Khiva Documentation

Release v0.5.0

Shapelets.io

May 20, 2020

Table of Contents

This is the documentation of **Khiva** library.

Khiva[1] is an open-source library of efficient algorithms to analyse time series in GPU and CPU. It can be used to extract insights from one or a group of time series. The large number of available methods allow us to understand the nature of each time series. Based on the results of this analysis, users can reduce dimensionality, find out recurrent motifs or discords, understand the seasonality or trend from a given time series, forecasting and detect anomalies.

Khiva provides a mean for time series analytics at scale. These analytics can be exploited in a wide range of use cases across several industries, like energy, finance, e-health, IoT, music industry, etc.

This is just the beginning, so stay tuned as more features are coming ...

Gitter is the place for discussions and questions about Khiva library. We use the GitHub Issue Tracker to manage bug reports and feature requests.

You can jump right into the package by looking into our *Getting Started*.

[1] Time series analysis comprises methods for analyzing time series data in order to extract meaningful statistics and other characteristics of the data (Source Wikipedia).

CHAPTER 1

Getting Started

1.1 Getting the source code

You can download the latest stable released version, or you can get the latest source code version by cloning our git repository:

```
git clone https://github.com/shapelets/khiva
```

1.2 Dependencies

Khiva relies on a number of open source libraries and tools which are required to get it running.

Tools:

- A Build manager to control the compilation process CMake.
- A library dependency manager Conan.
- Python 3.
- Pip3.
- Documentation builders Doxygen and sphinx.
- Graphviz and Dot.
- A C++ compiler, it can be either Clang, GCC or Visual Studio C++ Compiler.

Note: All versions of Khiva Library require a **fully C++11-compliant** compiler.

Libraries:

- OpenCL library for you GPU card (Intel, AMD, or Nvidia).

- To run on accelerators like GPUs, Arrayfire 3.6.2. Note that in order to use Arrayfire on Windows you need to install the Visual Studio 2015 (x64) runtime libraries.
- To test the functionality provided by Khiva, Google Test.
- To benchmark Khiva, Google Benchmark.
- Boost.
- Eigen.

1.3 Windows

1.3.1 Installation

Prerequisites

- Install Python-64bits and add the installation path to the environment variable path, 32-bits version won't work.
- Install ArrayFire 3.6.2 and add the installation path to the environment variable path.
- Install Vcpkg and add the installation path to the environment variable path.
- Install chocolatey to manage windows dependencies and add the installation path to the environment variable path.

Once we have installed all Khiva dependencies, we are ready to install Khiva by using the installers (Option 1) or from source code (Option 2).

(Option 1) Build using a batch installer

In the tools directory you can find the script install.bat. You must indicate the path to your vcpkg installation directory.

- Usage: install.bat <path_to_vcpkg>
- Example: install.bat c:vcpkg

(Option 2) Build from source code

If you prefer, you can build Khiva step by step. First, go to the source directory.

- Run `choco install cmake.install -NoNewWindow -Wait` Note: Add the installation path to the environment variable path and **before** than chocolately environment variable path.
- Run `choco install doxygen.install -NoNewWindow -Wait`.
- Run `choco install graphviz -NoNewWindow -Wait`.
- Run `python -m pip install --upgrade pip`.
- Run `pip3 install sphinx breathe sphinx_rtd_theme`.
- Run `vcpkg install --triplet x64-windows gtest eigen3 benchmark boost`.
- Create a `build` folder in the root path of the project.
- Browse inside the `build` folder.

- Run `cmake .. -DCMAKE_TOOLCHAIN_FILE="<PATH_TO_VPKG>/scripts/buildsystems/vcpkg.cmake" -DKHIVA_USE_CONAN=OFF -G "Visual Studio 15 2017 Win64"` (Note: Replace <PATH_TO_VPKG> with your vcpkg installation path and do not forget to clean the build directory every time before running this command).
- Run `cmake --build . --config Release -- /m` to compile.

Install Khiva library

To install Khiva just execute the following command:

- Run `cmake -DBUILD_TYPE=Release -P cmake_install.cmake`.

1.3.2 Generating the Khiva installer

We use Cpack and NSIS to generate the installer.

Notes: Before generating the installer, the project must be built by following the steps in the previous `Build from source code` section. The generated package is stored in the `build` folder.

- Run `choco install nsis -NoNewWindow -Wait`.
- The installer can be generated running the command `cpack -G NSIS`.

Note: We use the `cpack` command from `cmake`, be aware `chocolatey` has another cpack command. If you cannot run the proper command, check out the path from `cmake` is placed before the path from `chocolatey` in the environment variable path.

1.3.3 Generating documentation

- Run `pip install sphinx` to install Sphinx.
- Browse to the root path of the project.
- Run `sphinx-build.exe -b html doc/sphinx/source/ build/doc/html/`.

1.4 Linux

We use Ubuntu 16.04 LTS as our linux distribution example.

1.4.1 Prerequisites

- Install Python-64bits or run `apt-get install python3 python3-pip`, 32-bits version won't work.
- Download ArrayFire 3.6.2.
- Create destination folder `sudo mkdir -p /opt/arrayfire`
- Install ArrayFire `sudo bash arrayfire/ArrayFire-v3.6.2_Linux_x86_64.sh --prefix=/opt/arrayfire --skip-license`

Once we have installed all Khiva dependencies, we are ready to install Khiva from source code or by using the installers.

1.4.2 Build from source code

First, go to the source directory.

```
conan remote add conan-mpusz https://api.bintray.com/conan/mpusz/conan-mpusz
mkdir build
cd build
conan install .. --build missing
cmake ..
make -j8
make install
```

It installs the library in `/usr/local/lib` and `/usr/local/include` folders.

In case ArrayFire is not installed in the default directory, it is required to add the `Arrayfire lib` folder to the `LD_LIBRARY_PATH` environment variable.

```
export LD_LIBRARY_PATH="/pathToArrayfire/arrayfire/lib:$LD_LIBRARY_PATH"
```

1.4.3 Install Khiva library from source code

- Run `make install`.

1.4.4 Generating the khiva installer

We use CPack to generate the installers from source code.

Notes: Before generating the installer the project should be built following the process explained in the previous `Build from source code` section. The generated package will be stored in the `build` folder.

For linux, either a deb or a rpm installer package can be generated. This could be done by running the command `cpack -G DEB` or `cpack -G RPM` respectively inside the build folder.

1.4.5 Generating documentation

We use `sphinx + doxygen` to generate our documentation. You need to install the following packages:

- Sphinx: `brew install sphinx`.
- Doxygen: `brew install doxygen`.
- Read the Docs Theme: `pip install sphinx_rtd_theme`.
- Breathe: `pip install breathe`.

To generate the khiva documentation run the following command.

- Run `make documentation`.

1.5 Mac OS

1.5.1 Prerequisites

- Install Python-64bits or just run `brew install python3`, 32-bits version won't work.

- Install ArrayFire 3.6.2 and then execute the following lines to move the ArrayFire files from the default installation directory to the system path for libraries:

```
sudo mv /opt/arrayfire/include/* /usr/local/include
sudo mv /opt/arrayfire/lib/* /usr/local/lib
sudo mv /opt/arrayfire/share/* /usr/local/share
sudo rm -rf /opt/arrayfire
```

Once we have installed all Khiva dependencies, we are ready to build and install Khiva, either by using the installers of from source code.

1.5.2 Build from source code

First, go to the directory where the source code is stored:

```
conan remote add conan-mpusz https://api.bintray.com/conan/mpusz/conan-mpusz
mkdir build
cd build
conan install .. --build missing
cmake ..
make -j8
make install
```

It installs the library in `/usr/local/lib` and `/usr/local/include` folders.

1.5.3 Install Khiva library from source code

- Run `make install`.

1.5.4 Generating the khiva installer

For Mac OS, the installer can be generated by running the command `cpack -G productbuild` inside the build folder. Note that, before generating the installer you have to follow the previous `Build from source code` section.

1.5.5 Generating documentation

We use `sphinx + doxygen` to generate our documentation. You will need to install the following packages:

- Sphinx: `brew install sphinx`.
- Doxygen: `brew install doxygen`.
- Read the Docs Theme: `pip install sphinx_rtd_theme`.
- Breathe: `pip install breathe`.

To generate the khiva documentation run the following command.

- `make documentation`.

CHAPTER 2

Khiva API

This is the list of namespaces that comprise the Khiva library.

2.1 Namespace Array

namespace array

Functions

af::array **createArray**(**const** void **data*, unsigned *ndims*, **const** dim_t **dims*, int *type*)
Creates an af::array.

Return af::array Containing the data.

Parameters

- `data`: Data used to create the af::array.
- `ndims`: Number of dimensions of data.
- `dims`: Cardinality of dimensions of data.
- `type`: Data type.

void **deleteArray**(af_array *array*)
Decreases the references count for the given array.

Parameters

- `array`: The *Array* to be deleted.

void **getData**(**const** af::array &*array*, void **data*)
Retrieves the data from the device to the host.

Parameters

- `array`: The *Array* that contains the data to be retrieved.
- `data`: Pointer to a preallocated block of memory in the host.

af::dim4 **getDims**(**const** af::array *&array*)
Returns the dimensions from a given array.

Return af::dim4 The dimensions.

Parameters

- `array`: *Array* from which to get the dimensions.

int **getType**(**const** af::array *&array*)
Gets the type of the array.

Return int Value of the Dtype enumeration.

Parameters

- `array`: The array to obtain the type from.

af::array **join**(int *dim*, **const** af::array *&first*, **const** af::array *&second*)
Joins the first and second arrays along the specified dimension.

Return af::array The result of joining first and second along the specified dimension.

Parameters

- `dim`: The dimension along which the join occurs.
- `first`: The first input array.
- `second`: The second input array.

void **print**(**const** af::array *&array*)
Prints the content of an array.

Parameters

- `array`: The array to be printed.

af::array **from_af_array**(**const** af_array *array*)
Creates an af::array from its af_array C pointer. The resulting array does not acquire the input pointer passed. User of this function is responsible to release it.

Parameters

- `array`: The array to be printed.

af_array **increment_ref_count**(**const** af_array *array*)
Increments the reference count of the af_array C pointer passed throwing if there is an error. The user of this function is responsible to release the returned array by calling deleteArray.

Parameters

- `array`: The array whose reference count is going to be incremented.

template <**typename** T>
std::vector<int> **getRowsWithMaximals**(**const** khiva::*array*::*Array*<T> &*a*)
Gets the indices of all rows containing a maximal.

Return std::vector<int> with the indices of the rows with maximals.

Parameters

- `a`: The input array.

template <**typename** T>
std::vector<int> **getIndexMaxColumns**(**const** std::vector<T> &*r*)
Gets the indices of the columns with maximals.

Return std::vector<int> with the indices of the columns with maximals.

Parameters

- `r`: The input row.

template <class *T*>
class Array
#include </home/docs/checkouts/readthedocs.org/user_builds/khiva/checkouts/latest/include/khiva/array.h>
Array class, This class provides functionality manage Arrays on the host side.

Public Functions

Array()
Default Constructor of *Array* class.

Array(**const** af::array &*in*)
Constructor of *Array* class which receives and af::array.

Parameters
- `in`: The input af::array.

~Array()
Default destructor of *Array* class.

void **setNumX**(int *val*)
Sets the cardinality of the first dimension.

Parameters
- `val`: The value to be set.

void **setNumY**(int *val*)
Sets the cardinality of the second dimension.

Parameters
- `val`: The value to be set.

void **setNumW**(int *val*)
Sets the cardinality of the third dimension.

Parameters
- `val`: The value to be set.

void **setNumZ**(int *val*)
Sets the cardinality of the fourth dimension.

Parameters
- `val`: The value to be set.

void **setData**(T **pd*)
Sets the data to be stored in the *Array*.

Parameters
- `pd`: The data to be stored.

int **getNumX**() **const**
Gets the cardinality of the first dimension.

Return int the Cardinality of the first dimension.

int **getNumY**() **const**
Gets the cardinality of the second dimension.

Return int the Cardinality of the second dimension.

int **getNumW**() **const**
Gets the cardinality of the third dimension.

Return int the Cardinality of the third dimension.

int **getNumZ**() **const**
Gets the cardinality of the fourth dimension.

Return int the Cardinality of the fourth dimension.

int **getNumElements**() **const**
Gets the number of elements in data.

Return int the Cardinality of the number of elements.

std::vector<T> **getRow**(int *idx*) **const**
Gets the row number given by idx.

Return std::vector Containing the selected row.
Parameters
- `idx`: The row number to be extracted.

std::vector<T> **getColumn**(int *idx*) **const**
Gets the column number given by idx.

Return std::vector Containing the selected column.
Parameters
- `idx`: The column number to be extracted.

T **getElement**(int *row*, int *column*) **const**
Gets the element given by row and column.

Return T The element to be extracted.
Parameters
- `row`: The row number.
- `column`: The column number.

T ***getData**()
Gets a pointer to the data stored in the array.

Return T Pointer to data.

bool **isEmpty**()
Checks whether The *Array* is empty or not.

Return True if the *Array* is empty, false otherwise.

void **print**()
Prints the content of the array.

2.2 Namespace Clustering

namespace clustering

Functions

void **kMeans**(**const** af::array &*tss*, int *k*, af::array &*centroids*, af::array &*labels*, float *tolerance* = 0.0000000001, int *maxIterations* = 100)
Calculates the k-means algorithm.

[1] S. Lloyd. 1982. Least squares quantization in PCM. IEEE Transactions on Information Theory, 28, 2, Pages 129-137.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `k`: The number of means to be computed.
- `centroids`: The resulting means or centroids.
- `labels`: The resulting labels of each time series which is the closest centroid.
- `tolerance`: The error tolerance to stop the computation of the centroids.
- `maxIterations`: The maximum number of iterations allowed.

void **kShape**(**const** af::array &*tss*, int *k*, af::array &*centroids*, af::array &*labels*, float *tolerance* = 0.0000000001, int *maxIterations* = 100)
Calculates the k-shape algorithm.

[1] John Paparrizos and Luis Gravano. 2016. k-Shape: Efficient and Accurate Clustering of Time Series. SIGMOD Rec. 45, 1 (June 2016), 69-76.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `k`: The number of means to be computed.
- `centroids`: The resulting means or centroids.
- `labels`: The resulting labels of each time series which is the closest centroid.
- `tolerance`: The error tolerance to stop the computation of the centroids.
- `maxIterations`: The maximum number of iterations allowed.

2.3 Namespace Dimensionality

namespace dimensionality

Typedefs

using khiva::dimensionality::Point = typedef std::pair<float, float>

using khiva::dimensionality::Segment = typedef std::pair<int, int>

Functions

std::vector<Point> **PAA**(**const** std::vector<Point> &*points*, int *bins*)

Piecewise Aggregate Approximation (PAA) approximates a time series X of length n into vector $\bar{X} = (\bar{x}_1, \ldots, \bar{x}_M)$ of any arbitrary length $M \leq n$ where each of $\bar{x}_i$ is calculated as follows:

$$\bar{x}_i = \frac{M}{n} \sum_{j=n/M(i-1)+1}^{(n/M)i} x_j.$$

Which simply means that in order to reduce the dimensionality from n to M, we first divide the original time series into M equally sized frames and secondly compute the mean values for each frame. The sequence assembled from the mean values is the PAA approximation (i.e., transform) of the original time series.

Return result A vector of Points with the reduced dimensionality.

Parameters

- `points`: Set of points.
- `bins`: Sets the total number of divisions.

af::array **PAA**(**const** af::array &*a*, int *bins*)

Piecewise Aggregate Approximation (PAA) approximates a time series X of length n into vector $\bar{X} = (\bar{x}_1, \ldots, \bar{x}_M)$ of any arbitrary length $M \leq n$ where each of $\bar{x}_i$ is calculated as follows:

$$\bar{x}_i = \frac{M}{n} \sum_{j=n/M(i-1)+1}^{(n/M)i} x_j.$$

Which simply means that in order to reduce the dimensionality from n to M, we first divide the original time series into M equally sized frames and secondly compute the mean values for each frame. The sequence assembled from the mean values is the PAA approximation (i.e., transform) of the original time series.

Return af::array An array of points with the reduced dimensionality.

Parameters

- `a`: Set of points.
- `bins`: Sets the total number of divisions.

af::array **PIP**(**const** af::array &*ts*, int *numberIPs*)
Calculates the number of Perceptually Important Points (PIP) in the time series.

[1] Fu TC, Chung FL, Luk R, and Ng CM. Representing financial time series based on data point importance. Engineering Applications of Artificial Intelligence, 21(2):277-300, 2008.

Return af::array Array with the most Perceptually Important numPoints.

Parameters

- `ts`: Expects an input array whose dimension zero is the length of the time series.
- `numberIPs`: The number of points to be returned.

std::vector<Point> **PLABottomUp**(**const** std::vector<Point> &*ts*, float *maxError*)
Applies the Piecewise Linear Approximation (PLA BottomUP) to the time series.

[1] Zhu Y, Wu D, Li Sh (2007). A Piecewise Linear Representation Method of Time Series Based on Feature Points. Knowledge-Based Intelligent Information and Engineering Systems 4693:1066-1072.

Return std::vector Vector with the reduced number of points.

Parameters

- `ts`: Expects an input vector containing the set of points to be reduced.
- `maxError`: The maximum approximation error allowed.

af::array **PLABottomUp**(**const** af::array &*ts*, float *maxError*)
Applies the Piecewise Linear Approximation (PLA BottomUP) to the time series.

[1] Zhu Y, Wu D, Li Sh (2007). A Piecewise Linear Representation Method of Time Series Based on Feature Points. Knowledge-Based Intelligent Information and Engineering Systems 4693:1066-1072.

Return af::array with the reduced number of points.

Parameters

- `ts`: Expects an af::array containing the set of points to be reduced. The first component of the points in the first column and the second component of the points in the second column.
- `maxError`: The maximum approximation error allowed.

std::vector<Point> **PLASlidingWindow**(**const** std::vector<Point> &*ts*, float *maxError*)
Applies the Piecewise Linear Approximation (PLA Sliding Window) to the time series.

[1] Zhu Y, Wu D, Li Sh (2007). A Piecewise Linear Representation Method of Time Series Based on Feature Points. Knowledge-Based Intelligent Information and Engineering Systems 4693:1066-1072.

Return std::vector Vector with the reduced number of points.

Parameters

- `ts`: Expects an input vector containing the set of points to be reduced.
- `maxError`: The maximum approximation error allowed.

af::array **PLASlidingWindow**(**const** af::array &*ts*, float *maxError*)
Applies the Piecewise Linear Approximation (PLA Sliding Window) to the time series.

[1] Zhu Y, Wu D, Li Sh (2007). A Piecewise Linear Representation Method of Time Series Based on Feature Points. Knowledge-Based Intelligent Information and Engineering Systems 4693:1066-1072.

Return af::array with the reduced number of points.

Parameters

- `ts`: Expects an af::array containing the set of points to be reduced. The first component of the points in the first column and the second component of the points in the second column.
- `maxError`: The maximum approximation error allowed.

std::vector<Point> **ramerDouglasPeucker**(**const** std::vector<Point> &*pointList*, double *epsilon*)

The Ramer–Douglas–Peucker algorithm (RDP) is an algorithm for reducing the number of points in a curve that is approximated by a series of points. It reduces a set of points depending on the perpendicular distance of the points and epsilon, the greater epsilon, more points are deleted.

[1] Urs Ramer, "An iterative procedure for the polygonal approximation of plane curves", Computer Graphics and Image Processing, 1(3), 244–256 (1972) doi:10.1016/S0146-664X(72)80017-0.

[2] David Douglas & Thomas Peucker, "Algorithms for the reduction of the number of points required to represent a

digitized line or its caricature", The Canadian Cartographer 10(2), 112–122 (1973) doi:10.3138/FM57-6770-U75U-7727

Return std:vector<khiva::dimensionality::Point> with the selected points.

Parameters

- `pointList`: Set of input points.
- `epsilon`: It acts as the threshold value to decide which points should be considered meaningful or not.

af::array **ramerDouglasPeucker**(**const** af::array &*pointList*, double *epsilon*)

The Ramer–Douglas–Peucker algorithm (RDP) is an algorithm for reducing the number of points in a curve that is approximated by a series of points. It reduces a set of points depending on the perpendicular distance of the points and epsilon, the greater epsilon, more points are deleted.

[1] Urs Ramer, "An iterative procedure for the polygonal approximation of plane curves", Computer Graphics and Image Processing, 1(3), 244–256 (1972) doi:10.1016/S0146-664X(72)80017-0.

[2] David Douglas & Thomas Peucker, "Algorithms for the reduction of the number of points required to represent a

digitized line or its caricature", The Canadian Cartographer 10(2), 112–122 (1973) doi:10.3138/FM57-6770-U75U-7727

Return af::array with the selected points.

Parameters

- `pointList`: Set of input points.
- `epsilon`: It acts as the threshold value to decide which points should be considered meaningful or not.

af::array **SAX**(**const** af::array &*a*, int *alphabetSize*)

Symbolic Aggregate approXimation (SAX). It transforms a numeric time series into a time series of symbols with the same size. The algorithm was proposed by Lin et al.) and extends the PAA-based approach inheriting the original algorithm simplicity and low computational complexity while providing satisfactory sensitivity and selectivity in range query processing. Moreover, the use of a symbolic representation

opened a door to the existing wealth of data-structures and string-manipulation algorithms in computer science such as hashing, regular expression, pattern matching, suffix trees, and grammatical inference.

[1] Lin, J., Keogh, E., Lonardi, S. & Chiu, B. (2003) A Symbolic Representation of Time Series, with Implications for Streaming Algorithms. In proceedings of the 8th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery. San Diego, CA. June 13.

Return result An array of symbols.

Parameters

- `a`: Array with the input time series.
- `alphabetSize`: Number of element within the alphabet.

std::vector<Point> **visvalingam**(**const** std::vector<Point> &*pointList*, int64_t *numPoints*, int64_t *scale* = 1000000000)

Reduces a set of points by applying the Visvalingam method (minimum triangle area) until the number of points is reduced to numPoints.

[1] M. Visvalingam and J. D. Whyatt, Line generalisation by repeated elimination of points, The Cartographic Journal, 1993.

Return std:vector<khiva::dimensionality::Point> where the number of points has been reduced to numPoints.

Parameters

- `pointList`: Expects an input vector of points.
- `numPoints`: Sets the number of points returned after the execution of the method.
- `scale`: Sets the precision used to compute the areas of the triangularization, the longer, the more accurate.

af::array **visvalingam**(**const** af::array &*pointList*, int *numPoints*)

Reduces a set of points by applying the Visvalingam method (minimum triangle area) until the number of points is reduced to numPoints.

[1] M. Visvalingam and J. D. Whyatt, Line generalisation by repeated elimination of points, The Cartographic Journal, 1993.

Return af::array where the number of points has been reduced to numPoints.

Parameters

- `pointList`: Expects an input array formed by to columns where the first column is interpreted as the x cordinate of a point and the second column as the y coordinate.
- `numPoints`: Sets the number of points returned after the execution of the method.

2.4 Namespace Distances

namespace distances

Functions

double **dtw**(**const** std::vector<double> &*a*, **const** std::vector<double> &*b*)
Calculates the Dynamic Time Warping Distance.

Return array The resulting distance between a and b.

Parameters

- `a`: The input time series of reference.
- `b`: The input query.

af::array **dtw**(**const** af::array &*tss*)
Calculates the Dynamic Time Warping Distance.

Return af::array An upper triangular matrix where each position corresponds to the distance between two time series. Diagonal elements will be zero. For example: Position row 0 column 1 records the distance between time series 0 and time series 1.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

af::array **euclidean**(**const** af::array &*tss*)
Calculates euclidean distances between time series.

Return af::array An upper triangular matrix where each position corresponds to the distance between two time series. Diagonal elements will be zero. For example: Position row 0 column 1 records the distance between time series 0 and time series 1.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

af::array **hamming**(**const** af::array &*tss*)
Calculates hamming distances between time series.

Return af::array An upper triangular matrix where each position corresponds to the distance between two time series. Diagonal elements will be zero. For example: Position row 0 column 1 records the distance between time series 0 and time series 1.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

af::array **manhattan**(**const** af::array &*tss*)
Calculates manhattan distances between time series.

Return af::array An upper triangular matrix where each position corresponds to the distance between two time series. Diagonal elements will be zero. For example: Position row 0 column 1 records the distance between time series 0 and time series 1.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

af::array **sbd**(**const** af::array &*tss*)

Calculates the Shape-Based distance (SBD). It computes the normalized cross-correlation and it returns 1.0 minus the value that maximizes the correlation value between each pair of time series.

Return array An upper triangular matrix where each position corresponds to the distance between two time series. Diagonal elements will be zero. For example: Position row 0 column 1 records the distance between time series 0 and time series 1.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

af::array **squaredEuclidean**(**const** af::array &*tss*)

Calculates non squared version of the euclidean distance.

Return array An upper triangular matrix where each position corresponds to the distance between two time series. Diagonal elements will be zero. For example: Position row 0 column 1 records the distance between time series 0 and time series 1.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

2.5 Namespace Features

namespace features

Typedefs

using khiva::features::AggregationFuncDimT = typedef af::array (*)(const af::array &,

using khiva::features::AggregationFuncBoolDimT = typedef af::array (*)(const af::array

using khiva::features::AggregationFuncInt = typedef af::array (*)(const af::array &, c

Functions

af::array **absEnergy**(**const** af::array &*base*)

Calculates the absolute energy of the time series which is the sum over the squared values.

$$E = \sum_{i=1,\ldots,n} x_i^2$$

.

Return af::array An array with the same dimensions as tss, whose values (time series in dimension 0) contains the sum of the squares values in the time series.

Parameters

- `base`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

af::array **absoluteSumOfChanges**(**const** af::array *&tss*)

Calculates the sum over the absolute value of consecutive changes in the time series.

$$\sum_{i=1,\ldots,n-1} \mid x_{i+1} - x_i \mid$$

.

Return af::array An array with the same dimensions as tss, whose values (time series in dimension 0) contains absolute value of consecutive changes in the time series.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

af::array **aggregatedAutocorrelation**(**const** af::array *&tss*, AggregationFuncBoolDimT *aggregationFunction*)

Calculates the value of an aggregation function f_agg (e.g. var or mean) of the autocorrelation (Compare to http://en.wikipedia.org/wiki/Autocorrelation#Estimation), taken over different all possible lags (1 to length of x).

$$\frac{1}{n-1} \sum_{l=1,\ldots,n} \frac{1}{(n-l)\sigma^2} \sum_{t=1}^{n-l} (X_t - \mu)(X_{t+l} - \mu),$$

where n is the length of the time series X_i, σ^2 its variance and μ its mean.

Return af::array An array with the same dimensions as tss, whose values (time series in dimension 0) contains the aggregated correlation for each time series.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `aggregationFunction`: The function to summarise all autocorrelation with different lags.

af::array **aggregatedAutocorrelation**(**const** af::array *&tss*, AggregationFuncDimT *aggregationFunction*)

Calculates the value of an aggregation function f_agg (e.g. var or mean) of the autocorrelation (Compare to http://en.wikipedia.org/wiki/Autocorrelation#Estimation), taken over different all possible lags (1 to length of x).

$$\frac{1}{n-1} \sum_{l=1,\ldots,n} \frac{1}{(n-l)\sigma^2} \sum_{t=1}^{n-l} (X_t - \mu)(X_{t+l} - \mu),$$

where n is the length of the time series X_i, σ^2 its variance and μ its mean.

Return af::array An array with the same dimensions as tss, whose values (time series in dimension 0) contains the aggregated correlation for each time series.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

- `aggregationFunction`: The function to summarise all autocorrelation with different lags.

af::array **aggregatedAutocorrelation**(**const** af::array &*tss*, AggregationFuncInt *aggregationFunction*)

Calculates the value of an aggregation function f_agg (e.g. var or mean) of the autocorrelation (Compare to http://en.wikipedia.org/wiki/Autocorrelation#Estimation), taken over different all possible lags (1 to length of x).

$$\frac{1}{n-1}\sum_{l=1,\ldots,n}\frac{1}{(n-l)\sigma^2}\sum_{t=1}^{n-l}(X_t-\mu)(X_{t+l}-\mu),$$

where n is the length of the time series X_i, σ^2 its variance and μ its mean.

Return af::array An array with the same dimensions as tss, whose values (time series in dimension 0) contains the aggregated correlation for each time series.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `aggregationFunction`: The function to summarise all autocorrelation with different lags.

void **aggregatedLinearTrend**(**const** af::array &*t*, long *chunkSize*, AggregationFuncDimT *aggregationFunction*, af::array &*slope*, af::array &*intercept*, af::array &*rvalue*, af::array &*pvalue*, af::array &*stderrest*)

Calculates a linear least-squares regression for values of the time series that were aggregated over chunks versus the sequence from 0 up to the number of chunks minus one.

Parameters

- `t`: The time series to calculate the features of.
- `chunkSize`: The chunkSize used to aggregate the data.
- `aggregationFunction`: Function to be used in the aggregation.
- `slope`: Slope of the regression line.
- `intercept`: Intercept of the regression line.
- `rvalue`: Correlation coefficient.
- `pvalue`: Two-sided p-value for a hypothesis test whose null hypothesis is that the slope is zero, using Wald Test with t-distribution of the test statistic.
- `stderrest`: Standard error of the estimated gradient.

void **aggregatedLinearTrend**(**const** af::array &*t*, long *chunkSize*, AggregationFuncInt *aggregationFunction*, af::array &*slope*, af::array &*intercept*, af::array &*rvalue*, af::array &*pvalue*, af::array &*stderrest*)

Calculates a linear least-squares regression for values of the time series that were aggregated over chunks versus the sequence from 0 up to the number of chunks minus one.

Parameters

- `t`: The time series to calculate the features of.
- `chunkSize`: The chunkSize used to aggregate the data.
- `aggregationFunction`: Function to be used in the aggregation.

- `slope`: Slope of the regression line.
- `intercept`: Intercept of the regression line.
- `rvalue`: Correlation coefficient.
- `pvalue`: Two-sided p-value for a hypothesis test whose null hypothesis is that the slope is zero, using Wald Test with t-distribution of the test statistic.
- `stderrest`: Standard error of the estimated gradient.

af::array **`approximateEntropy`**(**`const`** af::array &*tss*, int *m*, float *r*)

Calculates a vectorized Approximate entropy algorithm (https://en.wikipedia.org/wiki/Approximate_entropy). For short time series, this method is highly dependent on the parameters, but should be stable for N > 2000, see:

[1] Yentes et al., The Appropriate Use of Approximate Entropy and Sample Entropy with Short Data Sets, (2012). Other shortcomings and alternatives discussed in: Richman & Moorman, Physiological time-series analysis using approximate entropy and sample entropy, (2000).

Return af::array An array with the same dimensions as tss, whose values (time series in dimension 0) contains the vectorized Approximate entropy for all the input time series in tss.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `m`: Length of compared run of data.
- `r`: Filtering level, must be positive.

af::array **`autoCorrelation`**(**`const`** af::array &*tss*, long *maxLag*, bool *unbiased* = false)

Calculates the autocorrelation of the specified lag for the given time series, according to the formula [1].

$$\frac{1}{(n-l)\sigma^2}\sum_{t=1}^{n-l}(X_t-\mu)(X_{t+l}-\mu),$$

where n is the length of the time series X_i, σ^2 its variance and μ its mean, l denotes the lag.

[1] https://en.wikipedia.org/wiki/Autocorrelation#Estimation

Return af::array The autocorrelation value for the given time series.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `maxLag`: The maximum lag to compute.
- `unbiased`: Determines whether it divides by (n - lag) (if true), or n (if false).

af::array **`autoCovariance`**(**`const`** af::array &*xss*, bool *unbiased* = false)

Calculates the auto-covariance the given time series.

Return af::array The auto-covariance value for the given time series.

Parameters

- `xss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `unbiased`: Determines whether it divides by n - lag (if true) or n (if false).

af::array **`binnedEntropy`**(**`const`** af::array &*tss*, int *max_bins*)

Calculates the binned entropy for the given time series and number of bins. It calculates the value of:

$$\sum_{k=0}^{min(max\_bins,len(x))} p_k log(p_k) \cdot \mathbf{1}_{(p_k>0)},$$

where p_k is the percentage of samples in bin k.

Return af::array The binned entropy value for the given time series.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `max_bins`: The number of bins.

af::array **`c3`**(**`const`** af::array &*tss*, long *lag*)

This function calculates the value of:

$$\frac{1}{n-2lag} \sum_{i=0}^{n-2lag} x_{i+2\cdot lag}^2 \cdot x_{i+lag} \cdot x_i,$$

which is:

$$\mathbb{E}[L^2(X)^2 \cdot L(X) \cdot X],$$

where $\mathbb{E}$ is the mean and L is the lag operator. It was proposed in [1] as a measure of non linearity in the time series.

[1] Schreiber, T. and Schmitz, A., Discrimination power of measures for nonlinearity in a time series, PHYSICAL REVIEW E, VOLUME 55, NUMBER 5, (1997).

Return af::array The non-linearity value for the given time series.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `lag`: The lag.

af::array **`cidCe`**(**`const`** af::array &*tss*, bool *zNormalize* = false)

This function calculator is an estimate for a time series complexity [1]. It calculates the value of:

$$\sqrt{\sum_{i=0}^{n-2lag} (x_i - x_{i+1})^2}.$$

.

[1] Batista, Gustavo EAPA, et al (2014). CID: an efficient complexity-invariant distance for time series. Data Mining and Knowledge Difscovery 28.3 (2014): 634-669.

Return af::array The complexity value for the given time series.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `zNormalize`: Controls whether the time series should be z-normalized or not.

af::array **`countAboveMean`**(**`const`** af::array *&tss*)
Calculates the number of values in the time series that are higher than the mean.

Return af::array The number of values in the time series that are higher than the mean.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

af::array **`countBelowMean`**(**`const`** af::array *&tss*)
Calculates the number of values in the time series that are lower than the mean.

Return af::array The number of values in the time series that are lower than the mean.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

af::array **`crossCovariance`**(**`const`** af::array *&xss*, **`const`** af::array *&yss*, bool *unbiased* = true)
Calculates the cross-covariance of the given time series.

Return af::array The cross-covariance value for the given time series.

Parameters

- `xss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `yss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `unbiased`: Determines whether it divides by n - lag (if true) or n (if false).

af::array **`crossCorrelation`**(**`const`** af::array *&xss*, **`const`** af::array *&yss*, bool *unbiased* = true)
Calculates the cross-correlation of the given time series.

Return af::array The cross-correlation value for the given time series.

Parameters

- `xss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `yss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `unbiased`: Determines whether it divides by n - lag (if true) or n (if false).

af::array **cwtCoefficients**(**const** af::array &*tss*, **const** af::array &*widths*, int *coeff*, int *w*)

Calculates a Continuous wavelet transform for the Ricker wavelet, also known as the "Mexican hat wavelet" which is defined by:

$$\frac{2}{\sqrt{3a}\pi^{\frac{1}{4}}}(1-\frac{x^2}{a^2})exp(-\frac{x^2}{2a^2}),$$

where a is the width parameter of the wavelet function. This feature calculator takes three different parameter: widths, coeff and w. The feature calculator takes all the different widths arrays and then calculates the cwt one time for each different width array. Then the values for the different coefficient for coeff and width w are returned.

Return af::array Result of calculated coefficients.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `widths`: Array that contains all different widths.
- `coeff`: Coefficient of interest.
- `w`: Width of interest.

af::array **energyRatioByChunks**(af::array *tss*, long *numSegments*, long *segmentFocus*)

Calculates the sum of squares of chunk i out of N chunks expressed as a ratio with the sum of squares over the whole series. segmentFocus should be lower than the number of segments.

Return af::array The energy ratio by chunk of the time series.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `numSegments`: The number of segments to divide the series into.
- `segmentFocus`: The segment number (starting at zero) to return a feature on.

af::array **fftAggregated**(**const** af::array &*tss*)

Calculates the spectral centroid (mean), variance, skew, and kurtosis of the absolute fourier transform spectrum.

Return af::array The spectral centroid (mean), variance, skew, and kurtosis of the absolute fourier transform spectrum.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

void **fftCoefficient**(**const** af::array &*tss*, long *coefficient*, af::array &*real*, af::array &*imag*, af::array &*abs*, af::array &*angle*)

Calculates the fourier coefficients of the one-dimensional discrete Fourier Transform for real input by using fast fourier transformation algorithm,

$$A_k = \sum_{m=0}^{n-1} a_m \exp\left\{-2\pi i \frac{mk}{n}\right\}, \qquad k = 0, \ldots, n-1.$$

.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `coefficient`: The coefficient to extract from the FFT.
- `real`: The real part of the coefficient.
- `imag`: The imaginary part of the coefficient.
- `abs`: The absolute value of the coefficient.
- `angle`: The angle of the coefficient.

af::array **`firstLocationOfMaximum`**(**`const`** af::array &*tss*)
Calculates the first relative location of the maximal value for each time series.

Return af::array The first relative location of the maximum value to the length of the time series, for each time series.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

af::array **`firstLocationOfMinimum`**(**`const`** af::array &*tss*)
Calculates the first location of the minimal value of each time series. The position is calculated relatively to the length of the series.

Return af::array the first relative location of the minimal value of each time series.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

af::array **`friedrichCoefficients`**(**`const`** af::array &*tss*, int *m*, float *r*)
Coefficients of polynomial $h(x)$, which has been fitted to the deterministic dynamics of Langevin model:

$$\dot(x)(t) = h(x(t)) + R(N)(0, 1)$$

as described by [1]. For short time series this method is highly dependent on the parameters.

[1] Friedrich et al., Physics Letters A 271, p. 217-222, Extracting model equations from experimental data, (2000).

Return af::array The coefficients for each time series.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `m`: Order of polynom to fit for estimating fixed points of dynamics.
- `r`: Number of quantiles to use for averaging.

af::array **`hasDuplicates`**(**`const`** af::array &*tss*)
Computes if the input time series contain duplicated elements.

Return af::array Array containing True if the time series contains duplicated elements and false otherwise.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

af::array **hasDuplicateMax**(**const** af::array &*tss*)
Computes if the maximum within time series is duplicated.

Return af::array Array containing True if the maximum value of the time series is duplicated and false otherwise.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

af::array **hasDuplicateMin**(**const** af::array &*tss*)
Computes if the minimum of input time series is duplicated.

Return af::array Array containing True if the minimum of the time series is duplicated and false otherwise.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

af::array **indexMassQuantile**(**const** af::array &*tss*, float *q*)
Calculates the relative index i where $q\%$ of the mass of the time series within tss lie at the left of i. For example for $q = 50\%$ this feature calculator will return the mass center of the time series.

Return af::array The relative indices i where q% of the mass of the time series lie at the left of i.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `q`: The quantile limit.

af::array **kurtosis**(**const** af::array &*tss*)
Returns the kurtosis of tss (calculated with the adjusted Fisher-Pearson standardized moment coefficient G2).

Return af::array The kurtosis of tss.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

af::array **largeStandardDeviation**(**const** af::array &*tss*, float *r*)
Checks if the time series within tss have a large standard deviation.

$$std(x) > r * (max(X) - min(X)).$$

.

Return af::array Array containing True for those time series in tss that have a large standard deviation.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `r`: The threshold value.

af::array **`lastLocationOfMaximum`**(**`const`** af::array *&tss*)
Calculates the last location of the maximum value of each time series. The position is calculated relatively to the length of the series.

Return af::array The last relative location of the maximum value of each time series.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

af::array **`lastLocationOfMinimum`**(**`const`** af::array *&tss*)
Calculates the last location of the minimum value of each time series. The position is calculated relatively to the length of the series.

Return af::array The last relative location of the minimum value of each series.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

af::array **`length`**(**`const`** af::array *&tss*)
Returns the length of the input time series.

Return af::array The length of tss.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

void **`linearTrend`**(**`const`** af::array *&tss*, af::array *&pvalue*, af::array *&rvalue*, af::array *&intercept*, af::array *&slope*, af::array *&stder*)
Calculate a linear least-squares regression for the values of the time series versus the sequence from 0 to length of the time series minus one.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `pvalue`: The p-values for all time series.
- `rvalue`: The r-values for all time series.
- `intercept`: The intercept values for all time series.
- `slope`: The slope for all time series.
- `stder`: The stderr values for all time series.

af::array **localMaximals**(**const** af::array &*tss*)
Calculates all Local Maximals for the time series in tss.

Return af::array The calculated local maximals for each time series in tss.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

af::array **longestStrikeAboveMean**(**const** af::array &*tss*)
Calculates the length of the longest consecutive subsequence in tss that is bigger than the mean of tss.

Return af::array the length of the longest consecutive subsequence in the input time series that is bigger than the mean.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

af::array **longestStrikeBelowMean**(**const** af::array &*tss*)
Calculates the length of the longest consecutive subsequence in tss that is below the mean of tss.

Return af::array The length of the longest consecutive subsequence in the input time series that is below the mean.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

af::array **maxLangevinFixedPoint**(**const** af::array &*tss*, int *m*, float *r*)
Largest fixed point of dynamics $\max_x h(x) = 0$ estimated from polynomial $h(x)$, which has been fitted to the deterministic dynamics of Langevin model:

$$\dot{x}(t) = h(x(t)) + R(N)(0, 1)$$

.

[1] Friedrich et al., Extracting model equations from experimental data, Physics Letters A 271, p. 217-222, (2000).

Return af::array Largest fixed point of deterministic dynamics.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series. NOTE: the time series should be sorted.
- `m`: Order of polynom to fit for estimating fixed points of dynamics.
- `r`: Number of quantiles to use for averaging.

af::array **maximum**(**const** af::array &*tss*)
Calculates the maximum value for each time series within tss.

Return af::array The maximum value of each time series within tss.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

af::array **mean**(**const** af::array &*tss*)
Calculates the mean value for each time series within tss.

Return af::array The mean value of each time series within tss.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

af::array **meanAbsoluteChange**(**const** af::array &*tss*)
Calculates the mean over the absolute differences between subsequent time series values in tss.

$$\frac{1}{n} \sum_{i=1,\ldots,n-1} |x_{i+1} - x_i|.$$

.

Return af::array The mean over the absolute differences between subsequent time series values.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

af::array **meanChange**(**const** af::array &*tss*)
Calculates the mean over the differences between subsequent time series values in tss.

$$\frac{1}{n} \sum_{i=1,\ldots,n-1} x_{i+1} - x_i.$$

.

Return af::array The mean over the differences between subsequent time series values.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

af::array **meanSecondDerivativeCentral**(**const** af::array &*tss*)
Calculates mean value of a central approximation of the second derivative for each time series in tss.

$$\frac{1}{n} \sum_{i=1,\ldots,n-1} \frac{1}{2}(x_{i+2} - 2 \cdot x_{i+1} + x_i).$$

.

Return af::array The mean value of a central approximation of the second derivative for each time series.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

af::array **median**(**const** af::array &*tss*)
Calculates the median value for each time series within tss.

Return af::array The median value of each time series within tss.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

af::array **minimum**(**const** af::array &*tss*)
Calculates the minimum value for each time series within tss.

Return af::array The minimum value of each time series within tss.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

af::array **numberCrossingM**(**const** af::array &*tss*, int *m*)
Calculates the number of m-crossings. A m-crossing is defined as two sequential values where the first value is lower than m and the next is greater, or viceversa. If you set m to zero, you will get the number of zero crossings.

Return af::array The number of m-crossings of each time series within tss.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `m`: The m value.

af::array **numberCwtPeaks**(**const** af::array &*tss*, int *maxW*)
This feature calculator searches for different peaks. To do so, the time series is smoothed by a ricker wavelet and for widths ranging from 1 to maxW. This feature calculator returns the number of peaks that occur at enough width scales and with sufficiently high Signal-to-Noise-Ratio (SNR).

Return af::array The number of peaks for each time series.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `maxW`: The maximum width to consider.

af::array **numberPeaks**(af::array *tss*, int *n*)
Calculates the number of peaks of at least support n in the time series tss. A peak of support n is defined as a subsequence of tss where a value occurs, which is bigger than its n neighbourgs to the left and to the right.

[1] Bioinformatics (2006) 22 (17): 2059-2065. doi: 10.1093/bioinformatics/btl355, http://bioinformatics.oxfordjournals.org/content/22/17/2059.long

Return af::array The number of peaks of at least support n.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `n`: The support of the peak.

af::array **`partialAutocorrelation`**(**`const`** af::array &*tss*, **`const`** af::array &*lags*)

Calculates the value of the partial autocorrelation function at the given lag. The lag k partial autocorrelation of a time series $\{x_t, t = 1 \dots T\}$ equals the partial correlation of x_t and x_{t-k}, adjusted for the intermediate variables $\{x_{t-1}, \dots, x_{t-k+1}\}$ ([1]). Following [2], it can be defined as:

$$\alpha_k = \frac{Cov(x_t, x_{t-k}|x_{t-1}, \dots, x_{t-k+1})}{\sqrt{Var(x_t|x_{t-1}, \dots, x_{t-k+1})Var(x_{t-k}|x_{t-1}, \dots, x_{t-k+1})}}$$

with (a) $x_t = f(x_{t-1}, \dots, x_{t-k+1})$ and (b) $x_{t-k} = f(x_{t-1}, \dots, x_{t-k+1})$ being AR(k-1) models that can be fitted by OLS. Be aware that in (a), the regression is done on past values to predict x_t whereas in (b), future values are used to calculate the past value x_{t-k}. It is said in [1] that, for an AR(p), the partial autocorrelations α_k will be nonzero for $k <= p$ and zero for $k > p$. With this property, it is used to determine the lag of an AR-Process.

[1] Box, G. E., Jenkins, G. M., Reinsel, G. C., & Ljung, G. M. (2015). Time series analysis: forecasting and control. John Wiley & Sons.

[2] https://onlinecourses.science.psu.edu/stat510/node/62

Return af::array The partial autocorrelation for each time series for the given lag.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `lags`: Indicates the lags to be calculated.

af::array **`percentageOfReoccurringDatapointsToAllDatapoints`**(**`const`** af::array &*tss*, bool *isSorted* = false)

Calculates the percentage of unique values, that are present in the time series more than once.

$$\frac{len(\textit{different values occurring more than once})}{len(\textit{different values})}$$

This means the percentage is normalized to the number of unique values, in contrast to the percentageOfReoccurringValuesToAllValues.

Return af::array The percentage of unique data points, that are present in the time series more than once.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `isSorted`: Indicates if the input time series is sorted or not. Defaults to false.

af::array **`percentageOfReoccurringValuesToAllValues`**(**`const`** af::array &*tss*, bool *isSorted* = false)

Calculates the percentage of unique values, that are present in the time series more than once.

$$\frac{\textit{number of data points occurring more than once}}{\textit{number of all data points})}$$

This means the percentage is normalized to the number of unique values, in contrast to the percentageOfReoccurringDatapointsToAllDatapoints.

Return af::array The percentage of unique values, that are present in the time series more than once.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `isSorted`: Indicates if the input time series is sorted or not. Defaults to false.

af::array **`quantile`**(**`const`** af::array &*tss*, **`const`** af::array &*q*, float *precision* = 100000000)
Returns values at the given quantile.

Return af::array Values at the given quantile.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `q`: Percentile(s) at which to extract score(s). One or many.
- `precision`: Number of decimals expected.

af::array **`rangeCount`**(**`const`** af::array &*tss*, float *min*, float *max*)
Counts observed values within the interval [min, max).

Return af::array Values at the given range.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `min`: Value that sets the lower limit.
- `max`: Value that sets the upper limit.

af::array **`ratioBeyondRSigma`**(**`const`** af::array &*tss*, float *r*)
Calculates the ratio of values that are more than $r * std(x)$ (so r sigma) away from the mean of x.

Return af::array The ratio of values that are more than $r * std(x)$ (so r sigma) away from the mean of x.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `r`: Number of times that the values should be away from.

af::array **`ratioValueNumberToTimeSeriesLength`**(**`const`** af::array &*tss*)
Calculates a factor which is 1 if all values in the time series occur only once, and below one if this is not the case. In principle, it just returns:

$$\frac{number\_unique\_values}{number\_values}$$

.

Return af::array The ratio of unique values with respect to the total number of values.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

af::array **`sampleEntropy`**(**`const`** af::array &*tss*)

Calculates a vectorized sample entropy algorithm. For short time-series this method is highly dependent on the parameters, but should be stable for N > 2000, see:

[1] Yentes et al., The Appropriate Use of Approximate Entropy and Sample Entropy with Short Data Sets, (2012).

[2] Richman & Moorman,Physiological time-series analysis using approximate entropy and sample entropy, (2000).

[3] https://en.wikipedia.org/wiki/Sample_entropy

[4] https://www.ncbi.nlm.nih.gov/pubmed/10843903?dopt=Abstract

Return af::array With the same dimensions as tss, whose values (time series in dimension 0) contains the vectorized sample entropy for all the input time series in tss.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

af::array **`skewness`**(**`const`** af::array &*tss*)

Calculates the sample skewness of tss (calculated with the adjusted Fisher-Pearson standardized moment coefficient G1).

Return af::array Containing the skewness of each time series in tss.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

af::array **`spktWelchDensity`**(**`const`** af::array &*tss*, int *coeff*)

Estimates the cross power spectral density of the time series tss at different frequencies. To do so, the time series is first shifted from the time domain to the frequency domain. Welch's method computes an estimate of the power spectral density by dividing the data into overlapping segments, computing a modified periodogram for each segment and averaging the periodograms.

[1] P. Welch, "The use of the fast Fourier transform for the estimation of power spectra: A method based on time

averaging over short, modified periodograms", IEEE Trans. Audio Electroacoust. vol. 15, pp. 70-73, 1967.

[2] M.S. Bartlett, "Periodogram Analysis and Continuous Spectra", Biometrika, vol. 37, pp. 1-16, 1950.

[3] Rabiner, Lawrence R., and B. Gold. "Theory and Application of Digital Signal Processing" Prentice-Hall, pp. 414-419, 1975.

Return af::array Containing the power spectrum of the different frequencies for each time series in tss.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `coeff`: The coefficient to be returned.

af::array **standardDeviation**(**const** af::array &*tss*)
Calculates the standard deviation of each time series within tss.

Return af::array The standard deviation of each time series within tss.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

af::array **sumOfReoccurringDatapoints**(**const** af::array &*tss*, bool *isSorted* = false)
Calculates the sum of all data points, that are present in the time series more than once.

Return af::array The sum of all data points, that are present in the time series more than once.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `isSorted`: Indicates if the input time series is sorted or not. Defaults to false.

af::array **sumOfReoccurringValues**(**const** af::array &*tss*, bool *isSorted* = false)
Calculates the sum of all values, that are present in the time series more than once.

Return af::array Returns the sum of all values, that are present in the time series more than once.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `isSorted`: Indicates if the input time series is sorted or not. Defaults to false.

af::array **sumValues**(**const** af::array &*tss*)
Calculates the sum over the time series tss.

Return af::array An array containing the sum of values in each time series.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

af::array **symmetryLooking**(**const** af::array &*tss*, float *r*)
Calculates if the distribution of tss *looks symmetric*. This is the case if

$$|mean(tss) - median(tss)| < r * (max(tss) - min(tss)).$$

.

Return af::array Denoting if the input time series look symmetric.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `r`: The percentage of the range to compare with.

af::array **timeReversalAsymmetryStatistic**(**const** af::array &*tss*, int *lag*)
This function calculates the value of:

$$\frac{1}{n-2lag}\sum_{i=0}^{n-2lag} x_{i+2\cdot lag}^2 \cdot x_{i+lag} - x_{i+lag} \cdot x_i^2,$$

which is:

$$\mathbb{E}[L^2(X)^2 \cdot L(X) - L(X) \cdot X^2],$$

where $\mathbb{E}$ is the mean and L is the lag operator. It was proposed in [1] as a promising feature to extract from time series.

[1] Fulcher, B.D., Jones, N.S. (2014). Highly comparative feature-based time-series classification. Knowledge and Data Engineering, IEEE Transactions on 26, 3026–3037.

Return af::array Containing the time reversal asymmetry statistic value in each time series.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `lag`: The lag to be computed.

af::array **valueCount**(**const** af::array &*tss*, float *v*)
Counts occurrences of value in the time series tss.

Return af::array Containing the count of the given value in each time series.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `v`: The value to be counted.

af::array **variance**(**const** af::array &*tss*)
Computes the variance for the time series tss.

Return af::array An array containing the variance in each time series.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

af::array **varianceLargerThanStandardDeviation**(**const** af::array &*tss*)
Calculates if the variance of tss is greater than the standard deviation. In other words, if the variance of tss is larger than 1.

Return af::array Denoting if the variance of tss is greater than the standard deviation.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

2.6 Namespace Library

namespace library

Typedefs

typedef *khiva_backend* **Backend**

Enums

enum khiva_backend

Values:

KHIVA_BACKEND_DEFAULT = af::Backend::AF_BACKEND_DEFAULT
Default backend order: OpenCL -> CUDA -> CPU.

KHIVA_BACKEND_CPU = af::Backend::AF_BACKEND_CPU
CPU a.k.a sequential algorithms.

KHIVA_BACKEND_CUDA = af::Backend::AF_BACKEND_CUDA
CUDA Compute Backend.

KHIVA_BACKEND_OPENCL = af::Backend::AF_BACKEND_OPENCL
OpenCL Compute Backend.

Functions

std::string **backendInfo**()
Get information from the active backend.

Return std::string The information of the backend.

void **setBackend**(khiva::*library*::*Backend* *be*)
Set the backend.

Parameters

- `be`: The desired backend.

khiva::*library*::*Backend* **getBackend**()
Get the active backend.

Return *khiva::library::Backend* The active backend.

int **getBackends**()
Get the available backends.

Return int The available backends.

void **setDevice**(int *device*)
Set the device.

Parameters

- `device`: The desired device.

int **getDevice**()
Get the active device.

Return int The active device.

int **getDeviceCount**()
Get the device count.

Return int The device count.

void **setDeviceMemoryInGB**(double *memory*)
Set the memory of the device in use. This information is used for splitting some algorithms and execute them in batch mode. The default value used if it is not set is 4GB.

Parameters

- `memory`: The device memory.

namespace internal

Enums

enum Complexity
Values:

LINEAR

CUADRATIC

CUBIC

Functions

void **setDeviceMemoryInGB**(double *memory*)
Set the memory of the device in use. This information is used for splitting some algorithms and execute them in batch mode. The default value used if it is not set is 4GB.

Parameters
- `memory`: The device memory.

long **getValueScaledToMemoryDevice**(long *value*, *Complexity complexity*)
Get the value scaled to the memory of the device taking into account the Memory complexity.

Return the scaled value.
Parameters
- `value`: The value to scale.
- `complexity`: The complexity to scale with.

2.7 Namespace LinAlg

namespace linalg

Functions

af::array **lls**(**const** af::array &*A*, **const** af::array &*b*)

Calculates the minimum norm least squares solution x ($\|Ax - b\|^2$) to $Ax = b$. This function uses the singular value decomposition function of Arrayfire. The actual formula that this function computes is $x = VD\dagger U^T b$. Where U and V are orthogonal matrices and $D\dagger$ contains the inverse values of the singular values contained in D if they are not zero, and zero otherwise.

Return af::array Contains the solution to the linear equation problem minimizing the norm 2.

Parameters

- `A`: Coefficient matrix containing the coefficients of the linear equation problem to solve.
- `b`: Vector with the measured values.

2.8 Namespace Matrix

namespace matrix

Functions

void **findBestNOccurrences**(**const** af::array &*q*, **const** af::array &*t*, long *n*, af::array &*distances*, af::array &*indexes*)

Calculates the N best matches of several queries in several time series.

The result has the following structure:

- 1st dimension corresponds to the nth best match.
- 2nd dimension corresponds to the number of queries.
- 3rd dimension corresponds to the number of time series.

For example, the distance in the position (1, 2, 3) corresponds to the second best distance of the third query in the fourth time series. The index in the position (1, 2, 3) is the is the index of the subsequence which leads to the second best distance of the third query in the fourth time series.

Parameters

- `q`: Array whose first dimension is the length of the query time series and the second dimension is the number of queries.
- `t`: Array whose first dimension is the length of the time series and the second dimension is the number of time series.
- `n`: Number of matches to return.
- `distances`: Resulting distances.
- `indexes`: Resulting indexes.

void **mass**(**const** af::array &*q*, **const** af::array &*t*, af::array &*distances*)

Mueen's Algorithm for Similarity Search.

The result has the following structure:

- 1st dimension corresponds to the index of the subsequence in the time series.

- 2nd dimension corresponds to the number of queries.
- 3rd dimension corresponds to the number of time series.

For example, the distance in the position (1, 2, 3) correspond to the distance of the third query to the fourth time series for the second subsequence in the time series.

[1] Chin-Chia Michael Yeh, Yan Zhu, Liudmila Ulanova, Nurjahan Begum, Yifei Ding, Hoang Anh Dau, Diego Furtado Silva, Abdullah Mueen, Eamonn Keogh (2016). Matrix Profile I: All Pairs Similarity Joins for Time Series: A Unifying View that Includes Motifs, Discords and Shapelets. IEEE ICDM 2016.

Parameters

- `q`: Array whose first dimension is the length of the query time series and the second dimension is the number of queries.
- `t`: Array whose first dimension is the length of the time series and the second dimension is the number of time series.
- `distances`: Resulting distances.

void **`findBestNMotifs`**(**`const`** af::array *&profile*, **`const`** af::array *&index*, long *m*, long *n*, af::array *&motifs*, af::array *&motifsIndices*, af::array *&subsequenceIndices*, bool *selfJoin* = false)

This function extracts the best N motifs from a previously calculated matrix profile.

Parameters

- `profile`: The matrix profile containing the minimum distance of each subsequence.
- `index`: The matrix profile index containing where each minimum occurs.
- `m`: Subsequence length value used to calculate the input matrix profile.
- `n`: Number of motifs to extract.
- `motifs`: The distance of the best N motifs.
- `motifsIndices`: The indices of the best N motifs.
- `subsequenceIndices`: The indices of the query sequences that produced the minimum reported in the motifs output array.
- `selfJoin`: Indicates whether the input profile comes from a self join operation or not. It determines whether the mirror similar region is included in the output or not.

void **`findBestNDiscords`**(**`const`** af::array *&profile*, **`const`** af::array *&index*, long *m*, long *n*, af::array *&discords*, af::array *&discordsIndices*, af::array *&subsequenceIndices*, bool *selfJoin* = false)

This function extracts the best N discords from a previously calculated matrix profile.

Parameters

- `profile`: The matrix profile containing the minimum distance of each subsequence.
- `index`: The matrix profile index containing where each minimum occurs.
- `m`: Subsequence length value used to calculate the input matrix profile.
- `n`: Number of discords to extract.
- `discords`: The distance of the best N discords.
- `discordsIndices`: The indices of the best N discords.

- `subsequenceIndices`: The indices of the query sequences that produced the discords reported in the discords output array.
- `selfJoin`: Indicates whether the input profile comes from a self join operation or not. It determines whether the mirror similar region is included in the output or not.

void **`stomp`**(**`const`** af::array &*ta*, **`const`** af::array &*tb*, long *m*, af::array &*profile*, af::array &*index*)

STOMP algorithm to calculate the matrix profile between 'ta' and 'tb' using a subsequence length of 'm'.

[1] Yan Zhu, Zachary Zimmerman, Nader Shakibay Senobari, Chin-Chia Michael Yeh, Gareth Funning, Abdullah Mueen, Philip Brisk and Eamonn Keogh (2016). Matrix Profile II: Exploiting a Novel Algorithm and GPUs to break the one Hundred Million Barrier for Time Series Motifs and Joins. IEEE ICDM 2016.

Parameters

- `ta`: Query time series.
- `tb`: Reference time series.
- `m`: Subsequence length.
- `profile`: The matrix profile, which reflects the distance to the closer element of the subsequence from 'ta' in 'tb'.
- `index`: The matrix profile index, which points to where the aforementioned minimum is located.

void **`stomp`**(**`const`** af::array &*t*, long *m*, af::array &*profile*, af::array &*index*)

STOMP algorithm to calculate the matrix profile between 't' and itself using a subsequence length of 'm'. This method filters the trivial matches.

[1] Yan Zhu, Zachary Zimmerman, Nader Shakibay Senobari, Chin-Chia Michael Yeh, Gareth Funning, Abdullah Mueen, Philip Brisk and Eamonn Keogh (2016). Matrix Profile II: Exploiting a Novel Algorithm and GPUs to break the one Hundred Million Barrier for Time Series Motifs and Joins. IEEE ICDM 2016.

Parameters

- `t`: Query and reference time series.
- `m`: Subsequence length.
- `profile`: The matrix profile, which reflects the distance to the closer element of the subsequence from 't' in a different location of itself.
- `index`: The matrix profile index, which points to where the aforementioned minimum is located.

void **`matrixProfile`**(**`const`** af::array &*tss*, long *m*, af::array &*profile*, af::array &*index*)

Calculates the matrix profile between 't' and itself using a subsequence length of 'm'. This method filters the trivial matches.

[1] Yan Zhu, Zachary Zimmerman, Nader Shakibay Senobari, Chin-Chia Michael Yeh, Gareth Funning, Abdullah Mueen, Philip Brisk and Eamonn Keogh (2016). Matrix Profile II: Exploiting a Novel Algorithm and GPUs to break the one Hundred Million Barrier for Time Series Motifs and Joins. IEEE ICDM 2016.

Parameters

- `tss`: Query time series.
- `m`: Subsequence length.
- `profile`: The matrix profile, which reflects the distance to the closer element of the subsequence from 'ta' in 'tb'.

- `index`: The matrix profile index, which points to where the aforementioned minimum is located.

void **`matrixProfile`**(**`const`** af::array &*ta*, **`const`** af::array &*tb*, long *m*, af::array &*profile*, af::array &*index*)

Calculates the matrix profile between 'ta' and 'tb' using a subsequence length of 'm'.

[1] Yan Zhu, Zachary Zimmerman, Nader Shakibay Senobari, Chin-Chia Michael Yeh, Gareth Funning, Abdullah Mueen, Philip Brisk and Eamonn Keogh (2016). Matrix Profile II: Exploiting a Novel Algorithm and GPUs to break the one Hundred Million Barrier for Time Series Motifs and Joins. IEEE ICDM 2016.

Parameters

- `ta`: Query and reference time series.
- `tb`: Query and reference time series.
- `m`: Subsequence length.
- `profile`: The matrix profile, which reflects the distance to the closer element of the subsequence from 't' in a different location of itself.
- `index`: The matrix profile index, which points to where the aforementioned minimum is located.

void **`matrixProfileLR`**(**`const`** af::array &*tss*, long *m*, af::array &*profileLeft*, af::array &*indexLeft*, af::array &*profileRight*, af::array &*indexRight*)

Calculates the matrix profile to the left and to the right between 't' and using a subsequence length of 'm'.

[1] Yan Zhu, Makoto Imamura, Daniel Nikovski, and Eamonn Keogh. Matrix Profile VII: Time Series Chains: A New Primitive for Time Series Data Mining. IEEE ICDM 2017

Notice that when there is no match the subsequence index is the length of tss.

Parameters

- `tss`: Time series to compute the matrix profile.
- `m`: Subsequence length.
- `profileLeft`: The matrix profile distance to the left.
- `indexLeft`: The subsequence index of the matrix profile to the left.
- `profileRight`: The matrix profile distance to the right.
- `indexRight`: The subsequence index of the matrix profile to the right.

void **`getChains`**(**`const`** af::array &*tss*, long *m*, af::array &*chains*)

Calculates all the chains within 'tss' using a subsequence length of 'm'.

[1] Yan Zhu, Makoto Imamura, Daniel Nikovski, and Eamonn Keogh. Matrix Profile VII: Time Series Chains: A New Primitive for Time Series Data Mining. IEEE ICDM 2017

Notice that the size of the first dimension is the maximum possible size which is n - m + 1. If the number of values belonging to a chain is lower than the maximum, the remaining values and indexes are 0. It implies that 0 is an invalid chain index.

Parameters

- `tss`: Time series to compute the chains within them.
- `m`: Subsequence length.
- `chains`: The calculated chains with the following topology:
  - 1st dimension corresponds to the chains indexes flattened.

- 2nd dimension:
  - [0] corresponds to all the indexes in the chains flattened
  - [1] corresponds to the index of the chain that the value in [0] belongs to.
- 3rd dimension corresponds to the number of time series.

namespace internal

Typedefs

using khiva::matrix::internal::DistancesVector = typedef std::vector<double>

using khiva::matrix::internal::IndexesVector = typedef std::vector<unsigned int>

using khiva::matrix::internal::MatrixProfilePair = typedef std::pair<DistancesVecto

using khiva::matrix::internal::LeftRightProfilePair = typedef std::pair<MatrixProfi

using khiva::matrix::internal::Chain = typedef std::vector<unsigned int>

using khiva::matrix::internal::ChainVector = typedef std::vector<Chain>

Functions

af::array **slidingDotProduct**(**const** af::array &*q*, **const** af::array &*t*)

Calculates the sliding dot product of the time series 'q' against t.

Return array Returns an array with as many elements as 't' in the first dimension and as many elements as the last dimension of 'q' in the last dimension.

Parameters

- `q`: Array whose first dimension is the length of the query time series and the last dimension is the number of time series to calculate.
- `t`: Array with the second time series in the first dimension.

void **meanStdev**(**const** af::array &*t*, af::array &*a*, long *m*, af::array &*mean*, af::array &*stdev*)

Calculates the moving average and standard deviation of the time series 't'.

Parameters

- `t`: Input time series. Multiple time series.
- `a`: Auxiliary array to be used in the function calculateDistanceProfile. Use the overloaded method without this parameter.
- `m`: Window size.
- `mean`: Output array containing the moving average.
- `stdev`: Output array containing the moving standard deviation.

void **meanStdev**(**const** af::array &*t*, long *m*, af::array &*mean*, af::array &*stdev*)

Calculates the moving average and standard deviation of the time series 't'.

Parameters

- `t`: Input time series. Multiple time series.
- `m`: Window size.
- `mean`: Output array containing the moving average.
- `stdev`: Output array containing the moving standard deviation.

void **calculateDistances**(**const** af::array &*qt*, **const** af::array &*a*, **const** af::array &*sum_q*, **const** af::array &*sum_q2*, **const** af::array &*mean_t*, **const** af::array &*sigma_t*, **const** af::array &*mask*, af::array &*distances*)

Calculates the distance between 'q' and the time series 't', which produced the sliding. Multiple queries can be computed simultaneously in the last dimension of 'q'.

Parameters

- `qt`: The sliding dot product of 'q' and 't'.
- `a`: Auxiliary array computed using the meanStdev function. This array contains a precomputed fixed value to speed up the distance calculation.
- `sum_q`: Sum of the values contained in 'q'.
- `sum_q2`: Sum of squaring the values contained in 'q'.
- `mean_t`: Moving average of 't' using a window size equal to the number of elements in 'q'.
- `sigma_t`: Moving standard deviation of 't' using a window size equal to the number of elements in 'q'.
- `mask`: Mask band matrix to filter the trivial match of a subsequence with itself.
- `distances`: Resulting distances.

void **calculateDistances**(**const** af::array &*qt*, **const** af::array &*a*, **const** af::array &*sum_q*, **const** af::array &*sum_q2*, **const** af::array &*mean_t*, **const** af::array &*sigma_t*, af::array &*distances*)

Calculates the distance between 'q' and the time series 't', which produced the sliding. Multiple queries can be computed simultaneously in the last dimension of 'q'.

Parameters

- `qt`: The sliding dot product of 'q' and 't'.
- `a`: Auxiliary array computed using the meanStdev function. This array contains a precomputed fixed value to speed up the distance calculation.
- `sum_q`: Sum of the values contained in 'q'.
- `sum_q2`: Sum of squaring the values contained in 'q'.
- `mean_t`: Moving average of 't' using a window size equal to the number of elements in 'q'.
- `sigma_t`: Moving standard deviation of 't' using a window size equal to the number of elements in 'q'.
- `distances`: Resulting distances.

bool **tileIsFarFromDiagonal**(long *bandSize*, long *numRows*, long *row*, long *numColumns*, long *column*)

Given a tile indices and sizes it returns true when tile would not be affected by a identity band matrix.

Return If it is far or not.

Parameters

- `bandSize`: The band size.
- `numRows`: Number of rows of the tile.
- `row`: Starting row of the tile.
- `numColumns`: Number of columns of the tile.
- `column`: Starting column of the tile.

af::array **generateMask**(long *m*, long *numRows*, long *row*, long *numColumns*, long *column*, long *nTimeSeries* = 1)

Generate an identity band matrix for a given tile indices.

Return The mask.

Parameters

- `m`: The query size.
- `numRows`: Number of rows of the tile.
- `row`: Starting row of the tile.
- `numColumns`: Number of columns of the tile.

- `column`: Starting column of the tile.
- `nTimeSeries`: Number of time series.

void **massWithMask**(af::array *q*, **const** af::array &*t*, **const** af::array &*a*, **const** af::array &*mean_t*, **const** af::array &*sigma_t*, **const** af::array &*mask*, af::array &*distances*)

Calculates the Mueen distance.

[1] Chin-Chia Michael Yeh, Yan Zhu, Liudmila Ulanova, Nurjahan Begum, Yifei Ding, Hoang Anh Dau, Diego Furtado Silva, Abdullah Mueen, Eamonn Keogh (2016). Matrix Profile I: All Pairs Similarity Joins for Time Series: A Unifying View that Includes Motifs, Discords and Shapelets. IEEE ICDM 2016.

Parameters

- `q`: Array whose first dimension is the length of the query time series and the last dimension is the number of time series to calculate.
- `t`: Array with the second time series in the first dimension.
- `a`: Auxiliary array computed using the meanStdev function. This array contains a precomputed fixed value to speed up the distance calculation.
- `mean_t`: Moving average of 't' using a window size equal to the number of elements in 'q'.
- `sigma_t`: Moving standard deviation of 't' using a window size equal to the number of elements in 'q'.
- `mask`: Specifies the elements that should not be considered in the computation.
- `distances`: Resulting distances.

void **mass**(af::array *q*, **const** af::array &*t*, **const** af::array &*a*, **const** af::array &*mean_t*, **const** af::array &*sigma_t*, af::array &*distances*)

Mueen's Algorithm for Similarity Search.

Parameters

- `q`: Array whose first dimension is the length of the query time series and the last dimension is the number of time series to calculate.
- `t`: Array with the second time series in the first dimension.
- `a`: Auxiliary array computed using the meanStdev function. This array contains a precomputed fixed value to speed up the distance calculation.
- `mean_t`: Moving average of 't' using a window size equal to the number of elements in 'q'.
- `sigma_t`: Moving standard deviation of 't' using a window size equal to the number of elements in 'q'.
- `distances`: Resulting distances.

void **stomp_batched**(**const** af::array &*ta*, af::array *tb*, long *m*, long *batch_size*, af::array &*profile*, af::array &*index*)

void **stomp_batched_two_levels**(af::array *ta*, af::array *tb*, long *m*, long *batch_size_b*, long *batch_size_a*, af::array &*profile*, af::array &*index*)

void **stomp_parallel**(**const** af::array &*ta*, af::array *tb*, long *m*, af::array &*profile*, af::array &*index*)

void **stomp_batched_two_levels**(af::array *t*, long *m*, long *batch_size_b*, long *batch_size_a*, af::array &*profile*, af::array &*index*)

void **stomp_parallel**(af::array *t*, long *m*, af::array &*profile*, af::array &*index*)

void **findBestN**(**const** af::array &*profile*, **const** af::array &*index*, long *m*, long *n*, af::array &*distance*, af::array &*indices*, af::array &*subsequenceIndices*, bool *selfJoin*, bool *lookForMotifs*)

void **scamp**(af::array *tss*, long *m*, af::array &*profile*, af::array &*index*)

void **scamp**(af::array *ta*, af::array *tb*, long *m*, af::array &*profile*, af::array &*index*)

void **getChains**(af::array *tss*, long *m*, af::array &*chains*)

ChainVector **extractAllChains**(**const** IndexesVector &*profileLeft*, **const** IndexesVector &*profileRight*)

LeftRightProfilePair **scampLR**(std::vector<double> &&*ta*, long *m*)

void **scampLR**(af::array *tss*, long *m*, af::array &*profileLeft*, af::array &*indexLeft*, af::array &*profileRight*, af::array &*indexRight*)

2.9 Namespace Normalization

namespace normalization

Functions

af::array **decimalScalingNorm**(**const** af::array &*tss*)

Normalizes the given time series according to its maximum value and adjusts each value within the range (-1, 1).

Return af::array An array with the same dimensions as tss, whose values (time series in dimension 0) have been normalized by dividing each number by 10^j, where j is the number of integer digits of the max number in the time series.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

void **decimalScalingNormInPlace**(af::array &*tss*)

Same as decimalScalingNorm, but it performs the operation in place, without allocating further memory.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

af::array **maxMinNorm**(**const** af::array &*tss*, double *high* = 1.0, double *low* = 0.0, double *epsilon* = 0.00000001)

Normalizes the given time series according to its minimum and maximum value and adjusts each value within the range [low, high].

Return af::array An array with the same dimensions as tss, whose values (time series in dimension 0) have been normalized by maximum and minimum values, and scaled as per high and low parameters.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `high`: Maximum final value (Defaults to 1.0).
- `low`: Minimum final value (Defaults to 0.0).

- `epsilon`: Safeguard for constant (or near constant) time series as the operation implies a unit scale operation between min and max values in the tss.

void **maxMinNormInPlace**(af::array *&tss*, double *high* = 1.0, double *low* = 0.0, double *epsilon* = 0.00000001)

Same as maxMinNorm, but it performs the operation in place, without allocating further memory.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `high`: Maximum final value (Defaults to 1.0).
- `low`: Minimum final value (Defaults to 0.0).
- `epsilon`: Safeguard for constant (or near constant) time series as the operation implies a unit scale operation between min and max values in the tss.

af::array **meanNorm**(**const** af::array *&tss*)

Normalizes the given time series according to its maximum-minimum value and its mean. It follows the following formulae:

$$\acute{x} = \frac{x - mean(x)}{max(x) - min(x)}.$$

Return af::array An array with the same dimensions as tss, whose values (time series in dimension 0) have been normalized by substracting the mean from each number and dividing each number by $max(x) - min(x)$, in the time series.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

void **meanNormInPlace**(af::array *&tss*)

Normalizes the given time series according to its maximum-minimum value and its mean. It follows the following formulae:

$$\acute{x} = \frac{x - mean(x)}{max(x) - min(x)}.$$

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

af::array **znorm**(**const** af::array *&tss*, double *epsilon* = 0.00000001)

Calculates a new set of timeseries with zero mean and standard deviation one.

Return af::array With the same dimensions as tss where the time series have been adjusted for zero mean and one as standard deviation.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `epsilon`: Minimum standard deviation to consider. It acts as a gatekeeper for those time series that may be constant or near constant.

void **`znormInPlace`**(af::array &*tss*, double *epsilon* = 0.00000001)
Adjusts the time series in the given input and performs z-norm inplace (without allocating further memory).

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `epsilon`: Minimum standard deviation to consider. It acts as a gatekeeper for those time series that may be constant or near constant.

2.10 Namespace Polynomial

`namespace polynomial`

Functions

af::array **`polyfit`**(**`const`** af::array &*x*, **`const`** af::array &*y*, int *deg*)
Least squares polynomial fit. Fit a polynomial $p(x) = p[0] * x^{deg} + ... + p[deg]$ of degree deg to points (x, y). Returns a vector of coefficients p that minimizes the squared error.

Return af::array Polynomial coefficients, highest power first.

Parameters

- `x`: x-coordinates of the M sample points $(x[i], y[i])$.
- `y`: y-coordinates of the sample points.
- `deg`: Degree of the fitting polynomial.

af::array **`roots`**(**`const`** af::array &*pp*)
Calculates the roots of a polynomial with coefficients given in p. The values in the rank-1 array p are coefficients of a polynomial. If the length of p is $n + 1$ then the polynomial is described by:

$$p[0] * x^n + p[1] * x^{n-1} + ... + p[n-1] * x + p[n]$$

.

Return af::array Containing the roots of the polynomial.

Parameters

- `pp`: Array of polynomial coefficients.

2.11 Namespace Regression

`namespace regression`

Functions

void **linear**(**const** af::array &*xss*, **const** af::array &*yss*, af::array &*slope*, af::array &*intercept*, af::array &*rvalue*, af::array &*pvalue*, af::array &*stderrest*)

Calculate a linear least-squares regression for two sets of measurements. Both arrays should have the same length.

Parameters

- `xss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `yss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `slope`: Slope of the regression line.
- `intercept`: Intercept of the regression line.
- `rvalue`: Correlation coefficient.
- `pvalue`: Two-sided p-value for a hypothesis test whose null hypothesis is that the slope is zero, using Wald Test with t-distribution of the test statistic.
- `stderrest`: Standard error of the estimated gradient.

2.12 Namespace Regularization

namespace regularization

Typedefs

using khiva::regularization::AggregationFuncDimT = typedef af::array (*)(const af::arr

using khiva::regularization::AggregationFuncBoolDimT = typedef af::array (*)(const af:

using khiva::regularization::AggregationFuncInt = typedef af::array (*)(const af::array

Functions

af::array **groupBy**(**const** af::array &*in*, AggregationFuncBoolDimT *aggregationFunction*, int *nColumnsKey* = 1, int *nColumnsValue* = 1)

Group by operation in the input array using nColumnsKey columns as group keys and nColumnsValue columns as values. The data is expected to be sorted. The aggregation function determines the operation to aggregate the values.

Return af::array Array with the values of the group keys aggregated using the aggregationFunction.

Parameters

- `in`: Input array containing the keys and values to operate with.
- `aggregationFunction`: This param determines the operation aggregating the values.
- `nColumnsKey`: Number of columns conforming the key.
- `nColumnsValue`: Number of columns conforming the value (they are expected to be consecutive to the column keys).

af::array **groupBy**(**const** af::array &*in*, AggregationFuncInt *aggregationFunction*, int *nColumnsKey* = 1, int *nColumnsValue* = 1)

Group by operation in the input array using nColumnsKey columns as group keys and nColumnsValue columns as values. The data is expected to be sorted. The aggregation function determines the operation to aggregate the values.

Return af::array Array with the values of the group keys aggregated using the aggregationFunction.

Parameters

- `in`: Input array containing the keys and values to operate with.
- `aggregationFunction`: This param determines the operation aggregating the values.
- `nColumnsKey`: Number of columns conforming the key.
- `nColumnsValue`: Number of columns conforming the value (they are expected to be consecutive to the column keys).

af::array **groupBy**(**const** af::array &*in*, AggregationFuncDimT *aggregationFunction*, int *nColumnsKey* = 1, int *nColumnsValue* = 1)

Group by operation in the input array using nColumnsKey columns as group keys and nColumnsValue columns as values. The data is expected to be sorted. The aggregation function determines the operation to aggregate the values.

Return af::array Array with the values of the group keys aggregated using the aggregationFunction.

Parameters

- `in`: Input array containing the keys and values to operate with.
- `aggregationFunction`: This param determines the operation aggregating the values.
- `nColumnsKey`: Number of columns conforming the key.
- `nColumnsValue`: Number of columns conforming the value (they are expected to be consecutive to the column keys).

2.13 Namespace Statistics

namespace statistics

Functions

af::array **covariance**(**const** af::array &*tss*, bool *unbiased* = true)

Returns the covariance matrix of the time series contained in tss.

Return af::array The covariance matrix of the time series.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `unbiased`: Determines whether it divides by n - 1 (if false) or n (if true).

af::array **kurtosis**(**const** af::array &*tss*)

Returns the kurtosis of tss (calculated with the adjusted Fisher-Pearson standardized moment coefficient G2).

Return af::array The kurtosis of tss.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

af::array **moment**(**const** af::array &*tss*, int *k*)

Returns the kth moment of the given time series.

Return af::array The kth moment of the given time series.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `k`: The specific moment to be calculated.

af::array **ljungBox**(**const** af::array &*tss*, long *lags*)

The Ljung–Box test checks that data within the time series are independently distributed (i.e. the correlations in the population from which the sample is taken are 0, so that any observed correlations in the data result from randomness of the sampling process). Data are no independently distributed, if they exhibit serial correlation.

The test statistic is:

$$Q = n\,(n+2)\sum_{k=1}^{h}\frac{\hat{\rho}_k^2}{n-k}$$

where ''n" is the sample size, $\hat{\rho}k$ is the sample autocorrelation at lag ''k'', and ''h" is the number of lags being tested. Under H_0 the statistic Q follows a $\chi^2(h)$. For significance level α, the $criticalregion$ for rejection of the hypothesis of randomness is:

$$Q > \chi^2_{1-\alpha,h}$$

where $\chi^2_{1-\alpha,h}$ is the α-quantile of the chi-squared distribution with ''h" degrees of freedom.

[1] G. M. Ljung G. E. P. Box (1978). On a measure of lack of fit in time series models. Biometrika, Volume 65, Issue 2, 1 August 1978, Pages 297–303.

Return af::array Ljung-Box statistic test.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.
- `lags`: Number of lags being tested.

af::array **quantile**(**const** af::array &*tss*, **const** af::array &*q*, float *precision* = 100000000)
Returns values at the given quantile.

Return af::array Values at the given quantile.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series. NOTE: the time series should be sorted.
- `q`: Percentile(s) at which to extract score(s). One or many.
- `precision`: Number of decimals expected.

af::array **quantilesCut**(**const** af::array &*tss*, float *quantiles*, float *precision* = 0.00000001)
Discretizes the time series into equal-sized buckets based on sample quantiles.

Return af::array Matrix with the categories, one category per row, the start of the category in the first column and the end in the second category.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series. NOTE: the time series should be sorted.
- `quantiles`: Number of quantiles to extract. From 0 to 1, step 1/quantiles.
- `precision`: Number of decimals expected.

af::array **sampleStdev**(**const** af::array &*tss*)
Estimates standard deviation based on a sample. The standard deviation is calculated using the "n-1" method.

Return af::array The sample standard deviation.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

af::array **skewness**(**const** af::array &*tss*)
Calculates the sample skewness of tss (calculated with the adjusted Fisher-Pearson standardized moment coefficient G1).

Return af::array Array containing the skewness of each time series in tss.

Parameters

- `tss`: Expects an input array whose dimension zero is the length of the time series (all the same) and dimension one indicates the number of time series.

- *Namespace LinAlg*
- *Namespace Matrix*
- *Namespace Normalization*
- *Namespace Polynomial*
- *Namespace Regression*
- *Namespace Regularization*
- *Namespace Statistics*

CHAPTER 3

Bindings

We have developed bindings to enable the execution of Khiva from the following languages. In order to make it work, you should first install Khiva library in your machine, explained in :ref: *chapter-gettingstarted*.

3.1 Python

In order to install the khiva-python binding of the library, you would need to fetch the latest version of the code from:

```
git clone https://github.com/shapelets/khiva-python.git
```

After cloning the repository, you can install khiva-python by executing the next commands:

```
cd /path_to_khiva-python
python3 setup.py install
```

If the installation is successful, you are ready to start playing with the library.

3.2 Java

In order to install the khiva-java binding of the library, you would need to fetch the latest version of the code from:

```
git clone https://github.com/shapelets/khiva-java.git
```

Once you have downloaded the code, you have to move to the source code directory and execute the following commands:

```
cd path_to_java_khiva_dir
mvn install
mvn javadoc:javadoc
```

If all steps finished as expected, you should be able to use the Khiva from your java projects.

3.3 R

In order to install the khiva-r binding of the library, you would need to fetch the latest version of the code from:

```
git clone https://github.com/shapelets/khiva-r.git
```

After downloading the code, you would need to open an R console and execute the following commands, to set the work directory and install the Khiva binding:

```
setwd(<project-root-dir>/)
devtools::install()
```

Once the installation of the binding has been carried out, you can make the library available by executing:

```
library(khiva)
```

If all previous steps were successful you will ready to start working with the library.

3.4 MATLAB

In order to install the khiva-matlab binding of the library, you would need to fetch the latest version of the code from:

```
git clone https://github.com/shapelets/khiva-matlab.git
```

Once the code is available, we just have to add the path to the khiva-matlab/+khiva folder to the MATLAB path. Thus, the user will be able to import and call our library.

CHAPTER 4

Building programs using Khiva with CMake

In order to build a program using the Khiva library with the CMake build system you need just a couple of lines in your `CmakeLists.txt`:

```
cmake_minimum_required(VERSION 3.1)
project(example)

find_package(Khiva REQUIRED)

add_executable(example example.cpp)
target_link_libraries(example Khiva::khiva)
```

`find_package(Khiva REQUIRED)` may be used when Khiva was installed system wide. Please follow the installation instructions for your operating system contained at the *Getting Started*.

CHAPTER 5

AUTHORS

5.1 Core Development Team

- Justo Ruiz-Ferrer (justo.ruiz@shapelets.io)
- Antonio Vilches (antonio.vilches@shapelets.io)
- Oscar Torreno (oscar.torreno@shapelets.io)
- David Cuesta (david.cuesta@shapelets.io)

5.2 Contributions

- Luis Sanchez (luis.sanchez@shapelets.io)

CHAPTER 6

Cite Us

If you use Khiva Library for a publication, please cite it as:

```
@misc{khiva-library,
   author = "David Cuesta and Justo Ruiz and Oscar Torreno and Antonio Vilches",
   title = "Khiva Library",
   howpublished = "\url{https://shapelets.io/khiva}"
}
```

Index

K