Package 'comparator'

May 8, 2026

Type Package

Title Comparison Functions for Clustering and Record Linkage

Version 0.1.4

Date 2025-03-08

Maintainer Neil Marchant <ngmarchant@gmail.com>

Description Implements functions for comparing strings, sequences and numeric vectors for clustering and record linkage applications. Supported comparison functions include: generalized edit distances for comparing sequences/strings, Monge-Elkan similarity for fuzzy comparison of token sets, and L-p distances for comparing numeric vectors. Where possible, comparison functions are implemented in C/C++ to ensure good performance.

License GPL (>= 2)

Imports Rcpp (>= 1.0.5), proxy (>= 0.4), methods, clue (>= 0.3)

LinkingTo Rcpp

RoxygenNote 7.1.2

Encoding UTF-8

URL https://github.com/ngmarchant/comparator

BugReports https://github.com/ngmarchant/comparator/issues

Collate 'Comparator.R' 'CppSeqComparator.R' 'PairwiseMatrix.R' 'SequenceComparator.R' 'StringComparator.R' 'BinaryComp.R' 'NumericComparator.R' 'Chebyshev.R' 'Constant.R' 'Levenshtein.R' 'DamerauLevenshtein.R' 'Minkowski.R' 'Euclidean.R' 'FuzzyTokenSet.R' 'Hamming.R' 'InVocabulary.R' 'Jaro.R' 'JaroWinkler.R' 'LCS.R' 'Lookup.R' 'Manhattan.R' 'TokenComparator.R' 'MongeElkan.R' 'OSA.R' 'RcppExports.R' 'generalized_mean.R' 'strcompr-package.R' 'util.R'

Suggests testthat

NeedsCompilation yes

Author Neil Marchant [aut, cre]

Repository CRAN

Date/Publication 2025-03-08 21:50:12 UTC

Contents

`BinaryComp` *Binary String/Sequence Comparator*

Description

Compares a pair of strings or sequences based on whether they are identical or not.

Usage

```
BinaryComp(score = 1, similarity = FALSE, ignore_case = FALSE)
```

Arguments

`score`	a numeric of length 1. Positive distance to return if the pair of strings/sequences are not identical. Defaults to 1.0.
`similarity`	a logical. If TRUE, similarities are returned instead of distances. Specifically `score` is returned if the strings agree, and 0.0 is returned otherwise.
`ignore_case`	a logical. If TRUE, case is ignored when comparing strings.

Details

If `similarity` = `FALSE` (default) the scores can be interpreted as distances. When $x = y$ the comparator returns a distance of 0.0, and when $x \neq y$ the comparator returns `score`.

If `similarity` = `TRUE` the scores can be interpreted as similarities. When $x = y$ the comparator returns `score`, and when $x \neq y$ the comparator returns 0.0.

Value

A `BinaryComp` instance is returned, which is an S4 class inheriting from `StringComparator`.

`Chebyshev` *Chebyshev Numeric Comparator*

Description

The Chebyshev distance (a.k.a. L-Inf distance or) between two vectors x and y is the greatest of the absolute differences between each coordinate:

$$\text{Chebyshev}(x, y) = \max_i |x_i - y_i|.$$

Usage

```
Chebyshev()
```

Value

A `Chebyshev` instance is returned, which is an S4 class inheriting from `NumericComparator`.

Note

The Chebyshev distance is a limiting case of the `Minkowski` distance where $p \to \infty$.

See Also

Other numeric comparators include `Manhattan`, `Euclidean` and `Minkowski`.

Examples

```
## Distance between two vectors
x <- c(0, 1, 0, 1, 0)
y <- seq_len(5)
Chebyshev()(x, y)

## Distance between rows (elementwise) of two matrices
comparator <- Chebyshev()
x <- matrix(rnorm(25), nrow = 5)
y <- matrix(rnorm(5), nrow = 1)
elementwise(comparator, x, y)

## Distance between rows (pairwise) of two matrices
pairwise(comparator, x, y)
```

Comparator-class *Virtual Comparator Class*

Description

This class represents a function for comparing pairs of objects. It is the base class from which other types of comparators (e.g. NumericComparator and StringComparator) are derived.

Slots

`.Data` a function which takes a pair of arguments `x` and `y`, and returns the elementwise scores.

`symmetric` a logical of length 1. If TRUE, the comparator is symmetric in its arguments—i.e. `comparator(x, y)` is identical to `comparator(y, x)`.

`distance` a logical of length 1. If `TRUE`, the comparator produces distances and satisfies `comparator(x, x) = 0`. The comparator may not satisfy all of the properties of a distance metric.

`similarity` a logical of length 1. If `TRUE`, the comparator produces similarity scores.

`tri_inequal` a logical of length 1. If `TRUE`, the comparator satisfies the triangle inequality. This is only possible (but not guaranteed) if `distance = TRUE` and `symmetric = TRUE`.

Constant *Constant String/Sequence Comparator*

Description

A trivial comparator that returns a constant for any pair of strings or sequences.

Usage

```
Constant(constant = 0)
```

Arguments

constant	a non-negative numeric vector of length 1. Defaults to zero.

Value

A Constant instance is returned, which is an S4 class inheriting from StringComparator.

CppSeqComparator-class

Virtual Class for a Sequence Comparator with a C++ Implementation

---

Description

This class is a trait possessed by SequenceComparators that have a C++ implementation. SequenceComparators without this trait are implemented in R, and may be slower to execute.

DamerauLevenshtein *Damerau-Levenshtein String/Sequence Comparator*

Description

The Damerau-Levenshtein distance between two strings/sequences x and y is the minimum cost of operations (insertions, deletions, substitutions or transpositions) required to transform x into y. It differs from the Levenshtein distance by including *transpositions* (swaps) among the allowable operations.

Usage

```
DamerauLevenshtein(
  deletion = 1,
  insertion = 1,
  substitution = 1,
  transposition = 1,
  normalize = FALSE,
  similarity = FALSE,
  ignore_case = FALSE,
  use_bytes = FALSE
)
```

Arguments

`deletion`	positive cost associated with deletion of a character or sequence element. Defaults to unit cost.
`insertion`	positive cost associated insertion of a character or sequence element. Defaults to unit cost.
`substitution`	positive cost associated with substitution of a character or sequence element. Defaults to unit cost.
`transposition`	positive cost associated with transposing (swapping) a pair of characters or sequence elements. Defaults to unit cost.
`normalize`	a logical. If TRUE, distances are normalized to the unit interval. Defaults to FALSE.
`similarity`	a logical. If TRUE, similarity scores are returned instead of distances. Defaults to FALSE.
`ignore_case`	a logical. If TRUE, case is ignored when comparing strings.
`use_bytes`	a logical. If TRUE, strings are compared byte-by-byte rather than character-by-character.

Details

For simplicity we assume `x` and `y` are strings in this section, however the comparator is also implemented for more general sequences.

A Damerau-Levenshtein similarity is returned if `similarity = TRUE`, which is defined as

$$\mathrm{sim}(x, y) = \frac{w_d|x| + w_i|y| - \mathrm{dist}(x, y)}{2},$$

where $|x|$, $|y|$ are the number of characters in x and y respectively, dist is the Damerau-Levenshtein distance, w_d is the cost of a deletion and w_i is the cost of an insertion.

Normalization of the Damerau-Levenshtein distance/similarity to the unit interval is also supported by setting `normalize = TRUE`. The normalization approach follows Yujian and Bo (2007), and ensures that the distance remains a metric when the costs of insertion w_i and deletion w_d are equal. The normalized distance dist_n is defined as

$$\mathrm{dist}_n(x, y) = \frac{2\mathrm{dist}(x, y)}{w_d|x| + w_i|y| + \mathrm{dist}(x, y)},$$

and the normalized similarity sim_n is defined as

$$\mathrm{sim}_n(x, y) = 1 - \mathrm{dist}_n(x, y) = \frac{\mathrm{sim}(x, y)}{w_d|x| + w_i|y| - \mathrm{sim}(x, y)}.$$

Value

A `DamerauLevenshtein` instance is returned, which is an S4 class inheriting from `Levenshtein`.

Note

If the costs of deletion and insertion are equal, this comparator is symmetric in x and y. In addition, the normalized and unnormalized distances satisfy the properties of a metric.

References

Boytsov, L. (2011), "Indexing methods for approximate dictionary searching: Comparative analysis", *ACM J. Exp. Algorithmics* **16**, Article 1.1.

Navarro, G. (2001), "A guided tour to approximate string matching", *ACM Computing Surveys (CSUR)*, **33**(1), 31-88.

Yujian, L. & Bo, L. (2007), "A Normalized Levenshtein Distance Metric", *IEEE Transactions on Pattern Analysis and Machine Intelligence* **29**, 1091-1095.

See Also

Other edit-based comparators include `Hamming`, `LCS`, `Levenshtein` and `OSA`.

Examples

```
## The Damerau-Levenshtein distance reduces to ordinary Levenshtein distance
## when the cost of transpositions is high
x <- "plauge"; y <- "plague"
DamerauLevenshtein(transposition = 100)(x, y) == Levenshtein()(x, y)

## Compare car names using normalized Damerau-Levenshtein similarity
data(mtcars)
cars <- rownames(mtcars)
pairwise(DamerauLevenshtein(similarity = TRUE, normalize=TRUE), cars)

## Compare sequences using Damerau-Levenshtein distance
x <- c("G", "T", "G", "C", "T", "G", "G", "C", "C", "C", "A", "T")
y <- c("G", "T", "G", "C", "G", "T", "G", "C", "C", "C", "A", "T")
DamerauLevenshtein()(list(x), list(y))
```

---

elementwise *Elementwise Similarity/Distance Vector*

Description

Computes elementwise similarities/distances between two collections of objects (strings, vectors, etc.) using the provided comparator.

Usage

```
elementwise(comparator, x, y, ...)

## S4 method for signature 'CppSeqComparator,list,list'
elementwise(comparator, x, y, ...)

## S4 method for signature 'StringComparator,vector,vector'
elementwise(comparator, x, y, ...)
```

```
## S4 method for signature 'NumericComparator,matrix,vector'
elementwise(comparator, x, y, ...)

## S4 method for signature 'NumericComparator,vector,matrix'
elementwise(comparator, x, y, ...)

## S4 method for signature 'NumericComparator,vector,vector'
elementwise(comparator, x, y, ...)

## S4 method for signature 'Chebyshev,matrix,matrix'
elementwise(comparator, x, y, ...)

## S4 method for signature 'FuzzyTokenSet,list,list'
elementwise(comparator, x, y, ...)

## S4 method for signature 'InVocabulary,vector,vector'
elementwise(comparator, x, y, ...)

## S4 method for signature 'Lookup,vector,vector'
elementwise(comparator, x, y, ...)

## S4 method for signature 'MongeElkan,list,list'
elementwise(comparator, x, y, ...)
```

Arguments

comparator	a comparator used to compare the objects, which is a sub-class of Comparator.
x, y	a collection of objects to compare, typically stored as entries in an atomic vector, rows in a matrix, or entries in a list. The required format depends on the type of comparator. If x and y do not contain the same number of objects, the smaller collection is recycled according to standard R behavior.
...	other parameters passed on to other methods.

Value

Every object in x is compared to every object in y elementwise (with recycling) using the given comparator, to produce a numeric vector of scores of length $maxsize(x), size(y)$.

Methods (by class)

- comparator = CppSeqComparator, x = list, y = list: Specialization for CppSeqComparator where x and y are lists of sequences (vectors) to compare.
- comparator = StringComparator, x = vector, y = vector: Specialization for StringComparator where x and y are vectors of strings to compare.
- comparator = NumericComparator, x = matrix, y = vector: Specialization for NumericComparator where x is a matrix of rows (interpreted as vectors) to compare with a vector y.

- comparator = NumericComparator, x = vector, y = matrix: Specialization for NumericComparator where x is a vector to compare with a matrix y of rows (interpreted as vectors).
- comparator = NumericComparator, x = vector, y = vector: Specialization for NumericComparator where x and y are vectors to compare.
- comparator = Chebyshev, x = matrix, y = matrix: Specialization for Chebyshev where x and y matrices of rows (interpreted as vectors) to compare. If x any y do not have the same number of rows, rows are recycled in the smaller matrix.
- comparator = FuzzyTokenSet, x = list, y = list: Specialization for FuzzyTokenSet where x and y are lists of token vectors to compare.
- comparator = InVocabulary, x = vector, y = vector: Specialization for InVocabulary where x and y are vectors of strings to compare.
- comparator = Lookup, x = vector, y = vector: Specialization for a Lookup where x and y are vectors of strings to compare
- comparator = MongeElkan, x = list, y = list: Specialization for MongeElkan where x and y lists of token vectors to compare.

Note

This function is not strictly necessary, as the comparator itself is a function that returns element-wise vectors of scores. In other words, comparator(x, y, ...) is equivalent to elementwise(comparator, x, y, ...).

Examples

```
## Compute the absolute difference between two sets of scalar observations
data("iris")
x <- as.matrix(iris$Sepal.Width)
y <- as.matrix(iris$Sepal.Length)
elementwise(Euclidean(), x, y)

## Compute the edit distance between columns of two linked data.frames
col.1 <- c("Hasna Yuhanna", "Korina Zenovia", "Phyllis Haywood", "Nicky Ellen")
col.2 <- c("Hasna Yuhanna", "Corinna Zenovia", "Phyllis Dorothy Haywood", "Nicole Ellen")
elementwise(Levenshtein(), col.1, col.2)
Levenshtein()(col.1, col.2)              # equivalent to above

## Recycling is used if the two collections don't contain the same number of objects
elementwise(Levenshtein(), "Cora Zenovia", col.1)
```

Euclidean *Euclidean Numeric Comparator*

Description

The Euclidean distance (a.k.a. L-2 distance) between two vectors x and y is the square root of the sum of the squared differences of the Cartesian coordinates:

$$\text{Euclidean}(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}.$$

Usage

```
Euclidean()
```

Value

A `Euclidean` instance is returned, which is an S4 class inheriting from `Minkowski`.

Note

The Euclidean distance is a special case of the `Minkowski` distance with $p = 2$.

See Also

Other numeric comparators include `Manhattan`, `Minkowski` and `Chebyshev`.

Examples

```
## Distance between two vectors
x <- c(0, 1, 0, 1, 0)
y <- seq_len(5)
Euclidean()(x, y)

## Distance between rows (elementwise) of two matrices
comparator <- Euclidean()
x <- matrix(rnorm(25), nrow = 5)
y <- matrix(rnorm(5), nrow = 1)
elementwise(comparator, x, y)

## Distance between rows (pairwise) of two matrices
pairwise(comparator, x, y)
```

FuzzyTokenSet *Fuzzy Token Set Comparator*

Description

Compares a pair of token sets x and y by computing the optimal cost of transforming x into y using single-token operations (insertions, deletions and substitutions). The cost of single-token operations is determined at the character-level using an internal string comparator.

Usage

```
FuzzyTokenSet(
  inner_comparator = Levenshtein(normalize = TRUE),
  agg_function = base::mean,
  deletion = 1,
  insertion = 1,
  substitution = 1
)
```

Arguments

`inner_comparator`	inner string distance comparator of class `StringComparator`. Defaults to normalized `Levenshtein` distance.
`agg_function`	function used to aggregate the costs of the optimal operations. Defaults to `base::mean`.
`deletion`	non-negative weight associated with deletion of a token. Defaults to 1.
`insertion`	non-negative weight associated insertion of a token. Defaults to 1.
`substitution`	non-negative weight associated with substitution of a token. Defaults to 1.

Details

A token set is an unordered enumeration of tokens, which may include duplicates. Given two token sets x and y, this comparator computes the optimal cost of transforming x into y using the following single-token operations:

- deleting a token a from x at cost $w_d \times \text{inner}(a, "")$
- inserting a token b in y at cost $w_i \times \text{inner}("", b)$
- substituting a token a in x for a token b in y at cost $w_s \times \text{inner}(a, b)$

where inner is an internal string comparator and w_d, w_i, w_s are non-negative weights, referred to as `deletion`, `insertion` and `substitution` in the parameter list. By default, the *mean* cost of the optimal set of operations is returned. Other methods of aggregating the costs are supported by specifying a non-default `agg_function`.

If the internal string comparator is a *distance* function, then the optimal set of operations *minimize* the cost. Otherwise, the optimal set of operations *maximize* the cost. The optimization problem is solved exactly using a linear sum assignment solver.

Note

This comparator is qualitatively similar to the `MongeElkan` comparator, however it is arguably more principled, since it is formulated as a cost optimization problem. It also offers more control over the costs of missing tokens (by varying the `deletion` and `insertion` weights). This is useful for comparing full names, when dropping a name (e.g. middle name) shouldn't be severely penalized.

Examples

```
## Compare names with heterogenous representations
x <- "The University of California - San Diego"
y <- "Univ. Calif. San Diego"
# Tokenize strings on white space
x <- strsplit(x, '\\s+')
y <- strsplit(y, '\\s+')
FuzzyTokenSet()(x, y)
# Reduce the cost associated with missing words
FuzzyTokenSet(deletion = 0.5, insertion = 0.5)(x, y)

## Compare full name with abbreviated name, reducing the penalty
## for dropping parts of the name
fullname <- "JOSE ELIAS TEJADA BASQUES"
name <- "JOSE BASQUES"
# Tokenize strings on white space
fullname <- strsplit(fullname, '\\s+')
name <- strsplit(name, '\\s+')
comparator <- FuzzyTokenSet(deletion = 0.5)
comparator(fullname, name) < comparator(name, fullname) # TRUE
```

---

gmean *Geometric Mean*

Description

Geometric Mean

Usage

```
gmean(x, ...)

## Default S3 method:
gmean(x, na.rm = FALSE, ...)
```

Arguments

x	An R object. Currently there are methods for numeric/logical vectors and date, date-time and time interval objects. Complex vectors are allowed for `trim = 0`, only.
...	further arguments passed to or from other methods.
na.rm	a logical value indicating whether NA values should be stripped before the computation proceeds.

Value

The geometric mean of the values in x is computed, as a numeric or complex vector of length one. If x is not logical (coerced to numeric), numeric (including integer) or complex, NA_real_ is returned, with a warning.

See Also

mean for the arithmetic mean and hmean for the harmonic mean.

Examples

```
x <- c(1:10, 50)
xm <- gmean(x)
```

Hamming *Hamming String/Sequence Comparator*

Description

The Hamming distance between two strings/sequences of equal length is the number of positions where the corresponding characters/sequence elements differ. It can be viewed as a type of edit distance where the only permitted operation is substitution of characters/sequence elements.

Usage

```
Hamming(
  normalize = FALSE,
  similarity = FALSE,
  ignore_case = FALSE,
  use_bytes = FALSE
)
```

Arguments

normalize	a logical. If TRUE, distances/similarities are normalized to the unit interval. Defaults to FALSE.
similarity	a logical. If TRUE, similarity scores are returned instead of distances. Defaults to FALSE.
ignore_case	a logical. If TRUE, case is ignored when comparing strings.
use_bytes	a logical. If TRUE, strings are compared byte-by-byte rather than character-by-character.

Details

When the input strings/sequences x and y are of different lengths ($|x| \neq |y|$), the Hamming distance is defined to be ∞.

A Hamming similarity is returned if `similarity = TRUE`. When $|x| = |y|$ the similarity is defined as follows:

$$\text{sim}(x, y) = |x| - \text{dist}(x, y),$$

where sim is the Hamming similarity and $dist$ is the Hamming distance. When $|x| \neq |y|$ the similarity is defined to be 0.

Normalization of the Hamming distance/similarity to the unit interval is also supported by setting `normalize = TRUE`. The raw distance/similarity is divided by the length of the string/sequence $|x| = |y|$. If $|x| \neq |y|$ the normalized distance is defined to be 1, while the normalized similarity is defined to be 0.

Value

A `Hamming` instance is returned, which is an S4 class inheriting from `StringComparator`.

Note

While the unnormalized Hamming distance is a metric, the normalized variant is not as it does not satisfy the triangle inequality.

See Also

Other edit-based comparators include `LCS`, `Levenshtein`, `OSA` and `DamerauLevenshtein`.

Examples

```
## Compare US ZIP codes
x <- "90001"
y <- "90209"
m1 <- Hamming()                                      # unnormalized distance
m2 <- Hamming(similarity = TRUE, normalize = TRUE)  # normalized similarity
m1(x, y)
m2(x, y)
```

`hmean` *Harmonic Mean*

Description

Harmonic Mean

Usage

```
hmean(x, ...)

## Default S3 method:
hmean(x, trim = 0, na.rm = FALSE, ...)
```

Arguments

x	An R object. Currently there are methods for numeric/logical vectors and date, date-time and time interval objects. Complex vectors are allowed for trim = 0, only.
...	further arguments passed to or from other methods.
trim	the fraction (0 to 0.5) of observations to be trimmed from each end of x before the mean is computed. Values of trim outside that range are taken as the nearest endpoint.
na.rm	a logical value indicating whether NA values should be stripped before the computation proceeds.

Value

If trim is zero (the default), the harmonic mean of the values in x is computed, as a numeric or complex vector of length one. If x is not logical (coerced to numeric), numeric (including integer) or complex, NA_real_ is returned, with a warning.

If trim is non-zero, a symmetrically trimmed mean is computed with a fraction of trim observations deleted from each end before the mean is computed.

See Also

mean for the arithmetic mean and gmean for the geometric mean.

Examples

```
x <- c(1:10, 50)
xm <- hmean(x)
```

InVocabulary *In-Vocabulary Comparator*

Description

Compares a pair of strings x and y using a reference vocabulary. Different scores are returned depending on whether both/one/neither of x and y are in the reference vocabulary.

Usage

```
InVocabulary(
  vocab,
  both_in_distinct = 0.7,
  both_in_same = 1,
  one_in = 1,
  none_in = 1,
  ignore_case = FALSE
)
```

Arguments

vocab	a vector containing in-vocabulary (known) strings. Any strings not in this vector are out-of-vocabulary (unknown).
both_in_distinct	score to return if the pair of values being compared are both in vocab and distinct. Defaults to 0.7, which would is appropriate for multiplying by similarity scores. If multiplying by distance scores, a value greater than 1 is likely to be more appropriate.
both_in_same	score to return if the pair of values being compared are both in vocab and identical. Defaults to 1.0, which would leave another score unchanged when multiplied by this one.
one_in	score to return if only one of the pair of values being compared is in vocab. Defaults to 1.0, which would leave another score unchanged when multiplied by this one.
none_in	score to return if none of the pair of values being compared is in vocab. Defaults to 1.0, which would leave another score unchanged when multiplied by this one.
ignore_case	a logical. If TRUE, case is ignored when comparing the strings.

Details

This comparator is not intended to produce useful scores on its own. Rather, it is intended to produce multiplicative factors which can be applied to other similarity/distance scores. It is particularly useful for comparing names when a reference list (vocabulary) of known names is available. For example, it can be configured to down-weight the similarity scores of distinct (known) names like "Roberto" and "Umberto" which are semantically very different, but deceptively similar in terms of edit distance. The normalized Levenshtein similarity for these two names is 75%, but their similarity can be reduced to 53% if multiplied by the score from this comparator using the default settings.

Value

An InVocabulary instance is returned, which is an S4 class inheriting from StringComparator.

Examples

```
## Compare names with possible typos using a reference of known names
known_names <- c("Roberto", "Umberto", "Alberto", "Emberto", "Norberto", "Humberto")
```

```
m1 <- InVocabulary(known_names)
m2 <- Levenshtein(similarity = TRUE, normalize = TRUE)
x <- "Emberto"
y <- c("Enberto", "Umberto")
# "Emberto" and "Umberto" are likely to refer to distinct people (since
# they are known distinct names) so their Levenshtein similarity is
# downweighted to 0.61. "Emberto" and "Enberto" may refer to the same
# person (likely typo), so their Levenshtein similarity of 0.87 is not
# downweighted.
similarities <- m1(x, y) * m2(x, y)
```

`Jaro` *Jaro String/Sequence Comparator*

Description

Compares a pair of strings/sequences `x` and `y` based on the number of greedily-aligned characters/sequence elements and the number of transpositions. It was developed for comparing names at the U.S. Census Bureau.

Usage

```
Jaro(similarity = TRUE, ignore_case = FALSE, use_bytes = FALSE)
```

Arguments

`similarity`	a logical. If TRUE, similarity scores are returned (default), otherwise distances are returned (see definition under Details).
`ignore_case`	a logical. If TRUE, case is ignored when comparing strings.
`use_bytes`	a logical. If TRUE, strings are compared byte-by-byte rather than character-by-character.

Details

For simplicity we assume `x` and `y` are strings in this section, however the comparator is also implemented for more general sequences.

When `similarity = TRUE` (default), the Jaro similarity is computed as

$$\mathrm{sim}(x, y) = \frac{1}{3}\left(\frac{m}{|x|} + \frac{m}{|y|} + \frac{m - \lfloor \frac{t}{2} \rfloor}{m}\right)$$

where m is the number of "matching" characters (defined below), t is the number of "transpositions", and $|x|, |y|$ are the lengths of the strings x and y. The similarity takes on values in the range $[0, 1]$, where 1 corresponds to a perfect match.

The number of "matching" characters m is computed using a greedy alignment algorithm. The algorithm iterates over the characters in x, attempting to align the i-th character x_i with the first

matching character in y. When looking for matching characters in y, the algorithm only considers previously un-matched characters within a window $[\max(0, i-w), \min(|y|, i+w)]$ where $w = \left\lfloor \frac{\max(|x|,|y|)}{2} \right\rfloor - 1$. The alignment process yields a subsequence of matching characters from x and y. The number of "transpositions" t is defined to be the number of positions in the subsequence of x which are misaligned with the corresponding position in y.

When `similarity = FALSE`, the Jaro distance is computed as

$$\mathrm{dist}(x, y) = 1 - \mathrm{sim}(x, y).$$

Value

A `Jaro` instance is returned, which is an S4 class inheriting from `StringComparator`.

Note

The Jaro distance is not a metric, as it does not satisfy the identity axiom $\mathrm{dist}(x, y) = 0 \Leftrightarrow x = y$.

References

Jaro, M. A. (1989), "Advances in Record-Linkage Methodology as Applied to Matching the 1985 Census of Tampa, Florida", *Journal of the American Statistical Association* **84**(406), 414-420.

See Also

The `JaroWinkler` comparator modifies the `Jaro` comparator by boosting the similarity score for strings/sequences that have matching prefixes.

Examples

```
## Compare names
Jaro()("Martha", "Mathra")
Jaro()("Eileen", "Phyllis")
```

JaroWinkler *Jaro-Winkler String/Sequence Comparator*

Description

The Jaro-Winkler comparator is a variant of the `Jaro` comparator which boosts the similarity score for strings/sequences with matching prefixes. It was developed for comparing names at the U.S. Census Bureau.

Usage

```
JaroWinkler(
  p = 0.1,
  threshold = 0.7,
  max_prefix = 4L,
  similarity = TRUE,
  ignore_case = FALSE,
  use_bytes = FALSE
)
```

Arguments

p	a non-negative numeric scalar no larger than 1/max_prefix. Similarity scores eligible for boosting are scaled by this factor.
threshold	a numeric scalar on the unit interval. Jaro similarities greater than this value are boosted based on matching characters in the prefixes of both strings. Jaro similarities below this value are returned unadjusted. Defaults to 0.7.
max_prefix	a non-negative integer scalar, specifying the size of the prefix to consider for boosting. Defaults to 4 (characters).
similarity	a logical. If TRUE, similarity scores are returned (default), otherwise distances are returned (see definition under Details).
ignore_case	a logical. If TRUE, case is ignored when comparing strings.
use_bytes	a logical. If TRUE, strings are compared byte-by-byte rather than character-by-character.

Details

For simplicity we assume x and y are strings in this section, however the comparator is also implemented for more general sequences.

The Jaro-Winkler similarity (computed when `similarity = TRUE`) is defined in terms of the Jaro similarity. If the Jaro similarity $sim_J(x, y)$ between strings x and y exceeds a user-specified threshold $0 \leq \tau \leq 1$, the similarity score is boosted in proportion to the number of matching characters in the prefixes of x and y. More precisely, the Jaro-Winkler similarity is defined as:

$$\text{sim}_{JW}(x, y) = \text{sim}_J(x, y) + \min(c(x, y), l)p(1 - \text{sim}_J(x, y)),$$

where $c(x, y)$ is the length of the common prefix, $l \geq 0$ is a user-specified upper bound on the prefix size, and $0 \leq p \leq 1/l$ is a scaling factor.

The Jaro-Winkler distance is computed when `similarity = FALSE` and is defined as

$$\text{dist}_{JW}(x, y) = 1 - \text{sim}_{JW}(x, y).$$

Value

A `JaroWinkler` instance is returned, which is an S4 class inheriting from `StringComparator`.

Note

Like the Jaro distance, the Jaro-Winkler distance is not a metric as it does not satisfy the identity axiom.

References

Jaro, M. A. (1989), "Advances in Record-Linkage Methodology as Applied to Matching the 1985 Census of Tampa, Florida", *Journal of the American Statistical Association* **84**(406), 414-420.

Winkler, W. E. (2006), "Overview of Record Linkage and Current Research Directions", Tech. report. Statistics #2006-2. Statistical Research Division, U.S. Census Bureau.

Winkler, W., McLaughlin G., Jaro M. and Lynch M. (1994), strcmp95.c, Version 2. United States Census Bureau.

See Also

This comparator reduces to the `Jaro` comparator when `max_prefix = 0L` or `threshold = 0.0`.

Examples

```
## Compare names
JaroWinkler()("Martha", "Mathra")
JaroWinkler()("Eileen", "Phyllis")

## Reduce the threshold for boosting
x <- "Matthew"
y <- "Martin"
JaroWinkler()(x, y) < JaroWinkler(threshold = 0.5)(x, y)
```

LCS *Longest Common Subsequence (LCS) Comparator*

Description

The Longest Common Subsequence (LCS) distance between two strings/sequences x and y is the minimum cost of operations (insertions and deletions) required to transform x into y. The LCS similarity is more commonly used, which can be interpreted as the length of the longest subsequence common to x and y.

Usage

```
LCS(
  deletion = 1,
  insertion = 1,
  normalize = FALSE,
  similarity = FALSE,
  ignore_case = FALSE,
  use_bytes = FALSE
)
```

Arguments

`deletion`	positive cost associated with deletion of a character or sequence element. Defaults to unit cost.
`insertion`	positive cost associated insertion of a character or sequence element. Defaults to unit cost.
`normalize`	a logical. If TRUE, distances are normalized to the unit interval. Defaults to FALSE.
`similarity`	a logical. If TRUE, similarity scores are returned instead of distances. Defaults to FALSE.
`ignore_case`	a logical. If TRUE, case is ignored when comparing strings.
`use_bytes`	a logical. If TRUE, strings are compared byte-by-byte rather than character-by-character.

Details

For simplicity we assume `x` and `y` are strings in this section, however the comparator is also implemented for more general sequences.

An LCS similarity is returned if `similarity = TRUE`, which is defined as

$$\mathrm{sim}(x, y) = \frac{w_d|x| + w_i|y| - \mathrm{dist}(x, y)}{2},$$

where $|x|$, $|y|$ are the number of characters in x and y respectively, $dist$ is the LCS distance, w_d is the cost of a deletion and w_i is the cost of an insertion.

Normalization of the LCS distance/similarity to the unit interval is also supported by setting `normalize = TRUE`. The normalization approach follows Yujian and Bo (2007), and ensures that the distance remains a metric when the costs of insertion w_i and deletion w_d are equal. The normalized distance dist_n is defined as

$$\mathrm{dist}_n(x, y) = \frac{2\mathrm{dist}(x, y)}{w_d|x| + w_i|y| + \mathrm{dist}(x, y)},$$

and the normalized similarity sim_n is defined as

$$\mathrm{sim}_n(x, y) = 1 - \mathrm{dist}_n(x, y) = \frac{\mathrm{sim}(x, y)}{w_d|x| + w_i|y| - \mathrm{sim}(x, y)}.$$

Value

A `LCS` instance is returned, which is an S4 class inheriting from `StringComparator`.

Note

If the costs of deletion and insertion are equal, this comparator is symmetric in x and y. In addition, the normalized and unnormalized distances satisfy the properties of a metric.

References

Bergroth, L., Hakonen, H., & Raita, T. (2000), "A survey of longest common subsequence algorithms", *Proceedings Seventh International Symposium on String Processing and Information Retrieval (SPIRE'00)*, 39-48.

Yujian, L. & Bo, L. (2007), "A Normalized Levenshtein Distance Metric", *IEEE Transactions on Pattern Analysis and Machine Intelligence* **29**, 1091–1095.

See Also

Other edit-based comparators include `Hamming`, `Levenshtein`, `OSA` and `DamerauLevenshtein`.

Examples

```
## There are no common substrings of size 3 for the following example,
## however there are two common substrings of size 2: "AC" and "BC".
## Hence the LCS similarity is 2.
x <- "ABCDA"; y <- "BAC"
LCS(similarity = TRUE)(x, y)

## Levenshtein distance reduces to LCS distance when the cost of
## substitution is high
x <- "ABC"; y <- "AAA"
LCS()(x, y) == Levenshtein(substitution = 100)(x, y)
```

`Levenshtein` *Levenshtein String/Sequence Comparator*

Description

The Levenshtein (edit) distance between two strings/sequences x and y is the minimum cost of operations (insertions, deletions or substitutions) required to transform x into y.

Usage

```
Levenshtein(
  deletion = 1,
  insertion = 1,
  substitution = 1,
  normalize = FALSE,
  similarity = FALSE,
  ignore_case = FALSE,
  use_bytes = FALSE
)
```

Arguments

`deletion`	positive cost associated with deletion of a character or sequence element. Defaults to unit cost.
`insertion`	positive cost associated insertion of a character or sequence element. Defaults to unit cost.
`substitution`	positive cost associated with substitution of a character or sequence element. Defaults to unit cost.
`normalize`	a logical. If TRUE, distances are normalized to the unit interval. Defaults to FALSE.
`similarity`	a logical. If TRUE, similarity scores are returned instead of distances. Defaults to FALSE.
`ignore_case`	a logical. If TRUE, case is ignored when comparing strings.
`use_bytes`	a logical. If TRUE, strings are compared byte-by-byte rather than character-by-character.

Details

For simplicity we assume `x` and `y` are strings in this section, however the comparator is also implemented for more general sequences.

A Levenshtein similarity is returned if `similarity = TRUE`, which is defined as

$$\mathrm{sim}(x, y) = \frac{w_d|x| + w_i|y| - \mathrm{dist}(x, y)}{2},$$

where $|x|$, $|y|$ are the number of characters in x and y respectively, dist is the Levenshtein distance, w_d is the cost of a deletion and w_i is the cost of an insertion.

Normalization of the Levenshtein distance/similarity to the unit interval is also supported by setting `normalize = TRUE`. The normalization approach follows Yujian and Bo (2007), and ensures that the distance remains a metric when the costs of insertion w_i and deletion w_d are equal. The normalized distance dist_n is defined as

$$\mathrm{dist}_n(x, y) = \frac{2\mathrm{dist}(x, y)}{w_d|x| + w_i|y| + \mathrm{dist}(x, y)},$$

and the normalized similarity sim_n is defined as

$$\mathrm{sim}_n(x, y) = 1 - \mathrm{dist}_n(x, y) = \frac{\mathrm{sim}(x, y)}{w_d|x| + w_i|y| - \mathrm{sim}(x, y)}.$$

Value

A `Levenshtein` instance is returned, which is an S4 class inheriting from `StringComparator`.

Note

If the costs of deletion and insertion are equal, this comparator is symmetric in x and y. In addition, the normalized and unnormalized distances satisfy the properties of a metric.

References

Navarro, G. (2001), "A guided tour to approximate string matching", *ACM Computing Surveys (CSUR)*, **33**(1), 31-88.

Yujian, L. & Bo, L. (2007), "A Normalized Levenshtein Distance Metric", *IEEE Transactions on Pattern Analysis and Machine Intelligence* **29**, 1091–1095.

See Also

Other edit-based comparators include `Hamming`, `LCS`, `OSA` and `DamerauLevenshtein`.

Examples

```
## Compare names with potential typos
x <- c("Brian Cheng", "Bryan Cheng", "Kondo Onyejekwe", "Condo Onyejekve")
pairwise(Levenshtein(), x, return_matrix = TRUE)

## When the substitution cost is high, Levenshtein distance reduces to LCS distance
Levenshtein(substitution = 100)("Iran", "Iraq") == LCS()("Iran", "Iraq")
```

`Lookup` *Lookup String Comparator*

Description

Compares a pair of strings x and y by retrieving their distance/similarity score from a provided lookup table.

Usage

```
Lookup(
  lookup_table,
  values_colnames,
  score_colname,
  default_match = 0,
  default_nonmatch = NA_real_,
  symmetric = TRUE,
  ignore_case = FALSE
)
```

Arguments

`lookup_table`	data frame containing distances/similarities for pairs of values
`values_colnames`	character vector containing the colnames corresponding to pairs of values (e.g. strings) in `lookup_table`
`score_colname`	name of column that contains distances/similarities in `lookup_table`

default_match — distance/similarity to use if the pair of values match exactly and do not appear in lookup_table. Defaults to 0.0.

default_nonmatch — distance/similarity to use if the pair of values are not an exact match and do not appear in lookup table. Defaults to NA.

symmetric — whether the underlying distance/similarity scores are symmetric. If TRUE lookup_table need only contain entries for one of the two pairs—i.e. an entry for value pair (y, x) is not required if an entry for (x, y) is already present.

ignore_case — a logical. If TRUE, case is ignored when comparing the strings.

Details

The lookup table should contain three columns corresponding to x, and y (values_colnames below) and the distance/similarity (score_colname below). If a pair of values x and y is not in the lookup table, a default distance/similarity is returned depending on whether $x = y$ (default_match below) or $x \neq y$ (default_nonmatch below).

Value

A Lookup instance is returned, which is an S4 class inheriting from StringComparator.

Examples

```
## Measure the distance between cities
lookup_table <- data.frame(x = c("Melbourne", "Melbourne", "Sydney"),
                           y = c("Sydney", "Brisbane", "Brisbane"),
                           dist = c(713.4, 1374.8, 732.5))

comparator <- Lookup(lookup_table, c("x", "y"), "dist")
comparator("Sydney", "Melbourne")
comparator("Melbourne", "Perth")
```

Manhattan *Manhattan Numeric Comparator*

Description

The Manhattan distance (a.k.a. L-1 distance) between two vectors x and y is the sum of the absolute differences of their Cartesian coordinates:

$$\text{Manhattan}(x, y) = \sum_{i=1}^{n} |x_i - y_i|.$$

Usage

```
Manhattan()
```

Value

A Manhattan instance is returned, which is an S4 class inheriting from Minkowski.

Note

The Manhattan distance is a special case of the Minkowski distance with $p = 1$.

See Also

Other numeric comparators include Euclidean, Minkowski and Chebyshev.

Examples

```
## Distance between two vectors
x <- c(0, 1, 0, 1, 0)
y <- seq_len(5)
Manhattan()(x, y)

## Distance between rows (elementwise) of two matrices
comparator <- Manhattan()
x <- matrix(rnorm(25), nrow = 5)
y <- matrix(rnorm(5), nrow = 1)
elementwise(comparator, x, y)

## Distance between rows (pairwise) of two matrices
pairwise(comparator, x, y)
```

Minkowski *Minkowski Numeric Comparator*

Description

The Minkowski distance (a.k.a. L-p distance) between two vectors x and y is the p-th root of the sum of the absolute differences of their Cartesian coordinates raised to the p-th power:

$$\text{Minkowski}(x, y) = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{1/p}.$$

Usage

```
Minkowski(p = 2)
```

Arguments

p	a positive numeric specifying the order of the distance. Defaults to 2 (Euclidean distance). If p < 1 the Minkowski distance does not satisfy the triangle inequality and is therefore not a proper distance metric.

Value

A `Minkowski` instance is returned, which is an S4 class inheriting from `NumericComparator`.

See Also

Other numeric comparators include `Manhattan`, `Euclidean` and `Chebyshev`.

Examples

```
## Distance between two vectors
x <- c(0, 1, 0, 1, 0)
y <- seq_len(5)
Minkowski()(x, y)

## Distance between rows (elementwise) of two matrices
comparator <- Minkowski()
x <- matrix(rnorm(25), nrow = 5)
y <- matrix(rnorm(5), nrow = 1)
elementwise(comparator, x, y)

## Distance between rows (pairwise) of two matrices
pairwise(comparator, x, y)
```

MongeElkan *Monge-Elkan Token Comparator*

Description

Compares a pair of token sets x and y by computing similarity scores between all pairs of tokens using an internal string comparator, then taking the mean of the maximum scores for each token in x.

Usage

```
MongeElkan(
  inner_comparator = Levenshtein(similarity = TRUE, normalize = TRUE),
  agg_function = base::mean,
  symmetrize = FALSE
)
```

Arguments

`inner_comparator` internal string comparator of class `StringComparator`. Defaults to `Levenshtein` similarity.

`agg_function` aggregation function to use when aggregating internal similarities/distances between tokens. Defaults to `mean`, however `hmean` may be a better choice when the comparator returns normalized similarity scores.

`symmetrize` logical indicating whether to use a symmetrized version of the Monge-Elkan comparator. Defaults to FALSE.

Details

A token set is an unordered enumeration of tokens, which may include duplicates. Given two token sets x and y, the Monge-Elkan comparator is defined as:

$$\mathrm{ME}(x, y) = \frac{1}{|x|} \sum_{i=1}^{|x|} \max_j \mathrm{sim}(x_i, y_j)$$

where x_i is the i-th token in x, $|x|$ is the number of tokens in x and sim is an internal string similarity comparator.

A generalization of the original Monge-Elkan comparator is implemented here, which allows for distance comparators in place of similarity comparators, and/or more general aggregation functions in place of the arithmetic mean. The generalized Monge-Elkan comparator is defined as:

$$\mathrm{ME}(x, y) = \mathrm{agg}(\mathrm{opt}_j \, \mathrm{inner}(x_i, y_j))$$

where inner is an internal distance or similarity comparator, opt is $\max$ if inner is a similarity comparator or $\min$ if it is a distance comparator, and agg is an aggregation function which takes a vector of scores for each token in x and returns a scalar.

By default, the Monge-Elkan comparator is asymmetric in its arguments x and y. If `symmetrize = TRUE`, a symmetric version of the comparator is obtained as follows

$$\mathrm{ME}_{sym}(x, y) = \mathrm{opt}\ \{\mathrm{ME}(x, y), \mathrm{ME}(y, x)\}$$

where opt is defined above.

Value

A `MongeElkan` instance is returned, which is an S4 class inheriting from `StringComparator`.

References

Monge, A. E., & Elkan, C. (1996), "The Field Matching Problem: Algorithms and Applications", In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD'96)*, pp. 267-270.

Jimenez, S., Becerra, C., Gelbukh, A., & Gonzalez, F. (2009), "Generalized Monge-Elkan Method for Approximate Text String Comparison", In *Computational Linguistics and Intelligent Text Processing*, pp. 559-570.

Examples

```
## Compare names with heterogenous representations
x <- "The University of California - San Diego"
y <- "Univ. Calif. San Diego"
# Tokenize strings on white space
x <- strsplit(x, '\\s+')
y <- strsplit(y, '\\s+')
```

```
MongeElkan()(x, y)

## The symmetrized variant is arguably more appropriate for this example
MongeElkan(symmetrize = TRUE)(x, y)

## Using a different internal comparator changes the result
MongeElkan(inner_comparator = BinaryComp(), symmetrize=TRUE)(x, y)
```

NumericComparator-class *Virtual Numeric Comparator Class*

Description

Represents a comparator for comparing pairs of numeric vectors.

Slots

`.Data` a function that calls the elementwise method for this class, with arguments `x`, `y` and `...`.

`symmetric` a logical of length 1. If TRUE, the comparator is symmetric in its arguments—i.e. `comparator(x, y)` is identical to `comparator(y, x)`.

`distance` a logical of length 1. If `TRUE`, the comparator produces distances and satisfies `comparator(x, x) = 0`. The comparator may not satisfy all of the properties of a distance metric.

`similarity` a logical of length 1. If `TRUE`, the comparator produces similarity scores.

`tri_inequal` a logical of length 1. If `TRUE`, the comparator satisfies the triangle inequality. This is only possible (but not guaranteed) if `distance = TRUE` and `symmetric = TRUE`.

OSA *Optimal String Alignment (OSA) String/Sequence Comparator*

Description

The Optimal String Alignment (OSA) distance between two strings/sequences x and y is the minimum cost of operations (insertions, deletions, substitutions or transpositions) required to transform x into y, subject to the constraint that *no substring/subsequence is edited more than once*.

Usage

```
OSA(
  deletion = 1,
  insertion = 1,
  substitution = 1,
  transposition = 1,
  normalize = FALSE,
  similarity = FALSE,
  ignore_case = FALSE,
  use_bytes = FALSE
)
```

Arguments

deletion	positive cost associated with deletion of a character or sequence element. Defaults to unit cost.
insertion	positive cost associated insertion of a character or sequence element. Defaults to unit cost.
substitution	positive cost associated with substitution of a character or sequence element. Defaults to unit cost.
transposition	positive cost associated with transposing (swapping) a pair of characters or sequence elements. Defaults to unit cost.
normalize	a logical. If TRUE, distances are normalized to the unit interval. Defaults to FALSE.
similarity	a logical. If TRUE, similarity scores are returned instead of distances. Defaults to FALSE.
ignore_case	a logical. If TRUE, case is ignored when comparing strings.
use_bytes	a logical. If TRUE, strings are compared byte-by-byte rather than character-by-character.

Details

For simplicity we assume x and y are strings in this section, however the comparator is also implemented for more general sequences.

An OSA similarity is returned if similarity = TRUE, which is defined as

$$\mathrm{sim}(x, y) = \frac{w_d|x| + w_i|y| - \mathrm{dist}(x, y)}{2},$$

where $|x|$, $|y|$ are the number of characters in x and y respectively, $dist$ is the OSA distance, w_d is the cost of a deletion and w_i is the cost of an insertion.

Normalization of the OSA distance/similarity to the unit interval is also supported by setting normalize = TRUE. The normalization approach follows Yujian and Bo (2007), and ensures that the distance remains a metric when the costs of insertion w_i and deletion w_d are equal. The normalized distance dist_n is defined as

$$\mathrm{dist}_n(x, y) = \frac{2\mathrm{dist}(x, y)}{w_d|x| + w_i|y| + \mathrm{dist}(x, y)},$$

and the normalized similarity sim_n is defined as

$$\mathrm{sim}_n(x, y) = 1 - \mathrm{dist}_n(x, y) = \frac{\mathrm{sim}(x, y)}{w_d|x| + w_i|y| - \mathrm{sim}(x, y)}.$$

Value

An `OSA` instance is returned, which is an S4 class inheriting from `StringComparator`.

Note

If the costs of deletion and insertion are equal, this comparator is symmetric in x and y. The OSA distance is not a proper metric as it does not satisfy the triangle inequality. The Damerau-Levenshtein distance is closely related—it allows the same edit operations as OSA, but removes the requirement that no substring can be edited more than once.

References

Boytsov, L. (2011), "Indexing methods for approximate dictionary searching: Comparative analysis", *ACM J. Exp. Algorithmics* **16**, Article 1.1.

Navarro, G. (2001), "A guided tour to approximate string matching", *ACM Computing Surveys (CSUR)*, **33**(1), 31-88.

Yujian, L. & Bo, L. (2007), "A Normalized Levenshtein Distance Metric", *IEEE Transactions on Pattern Analysis and Machine Intelligence* **29**: 1091–1095.

See Also

Other edit-based comparators include `Hamming`, `LCS`, `Levenshtein` and `DamerauLevenshtein`.

Examples

```
## Compare strings with a transposition error
x <- "plauge"; y <- "plague"
OSA()(x, y) != Levenshtein()(x, y)

## Unlike Damerau-Levenshtein, OSA does not allow a substring to be
## edited more than once
x <- "ABC"; y <- "CA"
OSA()(x, y) != DamerauLevenshtein()(x, y)

## Compare car names using normalized OSA similarity
data(mtcars)
cars <- rownames(mtcars)
pairwise(OSA(similarity = TRUE, normalize=TRUE), cars)
```

pairwise *Pairwise Similarity/Distance Matrix*

Description

Computes pairwise similarities/distances between two collections of objects (strings, vectors, etc.) using the provided comparator.

Usage

```
pairwise(comparator, x, y, return_matrix = FALSE, ...)

## S4 method for signature 'Comparator,ANY,missing'
pairwise(comparator, x, y, return_matrix = FALSE, ...)

## S4 method for signature 'CppSeqComparator,list,list'
pairwise(comparator, x, y, return_matrix = FALSE, ...)

## S4 method for signature 'CppSeqComparator,list,`NULL`'
pairwise(comparator, x, y, return_matrix = FALSE, ...)

## S4 method for signature 'StringComparator,vector,vector'
pairwise(comparator, x, y, return_matrix = FALSE, ...)

## S4 method for signature 'StringComparator,vector,`NULL`'
pairwise(comparator, x, y, return_matrix = FALSE, ...)

## S4 method for signature 'NumericComparator,matrix,vector'
pairwise(comparator, x, y, return_matrix = FALSE, ...)

## S4 method for signature 'NumericComparator,vector,matrix'
pairwise(comparator, x, y, return_matrix = FALSE, ...)

## S4 method for signature 'Chebyshev,matrix,matrix'
pairwise(comparator, x, y, return_matrix = FALSE, ...)

## S4 method for signature 'Chebyshev,matrix,`NULL`'
pairwise(comparator, x, y, return_matrix = FALSE, ...)

## S4 method for signature 'Minkowski,matrix,matrix'
elementwise(comparator, x, y, ...)

## S4 method for signature 'Minkowski,matrix,matrix'
pairwise(comparator, x, y, return_matrix = FALSE, ...)

## S4 method for signature 'Minkowski,matrix,`NULL`'
pairwise(comparator, x, y, return_matrix = FALSE, ...)
```

```
## S4 method for signature 'FuzzyTokenSet,list,list'
pairwise(comparator, x, y, return_matrix = FALSE, ...)

## S4 method for signature 'FuzzyTokenSet,vector,`NULL`'
pairwise(comparator, x, y, return_matrix = FALSE, ...)

## S4 method for signature 'InVocabulary,vector,vector'
pairwise(comparator, x, y, return_matrix = FALSE, ...)

## S4 method for signature 'InVocabulary,vector,`NULL`'
pairwise(comparator, x, y, return_matrix = FALSE, ...)

## S4 method for signature 'Lookup,vector,vector'
pairwise(comparator, x, y, return_matrix = FALSE, ...)

## S4 method for signature 'Lookup,vector,`NULL`'
pairwise(comparator, x, y, return_matrix = FALSE, ...)

## S4 method for signature 'MongeElkan,list,list'
pairwise(comparator, x, y, return_matrix = FALSE, ...)

## S4 method for signature 'MongeElkan,list,`NULL`'
pairwise(comparator, x, y, return_matrix = FALSE, ...)
```

Arguments

comparator	a comparator used to compare the objects, which is a sub-class of Comparator.
x, y	a collection of objects to compare, typically stored as entries in an atomic vector, rows in a matrix, or entries in a list. The required format depends on the type of comparator. y may be omitted or set to NULL to compare objects in x.
return_matrix	a logical of length 1. If FALSE (default), the pairwise similarities/distances will be returned as a PairwiseMatrix which is more space-efficient for symmetric comparators. If TRUE, a standard matrix is returned instead.
...	other parameters passed on to other methods.

Value

If both x and y are specified, every object in x is compared with every object in y using the comparator, and the resulting scores are returned in a size(x) by size(y) matrix.

If only x is specified, then the objects in x are compared with themselves using the comparator, and the resulting scores are returned in a size(x) by size(y) matrix.

By default, the matrix is represented as an instance of the PairwiseMatrix class, which is more space-efficient for symmetric comparators when y is not specified. However, if return_matrix = TRUE, the matrix is returned as an ordinary matrix instead.

Methods (by class)

- comparator = Comparator,x = ANY,y = missing: Compute a pairwise comparator when y
- comparator = CppSeqComparator,x = list,y = list: Specialization for CppSeqComparator where x and y are lists of sequences (vectors) to compare.
- comparator = CppSeqComparator,x = list,y = NULL: Specialization for CppSeqComparator where x is a list of sequences (vectors) to compare.
- comparator = StringComparator,x = vector,y = vector: Specialization for StringComparator where x and y are vectors of strings to compare.
- comparator = StringComparator,x = vector,y = NULL: Specialization for StringComparator where x is a vector of strings to compare.
- comparator = NumericComparator,x = matrix,y = vector: Specialization for NumericComparator where x is a matrix of rows (interpreted as vectors) to compare with a vector y.
- comparator = NumericComparator,x = vector,y = matrix: Specialization for NumericComparator where x is a vector to compare with a matrix y of rows (interpreted as vectors).
- comparator = Chebyshev,x = matrix,y = matrix: Specialization for Chebyshev where x and y matrices of rows (interpreted as vectors) to compare.
- comparator = Chebyshev,x = matrix,y = NULL: Specialization for Minkowski where x is a matrix of rows (interpreted as vectors) to compare among themselves.
- comparator = Minkowski,x = matrix,y = matrix: Specialization for a Minkowski where x and y matrices of rows (interpreted as vectors) to compare.
- comparator = Minkowski,x = matrix,y = matrix: Specialization for a Minkowski where x and y matrices of rows (interpreted as vectors) to compare.
- comparator = Minkowski,x = matrix,y = NULL: Specialization for Minkowski where x is a matrix of rows (interpreted as vectors) to compare among themselves.
- comparator = FuzzyTokenSet,x = list,y = list: Specialization for FuzzyTokenSet where x and y are lists of token vectors to compare.
- comparator = FuzzyTokenSet,x = vector,y = NULL: Specialization for FuzzyTokenSet where x is a list of token vectors to compare among themselves.
- comparator = InVocabulary,x = vector,y = vector: Specialization for InVocabulary where x and y are vectors of strings to compare.
- comparator = InVocabulary,x = vector,y = NULL: Specialization for InVocabulary where x is a vector of strings to compare among themselves.
- comparator = Lookup,x = vector,y = vector: Specialization for a Lookup where x and y are vectors of strings to compare
- comparator = Lookup,x = vector,y = NULL: Specialization for Lookup where x is a vector of strings to compare among themselves
- comparator = MongeElkan,x = list,y = list: Specialization for MongeElkan where x and y are lists of token vectors to compare.
- comparator = MongeElkan,x = list,y = NULL: Specialization for MongeElkan where x is a list of token vectors to compare among themselves.

Examples

```
## Computing the distances between a query point y (a 3D numeric vector)
## and a set of reference points x
x <- rbind(c(1,0,1), c(0,0,0), c(-1,2,-1))
y <- c(10, 5, 10)
pairwise(Manhattan(), x, y)

## Computing the pairwise similarities among a set of strings
x <- c("Benjamin", "Ben", "Benny", "Bne", "Benedict", "Benson")
comparator <- DamerauLevenshtein(similarity = TRUE, normalize = TRUE)
pairwise(comparator, x, return_matrix = TRUE)  # return an ordinary matrix
```

PairwiseMatrix-class *Pairwise Similarity/Distance Matrix*

Description

Represents a pairwise similarity or distance matrix.

Usage

```
as.PairwiseMatrix(x, ...)

## S4 method for signature 'matrix'
as.PairwiseMatrix(x, ...)

## S4 method for signature 'PairwiseMatrix'
as.matrix(x, ...)
```

Arguments

x	an R object.
...	additional arguments to be passed to methods.

Details

If the elements being compared are from the same set, the matrix may be symmetric if the comparator is symmetric. In this case, entries in the upper triangle and/or along the diagonal may not be stored in memory, since they are redundant.

Functions

- `as.PairwiseMatrix`: Convert an R object `x` to a `PairwiseMatrix`.
- `as.PairwiseMatrix,matrix-method`: Convert an ordinary `matrix` `x` to a `PairwiseMatrix`.
- `as.matrix,PairwiseMatrix-method`: Convert a `PairwiseMatrix` `x` to an ordinary `matrix`.

Slots

`.Data` entries of the matrix in column-major order. Entries in the upper triangle and/or on the diagonal may be omitted.

`Dim` integer vector of length 2. The dimensions of the matrix.

`Diag` logical indicating whether the diagonal entries are stored in `.Data`.

SequenceComparator-class

Virtual Sequence Comparator Class

Description

Represents a comparator for pairs of sequences.

Slots

`.Data` a function that calls the elementwise method for this class, with arguments `x`, `y` and `...`.

`symmetric` a logical of length 1. If TRUE, the comparator is symmetric in its arguments—i.e. `comparator(x, y)` is identical to `comparator(y, x)`.

`distance` a logical of length 1. If `TRUE`, the comparator produces distances and satisfies `comparator(x, x) = 0`. The comparator may not satisfy all of the properties of a distance metric.

`similarity` a logical of length 1. If `TRUE`, the comparator produces similarity scores.

`tri_inequal` a logical of length 1. If `TRUE`, the comparator satisfies the triangle inequality. This is only possible (but not guaranteed) if `distance = TRUE` and `symmetric = TRUE`.

StringComparator-class

Virtual String Comparator Class

Description

Represents a comparator for pairs of strings.

Slots

`.Data` a function that calls the elementwise method for this class, with arguments `x`, `y` and `...`.

`symmetric` a logical of length 1. If TRUE, the comparator is symmetric in its arguments—i.e. `comparator(x, y)` is identical to `comparator(y, x)`.

`distance` a logical of length 1. If `TRUE`, the comparator produces distances and satisfies `comparator(x, x) = 0`. The comparator may not satisfy all of the properties of a distance metric.

`similarity` a logical of length 1. If `TRUE`, the comparator produces similarity scores.

`tri_inequal` a logical of length 1. If `TRUE`, the comparator satisfies the triangle inequality. This is only possible (but not guaranteed) if `distance = TRUE` and `symmetric = TRUE`.

`ignore_case` a logical of length 1. If TRUE, case is ignored when comparing strings. Defaults to FALSE.

`use_bytes` a logical of length 1. If TRUE, strings are compared byte-by-byte rather than character-by-character.

`TokenComparator-class` *Virtual Token Comparator Class*

Description

Represents a comparator for pairs of token sequences.

Slots

`.Data` a function that calls the elementwise method for this class, with arguments `x`, `y` and `...`.

`symmetric` a logical of length 1. If TRUE, the comparator is symmetric in its arguments—i.e. `comparator(x, y)` is identical to `comparator(y, x)`.

`distance` a logical of length 1. If `TRUE`, the comparator produces distances and satisfies `comparator(x, x) = 0`. The comparator may not satisfy all of the properties of a distance metric.

`similarity` a logical of length 1. If `TRUE`, the comparator produces similarity scores.

`tri_inequal` a logical of length 1. If `TRUE`, the comparator satisfies the triangle inequality. This is only possible (but not guaranteed) if `distance = TRUE` and `symmetric = TRUE`.

`ordered` a logical of length 1. If TRUE, the comparator treats token sequences as ordered, otherwise they are treated as unordered.

Index