

# Package ‘reclin2’

February 9, 2024

**Type** Package

**Title** Record Linkage Toolkit

**Version** 0.5.0

**Description** Functions to assist in performing probabilistic record linkage and deduplication: generating pairs, comparing records, em-algorithm for estimating m- and u-probabilities (I. Fellegi & A. Sunter (1969) <[doi:10.1080/01621459.1969.10501049](https://doi.org/10.1080/01621459.1969.10501049)>, T.N. Herzog, F.J. Scheuren, & W.E. Winkler (2007), ``Data Quality and Record Linkage Techniques'', ISBN:978-0-387-69502-0), forcing one-to-one matching. Can also be used for pre- and post-processing for machine learning methods for record linkage. Focus is on memory, CPU performance and flexibility.

**BugReports** <https://github.com/djvanderlaan/reclin2/issues>

**URL** <https://github.com/djvanderlaan/reclin2>

**Depends** data.table, R (>= 3.6.0)

**Imports** stringdist, stats, utils, lpSolve, Rcpp, parallel

**Suggests** simplermardown

**LinkingTo** Rcpp

**VignetteBuilder** simplermardown

**License** GPL-3

**LazyLoad** yes

**Encoding** UTF-8

**RoxygenNote** 7.2.3

**NeedsCompilation** yes

**Author** Jan van der Laan [aut, cre] (<<https://orcid.org/0000-0002-0693-1514>>)

**Maintainer** Jan van der Laan <r@eoos.dds.nl>

**Repository** CRAN

**Date/Publication** 2024-02-09 08:30:02 UTC

**R topics documented:**

add_from_x . . . . .	2
cluster_call . . . . .	3
cluster_collect . . . . .	4
cluster_modify_pairs . . . . .	5
cluster_pair . . . . .	6
cluster_pair_blocking . . . . .	7
cluster_pair_minsim . . . . .	9
cmp_identical . . . . .	10
compare_pairs.cluster_pairs . . . . .	12
compare_vars.cluster_pairs . . . . .	13
deduplicate_equivalence . . . . .	15
get_inspect_pairs . . . . .	16
greedy . . . . .	17
link . . . . .	17
linkexample1 . . . . .	19
match_n_to_m . . . . .	19
merge_pairs.cluster_pairs . . . . .	20
pair . . . . .	21
pair_blocking . . . . .	22
pair_minsim . . . . .	23
predict.problink_em . . . . .	24
problink_em . . . . .	26
score_simple.cluster_pairs . . . . .	27
select_greedy.cluster_pairs . . . . .	29
select_threshold.cluster_pairs . . . . .	33
select_unique.cluster_pairs . . . . .	34
summary.problink_em . . . . .	36
tabulate_patterns.cluster_pairs . . . . .	37
town_names . . . . .	38
<b>Index</b>	<b>39</b>

---

 add\_from\_x

*Add a variable from one of the data sets to pairs*


---

**Description**

Add a variable from one of the data sets to pairs

**Usage**

```
add_from_x(pairs, variable, new_variable = variable, ...)
```

```
add_from_y(pairs, variable, new_variable = variable, ...)
```

**Arguments**

pairs	<a href="#">data.table</a> with pairs. Should contain the columns .x and .y.
variable	name of the variable that should be added
new_variable	optional variable name of the new variable in pairs. When omitted variable is used.
...	other parameters are passed on to compare_vars. Especially inplace, x and y might be of interest.

**Value**

Returns the pairs with the column added. When inplace = TRUE pairs is returned invisibly and the original pairs is modified.

---

cluster_call	<i>Call a function on each of the worker nodes and pass it the pairs</i>
--------------	--

---

**Description**

Call a function on each of the worker nodes and pass it the pairs

**Usage**

```
cluster_call(pairs, fun, ...)
```

**Arguments**

pairs	an object or type cluster_pairs as created for example by <a href="#">cluster_pair</a> .
fun	a function to call on each of the worker nodes. See details on the arguments of this function.
...	additional arguments are passed on to fun.

**Details**

The function will have to accept the following arguments as its first three arguments:

**pairs** the data.table with the pairs of the worker node.

**x** a data.table with the portion of x present on the worker node.

**y** a data.table with y.

**Value**

The function will return a list with for each worker the result of the function call. When the functions return NULL the result is returned invisibly. Because the result is returned to main node, make sure you don't accidentally return all pairs. If you don't want to return anything end your function with NULL.

**Examples**

```

# Generate some pairs
library(parallel)
data("linkexample1", "linkexample2")
cl <- makeCluster(2)

pairs <- cluster_pair(cl, linkexample1, linkexample2)
compare_pairs(pairs, c("lastname", "firstname", "address", "sex"))

# Add a new column to pairs
cluster_call(pairs, function(pairs, ...) {
  pairs[, name := firstname & lastname]
  # we don't want to return the pairs; so make sure to return something
  # else
  NULL
})

# Get the number of pairs on each node
lengths <- cluster_call(pairs, function(pairs, ...) {
  nrow(pairs)
})
lengths <- unlist(lengths)
lengths

# Cleanup
stopCluster(cl)

```

---

cluster_collect	<i>Collect pairs from cluster nodes</i>
-----------------	---

---

**Description**

Collect pairs from cluster nodes

**Usage**

```
cluster_collect(pairs, select = NULL, clear = FALSE)
```

**Arguments**

pairs	an object or type cluster_pairs as created for example by <a href="#">cluster_pair</a> .
select	the name of a logical column that is used to select the pairs that will be collected
clear	remove the pairs from the cluster nodes

**Value**

Returns an object of type pairs which is a data.table. This object can be used as a regular (non-cluster) set of pairs

**Examples**

```

library(parallel)
data("linkexample1", "linkexample2")
cl <- makeCluster(2)

pairs <- cluster_pair(cl, linkexample1, linkexample2)
local_pairs <- cluster_collect(pairs, clear = FALSE)

compare_pairs(pairs, c("lastname", "firstname", "address", "sex"))
model <- problink_em(~ lastname + firstname + address + sex, data = pairs)
predict(model, pairs, type = "mpost", add = TRUE, binary = TRUE)
# Select pairs with a mpost > 0.5
select_threshold(pairs, "selected", "mpost", 0.5)
# Collect the selected pairs
local_pairs <- cluster_collect(pairs, "selected")

stopCluster(cl)

```

---

`cluster_modify_pairs` *Call a function on each of the worker nodes to modify the pairs on the node*

---

**Description**

Call a function on each of the worker nodes to modify the pairs on the node

**Usage**

```
cluster_modify_pairs(pairs, fun, ..., new_name = NULL)
```

**Arguments**

<code>pairs</code>	an object or type <code>cluster_pairs</code> as created for example by <code>cluster_pair</code> .
<code>fun</code>	a function to call on each of the worker nodes. See details on the arguments of this function.
<code>...</code>	additional arguments are passed on to <code>fun</code> .
<code>new_name</code>	name of new object to assign the pairs to on the cluster nodes.

**Details**

The function will have to accept the following arguments as its first three arguments:

**pairs** the data.table with the pairs of the worker node.  
**x** a data.table with the portion of x present on the worker node.  
**y** a data.table with y.

The function should either return a `data.table` with the new pairs, or `NULL`. When a `data.table` is returned this values will replace the pairs when `new_name` is missing or create new pairs in the environment `new_name`. When the function returns `NULL` it is assumed that the function modified the pairs by reference (e.g. using `pairs[, new_var := new_val]`). Note that this also means that `new_name` is ignored.

### Value

Will return a `cluster_pairs` object. When `new_name` is not given it will return the input pairs invisibly. Otherwise it will return a new `cluster_pairs` object.

### Examples

```
# Generate some pairs
library(parallel)
data("linkexample1", "linkexample2")
cl <- makeCluster(2)

pairs <- cluster_pair(cl, linkexample1, linkexample2)
compare_pairs(pairs, c("lastname", "firstname", "address", "sex"))

# Create a new set of pairs containing a random sample of the original
# pairs.
sample <- cluster_call(pairs, new_name = "sample", function(pairs, ...) {
  sel <- sample(nrow(pairs), round(nrow(pairs)*0.1))
  pairs[sel, ]
})

# Cleanup
stopCluster(cl)
```

---

cluster\_pair

*Generate all possible pairs using multiple processes*

---

### Description

Generates all combinations of records from `x` and `y`.

### Usage

```
cluster_pair(cluster, x, y, deduplication = FALSE, name = "default")
```

### Arguments

<code>cluster</code>	a cluster object as created by <code>makeCluster</code> from <code>parallel</code> or from the <code>snow</code> package.
<code>x</code>	first <code>data.frame</code>
<code>y</code>	second <code>data.frame</code> . Ignored when <code>deduplication = TRUE</code> .

deduplication generate pairs from only x. Ignore y. This is useful for deduplication of x.  
 name the name of the resulting object to create locally on the different R processes.

### Details

Generating (all) pairs of the records of two data sets, is usually the first step when linking the two data sets.

x is split into `length{cluster}` parts which are distributed over the worker nodes. y is copied to each of the nodes. On the nodes then `pair` is called. The pairs are stored in the global object `reclin_env` on the nodes in the variable name. The pairs can then be further processed using functions such as `compare_pairs`, and `tabulate_patterns`. The function `cluster_collect` collects the pairs from each of the nodes.

### Value

A object of type `cluster_pairs` which is a list containing the cluster and the name of the pairs object on the cluster nodes. For the pairs objects created on the nodes see the documentation of `pair`.

### See Also

`cluster_pair_blocking` and `cluster_pair_minsim` are other methods to generate pairs.

### Examples

```
library(parallel)
data("linkexample1", "linkexample2")
cl <- makeCluster(2)

pairs <- cluster_pair(cl, linkexample1, linkexample2)
stopCluster(cl)
```

---

`cluster_pair_blocking` *Generate pairs using simple blocking using multiple processes*

---

### Description

Generates all combinations of records from x and y where the blocking variables are equal.

### Usage

```
cluster_pair_blocking(
  cluster,
  x,
  y,
  on,
  deduplication = FALSE,
  name = "default"
)
```

## Arguments

cluster	a cluster object as created by <code>makeCluster</code> from <code>parallel</code> or from the <code>snow</code> package.
x	first data.frame
y	second data.frame. Ignored when <code>deduplication = TRUE</code> .
on	the variables defining the blocks or strata for which all pairs of x and y will be generated.
deduplication	generate pairs from only x. Ignore y. This is useful for deduplication of x.
name	the name of the resulting object to create locally on the different R processes.

## Details

Generating (all) pairs of the records of two data sets, is usually the first step when linking the two data sets. However, this often results in a too large number of records. Therefore, blocking is usually applied.

x is split into `length{cluster}` parts which are distributed over the worker nodes. y is copied to each of the nodes. On the nodes then `pair_blocking` is called. The pairs are stored in the global object `reclin_env` on the nodes in the variable `name`. The pairs can then be further processed using functions such as `compare_pairs`, and `tabulate_patterns`. The function `cluster_collect` collects the pairs from each of the nodes.

## Value

A object of type `cluster_pairs` which is a list containing the cluster and the name of the pairs object on the cluster nodes. For the pairs objects created on the nodes see the documentation of `pair`.

## See Also

`cluster_pair` and `cluster_pair_minsim` are other methods to generate pairs.

## Examples

```
library(parallel)
data("linkexample1", "linkexample2")
cl <- makeCluster(2)

pairs <- cluster_pair_blocking(cl, linkexample1, linkexample2, "postcode")
stopCluster(cl)
```



---

cluster\_pair\_minsim *Generate pairs with a minimal similarity using multiple processes*

---

### Description

Generates all combinations of records from x and y where the blocking variables are equal.

### Usage

```
cluster_pair_minsim(
  cluster,
  x,
  y,
  on,
  minsim = 0,
  on_blocking = character(0),
  comparators = list(default_comparator),
  default_comparator = cmp_identical(),
  keep_simsum = TRUE,
  deduplication = FALSE,
  name = "default"
)
```

### Arguments

cluster	a cluster object as created by <code>makeCluster</code> from <code>parallel</code> or <code>makeCluster</code> from <code>snow</code> .
x	first data.frame
y	second data.frame. Ignored when <code>deduplication = TRUE</code> .
on	the variables defining the blocks or strata for which all pairs of x and y will be generated.
minsim	minimal similarity score.
on_blocking	variables for which the pairs have to match.
comparators	named list of functions with which the variables are compared. This function should accept two vectors. Function should either return a vector or a data.table with multiple columns.
default_comparator	variables for which no comparison function is defined using comparators is compares with the function <code>default_comparator</code> .
keep_simsum	add a variable <code>minsim</code> to the result with the similarity score of the pair.
deduplication	generate pairs from only x. Ignore y. This is usefull for deduplication of x.
name	the name of the resulting object to create locally on the different R processes.

**Details**

Generating (all) pairs of the records of two data sets, is usually the first step when linking the two data sets. However, this often results in a too large number of records. `pair_minsim` will only keep pairs with a similarity score equal or larger than `minsimsim`. The similarity score is calculated by summing the results of the comparators for all variables of `on`.

`x` is split into `length{cluster}` parts which are distributed over the worker nodes. `y` is copied to each of the nodes. On the nodes then `cluster_pair_minsim` is called. The pairs are stored in the global object `reclin_env` on the nodes in the variable name. The pairs can then be further processed using functions such as `compare_pairs`, and `tabulate_patterns`. The function `cluster_collect` collects the pairs from each of the nodes.

**Value**

A object of type `cluster_pairs` which is a list containing the cluster and the name of the pairs object on the cluster nodes. For the pairs objects created on the nodes see the documentation of `pair`.

**See Also**

`cluster_pair` and `cluster_pair_blocking` are other methods to generate pairs.

**Examples**

```
library(parallel)
data("linkexample1", "linkexample2")
cl <- makeCluster(2)

# Either address or postcode has to match to keep a pair
pairs <- cluster_pair_minsim(cl, linkexample1, linkexample2,
  on = c("postcode", "address"), minsimsim = 1)
stopCluster(cl)
```

---

cmp\_identical

*Comparison functions*

---

**Description**

Comparison functions

**Usage**

```
cmp_identical()

cmp_jarowinkler(threshold = 0.95)

jaro_winkler(threshold = 0.8)
```

```
cmp_lcs(threshold = 0.8)

lcs(threshold = 0.8)

cmp_jaccard(threshold = 0.8)

jaccard(threshold = 0.8)
```

### Arguments

threshold      threshold to use for the Jaro-Winkler string distance when creating a binary result.

### Details

A comparison function should accept two arguments: both vectors. When the function is called with both arguments it should compare the elements in the first vector to those in the second. When called in this way, both vectors have the same length. What the function should return depends on the methods used to score the pairs. Usually the comparison functions return a similarity score with a value of 0 indicating complete difference and a value > 0 indicating similarity (often a value of 1 will indicate perfect similarity).

Some methods, such as [problink\\_em](#), can handle similarity scores, but also need binary values (0/FALSE = complete dissimilarity; 1/TRUE = complete similarity). In order to allow for this the comparison function is called with one argument.

When the comparison is called with one argument, it is passed the result of a previous comparison. The function should translate that result to a binary (TRUE/FALSE or 1/0) result. The result should not contain missing values.

The `jaro_winkler`, `lcs` and `jaccard` functions use the corresponding methods from [stringdist](#) except that they are transformed from a distance to a similarity score.

### Value

The functions return a comparison function (see details).

### Warning

The functions `identical`, `jaro_winkler`, `lcs` and `jaccard` are deprecated and will be removed in future versions of the package. Instead use the functions `cmp_identical`, `cmp_jarowinkler`, `cmp_lcs` and `cmp_jaccard`.

### Examples

```
cmp <- cmp_identical()
x <- cmp(c("john", "mary", "susan", "jack"),
        c("johan", "mary", "susanna", NA))
# Applying the comparison function to the result of the comparison results
# in a logical result, with NA's and values of FALSE set to FALSE
cmp(x)
```

```
cmp <- cmp_jarowinkler(0.95)
x <- cmp(c("john", "mary", "susan", "jack"),
        c("johan", "mary", "susanna", NA))
# Applying the comparison function to the result of the comparison results
# in a logical result, with NA's and values below the threshold FALSE
cmp(x)
```

---

compare\_pairs.cluster\_pairs

*Compare pairs on a set of variables common in both data sets*

---

### Description

Compare pairs on a set of variables common in both data sets

### Usage

```
## S3 method for class 'cluster_pairs'
compare_pairs(
  pairs,
  on,
  comparators = list(default_comparator),
  default_comparator = cmp_identical(),
  new_name = NULL,
  ...
)

compare_pairs(
  pairs,
  on,
  comparators = list(default_comparator),
  default_comparator = cmp_identical(),
  ...
)

## S3 method for class 'pairs'
compare_pairs(
  pairs,
  on,
  comparators = list(default_comparator),
  default_comparator = cmp_identical(),
  x = attr(pairs, "x"),
  y = attr(pairs, "y"),
  inplace = FALSE,
```

```
    ...
  )
```

### Arguments

<code>pairs</code>	<code>data.table</code> with pairs. Should contain the columns <code>.x</code> and <code>.y</code> .
<code>on</code>	character vector of variables that should be compared.
<code>comparators</code>	named list of functions with which the variables are compared. This function should accept two vectors. Function should either return a vector or a <code>data.table</code> with multiple columns.
<code>default_comparator</code>	variables for which no comparison function is defined using <code>comparators</code> is compared with the function <code>default_comparator</code> .
<code>new_name</code>	name of new object to assign the pairs to on the cluster nodes.
<code>...</code>	Ignored for now
<code>x</code>	<code>data.table</code> with one half of the pairs.
<code>y</code>	<code>data.table</code> with the other half of the pairs.
<code>inplace</code>	logical indicating whether <code>pairs</code> should be modified in place. When <code>pairs</code> is large this can be more efficient.

### Details

It is assumed the variables in `on` are present in both `x` and `y`. Variables with the same names are added to `pairs`. When the comparator returns a `data.table` multiple columns are added to `pairs`. The names of these columns are `variable` pasted together with the names of the `data.table` returned by comparator (separated by "\_").

### Value

Returns the `data.table` `pairs` with one or more columns added in case of `compare_pairs.pairs`.

In case of `compare_pairs.cluster_pairs`, `compare_pair.pairs` is called on each cluster node and the resulting `pairs` are assigned to `new_name` in the environment `reclin_env`. When `new_name` is not given (or equal to `NULL`) the original `pairs` on the nodes are overwritten.

---

```
compare_vars.cluster_pairs
  Compare pairs on given variables
```

---

### Description

Compare pairs on given variables

**Usage**

```
## S3 method for class 'cluster_pairs'
compare_vars(
  pairs,
  variable,
  on_x = variable,
  on_y = on_x,
  comparator = cmp_identical(),
  new_name = NULL,
  ...
)

compare_vars(
  pairs,
  variable,
  on_x = variable,
  on_y = on_x,
  comparator = cmp_identical(),
  ...
)

## S3 method for class 'pairs'
compare_vars(
  pairs,
  variable,
  on_x = variable,
  on_y = on_x,
  comparator = cmp_identical(),
  x = attr(pairs, "x"),
  y = attr(pairs, "y"),
  inplace = FALSE,
  ...
)
```

**Arguments**

<code>pairs</code>	<code>data.table</code> with pairs. Should contain the columns <code>.x</code> and <code>.y</code> .
<code>variable</code>	character vector with name of resulting column name that is added to pairs.
<code>on_x</code>	character vector with the column names from <code>x</code> on which to compare.
<code>on_y</code>	character vector with the column names from <code>y</code> on which to compare.
<code>comparator</code>	function with which the variables are compared. When <code>on_x</code> and <code>on_y</code> have length 1, this function should accept two vectors. Otherwise it will receive two <code>data.tables</code> . Function should either return a vector or a <code>data.table</code> with multiple columns.
<code>new_name</code>	name of new object to assign the pairs to on the cluster nodes.
<code>...</code>	Passed on to the comparator function.

x	data.table with one half of the pairs.
y	data.table with the other half of the pairs.
inplace	logical indicating whether pairs should be modified in place. When pairs is large this can be more efficient.

**Details**

When comparator returns a data.table multiple columns are added to pairs. The names of these columns are variable pasted together with the names of the data.table returned by comparator (separated by "\_").

**Value**

Returns the data.table pairs with one or more columns added.

---

deduplicate\_equivalence

*Deduplication using equivalence groups*

---

**Description**

Deduplication using equivalence groups

**Usage**

```
deduplicate_equivalence(pairs, variable, selection, x = attr(pairs, "x"))
```

**Arguments**

pairs	a pairs object, such as generated by <a href="#">pair_blocking</a>
variable	name of the variable to create in x that will contain the group labels.
selection	a logical variable with the same length as pairs has rows, or the name of such a variable in pairs. Pairs are only selected when select is TRUE. When missing it is assumed all pairs are selected.
x	the first data set; when missing attr(pairs, "x") is used.

**Value**

Returns x with a variable containing the group labels. Records with the same group label (should) correspond to the same entity.

---

get_inspect_pairs	<i>Get a subset of pairs to inspect</i>
-------------------	---

---

### Description

Get a subset of pairs to inspect

### Usage

```
get_inspect_pairs(  
  pairs,  
  variable,  
  threshold,  
  position = NULL,  
  n = 11,  
  x = attr(pairs, "x"),  
  y = attr(pairs, "y")  
)
```

### Arguments

pairs	<code>data.table</code> with pairs.
variable	name of variable to base the selection on; should be a variable with the similarity score of the pairs.
threshold	the threshold around which to select pairs. Used when position is not given.
position	select pairs around this position (based on order of variable), e.g. position = 1 will select the pairs with the highest similarity score.
n	number of pairs to select. Pairs are selected symmetric around the threshold.
x	<code>data.table</code> with one half of the pairs.
y	<code>data.table</code> with the other half of the pairs.

### Value

Returns a list with elements `pairs` with the selected pairs; `x` records from `x` corresponding to the pairs; `y` records from `y` corresponding to the pairs; `position` position of the selected pairs; index of the pairs in `pairs`.



---

greedy	<i>Greedy one-to-one matching of pairs</i>
--------	--

---

**Description**

Greedy one-to-one matching of pairs

**Usage**

```
greedy(x, y, weight, n = 1L, m = 1L, include_ties = FALSE)
```

**Arguments**

x	id's of lhs of pairs; converted to integer
y	id's of rhs of pairs; converted to integer
weight	numeric vector with weight of pair
n	an integer. Each element of x can be linked to at most n elements of y.
m	an integer. Each element of y can be linked to at most m elements of x.
include_ties	when pairs for a given record have an equal weight, should all pairs be included.

**Details**

Pairs with the highest weight are selected as long as neither the lhs as the rhs are already selected in a pair with a higher weight. When `include_ties` is TRUE all pairs are included when multiple pairs for a given record have an equal weight.

**Value**

A logical vector with the same length as x.

---

link	<i>Use the selected pairs to generate a linked data set</i>
------	---

---

**Description**

Use the selected pairs to generate a linked data set

## Usage

```
link(  
  pairs,  
  selection = NULL,  
  all = FALSE,  
  all_x = all,  
  all_y = all,  
  x = attr(pairs, "x"),  
  y = attr(pairs, "y"),  
  suffixes = c(".x", ".y"),  
  keep_from_pairs = c(".x", ".y")  
)
```

## Arguments

<code>pairs</code>	a pairs object, such as generated by <a href="#">pair_blocking</a>
<code>selection</code>	a logical variable with the same length as <code>pairs</code> has rows, or the name of such a variable in <code>pairs</code> . Pairs are only selected when <code>select</code> is TRUE. When missing <code>attr(pairs, "selection")</code> is used when available.
<code>all</code>	return all records from <code>x</code> and <code>y</code> ; even those that don't match.
<code>all_x</code>	return all records from <code>x</code> .
<code>all_y</code>	return all records from <code>y</code> .
<code>x</code>	the first data set; when missing <code>attr(pairs, "x")</code> is used.
<code>y</code>	the second data set; when missing <code>attr(pairs, "y")</code> is used.
<code>suffixes</code>	a character vector of length 2 specifying the suffixes to be used for making unique the names of columns in the result.
<code>keep_from_pairs</code>	character vector with names of variables in <code>pairs</code> that should be included in the output.

## Details

Uses the selected pairs to link the two data sets to each other. Renames variables that are in both data sets.

## Value

Returns a `data.table` containing records from `x` and `y` and `pairs`. Columns that occur both in `x` and `y` gain a suffix indicating from which data set they are.

---

linkexample1

*Tiny example dataset for probabilistic linkage*


---

**Description**

Contains fictional records of 7 persons.

**Format**

Two data frames with resp. 6 and 5 records and 6 columns.

**Details**

- id the id of the person; this contains no errors and can be used to validate the linkage.
- lastname the last name of the person; contains errors.
- firstname the first name of the persons; contains errors.
- address the address; contains errors.
- sex the sex; contains errors and missing values.
- postcode the postcode; contains no errors.

---

match\_n\_to\_m

*Force n to m matching on a set of pairs*


---

**Description**

Force n to m matching on a set of pairs

**Usage**

```
match_n_to_m(x, y, w, n = 1, m = 1)
```

**Arguments**

- |   |   |
|---|---|
| x | a vector of identifiers for each x in each pair This vector should have a unique value for each element in x.   |
| y | a vector of identifiers for each y in each pair This vector should have a unique value for each element in y.   |
| w | a vector with weights for each pair. The algorithm will try to maximise the total weight of the selected pairs. |
| n | an integer. Each element of x can be linked to at most n elements of y.   |
| m | an integer. Each element of y can be linked to at most m elements of x.   |

**Details**

The algorithm will try to select pairs in such a way each element of  $x$  is matched to at most  $n$  elements of  $y$  and that each element of  $y$  is matched at most  $m$  elements of  $x$ . It tries to select elements in such a way that the total weight  $w$  of the selected elements is maximised.

**Value**

A logical vector with the same length as  $x$  indicating the selected records.

**Examples**

```
d <- data.frame(x=c(1,1,1,2,2,3,3), y=c(1,2,3,4,5,6,7), w=1:7)
# One-to-one matching:
d[match_n_to_m(d$x, d$y, d$w), ]

# N-to-one matching:
d[match_n_to_m(d$x, d$y, d$w, n=999), ]

# One-to-m matching:
d[match_n_to_m(d$x, d$y, d$w, m=999), ]

# N-to-M matching, e.g. select all pairs
d[match_n_to_m(d$x, d$y, d$w, n=999, m=999), ]
```

---

```
merge_pairs.cluster_pairs
```

*Merge two sets of pairs into one*

---

**Description**

Merge two sets of pairs into one

**Usage**

```
## S3 method for class 'cluster_pairs'
merge_pairs(
  pairs1,
  pairs2,
  name = paste(pairs1$name, pairs2$name, sep = "+"),
  ...
)

## S3 method for class 'cluster_pairs'
rbind(...)
```

```
merge_pairs(pairs1, pairs2, ...)

## S3 method for class 'pairs'
merge_pairs(pairs1, pairs2, ...)

## S3 method for class 'pairs'
rbind(...)
```

### Arguments

<code>pairs1</code>	the first set of pairs
<code>pairs2</code>	the second set of pairs
<code>name</code>	name of new object to assign the pairs to on the cluster nodes.
<code>...</code>	for <code>rbind</code> the pairs or <code>cluster_pairs</code> objects the need to be combined; for <code>merge_pairs</code> these are passed on to other methods.

### Details

The function will give an error when the two sets of pairs have different values for `attr(pairs1, "x")` and `attr(pairs1, "y")`. When these attributes are missing the code will not generate an error; the user is then responsible for ensuring that the indices in `pairs1` and `pairs2` refer to the same datasets.

### Value

Returns a `pairs` or `cluster_pairs` object where both sets of pairs are combined. Duplicate pairs are removed.

In case of `merge_pairs.cluster_pairs`, `merge_pairs.pairs` is called on each cluster node and the resulting pairs are assigned to `name` in the environment `reclin_env`.

---

<code>pair</code>	<i>Generate all possible pairs</i>
-------------------	------------------------------------

---

### Description

Generates all combinations of records from `x` and `y`.

### Usage

```
pair(x, y, deduplication = FALSE, add_xy = TRUE)
```

### Arguments

<code>x</code>	first data.frame
<code>y</code>	second data.frame. Ignored when <code>deduplication = TRUE</code> .
<code>deduplication</code>	generate pairs from only <code>x</code> . Ignore <code>y</code> . This is usefull for deduplication of <code>x</code> .
<code>add_xy</code>	add <code>x</code> and <code>y</code> as attributes to the returned pairs. This makes calling some subsequent operations that need <code>x</code> and <code>y</code> (such as <a href="#">compare_pairs</a> easier.

**Details**

Generating (all) pairs of the records of two data sets, is usually the first step when linking the two data sets.

**Value**

A `data.table` with two columns, `.x` and `.y`, is returned. Columns `.x` and `.y` are row numbers from `data.frames .x` and `.y` respectively.

**See Also**

`pair_blocking` and `pair_minsim` are other methods to generate pairs.

**Examples**

```
data("linkexample1", "linkexample2")
pairs <- pair(linkexample1, linkexample2)
```

---

pair\_blocking

*Generate pairs using simple blocking*

---

**Description**

Generates all combinations of records from `x` and `y` where the blocking variables are equal.

**Usage**

```
pair_blocking(x, y, on, deduplication = FALSE, add_xy = TRUE)
```

**Arguments**

<code>x</code>	first <code>data.frame</code>
<code>y</code>	second <code>data.frame</code> . Ignored when <code>deduplication = TRUE</code> .
<code>on</code>	the variables defining the blocks or strata for which all pairs of <code>x</code> and <code>y</code> will be generated.
<code>deduplication</code>	generate pairs from only <code>x</code> . Ignore <code>y</code> . This is useful for deduplication of <code>x</code> .
<code>add_xy</code>	add <code>x</code> and <code>y</code> as attributes to the returned pairs. This makes calling some subsequent operations that need <code>x</code> and <code>y</code> (such as <code>compare_pairs</code> ) easier.

**Details**

Generating (all) pairs of the records of two data sets, is usually the first step when linking the two data sets. However, this often results in a too large number of records. Therefore, blocking is usually applied.

**Value**

A `data.table` with two columns, `.x` and `.y`, is returned. Columns `.x` and `.y` are row numbers from `data.frames .x` and `.y` respectively.

**See Also**

`pair` and `pair_minsim` are other methods to generate pairs.

**Examples**

```
data("linkexample1", "linkexample2")
pairs <- pair_blocking(linkexample1, linkexample2, "postcode")
```

---

pair_minsim	<i>Generate pairs with a minimal similarity</i>
-------------	---

---

**Description**

Generates all combinations of records from `x` and `y` where the blocking variables are equal.

**Usage**

```
pair_minsim(
  x,
  y,
  on,
  minsim = 0,
  on_blocking = character(0),
  comparators = list(default_comparator),
  default_comparator = cmp_identical(),
  keep_simsum = TRUE,
  deduplication = FALSE,
  add_xy = TRUE
)
```

**Arguments**

<code>x</code>	first <code>data.frame</code>
<code>y</code>	second <code>data.frame</code> . Ignored when <code>deduplication = TRUE</code> .
<code>on</code>	the variables defining on which the pairs of records from <code>x</code> and <code>y</code> are compared.
<code>minsim</code>	minimal similarity score.
<code>on_blocking</code>	variables for which the pairs have to match.
<code>comparators</code>	named list of functions with which the variables are compared. This function should accept two vectors. Function should either return a vector or a <code>data.table</code> with multiple columns.

default_comparator	variables for which no comparison function is defined using comparators is compares with the function default_comparator.
keep_simsum	add a variable minsim to the result with the similarity score of the pair.
deduplication	generate pairs from only x. Ignore y. This is usefull for deduplication of x.
add_xy	add x and y as attributes to the returned pairs. This makes calling some subsequent operations that need x and y (such as <a href="#">compare_pairs</a> easier.

### Details

Generating (all) pairs of the records of two data sets, is usually the first step when linking the two data sets. However, this often results in a too large number of records. `pair_minsim` will only keep pairs with a similarity score equal or larger than `minsim`. The similarity score is calculated by summing the results of the comparators for all variables of `on`.

Missing values in the variables on which the pairs are compared count as a similarity of 0.

### Value

A [data.table](#) with two columns, `.x` and `.y`, is returned. Columns `.x` and `.y` are row numbers from `data.frames .x` and `.y` respectively.

### See Also

[pair](#) and [pair\\_blocking](#) are other methods to generate pairs.

### Examples

```
data("linkexample1", "linkexample2")
pairs <- pair_minsim(linkexample1, linkexample2,
  on = c("postcode", "address"), minsim = 1)
# Either address or postcode has to match to keep a pair

data("linkexample1", "linkexample2")
pairs <- pair_minsim(linkexample1, linkexample2, on_blocking = "postcode",
  on = c("lastname", "firstname", "address"), minsim = 2)
# Postcode has to match; from lastname, firstname, address there have to match
# two or more (e.g. one mismatch is allowed).
```

---

predict.problink\_em    *Calculate weights and probabilities for pairs*

---

### Description

Calculate weights and probabilities for pairs



**Usage**

```
## S3 method for class 'problink_em'
predict(
  object,
  pairs = newdata,
  newdata = NULL,
  type = c("weights", "mpost", "probs", "all"),
  binary = FALSE,
  add = FALSE,
  comparators,
  inplace = FALSE,
  new_name = NULL,
  ...
)
```

**Arguments**

object	an object of type <code>problink_em</code> as produced by <code>problink_em</code> .
pairs	a object with pairs for which to calculate weights.
newdata	an alternative name for the <code>pairs</code> argument. Specify <code>newdata</code> or <code>pairs</code> .
type	a character vector of length one specifying what to calculate. See results for more information.
binary	convert comparison vectors to binary vectors using the comparison function in <code>comparators</code> .
add	add the predictions to the original <code>pairs</code> object.
comparators	a list of comparison functions (see <code>compare_pairs</code> ). When missing <code>attr(pairs, 'comparators')</code> is used.
inplace	logical indicating whether <code>pairs</code> should be modified in place. When <code>pairs</code> is large this can be more efficient.
new_name	name of new object to assign the pairs to on the cluster nodes (only relevant when <code>pairs</code> is of type <code>cluster_pairs</code> ).
...	unused.

**Value**

When `pairs` is of type `pairs`, returns a `data.table` with either the `.x` and `.y` columns from `pairs` (when `add = FALSE`) or all columns of `pairs`. To these columns are added:

- In case of `type = "weights"` a column `weights` with the calculated weights.
- In case of `type = "mpost"` a column `mpost` with the calculated posterior probabilities (probability that pair is a match given comparison vector).
- In case of `type = "prob"` the columns `mprob` and `uprob` with the m and u-probabilities and `mpost` and `upost` with the posterior m- and u-probabilities.
- In case of `type = "all"` all of the above.

In case of `compare_pairs.cluster_pairs`, `compare_pair.pairs` is called on each cluster node and the resulting pairs are assigned to `new_name` in the environment `reclin_env`. When `new_name` is not given (or equal to `NULL`) the original pairs on the nodes are overwritten.

---

problink\_em

*Calculate EM-estimates of m- and u-probabilities*

---

## Description

Calculate EM-estimates of m- and u-probabilities

## Usage

```
problink_em(
  formula,
  data,
  patterns,
  mprobs0 = list(0.95),
  uprobs0 = list(0.02),
  p0 = 0.05,
  tol = 1e-05,
  mprob_max = 0.999,
  uprobs_min = 1e-04
)
```

## Arguments

<code>formula</code>	a formula object with the variables for which to calculate the m- and u-probabilities. Should be of the form $\sim \text{var1} + \text{var2}$ .
<code>data</code>	data set with pairs on which to estimate the model. Alternatively one can use the <code>patterns</code> argument.
<code>patterns</code>	table of patterns (as output by <code>tabulate_patterns</code> ).
<code>mprobs0</code> , <code>uprobs0</code>	initial values of the m- and u-probabilities. These should be lists with numeric values. The names of the elements in the list should correspond to the names in <code>by_x</code> in <code>compare_pairs</code> .
<code>p0</code>	the initial estimate of the probability that a pair is a match.
<code>tol</code>	when the change in the m and u-probabilities is smaller than <code>tol</code> the algorithm is stopped.
<code>mprob_max</code>	maximum values of the estimated m-probabilities. Values equal to one can lead to numerical instabilities.
<code>uprobs_min</code>	maximum values of the estimated m-probabilities. Values equal to zero can lead to numerical instabilities.

**Value**

Returns an object of type `problink_em`. This is a list containing the estimated `mprobs`, `uprobs` and overall linkage probability `p`. It also contains the table of comparison patterns.

**References**

Fellegi, I. and A. Sunter (1969). "A Theory for Record Linkage", *Journal of the American Statistical Association*. 64 (328): pp. 1183-1210. doi:10.2307/2286061.

Herzog, T.N., F.J. Scheuren and W.E. Winkler (2007). *Data Quality and Record Linkage Techniques*, Springer.

**Examples**

```
data("linkexample1", "linkexample2")
pairs <- pair_blocking(linkexample1, linkexample2, "postcode")
pairs <- compare_pairs(pairs, c("lastname", "firstname", "address", "sex"))
model <- probblink_em(~ lastname + firstname + address + sex, data = pairs)
summary(model)
```

---

score\_simple.cluster\_pairs

*Score pairs based on a number of comparison vectors*

---

**Description**

Score pairs based on a number of comparison vectors

**Usage**

```
## S3 method for class 'cluster_pairs'
score_simple(
  pairs,
  variable,
  on,
  w1 = 1,
  w0 = 0,
  wna = 0,
  new_name = NULL,
  ...
)

score_simple(pairs, variable, on, w1 = 1, w0 = 0, wna = 0, ...)
```

```
## S3 method for class 'pairs'
score_simple(
  pairs,
```

```

    variable,
    on,
    w1 = 1,
    w0 = 0,
    wna = 0,
    inplace = FALSE,
    ...
  )

```

## Arguments

<code>pairs</code>	a <code>pairs</code> object, such as generated by <code>pair_blocking</code>
<code>variable</code>	the name of the new variable to create in <code>pairs</code> . This will be a logical variable with a value of <code>TRUE</code> for the selected <code>pairs</code> .
<code>on</code>	character vector of variables on which the score should be based.
<code>w1</code>	a vector or list with weights for agreement for each of the variables. It can either be a numeric vector of length 1 in which case the same weight is used for all variables; A numeric vector of length equal to the length of <code>on</code> in which case the weights correspond one-to-one to the variables in <code>on</code> ; A named numeric vector where the names correspond to those in <code>on</code> , missing values are assigned a value of 1; or a named list with numeric values. See details for more information.
<code>w0</code>	a vector or list with weights for non-agreement for each of the variables. See details for more information. For the format see <code>w1</code> .
<code>wna</code>	a vector or list with weights for agreement for each of the variables. See details for more information. For the format see <code>w1</code> .
<code>new_name</code>	name of new object to assign the <code>pairs</code> to on the cluster nodes.
<code>...</code>	ignored
<code>inplace</code>	logical indicating whether <code>pairs</code> should be modified in place. When <code>pairs</code> is large this can be more efficient.

## Details

The individual contribution of a variable  $x$  to the total score is given by  $x * w1 + (1-x) * w0$  in case of non-NA values and `wna` in case of NA. This assumes that the values 1 corresponds to complete agreement and the value 0 to complete non-agreement. In case of complete agreement a variable contributes `w1` to the total score and in case of complete non-agreement it contributes `w0` to the total score.

## Value

Returns the `data.table` `pairs` with the column `variable` added in case of `score_simple.pairs`.

In case of `score_simple.cluster_pairs`, `score_simple.pairs` is called on each cluster node and the resulting `pairs` are assigned to `new_name` in the environment `reclin_env`. When `new_name` is not given (or equal to `NULL`) the original `pairs` on the nodes are overwritten.

**Examples**

```

data("linkexample1", "linkexample2")
pairs <- pair_blocking(linkexample1, linkexample2, "postcode")
compare_pairs(pairs, on = c("firstname", "lastname", "sex"), inplace = TRUE)

score_simple(pairs, "score", on = c("firstname", "lastname", "sex"))

# Change the default weights
score_simple(pairs, "score", on = c("firstname", "lastname", "sex"),
  w1 = 2, w0 = -1, wna = NA)

# Use a named vector; omitted elements from w1 get a weight of 1; those from
# w0 and wna a weight of 0.
score_simple(pairs, "score", on = c("firstname", "lastname", "sex"),
  w1 = c("firstname" = 2, "lastname" = 3),
  w0 = c("firstname" = -1, "lastname" = -0.5))

# Use a named list; omitted elements from w1 get a weight of 1; those from
# w0 and wna a weight of 0.
score_simple(pairs, "score", on = c("firstname", "lastname", "sex"),
  w1 = list("firstname" = 2, "lastname" = 3),
  w0 = list("firstname" = -1, "lastname" = -0.5))

```

---

```
select_greedy.cluster_pairs
```

*Select matching pairs enforcing one-to-one linkage*

---

**Description**

Select matching pairs enforcing one-to-one linkage

**Usage**

```

## S3 method for class 'cluster_pairs'
select_greedy(
  pairs,
  variable,
  score,
  threshold,
  preselect = NULL,
  id_x = NULL,
  id_y = NULL,
  ...
)

## S3 method for class 'cluster_pairs'
select_n_to_m(

```

```
    pairs,
    variable,
    score,
    threshold,
    preselect = NULL,
    id_x = NULL,
    id_y = NULL,
    ...
)

select_greedy(
  pairs,
  variable,
  score,
  threshold,
  preselect = NULL,
  id_x = NULL,
  id_y = NULL,
  ...
)

## S3 method for class 'pairs'
select_greedy(
  pairs,
  variable,
  score,
  threshold,
  preselect = NULL,
  id_x = NULL,
  id_y = NULL,
  x = attr(pairs, "x"),
  y = attr(pairs, "y"),
  inplace = FALSE,
  include_ties = FALSE,
  n = 1L,
  m = 1L,
  ...
)

select_n_to_m(
  pairs,
  variable,
  score,
  threshold,
  preselect = NULL,
  id_x = NULL,
  id_y = NULL,
  ...
)
```

```

)

## S3 method for class 'pairs'
select_n_to_m(
  pairs,
  variable,
  score,
  threshold,
  preselect = NULL,
  id_x = NULL,
  id_y = NULL,
  x = attr(pairs, "x"),
  y = attr(pairs, "y"),
  inplace = FALSE,
  ...
)

```

### Arguments

<code>pairs</code>	a <code>pairs</code> object, such as generated by <a href="#">pair_blocking</a>
<code>variable</code>	the name of the new variable to create in <code>pairs</code> . This will be a logical variable with a value of <code>TRUE</code> for the selected pairs.
<code>score</code>	name of the score/weight variable of the pairs. When not given and <code>attr(pairs, "score")</code> is defined, that is used.
<code>threshold</code>	the threshold to apply. Pairs with a score above the threshold are selected.
<code>preselect</code>	a logical variable with the same length as <code>pairs</code> has rows, or the name of such a variable in <code>pairs</code> . Pairs are only selected when <code>preselect</code> is <code>TRUE</code> . This interacts with <code>threshold</code> (pairs have to be selected with both conditions).
<code>id_x</code>	a integer vector with the same length as the number of rows in <code>pairs</code> , or the name of a column in <code>x</code> . This vector should identify unique objects in <code>x</code> . When not specified it is assumed that each element in <code>x</code> is unique.
<code>id_y</code>	a integer vector with the same length as the number of rows in <code>pairs</code> , or the name of a column in <code>y</code> . This vector should identify unique objects in <code>y</code> . When not specified it is assumed that each element in <code>y</code> is unique.
<code>...</code>	Used to pass additional arguments to methods
<code>x</code>	<code>data.table</code> with one half of the pairs.
<code>y</code>	<code>data.table</code> with the other half of the pairs.
<code>inplace</code>	logical indicating whether <code>pairs</code> should be modified in place. When <code>pairs</code> is large this can be more efficient.
<code>include_ties</code>	when pairs for a given record have an equal weight, should all pairs be included.
<code>n</code>	an integer. Each element of <code>x</code> can be linked to at most <code>n</code> elements of <code>y</code> .
<code>m</code>	an integer. Each element of <code>y</code> can be linked to at most <code>m</code> elements of <code>x</code> .

## Details

Both methods force one-to-one matching. `select_greedy` uses a greedy algorithm that selects the first pair with the highest weight. `select_n_to_m` tries to optimise the total weight of all of the selected pairs. In general this will result in a better selection. However, `select_n_to_m` uses much more memory and is much slower and, therefore, can only be used when the number of possible pairs is not too large.

Note that when `include_ties = TRUE` the same record can still be selected more than once. In that case the pairs will have an equal weight.

## Value

Returns the pairs with the variable given by `variable` added. This is a logical variable indicating which pairs are selected as matches.

## Examples

```
data("linkexample1", "linkexample2")
pairs <- pair_blocking(linkexample1, linkexample2, "postcode")
pairs <- compare_pairs(pairs, c("lastname", "firstname", "address", "sex"))
model <- problink_em(~ lastname + firstname + address + sex, data = pairs)
pairs <- predict(model, pairs, type = "mpost", add = TRUE, binary = TRUE)

# Select pairs with a mpost > 0.5 and force one-to-one linkage
pairs <- select_n_to_m(pairs, "ntom", "mpost", 0.5)
pairs <- select_greedy(pairs, "greedy", "mpost", 0.5)
table(pairs$ntom, pairs$greedy)

# The same example as above using a cluster;
library(parallel)
cl <- makeCluster(2)

pairs <- cluster_pair_blocking(cl, linkexample1, linkexample2, "postcode")
compare_pairs(pairs, c("lastname", "firstname", "address", "sex"))
model <- problink_em(~ lastname + firstname + address + sex, data = pairs)
predict(model, pairs, type = "mpost", add = TRUE, binary = TRUE)
# Select pairs with a mpost > 0.5 and force one-to-one linkage
# select_n_to_m and select_greedy only work on pairs that are local;
# therefore we first collect the pairs
select_threshold(pairs, "selected", "mpost", 0.5)
local_pairs <- cluster_collect(pairs, "selected")
local_pairs <- select_n_to_m(local_pairs, "ntom", "mpost", 0.5)
local_pairs <- select_greedy(local_pairs, "greedy", "mpost", 0.5)
table(local_pairs$ntom, local_pairs$greedy)

stopCluster(cl)
```



---

```
select_threshold.cluster_pairs
```

*Select matching pairs with a score above or equal to a threshold*

---

## Description

Select matching pairs with a score above or equal to a threshold

## Usage

```
## S3 method for class 'cluster_pairs'
select_threshold(pairs, variable, score, threshold, new_name = NULL, ...)

select_threshold(pairs, variable, score, threshold, ...)

## S3 method for class 'pairs'
select_threshold(pairs, variable, score, threshold, inplace = FALSE, ...)
```

## Arguments

<code>pairs</code>	a pairs object, such as generated by <a href="#">pair_blocking</a>
<code>variable</code>	the name of the new variable to create in pairs. This will be a logical variable with a value of TRUE for the selected pairs.
<code>score</code>	name of the score/weight variable of the pairs. When not given and <code>attr(pairs, "score")</code> is defined, that is used.
<code>threshold</code>	the threshold to apply. Pairs with a score above or equal to the threshold are selected.
<code>new_name</code>	name of new object to assign the pairs to on the cluster nodes.
<code>...</code>	ignored
<code>inplace</code>	logical indicating whether pairs should be modified in place. When pairs is large this can be more efficient.

## Value

Returns the pairs with the variable given by `variable` added. This is a logical variable indicating which pairs are selected a matches.

## Examples

```
data("linkexample1", "linkexample2")
pairs <- pair_blocking(linkexample1, linkexample2, "postcode")
pairs <- compare_pairs(pairs, c("lastname", "firstname", "address", "sex"))
model <- problink_em(~ lastname + firstname + address + sex, data = pairs)
pairs <- predict(model, pairs, type = "mpost", add = TRUE, binary = TRUE)
# Select pairs with a mpost > 0.5
select_threshold(pairs, "selected", "mpost", 0.5, inplace = TRUE)
```

```

# Example using cluster;
# In general the syntax is exactly the same except for the first call to
# to cluster_pair. Note the in general `inplace = TRUE` is implied when
# working with a cluster; therefore the assignment back to pairs can be
# omitted (also not a problem if it is not).
library(parallel)
data("linkexample1", "linkexample2")
cl <- makeCluster(2)

pairs <- cluster_pair(cl, linkexample1, linkexample2)
compare_pairs(pairs, c("lastname", "firstname", "address", "sex"))
model <- problink_em(~ lastname + firstname + address + sex, data = pairs)
predict(model, pairs, type = "mpost", add = TRUE, binary = TRUE)
# Select pairs with a mpost > 0.5
# Unlike the regular pairs: inplace = TRUE is implied here
select_threshold(pairs, "selected", "mpost", 0.5)
stopCluster(cl)

```

---

select\_unique.cluster\_pairs

*Deselect pairs that are linked to multiple records*

---

## Description

Deselect pairs that are linked to multiple records

## Usage

```

## S3 method for class 'cluster_pairs'
select_unique(
  pairs,
  variable,
  preselect = NULL,
  n = 1,
  m = 1,
  id_x = NULL,
  id_y = NULL,
  ...
)

select_unique(
  pairs,
  variable,
  preselect = NULL,
  n = 1,
  m = 1,

```

```

    id_x = NULL,
    id_y = NULL,
    ...
)

## S3 method for class 'pairs'
select_unique(
  pairs,
  variable,
  preselect = NULL,
  n = 1,
  m = 1,
  id_x = NULL,
  id_y = NULL,
  x = attr(pairs, "x"),
  y = attr(pairs, "y"),
  inplace = FALSE,
  ...
)

```

### Arguments

<code>pairs</code>	a <code>pairs</code> object, such as generated by <a href="#">pair_blocking</a>
<code>variable</code>	the name of the new variable to create in <code>pairs</code> . This will be a logical variable with a value of <code>TRUE</code> for the selected pairs.
<code>preselect</code>	a logical variable with the same length as <code>pairs</code> has rows, or the name of such a variable in <code>pairs</code> . Pairs are only selected when <code>preselect</code> is <code>TRUE</code> .
<code>n</code>	do not select pairs with a <code>y</code> -record that is linked to more than <code>n</code> records.
<code>m</code>	do not select pairs with a <code>m</code> -record that is linked to more than <code>m</code> records.
<code>id_x</code>	a integer vector with the same length as the number of rows in <code>pairs</code> , or the name of a column in <code>x</code> . This vector should identify unique objects in <code>x</code> . When not specified it is assumed that each element in <code>x</code> is unique.
<code>id_y</code>	a integer vector with the same length as the number of rows in <code>pairs</code> , or the name of a column in <code>y</code> . This vector should identify unique objects in <code>y</code> . When not specified it is assumed that each element in <code>y</code> is unique.
<code>...</code>	Used to pass additional arguments to methods
<code>x</code>	<code>data.table</code> with one half of the pairs.
<code>y</code>	<code>data.table</code> with the other half of the pairs.
<code>inplace</code>	logical indicating whether <code>pairs</code> should be modified in place. When <code>pairs</code> is large this can be more efficient.

### Details

This function can be used to remove pairs for which there is ambiguity. For example when a record from `x` is linked to multiple records from `y` and we know that there are no duplicate records in `y` (records that belong to the same object), then we know that at least one of the two links is incorrect

but we cannot decide which of the two. In that case we may want to decide that we will not link both records. Running `select_unique` with `m == 1` will remove both records.

In case one wants to select one of the records randomly: `select_greedy` will select the pair with the highest weight and in case of an equal weight the first. Adding a random component to the weights will ensure a random selection.

### Value

Returns the pairs with the variable given by `variable` added. This is a logical variable indicating which pairs are selected as matches.

### Examples

```
data("linkexample1", "linkexample2")
pairs <- pair_blocking(linkexample1, linkexample2, "postcode")
compare_pairs(pairs, on = c("lastname", "firstname", "address", "sex"),
  default_comparator = jaro_winkler(0.9), inplace = TRUE)
score_simple(pairs, "score",
  on = c("lastname", "firstname", "address", "sex"),
  w1 = list(lastname = 2), inplace = TRUE)
select_threshold(pairs, variable = "select",
  score = "score", threshold = 4.0, inplace = TRUE)
select_unique(pairs, variable = "select_unique", preselect = "select")
```

---

summary.problink\_em    Summarise the results from [problink\\_em](#)

---

### Description

Summarise the results from [problink\\_em](#)

### Usage

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

### Arguments

`object`            the [problink\\_em](#) object.  
`...`                ignored;

### Value

Returns the original object with a data frame with the patterns and corresponding m-, u-probabilities and weights added.

---

```
tabulate_patterns.cluster_pairs
```

*Create a table of comparison patterns*

---

**Description**

Create a table of comparison patterns

**Usage**

```
## S3 method for class 'cluster_pairs'
tabulate_patterns(pairs, on, comparators, complete = TRUE, ...)

tabulate_patterns(pairs, on, comparators, complete = TRUE, ...)

## S3 method for class 'pairs'
tabulate_patterns(pairs, on, comparators, complete = TRUE, ...)
```

**Arguments**

<code>pairs</code>	a pairs object, such as generated by <a href="#">pair_blocking</a>
<code>on</code>	variables from pairs defining the comparison patterns. When missing names( <code>comparators</code> ) is used.
<code>comparators</code>	a list with comparison functions for each of the columns. When missing or NULL, the function looks for columns in <code>pairs</code> with a <code>comparator</code> attribute.
<code>complete</code>	add patterns that do not occur in the dataset to the result (with <code>n = 0</code> ).
<code>...</code>	passed on to other methods.

**Details**

Since comparison vectors can contain continuous numbers (usually between 0 and 1), this could result in a very large number of possible comparison vectors. Therefore, the comparison vectors are passed on to the comparators in order to threshold them. This usually results in values 0 or 1. Missing values are usually codes as 0. However, this all depends on the comparison functions used. For more information see the documentation on the [comparison functions](#).

**Value**

Returns a `data.frame` with all unique comparison patterns that exist in `pairs`, with a column `n` added with the number of times each pattern occurs.

**Examples**

```
data("linkexample1", "linkexample2")
pairs <- pair_blocking(linkexample1, linkexample2, "postcode")
pairs <- compare_pairs(pairs, c("lastname", "firstname", "address", "sex"))
tabulate_patterns(pairs)
```

---

town\_names

*Spelling variations of a set of town names*

---

**Description**

Contains spelling variations found in various files of a set of town/village names. Names were selected that contain 'rdam' or 'rdm'. The correct/official names are also given. This data set can be used as an example data set for deduplication

**Format**

Data frames with 584 records and two columns.

**Details**

- name the name of the town/village as found in the files
- official\_name the official/correct name

# Index

- \* **datasets**
  - linkexample1, 19
  - town\_names, 38
- add\_from\_x, 2
- add\_from\_y (add\_from\_x), 2
- cluster\_call, 3
- cluster\_collect, 4, 7, 8, 10
- cluster\_modify\_pairs, 5
- cluster\_pair, 3–5, 6, 8, 10
- cluster\_pair\_blocking, 7, 7, 10
- cluster\_pair\_minsim, 7, 8, 9, 10
- cmp\_identical, 10
- cmp\_jaccard (cmp\_identical), 10
- cmp\_jarowinkler (cmp\_identical), 10
- cmp\_lcs (cmp\_identical), 10
- compare\_pairs, 7, 8, 10, 21, 22, 24–26
- compare\_pairs
  - (compare\_pairs.cluster\_pairs), 12
- compare\_pairs.cluster\_pairs, 12
- compare\_vars
  - (compare\_vars.cluster\_pairs), 13
- compare\_vars.cluster\_pairs, 13
- comparison functions, 37
- data.table, 3, 13, 14, 16, 22–24
- deduplicate\_equivalence, 15
- get\_inspect\_pairs, 16
- greedy, 17
- jaccard (cmp\_identical), 10
- jaro\_winkler (cmp\_identical), 10
- lcs (cmp\_identical), 10
- link, 17
- linkexample1, 19
- linkexample2 (linkexample1), 19
- makeCluster, 6, 8, 9
- match\_n\_to\_m, 19
- merge\_pairs
  - (merge\_pairs.cluster\_pairs), 20
- merge\_pairs.cluster\_pairs, 20
- pair, 7, 8, 10, 21, 23, 24
- pair\_blocking, 8, 15, 18, 22, 22, 24, 28, 31, 33, 35, 37
- pair\_minsim, 22, 23, 23
- predict.problink\_em, 24
- problink\_em, 11, 25, 26, 36
- rbind.cluster\_pairs
  - (merge\_pairs.cluster\_pairs), 20
- rbind.pairs
  - (merge\_pairs.cluster\_pairs), 20
- score\_simple
  - (score\_simple.cluster\_pairs), 27
- score\_simple.cluster\_pairs, 27
- select\_greedy
  - (select\_greedy.cluster\_pairs), 29
- select\_greedy.cluster\_pairs, 29
- select\_n\_to\_m
  - (select\_greedy.cluster\_pairs), 29
- select\_threshold
  - (select\_threshold.cluster\_pairs), 33
- select\_threshold.cluster\_pairs, 33
- select\_unique
  - (select\_unique.cluster\_pairs), 34
- select\_unique.cluster\_pairs, 34
- stringdist, 11
- summary.problink\_em, 36
- tabulate\_patterns, 7, 8, 10, 26

tabulate\_patterns  
    (tabulate\_patterns.cluster\_pairs),  
    37  
tabulate\_patterns.cluster\_pairs, 37  
town\_names, 38