Package ‘matrixprofiler’

November 23, 2021

Type Package
Title Matrix Profile for R
Version 0.1.7
Maintainer Francisco Bischoff <fbischoff@med.up.pt>
Description This is the core functions needed by the ‘tsmp’ package. The low level and carefully checked mathematical functions are here. These are implementations of the Matrix Profile concept that was created by CS-UCR <http://www.cs.ucr.edu/~eamonn/MatrixProfile.html>.
License GPL-3
URL https://github.com/matrix-profile-foundation/matrixprofiler
BugReports https://github.com/matrix-profile-foundation/matrixprofiler/issues
Depends R (>= 4.0)
Imports checkmate (>= 2.0.0), Rcpp (>= 1.0.6), RcppParallel (>= 5.1.4)
Suggests testthat (>= 3.0.2), debugme (>= 1.1.0), spelling (>= 2.2.0),
LinkingTo Rcpp (>= 1.0.6), RcppParallel (>= 5.1.4), RcppProgress (>= 0.4.2), RcppThread (>= 1.0.0)
Encoding UTF-8
Language en-US
LazyData true
NeedsCompilation yes
RoxygenNote 7.1.2
SystemRequirements GNU make
Config/testthat/edition 3
Config/testthat/parallel true
Author Francisco Bischoff [aut, cre] (<https://orcid.org/0000-0002-5301-8672>),
Michael Yeh [res, ccp, ctb] (<https://orcid.org/0000-0002-9807-2963>),
Diego Silva [res, ccp, ctb] (<https://orcid.org/0000-0002-5184-9413>),
Yan Zhu [res, ccp, ctb] (<https://orcid.org/0000-0002-5952-2108>),
Hoang Dau [res, ccp, ctb] (<https://orcid.org/0000-0003-2439-5185>),
Michele Linardi [res, ccp, ctb] (<https://orcid.org/0000-0002-3249-2068>)
**mass**

Computes the Distance between the 'data' and the 'query'.

**Description**

This algorithm will use a rolling window, to computes the distance thorough the whole data. This means that the minimum distance found is the *motif* and the maximum distance is the *discord* on that time series. **Attention** you need first to create an object using `mass_pre()`. Read below.

**Usage**

```r
mass(
  pre_obj,
  data,
  query = data,
  index = 1,
  version = c("v3", "v2"),
  n_workers = 1
)

mass_pre(
  data,
  window_size,
  query = NULL,
  type = c("normalized", "non_normalized", "absolute", "weighted"),
  weights = NULL
)
```

**Arguments**

- `pre_obj` Required. This is the object resulting from `mass_pre()`. The is no MASS without a `pre`.
- `data` Required. Any 1-dimension series of numbers (matrix, vector, ts etc.)
query Optional. Accepts the same types as data and is used for join-similarity. Defaults to data for self-similarity. IMPORTANT Use the same data used on mass_pre().

index An integer. This is the index of the rolling window that will be used. Must be between 1 and length(data) - window_size + 1.

version A string. Chooses the version of MASS what will be used. Ignored if mass_pre() is not the "normalized" type.

n_workers An integer The number of threads using for computing. Defaults to 1.

window_size Required. An integer defining the rolling window size.

type This changes how the MASS algorithm will compare the rolling window and the data. (See details).

weights Optional. It is used when the type is weighted, and has to be the same size as the window_size.

Details

There are currently four ways to compare the window with the data:

1. normalized: this normalizes the data and the query window. This is the most frequently used.
2. non_normalized: this won’t normalize the query window. The data still being normalized.
3. absolute: this won’t normalize both the data and the query window.
4. weighted: this normalizes the data and query window, and also apply a weight vector on the query.

Value

mass() returns a list with the distance_profile and the last_product that is only useful for computing the Matrix Profile.

mass_pre() returns a list with several precomputations to be used on MASS later. Attention use this before mass().

Examples

pre <- mass_pre(motifs_discords_small, 50)
dist_profile <- mass(pre, motifs_discords_small)
pre <- mass_pre(motifs_discords_small, 50)
dist_profile <- mass(pre, motifs_discords_small)
Description

This package is derived from the former package tsmp. It is intended to make a clear separation of what is the Matrix Profile computation and what are the data mining process we can do using Matrix Profile.

Details

The Matrix Profile, has the potential to revolutionize time series data mining because of its generality, versatility, simplicity and scalability. In particular it has implications for time series motif discovery, time series joins, shapelet discovery (classification), density estimation, semantic segmentation, visualization, rule discovery, clustering etc.

Parallel backend

This package uses RcppParallel in order to do multithreading computations. By default it uses the 'TBB' backend. If for any reason you want to change the backend to 'tinythread', you may use: Sys.setenv(RCPP_PARALLEL_BACKEND = "tinythread"). To configure back to 'TBB', use Sys.setenv(RCPP_PARALLEL_BACKEND = "tbb").

References


Website: http://www.cs.ucr.edu/~eamonn/MatrixProfile.html
motifs_discords_small  Just a synthetic dataset for testing

Description
Just a synthetic dataset for testing

Usage
  motifs_discords_small

Format
A vector with 875 observations

mov_mean  Several moving window functions

Description
These functions do not handle NA values

Usage
  mov_mean(
    data,
    window_size,
    type = c("ogita", "normal", "weighted", "fading"),
    eps = 0.9
  )

  mov_var(
    data,
    window_size,
    type = c("ogita", "normal", "weighted", "fading"),
    eps = 0.9
  )

  mov_sum(
    data,
    window_size,
    type = c("ogita", "normal", "weighted", "fading"),
    eps = 0.9
  )
mov_max(data, window_size)
mov_min(data, window_size)
mov_std(data, window_size, rcpp = TRUE)
movmean_std(data, window_size, rcpp = TRUE)
muinvn(data, window_size, n_workers = 1)
zero_crossing(data, window_size)

Arguments

data 
A vector or a column matrix of numeric.

window_size 
An integer. The size of the rolling window.

type 
A string. Select between several algorithms. Default is ogita (See details).

eps 
A numeric. Used only for fading algorithms (See details), otherwise has no effect.

rcpp 
A logical. If TRUE will use the Rcpp implementation, otherwise will use the R implementation, that may or not be slower.

n_workers 
An integer. The number of threads using for computing. Defaults to 1.

Details

Some functions may use different algorithms to compute the results. The available types are:

1. ogita: This is the default. It uses the Ogita et al., Accurate Sum, and Dot Product for precision. It is not the fastest algorithm, but the time spent vs. guarantee of precision worth it.

2. normal: This uses the cumsum method that is faster, but unreliable in some situations (I have to find the references, but is true).

3. weighted: This uses Rodrigues P., et al. algorithm that uses a weighted window for online purposes. The eps argument controls the factor. (The function is not online yet)

4. fading: This also uses Rodrigues P., et al. algorithm that in this case, uses a fading factor, also for online purposes. The eps argument controls the factor. (The function is not online yet)

Another important detail is that the standard deviation we use for all computations is the population (i.e.: divided by n), not the sample (i.e.: divided by n -1). That is why we also provide the internally the :::std() function that computes the population, differently from stats::sd() that is the sample kind. Further more, movmean_std() shall be used when you need both results in one computation. This is faster than call mov_mean() followed by mov_std(). Finally, muinvn() is kept like that for historical reasons, as it is the function used by mpx(). It returns the sig (stable inverse centered norm) instead of std (sig is equals to 1 / (std * sqrt(window_size))).
**Value**

- `mov_mean()` returns a vector with moving avg.
- `mov_var()` returns a vector with moving var.
- `mov_sum()` returns a vector with moving sum.
- `mov_max()` returns a vector with moving max.
- `mov_min()` returns a vector with moving min.
- `mov_std()` returns a vector with moving sd.
- `movmean_std()` returns a list with vectors of the moving avg, sd, sig, sum and sqrsum.
- `muinvn()` returns a list with vectors of moving avg and sig.
- `zero_crossing()` returns a vector of times the data crossed the 'zero' line inside a rolling window.

**Examples**

```r
mov <- mov_mean(motifs_discords_small, 50)
mov <- mov_var(motifs_discords_small, 50)
mov <- mov_sum(motifs_discords_small, 50)
mov <- mov_max(motifs_discords_small, 50)
mov <- mov_min(motifs_discords_small, 50)
mov <- mov_std(motifs_discords_small, 50)
mov <- movmean_std(motifs_discords_small, 50)
mov <- muinvn(motifs_discords_small, 50)
zero_cross <- zero_crossing(motifs_discords_small, 50)
```

**Description**

STAMP Computes the best so far Matrix Profile and Profile Index for Univariate Time Series.

STOMP is a faster implementation with the caveat that is not anytime as STAMP or SCRIMP.

SCRIMP is a faster implementation, like STOMP, but has the ability to return anytime results as STAMP.

MPX is by far the fastest implementation with the caveat that is not anytime as STAMP or SCRIMP.

**Usage**

```r
stamp(
  data,
  window_size,
  query = NULL,
  exclusion_zone = 0.5,
  s_size = 1,
  n_workers = 1,
  progress = TRUE
)```


crimp(
  data,
  window_size,
  query = NULL,
  exclusion_zone = 0.5,
  s_size = 1,
  pre_scrimp = 0.25,
  n_workers = 1,
  progress = TRUE
)

mpx(
  data,
  window_size,
  query = NULL,
  exclusion_zone = 0.5,
  s_size = 1,
  idxs = TRUE,
  distance = c("euclidean", "pearson"),
  n_workers = 1,
  progress = TRUE
)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>Required. Any 1-dimension series of numbers (matrix, vector, ts etc.) (See details).</td>
</tr>
<tr>
<td>window_size</td>
<td>Required. An integer defining the rolling window size.</td>
</tr>
<tr>
<td>query</td>
<td>(not yet on crimp()) Optional. Another 1-dimension series of numbers for an AB-join similarity. Default is NULL (See details).</td>
</tr>
<tr>
<td>exclusion_zone</td>
<td>A numeric. Defines the size of the area around the rolling window that will be ignored to avoid trivial matches. Default is 0.5, i.e., half of the window size.</td>
</tr>
<tr>
<td>s_size</td>
<td>A numeric. Used on anytime algorithms (stamp, crimp, mpx) if only part of the computation is needed. Default is 1.0 (means 100%).</td>
</tr>
<tr>
<td>n_workers</td>
<td>An integer. The number of threads using for computing. Defaults to 1.</td>
</tr>
<tr>
<td>progress</td>
<td>A logical. If TRUE (the default) will show a progress bar. Useful for long computations. (See details)</td>
</tr>
</tbody>
</table>
The Matrix Profile, has the potential to revolutionize time series data mining because of its generality, versatility, simplicity and scalability. In particular it has implications for time series motif discovery, time series joins, shapelet discovery (classification), density estimation, semantic segmentation, visualization, rule discovery, clustering etc.

progress, it is really recommended to use it as feedback for long computations. It indeed adds some (neglectable) overhead, but the benefit of knowing that your computer is still computing is much bigger than the seconds you may lose in the final benchmark. About \texttt{n\_workers}, for Windows systems, this package uses TBB for multithreading, and Linux and macOS, use TinyThread++. This may or not raise some issues in the future, so we must be aware of slower processing due to different mutexes implementations or even unexpected crashes. The Windows version is usually more reliable. The \texttt{data} and \texttt{query} parameters will be internally converted to a single vector using \texttt{as.numeric()}, thus, bear in mind that a multidimensional matrix may not work as you expect, but most 1-dimensional data types will work normally. If \texttt{query} is provided, expect the same pre-processing done for \texttt{data}; in addition, \texttt{exclusion\_zone} will be ignored and set to 0. Both \texttt{data} and \texttt{query} doesn’t need to have the same size and they can be interchanged if both are provided. The difference will be in the returning object. AB-Join returns the Matrix Profile ‘A’ and ‘B’ i.e., the distance between a rolling window from \texttt{query} to \texttt{data} and from \texttt{data} to \texttt{query}.

\textbf{stamp:}

The anytime STAMP computes the Matrix Profile and Profile Index in such manner that it can be stopped before its complete calculation and return the best so far results allowing ultra-fast approximate solutions.

\textbf{stomp:}

The STOMP uses a faster implementation to compute the Matrix Profile and Profile Index. It can be stopped earlier by the user, but the result is not considered anytime, just incomplete. For a anytime algorithm, use \texttt{stamp()} or \texttt{scrimp()}.

\textbf{scrimp:}

The SCRIMP algorithm was the anytime solution for stomp. It is as fast as stomp but allows the user to cancel the computation and get an approximation of the final result. This implementation uses the SCRIMP++ code. This means that, at first, it will compute the pre-scrimp (a very fast and good approximation), and continue improving with scrimp. The exception is if you use multithreading, that skips the pre-scrimp stage.

\textbf{mpx:}

This algorithm was developed apart from the main Matrix Profile branch that relies on Fast Fourier Transform (FFT) at least in one part of the process. This algorithm doesn’t use FFT at all and is several times faster. It also relies on Ogita’s work for better precision computing mean and standard deviation (part of the process).
Value
Returns a list with the matrix_profile, profile_index (if idxs is TRUE in mpx()), and some information about the settings used to build it, like ez and partial when the algorithm is finished early.

This document
Last updated on 2021-11-23 using R version 4.1.2.

References


Website: http://www.cs.ucr.edu/~eamonn/MatrixProfile.html

See Also
mass() for the underlying algorithm that finds best match of a query.
mpxab() for the forward and reverse join-similarity.

Examples

```r
mp <- stamp(motifs_discords_small, 50)
mp <- stomp(motifs_discords_small, 50)
mp <- scrimp(motifs_discords_small, 50)
mp <- mpx(motifs_discords_small, 50)
```

---

### znorm Math Functions

#### Description

znorm(): Normalizes data for mean Zero and Standard Deviation One
ed_corr(): Converts euclidean distances into correlation values
corr_ed(): Converts correlation values into euclidean distances
mode(): Returns the most common value from a vector of integers
std(): Population SD, as R always calculate with n-1 (sample), here we fix it.
**znorm**

normalize(): Normalizes data to be between min and max.
complexity(): Computes the complexity index of the data
binary_split(): Creates a vector with the indexes of binary split.

**Usage**

znorm(data, rcpp = TRUE)
ed_corr(data, w, rcpp = TRUE)
corr_ed(data, w, rcpp = TRUE)
mode(x, rcpp = FALSE)
std(data, na.rm = FALSE, rcpp = TRUE)
normalize(data, min_lim = 0, max_lim = 1, rcpp = FALSE)
complexity(data)
binary_split(n, rcpp = TRUE)

**Arguments**

data a vector of numeric.
rcpp A logical. If TRUE will use the Rcpp implementation, otherwise will use the R implementation, that may or not be slower.
w the window size
x a vector of integers.
na.rm A logical. If TRUE remove the NA values from the computation.
min_lim A number
max_lim A number
n size of the vector

**Value**

znorm(): Returns the normalized data
ed_corr(): Returns the converted values from euclidean distance to correlation values.
corr_ed(): Returns the converted values from euclidean distance to correlation values.
mode(): Returns the most common value from a vector of integers.
std(): Returns the corrected standard deviation from sample to population.
normalize(): Returns the normalized data between min and max.
complexity(): Returns the complexity index of the data provided (normally a subset).
complexity(): Returns a vector with the binary split indexes.
Examples

```r
normalized <- znorm(motifs_discords_small)
fake_data <- c(rep(3, 100), rep(2, 100), rep(1, 100))
correlation <- ed_corr(fake_data, 50)
fake_data <- c(rep(0.5, 100), rep(1, 100), rep(0.1, 100))
euclidean <- corr_ed(fake_data, 50)
fake_data <- c(1, 1, 4, 5, 2, 3, 1, 7, 9, 4, 5, 2, 3)
mode <- mode(fake_data)
fake_data <- c(1, 1.4, 4.3, 5.1, 2, 3.6, 1.24, 2, 9, 4.3, 5, 2.1, 3)
res <- std(fake_data)
fake_data <- c(1, 1.4, 4.3, 5.1, 2, 3.6, 1.24, 1, 9, 4.3, 5, 2.1, 3)
res <- normalize(fake_data)
fake_data <- c(1, 1.4, 4.3, 5.1, 2, 3.6, 1.24, 8, 9, 4.3, 5, 2.1, 3)
res <- complexity(fake_data)
fake_data <- c(10)
res <- binary_split(fake_data)
```
Index

* datasets
  motifs_discords_small, 5
* matrix profile computations
  stamp, 7
  
  binary_split (znorm), 10
  complexity (znorm), 10
  corr_ed (znorm), 10
  
  ed_corr (znorm), 10
  
  mass, 2
  mass_pre (mass), 2
  matrixprofiler, 4
  mode (znorm), 10
  motifs_discords_small, 5
  mov_max (mov_mean), 5
  mov_mean, 5
  mov_min (mov_mean), 5
  mov_std (mov_mean), 5
  mov_sum (mov_mean), 5
  mov_var (mov_mean), 5
  movmean_std (mov_mean), 5
  mpx (stamp), 7
  muinvn (mov_mean), 5
  
  normalize (znorm), 10
  
  scrimp (stamp), 7
  stamp, 7
  std (znorm), 10
  stomp (stamp), 7
  
  zero_crossing (mov_mean), 5
  znorm, 10