Free University of Bozen
Faculty of Computer Science
Degree Course in Computer Science

Interactive exploration of time series through motif discovery

Author:
FRANCESCO PICCOLI

Supervisor:
DR. MATTEO CECCARELLO

Co-supervisor:
PROF. DR. JOHANN GAMPER

A.Y. 2020/2021

Acknowledgments

I would like to express my deepest gratitude to my supervisor Dr. Matteo Ceccarello for his expertise, for his guidance, patience and for his support throughout this project.
I would also like to thank my co-supervisor Dr. Johann Gamper for introducing me to the PREMISE project and for giving me the chance to take part in it. In addition, I would like to thank all the researchers and all the people involved in the PREMISE project, for all the work they have done and are still doing.

I am grateful for my parents for their love and support that keep me confident and motivated. I thank them for always believing in me and encouraging me.
Thank you to my brother for always being a model I could look up to and ask for advice.

Very special thanks to my friends Gian Marco, Francesco, Tomaso, Federico and Pietro and to all the friends I met in Bolzano, in particular Martin, Luigi, Agata, Margherita, Matteo, Jakob and Raffaele, for the wonderful moments we shared.

Abstract

Visualizing time series is the first and foremost step to proceed with their analysis. Additionally, the visualization of the motifs and the patterns of a time series is extremely important to detect possible problems, to generate hypotheses on the behavior of the time series, and more in general to help analysts understand the significance of data and get new insights.

Time series visualization is even more effective when the user can interact with the data being displayed, to directly manipulate it and explore it.

This thesis presents an interactive visualization interface to visualize time series, interact with them, find recurring patterns, also known as motifs, using different motif discovery algorithms and get information on the motifs found.

The thesis is part of the PREMISE project, which sees the researchers of the Database Systems Group of the Free University of Bozen involved in the study and the creation of algorithms capable of forecasting maintenance needs to make production planning easier for two south Tyrolean companies: Durst and Technoalpin. Given the large amounts of sensor data that these companies collect, an effective way of efficiently exploring such data is of paramount importance.

Given the vast and diverse array of motif discovery approaches, where each approach has different requirements and might yield different results, the web application was developed following a plugin architecture, so that one can add a new motif discovery algorithm by just adding a plugin. This makes the tool easy to extend to new algorithms.

Using this plugin infrastructure, the tool already provides 3 different algorithms to find motifs, in order to allow a deeper exploration of the time series and to get possible different results.

To guide the user in the choice between the different algorithms, this thesis also includes an experimental evaluation of the runtime performance and solution quality of the different algorithms.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

The underlying theme of this study is the analysis of time series for predictive maintenance. A time series can be defined as a collection of values obtained from measurements over time. Predictive maintenance is an approach that promises cost savings and seamless production through the early recognition of faults based on time series data.

In particular, this work will focus on how the interactive visualization of the motifs could help in predicting possible failures in industrial machines. A motif represents repeated subsequences or patterns in a time series, obtained through the application of motif discovery data-mining techniques and algorithms.

This work considers the use cases of two south Tyrolean companies: Durst, a manufacturer of large-scale printing equipment, and Technoalpin, which produces snow-making systems. These two companies collect time series data (i.e temperature, pressure, etc.) from their industrial machines through sensors.

The thesis aims at creating an interactive visualization interface that allows the user to choose and set a time series, the desired time span, and the motif discovery algorithm along with a set of parameters, and visualize the time series' motifs as a starting point to conduct further analysis.

Time series visualization is the first crucial step for their analysis. It simplifies the task of detecting trends, outliers, cycles, and seasonality that can influence and condition the subsequent analysis steps. Additionally, the visualization of motifs allows to detect particular behaviors and trends that provide insights, might alert to possible problems, and help us in generating hypotheses.

This work is part of the PREMISE (Predictive maintenance for industrial equipment) project, a two-year research project funded by the European Regional Development Fund (ERDF 2014-2020) and the Autonomous Province of Bozen. The project sees the researchers of the Database Systems Group of the Faculty of the Computer Science of the Free University of Bozen involved in the study and the creation of algorithms capable of forecasting maintenance needs in order to make production planning easier for the aforementioned companies.

1.2 Organisation

Chapter 2 presents background concepts and definitions along with the current state of the art in motif discovery. Chapter 3 illustrates and explains the functioning of the motif discovery algorithms that were implemented in the web application developed. Chapter 4 covers the most important aspects behind the implementation of the program. In Chapter 5, a discussion on the obtained experimental results is carried out, and finally, Chapter 6 closes the work with some considerations and ideas for possible future developments.

Chapter 2

Background

This chapter provides a general overview of several background concepts for the research conducted in this thesis. An introduction to time series is reported in Section 2.1.

Sections 2.2 and 2.3 will discuss two problems related to time series: similarity and distance measures, and the high-dimensionality of data respectively.

Section 2.4 will review the chief concepts behind motif discovery, Section 2.5 will go through the notion of matrix profile, and finally, Section 2.6 will discuss the state-of-the-art of the motif discovery literature.

2.1 Time series

A time series represents a collection of values obtained from sequential measurements over time [4]. Today's time series data are being generated at an extremely fast rate in many different application domains, e.g. finance, science, medicine, and industrial production [3]. For instance, Technoalpin is currently collecting about 125 million sensor measurements per day regarding its industrial machines. Durst instead, is adding about 170 thousand sensor measurements per day just for its Tau printer model.

We can define a time series as follows [16]:

Definition 2.1 (Time Series)**.** A time-series T is an ordered sequence of n real-valued variables, where the ordering is typically temporal: $T = (t_1, ..., t_n), t_i \in \mathbb{R}$

The total number of data points n of a time series, is defined as the length of the time series.

A time series is usually the outcome of the observation of a process in which values are obtained from measurements taken at equally spaced points in time according to a certain sampling frequency [4].

A time series can be univariate if it has a unique dimension and one single variable is varying over time, or multivariate when it has multiple dimensions and several variables vary over the same time range [4].

Time series are inherently related to large data size, analysing them often requires subsequences of the time series to be translated into vectors in a high dimensional space through the use of long sliding windows. This often makes their analysis non-trivial, in addition, the continuous updates in the time series require the algorithms to react to updates in real-time and therefore to be very fast [5].

As in the case of motif discovery (see Section 2.4), the main interest in time series lies typically in the local regions, known as subsequences, and not in the global properties of the time series [10], a subsequence can be defined as follows [4]:

Definition 2.2 (Subsequence)**.** Given a time series T = $(t_1, ..., t_n)$, of length n, a subsequence S of T is a series of length $m \leq n$ consisting of contiguous time instants from T.

$$S = (t_k, t_{k+1}, ..., t_{k+m-1})$$

with $1 \leq k \leq n - m + 1$.

An example of a subsequence is depicted in Figure 2.4.

2.2 Similarity measures

A variety of data mining tasks in time series rely on a concept of similarity between series or subsequences of series [4, 5].

Definition 2.3 (Similarity measure)**.** The similarity measure D(T,U) between time series T and U is a function taking two time series as inputs and returning the distance d between these series. [4]

Simply put, distance measures quantify how distant a pair of time series are, by providing a numerical value [17].

Unlike for other data types (e.g. ordinal variables), devising an appropriate distance measure between time series is by no means a trivial task [3, 5]. As pointed out by Paparizzos et al. (2020) [17], in time series, similarity measures consider whole sequences of observations together, differently from other data types where similarity measures consider independent observations.

The human eye can effortlessly recognize perceptually similar time series, however, deriving definitions to reflect mathematically non-identical time series can be challenging [17].

The main challenges we face when we try to find patterns or more in general similar time series, are related to:

- The possibles shifts: when time series are similar but have different heights.
- The scaling problem: when two time series have a similar pattern but they have different amplitudes.
- The warping problem: when two time series present similar patterns but one is more "stretched" or "compressed" compared to the other.

The more the distance measure is able to handle these 3 problems, the higher its robustness.

For this reason, many different similarity measures have been proposed through the years [17], e.g. Euclidean distance (ED), Z-Normalized Euclidean distance (Z-ED), Dynamic Time Warping (DTW), distance based on Longest Common Subsequence (LCSS), Spatial Assembling Distance (SpADe), etc. [3]

For the scope of this thesis, we will focus on two families of distance measures: lock-step measures (e.g. ED in Subsection 2.2.1 and Z-ED in Subsection 2.2.2), where given two time series T_1 and T_2, to compute the distance we compare the i-th point of one time series to the i-th point of the other [3]; and elastic measure (e.g. DTW in Subsection 2.2.3) which creates a non-linear mapping between time series through a one-to-many comparison between points, to align or stretch points [17].

2.2.1 Euclidean Distance

The most used similarity measure is Euclidean distance (ED) [4, 5, 18].

Even though it does not deal with any of the 3 aforementioned challenges, its ease of use and its efficiency (runs in linear time) makes it very popular. In addition, it is parameter-free and it allows indexing [3, 17].

Despite being very sensitive to noise, shifts, and misalignment in time, ED has been shown to be incredibly competitive with other more complex approaches, especially for large datasets [3].

Moreover, Mueen (2014) [13] argues that distance measures are less important for motif discovery since time series that are similar under one similarity measure are also similar under other measures. Consequently, one should opt for the easiest and most effective one, the Euclidean distance.

However, there are conflicting opinions: as Fu (2011) [5] mentions, ED is not always an appropriate similarity function: in some specific domains such as stock time series or time series regarding scientific data such as electrocardiograms, it does not provide an accurate distant measure.

Furthermore, elastic measures such as DTW (see Subsection 2.2.3), are remarkably more accurate than ED when considering small datasets [3]. In fact, as mentioned at the beginning of this subsection, ED does not fulfill any of the robustness attributes: to solve scaling and noise issue we can add some preprocessing steps, but to address warping and outliers we need more sophisticated techniques, which make ED no longer fast and straightforward to implement [4].

2.2.2 Z-normalized Euclidean Distance

Another interesting approach to compute distance measures between time series, is the z-normalized Euclidean distance (Z-ED), widely used in the motif discovery literature.

It is still the Euclidean distance that we are considering, but Z-score normalization is applied as follows [17]:

$$\vec{x'} = \frac{\vec{x} - average(\vec{x})}{std(\vec{x})}$$

We are basically shifting each subsequence by its mean and diving it by its standard deviation. The resulting z-normalized subsequence will have mean equal to 0 and variance equal to 1 [14, 17].

To compute the z-normalized Euclidean distance between two time series X and Y of same length m, we will use the following formula:

$$\sqrt{\sum_{i=1}^{m}(x_i - y_i)^2}$$

where $x_i = \frac{1}{\sigma_x}(X_i - \mu_x)$, $y_i = \frac{1}{\sigma_y}(Y_i - \mu_y)$ and i and m are respectively the start and ending point of the time series subsequence [14].

Additionally, starting from the z-normalized Euclidean distance between two subsequences of length m, it is possible to easily compute the Pearson correlation coefficient between them by applying the following formula [15]:

$$\rho(x, y) = 1 - \frac{1}{2w} d_{znorm}^2(x, y)$$

where d_{znorm} is the z-normalized Euclidean distance between the subsequences and w is the length of x and y (i.e. in this case the length of the subsequences). This formula is noteworthy because it means that all algorithms that look for motifs that minimize the z-normalized Euclidean distance are looking at motifs maximizing the correlation. The correlation is a much more intuitive measure to interpret since it has a predefined range (from -1 to 1), instead, the Euclidean distance is more difficult to interpret.

By z-normalizing a time series, we achieve scale and offset invariance, when in a later stage we compute distance [7, 14]. This is a crucial step, in fact, without this normalization process every small difference in amplitude and offset risks to compromise the value of the distance measure and consequently, the finding of motifs and patterns [14]: when analyzing motifs, we are more interested in finding "shapes" and not exact offset and amplitude matches between subsequences [1, 7].

The downside of this similarity measure lies in the fact that it does not deal with warping and the fact that it is tied to a given window length, namely a subsequence length that must be set as a parameter.

2.2.3 Dynamic Time Warping

Dynamic time warping (DTW) is the only one of the three considered distance measures that deals with the warping problem.

This elastic distance measure allows us to handle local distortions of the time axis: by permitting the warping of the time axis we are able to find matches and patterns between time series subsequences even when one is more "stretched" or "compressed" compared to the other [3, 4].

The alignment of two time series, P and Q, using DTW, is done by first constructing an n-by-m matrix M. The (ith, jth) element of the matrix, m_{ij}, contains the distance $d(q_i, p_j)$ between the two points q_i and p_j (an Euclidean distance is typically used). It corresponds to the alignment between the points q_i and p_j. A warping path, W, is a contiguous set of matrix elements that defines a mapping between Q and P. Its kth element is defined as $w_k = (i_k, j_k)$ and $W = w_1, w_2, ..., w_k, ..., w_K$ where $max(m, n) \leq K < m + n - 1$ [5].

DTW computation has a time complexity $O(n^2)$ [4], therefore, its computation is dramatically more expensive than the ones of ED and Z-ED which have approximately a linear time complexity $O(n)$.

2.3 Dimensionality reduction and time series representation

As mentioned in Section 2.1, time series are inherently high dimensional data, processing such data in its raw format is very expensive from a computational point of view [3]. For such reasons, it is preferable to utilize representation techniques to reduce the dimensionality (namely the length) of the time series, while still retaining its essential features [3].

In fact, a representation technique aims at simplifying a time series by reducing its length and at the same time retaining the time series distinctive characteristics [4].

Several dimensionality-reduction techniques have been devised, for instance, Single Value Decomposition (SVD), Discrete Fourier Transformation (DFT), Piecewise Aggregate Approximation (PAA), Chebyshev polynomials (CHEB), Symbolic Aggregate approXimation (SAX), etc. [3]

For the analysis conducted in this work, the prevailing representation techniques used were PAA and SAX which will be briefly presented in the following subsections.

2.3.1 Piecewise Aggregate Approximation

A time series C of length n can be represented in a w-dimensional space by a vector $\bar{C} = \bar{c_1}, ...\bar{c_w}$ [1, 18]. The i^{th} element of $\bar{C}$ is calculated using the following equation:

$$\bar{C_i} = \frac{w}{n} \sum_{j=\frac{n}{w}(i-1)+1}^{\frac{n}{w}i} c_j$$

In simple terms, the reduction of the time series from n dimensions to w is done by dividing the data into w equal-sized "frames", the mean of the data within a frame is computed and a vector of these values becomes the dimensionality-reduced representation [1, 18]. The original subsequence can therefore be approximately represented using the PAA coefficients [6].

This simple and intuitive representation, known as Piecewise Aggregate Approximation (PAA) has been proven to compete with more complex dimensionality reduction techniques such as Fourier transforms and wavelets [1, 18].

Moreover, PAA has some key advantages over its competitors: it is significantly faster to calculate, and it can support several distance functions including for example dynamic time warping (DTW) (see Section 2.2.3) [18].

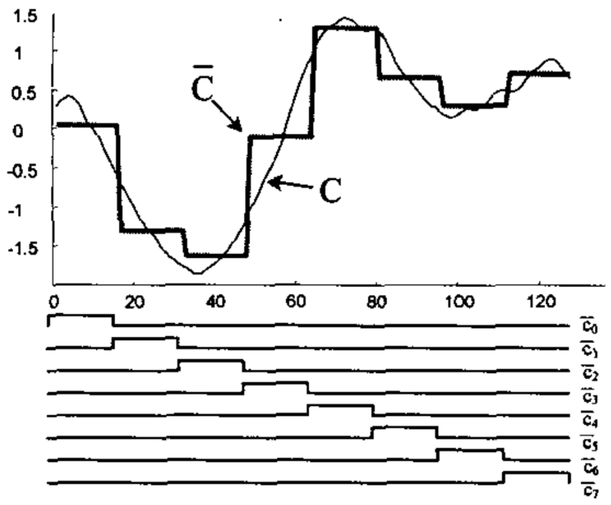


Figure 2.1: The PAA representation can be easily seen as a way of using a linear combination of box basis functions to model a sequence. In this figure, a 128-point sequence has been brought down to 8 dimensions. Figure taken from [18].

2.3.2 Symbolic Aggregate approXimation

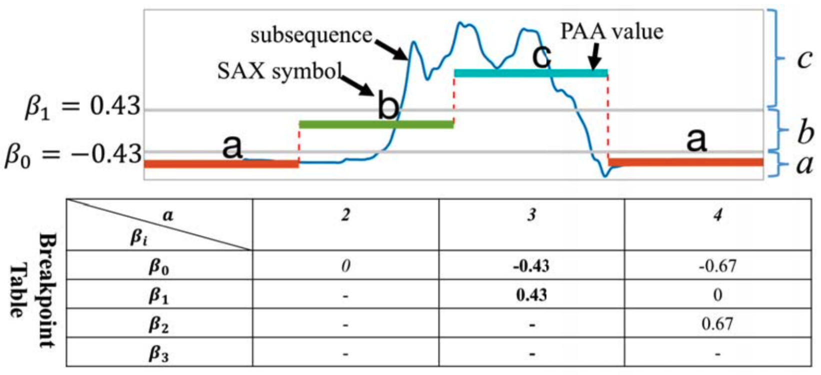
Symbolic Aggregate approXimation (SAX), is a widespread technique utilized to discretize univariate time series [6].

SAX representation discretizes both time and y-values and represents the time series with words made of alphabet symbols. Two parameters condition the final quality of the discretization process, namely the word size w and the alphabet size a [14].

Given a z-normalized subsequence from a time series and a reduced dimension size w, SAX first converts the subsequence into its lower-dimensional PAA representation (see Section 2.3.1) of size w [6, 7]. Then, the resulting PAA coefficient vector is mapped to w symbols with alphabet size a according to a breakpoints table, defined so that the areas are more or less equal-probable under Gaussian distribution [6, 7]. In fact, since normalized subsequences have a highly Gaussian distribution, we can determine "breakpoints" that yield equal-sized areas under Gaussian curve [1].

Definition 2.4 (Breakpoints)**.** Breakpoints are a sorted list of numbers $B = \beta_1, ..., \beta_{a-1}$ such that the area under a $N(0, 1)$ Gaussian curve from β_i to $\beta_{i+1} = \frac{1}{a}$ (β_0 and β_α are defined as $-\infty$ and ∞, respectively) [1].

These breakpoints can be defined by looking them up in a statistical table [1]. Figure 2.2 sums up the whole process. The bold colored flat lines depict the PAA coefficients values calculated from their respective segments in the subsequence; the figure represents also the breakpoint table with a from 2 to 4. Since we set the alphabet size a to 3, the second column indicates the breakpoints, based on which three regions are generated [6]: $[-\infty, -0.43), [-0.43, 0.43), [0.43, \infty]$. All PAA coefficients below the smallest breakpoint are therefore mapped to the symbol *"a"*, all coefficients greater than or equal to the smallest breakpoint and less than the second smallest breakpoint are mapped to the symbol *"b"* and so on [18]. In Figure 2.2, the PAA coefficients within these three regions are depicted by symbols a, b and c respectively. In the example shown, the SAX word becomes *abca* [6].



a / β_i	2	3	4
β_0	0	-0.43	-0.67
β_1	-	0.43	0
β_2	-	-	0.67
β_3	-	-	-

Figure 2.2: Example of the generation process of a SAX word *"abca"* where $a = 3$ and $w = 4$. Figure taken from [6].

2.4 Motif discovery

After a brief introduction on time series, dimensionality reduction techniques, and similarity measures, we will now focus on the motif discovery data mining task, which stands at the very core of this thesis.

Motif discovery is a useful exploratory tool to generate hypotheses [1], as in our case, where motif discovery is used as a starting point for the analysis of data for predictive maintenance.

In addition to generating hypotheses, motif discovery has numerous applications in several data mining fields, including for instance rule discovery, classification and clustering [10].

Time series motifs are repeated segments (namely, subsequences) within the time series that convey insightful information about the underlying system; in particular, these repeated segments are interesting when they are unexpected [13].

Motif discovery yields more meaningful results when the size of data is big, but discovering repetitions in long time series can be considered a nontrivial task. Often some preprocessing steps on the time series are needed (e.g. interpolation, data alignment, transformation...) [13].

There are two possible ways to define motifs: one is based on the similarity between motifs, the other on the frequencies of such motifs [13]. We can consider motifs interesting either when two motifs are too similar to each other to happen randomly, or instead based on how frequently they occur [13].

To understand the differences between the two possible definitions, we can represent each segment of a given length as a high dimensional point, the closer two points in the space, the more similar they are. We define also a threshold R which specifies the minimum allowable similarity or the maximum allowable distance between two occurrences of a motif [13]. A two-dimensional simple dataset is represented in Figure 2.3.

Definition 2.5 (Similarity-based motifs)**.** Given a time series and a length, time series motifs are the repeated segments in order of their similarities among the repeated occurrences within an R-ball [13].

Definition 2.6 (Support-based motifs)**.** Given a time series and a length, time series motifs are the segments that have the most number of repetitions within an R-ball [13].

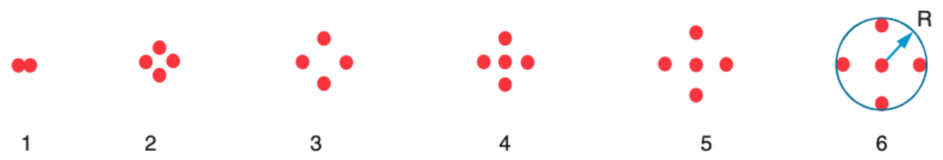


Figure 2.3: A 2D small dataset with clusters of points that could represent motifs [13]. The similarity-based definition would produce motifs in the order: 1-2-3-4-5-6. Instead, by applying the support-based definition, motifs would be produced in the order: 6-5-4-3-2-1. Figure taken from [13].

In the above definitions the length parameter depends generally on the domain, a significant knowledge of the domain is thus required to properly set it.

It is obviously possible to discover motifs at different resolutions by searching for different lengths. In this case, we talk about variable-length motif discovery [10]:

Definition 2.7 (Variable-Length Motif Pair Discovery)**.** Given a data series T and a subsequence length-range $[l_{min}, ..., l_{max}]$, we want to find the data series motif pairs of all lengths in $[l_{min}, ..., l_{max}]$, occurring in T.

However, such an exhaustive search has a massive computational cost and requires new techniques such as the utilization of a new lower bounding approach, which can efficiently search a large space of possible solutions [10].

For the support-based definition of motif, the threshold parameter R is as necessary as the motif length, and it is also domain-dependent [13].

This threshold parameter R, allows us to tell when two subsequences are similar. This idea of similarity between subsequences is formalized in the concept of match (see Figure 2.4) [1]:

Definition 2.8 (Match)**.** Given a positive real number R, called range, and a time series T containing a subsequence C beginning at position p and a subsequence M beginning at q, if $D(C, M) \leq R$, then M is called a matching subsequence of C.

Additionally, we should also define the concept of trivial match: the best matches to a subsequence (apart obviously from itself) are generally situated a few points to its left and its right [1]. We want to exclude such matches since they do not carry any useful information [1, 18]. Figure 2.5 shows the situation.

Definition 2.9 (Trivial Match)**.** Given a time series T, containing a subsequence C beginning at position p and a matching subsequence M beginning at q, we say that M is a trivial match to C if either p=q or there does not exist a subsequence M' beginning at q' such that $D(C, M') > R$, and either $q < q' < p$ or $p < q' < q$ [1].

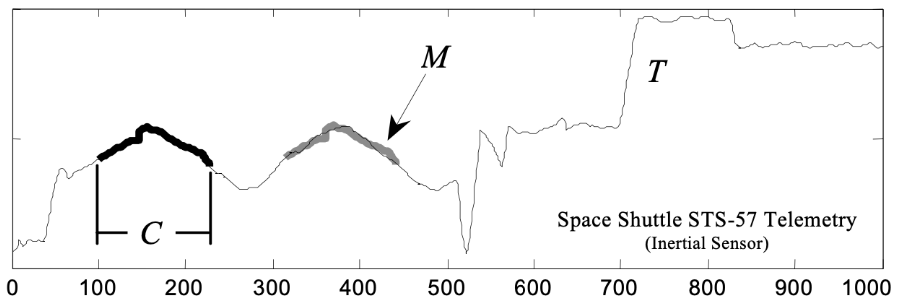


Figure 2.4: A visual example of a time series T (thin line), a subsequence C (thick black line) and a subsequence M (thick gray line) that is a match to C. Figure taken from [1].

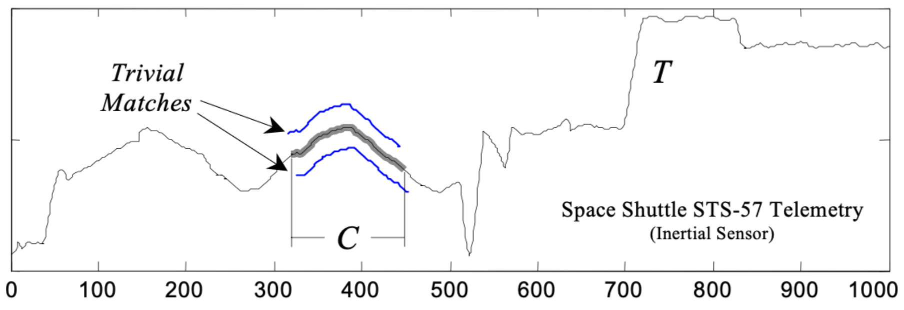


Figure 2.5: The best matches are almost always the trivial subsequences right to the left and right of the subsequence (in this case C). Figure taken from [1].

2.5 Matrix profile

One of the most used approaches for the localization of exact motifs is the matrix profile, a vector that is computed by finding the minimum value of every distance profile in the time series [2, 10].

The distance profile is in turn a vector storing all the distances from one subsequence to all the other subsequences [21], more formally:

Definition 2.10 (Distance profile)**.** A distance profile $D \in \mathbb{R}^{(n-l+1)}$ of a data series T regarding subsequence $T_{i,l}$ is a vector that stores $dist(T_{i,l}, T_{j,l})\ \forall\ j \in [1, 2, ..., n-l+1]$ where $i \neq j$ [10].

Definition 2.11 (Matrix profile)**.** A matrix profile MP $\in \mathbb{R}^{(n-l+1)}$ of a data series T is a meta data series that stores the z-normalized Euclidean distance between each subsequence and its nearest neighbor, where n is the length of T and l is the given subsequence length [10].

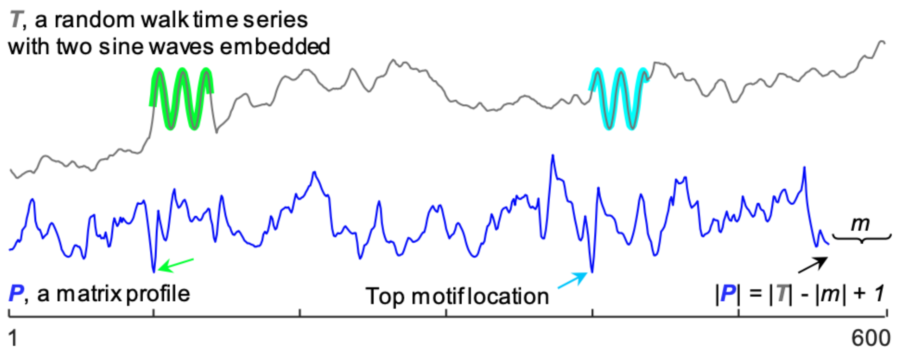


Figure 2.6: The higher grey line T shows the time series, the lower blue line instead represents its matrix profile P. The element of P at position i, is the Euclidean distance between the subsequence starting from position i of T and its nearest neighbor. Figure taken from [2].

Therefore, it is clear to understand that the lowest point of the matrix profile represents the subsequence with the smallest distance to another subsequence, which corresponds to the best possible motif for the given time series and motif length.

The vector is called matrix profile because one (space-inefficient and naïve) approach to calculate it would be to calculate the full distance matrix of all pairs of subsequences in the time series T, and then find the minimum value of each column [21].

2.6 State of the art in motif discovery

Time series motif discovery was first introduced in 2002 [11] with a technique based on hashing to find repeated motifs. Since then, many algorithms for motif discovery have been devised focusing on many different applications [14].

Nowadays significant progress has been made on the scalability of motif discovery, and datasets with lengths in the order of tens of millions can be easily searched on conventional hardware [2].

Many of the state-of-the-art algorithms are based on finding a discrete approximation of the time series (several of them for example use the SAX representation [14, 18]), inspired also by the vast literature of motif discovery in discrete data (e.g. DNA sequences) [1, 4, 14].

Moreover, most of the research has focused on the creation of fast approximate algorithms for motif discovery [7, 16]: although it is more desirable to solve a problem (namely, finding motifs) exactly than solving it approximately, solving it exactly is harder and requires higher computational and time overheads [13].

Another critical improvement achieved during the years is in the number of parameters required. For one of the older motif discovery algorithms (PROJECTION [1]), many parameters needed to be set, and still, the results produced were often only approximately correct [10]. Researchers managed to diminish the parameters' number during the years [16], and for current state-of-the-art algorithms, one single parameter needs to be set, namely the desired motif length [10].

Many state-of-the-art algorithms are based on efficient ways to compute the matrix profile [2, 9, 10, 20, 21], to find the best matches for each subsequence. The motif pair is the smallest of these matches. However, this exact search of motif pairs is only computationally feasible by fixing a predefined length for the motif.

Even if the user is knowledgeable about the domain of the data, in several cases, searching only for a unique, fixed motif length is insufficient, since there can be motifs of different lengths in the same time series [10]. In fact, recent research has shown that to find motifs in many real-world applications, it is required to have different motif lengths. [6]. Thus, research (e.g. VALMOD algorithm [10, 14]) in the last years has focused also on variable-length motif discovery.

Another common approach in motif discovery is to use random projection to discover approximate motifs at a given length [1, 7]. However, due to its probabilistic nature, this approach does not guarantee the finding of the exact (best) set of motifs [4].

Mueen, Keogh, Zhu, Cash and Westover (2009) [16] proposed an algorithm, the MK algorithm, to find exact motifs. The algorithm can analyze very large datasets by using early abandoning on a linear reordering of data [4].

In summary, we can say that in the last two decades much progress was done to improve motif discovery algorithms. Research is currently moving towards more scalable, parameter-light, variable-length, and efficient motif discovery algorithms.

Chapter 3

Motif discovery algorithms

This chapter will focus on the analysis of the motif discovery algorithms that have been integrated into the interactive visualization interface. Section 3.1 will provide a brief explanation of the brute force algorithm to find motifs, considered the very first and naive approach to motif discovery. Section 3.2 will discuss the Mueen-Keogh (MK) algorithm, Section 3.3 will give a high-level explanation of how the STUMPY package (based on the STOMP algorithm) works, and finally, Section 3.4 will go through the random projection algorithm devised in [1].

3.1 Brute force algorithm

The brute force algorithm is considered the naive approach to motif discovery. Despite the fact that it finds motifs exactly, it is not utilized in practice since it has unfeasible running times. However, it has been the starting point of several state-of-the-art motif discovery algorithms (i.e. MK algorithm [16] discussed in Section 3.2) which try to improve its efficiency by pruning off parts of the search space using some heuristic information. The algorithm tests every single combination of pairs of subsequences through a pair of nested loops, while testing it keeps a running minimum (*best-so-far*) which is updated whenever a pair of subsequences having a smaller distance between them is found. Finally, it returns the time series pair L_1, L_2 which have the minimum distance between them [16]. Its time complexity is $O(m^2w)$ where m is the total number of possible subsequences and w is the length of the time series. The following table, taken from [16], presents the pseudocode of the algorithm.

Algorithm 1 Brute Force Motif Discovery

```
    Input    D:          Database of Time Series
    Output   $L_1, L_2$:  Locations for a Motif
1: procedure $[L_1, L_2] = BruteForce\_Motif(D)$
2:     best-so-far = INF
3:     for $i = 1$ to $m$ do
4:         for $j = i + 1$ to $m$ do
5:             if $d(D_i, D_j) <$ best-so-far then
6:                 best-so-far $= d(D_i, D_j)$
7:                 $L_1 = i, L_2 = j$
```

3.2 MK algorithm

The Mueen-Keogh (MK) algorithm devised by Mueen et al. (2009) [16], optimizes the double nested loop naive algorithm (see Section 3.1) by pruning off a large fraction of the search space using some heuristics information. As the brute force algorithm, it is an exact algorithm.

In the remaining part of this section, a high-level description of the intuition behind the MK approach to motif discovery is provided, summarizing the more detailed explanation provided in [16].

Each subsequence is initially represented as an object (a point) in a 2D space, where the position of an object in the space depends on the features of the subsequence (see Figure 3.1.A).

Before the MK algorithm starts, we suppose that the *best-so-far* distance for the motif pair is equal to infinity. Then, we choose a random object as a reference point, and we order all the other objects based by their distances to the chosen reference point. This step allows us to project the time series in a 1D representation (see Figure 3.1.B). As a consequence of this step, it is possible to use the distance between the reference object and its nearest neighbor, to update the *best-so-far* distance. As we sort the objects we can find the distances between adjoining pairs (see Figure 3.1.C). However, these distances are generally not the real distances between objects in the original space, rather they are *lower bounds* to those real distances.

This linear ordering of data tells us useful heuristic information to lead the motif search process: if two objects are close to each other in the original space, then they must also be near to each other in the linear ordering. However, the opposite is false: Two objects could be near to each other in the linear ordering but very distant in the original space.

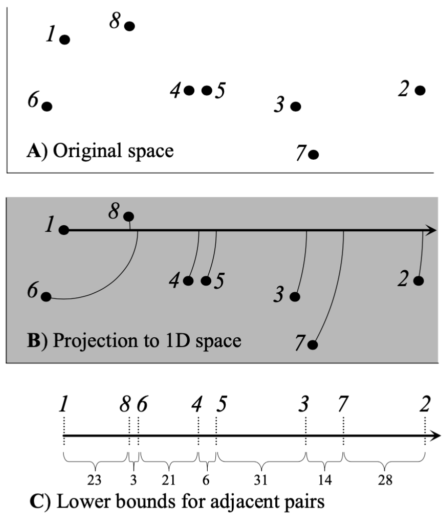


Figure 3.1: **A**) A sample dataset of 2D time series objects. **B**) These objects can be displayed in a 1D representation by computing their distance to a randomly chosen point, in this case O_1. **C**) The distances between adjoining pairs along the linear projection is a lower bound to the real distance between them. Figure taken from [16].

In the next stage, the algorithm scans across the linear ordering and measures the real distances between adjoining pairs. If during this process, we find a pair that has a distance less than the current *best-so-far*, we can update it (see Figure 3.2).

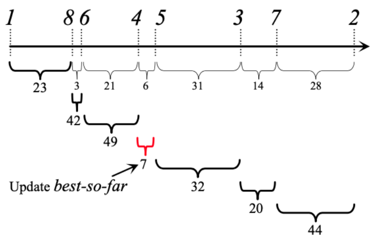


Figure 3.2: By scanning the objects from left to right, we then measure the real distances between them. Figure taken from [16].

At this point, we do not know whether the current best-so-far is the true motif. However, it is possible to utilize the linear representation together with the *best-so-far* to cut off a huge part of the search space. Specifically, it is possible to take a sliding window with a width equal to the *best-so-far*, and slide it across the linear order, testing for possible pairs of objects that could be the true motif. As shown in Figure 3.3 a requirement for two objects to be the motif is that they both need to intersect the sliding window at the same time.

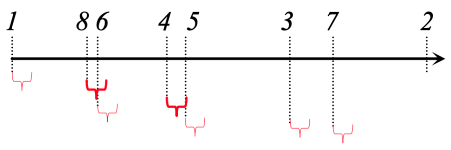


Figure 3.3: A condition needed for two objects to be the motif is that both intersect (at the same time) a sliding window of width *best-so-far*. Only the pairs $\{O_8, O_6\}$ and $\{O_4, O_5\}$ remain after the sliding window pruning test. Figure taken from [16].

This is the main idea behind the MK algorithm. The full algorithm varies in some parts from the previous high-level explanation:

- Not all objects are as good as a reference point, by using a simple heuristic it is possible to to find good reference points.
- For large datasets we might need different rounds of pruning: we can (and should) repeat the pruning step with several reference points to reduce the search space.

As depicted in Figure 3.4, average-case performances of the MK algorithm are dramatically better compared to the ones of the brute force algorithm; however, it should be noted that worst-case running time for MK is still quadratic ($O(n^2)$ as for brute force).

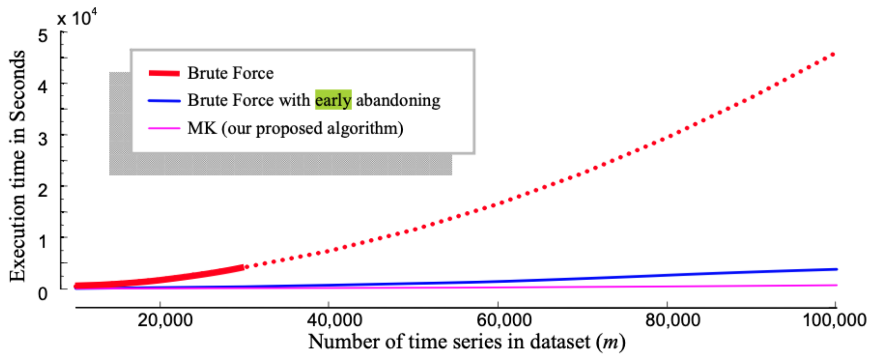


Figure 3.4: Three algorithms are here compared in the time they take to discover the motif pair in progressively larger datasets. The differences in the execution times are dramatic, for 100,000 objects brute force requires approximately 13 hours, while MK algorithm takes only about 12 minutes (with a standard deviation of about 1 minute). Figure taken from [16].

3.3 Matrix profile algorithm

The implemented algorithm that exploits the matrix profile approach, is based on the *Stumpy* library [9]. This library provides a host of different functionalities among which the efficient computation of the distance profile, and more importantly of the matrix profile.

In order to compute the latter in an efficient way, *Stumpy* provides the *stump* method which uses a technique based on the STOMP (Scalable Time series Ordered-search Matrix Profile) algorithm introduced in [21]. In the remaining part of this section, I will summarize some of the insights on the STOMP algorithm provided in [21].

The STOMP algorithm is similar to the previously devised STAMP (Scalable Time series Anytime Matrix Profile) [20] as they both can be considered highly optimized nested loop searches, where in the inner loop the distance profiles are repeatedly calculated.

To compute the distance profile, the z-normalized Euclidean distance is used. In order to compute it more efficiently, they leverage the fact that subsequences are overlapping to compute the dot products of a vector with all the subsequences of the time series in time $O(n)$, where n is the length of the time series. Similarly, they compute the means and standard deviations of all subsequences in time linear in the length of the time series.

The difference between STAMP and STOMP lays in the fact that STAMP evaluates the distance profiles in random order (to permit its anytime behavior), instead, STOMP does an ordered search. By exploiting the locality of these searches, the time complexity can be reduced by a factor of $O(logn)$, in fact the time complexity of STOMP is $O(n^2)$ while the one of STAMP is $O(n^2 logn)$. Even though the $O(logn)$ speedup does not make a great difference for small time series, (few tens of thousands of data points) when we consider datasets with millions of entries, it results in an extremely useful order-of-magnitude speedup.

A time complexity of $O(n^2)$ can be considered optimal for any exact motif algorithm, in addition, it should be noted that $O(n^2)$ is a great improvement from the brute-force approach to computing the matrix profile which takes $O(n^2 m)$ where n is the length of the matrix profile and m the length of the subsequence. The naive matrix profile algorithm is similar to the brute force algorithm introduced in Section 3.1.

3.4 Random projection algorithm

The random projection algorithm introduced by Chiu, Keogh, and Lonardi (2003) in [1], discovers motifs using a probabilistic method derived from approximate matching in DNA string. It is a fixed-length approximate motif discovery algorithm.

The following figures will consider a time series with 1000 data points, containing two occurrences of a motif of length 16 at time T_1 and time T_{58}. The occurrence at time T_{58} is corrupted by some noise in positions 8 to 12, for simplicity, we assume this is the only high-quality motif in the dataset. These figures, as well as the following algorithm description, are a summary of the thorough explanation provided by Chiu, Keogh, and Lonardi (2003) in [1].

The algorithm begins with the extraction of subsequences of length n (where n is the desired motif length) from the original time series. Each of these subsequences is transformed into its symbolic SAX representation and it is placed into matrix $\hat{S}$.

In the simple example, we are considering the $\hat{S}$ matrix has $m-n+1$ rows (where m is the length of the time series). However, generally, the $\hat{S}$ matrix is smaller: by excluding useless subsequences (for instance the ones which are close to straight lines) and by grouping identical ones (this is made possible by the use of the SAX symbolic representation), the $\hat{S}$ matrix can be reduced to even up to one-fifth of the length of the original time series.

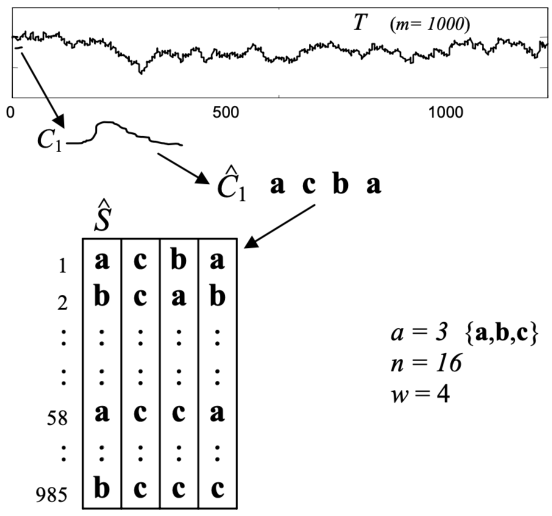


Figure 3.5: A time series is here preprocessed ready to go for the random projection stage. A sliding window takes the first subsequence C_1, and converts it into the symbolic version $\hat{C}_1$, this is then inserted in the matrix $\hat{S}$ The index of $\hat{S}$ points to the original location (index) of the subsequence. Figure taken from [1].

Once the $\hat{S}$ matrix construction is over (Figure: 3.5), the random projection step is ready to start. We select randomly c columns of $\hat{S}$ (in the example c is 2) to act as a mask. For example in Figure 3.6, columns 1 and 2 were selected to act as the mask, hence the $k = 985$ words in the $\hat{S}$ matrix are hashed into buckets based only on their values in the 1st and 2nd columns.

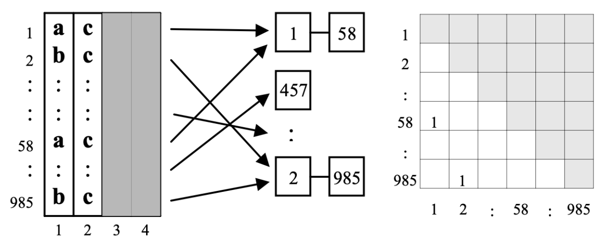


Figure 3.6: *Left*) A mask $\{1, 2\}$ was randomly chosen, thus the values in columns $\{1, 2\}$ were utilized to hash the matrix $\hat{S}$ into the buckets. *Right*) Collisions are registered by incrementing the right location in the collision matrix. Figure taken from [1].

If two words corresponding to subsequences i and j are hashed to the same bucket, we increase the count of the cell *(i,j)* in a collision matrix, that was previously initialized to all zeros. In order to avoid false positives and false negatives, we need to repeat this operation several times using different, randomly chosen masks. For each iteration, the buckets are reset while the collision matrix remains the same (see Figure 3.7).

After repeating the process several times, we analyze the collision matrix.

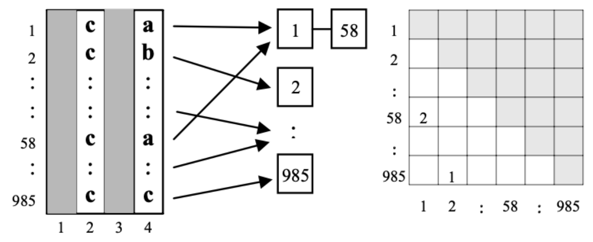


Figure 3.7: *Left*) A mask $\{2, 4\}$ was randomly chosen, thus the values in columns $\{2, 4\}$ were utilized to hash the matrix $\hat{S}$ into the buckets. *Right*) Also here collisions are registered by incrementing the right location in the collision matrix. Figure taken from [1].

A big value in a cell does not assure the presence of a corresponding motif, but it is a very strong indicator since it means that several subsequences are represented by the same SAX word. To confirm we must analyze the original data. Therefore, we retrieve the two original subsequences corresponding to the indices of the biggest value cell in our collision matrix, in our case, C_1 and C_{58}. We compute their distance (here ED is used). Supposing that the two subsequences are within R of each other, they form a tentative motif.

However, other subsequences might be within R of the subsequences, and thus should be added to this provisional motif. The way to find them mainly depends on how our original time series T is stored. If the time series is stored in the main memory, we can just do a sliding window scan to find additional members of the motif. Instead, if T resides on disk, we should limit costly disk accesses as much as possible. To achieve this, we use the collision matrix as a heuristic. If a subsequence C_i is similar to C_1 and C_{58} then the cells of the collision matrix at $(i, 1)$ and $(i, 58)$ must also have values that are significant. Therefore, it is possible to analyze the collision matrix to discover promising candidates for retrieval. However, before accessing the disk to retrieve them, we can run an additional check to see whether the candidate is a real motif or not.

Since $\hat{S}$ is generally stored in main memory, we know the words that correspond to all the subsequences, and we also know R. Thus, we will retrieve C_i from disk if and only if $MINDIST(C_i, C_1) \leq R$ or $MINDIST(C_i, C_{58}) \leq R$, where $MINDIST$ is a function that returns a lower bound to the distance between the original time series represented by two words, defined as follows:

$$MINDIST(\hat{Q}, \hat{C}) \equiv \sqrt{\frac{n}{w}}\sqrt{\sum_{i=1}^{w} \left(dist(\hat{q}_i, \hat{c}_i)\right)^2}$$

Once all matching subsequences within R of C_1 and C_{58} have been discovered, they can be reported to the user, and we begin to iteratively examine the collision matrix for the next biggest value which has not been previously analyzed, and which is not within R of a previously reported motif.

Regarding the stopping criteria, there are three possibilities:

- Establish a fixed iterations' number i.
- Stop when the user does not want to spend more time waiting. Given that the algorithm is an anytime algorithm, the user might be satisfied with the first few motifs they see, or be dissatisfied and wish to modify the parameters.
- Stop when the biggest value in the collision matrix is no greater than it would be expected by chance.

The time complexity of this algorithm has been shown to be $O(i|\hat{S}|)$, where i is the number of iterations, $\hat{S}$ the collision matrix, and $|\hat{S}|$ corresponds to the number of cells in the collision matrix in the worst-case scenario which have a non-zero entry. Since the size of the sparse collision matrix is linear in $|T|$, we can define the time complexity of the algorithm as $O(|T|)$. This is a significant improvement from the brute approach which has a time complexity of $O(|T|^2)$. Both the algorithms have a space complexity $O(|T|)$. Several experiments reported in [1] have proved that this algorithm is also reasonably robust to noise. It should be noted that this algorithm requires many parameters to be set, therefore, its utilization requires a deep knowledge of the data domain and of the algorithm itself.

Chapter 4

Implementation

This chapter provides detailed information on how the interactive visualization interface was implemented. In particular, Section 4.1 will discuss the technologies utilized, Section 4.2 will explain the sources and the format of the data used and Section 4.3 will provide some insights on the different visualization tools that are available and which one was decided. Then, Section 4.4 will analyze how the MVC structure was implemented in the application, Section 4.5 will outline the algorithms used in the project and how they were chosen and finally, Section 4.6 will explain how and why the application uses a plugin architecture and how a new motif discovery algorithm can be implemented in the program using this plugin structure.

4.1 Technologies

For the development of the online interface and the analysis of data, I decided to work entirely with *Python* which has emerged in the last years as the number-one language for data science and more in general for scientific computing tasks [19]. The usefulness and popularity of *Python* stem mainly from the wide range of third-party packages, in particular for this project: *Numpy* for the efficient manipulation of array-based data, *Pandas* to manipulate and analyze labeled data (i.e. time series), *Plotly* for interactive data visualization (see Section 4.3).

The interactive visualization interface was build using *Dash*, an open-source *Python* framework for the development of web applications. *Dash* is perfect for developing data visualization applications with highly customized user interfaces in plain *Python.* In fact, despite the web app is rendered in the web browser, no *Html* or *Javascript* code is required. Dash applications are based on a *Flask* server that interacts with front-end *React* components using *JSON* packets over *HTTP* requests. For the graphs and charts components, *Dash* uses the *Plotly's Python* graphing library which is in turn powered by the *Plotly.js* standalone *Javascript* data visualization library. The very *Dash* framework was developed by the developers of *Plotly.js.*

Dash applications consist of two parts, the first one is the "layout", which defines the application front-end. For the layout, we might have a wide range of possible components, the two main families of components are *dash_core_components* and the *dash_html_components.* The former provides charts, graphs, sliders, and many more elements related to interactive data visualization, the latter instead supplies the *Python* version of *html* tags such as *div, h1, p...*

The second of the two parts constituting *Dash* applications is the "callback" which allows your application to react to user interactions. *Dash*'s callback functions are *Python* functions with an *"app.callback"* decorator: when an input component's property changes, a callback function is automatically triggered. The function performs some defined operations, like filtering a dataset, and returns the output to the application. To put it simply, callbacks link inputs and outputs in your app.

4.2 Data

As mentioned in Chapter 1 in the introduction, the data utilized for this thesis were collected by sensors placed on industrial machines which take measurements at regular intervals (i.e. 10 seconds, 5 minutes). This vast amount of collected data has then been cleaned and stored in databases.

In order to access the Technoalpin data, the API provided by Dr. Matteo Ceccarello was used. The implementation of the API can be found in the *technoalpin.py* file, it allows to easily retrieve as lists the resorts, the machine rooms, the available sensors, and eventually to retrieve a *Pandas* dataframe containing the time series for a sensor. In particular, the dataframe has the timestamps of when the measurements were taken as index, some columns (running, warning, error) containing boolean flags, and a column named as the sensor containing the value of the measurement. The following table shows an example:

	running	warning	error	AT201
timestamp				
2021-02-15 11:00:00	True	False	True	-5.384000
2021-02-15 11:10:00	True	False	True	-5.080000
2021-02-15 11:20:00	True	False	True	-4.801667

To access the Durst data, instead, another API developed by Dr. Anton Dignös and Dr. Matteo Ceccarello was utilized. The API implementation can be found in the *durst.py* file.

Also in this case, it allows to easily obtain the list of the printers, the available sensors for the selected printer, and eventually a dataframe containing the time series for the selected sensor. The indexes of the dataframe will be the timestamps of when the measurements were taken, the column represents the sensor (in this case the name of the sensor is “20”). The next table depicts an example of how Durst data is retrieved.

	20
timestamp	
2020-07-30 00:20:00+02:00	39.500000
2020-07-30 00:30:00+02:00	40.744737
2020-07-30 00:40:00+02:00	39.500000

Both the APIs allow setting a *downsample* parameter when recovering the time series dataframe with the *get_timeseries* method, to reduce the number of data points. For example, we can set the downsample to be “10 minutes” and the *aggregation* style (another parameter) “avg” to get a single data point for every 10 minutes of data; this data point will be the average of all the measurements we have within that 10-minute timespan. This is an example of the PAA technique explained in Subsection 2.3.1.

It is also possible to upload a *csv* file or an *excel* document containing a time series as input data of the application. In this case, the table must have just two columns: the index and the sensor. There should also be a header indicating the names of the two columns.

4.3 Visualization tools

To visualize time series, or more generally data, *Python* provides a wide variety of libraries and frameworks, such as *matplotlib, seaborn, ggplot, plotly, bokeh, pygal* [12]. Figuring out which one works better, highly depends on what you are trying to accomplish.

Matplotlib is arguably the most popular one. Despite being over a decade old it is still very good for quick static visualization; *matplotlib* proves to be very powerful but with power often comes complexity [12].

Another popular choice is *seaborn* which can be seen as an evolution of *matplotlib*. It was designed to be more modern and visually pleasing, it also aims to make more complicated plots easier to create [12].

Both the aforementioned libraries were not designed for a high level of interactivity with the end-user, in fact, they prove to be particularly good for static visualization, but they are not the best choice when it comes to creating interactive visualizations. Since the web app developed for this project required a high level of user interaction, the choice of the visualization tools narrowed down to *plotly* and *bokeh*.

Each of them comes with its own advantages and disadvantages, the remaining part of this section will analyze them and provide a summary mainly based on what is reported in [8] and on my own experience.

Both the libraries allow the creation of figures via an object-oriented interface. They are both built on a producer-consumer architecture that communicates over *JSON*. It means that *JSON* objects are generated on the server and then they are sent to a JavaScript library running browser-side. In particular, the *plotly.js* library and the *bokeh.js* library respectively, handle the displaying of the data in the browser. No *Javascript* knowledge is required to use the two tools.

Both the libraries integrate with a python-based web server: Flask for *plotly* and Tornado for *bokeh*. Knowledge of how these servers work is not necessary unless one wants to heavily customize the app. However, it should be mentioned that in the *bokeh* backend, *Tornado*, communication between server and client is carried out on a continuously connected "pipe". This means it is faster and it is asynchronous. Conversely, the *plotly* server backend *Flask*, is configured to be synchronous: for instance, *plotly* dashboards cannot easily save intermediary calculations.

After extensive research on the internet, and after experimenting with both the libraries, I arrived at the conclusion that *plotly* was the best suit. Its easiness of use, the high level of interactivity, the extensive online documentation, and the huge online community were the key factors that led me to its choice.

However, the choice of the *plotly* library (and consequently the *Dash* framework), led to several obstacles. The fact that *Dash* does not allow the use of global shared variables between callbacks, is stateless, and the fact that a component (i.e. a graph) can be the output of maximum one callback function, made the development of some functionalities harder. Since the application graphs and components are highly interconnected, share data and depend on each other, in some cases, to make components work, it was necessary to write lengthy callback functions with many input parameters and many outputs.

4.4 MVC architecture

Dash applications are basically dashboards, namely user interfaces (UIs) which visualize data in an organized fashion. UIs lend themselves to the MVC structure, therefore for the development of this web application I decided to follow the model-view-controller pattern, which guarantees several advantages, such as improved application maintainability.

To briefly summarize this software design pattern we can say that an application can be divided into 3 layers that interact with each other, where each layer has a specific function, in particular:

- *Model*, handles the application data and data management.
- *Controller* stands between the view and the model. It listens for events initiated by the view and performs the proper action. In most cases, the action consists of invoking a function in the model.
- *View*, where data is presented to the user, namely the front end.

In the particular case of *Dash* applications, it is easy to notice how the *layout* part matches the *view* layer functionalities, while the *callback* part plays the role of the *controller*. Finally, the *model* layer is represented by the remaining code which represents the application data, data manipulation and management, plugins, and more generally the application data structures and logic.

4.5 Algorithms

The application developed provides 3 different motif discovery algorithms: the MK algorithm [16], the algorithm based on the matrix-profile approach presented in [9], and the algorithm explained in [1] based on the probabilistic discovery approach.

Since writing a motif discovery algorithm (either an existing one or a new one) was beyond the scope and the limits of this thesis, I had to rely on already-existing implementations of the algorithms.

In particular, I went through a vast list of motif discovery papers that presented different algorithms using different approaches. For all the algorithms found, an extensive search on the

Internet was carried out, looking for Python implementations of them. The results of the research led to the finding of the implementations for the three 3 aforementioned algorithms, all 3 the implementations were found on *GitHub*, the links to these resources are provided in the application code. While the STUMPY package (which provides the implementation of the matrix profile algorithm) is widely used, supported, and documented, the other two implementations found on *GitHub* were not as reliable and documented as STUMPY.

In particular, the MK algorithm implementation shows often a bug, namely, it returns as a motif a subsequence at the very end of the time series, of a different length from the one indicated in the motif length input, causing an error in the application. Additionally, the occurrences returned by MK are often not precise, contradicting the exact nature of the algorithm itself.

The probabilistic discovery implementation found, seems reliable, despite its long running time, and despite the fact that it uses a parameter d (to compute the distances between subsequences) which is not discussed and used by the algorithm's authors.

Finally, it is possible to implement new algorithms, either by writing them from scratch in Python or by using other already-existing implementations which I did not found or by writing some Python wrappers which convert algorithms written in other languages to Python format. The next Section will explain how an algorithm can be easily added to the program.

4.6 Plugin Architecture

One of the chief requirements for this project was the possibility to further extend the application in the future beyond this thesis' scope, with the addition of new motif discovery algorithms.

In order to comply with this need, a plugin architecture has been adopted, which easily allows the implementation of new motif discovery algorithms:

A global dictionary in the *callbacks.py* file stores each available algorithm as a key, and an instance of the respective algorithm plugin class as corresponding value. As just mentioned, each algorithm needs a respective *"-plugin"* class that extends the *"MotifDiscoveryAlgorithm"* class whose implementation can be found in the *Plugin_interface.py* file.

Each *"-plugin"* class must override the following parent methods: *find_motifs*, *get_parameters*, *get_motif_occurrences*, *clear_attributes* and create a *get_motifs* method, also the *has_next_motifs* boolean variable must be overridden.

Even though this may seem a lot of work, most of the just mentioned methods are trivial and short to override, by just looking at already existing plugins. In any case, a detailed explanation ensues.

The *find_motifs* method will take the time series and a dictionary of parameters as input, it will run the algorithm (a python implementation or wrapper must be provided) with the given parameters and it will save the output of the algorithm in some global variables declared and initialized in the child class constructor. In particular, a global list called *motifs* should save all the motifs as tuples, where each tuple contains the beginning indexes of the two occurrences of the motif. In case the algorithm returns many motifs, it is possible to remove the trivial motifs and keep only the most relevant ones by calling the parent class *remove_trivial_motifs* method with the newly created (and populated) "motifs" list and the motif length as parameters.

The method *get_motif_occurrences* is the method called in the *callbacks.py* file. The method takes the time series, a dictionary of parameters, and a counter has inputs. It should just invoke the parent class respective method and pass the right parameters. The already-defined parent class method deals with all the logic to return the right motif and all its occurrences. This method checks whether the algorithm was already run or if it still has to run, by checking the content of global variables. Additionally, using the counter value it will understand which motif to return.

The order and the type of the parameters in *parameters_dictionary* (which is one of the two inputs of the *find_motifs* method and also an input of *get_motif_occurrences*) are defined by the dictionary returned by the *get_parameters* method which, has mentioned before, should be overridden and defined according to the new algorithm, the keys in the dictionary contain the names of the parameter, the values contain their respective type (*number* or *text*).

The *clear_attributes* method should reset and reinitialize all the global variables, in order to make sure that no data from previous computations (of different time series and/or parameters) is kept cached, once a new computation is required.

The child class constructor should define and initialize the global variables needed, typically three: motif length, the time series, and *motifs* which is a list containing the motifs (as tuples) found by the motif discovery algorithm.

Since some algorithms return just the top one motif and not a list of motifs, a boolean flag *has_next_motifs* was also needed, also this global variable defined in the parent class, should be overridden.

Finally, a new method *get_motifs* should be defined, this method returns just the global variable *motifs* mentioned earlier.

In any case, the best way to understand how the plugin structure works is to look at the already implemented plugin classes and at the *MotifDiscoveryAlgorithm* class and read the comments.

Chapter 5

Experimental results

This chapter discusses the results and the findings achieved after thoroughly testing the 3 different algorithms implemented in the program. In particular, Section 5.1 will explain how the tests were carried out, Section 5.2 will analyze the different running times of the algorithms and Section 5.3 will examine the quality and the precision of the motifs found. Section 5.4 will describe the limitations of the tests, and, finally Section 5.5 will provide insights about the algorithms in order to guide the user in the choice of the right one.

The tests were done on a MacBook Pro 2015, with a 2,7 GHz Dual-Core Intel i5 processor and 8GB of RAM. The algorithms were coded in Python and run on the PyCharm IDE. The results of the tests are available in the code repository associated to this thesis[1], where it is possible to see the precise parameters used for each test.

5.1 Tests

Several tests were conducted, using different time series with different lengths, different parameters for each algorithm, and different motif lengths.

Given the buggy behaviors of the MK algorithm implementation (see Section 5.4), I initially decided to focus on the other two algorithms: the matrix profile and the probabilistic discovery algorithm. Even though several tests were also run using the MK algorithm, I do not consider it wise and meaningful to draw conclusions on an algorithm based on empirical data from a buggy implementation of the algorithm itself.

The datasets used for the tests are two: a time series of a sensor (*AT201*) of Technoalpin, and a randomly generated dataset created using the *numpy* package *randn* and *cumsum* functions.

The next step was to understand how to properly set the (many) parameters of the probabilistic discovery algorithm. The only example I had was from [1] for a 1000-point dataset. Since the data I had was insufficient to understand how to set the parameters, I decided to conduct some tests using different parameter settings and then I compared the results. In particular, I noticed that keeping the number of iterations over 50 and the SAX word length longer than 10, was not meaningful since it considerably increased the running times while the quality of the motifs found was almost always the same as the one of the motifs found using fewer iterations and a shorter word length.

Next, I started testing the two different datasets with the two aforementioned algorithms. Firstly, I considered different sub-portions $(100, 1000, 2500, 5000, 10000)$ of the datasets with respective motif lengths $(15, 75, 150, 300, 500)$, then I considered sub-portions $(2500, 5000, 7500, 10000)$ of the datasets with always the same motif length (250).

Given the great performance shown by the matrix profile algorithm, I decided to test it with longer input datasets $(10000, 20000, 40000, 80000, 160000)$, using always the same motif length of 2500.

Finally, I tested the MK algorithm. Its faulty implementation forced me to run the tests several times. Sometimes in order to get a result, I had to change the inputs time series and motif lengths (which I tried to keep as similar as possible to the ones of the tests of the other two algorithms). Also the MK algorithm was tested with the very long input datasets mentioned above for the matrix algorithm.

The next section will discuss the results of the tests in terms of running times.

[1] `https://gitlab.inf.unibz.it/Francesco.Piccoli/thesis/`

5.2 Running times analysis

The vast majority of the tests run were focused on analyzing the running times of the algorithms.

The theory behind the algorithms suggested that MK has as worst-case time complexity $O(n^2)$, the same as the brute force motif discovery algorithm, however, tests on the algorithm reported in [16] show that the average time complexity is lower than the brute force one (see Figure 3.4). The STOMP matrix profile has also time complexity $O(n^2)$, ideal for any exact motif discovery algorithm. Finally, the probabilistic discovery algorithm according to [1] has a linear time complexity $O(n)$, which is an excellent running time for an approximate motif discovery algorithm.

However, the tests on the implementations of the algorithms showed different results. The differences and the apparent contradictions between theory and practice are due to several reasons: firstly, the python implementations used for the MK and for the probabilistic algorithm cannot be considered completely reliable as they are not properly documented, tested, and reviewed. Secondly, some huge constant factors which are not indicated in the time complexity notation might significantly influence the running times, especially for the probabilistic approach. And ultimately, the different processes run by the computer, the parallel computing, and the multithreading behind each algorithm might also heavily condition the total times elapsed.

The experiments showed that the first run of each algorithm took considerably longer compared to the next runs, especially for the STUMPY matrix profile algorithm. This is probably due to some one-time initializations of internal data structures carried out during the first run. In order to have more reliable results, each single test case was repeated several times: 10 times for the matrix profile and the MK algorithms, 3 times for the probabilistic algorithm given the long running times. Then the average of all the n runs was taken, excluding from the computation the minimum and the maximum values. The maximum value, which usually corresponds to the very first run, is excluded because it includes the times for the initialization of the internal data structures, as outlined earlier. In order to offset the exclusion of the maximum value and obtain more trustworthy results, also the fastest (minimum) running time was excluded.

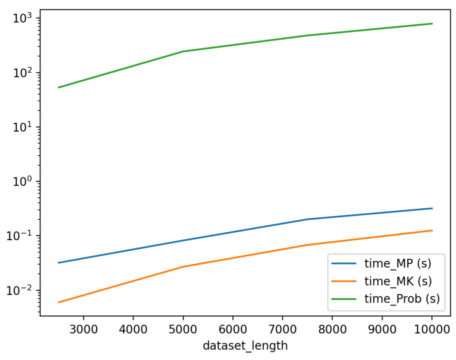
The matrix profile and the MK algorithms show the best running times: for the small datasets considered, their performance is very similar, with a slight edge for the MK algorithm. However MK's low precision (see Section 5.3) and its limitations (see Section 5.4), make it hard to tell whether this implementation's running times truly reflect the ones of the original algorithm discussed in [16].

The probabilistic discovery algorithm, which theoretically was the fastest, is instead the slowest, with poor performance even for short datasets.

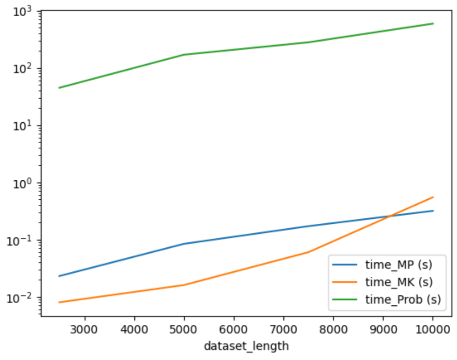
Another interesting result that emerged from the tests and is visible in the following table, is that the performance of matrix profile seems to be data independent: for a fixed dataset length the difference in running time between the two time series is less pronounced for the matrix profile algorithm than for the other two. Even though it's hard to make conclusive statements comparing just two time series, it is surely a noteworthy remark.

The following table shows the total times (in seconds) that the 3 different algorithms took to find the motifs. "MP" stands for the Matrix Profile algorithm, while "Prob" stands for Probabilistic algorithm. In Figure 5.1 a plot (on a logarithmic scale) of the table content.

datasource	dataset_length	motif_length	time_MP(s)	time_MK(s)	time_Prob(s)
technoalpin	2500	250	0.032	0.006	53.224
random_input	2500	250	0.023	0.008	44.874
technoalpin	5000	250	0.082	0.027	244.266
random_input	5000	250	0.084	0.016	169.685
technoalpin	7500	250	0.201	0.068	480.084
random_input	7500	250	0.171	0.06	279.238
technoalpin	10000	250	0.319	0.125	791.043
random_input	10000	250	0.317	0.547	592.039



(a) Technoalpin data



(b) Random dataset

Figure 5.1: A visual comparison of the running times of the 3 algorithms.

The next table, instead, shows the total times (in seconds) that the top two time-efficient algorithms (MK and Matrix Profile) took to find motifs on significantly longer randomly generated datasets. It can be noticed the quadratic behavior of the two algorithms: every time that the size of the input dataset doubles, the running times more or less quadruple. In Figure 5.2 a plot of the table content.

dataset_length	motif_length	time_MP(s)	time_MK(s)
10000	2500	0.366	0.021
20000	2500	2.042	0.102
40000	2500	6.096	3.317
80000	2500	26.603	12.877
160000	2500	109.634	61.292

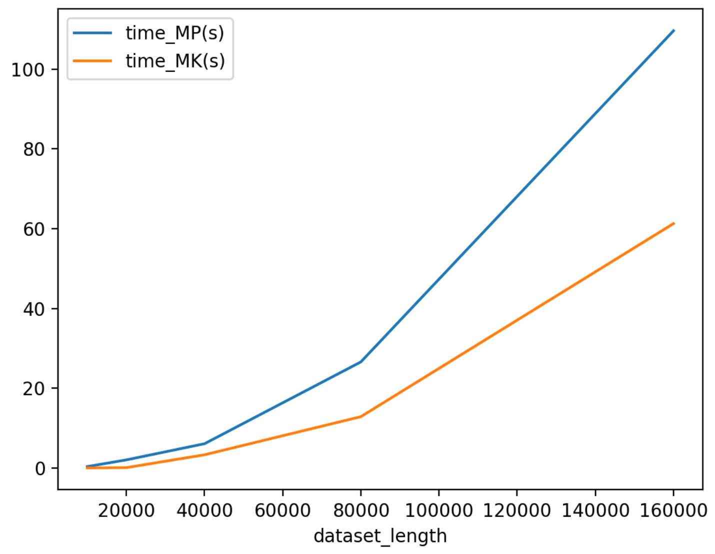


Figure 5.2: A visual comparison of the running times of the MK and Matrix Profile algorithms for long inputs.

5.3 Results precision

In order to measure the quality and the precision of the motifs found, I computed the Pearson correlation between the top two occurrences of the top one motif. As indicated in Subsection 2.2.2, one can easily retrieve also the z-normalized Euclidean distance using the Pearson correlation.

Before conducting the tests, one could easily deduct that given the exact nature of the matrix profile and MK algorithms, they would return the (same) best possible motif, while the probabilistic algorithm, which is an approximate motif discovery algorithm was supposed to perform worse.

The tests showed that the best results were always produced by the matrix profile approach, with the highest correlation values and the smallest z-euclidean distances between the top two occurrences.

Following (and unexpectedly), we have the probabilistic approximate algorithm which yielded always extremely similar results (if not identical) to the ones found by the matrix profile algorithm.

The least precise of the three, was the MK algorithm, with results significantly less precise if compared to the other two. This contradicts the MK exact nature, but, as already mentioned in the above sections, this is probably due to the faulty and incorrect implementation of the algorithm that was used (see Section 5.4).

In the following table, the results of the tests related to the precision of the motifs found by the different algorithms in the Technoalpin input dataset. As mentioned above, the Pearson correlation is the one between the top two occurrences of the motif.

dataset_length	motif_length	Pearson_MP	Pearson_MK	Pearson_Prob
2500	150	0.9927	0.8722	0.9927
2500	250	0.98625	0.86386	0.98607
5000	250	0.98625	0.80426	0.98607
5000	300	0.98576	0.70864	0.98559
7500	250	0.98625	0.80426	0.98607
10000	250	0.98625	0.80426	0.98607
10000	500	0.95875	0.67295	0.9581

5.4 Limitations

The main limitation of the tests is the fact that, since I did not implement the algorithms myself, it was hard to justify some behaviors which apparently contradict the theories behind the algorithms.

In particular, I am referring to the MK algorithm, whose imprecise results and its very fast running times contradict its exact motif discovery algorithm nature. Moreover, for some input (time series length and motif length) the MK algorithm cannot find motifs, in these cases, one should slightly change the input lengths and try again.

Additionally, regarding the probabilistic discovery algorithm, the lack of examples and references on how to properly set the parameters, constituted a limit, since the results might have been more precise and the running times shorter, if the parameters were set in an optimal way. It is worth noting that setting the parameters optimally is domain-dependant, and therefore requires a thorough knowledge of the data domain that I did not have.

Finally, regarding the matrix profile algorithm based on STOMP, it should be mentioned that there is a wide range of algorithms based on the matrix profile approach, therefore, it is not possible to draw precise conclusions on a general approach based on a single specific algorithm that uses this approach.

5.5 Final notes

This final section will provide insights on the different algorithms to help the user in the choice of the best algorithm for its own needs. The knowledge behind the following reflections was the results of testing and using the application and the algorithms over an extended period of time.

If the user does not care about the best possible (exact) motif, and just wants to identify approximate motifs in the time series in the shortest time possible, then the use of the MK algorithm with the "euclidean" distance measure is suggested. Despite the contradictions related to this algorithm discussed in 5.4, the algorithm is the fastest and yields decent results, especially when considering long datasets. It does not truly reflect the MK algorithm discussed in [16] but it is still able to find approximate motifs. Using the “dtw” distance measure yields more precise results but takes considerably longer.

The MP algorithm is arguably the best trade-off. It has very good running times and it is the most precise, and it takes a single parameter. Its use is suggested in all the standard situations where the user needs to find the best possible motif in the shortest possible time.

Finally, the use of the probabilistic algorithm is not recommended, it takes dramatically longer to find motifs compared to the other two algorithms, even for small datasets. It is difficult to set its many parameters, and even though its results are very similar to the (exact) ones of the matrix profile algorithm, it is still an approximate algorithm. The only reason for its use is probably when the user wants to test the algorithm itself more than finding motifs in general.

Chapter 6

Conclusion

Starting from the problem and context definitions in the initial chapters, this work described the steps followed to design and develop an interactive interface to visualize time series (with the possibility to set and customize different parameters) and discover their motifs in an intuitive and convenient way, with the additional display of insightful information regarding the motifs.

One of the key features of the application developed, is the possibility to utilize and compare different motif discovery algorithms to obtain precise insights on the motifs of a time series from the comparisons of different algorithms. The explanations and the intuitions behind each implemented algorithm were also provided in this document, allowing the reader to understand the different behaviors and the key concepts standing behind those algorithms.

In order to expand the comparison between the algorithms, the application has been designed in such a way that the addition of a new motif discovery algorithm is as easy as possible, and does not require modifications to the core code. In particular, a plugin structure has been adopted, where each algorithm is just a plugin, and adding a new algorithm corresponds to adding a new plugin.

Once the application was built, several tests were run to analyze the performances of the different algorithms. In the previous chapter, the results of the tests are reported. All the test results converge to one conclusion: the matrix profile algorithm based on the STOMP algorithm and implemented through the STUMPY library showed the best performances when considering both the running times and the quality of the motifs. The use of this algorithm is therefore recommended over the other two.

The possible future developments of this application are many, the very initial idea of this project was to build an interactive tool for the visualization of time series motifs that could have been the starting point for an even more advanced tool for time series in general. In addition to the possibility discussed above of adding new motif discovery algorithms, another possible improvement would be the inclusion of other visualizations, such as heatmaps. Moreover, the application could be expanded in order to include the possibility to analyze multivariate time series or add anytime algorithms to allow the user to stop the algorithm on their will, or again, the possibility to add variable-length motif discovery algorithms.

Bibliography

[1] Bill Chiu, Eamonn Keogh, and Stefano Lonardi. "Probabilistic discovery of time series motifs". In: *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*. 2003, pp. 493–498.

[2] Hoang Anh Dau and Eamonn Keogh. "Matrix profile V: A generic technique to incorporate domain knowledge into motif discovery". In: *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*. 2017, pp. 125–134.

[3] Hui Ding et al. "Querying and mining of time series data: experimental comparison of representations and distance measures". In: *Proceedings of the VLDB Endowment* 1.2 (2008), pp. 1542–1552.

[4] Philippe Esling and Carlos Agon. "Time-series data mining". In: *ACM Computing Surveys (CSUR)* 45.1 (2012), pp. 1–34.

[5] Tak-chung Fu. "A review on time series data mining". In: *Engineering Applications of Artificial Intelligence* 24.1 (2011), pp. 164–181.

[6] Yifeng Gao and Jessica Lin. "Discovering Subdimensional Motifs of Different Lengths in Large-Scale Multivariate Time Series". In: *2019 IEEE International Conference on Data Mining (ICDM)*. IEEE. 2019, pp. 220–229.

[7] Yifeng Gao and Jessica Lin. "Efficient discovery of variable-length time series motifs with large length range in million scale time series". In: *arXiv preprint arXiv:1802.04883* (2018).

[8] Paul Iacomi. *Plotly vs. Bokeh: Interactive Python Visualisation Pros and Cons*. 2020. URL: `https://pauliacomi.com/2020/06/07/plotly-v-bokeh.html` (visited on 04/27/2021).

[9] Sean M Law. "STUMPY: A powerful and scalable Python library for time series data mining". In: *Journal of Open Source Software* 4.39 (2019), p. 1504.

[10] Michele Linardi et al. "Matrix profile X: VALMOD-scalable discovery of variable-length motifs in data series". In: *Proceedings of the 2018 International Conference on Management of Data*. 2018, pp. 1053–1066.

[11] JLEKS Lonardi and Pranav Patel. "Finding motifs in time series". In: *Proc. of the 2nd Workshop on Temporal Data Mining*. 2002, pp. 53–68.

[12] Chris Moffitt. *Overview of Python Visualization Tools*. 2015. URL: `https://pbpython.com/visualization-tools-1.html` (visited on 04/27/2021).

[13] Abdullah Mueen. "Time series motif discovery: dimensions and applications". In: *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 4.2 (2014), pp. 152–159.

[14] Abdullah Mueen and Nikan Chavoshi. "Enumeration of time series motifs of all lengths". In: *Knowledge and Information Systems* 45.1 (2015), pp. 105–132.

[15] Abdullah Mueen, Suman Nath, and Jie Liu. "Fast approximate correlation for massive time-series data". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*. Ed. by Ahmed K. Elmagarmid and Divyakant Agrawal. ACM, 2010, pp. 171–182. DOI: `10.1145/1807167.1807188`. URL: `https://doi.org/10.1145/1807167.1807188`.

[16] Abdullah Mueen et al. "Exact discovery of time series motifs". In: *Proceedings of the 2009 SIAM international conference on data mining*. SIAM. 2009, pp. 473–484.

[17] John Paparrizos et al. "Debunking Four Long-Standing Misconceptions of Time-Series Distance Measures". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, pp. 1887–1905.

[18] Pranav Patel et al. "Mining motifs in massive time series databases". In: *2002 IEEE International Conference on Data Mining, 2002. Proceedings.* IEEE. 2002, pp. 370–377.

[19] Jake VanderPlas. *Python data science handbook: Essential tools for working with data.* " O'Reilly Media, Inc.", 2016.

[20] Chin-Chia Michael Yeh et al. "Matrix profile I: all pairs similarity joins for time series: a unifying view that includes motifs, discords and shapelets". In: *2016 IEEE 16th international conference on data mining (ICDM).* Ieee. 2016, pp. 1317–1322.

[21] Yan Zhu et al. "Matrix profile II: Exploiting a novel algorithm and gpus to break the one hundred million barrier for time series motifs and joins". In: *2016 IEEE 16th international conference on data mining (ICDM).* IEEE. 2016, pp. 739–748.