting match. It considers one variable per iteration, the current worst, so balance on the others is incidental. It computes no bound, so it cannot say how far its answer sits from the largest balanced sample; `best_possible` and `gap` are `NA` and `certified` is `FALSE`.

Value

A `matching_result` object. Beyond the fields every matching carries it holds:

- `cardinality` - the report: `n_matched`, `best_possible`, `gap`, `gap_fraction`, `certified`, `stopped_on`, `n_nodes`, and the state of every stated constraint. See `print.cardinality_report()`.

- `certificate` - present when the search certified optimality, with `arithmetic = "exact"`: the bounds, the comparisons between them and the matched sample's objective and constraint values it rests on are all decided exactly.
- `potentials` - the dual potentials of the network solve the matched sample came from, a list with elements `left` and `right` holding one value per unit, named by id, in distance terms: the reduced cost of a pair is its distance less the two, with the tier weights already folded in. Under `engine = "flow"` that solve is the whole problem, and the potentials certify the matched sample as its optimum. Under branch and bound it is the solve that pairs the selected units, and the potentials certify that pairing as the cheapest one for those units; the bound on the whole problem is `cardinality$best_possible`. Present when that solve was certified.
- `status` - `"optimal"`, `"iteration_limit"`, or `"heuristic"`.
- `info$engine` - the engine that answered.

`info$pruning_iterations` and `info$pairs_removed` are carried by `engine = "heuristic"`.

References

Zubizarreta, J. R. (2012). Using mixed integer programming for matching in an observational study of kidney failure after surgery. *Journal of the American Statistical Association*, 107(500), 1360-1371.

Rosenbaum, P. R., Ross, R. N. and Silber, J. H. (2007). Minimum distance matched sampling with fine balance in an observational study of treatment for ovarian cancer. *Journal of the American Statistical Association*, 102(477), 75-83.

Pimentel, S. D., Kelz, R. R., Silber, J. H. and Rosenbaum, P. R. (2015). Large, sparse optimal matching with refined covariate balance in an observational study of the health outcomes produced by new surgeons. *Journal of the American Statistical Association*, 110(510), 515-527.

See Also

`match_couples()` for distance-minimizing matching, `balance_diagnostics()` for reading balance off any matched sample.

Examples

```
set.seed(42)
left <- data.frame(id = 1:20, x = rnorm(20), y = rnorm(20),
                   region = rep(c("A", "B"), length.out = 20))
right <- data.frame(id = 21:50, x = rnorm(30, 0.5), y = rnorm(30, 0.3),
                    region = rep(c("A", "B"), length.out = 30))

# Exact fine balance on region, no moment constraint: one flow solve,
# answered with a certificate.
fit <- cardinality_match(left, right, vars = c("x", "y"),
                         fine = "region")
fit$cardinality

# A standardized-difference bound as well: the same match, searched over
# unit selections under a small node budget.
bb <- cardinality_match(left, right, vars = c("x", "y"),
```

```
                        fine = "region", max_std_diff = 0.1,
                        node_limit = 25L)
bb$cardinality
```

`cem_match` *Coarsened Exact Matching*

Description

Coarsens continuous variables into bins, then performs exact matching on the coarsened values. Units in strata containing both left and right units are kept; others are pruned. Matched units receive weights inversely proportional to stratum sizes to maintain balance.

Usage

```
cem_match(
  left,
  right,
  vars,
  cutpoints = NULL,
  n_bins = "sturges",
  grouping = NULL,
  keep = "all",
  left_id = "id",
  right_id = "id"
)
```

Arguments

`left`	Data frame of left (treated) units
`right`	Data frame of right (control) units
`vars`	Character vector of variable names to coarsen and match on
`cutpoints`	Named list of break vectors per variable. If NULL, automatic binning is used.
`n_bins`	Binning method when `cutpoints` is NULL: `"sturges"` (default), `"fd"` (Freedman-Diaconis), `"scott"`, or an integer specifying the number of bins for all variables.
`grouping`	Character vector of variable names to match exactly (without coarsening). These are typically categorical variables.
`keep`	Which units to return: `"all"` (default) returns all units with weight 0 for unmatched, `"matched"` drops unmatched units.
`left_id`	Name of ID column in left (default: `"id"`)
`right_id`	Name of ID column in right (default: `"id"`)

Details

CEM algorithm:

1. Coarsen each numeric variable using `cut` with either user-specified breakpoints or automatic binning (Sturges, FD, or Scott rule)
2. Categorical variables in `grouping` are kept as-is
3. Create strata by concatenating all coarsened values
4. Drop strata with 0 left or 0 right units
5. Compute CEM weights: left units get weight 1, right units get weight `n_left_in_stratum / n_right_in_stratum` so that the total weight of right units in each stratum equals the number of left units

Value

An S3 object of class `c("cem_result", "couplr_result")` containing:

matched Tibble with columns `id`, `side`, `stratum`, `weight`

strata_summary Tibble with per-stratum counts

info List with `n_strata`, `n_matched_left`, `n_matched_right`, `n_pruned_left`, `n_pruned_right`, `method`, `vars`

Examples

```
set.seed(42)
left <- data.frame(
  id = 1:20, age = rnorm(20, 40, 10),
  income = rnorm(20, 50000, 10000)
)
right <- data.frame(
  id = 21:60, age = rnorm(40, 42, 10),
  income = rnorm(40, 52000, 10000)
)
result <- cem_match(left, right, vars = c("age", "income"))
print(result)
```

`compute_distances` *Compute and Cache a Distance Matrix*

Description

Precomputes a distance matrix between left and right datasets, allowing it to be reused across multiple matching operations with different constraints. This is particularly useful when exploring different matching parameters (max_distance, calipers, methods) without recomputing distances.

Usage

```
compute_distances(
  left,
  right,
  vars,
  distance = "euclidean",
  weights = NULL,
  scale = FALSE,
  auto_scale = FALSE,
  left_id = "id",
  right_id = "id",
  block_id = NULL,
  memory_mode = "auto"
)

is_distance_object(x)

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

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

## S3 method for class 'distance_object'
as.matrix(x, ...)
```

Arguments

left	Left dataset (data frame)
right	Right dataset (data frame)
vars	Character vector of variable names to use for distance computation
distance	Distance metric (default: "euclidean")
weights	Optional numeric vector of variable weights
scale	Scaling method: FALSE, "standardize", "range", or "robust"
auto_scale	Apply automatic preprocessing (default: FALSE)
left_id	Name of ID column in left (default: "id")
right_id	Name of ID column in right (default: "id")
block_id	Optional block ID column name for blocked matching
memory_mode	One of "auto" (default), "dense", or "lazy". "auto" warns (or, for a built-in distance metric, switches) when the dense matrix would consume a large fraction of free system RAM. "lazy" returns a `lazy_cost_spec` instead of a matrix (built-in distance metrics only); pass it straight to `match_couples()` the same way as a dense `cost_matrix`. "dense" skips the RAM check entirely.
x	A `distance_object`; any object for `is_distance_object()`.
...	Additional arguments (ignored)
object	A distance_object

Details

This function computes distances once and stores them in a reusable object. The resulting distance_object can be passed to `match_couples()` (optionally with `method = "greedy"`) instead of providing datasets and variables.

Benefits:

- **Performance**: Avoid recomputing distances when trying different constraints
- **Exploration**: Quickly test max_distance, calipers, or methods
- **Consistency**: Ensures same distances used across comparisons

With the default `memory_mode = "auto"/"dense"`, the stored `cost_matrix` is a plain dense matrix; couplr has no sparse matrix representation, so forbidden pairs (`Inf`) still occupy a cell rather than being dropped from storage. With `memory_mode = "lazy"`, `cost_matrix` is instead a `lazy_cost_spec` that recomputes distances on demand from the underlying feature data.

The distance_object stores the original datasets, allowing downstream functions like `join_matched()` to work seamlessly.

Value

An S3 object of class "distance_object" containing:

- `cost_matrix`: Numeric matrix of distances
- `left_ids`: Character vector of left IDs
- `right_ids`: Character vector of right IDs
- `block_id`: Block ID column name (if specified)
- `metadata`: List with computation details (vars, distance, scale, etc.)
- `original_left`: Original left dataset (for later joining)
- `original_right`: Original right dataset (for later joining)

Examples

```
# Compute distances once
left <- data.frame(id = 1:5, age = c(25, 30, 35, 40, 45), income = c(45, 52, 48, 61, 55) * 1000)
right <- data.frame(id = 6:10, age = c(24, 29, 36, 41, 44), income = c(46, 51, 47, 60, 54) * 1000)

dist_obj <- compute_distances(
  left, right,
  vars = c("age", "income"),
  scale = "standardize"
)

# Reuse for different matching strategies
result1 <- match_couples(dist_obj, max_distance = 0.5)
result2 <- match_couples(dist_obj, max_distance = 1.0)
result3 <- match_couples(dist_obj, method = "greedy", strategy = "sorted")

# All use the same precomputed distances
```

`diagnose_distance_matrix` *Diagnose a Distance Matrix*

Description

Comprehensive diagnostics for a distance matrix with actionable suggestions.

Usage

```
diagnose_distance_matrix(
  cost_matrix,
  left = NULL,
  right = NULL,
  vars = NULL,
  warn = TRUE
)

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

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

Arguments

`cost_matrix`	Numeric matrix of distances
`left`	Left dataset (for variable checking)
`right`	Right dataset (for variable checking)
`vars`	Variables used for matching
`warn`	If TRUE, issue warnings
`x`, `object`	A `distance_diagnostics` object.
`...`	Ignored.

Value

A `distance_diagnostics` list with the cost distribution, variable issues, problem variables, suggestions, and quality assessment. `summary()` returns these findings as `summary.distance_diagnostics`.

`estimate_dense_matrix_mb` *Estimate the Memory of a Dense Cost Matrix*

Description

What building the matrix costs a process: the 8 bytes of each cell, written in place by the distance code, and little besides. Building the matrix and stopping there has peaked at between 1.0 and 1.1 times the raw cell bytes from 2,000 to 20,000 units on the memory benchmark, so the default multiplier is above what has been measured rather than fitted to it. `n`/`m` are coerced to `double` before multiplying so the estimate itself can't overflow.

Usage

```
estimate_dense_matrix_mb(n, m, overhead_factor = 2)
```

Arguments

`n`, `m`	Problem dimensions.
`overhead_factor`	Multiplier on the raw cell bytes.

Details

This is the matrix, not the solve. `estimate_dense_solve_mb()` is what the memory guard reads; see there for why the two differ by more than a copy.

Value

Numeric scalar, the estimated footprint of the matrix in megabytes.

Examples

```
estimate_dense_matrix_mb(5000, 5000)
```

`estimate_dense_solve_mb` *Estimate the Peak Memory of a Dense Solve*

Description

The matrix is the allocation a caller can reason about and it is not what bounds a dense solve. The solver works on its own copy of the costs, with a byte per cell marking the forbidden pairs, beside the matrix R holds, and the peak is what decides whether the solve fits, so that is the quantity the guard compares against available RAM.

Usage

```
estimate_dense_solve_mb(n, m, solve_factor = 4)
```

Arguments

n, m	Problem dimensions.
solve_factor	Multiplier on the raw cell bytes.

Details

The multiplier is read off the measurement. On the memory benchmark – one fresh R session per arm, peak resident set taken from outside and read against an idle session that loaded the same packages – a dense one-to-one solve has peaked at between 2.2 and 3.0 times the raw matrix bytes from 2,000 to 20,000 units, the larger ratios at the small sizes, where fixed costs weigh most. The default of 4 sits above every measured peak, which is the direction a guard should err in: it exists to refuse a solve that will not fit, not to predict where the peak lands.

Value

Numeric scalar, the estimated peak footprint of a dense solve in megabytes.

Examples

```
estimate_dense_solve_mb(5000, 5000)
```

---

example_costs *Example Cost Matrices*

Description

Small example datasets for demonstrating couplr functionality across different assignment problem types: square, rectangular, sparse, and binary.

Usage

```
example_costs
```

Format

A list containing four example cost matrices:

simple_3x3 A 3x3 cost matrix with costs ranging from 2-7. Optimal assignment: row 1 -> col 2 (cost 2), row 2 -> col 1 (cost 3), row 3 -> col 3 (cost 4). Total optimal cost: 9.

rectangular_3x5 A 3x5 rectangular cost matrix demonstrating assignment when rows < columns. Each of 3 rows is assigned to one of 5 columns; 2 columns remain unassigned. Costs range 1-6.

sparse_with_na A 3x3 matrix with NA values indicating forbidden assignments. Use this to test algorithms' handling of constraints. Position (1,3), (2,2), and (3,1) are forbidden.

binary_costs A 3x3 matrix with binary (0/1) costs, suitable for testing the HK01 algorithm. Diagonal entries are 0 (preferred), off-diagonal entries are 1 (penalty).

Details

These matrices are designed to test different aspects of LAP solvers:

simple_3x3: Basic functionality test. Any correct solver should find total cost = 9.

rectangular_3x5: Tests handling of non-square problems. The optimal solution assigns all 3 rows with minimum total cost.

sparse_with_na: Tests constraint handling. Algorithms must avoid NA positions while finding an optimal assignment among valid entries.

binary_costs: Tests specialized binary cost algorithms. The optimal assignment uses all diagonal entries (total cost = 0).

See Also

lap_solve, example_df

Examples

```
# Simple 3x3 assignment
result <- lap_solve(example_costs$simple_3x3)
print(result)
# Optimal: sources 1,2,3 -> targets 2,1,3 with cost 9

# Rectangular problem (3 sources, 5 targets)
result <- lap_solve(example_costs$rectangular_3x5)
print(result)
# All 3 sources assigned; 2 targets unassigned

# Sparse problem with forbidden assignments
result <- lap_solve(example_costs$sparse_with_na)
print(result)
# Avoids NA positions

# Binary costs - test HK01 algorithm
result <- lap_solve(example_costs$binary_costs, method = "hk01")
print(result)
# Finds diagonal assignment (cost = 0)
```

`example_df` *Example Assignment Data Frame*

Description

A tidy data frame representation of assignment problems, suitable for use with grouped workflows and batch solving. Contains two independent 3x3 assignment problems in long format.

Usage

```
example_df
```

Format

A tibble with 18 rows and 4 columns:

sim Simulation/problem identifier. Integer with values 1 or 2, distinguishing two independent assignment problems. Use with `group_by(sim)` for grouped solving.

source Source node index. Integer 1-3 representing the row (source) in each 3x3 cost matrix.

target Target node index. Integer 1-3 representing the column (target) in each 3x3 cost matrix.

cost Cost of assigning source to target. Numeric values ranging from 1-7. Each source-target pair has exactly one cost entry.

Details

This dataset demonstrates couplr's data frame interface for LAP solving. The long format (one row per source-target pair) is converted internally to a cost matrix for solving.

Simulation 1: Costs from `example_costs$simple_3x3`

- Optimal assignment: (1->2, 2->1, 3->3)
- Total cost: 9

Simulation 2: Different cost structure

- Optimal assignment: (1->1, 2->3, 3->3) or equivalent
- Total cost: 4

See Also

`lap_solve`, `lap_solve_batch`, `example_costs`

Examples

```
library(dplyr)

# Solve both problems with grouped workflow
example_df |>
  group_by(sim) |>
  lap_solve(source, target, cost)

# Batch solving for efficiency
example_df |>
  group_by(sim) |>
  lap_solve_batch(source, target, cost)

# Inspect the data structure
example_df |>
  group_by(sim) |>
  summarise(
    n_pairs = n(),
    min_cost = min(cost),
    max_cost = max(cost)
  )
```

---

explain_dispatch *Explain the Automatic Solver Choice*

Description

`assignment(method = "auto")` picks a solver from a small set of rules evaluated against one pass over the cost matrix. This reports the decision without solving: which rule fired, the property that triggered it, the rules that were tested first and did not fire, and the internal representation the problem will be handed to the solver in.

Usage

```
explain_dispatch(
  cost,
  method = "auto",
  maximize = FALSE,
  cardinality = c("complete", "maximum", "fixed"),
  n_matches = NULL,
  unmatched_penalty = NULL
)

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

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

Arguments

cost	Numeric matrix, as passed to assignment(). NA or Inf entries are treated as forbidden.
method	Character scalar. "auto" (default) reports the automatic decision. Naming any other method reports that the choice was explicit and which rule "auto" would have picked instead.
maximize, cardinality, n_matches, unmatched_penalty	As in assignment(). A cardinality other than "complete" appends dummy columns before the solver sees the problem, which changes its shape and sparsity, so the rules are evaluated against the same matrix assignment() would hand the solver.
x	A dispatch_explanation object.
...	Ignored.
object	A dispatch_explanation object.

Details

The same rule table drives the dispatch itself, so the report cannot drift from the behaviour.

Value

An object of class dispatch_explanation, a list with elements:

- method - the solver that will run.
- explicit - TRUE when method was named rather than dispatched.
- rule, condition, reason - the rule that fired under "auto".
- considered - data frame of every rule tested, in order, with whether it fired.
- n, m, transposed - problem shape, and whether the solver sees the transpose. Rectangular problems are transposed internally so the solver always has at least as many columns as rows.
- probe - the single-pass summary the rules read.

print() gives the selected solver and reason. summary() returns a summary.dispatch_explanation with all decision and probe fields; printing it also lists the rules tested and the observed cost properties.

Invisibly returns x.

See Also

assignment()

Examples

```
explain_dispatch(matrix(runif(400), 20, 20))
explain_dispatch(matrix(sample(0:1, 400, TRUE), 20, 20))
```

`full_match` *Full Matching*

Description

Assigns every unit (left and right) to a matched group with variable ratios (1:k or k:1). Unlike 1:1 matching, full matching does not discard units, producing matched groups where each group contains at least one left and one right unit.

Usage

```
full_match(
  left,
  right,
  vars,
  distance = "euclidean",
  min_controls = 1,
  max_controls = Inf,
  caliper = NULL,
  caliper_sd = NULL,
  weights = NULL,
  scale = FALSE,
  auto_scale = FALSE,
  sigma = NULL,
  left_id = "id",
  right_id = "id",
  method = "optimal",
  memory_mode = "auto"
)
```

Arguments

`left`	Data frame of left (treated) units
`right`	Data frame of right (control) units
`vars`	Character vector of variable names to match on
`distance`	Distance metric: `"euclidean"` (default), `"mahalanobis"`, `"manhattan"`, or a custom function
`min_controls`	Minimum number of right units per group (default: 1)
`max_controls`	Maximum number of right units per group (default: Inf)
`caliper`	Maximum allowable distance for a match. Units with no eligible partner within the caliper are left unmatched.
`caliper_sd`	If not NULL, caliper is expressed in standard deviations of the pooled distance distribution rather than absolute units.
`weights`	Named numeric vector of variable weights

scale	Scaling method: FALSE (default), "robust", "standardize", or "range"
auto_scale	If TRUE, automatically preprocess and scale variables
sigma	Optional covariance matrix for Mahalanobis distance
left_id	Name of ID column in left (default: "id")
right_id	Name of ID column in right (default: "id")
method	Matching algorithm: "optimal" (default) uses min-cost max-flow to find the globally optimal group assignment minimizing total distance; "greedy" uses a fast two-pass heuristic.
memory_mode	One of "auto" (default), "dense" or "implicit". "auto" warns if the dense cost matrix would consume a large fraction of free system RAM. "dense" skips the RAM check entirely. "implicit" solves method = "optimal" without building the pair set: the flow is solved over a growing subset of pairs, the pairs it omits are priced against the flow's node potentials, and the subset grows until none prices below zero, so the groups are optimal over every pair; a search element records what that cost. Under "implicit" caliper_sd reads every pair's distance once to take its standard deviation, holding two running sums rather than the distances. "lazy" is not available here: a flow solved over every pair holds every pair.

Details

full_match() builds matched groups of variable size. Under the default min_controls = 1 it solves full matching in the sense of Hansen and Klopfer (2006): a group holds either one left unit and several right ones or one right unit and several left ones, and both shapes may appear in the same solution. max_controls bounds the many side there, whichever side that is.

A lower bound above one admits only the one-to-many shape, because a group built around a single right unit holds exactly one of them and cannot meet a lower bound of two. Every group is then one left unit with between min_controls and max_controls right ones. Both bounds count right units whichever side holds more, so an instance with too few right units to give every left unit that many is refused as "infeasible" rather than answered with groups counted the other way round.

Two algorithms are available:

Optimal (method = "optimal", default): Solves a min-cost max-flow problem that minimizes total distance across all group assignments simultaneously, with the optimum found via Dijkstra's algorithm with Johnson potentials. At min_controls = 1 the network is an edge cover over the admissible pairs, which is what lets a group be centred on either side; above one it is one centre per group and the centres are the left units.

Greedy (method = "greedy"): A fast two-pass heuristic:

1. Each left unit picks its nearest eligible right unit
2. Remaining right units are assigned to their nearest already-matched left unit, respecting max_controls

This is faster but does not guarantee globally optimal results.

Every left unit carries weight 1, and the right units of a group share a total weight equal to the number of left units in that group, so the two sides of a group weigh the same. A group holding one left unit and k right units therefore gives each right unit 1/k; a group holding k left units and one right unit gives that right unit k. "greedy" builds one left unit per group and produces only the first

shape. "optimal" at min_controls = 1 centres a group on whichever side is larger, so both arise; above one the centres are the left units and the first shape is the only one.

Value

An S3 object of class c("full_matching_result", "couplr_result") containing:

groups Tibble with columns group_id, id, side ("left"/"right"), and weight

status What the solver terminated on, one of "optimal" (every unit placed in a group meeting min_controls, at minimum total distance), "partial" (groups formed, some units left over, under a maximum-cardinality-then-minimum-cost objective), "infeasible" (no group meets the requested bounds), or "heuristic" (method = "greedy", which neither claims nor checks optimality). See solver_status_values.

info List with n_groups, n_left, n_right, n_unmatched_left, n_unmatched_right, method, vars

unmatched Left and right IDs that no group contains. Every unit is either a row of groups or an entry here.

potentials Node potentials from the flow solve, a list with elements left and right holding one value per unit. They are one representative of the dual, in the gauge the solver fixes. Present for method = "optimal" only.

certificate A flow_certificate from verify_flow, checking the solved flow and its potentials against the optimality conditions. status says what the solver terminated on; this says what was checked. Present for method = "optimal" only. Under memory_mode = "implicit" the check covers the pairs the flow was solved over, and omitted_proven_floor bounds the reduced cost of every pair it omitted; certified_optimal holds both together.

search Under memory_mode = "implicit": the seed width, the pairs generated (candidate_edges) against the pairs there are (possible_edges), the distances computed (edges_evaluated) and one row per round.

Examples

```
set.seed(42)
left <- data.frame(id = 1:5, age = c(25, 35, 45, 55, 65))
right <- data.frame(id = 6:20, age = runif(15, 20, 70))
result <- full_match(left, right, vars = "age")
print(result)
```

---

hospital_staff *Hospital Staff Example Data*

Description

A comprehensive example dataset for demonstrating couplr functionality across vignettes. Contains synthetic hospital staff scheduling data with nurses, shifts, costs, and preference scores suitable for assignment problems, as well as nurse characteristics for matching workflows.

Usage

```
hospital_staff
```

Format

A list containing eight related datasets:

basic_costs A 10x10 numeric cost matrix for assigning 10 nurses to 10 shifts. Values range from approximately 1-15, where lower values indicate better fit (less overtime, matches skills, respects preferences). Use with `lap_solve()` for basic assignment.

preferences A 10x10 numeric preference matrix on a 0-10 scale, where higher values indicate stronger nurse preference for a shift. Use with `lap_solve(..., maximize = TRUE)` to optimize preferences rather than minimize costs.

schedule_df A tibble with 100 rows (10 nurses x 10 shifts) in long format for data frame workflows:

- **nurse_id** Integer 1-10. Unique identifier for each nurse.
- **shift_id** Integer 1-10. Unique identifier for each shift.
- **cost** Numeric. Assignment cost (same values as basic_costs).
- **preference** Numeric 0-10. Nurse preference score.
- **skill_match** Integer 0/1. Binary indicator: 1 if nurse skills match shift requirements, 0 otherwise.

nurses A tibble with 10 rows describing nurse characteristics:

- **nurse_id** Integer 1-10. Links to schedule_df and basic_costs rows.
- **experience_years** Numeric 1-20. Years of nursing experience.
- **department** Character. Primary department: "ICU", "ER", "General", or "Pediatrics".
- **shift_preference** Character. Preferred shift type: "day", "evening", or "night".
- **certification_level** Integer 1-3. Certification level where 3 is highest (e.g., 1=RN, 2=BSN, 3=MSN).

shifts A tibble with 10 rows describing shift requirements:

- **shift_id** Integer 1-10. Links to schedule_df and basic_costs cols.
- **department** Character. Department needing coverage.
- **shift_type** Character. Shift type: "day", "evening", or "night".
- **min_experience** Numeric. Minimum years of experience required.
- **min_certification** Integer 1-3. Minimum certification level.

weekly_df A tibble for batch solving with 500 rows (5 days x 10 nurses x 10 shifts):

- **day** Character. Day of week: "Mon", "Tue", "Wed", "Thu", "Fri".
- **nurse_id** Integer 1-10. Nurse identifier.
- **shift_id** Integer 1-10. Shift identifier.
- **cost** Numeric. Daily assignment cost (varies by day).
- **preference** Numeric 0-10. Daily preference score.

Use with `group_by(day)` for solving each day's schedule.

nurses_extended A tibble with 200 nurses for matching examples, representing a treatment group (e.g., full-time nurses):

nurse_id Integer 1-200. Unique identifier.

age Numeric 22-65. Nurse age in years.

experience_years Numeric 0-40. Years of nursing experience.

hourly_rate Numeric 25-75. Hourly wage in dollars.

department Character. Primary department assignment.

certification_level Integer 1-3. Certification level.

is_fulltime Logical. TRUE for full-time status.

controls_extended A tibble with 300 potential control nurses (e.g., part-time or registry nurses) for matching. Same structure as nurses_extended. Designed to have systematic differences from nurses_extended (older, less experience on average) to demonstrate matching's ability to create comparable groups.

Details

This dataset is used throughout the couplr documentation to provide a consistent, realistic example that evolves in complexity. It supports three use cases: (1) basic LAP solving with cost matrices, (2) batch solving across multiple days, and (3) matching workflows comparing nurse groups.

The dataset is designed to demonstrate progressively complex scenarios:

Basic LAP (`vignette("couplr")`):

- `basic_costs`: Simple 10x10 assignment
- `preferences`: Maximization problem
- `schedule_df`: Data frame input, grouped workflows
- `weekly_df`: Batch solving across days

Algorithm comparison (`vignette("algorithms")`):

- Use `basic_costs` to compare algorithm behavior
- Modify with NA values for sparse scenarios

Matching workflows (`vignette("matching-workflows")`):

- `nurses_extended`: Treatment group (full-time nurses)
- `controls_extended`: Control pool (part-time/registry nurses)
- Match on age, experience, department for causal analysis

See Also

`lap_solve` for basic assignment solving, `lap_solve_batch` for batch solving, `match_couples` for matching workflows, `vignette("couplr")` for introductory tutorial

Examples

```
# Basic assignment: assign nurses to shifts minimizing cost
lap_solve(hospital_staff$basic_costs)

# Maximize preferences instead
lap_solve(hospital_staff$preferences, maximize = TRUE)

# Data frame workflow
library(dplyr)
hospital_staff$schedule_df |>
  lap_solve(nurse_id, shift_id, cost)

# Batch solve weekly schedule
hospital_staff$weekly_df |>
  group_by(day) |>
  lap_solve(nurse_id, shift_id, cost)

# Matching workflow: match full-time to part-time nurses
match_couples(
  left = hospital_staff$nurses_extended,
  right = hospital_staff$controls_extended,
  vars = c("age", "experience_years", "certification_level"),
  auto_scale = TRUE
)
```

---

join_matched *Join Matched Pairs with the Original Data*

Description

Creates an analysis-ready dataset by joining matched pairs with variables from the original left and right datasets. This eliminates the need for manual joins and provides a convenient format for downstream analysis.

Usage

```
join_matched(result, ...)

## S3 method for class 'matching_result'
join_matched(
  result,
  left,
  right,
  left_vars = NULL,
  right_vars = NULL,
  left_id = "id",
  right_id = "id",
```

```
  suffix = c("_left", "_right"),
  include_distance = TRUE,
  include_pair_id = TRUE,
  include_block_id = TRUE,
  ...
)

## S3 method for class 'full_matching_result'
join_matched(result, left, right, left_id = "id", right_id = "id", ...)

## S3 method for class 'cem_result'
join_matched(result, left, right, left_id = "id", right_id = "id", ...)

## S3 method for class 'subclass_result'
join_matched(result, data = NULL, ...)

## S3 method for class 'couplr_result'
augment(x, left, right, ...)
```

Arguments

result	A result object from match_couples(), full_match(), or cem_match()
...	Additional arguments passed to join_matched()
left	The original left dataset
right	The original right dataset
left_vars	Character vector of variable names to include from left. If NULL (default), includes all variables except the ID column.
right_vars	Character vector of variable names to include from right. If NULL (default), includes all variables except the ID column.
left_id	Name of the ID column in left dataset (default: "id")
right_id	Name of the ID column in right dataset (default: "id")
suffix	Character vector of length 2 specifying suffixes for left and right variables (default: c("_left", "_right"))
include_distance	Include the matching distance in output (default: TRUE)
include_pair_id	Include pair_id column (default: TRUE)
include_block_id	Include block_id if blocking was used (default: TRUE)
data	Data frame used for subclassification
x	A matching_result object

Details

This function simplifies the common workflow of joining matched pairs with original data. Instead of manually merging result$pairs with left and right datasets, `join_matched()` handles the joins automatically and applies consistent naming conventions.

When variables appear in both left and right datasets, suffixes are appended to distinguish them (e.g., "age_left" and "age_right"). This makes it easy to compute differences or use both values in models.

Value

A tibble with one row per matched pair, containing:

- `pair_id`: Sequential pair identifier (if include_pair_id = TRUE)
- `left_id`: ID from left dataset
- `right_id`: ID from right dataset
- `distance`: Matching distance (if include_distance = TRUE)
- `block_id`: Block identifier (if blocking used and include_block_id = TRUE)
- Variables from left dataset (with left suffix)
- Variables from right dataset (with right suffix)

Examples

```
# Basic usage
left <- data.frame(
  id = 1:5,
  treatment = 1,
  age = c(25, 30, 35, 40, 45),
  income = c(45000, 52000, 48000, 61000, 55000)
)

right <- data.frame(
  id = 6:10,
  treatment = 0,
  age = c(24, 29, 36, 41, 44),
  income = c(46000, 51500, 47500, 60000, 54000)
)

result <- match_couples(left, right, vars = c("age", "income"))
matched_data <- join_matched(result, left, right)
head(matched_data)

# Specify which variables to include
matched_data <- join_matched(
  result, left, right,
  left_vars = c("treatment", "age", "income"),
  right_vars = c("age", "income"),
  suffix = c("_treated", "_control")
)
```

```
# Without distance or pair_id
matched_data <- join_matched(
  result, left, right,
  include_distance = FALSE,
  include_pair_id = FALSE
)
```

`lap_animate` *Animate an Assignment Algorithm*

Description

Produce an interactive bipartite-graph animation showing how a chosen linear-assignment algorithm transforms the matching over time. The result is an htmlwidget suitable for use in R Markdown, Quarto, pkgdown vignettes, Shiny apps, or standalone HTML output.

Usage

```
lap_animate(
  cost,
  method = "hungarian",
  maximize = FALSE,
  width = NULL,
  height = NULL,
  elementId = NULL,
  ...
)
```

Arguments

`cost`	Numeric cost matrix. Rows = workers/sources, columns = jobs/targets. `NA` or `Inf` entries mark forbidden assignments.
`method`	Character; the algorithm to animate. Must match one of the methods registered for animation (see `animated_methods()` for the current list). Method names are the same strings used by `assignment()`, e.g. `"hungarian"`, `"jv"`, `"auction"`, `"gabow_tarjan"`.
`maximize`	Logical; if `TRUE`, animate the maximization variant.
`width, height`	Optional explicit widget dimensions (pixels or CSS units).
`elementId`	Optional DOM id for the widget container.
`...`	Algorithm-specific extra arguments forwarded to the trace function (e.g. `auction_eps`).

Details

This is a *teaching* interface. It runs a slower R reference implementation that emits a state trace at every step, then plays it back in the browser. For production solving, use `assignment()` or `lap_solve()` which call the fast C++ backends.

Value

An `htmlwidget` object.

Animated methods

Animation support is added incrementally. Call `animated_methods()` to see which method strings currently have a registered trace.

See Also

`assignment()` for production solving, `lap_solve()` for the tidy interface.

Examples

```
cost <- matrix(c(4, 2, 5,
                 3, 3, 6,
                 7, 5, 4), nrow = 3, byrow = TRUE)
if (requireNamespace("htmlwidgets", quietly = TRUE)) {
  widget <- lap_animate(cost, method = "hungarian")
  if (interactive()) print(widget)
}
```

---

lap_solve *Solve Linear Assignment Problems with a Tidy Interface*

Description

Provides a tidy interface for solving the linear assignment problem using 'Hungarian' or 'Jonker-Volgenant' algorithms. Supports rectangular matrices, NA/Inf masking, and data frame inputs.

Usage

```
lap_solve(
  x,
  source = NULL,
  target = NULL,
  cost = NULL,
  maximize = FALSE,
  method = "auto",
  forbidden = NA
)
```

Arguments

x	Cost matrix, data frame, or tibble. If a data frame/tibble, must include columns specified by source, target, and cost.
source	Column name for source/row indices (if x is a data frame)
target	Column name for target/column indices (if x is a data frame)
cost	Column name for costs (if x is a data frame)
maximize	Logical; if TRUE, maximizes total cost instead of minimizing (default: FALSE)
method	Algorithm to use. One of: • "auto" (default): automatically selects best algorithm • "jv": 'Jonker-Volgenant' algorithm (general purpose, fast) • "hungarian": Classic 'Hungarian' algorithm • "auction": 'Bertsekas' auction algorithm (good for large dense problems) • "sap": Sparse assignment (good for sparse/rectangular problems) • "hk01": 'Hopcroft-Karp' for binary/uniform costs
forbidden	Value to mark forbidden assignments (default: NA). Can also use Inf.

Value

A tibble with columns:

- source: row/source indices
- target: column/target indices
- cost: cost of each assignment
- total_cost: total cost (attribute)

Examples

```
# Matrix input
cost <- matrix(c(4, 2, 5, 3, 3, 6, 7, 5, 4), nrow = 3)
lap_solve(cost)

# Data frame input
library(dplyr)
df <- tibble(
  source = rep(1:3, each = 3),
  target = rep(1:3, times = 3),
  cost = c(4, 2, 5, 3, 3, 6, 7, 5, 4)
)
lap_solve(df, source, target, cost)

# With NA masking (forbidden assignments)
cost[1, 3] <- NA
lap_solve(cost)

# Grouped data frames
df <- tibble(
```

```
  sim = rep(1:2, each = 9),
  source = rep(1:3, times = 6),
  target = rep(1:3, each = 3, times = 2),
  cost = runif(18, 1, 10)
)
df |> group_by(sim) |> lap_solve(source, target, cost)
```

`lap_solvers` *Assignment Solvers*

Description

`lap_solvers()` lists the solvers `assignment()` accepts as `method`, and `register_lap_solver()` adds one.

Usage

```
lap_solvers()

register_lap_solver(
  name,
  solve,
  algorithm = name,
  potentials = FALSE,
  overwrite = FALSE
)
```

Arguments

`name`	Name under which the solver is selected with `assignment(method = name)`.
`solve`	The solver function, see Details.
`algorithm`	Short description shown by `lap_solvers()`.
`potentials`	Logical; whether `solve` returns optimal dual potentials `u` and `v` of the matrix it receives.
`overwrite`	Logical; whether to replace an existing solver of the same name. Built-in solvers cannot be replaced.

Details

A solver is a function `solve(cost, maximize, ...)` that receives a numeric matrix with no more rows than columns (forbidden pairs are `Inf`, or `-Inf` when `maximize = TRUE`) and returns either an integer vector giving the column assigned to each row, 0 for an unmatched row, or a list with that vector as `match` and optionally `total_cost`, `status` and the dual potentials `u` (rows) and `v` (columns). `assignment()` transposes wider problems before calling it, computes the status and, for a solver registered here, attaches a certificate by default: `verify_assignment()` checks the returned potentials when `potentials = TRUE` and otherwise computes optimal ones, so a registered solver's answer is proved optimal or reported as not.

Value

`lap_solvers()` returns a tibble with one row per solver and columns `method`, `algorithm`, `reference`, `potentials` (returns optimal dual potentials), `lazy` (available with `memory_mode = "lazy"`) and `builtin`. `register_lap_solver()` returns `name` invisibly.

Examples

```
lap_solvers()

# A greedy heuristic: each row in turn takes its cheapest free column.
greedy <- function(cost, maximize, ...) {
  if (maximize) cost <- -cost
  match <- integer(nrow(cost))
  free <- rep(TRUE, ncol(cost))
  for (i in seq_len(nrow(cost))) {
    j <- which(free)[which.min(cost[i, free])]
    match[i] <- j
    free[j] <- FALSE
  }
  match
}
register_lap_solver("greedy_rows", greedy, algorithm = "Row-wise greedy")
cost <- matrix(c(1, 2, 3, 2, 4, 6, 3, 6, 10), 3)
res <- assignment(cost, method = "greedy_rows")
certificate(res)
```

---

lap_solve_batch *Solve Many Assignment Problems*

Description

Solve many independent assignment problems at once. Supports lists of matrices, 3D arrays, or grouped data frames. Optional parallel execution via `n_threads`.

Usage

```
lap_solve_batch(
  x,
  source = NULL,
  target = NULL,
  cost = NULL,
  maximize = FALSE,
  method = "auto",
  n_threads = 1,
  forbidden = NA
)
```

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

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

is_lap_solve_batch_result(x)
```

Arguments

x	Object to test
source	Column name for source indices (if x is a grouped data frame)
target	Column name for target indices (if x is a grouped data frame)
cost	Column name for costs (if x is a grouped data frame)
maximize	Logical; if TRUE, maximizes total cost (default: FALSE)
method	Algorithm to use (default: "auto"). See lap_solve for options.
n_threads	Number of threads for parallel execution (default: 1). Set to NULL to use all available cores but one, as reported by parallelly::availableCores().
forbidden	Value to mark forbidden assignments (default: NA)
...	Additional arguments passed to print(). Currently ignored.
object	A lap_solve_batch_result.

Value

A tibble with columns:

- problem_id: identifier for each problem
- source: source indices for assignments
- target: target indices for assignments
- cost: cost of each assignment
- total_cost: total cost for each problem
- method_used: algorithm used for each problem

summary() returns a summary.lap_solve_batch_result whose problems table has one row per input problem, with grouping keys, assignment counts, total costs, and solver names. Each problem contributes once.

Examples

```
# List of matrices
costs <- list(
  matrix(c(1, 2, 3, 4), 2, 2),
  matrix(c(5, 6, 7, 8), 2, 2)
)
lap_solve_batch(costs)
```

```
# 3D array
arr <- array(runif(2 * 2 * 10), dim = c(2, 2, 10))
lap_solve_batch(arr)

# Grouped data frame
library(dplyr)
df <- tibble(
  sim = rep(1:5, each = 9),
  source = rep(1:3, times = 15),
  target = rep(1:3, each = 3, times = 5),
  cost = runif(45, 1, 10)
)
df |> group_by(sim) |> lap_solve_batch(source, target, cost)

# Parallel execution (requires n_threads > 1)
lap_solve_batch(costs, n_threads = 2)
```

---

lap_solve_kbest *Find the k Best Assignments*

Description

Returns the top k optimal (or near-optimal) assignments using 'Murty' algorithm. Useful for exploring alternative optimal solutions or finding robust assignments.

Usage

```
lap_solve_kbest(
  x,
  k = 3,
  source = NULL,
  target = NULL,
  cost = NULL,
  maximize = FALSE,
  method = "murty",
  single_method = "jv",
  forbidden = NA
)

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

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

is_lap_solve_kbest_result(x)
```

Arguments

x	Object to test
k	Number of best solutions to return (default: 3)
source	Column name for source/row indices (if x is a data frame)
target	Column name for target/column indices (if x is a data frame)
cost	Column name for costs (if x is a data frame)
maximize	Logical; if TRUE, finds k-best maximizing assignments (default: FALSE)
method	Algorithm for each sub-problem (default: "murty"). Future versions may support additional methods.
single_method	Algorithm used for solving each node in the search tree (default: "jv")
forbidden	Value to mark forbidden assignments (default: NA)
...	Additional arguments (unused).
object	An object of class lap_solve_kbest_result.

Value

A tibble with columns:

- rank: ranking of solutions (1 = best, 2 = second best, etc.)
- solution_id: unique identifier for each solution
- source: source indices
- target: target indices
- cost: cost of each edge in the assignment
- total_cost: total cost of the complete solution

Examples

```
# Matrix input - find 5 best solutions
cost <- matrix(c(4, 2, 5, 3, 3, 6, 7, 5, 4), nrow = 3)
lap_solve_kbest(cost, k = 5)

# Data frame input
library(dplyr)
df <- tibble(
  source = rep(1:3, each = 3),
  target = rep(1:3, times = 3),
  cost = c(4, 2, 5, 3, 3, 6, 7, 5, 4)
)
lap_solve_kbest(df, k = 3, source, target, cost)

# With maximization
lap_solve_kbest(cost, k = 3, maximize = TRUE)
```

`lap_solve_line_metric` *Solve a One-Dimensional Assignment Problem*

Description

Solves the linear assignment problem when both sources and targets are ordered points on a line. Uses efficient O(n*m) dynamic programming for rectangular problems and O(n) sorting for square problems.

Usage

```
lap_solve_line_metric(x, y, cost = "L1", maximize = FALSE)

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

Arguments

`x`	Numeric vector of source positions (will be sorted internally)
`y`	Numeric vector of target positions (will be sorted internally)
`cost`	Cost function for distance. Either: • "L1" (default): absolute distance ('Manhattan' distance) • "L2": squared distance (squared 'Euclidean' distance) Can also use aliases: "abs", "manhattan" for L1; "sq", "squared", "quadratic" for L2
`maximize`	Logical; if TRUE, maximizes total cost instead of minimizing (default: FALSE)
`object`	A result returned by this function.
`...`	Ignored by the summary method.

Details

This is a specialized solver that exploits the structure of 1-dimensional assignment problems where costs depend only on the distance between points on a line. It is much faster than general LAP solvers for this special case.

The algorithm works as follows:

Square case (n == m): Both vectors are sorted and matched in order: `x[1]` -> `y[1]`, `x[2]` -> `y[2]`, etc. This is optimal for any metric cost function on a line.

Rectangular case (n < m): Uses dynamic programming to find the optimal assignment that matches all n sources to a subset of the m targets, minimizing total distance. The DP recurrence is:

`dp[i][j] = min(dp[i][j-1], dp[i-1][j-1] + cost(x[i], y[j]))`

This finds the minimum cost to match the first i sources to the first j targets.

Complexity:

- Time: O(n*m) for rectangular, O(n log n) for square
- Space: O(n*m) for DP table

Value

A list with components:

- match: Integer vector of length n with 1-based column indices
- total_cost: Total cost of the assignment

summary() returns a summary.assignment_result with the objective and the complete row-to-column assignment table.

Examples

```
# Square case: equal number of sources and targets
x <- c(1.5, 3.2, 5.1)
y <- c(2.0, 3.0, 5.5)
result <- lap_solve_line_metric(x, y, cost = "L1")
print(result)

# Rectangular case: more targets than sources
x <- c(1.0, 3.0, 5.0)
y <- c(0.5, 2.0, 3.5, 4.5, 6.0)
result <- lap_solve_line_metric(x, y, cost = "L2")
print(result)

# With unsorted inputs (will be sorted internally)
x <- c(5.0, 1.0, 3.0)
y <- c(4.5, 0.5, 6.0, 2.0, 3.5)
result <- lap_solve_line_metric(x, y, cost = "L1")
print(result)
```

---

matchmaker *Create Blocks for Stratified Matching*

Description

Constructs blocks (strata) for matching, using either grouping variables or clustering algorithms. Returns the input data frames with block IDs assigned, along with block summary statistics.

Usage

```
matchmaker(
  left,
  right,
  block_type = c("none", "group", "cluster"),
  block_by = NULL,
  block_vars = NULL,
  block_method = "kmeans",
  n_blocks = NULL,
```

```
  min_left = 1,
  min_right = 1,
  drop_imbalanced = FALSE,
  imbalance_threshold = Inf,
  return_dropped = TRUE,
  ...
)

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

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

Arguments

`left`	Data frame of "left" units (e.g., treated, cases)
`right`	Data frame of "right" units (e.g., control, controls)
`block_type`	Type of blocking to use: • "none": No blocking (default) • "group": Block by existing categorical variable(s) • "cluster": Block using clustering algorithm
`block_by`	Variable name(s) for grouping (if block_type = "group")
`block_vars`	Variable names for clustering (if block_type = "cluster")
`block_method`	Clustering method (if block_type = "cluster"): • "kmeans": K-means clustering • "hclust": Hierarchical clustering
`n_blocks`	Target number of blocks (for clustering)
`min_left`	Minimum number of left units per block
`min_right`	Minimum number of right units per block
`drop_imbalanced`	Drop blocks with extreme imbalance
`imbalance_threshold`	Maximum allowed |n_left - n_right| / max(n_left, n_right)
`return_dropped`	Include dropped blocks in output
`...`	Additional arguments passed to the clustering function; ignored by the print and summary methods.
`x`	A matchmaker_result object
`object`	A `matchmaker_result` object.

Details

This function does NOT perform matching - it only creates the block structure. Use `match_couples()` to perform matching within blocks.

Value

A list with class "matchmaker_result" containing:

- `left`: Left data frame with block_id column added
- `right`: Right data frame with block_id column added
- `block_summary`: Summary statistics for each block
- `dropped`: Information about dropped blocks (if any)
- `info`: Metadata about blocking process

`print()` shows retained unit and block counts. `summary()` returns a `summary.matchmaker_result` with `info`, `block_summary`, and `dropped`; printing it also shows every block and the reasons for dropped blocks.

Examples

```
# Group blocking
left <- data.frame(id = 1:10, region = rep(c("A", "B"), each = 5), x = rnorm(10))
right <- data.frame(id = 11:20, region = rep(c("A", "B"), each = 5), x = rnorm(10))
blocks <- matchmaker(left, right, block_type = "group", block_by = "region")
print(blocks$block_summary)

# Clustering
blocks <- matchmaker(left, right, block_type = "cluster",
                     block_vars = "x", n_blocks = 3)
```

---

match_couples *Optimal and Greedy Pair Matching*

Description

Performs one-to-one matching between two datasets. Supports blocking, distance constraints, and various distance metrics.

Usage

```
match_couples(
  left,
  right = NULL,
  vars = NULL,
  left_id = NULL,
  right_id = NULL,
  distance = "euclidean",
  weights = NULL,
  scale = FALSE,
  auto_scale = FALSE,
  max_distance = Inf,
```

```
  calipers = NULL,
  block_id = NULL,
  ignore_blocks = FALSE,
  require_full_matching = FALSE,
  method = "auto",
  strategy = c("row_best", "sorted", "pq"),
  return_unmatched = TRUE,
  return_diagnostics = FALSE,
  parallel = FALSE,
  replace = FALSE,
  ratio = 1L,
  check_costs = TRUE,
  sigma = NULL,
  memory_mode = "auto",
  certify = NULL
)
```

Arguments

`left`	Data frame of "left" units (e.g., treated, cases)
`right`	Data frame of "right" units (e.g., control, controls)
`vars`	Variable names to use for distance computation
`left_id, right_id`	Name of the column holding the unit identifier, or NULL (default) to use a column called `id`, then meaningful row names, then synthesized ids `left_1 ... left_n` / `right_1 ... right_m` with a warning. The values of this column are what `pairs$left_id` and `pairs$right_id` carry, and what `join_matched()`, `match_data()`, `balance_diagnostics()`, `sensitivity_analysis()` and `as_matchit()` join on, so the same column name is passed to those verbs. Ids read from the data must be unique.
`distance`	Distance metric: "euclidean", "manhattan", "mahalanobis", or a custom function
`weights`	Optional named vector of variable weights
`scale`	Scaling method: FALSE (none), "standardize", "range", or "robust"
`auto_scale`	If TRUE, automatically check variable health and select scaling method (default: FALSE)
`max_distance`	Maximum allowed distance (pairs exceeding this are forbidden)
`calipers`	Named list of maximum absolute differences, each on a column of both data frames: a matching variable or another one, such as a propensity score.
`block_id`	Column name containing block IDs (for stratified matching)
`ignore_blocks`	If TRUE, ignore block_id even if present
`require_full_matching`	If TRUE, error if any units remain unmatched
`method`	Matching method. A LAP solver for optimal matching ("auto", "hungarian", "jv", "gabow_tarjan", ...), or "greedy" for fast approximate matching (see `strategy`).

`strategy` Greedy strategy, used only when `method = "greedy"`. All three strategies solve the same full cost matrix already built by `match_couples()`; none of them reduce the O(n*m) memory that matrix takes.

- "row_best": for each row, take its best available column (default). The only strategy that needs no extra storage beyond the cost matrix.
- "sorted": collect every valid pair, sort by distance, greedily assign
- "pq": collect every valid pair into a heap and pop the smallest first. Avoids the upfront sort but holds the same number of candidate pairs as "sorted", so it is not more memory-efficient

`return_unmatched` Include unmatched units in output

`return_diagnostics` Include detailed diagnostics in output

`parallel` Enable parallel processing for blocked matching. Requires 'future' and 'future.apply' packages. Can be:

- `FALSE`: Sequential processing (default)
- `TRUE`: Auto-configure parallel backend
- Character: Specify future plan (e.g., "multisession", "multicore")

`replace` If TRUE, allow matching with replacement (same right unit can be matched to multiple left units). Default: FALSE.

`ratio` Integer, number of right units to match per left unit. Default: 1 (one-to-one matching). For k:1 matching, set ratio = k.

`check_costs` If TRUE, check distance distribution for potential problems and provide helpful warnings before matching (default: TRUE)

`sigma` Optional covariance matrix for Mahalanobis distance. If NULL (default), the pooled sample covariance is used. Only relevant when `distance = "mahalanobis"`.

`memory_mode` One of "auto" (default), "dense", "lazy" or "implicit". "auto" warns (or, when `method` is `"jv"`/`"auction"`, switches) when the dense cost matrix would consume a large fraction of free system RAM. "lazy" computes each pairwise distance from the underlying feature data as the solver needs it, instead of allocating the full n_left x n_right matrix; supported for `method = "jv"`/`"auction"`, including `replace = TRUE`, where each left unit's cheapest partners are found one row at a time, and `ratio > 1`, where the left units' covariates are replicated rather than their rows of distances, and not yet for `method = "greedy"` (blocking via `block_id` is the other option that reduces memory, by solving smaller subproblems). A custom distance function is called on a block of left units against every right unit, the same contract the dense path calls it under, with the block sized so the matrix it returns stays near a million cells; each matched pair is then evaluated again alone, and a distance that depends on the other units in its call is an error. When the constraints admit no complete matching, the largest matching they admit, cheapest among those, is found by the edge-generation loop over the same specification, as the dense path finds it by padding. Where the metric carries a ball bound, the column set is held in a ball tree and a subtree whose bound cannot beat the current threshold is discarded without being read: `"mahalanobis"` always, the metrics linear in the covariates up to six of them.

"manhattan" and "chebyshev", a covariance with no Cholesky factor, and a higher-dimensional linear metric read the columns instead. "implicit" states the problem over every pair and solves it over a fraction of them, generating the pairs the answer turns out to need and proving that the ones it never generated could not have improved it; same requirements as "lazy". On the eight-covariate problem the benchmarks use it leads "lazy" from 5,000 units upward, by 1.1x at 5,000 rising to 3.1x at 50,000, and loses below that where the loop's fixed costs are still visible; what it buys at every size is the certificate over the complete problem. "auto" never selects it. "dense" skips the RAM check entirely.

`certify` Logical; whether the result carries a checked `assignment_certificate`, read with `certificate()`. Defaults to `TRUE` under `memory_mode = "implicit"`, where the certificate is part of the solve, and to `FALSE` elsewhere, where it costs one extra pass over the admissible pairs. Available for one-to-one and `ratio`-to-one matching without blocks; greedy matching and matching with replacement solve no assignment problem and carry none.

Details

With `method` set to a LAP solver (the default `"auto"`, or `"jv"`, `"hungarian"`, ...) it finds the matching that minimizes total distance among all feasible matchings. With `method = "greedy"` it uses a fast greedy strategy (selected by `strategy`) that does not guarantee the optimal total distance but scales to very large datasets.

Value

A list with class "matching_result" containing:

- `pairs`: Tibble of matched pairs with distances
- `unmatched`: List of unmatched left and right IDs
- `info`: Matching diagnostics and metadata
- `status`: One of `solver_status_values()`, computed from what the solve achieved. `"optimal"` when every left unit found a partner under an optimal method, `"partial"` when constraints left some unmatched, `"heuristic"` when a greedy method ran, either because it was asked for or because the constrained path fell back to it, and `"infeasible"` when nothing could be matched.
- `potentials`: The dual potentials of the design's linear program, a list with elements `left` and `right` holding one value per unit, named by id. The reduced cost of a pair is its distance minus the two potentials: at least zero on every admissible pair the matching left out, zero on every pair of a 1:1 matching, and at most zero on the pairs of a k:1 or with-replacement matching, whose pair arcs carry at most one unit. A left unit's potential on the k:1 design is the largest of its replicas'. With replacement the LP separates by row: a left unit's potential is the distance to the farthest partner it took and every right unit's is zero. A unit with no admissible partner, or in a block with none, reads `NA`. Absent for `method = "greedy"` and for a constrained problem answered by maximum cardinality, which is not the LP the potentials belong to. On the 1:1 design they are the duals `verify_assignment()` checks, and a method that returns none has them computed by `assignment_duals()`.

Under `memory_mode = "implicit"` it also carries `certificate`, the checked statement of optimality (see `verify_assignment()`, which names the arithmetic it was decided in), and `search`: the pairs the loop generated out of the pairs the problem states, the pairs a cost was computed for, and one row per round of what each round did. An infeasible answer carries `witness` instead, naming the units that could not be matched and the partners they have between them.

Examples

```
# Basic matching
left <- data.frame(id = 1:5, x = c(1, 2, 3, 4, 5), y = c(2, 4, 6, 8, 10))
right <- data.frame(id = 6:10, x = c(1.1, 2.2, 3.1, 4.2, 5.1), y = c(2.1, 4.1, 6.2, 8.1, 10.1))
result <- match_couples(left, right, vars = c("x", "y"))
print(result$pairs)

# With constraints
result <- match_couples(left, right, vars = c("x", "y"),
                        max_distance = 1,
                        calipers = list(x = 0.5))

# With blocking
left$region <- c("A", "A", "B", "B", "B")
right$region <- c("A", "A", "B", "B", "B")
blocks <- matchmaker(left, right, block_type = "group", block_by = "region")
result <- match_couples(blocks$left, blocks$right, vars = c("x", "y"))

# Fast greedy matching for large datasets
result <- match_couples(left, right, vars = c("x", "y"),
                        method = "greedy", strategy = "sorted")
```

---

match_data *Extract Analysis-Ready Data from a Matching Result*

Description

A generic function that converts any couplr matching result into a single analysis-ready data frame with `weights`, `subclass`, and `distance` columns. This is the couplr equivalent of MatchIt's `match.data()`.

Usage

```
match_data(result, ...)

## S3 method for class 'matching_result'
match_data(result, left, right, left_id = "id", right_id = "id", ...)

## S3 method for class 'full_matching_result'
match_data(result, left, right, left_id = "id", right_id = "id", ...)
```

```
## S3 method for class 'cem_result'
match_data(result, left, right, left_id = "id", right_id = "id", ...)

## S3 method for class 'subclass_result'
match_data(result, data = NULL, ...)
```

Arguments

result	A couplr result object (matching_result, full_matching_result, cem_result, or subclass_result)
...	Additional arguments passed to methods
left	Data frame of left (treated) units
right	Data frame of right (control) units
left_id	Name of ID column in left (default: "id")
right_id	Name of ID column in right (default: "id")
data	Data frame containing all units (for CEM and subclassification, left and right are not always needed separately)

Details

The output format is compatible with downstream packages like **cobalt**, **WeightIt**, and **marginaleffects**. The stacked (long) format with `treatment` and `weights` columns is the standard layout expected by these tools.

Value

A tibble with all original variables plus standardized columns:

id Unit identifier

treatment 1 for left/treated, 0 for right/control

weights Matching weights

subclass Matched group/stratum identifier

distance Matching distance (where applicable)

Examples

```
set.seed(42)
left <- data.frame(id = 1:5, age = c(25, 35, 45, 55, 65))
right <- data.frame(id = 6:15, age = runif(10, 20, 70))
result <- match_couples(left, right, vars = "age")
md <- match_data(result, left, right)
head(md)
```

match_path *Solve a Sequence of Matchings over One Design Parameter*

Description

Solves the matching once for each value of one argument and returns what happened at each: how many units were matched, at what total distance, how balanced the matched sample is, and the certificate saying the matching is optimal for that value. The points are solved as one sequence rather than one at a time, each starting from the matching the previous value found.

Usage

```
match_path(
  left,
  right,
  vars,
  left_id = NULL,
  right_id = NULL,
  vary = "max_distance",
  values,
  distance = "euclidean",
  weights = NULL,
  scale = FALSE,
  auto_scale = FALSE,
  calipers = NULL,
  sigma = NULL,
  certify = TRUE,
  keep_per_row = .implicit_defaults()$keep_per_row,
  width = .implicit_defaults()$width,
  tol = .implicit_defaults()$tol,
  max_rounds = .implicit_defaults()$max_rounds
)

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

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

## S3 method for class 'couplr_path'
as.data.frame(
  x,
  row.names = NULL,
  optional = FALSE,
  what = c("path", "balance"),
  ...
)
```

```
## S3 method for class 'couplr_path'
plot(x, threshold = 0.1, ...)
```

Arguments

left, right	Data frames to match. Both are required.
vars	Character vector of matching variables, present in both.
left_id, right_id	Name of the column holding the unit identifier, or NULL (default) to use a column called id, then meaningful row names, then synthesized ids with a warning. See match_couples().
vary	The argument to sweep. "max_distance" sweeps the distance cut.
values	The values to sweep it through, ascending.
distance	Distance metric, as in match_couples().
weights, scale, auto_scale, sigma	Distance construction, as in match_couples().
calipers	Per-variable calipers held fixed across the path, as in match_couples(). Balance is read on the caller's own variables rather than on the coordinates the solver worked in, so a scaled or weighted distance does not change the scale the balance is reported on. It is the same reading balance_diagnostics() gives for a single matching, taken at every point of the sweep.
certify	Whether each point carries a checked certificate. TRUE by default: the certificate is what says a point's matching is the optimal one for its value, which is the claim a path is read for.
keep_per_row, width, tol, max_rounds	The edge-generation loop's search knobs, shared with memory_mode = "implicit". Each point converges on any of them. width is the columns the seed gives a row, and 0, the default, sizes it from the number of columns; $search$seed_width reports what the path used.
x, object	A couplr_path from match_path().
...	Ignored.
row.names, optional	Ignored.
what	For as.data.frame(): "path" (one row per point) or "balance" (one row per point and variable).
threshold	For plot(): reference line for the largest absolute standardized difference (default 0.1).

Details

values must ascend. Each point is solved from the point before it, which works because a wider value only ever adds pairs to choose from; a descending sweep takes pairs away from a matching already using them, and is refused rather than reordered.

Value

An object of class `couplr_path`: `$path`, one row per point, `$balance`, one row per point per variable, and the match vector, certificate, round record and Hall witness for each of them.

Examples

```
set.seed(1)
left <- data.frame(id = 1:20, x = rnorm(20), y = rnorm(20))
right <- data.frame(id = 1:60, x = rnorm(60), y = rnorm(60))
path <- match_path(left, right, vars = c("x", "y"),
                   vary = "max_distance", values = c(0.5, 1, 2, Inf))
as.data.frame(path)
summary(path)
```

---

pixel_morph *Pixel-Level Image Morphing*

Description

Computes optimal pixel assignment from A to B and returns the final transported frame (without intermediate animation frames).

Usage

```
pixel_morph(
  imgA,
  imgB,
  n_frames = 16L,
  mode = c("color_walk", "exact", "recursive", "color_match"),
  lap_method = "jv",
  maximize = FALSE,
  quantize_bits = 5L,
  downscale_steps = 0L,
  alpha = 1,
  beta = 0,
  patch_size = 1L,
  upscale = 1,
  show = interactive()
)
```

Arguments

`imgA`	Source image (file path or magick image object)
`imgB`	Target image (file path or magick image object)
`n_frames`	Internal parameter for rendering (default: 16)

mode	Assignment algorithm: "color_walk" (default), "exact", "recursive", or "color_match" (spatially match pixels of identical quantized colour, filling the remainder by identity)
lap_method	LAP solver method (default: "jv")
maximize	Logical, maximize instead of minimize cost (default: FALSE)
quantize_bits	Color quantization for "color_walk" mode (default: 5)
downscale_steps	Number of 2x reductions before computing assignment (default: 0)
alpha	Weight for color distance in cost function (default: 1)
beta	Weight for spatial distance in cost function (default: 0)
patch_size	Tile size for tiled modes (default: 1)
upscale	Post-rendering upscaling factor (default: 1)
show	Logical, display result in viewer (default: interactive())

Details

Transport-Only Semantics:

This function returns a SHARP, pixel-perfect transport of A's pixels to positions determined by the assignment to B.

Key Points:

- Assignment computed using: `cost = alpha * color_dist + beta * spatial_dist`
- B's COLORS influence assignment but DO NOT appear in output
- Result has A's colors arranged to match B's layout
- No motion blur (unlike intermediate frames in animation)

See `pixel_morph_animate` for detailed explanation of assignment vs rendering semantics.

Permutation Warnings:

Assignment is guaranteed to be a bijection (permutation) ONLY when:

- `downscale_steps = 0` (no resolution changes)
- `mode = "exact"` with `patch_size = 1`

With downscaling or tiled modes, assignment may have:

- **Overlaps:** Multiple source pixels map to same destination (last write wins)
- **Holes:** Some destinations never filled (remain transparent)

If assignment is not a bijection (due to downscaling or tiling), a warning will be issued. The result may contain:

- Overlapped pixels (multiple sources -> one destination)
- Transparent holes (some destinations unfilled)

For guaranteed pixel-perfect results, use:

```
pixel_morph(A, B, mode = "exact", downscale_steps = 0)
```

Value

magick image object of the final transported frame

See Also

pixel_morph_animate for animated version

Examples

```
if (requireNamespace("magick", quietly = TRUE)) {
  imgA <- system.file("extdata/icons/circleA_40.png", package = "couplr")
  imgB <- system.file("extdata/icons/circleB_40.png", package = "couplr")
  result <- pixel_morph(imgA, imgB, n_frames = 2, show = FALSE)
  result
}
```

---

pixel_morph_animate *Animated Pixel-Level Image Morphing*

Description

Creates an animated morph by computing optimal pixel assignment from image A to image B, then rendering intermediate frames showing the transport.

Usage

```
pixel_morph_animate(
  imgA,
  imgB,
  n_frames = 16L,
  fps = 10L,
  format = c("gif", "webp", "mp4"),
  outfile = NULL,
  show = interactive(),
  mode = c("color_walk", "exact", "recursive", "color_match"),
  lap_method = "jv",
  maximize = FALSE,
  quantize_bits = 5L,
  downscale_steps = 0L,
  alpha = 1,
  beta = 0,
  patch_size = 1L,
  upscale = 1
)

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

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

Arguments

imgA	Source image (file path or magick image object)
imgB	Target image (file path or magick image object)
n_frames	Integer number of animation frames (default: 16)
fps	Frames per second for playback (default: 10)
format	Output format: "gif", "webp", or "mp4"
outfile	Optional output file path
show	Logical, display animation in viewer (default: interactive())
mode	Assignment algorithm: "color_walk" (default), "exact", "recursive", or "color_match" (spatially match pixels of identical quantized colour, filling the remainder by identity)
lap_method	LAP solver method (default: "jv")
maximize	Logical, maximize instead of minimize cost (default: FALSE)
quantize_bits	Color quantization for "color_walk" mode (default: 5)
downscale_steps	Number of 2x reductions before computing assignment (default: 0)
alpha	Weight for color distance in cost function (default: 1)
beta	Weight for spatial distance in cost function (default: 0)
patch_size	Tile size for tiled modes (default: 1)
upscale	Post-rendering upscaling factor (default: 1)
x, object	A pixel_morph_animation result.
...	Ignored.

Details

Assignment vs Rendering Semantics:

CRITICAL: This function has two separate phases with different semantics:

Phase 1 - Assignment Computation:

The assignment is computed by minimizing:

```
cost(i,j) = alpha * color_distance(A[i], B[j]) +
            beta * spatial_distance(pos_i, pos_j)
```

This means B's COLORS influence which pixels from A map to which positions.

Phase 2 - Rendering (Transport-Only):

The renderer uses ONLY A's colors:

- Intermediate frames: A's pixels move along paths with motion blur
- Final frame: A's pixels at their assigned positions (sharp, no blur)
- B's colors NEVER appear in the output

Result: You get A's colors rearranged to match B's geometry/layout.

What This Means:

- B influences WHERE pixels go (via similarity in cost function)
- B does NOT determine WHAT COLORS appear in output
- Final image has A's palette arranged to mimic B's structure

Parameter Guidance:

For pure spatial rearrangement (ignore B's colors in assignment):

```
pixel_morph_animate(A, B, alpha = 0, beta = 1)
```

For color-similarity matching (default):

```
pixel_morph_animate(A, B, alpha = 1, beta = 0)
```

For hybrid (color + spatial):

```
pixel_morph_animate(A, B, alpha = 1, beta = 0.2)
```

Permutation Guarantees:

Assignment is guaranteed to be a bijection (permutation) ONLY when:

- `downscale_steps = 0` (no resolution changes)
- `mode = "exact"` with `patch_size = 1`

With downscaling or tiled modes, assignment may have:

- **Overlaps:** Multiple source pixels map to same destination (last write wins)
- **Holes:** Some destinations never filled (remain transparent)

A warning is issued if overlaps/holes are detected in the final frame.

Value

Invisibly returns a `pixel_morph_animation` list with animation object and metadata:

`animation`	magick animation object
`width`	Image width in pixels
`height`	Image height in pixels
`assignment`	Integer vector of 1-based assignment indices (R convention)
`n_pixels`	Total number of pixels
`mode`	Mode used for matching
`upscale`	Upscaling factor applied

`summary()` returns a `summary.pixel_morph_animation` with frame counts and assignment coverage, including repeated and unfilled destinations.

Examples

```
if (requireNamespace("magick", quietly = TRUE)) {
  imgA <- system.file("extdata/icons/circleA_40.png", package = "couplr")
  imgB <- system.file("extdata/icons/circleB_40.png", package = "couplr")
  outfile <- tempfile(fileext = ".gif")
  pixel_morph_animate(imgA, imgB, outfile = outfile, n_frames = 4, show = FALSE)
  file.exists(outfile)
}
```

`plot.matching_result` *Matching Results*

Description

Every matching function returns an object inheriting from class `"couplr_result"`:

Function	Class	Matched units
`match_couples()`, `cardinality_match()`	`"matching_result"`	pairs
`full_match()`	`"full_matching_result"`	groups
`cem_match()`	`"cem_result"`	strata
`subclass_match()`	`"subclass_result"`	subclasses

`print()` gives a short overview and `summary()` the full report: the distribution of pair distances, the certificate conditions, the column generation rounds and the balance constraints of a cardinality matching, where present. `glance()` returns the headline numbers as a one-row tibble and `as.data.frame()` (or `tidy()`) the matched units, one row per pair or per unit. `certificate()`, `potentials()` and `unmatched_units()` read the certificate, the dual potentials and the unmatched units. Further methods take any of these results: `balance_diagnostics()`, `join_matched()`, `match_data()`, `as_matchit()` and `cobalt::bal.tab()`.

Usage

```
## S3 method for class 'matching_result'
plot(x, type = c("histogram", "density", "ecdf"), ...)

## S3 method for class 'matching_result'
autoplot(object, type = c("histogram", "density", "ecdf"), ...)

## S3 method for class 'couplr_result'
glance(x, ...)

## S3 method for class 'couplr_result'
tidy(x, ...)

## S3 method for class 'couplr_result'
as.data.frame(x, row.names = NULL, optional = FALSE, ...)

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

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

Arguments

x, object	A matching result.
type	Type of plot: "histogram" (default), "density", or "ecdf"
...	Ignored.
row.names, optional	Ignored; present for consistency with the generic.

Value

print() returns x invisibly. summary() returns an object of class "summary.couplr_result". glance() returns a one-row tibble with columns design, n_left, n_right, n_matched_left, n_matched_right, n_pairs, n_groups, total_distance, status, certified and max_suboptimality, plus best_possible, gap, n_nodes, stopped_on, size_certified (whether the largest size the constraints admit was proven, which certified also requires of the balance and distance stages) and size_seconds (how long proving it took, NA when one network solve answered the call) for a cardinality matching and n_rounds, candidate_edges and possible_edges for column generation. as.data.frame() and tidy() return the matched units.

Examples

```
set.seed(1)
left <- data.frame(id = 1:5, x = rnorm(5))
right <- data.frame(id = 6:15, x = rnorm(10))
m <- match_couples(left, right, vars = "x", certify = TRUE)
m
glance(m)
as.data.frame(m)
certificate(m)
summary(m)
```

---

preprocess_matching_vars

Check Matching Variables and Choose Scaling

Description

Main preprocessing function that orchestrates variable health checks, and automatic scaling selection. The returned data frames retain their input values; match_couples() applies the selected scaling when building distances.

Usage

```
preprocess_matching_vars(
  left,
  right,
  vars,
```

```
  auto_scale = TRUE,
  scale_method = "auto",
  check_health = TRUE,
  remove_problematic = TRUE,
  verbose = TRUE
)

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

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

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

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

Arguments

left	Data frame of left units
right	Data frame of right units
vars	Character vector of variable names
auto_scale	Logical, whether to perform automatic preprocessing (default: TRUE)
scale_method	Scaling method: "auto", "standardize", "range", "robust", or FALSE
check_health	Logical, whether to check variable health (default: TRUE)
remove_problematic	Logical, automatically exclude constant/all-NA variables (default: TRUE)
verbose	Logical, whether to print warnings (default: TRUE)
x	A preprocessing_result object
...	Additional arguments (ignored)
object	A preprocessing result or its nested variable-health report.

Value

A list with class "preprocessing_result" containing:

- left: Input left data frame
- right: Input right data frame
- vars: Final variable names (after exclusions)
- health: Variable health diagnostics
- scaling_method: Selected scaling method
- excluded_vars: Variables that were excluded
- warnings: List of warnings issued

summary() returns a summary.preprocessing_result with the retained variables, group sizes, scaling choice, exclusions, warnings, and the detailed variable-health report.

---

ps_match *Propensity Score Matching*

Description

Matches treated and control units based on estimated propensity scores. Fits a logistic regression model (or accepts a pre-fitted one), computes logit propensity scores, and calls match_couples() with a caliper on the logit scale.

Usage

```
ps_match(
  formula = NULL,
  data = NULL,
  treatment = NULL,
  ps_model = NULL,
  caliper_sd = 0.2,
  method = "auto",
  replace = FALSE,
  ratio = 1L,
  ...
)
```

Arguments

formula	Formula for propensity score model (treatment ~ covariates). Required if ps_model is NULL.
data	Combined dataset containing both treated and control units
treatment	Name of the binary treatment column (0/1 or logical)
ps_model	Pre-fitted glm object (alternative to formula). If provided, formula is ignored.
caliper_sd	Caliper width in standard deviations of logit(PS). Default: 0.2 (Rosenbaum and Rubin recommendation).
method	LAP solver method (default: "auto")
replace	If TRUE, match with replacement (default: FALSE)
ratio	Integer k for k:1 matching (default: 1)
...	Additional arguments passed to match_couples()

Details

The propensity score is the probability of treatment assignment conditional on observed covariates. Matching is performed on the logit of the propensity score (Rosenbaum and Rubin 1985), which provides better distributional properties than matching on the raw probability scale.

The default caliper of 0.2 SD of logit(PS) is recommended by Austin (2011) as removing approximately 98% of bias.

Value

A matching_result object with additional propensity score info in `result$info$ps_model` and `result$info$caliper_value`.

Examples

```
set.seed(42)
n <- 100
data <- data.frame(
  id = seq_len(n),
  treated = rbinom(n, 1, 0.4),
  age = rnorm(n, 50, 10),
  income = rnorm(n, 50000, 15000)
)
result <- ps_match(treated ~ age + income, data = data, treatment = "treated")
print(result)
```

---

result_accessors *Certificate, Potentials and Unmatched Units of a Result*

Description

Accessors for the parts of a matching or assignment result that later analyses read.

Usage

```
certificate(x, ...)

potentials(x, ...)

unmatched_units(x, ...)

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

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

## S3 method for class 'cardinality_certificate'
glance(x, ...)

## S3 method for class 'assignment_certificate'
glance(x, ...)

## S3 method for class 'flow_certificate'
glance(x, ...)
```

Arguments

x, object	A result of match_couples(), full_match(), cardinality_match(), cem_match(), subclass_match(), match_path(), assignment() or assignment_duals(); for glance(), a certificate from verify_assignment() or verify_flow().
...	Ignored.

Details

- certificate() returns the checked optimality certificate, an assignment_certificate or flow_certificate (see verify_assignment()), or NULL when none was computed. A matching carries one when it was solved with certify = TRUE or in implicit mode. Cardinality matching returns a cardinality_certificate recording its completed search and checked bounds. Its summary exposes the size, balance, and distance stages and the constraint report.
- potentials() returns the dual potentials as a list with elements left and right (named by unit id) for a matching, or u and v for an assignment of a cost matrix; NULL when the solver returned none.
- unmatched_units() returns the ids of the unmatched units as a list with elements left and right, or the unmatched row indices of an assignment.

glance() on a certificate returns one row with certified, arithmetic, primal_objective, dual_objective, duality_gap and, for an assignment certificate, max_suboptimality.

Value

See Details.

Examples

```
set.seed(1)
cost <- matrix(runif(16), 4)
res <- assignment_duals(cost)
potentials(res)
certificate(assignment(cost, certify = TRUE))
```

---

sinkhorn *Entropic Optimal Transport by Sinkhorn-Knopp Iteration*

Description

Compute an entropy-regularized optimal transport plan using the 'Sinkhorn-Knopp' algorithm. Unlike other LAP solvers that return a hard 1-to-1 assignment, this returns a soft assignment (doubly stochastic matrix).

Usage

```
sinkhorn(
  cost,
  lambda = 10,
  tol = 1e-09,
  max_iter = 1000,
  r_weights = NULL,
  c_weights = NULL
)

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

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

## S3 method for class 'sinkhorn_result'
as.matrix(x, ...)
```

Arguments

cost	Numeric matrix of transport costs. NA or Inf entries are treated as very high cost (effectively forbidden).
lambda	Regularization parameter (default 10). Higher values produce sharper (more deterministic) transport plans; lower values produce smoother distributions. Typical range: 1-100.
tol	Convergence tolerance (default 1e-9).
max_iter	Maximum iterations (default 1000).
r_weights	Optional numeric vector of row marginals (source distribution). Default is uniform. Will be normalized to sum to 1.
c_weights	Optional numeric vector of column marginals (target distribution). Default is uniform. Will be normalized to sum to 1.
x, object	A sinkhorn_result.
...	Ignored.

Details

The 'Sinkhorn-Knopp' algorithm solves the entropy-regularized optimal transport problem:

$$P^* = \arg\min_P \langle C, P \rangle - \frac{1}{\lambda} H(P)$$

subject to row sums = r_weights and column sums = c_weights.

The entropy term H(P) encourages spread in the transport plan. As lambda -> Inf, the solution approaches the standard (unregularized) optimal transport.

Key differences from standard LAP solvers:

- Returns a soft assignment (probabilities) not a hard 1-to-1 matching
- Supports unequal marginals (weighted distributions)
- Differentiable, making it useful in ML pipelines
- Very fast: O(n^2) per iteration with typically O(1/tol^2) iterations

Use `sinkhorn_to_assignment()` to round the soft assignment to a hard matching.

Value

A `sinkhorn_result` list with elements:

- `transport_plan` — numeric matrix, the optimal transport plan P. Row sums approximate r_weights, column sums approximate c_weights.
- `cost` — the transport cost <C, P> (without entropy term).
- u, v — scaling vectors (P = diag(u) * K * diag(v) where K = exp(-lambda*C)).
- `converged` — logical, whether the algorithm converged.
- `iterations` — number of iterations used.
- `lambda` — the regularization parameter used.

`as.matrix()` returns the transport plan. `summary()` returns a `summary.sinkhorn_result` with convergence, transported mass, support size, and the observed row and column marginals.

References

Cuturi, M. (2013). 'Sinkhorn Distances': Lightspeed Computation of Optimal Transport. *Advances in Neural Information Processing Systems*, 26.

See Also

`assignment()` for hard 1-to-1 matching, `sinkhorn_to_assignment()` to round soft assignments.

Examples

```
cost <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow = 3, byrow = TRUE)

# Soft assignment with default parameters
result <- sinkhorn(cost)
print(round(result$transport_plan, 3))

# Sharper assignment (higher lambda)
result_sharp <- sinkhorn(cost, lambda = 50)
print(round(result_sharp$transport_plan, 3))

# With custom marginals (more mass from row 1)
result_weighted <- sinkhorn(cost, r_weights = c(0.5, 0.25, 0.25))
print(round(result_weighted$transport_plan, 3))

# Round to hard assignment
hard_match <- sinkhorn_to_assignment(result)
print(hard_match)
```

`sinkhorn_to_assignment` *Round a Sinkhorn Transport Plan to an Assignment*

Description

Convert a soft transport plan from `sinkhorn()` to a hard 1-to-1 assignment using greedy rounding.

Usage

```
sinkhorn_to_assignment(result)
```

Arguments

`result`	Either a result from `sinkhorn()` or a transport plan matrix.

Details

Greedy rounding iteratively assigns each row to its most probable column, ensuring no column is assigned twice. This may not give the globally optimal hard assignment; for that, use the transport plan as a cost matrix with `assignment()`.

Value

Integer vector with one 1-based column assignment per row.

See Also

`sinkhorn()`

Examples

```
cost <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow = 3, byrow = TRUE)
result <- sinkhorn(cost, lambda = 20)
hard_match <- sinkhorn_to_assignment(result)
print(hard_match)
```

`solver_status_values` *Solver Status Values*

Description

The closed set of values a couplr solve result's `status` field can take.

Usage

```
solver_status_values()
```

Details

`"optimal"` The solver terminated on its own optimality condition and every row it was asked to match is matched.

`"partial"` Feasible, but fewer rows are matched than requested, under a maximum-cardinality-then-minimum-cost objective.

`"infeasible"` No assignment of the requested cardinality exists.

`"eps_optimal"` An auction solve terminated at a caller-supplied epsilon. The result is within `n * epsilon` of optimal, which is a bound, not a certificate.

`"iteration_limit"` The solver stopped on an iteration cap rather than on optimality. The solution is feasible; its optimality is unproven.

`"interrupted"` The solver stopped on a time budget or a user interrupt, part way through. What it had placed respects every capacity, and is short of what the problem asked it to place, so it is neither an answer nor evidence that no answer exists.

`"heuristic"` Produced by a heuristic. Optimality is neither claimed nor checked.

Value

Character vector of the permitted status values.

See Also

`verify_assignment()` to turn a status into a checked certificate.

Examples

```
solver_status_values()
```

`subclass_match` *Subclassification on Propensity Score*

Description

Divides units into K strata based on quantiles of the propensity score, then computes within-stratum weights for treatment effect estimation. This is a simple, transparent approach to propensity score adjustment that allows visual inspection of balance within each subclass.

Usage

```
subclass_match(
  formula = NULL,
  data = NULL,
  treatment = NULL,
  n_subclasses = 5L,
  ps = NULL,
  ps_model = NULL,
  estimand = "ATT"
)
```

Arguments

`formula`	Formula for propensity score model (e.g., `treatment ~ age + income`). Ignored if `ps` is provided.
`data`	Data frame containing all variables
`treatment`	Character, name of the binary treatment column (0/1)
`n_subclasses`	Integer, number of subclasses to create (default: 5). Cochran (1968) showed that 5 subclasses removes over 90\ a single covariate.
`ps`	Optional pre-computed numeric vector of propensity scores (one per row in `data`). If NULL, a logistic regression model is fit using `formula`.
`ps_model`	Optional pre-fitted `glm` object for propensity scores
`estimand`	Target estimand: `"ATT"` (default), `"ATE"`, or `"ATC"`

Details

The algorithm:

1. Estimate propensity scores via logistic regression (or use pre-computed scores)
2. Divide the propensity score distribution into K quantile-based strata
3. For each stratum, check overlap (both treated and control units present)
4. Compute within-stratum weights based on the target estimand:
   - **ATT**: Treated units get weight 1; control units get weight `n_treated_in_stratum / n_control_in_stratum`

- **ATE**: Both groups get weight proportional to stratum size relative to total sample
- **ATC**: Control units get weight 1; treated units get weight `n_control_in_stratum / n_treated_in_stratum`

Value

An S3 object of class `c("subclass_result", "couplr_result")` containing:

matched Tibble with columns `id`, `side`, `subclass`, `ps`, `weight`

subclass_summary Tibble with per-subclass statistics: counts, mean PS, and overlap status

info List with `n_subclasses`, `estimand`, `n_left`, `n_right`, `method`, `vars`

Examples

```
set.seed(42)
n <- 200
data <- data.frame(
  id = 1:n,
  age = rnorm(n, 40, 10),
  income = rnorm(n, 50000, 15000)
)
data$treatment <- rbinom(n, 1, plogis(-2 + 0.05 * data$age))
result <- subclass_match(treatment ~ age + income, data, treatment = "treatment")
print(result)
```

---

update_constraints *Update the Constraints of a Distance Object*

Description

Apply new constraints to a precomputed distance object without recomputing the underlying distances. This is useful for exploring different constraint scenarios quickly.

Usage

```
update_constraints(dist_obj, max_distance = Inf, calipers = NULL)
```

Arguments

`dist_obj`	A distance_object from `compute_distances()`
`max_distance`	Maximum allowed distance (pairs with distance > max_distance become Inf)
`calipers`	Named list of calipers, each on a column of both data frames the distance object was built from.

Details

This function creates a new distance_object with modified constraints applied to the cost matrix. The original distance_object is not modified.

Constraints:

- `max_distance`: Sets cost to Inf for pairs exceeding this threshold
- `calipers`: Per-variable restrictions (e.g., calipers = list(age = 5))

The function returns a new object rather than modifying in place, following R's copy-on-modify semantics.

Value

A new distance_object with updated cost_matrix

Examples

```
left <- data.frame(id = 1:5, age = c(25, 30, 35, 40, 45))
right <- data.frame(id = 6:10, age = c(24, 29, 36, 41, 44))
dist_obj <- compute_distances(left, right, vars = "age")

# Apply constraints
constrained <- update_constraints(dist_obj, max_distance = 2)
result <- match_couples(constrained)
```

---

verify_assignment *Verify That an Assignment Is Optimal*

Description

Checks a solved assignment against the linear-programming optimality conditions and returns the result of each check. Unlike the `status` field on a solve result, which records what the solver terminated on, this is a statement about the matching: `certified_optimal` is `TRUE` only when every condition holds.

Usage

```
verify_assignment(
  x,
  cost = NULL,
  duals = NULL,
  maximize = FALSE,
  tol = 1e-09,
  arithmetic = c("auto", "exact", "double")
)

## S3 method for class 'assignment_certificate'
```

```
print(x, ...)

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

Arguments

x	An assignment_certificate object.
cost	Numeric cost matrix the assignment was computed on, or a lazy cost specification from compute_distances(). Required unless x already carries one.
duals	Optional list with elements u and v giving row and column potentials in the orientation of cost. Overrides any duals on x. Either element may be a matrix with one row per potential, as exact_u and exact_v are, in which case each potential is the exact sum of its row.
maximize	Logical; whether the assignment maximized rather than minimized. Defaults to FALSE.
tol	Numeric tolerance for the feasibility and slackness comparisons of a numerical certificate, ignored by an exact one. The duality-gap comparison scales it by the magnitude of the objective, since an absolute tolerance on a sum of many terms is not reachable in double precision.
arithmetic	One of "auto", "exact" or "double". "auto", the default, reports the exact conclusion when the exact conditions hold and the numerical one otherwise, which is the strongest statement the instance supports; the exact conditions imply the numerical ones at any non-negative tol, so "auto" certifies nothing "double" would refuse. "exact" refuses to fall back, so certified_optimal is then TRUE only on an exact certificate. "double" decides everything within tol.
...	Ignored.
object	A certificate.

Value

An object of class assignment_certificate, a list with elements:

- certified_optimal — logical, the conclusion. TRUE only when every condition below holds.
- arithmetic — "exact" or "double", the arithmetic the conclusion was reached in.
- exact_certificate — logical; whether the conditions hold in exact arithmetic. arithmetic = "double" does not ask the question, and reports FALSE here and in exact_available.
- exact_available — logical; whether the exact question was asked of every condition. It is FALSE under arithmetic = "double", and on a certificate assembled from a scan that priced part of the problem against a tolerance rather than evaluating it, which is what the edge-generation loop hands back.
- n_exact_violations, n_exact_untight — pairs failing exact dual feasibility, and matched pairs not exactly tight, under the duals given. Potentials recovered from the matching prove the matching and leave these counts describing the duals that missed.

- exact_duals_source — which potentials decided the exact certificate: "solver" for the duals as supplied or solved, "supplied" for expansions passed in duals, "recovered" for potentials recovered from the matching, and "none" when no exact certificate was reached.
- exact_u, exact_v — on an exact certificate, the row and column potentials that decided it, one row per potential whose sum is its exact value (a single column when the solver's doubles decided it); NULL when there is no exact certificate.
- structurally_valid_matching — logical; no column claimed twice, no forbidden pair matched, no index out of range. Unmatched rows are permitted, so this holds for a partial matching.
- all_rows_matched — logical; every row of the short side holds a column.
- primal_feasible — logical; both of the two above. The primal constrains every row of the short side to hold exactly one pair, so a partial matching is a valid matching and not a feasible solution, and no conclusion rests on it. primal_objective is still reported for one, since an unmatched row costs nothing and leaves the sum meaningful.
- dual_feasible — logical; c_ij - u_i - v_j >= -tol over every admissible pair, and, when there are more columns than rows, v_j <= tol for every column.
- complementary_slackness — logical; both halves.
- cs_matched_tight, cs_unmatched_free — the two halves separately.
- primal_objective, dual_objective, duality_gap — numeric.
- max_suboptimality — numeric; the most any feasible solution can beat this one by, in the cost unit. It adds to the duality gap the slack the dual conditions were allowed: n_rows times the depth the reduced costs were permitted below zero, plus, where the sign condition applies, n_cols times the height the column duals were permitted above it, plus an envelope for each objective's own rounding. Compensated summation buys back the accumulation error rather than removing it, so each sum is charged (2u + gamma_n^2) times the sum of its terms' magnitudes and the assembly is rounded outward at every step. The number is an upper bound in double arithmetic and not an estimate of one. An exact certificate proves the matching optimal and reports zero. NA when primal_feasible is FALSE: the quantity is what a feasible solution can beat this one by, and there is no answer for a candidate that is not one.
- certified_reduced_cost_floor — numeric; the lower bound proved for the reduced cost of every admissible pair, the ones never evaluated included. Equal to min_reduced_cost when every pair was visited, and below it when a pruning pricer proved only its own threshold.
- min_reduced_cost, worst_i, worst_j — the most violated pair, if any.
- max_matched_slack, max_v_unmatched, max_v — the quantities the slackness and sign conditions bound.
- n_matched, n_rows, n_cols, transposed, tolerance.

Invisibly returns x.

What the certificate proves

The conditions are decided in one of two arithmetics, and the certificate says which one it used.

An **exact certificate** decides every condition in exact arithmetic, with no tolerance anywhere. It proves that the matching attains the minimum total cost of the cost matrix as supplied. The three quantities in each condition are IEEE doubles, and a double is a rational number, so c_ij - u_i -

v_j has an exact sign; the check evaluates that sign exactly rather than reading the sign of a rounded difference. A cost matrix that is itself a rounding of something else — Mahalanobis distances, say — is still certified as the matrix it is, which is the problem the solver was given.

A **numerical certificate** decides the same conditions within tol. It establishes optimality up to that tolerance and no further. It is what is available when the potentials are not exactly optimal for the matrix: a potential computed as c_ij - v_j misses exact tightness by the rounding of that subtraction, and a matched arc off by one unit in the last place is enough to put the exact conclusion out of reach.

The exact certificate does not depend on the solver's potentials being exact. Optimal duals are sums and differences of cost entries, and a double often cannot hold one: on computed costs such as Euclidean or Mahalanobis distances the solver's potentials miss exact tightness on some matched arc by a unit in the last place. When the duals given pass the numerical reading but not the exact one, exact potentials are recovered from the matching itself, as shortest-path distances over the cost entries held in exact multi-component arithmetic (Shewchuk 1997), and the conditions are decided on those. A matching that is not optimal has no such potentials, so the recovery cannot certify one; it finds the improving cycle or path instead, and the numerical reading stands. Duals that fail even the numerical reading certify nothing, and the check fails on them. The recovery costs one pass over the admissible pairs.

Exact potentials generally do not fit in one double each, so they are returned as exact_u and exact_v, matrices with one row per potential whose row sums are the exact values. Summing each row in rational arithmetic (for example with gmp::as.bigq) reproduces them, and passing them back as duals has the conditions checked again from the cost matrix alone, so the certificate can be re-checked without trusting this code.

The check needs dual variables. If x carries them (as assignment_duals() results do), they are used. Otherwise they are obtained by solving cost with assignment_duals(), which costs a second solve. Either way the duals are verified, not trusted: dual feasibility is checked over every admissible pair, so duals that do not certify anything cause the verification to fail rather than pass.

Optimal duals are shared by all optimal solutions of a linear program, so a matching from one solver can be certified against duals from another. That is what makes it possible to certify solvers that return no duals of their own.

See Also

assignment(), assignment_duals(), solver_status_values()

Examples

```
set.seed(1)
cost <- matrix(runif(100), 10, 10)
verify_assignment(assignment(cost), cost)

# A rectangular problem, where the condition on unmatched columns bites.
# Passing the duals result reuses its duals instead of solving again.
rect <- matrix(runif(120), 6, 20)
verify_assignment(assignment_duals(rect), rect)

# Integer costs carry an integer optimal dual solution, so the conditions
# hold with no tolerance and the certificate is exact.
```

```
int_cost <- matrix(sample(1:100, 64, replace = TRUE), 8, 8)
cert <- verify_assignment(assignment(int_cost), int_cost)
cert$arithmetic
```

---

verify_flow *Verify That a Flow Is Optimal*

Description

Checks a flow and a set of node potentials against the linear-programming optimality conditions for the minimum-cost flow problem, and returns the result of each check. Unlike the `status` field on a solve result, which records what the solver terminated on, this is a statement about the flow: `certified_optimal` is `TRUE` only when every condition holds.

Usage

```
verify_flow(
  x,
  problem = NULL,
  potential = NULL,
  tol = 1e-09,
  arithmetic = c("auto", "exact", "double")
)

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

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

Arguments

`x`	A `flow_certificate` object.
`problem`	The flow problem the flow belongs to: a `couplr_flow_problem` or a list with elements `n_nodes`, `supply` and `arcs`, where `arcs` is a data frame with columns `tail`, `head`, `lower`, `upper` and `cost` and node ids run from 1. Required unless `x` already carries one.
`potential`	Optional numeric vector of node potentials, one per node. Overrides any potentials on `x`. It may also be a matrix with one row per node, as `exact_potential` is, in which case each potential is the exact sum of its row.
`tol`	Relative tolerance for the feasibility and slackness comparisons. Each arc scales it by the largest of its cost and its two potentials, and the duality-gap comparison scales it by the magnitude of the objective, since a threshold below the resolution of the arithmetic that produced a number is not one a correct answer can meet. Ignored by an exact certificate.

arithmetic	One of "auto", "exact" or "double", as for verify_assignment(). "auto", the default, reports the exact conclusion when the exact conditions hold and the numerical one otherwise.
...	Ignored.
object	A certificate.

Details

The conditions are decided in one of two arithmetics, as they are by verify_assignment(), and the certificate says which. The **exact** reading decides the sign of every residual arc's reduced cost with no tolerance: the flow is integral, so conservation and the bounds are exact already, and the duality gap, being the sum of the slackness violations weighted by |cbar|, is then exactly zero. When the potentials supplied pass the numerical reading but miss exactness, which on computed costs they usually do by a unit in the last place, exact potentials are recovered from the flow itself as the shortest-path distances of its residual graph, held in exact multi-component arithmetic. A flow that is not optimal has none, because its residual graph carries a negative cycle, so the recovery cannot certify one, and potentials that fail even the numerical reading certify nothing. The **numerical** reading makes the comparisons within a relative tolerance and establishes optimality to that tolerance.

The problem is

```
min  sum_a cost(a) f(a)
s.t. (flow out of v) - (flow into v) = supply(v)   at every node v
     lower(a) <= f(a) <= upper(a)                  on every arc a
```

and, writing cbar(a) = cost(a) + pi(tail(a)) - pi(head(a)) for the reduced cost under the potentials pi, the conditions checked are primal feasibility, cbar(a) >= -tol(a) on every arc that can still take flow, and cbar(a) <= tol(a) on every arc carrying more than its lower bound. Their objective form is checked too: the duality gap is the sum of the slackness violations weighted by |cbar|, so it is where a violation too small to trip the per-arc tolerance still accumulates.

tol is relative. Each arc is compared against

```
tol(a) = tol * max(1, |cost(a)|, |pi(tail(a))|, |pi(head(a))|)
```

because cbar(a) is computed from those three numbers, and its last bits are worth the largest of them times the machine epsilon. A design that stacks lexicographic tier weights, as cardinality_match() does to rank cardinality above balance above distance, reaches potentials in the millions, where one unit in the last place is around 1e-9 and an exactly optimal flow cannot meet an absolute 1e-9. The scale never falls below 1, so a problem whose costs and potentials are of order 1 is checked against tol itself. The widest tolerance any comparison used is reported as dual_tolerance.

The check needs potentials. If x carries them, as a solve result does, they are used. Otherwise they are obtained by solving problem, which costs a second solve. Either way they are verified, not trusted: an arc priced below the tolerance makes the verification fail rather than pass.

Optimal potentials are shared by all optimal solutions of a linear program, so a flow from one solver can be certified against potentials from another.

Value

An object of class `flow_certificate`, a list with elements:

- `certified_optimal` - logical, the conclusion. `TRUE` only when every condition below holds.
- `primal_feasible` - logical; every arc inside its bounds and every node's net flow equal to its supply.
- `n_capacity_violations`, `n_conservation_violations`, `max_conservation_error` - what primal feasibility failed on.
- `dual_feasible` - logical; no arc that can still take flow prices below its own `-tol(a)`.
- `complementary_slackness` - logical; no arc above its lower bound prices above its own `tol(a)`.
- `n_cs_violations`, `min_residual_reduced_cost`, `worst_arc` - the smallest reduced cost over the residual graph and the arc attaining it, which is a violation when it falls below that arc's `-tol(a)`. `worst_arc` is 0 when no arc can either take or give up flow.
- `dual_tolerance` - the widest `tol(a)` any comparison was made against, so the verdict names the resolution it was reached at.
- `primal_objective`, `dual_objective`, `duality_gap` - numeric.
- `arithmetic` - `"exact"` or `"double"`, the arithmetic the conclusion was reached in.
- `exact_available`, `exact_certificate` - whether the exact question was asked, and whether its conditions hold.
- `n_exact_violations` - residual arcs whose reduced cost has the wrong sign exactly, under the potentials given.
- `exact_potentials_source` - which potentials decided the exact certificate: `"solver"`, `"supplied"`, `"recovered"`, or `"none"`.
- `exact_potential` - on an exact certificate, the node potentials that decided it, one row per node whose sum is its exact value; `NULL` when there is no exact certificate.
- `tolerance` - the relative `tol` as supplied.

Invisibly returns `x`.

See Also

`verify_assignment()`, `solver_status_values()`

Examples

```
# Two supply nodes shipping to two demand nodes, stated directly as a flow.
prob <- list(
  n_nodes = 4,
  supply  = c(2, 1, -2, -1),
  arcs = data.frame(
    tail  = c(1, 1, 2, 2),
    head  = c(3, 4, 3, 4),
    lower = c(0, 0, 0, 0),
    upper = c(2, 2, 2, 2),
    cost  = c(1, 3, 2, 1)
```

```
  )
)

# Both of node 1's units go to node 3, node 2's unit goes to node 4.
verify_flow(c(2, 0, 0, 1), prob, potential = c(0, 0, 1, 1))

# The same flow against potentials that certify nothing.
verify_flow(c(2, 0, 0, 1), prob, potential = c(0, 0, 0, 0))
```

Index

∗ **datasets**
hospital_staff, 38

animated_methods, 3
as.data.frame.assignment_duals_result *(*assignment_duals*)*, 9
as.data.frame.couplr_path *(*match_path*)*, 61
as.data.frame.couplr_result *(*plot.matching_result*)*, 68
as.data.frame.sensitivity_analysis *(*autoplot.sensitivity_analysis*)*, 13
as.matrix.distance_object *(*compute_distances*)*, 26
as.matrix.sinkhorn_result *(*sinkhorn*)*, 73
as_assignment_matrix, 11
as_matchit, 11
as_matchit(), *15*, *56*, *68*
assignment, 3
assignment(), *10*, *20*, *35*, *44, 45*, *47*, *73*, *75, 76*, *83*
assignment_duals, 9
assignment_duals(), *8*, *58*, *73*, *83*
augment.couplr_result *(*join_matched*)*, 41
autoplot.balance_diagnostics *(*balance_diagnostics*)*, 15
autoplot.matching_result *(*plot.matching_result*)*, 68
autoplot.sensitivity_analysis, 13

bal.tab.couplr_result, 15
bal.tab.subclass_result *(*bal.tab.couplr_result*)*, 15
balance_diagnostics, 15
balance_diagnostics(), *24*, *56*, *62*, *68*
balance_table, 18
bottleneck_assignment, 19
bottleneck_assignment(), *8*

cardinality_match, 20
cardinality_match(), *68*, *73*
cem_match, 25
cem_match(), *68*, *73*
certificate *(*result_accessors*)*, 72
certificate(), *58*, *68*
compute_distances, 26
compute_distances(), *9*, *81*
couplr_result *(*plot.matching_result*)*, 68
cut, *26*

diagnose_distance_matrix, 29

estimate_dense_matrix_mb, 30
estimate_dense_solve_mb, 30
estimate_dense_solve_mb(), *30*
example_costs, 31, *33*
example_df, *32*, 33
explain_dispatch, 34
explain_dispatch(), *7*

full_match, 36
full_match(), *68*, *73*

get_method_used *(*assignment*)*, 3
get_total_cost *(*assignment*)*, 3
glance.assignment_certificate *(*result_accessors*)*, 72
glance.cardinality_certificate *(*result_accessors*)*, 72
glance.couplr_result *(*plot.matching_result*)*, 68
glance.flow_certificate *(*result_accessors*)*, 72

hospital_staff, 38

is_distance_object *(*compute_distances*)*, 26
is_lap_solve_batch_result *(*lap_solve_batch*)*, 48

is_lap_solve_kbest_result
(lap_solve_kbest), 50
is_lap_solve_result (assignment), 3

join_matched, 41
join_matched(), *56*, *68*

lap_animate, 44
lap_animate(), *3*
lap_solve, *32, 33*, *40*, 45
lap_solve(), *8*, *20*, *44, 45*
lap_solve_batch, *33*, *40*, 48
lap_solve_kbest, 50
lap_solve_kbest(), *8*
lap_solve_line_metric, 52
lap_solve_line_metric(), *5*
lap_solvers, 47

match_couples, *40*, 55
match_couples(), *13*, *24*, *54*, *62*, *68, 69*, *71*, *73*
match_data, 59
match_data(), *56*, *68*
match_path, 61
match_path(), *62*, *73*
matchmaker, 53

parallelly::availableCores(), *49*
pixel_morph, 63
pixel_morph_animate, *64, 65*, 65
plot.balance_diagnostics
(balance_diagnostics), 15
plot.couplr_path (match_path), 61
plot.matching_result, 68
plot.sensitivity_analysis
(autoplot.sensitivity_analysis), 13
potentials (result_accessors), 72
potentials(), *68*
preprocess_matching_vars, 69
print.assignment_certificate
(verify_assignment), 80
print.assignment_duals_result
(assignment_duals), 9
print.balance_diagnostics
(balance_diagnostics), 15
print.cardinality_certificate
(result_accessors), 72
print.cardinality_report(), *23*
print.couplr_path (match_path), 61
print.couplr_result
(plot.matching_result), 68
print.dispatch_explanation
(explain_dispatch), 34
print.distance_diagnostics
(diagnose_distance_matrix), 29
print.distance_object
(compute_distances), 26
print.flow_certificate (verify_flow), 84
print.lap_solve_batch_result
(lap_solve_batch), 48
print.lap_solve_kbest_result
(lap_solve_kbest), 50
print.lap_solve_result (assignment), 3
print.matchmaker_result (matchmaker), 53
print.pixel_morph_animation
(pixel_morph_animate), 65
print.preprocessing_result
(preprocess_matching_vars), 69
print.sensitivity_analysis
(autoplot.sensitivity_analysis), 13
print.sinkhorn_result (sinkhorn), 73
print.variable_health
(preprocess_matching_vars), 69
ps_match, 71

register_lap_solver (lap_solvers), 47
result_accessors, 72

sensitivity_analysis
(autoplot.sensitivity_analysis), 13
sensitivity_analysis(), *56*
sinkhorn, 73
sinkhorn(), *8*, *76*
sinkhorn_to_assignment, 76
sinkhorn_to_assignment(), *75*
solver_status_values, *38*, 77
solver_status_values(), *7*, *58*, *83*, *86*
subclass_match, 78
subclass_match(), *68*, *73*
summary.assignment_certificate
(verify_assignment), 80
summary.assignment_duals_result
(assignment_duals), 9
summary.balance_diagnostics
(balance_diagnostics), 15

summary.bottleneck_result
(bottleneck_assignment), 19
summary.cardinality_certificate
(result_accessors), 72
summary.couplr_path (match_path), 61
summary.couplr_result
(plot.matching_result), 68
summary.dispatch_explanation
(explain_dispatch), 34
summary.distance_diagnostics
(diagnose_distance_matrix), 29
summary.distance_object
(compute_distances), 26
summary.flow_certificate (verify_flow),
84
summary.lap_line_metric_result
(lap_solve_line_metric), 52
summary.lap_solve_batch_result
(lap_solve_batch), 48
summary.lap_solve_kbest_result
(lap_solve_kbest), 50
summary.lap_solve_result (assignment), 3
summary.matchmaker_result (matchmaker),
53
summary.pixel_morph_animation
(pixel_morph_animate), 65
summary.preprocessing_result
(preprocess_matching_vars), 69
summary.sensitivity_analysis
(autoplot.sensitivity_analysis),
13
summary.sinkhorn_result (sinkhorn), 73
summary.variable_health
(preprocess_matching_vars), 69

tidy.couplr_result
(plot.matching_result), 68

unmatched_units (result_accessors), 72
unmatched_units(), *68*
update_constraints, 79

verify_assignment, 80
verify_assignment(), *6–10*, *47*, *58*, *59*, *73*,
77, *85*, *86*
verify_flow, *38*, 84
verify_flow(), *73*