

Package: parallelDist (via r-universe)

October 7, 2024

Type Package

Title Parallel Distance Matrix Computation using Multiple Threads

Version 0.2.6

Author Alexander Eckert [aut, cre], Lucas Godoy [ctb], Srikanth KS [ctb]

Maintainer Alexander Eckert <info@alexandereckert.com>

Description A fast parallelized alternative to R's native 'dist' function to calculate distance matrices for continuous, binary, and multi-dimensional input matrices, which supports a broad variety of 41 predefined distance functions from the 'stats', 'proxy' and 'dtw' R packages, as well as user-defined functions written in C++. For ease of use, the 'parDist' function extends the signature of the 'dist' function and uses the same parameter naming conventions as distance methods of existing R packages. The package is mainly implemented in C++ and leverages the 'RcppParallel' package to parallelize the distance computations with the help of the 'TinyThread' library. Furthermore, the 'Armadillo' linear algebra library is used for optimized matrix operations during distance calculations. The curiously recurring template pattern (CRTP) technique is applied to avoid virtual functions, which improves the Dynamic Time Warping calculations while the implementation stays flexible enough to support different DTW step patterns and normalization methods.

License GPL (>= 2)

URL <https://github.com/alexeckert/parallelDist>,
<https://www.alexandereckert.com/projects/#r-packages>

BugReports <https://github.com/alexeckert/parallelDist/issues>

NeedsCompilation yes

Depends R (>= 3.0.2)

Imports Rcpp (>= 0.12.6), RcppParallel (>= 4.3.20)

LinkingTo Rcpp, RcppParallel, RcppArmadillo

SystemRequirements C++11

Suggests dtw, ggplot2, proxy, testthat, RcppArmadillo, RcppXPtrUtils

Repository <https://fastverse.r-universe.dev>

RemoteUrl <https://github.com/alexeckert/parallelDist>

RemoteRef HEAD

RemoteSha ef7086596f301bc2ea619e2998f0a79c69a87b52

Contents

parDist	2
Index	10

parDist	<i>Parallel Distance Matrix Computation using multiple Threads</i>
---------	--------------------------------------------------------------------

Description

Calculates distance matrices in parallel using multiple threads. Supports 41 predefined distance measures and user-defined distance functions.

Usage

```
parDist(x, method = "euclidean", diag = FALSE, upper = FALSE, threads = NULL, ...)
parallelDist(x, method = "euclidean", diag = FALSE, upper = FALSE, threads = NULL, ...)
```

Arguments

x	a numeric matrix (each row is one series) or list of numeric matrices for multi-dimensional series (each matrix is one series, a row is a dimension of a series)
method	the distance measure to be used. A list of all available distance methods can be found in the details section below.
diag	logical value indicating whether the diagonal of the distance matrix should be printed by print.dist.
upper	logical value indicating whether the upper triangle of the distance matrix should be printed by print.dist
threads	number of cpu threads for calculating a distance matrix. Default is the maximum amount of cpu threads available on the system.
...	additional parameters which will be passed to the distance methods. See details section below.

Details

User-defined distance functions:

custom Defining and compiling a user-defined C++ distance function, as well as creating an external pointer to the function can easily be achieved with the `cppXPtr` function of the **RcppXPtrUtils** package. The resulting Xptr external pointer object needs to be passed to `parDist` using the `func` parameter.

Parameters:

- `func (Xptr)` External pointer to a user-defined distance function with the following signature:
`double customDist(const arma::mat &A, const arma::mat &B)`
 Note that the return value must be a double and the two parameters must be of type `const arma::mat ¶m`.

More information about the Armadillo library can be found at <http://arma.sourceforge.net/docs.html> or as part of the documentation of the **RcppArmadillo** package.

An exemplary definition and usage of an user-defined euclidean distance function can be found in the examples section below.

Available predefined distance measures (written for two vectors x and y):

Distance methods for continuous input variables

`bhattacharyya` The Bhattacharyya distance.

Type: continuous

Formula: $\sqrt{\sum_i (\sqrt{x_i} - \sqrt{y_i})^2}$.

Details: See `pr_DB$get_entry("bhattacharyya")` in **proxy**.

`bray` The Bray/Curtis dissimilarity.

Type: continuous

Formula: $\sum_i |x_i - y_i| / \sum_i (x_i + y_i)$.

Details: See `pr_DB$get_entry("bray")` in **proxy**.

`canberra` The Canberra distance (with compensation for excluded components). Terms with zero numerator and denominator are omitted from the sum and treated as if the values were missing.

Type: continuous

Formula: $\sum_i |x_i - y_i| / |x_i + y_i|$.

Details: See `pr_DB$get_entry("canberra")` in **proxy**.

`chord` The Chord distance.

Type: continuous

Formula: $\sqrt{2 * (1 - xy / \sqrt{xx * yy})}$.

Details: See `pr_DB$get_entry("chord")` in **proxy**.

`divergence` The Divergence distance.

Type: continuous

Formula: $\sum_i (x_i - y_i)^2 / (x_i + y_i)^2$.

Details: See `pr_DB$get_entry("divergence")` in **proxy**.

`dtw` Implementation of a multi-dimensional Dynamic Time Warping algorithm.

Type: continuous

Formula: Euclidean distance $\sqrt{\sum_i (x_i - y_i)^2}$.

Parameters:

- **window.size (integer, optional)** Size of the window of the Sakoe-Chiba band. If the absolute length difference of two series *x* and *y* is larger than the *window.size*, the *window.size* is set to the length difference.
- **norm.method (character, optional)** Normalization method for DTW distances.
 - *path.length* Normalization with the length of the warping path.
 - *n* Normalization with *n*. *n* is the length of series *x*.
 - *n+m* Normalization with *n + m*. *n* is the length of series *x*, *m* is the length of series *y*.
- **step.pattern (character or stepPattern of dtw package, default: symmetric1)** The following step patterns of the **dtw** package are supported:
 - *asymmetric* (Normalization hint: *n*)
 - *asymmetricP0* (Normalization hint: *n*)
 - *asymmetricP05* (Normalization hint: *n*)
 - *asymmetricP1* (Normalization hint: *n*)
 - *asymmetricP2* (Normalization hint: *n*)
 - *symmetric1* (Normalization hint: *path.length*)
 - *symmetric2* or *symmetricP0* (Normalization hint: *n+m*)
 - *symmetricP05* (Normalization hint: *n+m*)
 - *symmetricP1* (Normalization hint: *n+m*)
 - *symmetricP2* (Normalization hint: *n+m*)

For a detailed description see [stepPattern](#) of the **dtw** package.

euclidean The Euclidean distance/ L_2 -norm (with compensation for excluded components).

Type: continuous

Formula: $\sqrt{\sum_i (x_i - y_i)^2}$.

Details: See `pr_DB$get_entry("euclidean")` in **proxy**.

fJaccard The fuzzy Jaccard distance.

Type: binary

Formula: $\sum_i (\min x_i, y_i) / \sum_i (\max x_i, y_i)$.

Details: See `pr_DB$get_entry("fJaccard")` in **proxy**.

geodesic The geodesic distance, i.e. the angle between *x* and *y*.

Type: continuous

Formula: $\arccos(xy / \sqrt{xx * yy})$.

Details: See `pr_DB$get_entry("geodesic")` in **proxy**.

hellinger The Hellinger distance.

Type: continuous

Formula: $\sqrt{\sum_i (\sqrt{x_i / \sum_i x} - \sqrt{y_i / \sum_i y})^2}$.

Details: See `pr_DB$get_entry("hellinger")` in **proxy**.

kullback The Kullback-Leibler distance.

Type: continuous

Formula: $\sum_i [x_i * \log((x_i / \sum_j x_j) / (y_i / \sum_j y_j)) / \sum_j x_j]$.

Details: See `pr_DB$get_entry("kullback")` in **proxy**.

mahalanobis The Mahalanobis distance. The Variance-Covariance-Matrix is estimated from the input data if unspecified.

Type: continuous

Formula: $\sqrt{(x - y) \text{Sigma}^{-1} (x - y)}$.

Parameters:

- **cov (numeric matrix, optional)** The covariance matrix (p x p) of the distribution.
- **inverted (logical, optional)** If TRUE, cov is supposed to contain the inverse of the covariance matrix.

Details: See `pr_DB$get_entry("mahalanobis")` in **proxy** or `mahalanobis` in **stats**.

manhattan The Manhattan/City-Block/Taxi/L₁-norm distance (with compensation for excluded components).

Type: continuous

Formula: $\sum_i |x_i - y_i|$.

Details: See `pr_DB$get_entry("manhattan")` in **proxy**.

maximum The Maximum/Supremum/Chebyshev distance.

Type: continuous

Formula: $\max_i |x_i - y_i|$.

Details: See `pr_DB$get_entry("maximum")` in **proxy**.

minkowski The Minkowski distance/p-norm (with compensation for excluded components).

Type: continuous

Formula: $(\sum_i (x_i - y_i)^p)^{1/p}$.

Parameters:

- **p (double, optional)** The p th root of the sum of the p th powers of the differences of the components.

Details: See `pr_DB$get_entry("minkowski")` in **proxy**.

podani The Podany measure of discordance is defined on ranks with ties. In the formula, for two given objects x and y, n is the number of variables, a is the number of pairs of variables ordered identically, b the number of pairs reversely ordered, c the number of pairs tied in both x and y (corresponding to either joint presence or absence), and d the number of all pairs of variables tied at least for one of the objects compared such that one, two, or three scores are zero.

Type: continuous

Formula: $1 - 2 * (a - b + c - d) / (n * (n - 1))$.

Details: See `pr_DB$get_entry("podani")` in **proxy**.

soergel The Soergel distance.

Type: continuous

Formula: $\sum_i |x_i - y_i| / \sum_i \max(x_i, y_i)$.

Details: See `pr_DB$get_entry("soergel")` in **proxy**.

wave The Wave/Hedges distance.

Type: continuous

Formula: $\sum_i (1 - \min(x_i, y_i) / \max(x_i, y_i))$.

Details: See `pr_DB$get_entry("wave")` in **proxy**.

whittaker The Whittaker distance.

Type: continuous

Formula: $\sum_i |x_i / \sum_i x - y_i / \sum_i y| / 2$.

Details: See `pr_DB$get_entry("whittaker")` in **proxy**.

Distance methods for binary input variables

Notation:

- a: number of (TRUE, TRUE) pairs
- b: number of (FALSE, TRUE) pairs
- c: number of (TRUE, FALSE) pairs

- d: number of (FALSE, FALSE) pairs

Note: Similarities are converted to distances.

binary The Jaccard Similarity for binary data. It is the proportion of (TRUE, TRUE) pairs, but not considering (FALSE, FALSE) pairs.

Type: binary

Formula: $a/(a + b + c)$.

Details: See `pr_DB$get_entry("binary")` in **proxy**.

braun-blanquet The Braun-Blanquet similarity.

Type: binary

Formula: $a/\max(a + b), (a + c)$.

Details: See `pr_DB$get_entry("braun-blanquet")` in **proxy**.

cosine The cosine similarity.

Type: continuous

Formula: $(a * b)/(|a| * |b|)$.

Details: See `pr_DB$get_entry("cosine")` in **proxy**.

dice The Dice similarity.

Type: binary

Formula: $2a/(2a + b + c)$.

Details: See `pr_DB$get_entry("dice")` in **proxy**.

fager The Fager / McGowan distance.

Type: binary

Formula: $a/\sqrt{((a + b)(a + c))} - \sqrt{(a + c)}/2$.

Details: See `pr_DB$get_entry("fager")` in **proxy**.

faith The Faith similarity.

Type: binary

Formula: $(a + d/2)/n$.

Details: See `pr_DB$get_entry("faith")` in **proxy**.

hamman The Hamman Matching similarity for binary data. It is the proportion difference of the concordant and discordant pairs.

Type: binary

Formula: $([a + d] - [b + c])/n$.

Details: See `pr_DB$get_entry("hamman")` in **proxy**.

hamming The hamming distance between two vectors A and B is the fraction of positions where there is a mismatch.

Formula: $\# of (A \neq B) / \# in A (or B)$

kulczynski1 Kulczynski similarity for binary data. Relates the (TRUE, TRUE) pairs to discordant pairs.

Type: binary

Formula: $a/(b + c)$.

Details: See `pr_DB$get_entry("kulczynski1")` in **proxy**.

kulczynski2 Kulczynski similarity for binary data. Relates the (TRUE, TRUE) pairs to the discordant pairs.

Type: binary

Formula: $[a/(a + b) + a/(a + c)]/2$.

Details: See `pr_DB$get_entry("kulczynski2")` in **proxy**.

michael The Michael similarity.

Type: binary

Formula: $4(ad - bc)/[(a + d)^2 + (b + c)^2]$.

Details: See `pr_DB$get_entry("michael")` in **proxy**.

mountford The Mountford similarity for binary data.

Type: binary

Formula: $2a/(ab + ac + 2bc)$.

Details: See `pr_DB$get_entry("mountford")` in **proxy**.

mozley The Mozley/Margalef similarity.

Type: binary

Formula: $an/(a + b)(a + c)$.

Details: See `pr_DB$get_entry("mozley")` in **proxy**.

ochiai The Ochiai similarity.

Type: binary

Formula: $a/\sqrt{(a + b)(a + c)}$.

Details: See `pr_DB$get_entry("ochiai")` in **proxy**.

phi The Phi similarity (= Product-Moment-Correlation for binary variables).

Type: binary

Formula: $(ad - bc)/\sqrt{(a + b)(c + d)(a + c)(b + d)}$.

Details: See `pr_DB$get_entry("phi")` in **proxy**.

russel The Russel/Raosimilarity for binary data. It is just the proportion of (TRUE, TRUE) pairs.

Type: binary

Formula: a/n .

Details: See `pr_DB$get_entry("russel")` in **proxy**.

simple matching The Simple Matching similarity for binary data. It is the proportion of concordant pairs.

Type: binary

Formula: $(a + d)/n$.

Details: See `pr_DB$get_entry("simple matching")` in **proxy**.

simpson The Simpson similarity.

Type: binary

Formula: $a/\min(a + b, (a + c))$.

Details: See `pr_DB$get_entry("simpson")` in **proxy**.

stiles The Stiles similarity. Identical to the logarithm of Krylov's distance.

Type: binary

Formula: $\log(n(|ad - bc| - 0.5n)^2/[(a + b)(c + d)(a + c)(b + d)])$.

Details: See `pr_DB$get_entry("stiles")` in **proxy**.

tanimoto The Rogers/Tanimoto similarity for binary data. Similar to the simple matching coefficient, but putting double weight on the discordant pairs.

Type: binary

Formula: $(a + d)/(a + 2b + 2c + d)$.

Details: See `pr_DB$get_entry("tanimoto")` in **proxy**.

yule The Yule similarity.

Type: binary

Formula: $(ad - bc)/(ad + bc)$.

Details: See `pr_DB$get_entry("yule")` in **proxy**.

yule2 The Yule similarity.

Type: binary

Formula: $(\sqrt{ad} - \sqrt{bc})/(\sqrt{ad} + \sqrt{bc})$.
 Details: See `pr_DB$get_entry("yule2")` in **proxy**.

Value

`parDist` returns an object of class "dist".

The lower triangle of the distance matrix stored by columns in a vector, say `do`. If n is the number of observations, i.e., $n \leftarrow \text{attr}(\text{do}, "Size")$, then for $i < j \leq n$, the dissimilarity between (row) i and j is `do[n*(i-1) - i*(i-1)/2 + j-i]`. The length of the vector is $n * (n - 1)/2$, i.e., of order n^2 .

The object has the following attributes (besides "class" equal to "dist"):

Size	integer, the number of observations in the dataset.
Labels	optionally, contains the labels, if any, of the observations of the dataset.
Diag, Upper	logicals corresponding to the arguments <code>diag</code> and <code>upper</code> above, specifying how the object should be printed.
call	optionally, the <code>call</code> used to create the object.
method	optionally, the distance method used; resulting from <code>parDist()</code> , the <code>(match.arg())</code> ed method argument.

Examples

```
## Not run:
## predefined distance functions
# defining a matrix, where each row corresponds to one series
sample.matrix <- matrix(c(1:100), ncol = 10)

# euclidean distance
parDist(x = sample.matrix, method = "euclidean")
# minkowski distance with parameter p=2
parDist(x = sample.matrix, method = "minkowski", p=2)
# dynamic time warping distance
parDist(x = sample.matrix, method = "dtw")
# dynamic time warping distance normalized with warping path length
parDist(x = sample.matrix, method = "dtw", norm.method="path.length")
# dynamic time warping with different step pattern
parDist(x = sample.matrix, method = "dtw", step.pattern="symmetric2")
# dynamic time warping with window size constraint
parDist(x = sample.matrix, method = "dtw", step.pattern="symmetric2", window.size=1)

## multi-dimensional distance functions using list of matrices
# defining a list of matrices, where each list entry row corresponds to a two dimensional series
tmp.mat <- matrix(c(1:40), ncol = 10)
sample.matrix.list <- list(tmp.mat[1:2,], tmp.mat[3:4,])

# multi-dimensional euclidean distance
parDist(x = sample.matrix.list, method = "euclidean")
# multi-dimensional dynamic time warping
parDist(x = sample.matrix.list, method = "dtw")
```



```
## user-defined distance function
library(RcppArmadillo)
# Use RcppXPtrUtils for simple usage of C++ external pointers
library(RcppXPtrUtils)

# compile user-defined function and return pointer (RcppArmadillo is used as dependency)
euclideanFuncPtr <- cppXPtr(
  "double customDist(const arma::mat &A, const arma::mat &B) {
    return sqrt(arma::accu(arma::square(A - B)));
  }", depends = c("RcppArmadillo"))

# distance matrix for user-defined euclidean distance function (note that method is set to "custom")
parDist(matrix(1:16, ncol=2), method="custom", func = euclideanFuncPtr)
## End(Not run)
```

Index

`call`, [8](#)

`cppXPtr`, [3](#)

`match.arg`, [8](#)

`parallelDist(parDist)`, [2](#)

`parDist`, [2](#), [3](#), [8](#)

`stepPattern`, [4](#)