kNN matrix profile for knowledge discovery from time series

Tanmoy Mondal, Reza Akbarinia, Florent Masseglia

kNN Matrix Profile for Knowledge Discovery from Time Series

Tanmoy Mondal[1,2*], Reza Akbarinia[2] and Florent Masseglia[2]

[1*]ZENITH Team, INRIA & LIRMM, Montpellier, France.
[2]Mathematical & Electrical Engineering Department, IMT Atlantique, Brest, France.

*Corresponding author(s). E-mail(s):
tanmoy.mondal@imt-atlantique.fr;
Contributing authors: reza.akbarinia@inria.fr;
florent.masseglia@inria.fr;

Abstract

Matrix Profile (MP) has been proposed as a powerful technique for knowledge extraction from time series. Several algorithms have been proposed for computing it, *e.g.*, STAMP and STOMP. Currently, MP is computed based on 1NN search in all subsequences of the time series. In this paper, we claim that a kNN MP can be more useful than the 1NN MP for knowledge extraction, and propose an efficient technique to compute such a MP. We also propose an algorithm for parallel execution of kNN MP by using multiple cores of an off-the-shelf computer. We evaluated the performance of our solution by using multiple real datasets. The results illustrate the superiority of kNN MP for knowledge discovery compared to 1NN MP.

Keywords: Time series analysis, STAMP, STOMP, All-pairs-similarity search, Motifs and discord discovery, Outliers detection, Anomaly detection, Joins

1 Introduction

A time series is a series of data points ordered in time. As examples of time series, we can mention the height of ocean tides level captured every minute, the vibration of an aircraft engine captured every second, or the number of steps measured by a

smart watch day after day. Analyzing the time series can give us precious information about the underlying applications, *e.g.*, the anomalies in an aircraft engine.

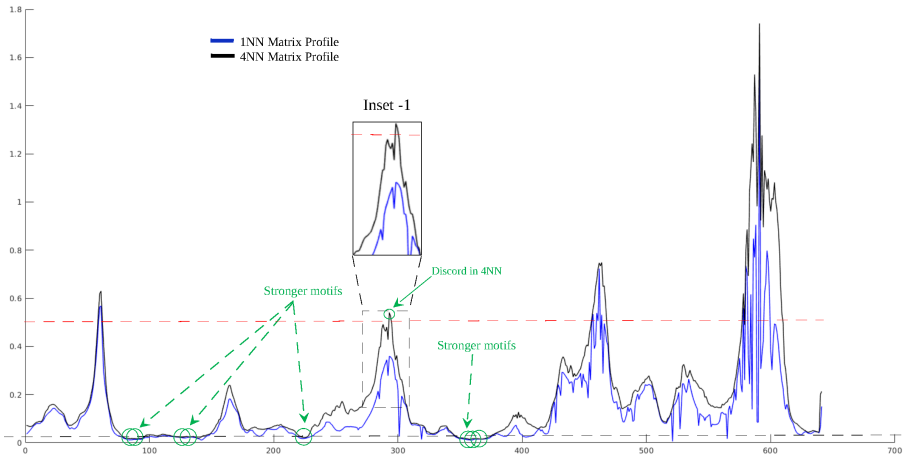


Figure 1: The 1NN and 4NN MP are plotted with different colors in which the motifs and discords are marked (please see the online color version for better visibility).

Matrix Profile (MP) [1] has been proposed as a powerful technique for time series analysis, *e.g.*, detecting motifs or anomalies. Efficient algorithms have been proposed for MP computation, *e.g.*, STAMP [2], STOMP [2] and SCRIMP++ [3]. The definition of MP in the literature is as follows [1]. Given a time series T and a subsequence length m, the MP returns for each subsequence included in T its distance to the most similar subsequence ($1NN$) in the time series. We call this type of MP as *1NN MP*. It is very useful for data analysis, *e.g.*, detecting the motifs (represented by low values), or anomalies (represented by high values).

Although 1NN MP has been shown useful for knowledge discovery, it has its own drawbacks and can miss some important motifs and anomalies (also called discords). Particularly, it does not allow to detect a cluster of discords, *e.g.*, two subsequences that are similar to each other, but dissimilar to all other subsequences. In addition, 1NN MP does not permit to distinguish a *weak motif* (*i.e.*, a subsequence that has only one similar subsequence in the series) from *strong motifs* (*i.e.*, those that have several similar subsequences).

We believe that a more powerful MP based on kNN search is of high interest, where for each subsequence, its k^{th} nearest neighbor is used for generating the MP. We call it *kNN MP*. An example is depicted in Fig. 1 where *1NN* and *kNN* (*4NN* in this case) MPs are drawn for a time series of protein spectrum, representing protein rates measured in 10 different products (only certain portion of the complete MP is shown here). As seen in Fig.1, the *strong motifs* can be detected in *4NN* MP. In the figure, the distance values in $1NN$ and $4NN$ MPs, which are less than a user defined

low threshold value (marked as black dotted line at the bottom of the Fig. 1), are termed as *"strong motifs"*. This analogy signifies that the subsequence has not only one close match (*i.e.*, *1NN* which may occur due to some noise or outlier elements) but multiple close matches, *i.e.*, *1NN*, *2NN*, *3NN* and *4NN*. This reasoning helps to increase the certitude of validating a subsequence as motif.

There are also situations where the distance in $1NN$ MP for an anomaly is quite low because the subsequence has one close match but the match can simply be another anomaly. An interesting example is shown in Fig. 1 (Inset 1), where based on the user defined threshold (dotted red line in Fig. 1), the discord can be detected in $4NN$ MP, while it is not detected in $1NN$ MP (see more examples in Section 6).

In this paper, we propose the kNN MP and illustrate its utility for knowledge discovery from time series. Our contributions are as follows:

1. We define the kNN MP, and propose a fast algorithm to calculate it in a time series.
2. We propose a technique for parallel execution of the proposed algorithm by using multiple cores of an off-the-shelf computer.
3. We evaluate the performance of our technique experimentally by using multiple real datasets, *e.g.*, UCR dataset and Yahoo anomaly detection dataset. The experimental evaluation illustrates the efficiency of our solution for kNN MP computation[1]. The results also show how qualitatively the *kNN* MP is useful for knowledge discovery compared to the *1NN* MP. For example, the accuracy of anomaly detection can be improved from 37% with $1NN$ MP to 99% with $10NN$ MP, for one of the benchmarks of the Yahoo dataset.

The rest of the paper is organized as follows. In Section 2, we give the necessary background and definitions. In Section 3, we discuss the related work. Our solution for sequential computations of *kNN* MP is presented in Section 4. The parallel computation of *kNN* MP is presented in Section 5. In Section 6, an extensive experimental evaluation is reported. Finally, Section 7 concludes.

2 Problem Definition

In this section, we give the formal definition of kNN MP, and describe the problem we address. A summary of notations is shown in Table 1.

Definition 1. ***Time series:*** *A time series T is a sequence of real-valued numbers $T = \langle t_1, \ldots, t_n \rangle$ where n is the length of T.*

A subsequence of a time series is defined as follows.

Definition 2. ***Subsequence:*** *Let m be a given integer value such that $1 \leq m \leq n$. A subsequence T_i of a time series T is a continuous sequence of values in T of length m starting from position i. Formally, $T_i = \langle t_i, \ldots, t_{i+m-1} \rangle$ where $1 \leq i \leq n - m + 1$. We call i the start position of T_i.*

[1]The source code and the tested datasets are publicly available at: https://sites.google.com/view/knnmatrixprofile/home

Table 1: Summary of notations

Notation	Description
T	Time series
T_i	Subsequence beginning at index i
n	Time series length
m	Subsequence length
τ	A user given threshold
$Dist(Ti, Tj)$ or $D_{i,j}$	Euclidean distance of subsequences T_i and T_j
$QT_{i,j}$	Dot product between the subsequence T_i and T_j
μ_i	Mean of subsequence T_i
σ_i	Standard deviation (STD) of subsequence T_i
I or I_T	The vector or array which contains the indexes of the time series T
P_T	The vector or array which contains the matrix profile distance of the time series T
$\mathfrak{D}^T$	A dataset which contains several small time series

One of the primary goals of time series analysis is to perform time series subsequence matching. Given a positive real number τ (called *threshold*) and a time series T, two subsequences T_i (beginning at index i) and T_j (beginning at index j) of length m, and a distance function $Dist(T_i, T_j)$ that measures the Euclidean distance between T_i and T_j. If $Dist(T_i, T_j) \leq \tau$, then T_j is called a *matching* subsequence of T_i.

Definition 3. ***Distance profile:*** The distance between a subsequence T_i with all other subsequences of the time series T gives a vector of distances, called distance profile of T_i.

The minimum value of this distance profile represents the closest match or $1NN$ and the top k minimum values of this vector represent the kNN matches of the subsequence T_i. The classical *MP* is defined as a vector of nearest neighbor ($1NN$) distances of all the subsequences of time series T.

Definition 4. ***kNN MP:*** *The kNN MP of a time series T is a vector $P = \langle p_1, \ldots, p_{n-m+1} \rangle$ such that p_i is the distance of the subsequence T_i to its k^{th} nearest neighbor among the subsequences of T.*

The *kNN MP index* is a vector $I = \langle s_1, \ldots, s_{n-m+1} \rangle$ such that s_i is the index of the k^{th} nearest neighbor of the subsequence T_i in the time series T.

Definition 5. ***Motif:*** *A motif pair is defined as a pair of subsequences $\langle T_i, T_j \rangle$ whose distance is less than a user defined threshold τ (i.e., $Dist(T_i, T_j) < \tau$), and their starting positions are at least w elements apart ($|i - j| \geq w$), where w is a user defined threshold which tells that two subsequences in a pair should be w elements apart. If $|i - j| < w$, then the motif pair $\langle T_i, T_j \rangle$ is called a trivial match.*

From the definition of *1NN* motif, we can define the *kNN* motif as follows.

Definition 6. ***kNN Motif:*** *A subsequence T_i is a* kNN motif *if its distance to its k^{th} nearest neighbor (say T_j^k) is less than the user defined threshold τ (i.e. $Dist(T_i, T_j^k) < \tau$) and their starting positions are at least w elements apart.*

Traditionally, a discord is defined as a subsequence (say T_i) whose distance from all other subsequences of the time series is higher than a given threshold ω. Hence, the distance between T_i and it's nearest neighbor is higher than ω. We call this type of discord as 1NN discord. Let us now define the kNN discord.

Definition 7. ***kNN Discord:*** *Let T be a time series, and ω a high distance threshold given by the user. A subsequence $T_i \in T$ is a kNN discord, if the distance of T_i to its k^{th} nearest neighbor is higher than ω.*

Let us now define the problem which we address in this paper. Given a number k, a time series T, and a subsequence length m, our goal is to efficiently compute the kNN MP.

3 Related Work

MP is an efficient solution to the problem of *similarity join*, which can be defined as: given a set of data objects (for our case subsequences), retrieve the nearest neighbors for each object. The solution to this problem can be useful for motif and anomaly discovery from time series in many application domains such as bioinformatics [4], speech processing [5], Seismology [6], etc. Similarity join can be categorized into two principal categories, *1NN similarity join* and *kNN similarity join*. With *kNN similarity join*, for each given object the k nearest neighbors are returned, where k is a positive number given by the user. The *1NN similarity join* is a special case of *kNN similarity join* with $k = 1$. To the best of our knowledge, the MP algorithms in the literature only take into account the 1NN category.

The authors in [7] propose an approach to optimize the calculation of *Euclidean distance* between all subsequences. The idea is to interleave the early abandoning calculations of *Euclidean distance* with the concept of online *z-normalization*. In [8], the authors propose an algorithm based on intelligent caching, reusing computations and the pruning of the search space. By reusing the computations of z-normalized distances for overlapping subsequences, the solution highly saves the computation time and reduces the search space.

In [9] [8], Mueen et al. propose MASS, an efficient algorithm for similarity search in time series. It exploits the consecutive subsequence overlapping property to calculate *Z normalized* distance by *Fast Fourier Transform (FFT)* based convolution. Thanks to the use of *FFT*, in recent years the MASS algorithm has emerged as a significant contribution in subsequence similarity search for many similarity based pattern matching problems such as motif and discord discovery, nearest neighbor matching, etc. [10], [11], [12], [13].

Yeh et al. [2] proposed *MP*, an efficient technique for *similarity join* in time series [14]. The authors use the convolution property of *FFT* and *Inverse FFT* for the fast

calculation of MP by an algorithm called STAMP. Furthermore, an incremental algorithm, called STOMP, is adapted for distance computation of overlapping sequential subsequences in which *MASS* algorithm is used for time series similarity search by computing *z-normalized* euclidean distance between the subsequences. However, the techniques proposed in [2] are mainly designed for *1NN similarity join*.

In [3], the authors introduce an anytime algorithm, named *SCRIMP++* by combining the best features of *STAMP* and *STOMP* for fast MP computation. But this technique is also designed for *1NN* MP. SCAMP [15, 16] is an efficient GPU-based extension of STOMP algorithm for computing 1NN matrix profile using GPUs. It is an improved version of the earlier GPU-STOMP algorithm.

Neighbor profile [17] is a new technique for knowledge discovery from time series. The idea is to take multiple small samples from the set of subsequences and to find the anomalies/motifs inside the samples. In this way, if an abnormal subsequence has some similar anomalies, then probably they will not be in the same sample, and thus the subsequence may be returned as a discord from the sample. This may particularly avoid defiant-gravity behavior. However, the neighbor profile may increase the probability of false anomalies, *i.e.*, those that are not really anomalies but are far from other subsequences in the small chosen sample.

The state-of-the-art MP techniques (*e.g. STAMP* and *STOMP*) compute only $1NN$ MP. To the best of our knowledge, in the literature there is no efficient solution for computing kNN MP. As illustrated in Section 1, this type of MP can be very useful for knowledge discovery from time series. In this paper, we propose an efficient solution for computing it.

4 kNN MP

Here, we discuss our technique for kNN MP computation of a time series T in a sequential computing environment.

4.1 MP Algorithm

Given a time series T, a subsequence size m, and a number k, the goal is to find the k closest matches of all the subsequences T_i in T. Our algorithm for computing kNN MP of the time series T is inspired by the STOMP algorithm [18]. Let us briefly present the idea behind STOMP.

4.1.1 Brief description of STOMP algorithm

The *Scalable Time Series Ordered Search MP (STOMP)* algorithm [11] is a variant of *STAMP* (see Appendix: IV for more details) in which we perform an ordered search (from left to right). To calculate the distance, *STOMP* takes benefit of the common part between two adjacent subsequences which is the same except the fist and last elements. The Z-normalized euclidean distance ($D_{i,j}$) between two time series subsequences T_j and T_i is calculated by using the following Equation 1. The dot product between these two subsequences are mentioned as $QT_{i,j}$.

$$D_{i,j} = \sqrt{2m\left(1 - \frac{QT_{i,j} - m\mu_i\mu_j}{m\sigma_i\sigma_j}\right)} \tag{1}$$

where m is the subsequence length, μ_i and μ_j are the mean of T_i and T_j subsequences respectively, σ_i is the standard deviation of T_i, σ_j is the standard deviation of T_j, and $QT_{i,j}$ is the dot product between two subsequences T_i and T_j. The dot product $QT_{i,j}$ can be computed in $\mathcal{O}(1)$ time when $QT_{i-1,j-1}$ has already been calculated. The term $QT_{i-1,j-1}$ can be decomposed as : $QT_{i-1,j-1} = \sum_{k=0}^{m-1} T_{i-1+k}T_{j-1+k}$ and the term $QT_{i,j}$ can be decomposed as : $QT_{i,j} = \sum_{k=0}^{m-1} T_{i+k}T_{j+k}$ Thus by combining these two terms we can get : $QT_{i,j} = QT_{i-1,j-1} - T_{i-1}T_{j-1} + T_{i+m-1}T_{j+m-1}$. The relationship between $QT_{i,j}$ and $QT_{i-1,j-1}$ indicates that from the distance profile of query subsequence $T_{j-1,m}$, we can compute the distance profile of $T_{j,m}$ in $\mathcal{O}(1)$ time.

4.1.2 1NN Matrix Profile algorithm

The pseudocode of the algorithm for computing 1NN MP is shown in Algorithm 1. Let's consider T be a big time series, which may be obtained by concatenating several small individual time series.The goal is to find the closest match of all the subsequences $T[i]$ with all the remaining subsequences of T and these matches should be separated by w elements from the subsequence in question *i.e.*, $T[i]$. The fast computation of mean and standard deviation (STD) (see Appendix: II for more details) of the time series T is performed in Line 3 of Algorithm 1. Then, the *MASS* algorithm (see Appendix: III for more details) is applied in Line 5 to compute the dot product (QT) (distance D^{ignore} is ignored) between the first subsequence $(subSeq_1)$ and other subsequences of time series T. The arguments of *MASS* are: 1^{st} subsequence $(subSeq_1)$, mean $(\mu_T[1])$ and STD $(\sigma_T[1])$ of $subSeq_1$, complete time series T, mean (μ_T) and STD (σ_T) vector (*i.e.*, mean and STD of all the subsequences) of T.

Then, the algorithm loops through all the subsequences of T (see Line 9) and takes each subsequence $(cutSubSeq)$ in Line 10. Only for the first subsequence (when $i = 1$), it applies *MASS* algorithm to compute the dot product (QT) and distance vector $(Dist_{cutSubSeq})$ between first target subsequence $(cutSubSeq)$ and the remaining subsequences of time series T. For $i > 1$ on-wards, the distance of target subsequence $(cutSubSeq)$ with all the remaining subsequences of T is incrementally calculated by using *IndependentSTOMP* algorithm (see Appendix: IV.1 for more details). The following arguments are passed in *IndependentSTOMP* for the distance calculation: each subsequences $(cutSubSeq)$, the mean $(\mu_T[i])$ and STD $(\sigma_T[i])$ of T, the dot product value $(QT_{initial}[i])$ of very first subsequence $(subSeq_1)$ and i^{th} subsequences $(T[i \text{ to } (i+m-1)])$, already computed dot product vector (QT), mean (μ_T) and STD (σ_T) of T.

Then the matrix profile array (P_T) and index profile array (I_T) are updated based on the distance of each subsequence $(cutSubSeq)$ of T to its nearest neighbor (see Lines $15 - 16$). Finally, these two vectors (P_T) and (I_T) are returned as the results of this algorithm.

Algorithm 1: 1NN-MP(T, m)

Input: Time series T, and the subsequence length m
Output: Matrix profile (P_T) and its associated index (I_T)

$n_T \leftarrow length(T)$ // get the length of time series T
$Idx_T \leftarrow n_T - m + 1$ // get the total number of subsequences in T

$[\mu_T, \sigma_T] \leftarrow ComputeMeanStd(T)$
$subSeq_1 \leftarrow T[1 \text{ to } (1 + m - 1)]$ // get the 1^{st} subsequence from Q
$[QT, D^{ignore}] \leftarrow MASS(subSeq_1, \mu_T[1], \sigma_T[1], T, \mu_T, \sigma_T)$ // apply $MASS$ with arguments as 1^{st} subsequence of T i.e. $subSeq_1$ and remaining subsequences of time series T. See Algorithm 7 in Appendix: III

$QT_{initial} \leftarrow QT$ // keeping a copy of the very first dot product
$P_T \leftarrow$ Initialize this $1D$ vector with inf
$I_T \leftarrow$ Initialize this $1D$ vector with zeros

for $i \leftarrow 1$ ***to*** Idx_T **do**
  $cutSubSeq \leftarrow T[i \text{ to } (i + m - 1)]$ // get target subsequence by chopping T from index i to $(i + m - 1)$
  if $i == 1$ **then**
    $[QT, Dist_{cutTarget}] \leftarrow MASS(cutSubSeq, \mu_T[i], \sigma_T[i], T, \mu_T, \sigma_T)$ // apply $MASS$ with arguments as 1^{st} subsequence of T and complete time series T.
  else
    $[QT, Dist_{cutSubSeq}] \leftarrow IndependentSTOMP(cutSubSeq, \mu_T[i], \sigma_T[i], QT_{initial}[i], T, QT, \mu_T, \sigma_T)$ // calculate distance between a subsequence and all the remaining subsequences in T.
  $P_T \leftarrow computeElementwiseMin(Dist_{cutSubSeq}, P_T)$ // perform element-wise minimum value comparison of P_T and $Dist_{cutSubSeq}$, and maintain the minimum values in P_T
  $I_T \leftarrow i$ // update I_T at the indexes where element-wise minimum operation replaces the previously stored value in P_T with the new value from $Dist_{cutSubSeq}$
return P_T and I_T // return the P_T and I_T array

4.2 Computing kNN Matrix Profile

The MP algorithms in the literature (*e.g.*, *STAMP* and *STOMP* algorithms [2]) are designed to find the best match ($1NN$) of each subsequence. For computing a kNN matrix profile algorithm, the main issue is the management of k nearest neighbors of each subsequence T_i. In fact, efficient methods are needed to update the kNN matches of T_i after computing its distance with another subsequence. In this section, we propose three techniques to find and manage the kNN matches of each subsequence: 1) Sort based; 2) Maximum based; 3) Heap based.

Algorithm 2: SORTING-BASED-KNN-MP (T, m, k)

Input: The time series T, the subsequence length m, user given number k
Output: A MP P_T and associated MP index I_T

....
....
7 $P_T \leftarrow$ Initialize a $2D$ vector of size $\{(k+1) \times Idx_T\}$ with inf
8 $I_T \leftarrow$ Initialize a $2D$ vector of size $\{(k+1) \times Idx_T\}$ with $zeros$
9 **for** $i \leftarrow 2$ ***to*** $Idxs_{conCat}$ **do**
10 $cutSubSeq \leftarrow$ ...
....
....
15 **if** $i <= k$ **then**
16 **for** $p \leftarrow 1$ ***to*** Idx_T **do**
17 $P_T[i,p] \leftarrow Dist_{cutTarget}[p]$
18 $I_T[i,p] \leftarrow i$
else
19 **for** $p \leftarrow 1$ ***to*** Idx_T **do**
20 $P_T[(k+1),p] \leftarrow Dist_{cutTarget}[p]$
21 $I_T[(k+1),p] \leftarrow i$
22 $[sortVals, sortIdxs] \leftarrow Sort(P_T[1 \text{ to } (k+1),p])$
23 **for** $t \leftarrow 1$ ***to*** k **do**
24 $P_T[t,p] \leftarrow sortVals[t,p]$
25 $I_T[t,p] \leftarrow I_T[sortIdxs[t],p]$
26 $P_T^{final} \leftarrow P_T[1 \text{ to } k, 1 \text{ to } Idx_T]$ *// perform element-wise minimum of value comparison of P_T and $Dist_{cutSubSeq}$ then keep the minimum values in P_T*
27 $I_T^{final} \leftarrow I_T[1 \text{ to } k, 1 \text{ to } Idx_T]$ *// update I_T at indexes where element-wise minimum operation replaces the previously stored value in P_T with the new value from $Dist_{cutSubSeq}$*
return P_T^{final} and I_T^{final} *// return the P_T^{final} and I_T^{final} array*

4.2.1 Sort based kNN search

Given a time series T, we need to keep updated the list of the k *nearest neighbors* of each subsequence T_i, when its distance with a subsequence of T is calculated. The idea of the sort based approach is to create a list containing the distance of the current kNN matches and the new computed distance, sort the list and take the first k distances and their corresponding subsequences.

The pseudo-code of the sort based approach is mentioned in Algorithm 2 which is the same as Algorithm 1 until Line 6 (so we avoid to mention it again). In Lines $7-8$, two $2D$ arrays, named as : P_T and I_T are created for storing kNN. Both of these arrays are of size $\{(k+1) \times Idx_T\}$. $(k+1)$ number of rows are needed to keep k nearest neighbors, and the $(k+1)^{th}$ row is needed to temporarily hold newly calculated distance. In Line 9, we loop through all the subsequences of time series (T) and

in Line 10, each subsequence is chopped as usual (same as Algorithm 1)). After that until Line 14, we perform the same operations as in Line $11-14$ of Algorithm 1. The initial k number of distances and index profiles are simply stored in P_T and I_T arrays (see Line $15-18$). From $(k+1)^{th}$ subsequence on-wards (the *else* portion in Line $19-25$), the newly calculated distance profile is saved at $k+1^{th}$ row (Line 20) and consequently the target subsequence index $\boldsymbol{i}$ is stored in $k+1^{th}$ row (Line 21). Then the distances stored from index 1 to $(k+1)$ are sorted in ascending order and from this sorting operation, we will get the sorted distance values ($sortVals$) and their corresponding indexes ($sortIdxs$) (see Line 22). After that the top k sorted values are updated in P_T matrix (Line 24) and corresponding stored indexes are also updated (Line 25) with the help of sorted indexes ($sortIdxs$). This process is repeated iteratively for all the subsequences, and finally the MP P_T and the MP index I_T from the index 1 to k are returned (as P_T^{final} and I_T^{final}) as the output of this algorithm.

The average time complexity of sorting k elements is $\mathcal{O}(k \log k)$ and we need to perform this O$((n-m)^2)$ times, where n is the length of the time series T, and m the subsequence length. Hence, the total complexity is O$((n-m)^2 \times k \log k)$. To improve the computational time, in the next subsection we propose to replace *sorting* by finding *maximum* of top k distance values for each subsequences.

Algorithm 3: MAX-BASED-KNN-MP (T, m, k)

```
....
....
else
19    for p ← 1 to Idx_T do
20        P_T[(k+1), p] ← Dist_cutTarget[p]
21        I_T[(k+1), p] ← i
22        [maxVal, maxIdx] ← FindMax(P_T[1 to k, p])
23        if (P_T[(k+1), p] < maxVal) then
24            P_T[maxIdx, p] ← P_T[(k+1), p]
25            I_T[maxIdx, p] ← I_T[(k+1), p]
26  P_T^final ← P_T[1 to k, 1 to Idx_T]
27  I_T^final ← I_T[1 to k, 1 to Idx_T]
....
return P_T and I_T // return the P_T and I_T array
```

4.2.2 Maximum based kNN search

In this method, instead of *sorting*, we use the *maximum* distance of the subsequence T_i to its kNN matches (*i.e.*, the distance to its k nearest neighbor), and then compare this maximum value with the newly computed distance. The time complexity of finding maximum in a list of k elements is $\mathcal{O}(k)$, which is already less than the time complexity of the sort based algorithm, *i.e.*, $\mathcal{O}(k \log k)$.

The pseudo code of this approach is mentioned in Algorithm 3. The *Else* portion of Algorithm 2 (Line $19-25$), needs to be replaced by the pseudo code of Algorithm 3 (Line $19-25$). As usual, the newly calculated distance profile and subsequence index

are stored at $(k+1)^{th}$ row of P_T and I_T matrix (Line $20-21$). Then, we find the maximum value of top k elements and corresponding index from P_T matrix (Line 22). Now this maximum value is compared with the newly arrived value, which is temporarily kept at $(k+1)^{th}$ index (Line 23). If the newly arrived value is less than the existing maximum value, then the old maximum value is replaced by the new value ($P_T[maxIdx, p]$) (Line 24) and it's index is stored in I_T matrix (Line 25). This process is repeated for all the subsequences and finally the results P_T^{final} and I_T^{final} are returned as output of the algorithm.

4.2.3 Heap based kNN search

Finding the maximum value in a vector of size k in the classical manner has a time complexity of $\mathcal{O}(k)$. Here, we propose to find the maximum of a vector by using the heap based priority queue whose time complexity is $\mathcal{O}\log(k)$. At first, we need to organize the kNN matches into a heap structure, thus the first element in the heap will contain the maximum value of the array.

Algorithm 4: HEAPMAX-BASED-KNN-MP (T, m, k)

```
....
....
else
19    for p ← 1 to Idx_T do
20        P_T[(k+1), p] ← Dist_cutTarget[p]
21        I_T[(k+1), p] ← i
22    if i == (k+1) then
23        for p ← 1 to Idx_T do
24            [P_T[1 to k, p], heapSortIdxs] ← BuildMaxHeap(P_T[1 to
                k, p], k)
25            I_T[1 to k, p] = I_T[heapSortIdxs[1 to k], p]
26    for p ← 1 to Idx_T do
27        if (P_T[(k+1), p] < P_T[(1), p]) then
28            P_T[1, p] ← P_T[(k+1), p]
29            I_T[1, p] ← I_T[(k+1), p]
30            [P_T[1 to k, p], heapSortIdxs] ← BuildMaxHeap(P_T[1 to
                k, p], k)
31            I_T[1 to k, p] = I_T[heapSortIdxs[1 to k], p]
32  P_T^final ← P_T[1 to k, 1 to Idx_T]
33  I_T^final ← I_T[1 to k, 1 to Idx_T]
....
return P_T and I_T // return the P_T and I_T array
```

The pseudo-code of the heap based approach is shown in Algorithm 4. Until Line 18, the algorithm remains the same as Algorithm 2. The *Else* portion of Algorithm 2 (Line 19−25), needs to be replaced by the pseudo code of Algorithm 4 (Line 19−33). In Line 22, we verify whether $i == (k+1)$, *i.e.* when we are handling $(k+1)^{th}$ subsequence of target, the elements of P_T array are organized for the first time in the structure of heap based priority queue. So in Line 23, we loop through each subsequence, and for each of such subsequences we organize the elements (k elements) of each column in a heap based priority queue. The heap structure is updated in P_T (Line 24). Thanks to this operation, we will have the maximum value at the first row of P_T matrix. We loop through all subsequences (Line 26) and compare the maximum value with the newly arrived value at $(k+1)^{th}$ index. If this newly arrived value is less than the existing maximum value (stored in first row) then we replace the value in the first row with the new value. Then, we apply the restructuring operation on the heap based priority queue to put the maximum value out of top k values at the first row. This process is repeated for all the subsequences in order to produce the final k distances and corresponding indexes in P_T^{final} and I_T^{final} matrices.

5 Multi-core based parallel computing

In this section, we propose an approach to perform the parallel computation of kNN MP by exploiting multiple cores. We consider the situation where several small time series are concatenated to generate one big time series. Let's say there are $n_{\mathfrak{D}^T}$ number of individual time series in the time series dataset $\mathfrak{D}^T$ i.e. $\{\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3, ..., \mathcal{T}_{n_{\mathfrak{D}^T}}\} \in \mathfrak{D}^T$. By concatenating all these time series, we can obtain a big time series T. The objective is to compute *kNN* MP of T, such that we should be able to identify the matches, coming from which individual time series along with the index of the match in the time series.

Let $\mathfrak{n}_{cores}$ be the total number of available cores, then the idea is to divide the big time series T into $\mathfrak{g}$ portions (*i.e.*, $\mathcal{G}_1$, $\mathcal{G}_2$, ...,$\mathcal{G}_\mathfrak{g}$) and to give each portion to one individual core to process. The number of groups are determined by the availability of total number of cores of the processor (say g). An example of kNN MP computation by using multiple cores is depicted in Fig. 2. As seen, the time series T is equally divided into 6 groups, and each group is processed by a separate core. The pseudo code of this technique is shown in Algorithm 5.

The core algorithm for performing parallel computation of *kNN* MP is presented in Algorithm 5. The concatenation of $n_{\mathfrak{D}^T}$ number of small time series of different lengths from the time series database $\mathfrak{D}^T$ is done by using *concatenate_TimeSeries()* function in Line 1. After concatenating, we obtain a big time series[2] T. The fast computation of mean and standard deviation of all the subsequences in T is computed in Line 2.

Then in Line 3, we obtain the number of available cores. If the remainder after division between the total number of individual time series, *i.e.*, $n_{\mathfrak{D}^T}$ and total number of available cores *i.e.* n_{cores} is zero then the variable $\mathcal{Y}$ (Line 8) will simply hold

[2]this case is only applicable when we don't have a single big time series, but several small time series in the database. Hence, a big time series is formed by concatenating these small time series

Function `concatenate_TimeSeries` $(\mathfrak{D}^T)$**:**

$n_{\mathfrak{D}^T} \leftarrow length(\mathfrak{D}^T)$ *// count the total number of time series in the data base* $\mathfrak{D}^T$

$Len^1 \leftarrow lenth(\mathfrak{D}^T[1])$ *// get the length of first time series from database* $\mathfrak{D}^T$

$T \leftarrow \mathfrak{D}^T[1]$ *// create a new vector* T *and initialize it by copying the first time series* $\mathfrak{D}^T[1]$ *in it*

$Info_{conCat} \leftarrow [1, Len^1, `file1.csv`]$ *// when an individual time series is merged in* T*, keep the indexes at where it starts and ends in concatenated time series* T

for $iSeries \leftarrow 2$ ***to*** $n_{\mathfrak{D}^T}$ **do**

  $T \leftarrow [T, \mathfrak{D}^T[iSeries]]$ *// keep concatenating individual time series from* $\mathfrak{D}^T$

  $Info_{conCat} \leftarrow [startIdx, endIdx, fileName]$ *// store the start, end indexes and the file name after concatenating an individual time series in* T

return $n_{\mathfrak{D}^T}$; $Info_{conCat}$ and T

the indexes of first and last values of T assigned to each group. Otherwise, if the remainder is more than zero then the number of individual time series in each group is calculated by subtracting $\mathfrak{r}$ from $n_{\mathfrak{D}^T}$ and then dividing it by n_{cores} (Line 11). The indexes of first and last values of all the groups except last group are stored in $\mathcal{Y}$ (Line 13). The remaining $\mathfrak{r}$ number of time series, belongs to the last group are stored at the last cell of $\mathcal{Y}$ in Line 14.

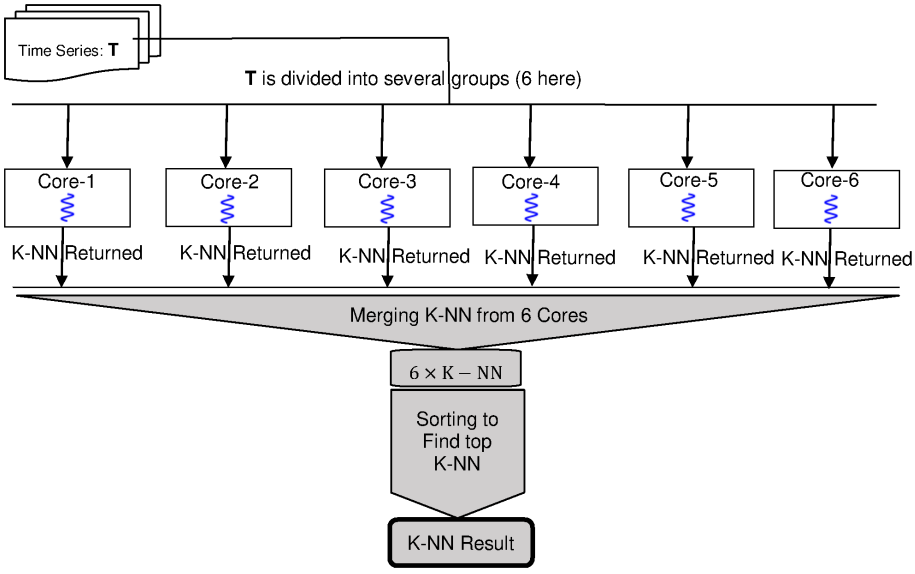


Figure 2: The proposed architecture of multi core based parallel processing.

By using the variable $Info_{conCat}$ (which keeps the indexes and length of each individual time series) and the variable $\mathcal{Y}$ (which keeps the indexes of first and last

Algorithm 5: kNN-Parallel($\mathfrak{D}^T$, m)

Input: The target time series data base ($\mathfrak{D}^T$, m)
Output: A MP (P_T) and associated MP index (I_T)

$[n_{\mathfrak{D}^T},\ Info_{conCat},\ T] \leftarrow concatenate\_TimeSeries(\mathfrak{D}^T)$
$[\mu_T, \sigma_T] \leftarrow ComputeMeanStd(T)$ // see Equation 2 in Appendix: II
$n_{cores} \leftarrow cluster.numWorkers$ // get the number of workers/cores available
$\mathcal{Y} \leftarrow (n_{cores} \times 2)\ array$ // these are 2D matrix, initialized with zeros
if $(n_{\mathfrak{D}^T}\ \%\ n_{cores} == 0)$ **then**
    $\mathfrak{s} \leftarrow 1;\ \mathfrak{g} \leftarrow n_{\mathfrak{D}^T}/n_{cores}$ // number of individual time series in each group is calculated in $\mathfrak{g}$
    for $iClus \leftarrow 1$ ***to*** n_{cores} **do**
        $\mathcal{Y}[iClus][1] \leftarrow \mathfrak{s};\ \mathcal{Y}[iClus][2] \leftarrow \mathfrak{s} + \mathfrak{g} - 1;\ \mathfrak{s} = \mathfrak{s} + \mathfrak{g}$ // the indexes of first and last individual time series is stored in $\mathcal{Y}$
else
    if $(n_{\mathfrak{D}^T}\ \%\ n_{cores} > 0)$ **then**
        $\mathfrak{s} \leftarrow 1;\ \mathfrak{r} \leftarrow n_{\mathfrak{D}^T}/n_{cores};\ \mathfrak{g} \leftarrow (n_{\mathfrak{D}^T} - \mathfrak{r})/n_{cores}$
        for $iClus \leftarrow 1$ ***to*** $n_{cores} - 1$ **do**
            $\mathcal{Y}[iClus][1] \leftarrow \mathfrak{s};\ \mathcal{Y}[iClus][2] \leftarrow \mathfrak{s} + \mathfrak{g} - 1;\ \mathfrak{s} = \mathfrak{s} + \mathfrak{g}$
        $\mathcal{Y}[n_{cores}][1] \leftarrow \mathfrak{s};\ \mathcal{Y}[n_{cores}][2] \leftarrow n_{\mathfrak{D}^T};$ // the indexes of start and last individual time series of last group are saved in the last cell of $\mathcal{Y}$
$P_T^{All}, I_T^{All} \leftarrow (n_{cores} \times 1)\ array$ // these are 1D vectors each cell contains
/* // following for loop is run in parallel i.e. the contents inside the loop are given to each core */
for $iCore \leftarrow 1$ ***to*** n_{cores} **do**
    $\mathcal{ST} = Info_{conCat}[\ \mathcal{Y}[iCore][1]\ ]$ // get the starting index of the part/portion of full time series, handled by this core
    $\mathcal{ED} = Info_{conCat}[\ \mathcal{Y}[iCore][2]\ ]$ // get the end index of the part/portion of full time series, handled by this core
    $[P_T^{core}, I_T^{core}] \leftarrow knn\_MP\_Parallel(T,\ \mu_T,\ \sigma_T,\ \mathcal{ST},\ \mathcal{ED},\ m)$
    $P_T^{All}[iCore] \leftarrow P_T^{core};\ I_T^{All}[iCore] \leftarrow I_T^{core}$
$P_T^{concat} \leftarrow P_T^{All}[1];\ I_T^{concat} \leftarrow I_T^{All}[1]$ // initialized with the first matrices i.e. $P_T^{All}[1]$ and $I_T^{All}[1]$
$nLen \leftarrow n_{cores} \times kNN$
for $i \leftarrow 2$ ***to*** n_{cores} **do**
    $P_T^{concat} \leftarrow [P_T^{concat}, P_T^{All}[i]];\ \ I_T^{concat} \leftarrow [I_T^{concat}, I_T^{All}[i]]$ // concatenating the matrices
$P_{sort}[1:nLen][1:Idx_{conCat}], indx_{sort}[1:nLen][1:Idx_{conCat}] \leftarrow sortColWise\ (\ P_T^{concat}[1:nLen][1:Idx_{conCat}]\ )$ // sort the concatenated matrix P_T^{concat} column wise
$I_{sort}[1:nLen][1:Idx_{conCat}] \leftarrow I_T^{concat}(\ indx_{sort}[1:nLen][1:Idx_{conCat}]\ )$ // using the sorted index i.e. $indxsort$ rearrange I_T^{concat}
return $P_T^{Final} \leftarrow P_{sort}[1:kNN][1:Idx_{conCat}]$ and $I_T^{Final} \leftarrow I_{sort}[1:kNN][1:Idx_{conCat}]$

Function knn_MP_Parallel (T, μ_T, σ_T, $\mathfrak{st}$, $\mathfrak{ed}$, m) **:**

4 $subSeq_1 \leftarrow T[\mathfrak{st}$ to $(\mathfrak{st} + m - 1)]$ ▷ *get the* 1^{st} *subsequence*

5 $[QT, D^{ignore}] \leftarrow MASS(subSeq_1, \mu_T[\mathfrak{st}], \sigma_T[\mathfrak{st}], T, \mu_T, \sigma_T)$

.....

9 **for** $i \leftarrow \mathfrak{st}$ ***to*** $\mathfrak{ed} - m + 1$ **do**

10 $cutSubSeq \leftarrow T[i$ to $(i + m - 1)]$ ▷ *get target subsequence by chopping T from index i to* $(i + m - 1)$

.....

17 **return** P_T and I_T ▷ *return the* P_T *and* I_T *array*

Figure 3: The start and end indexes of each portions/parts of time series T, after dividing it into $\mathfrak{g}$ (= 6 here) number of groups (please see the online color version for better visibility).

individual time series of all the groups), we can obtain start and end indexes of each parts/portions of time series T, which is divided into $\mathfrak{g}$ parts (Line 17 − 18). An example is shown in Fig. 3, where the start (St) and end (Ed) indexes of each part are visually depicted. In every core, the local kNN matches are calculated in parallel for all the subsequence of T by using the *"knn_MP_Parallel ()"* function (Line 19). The idea is to calculate kNN matches of all the subsequences of T, where the matches come from one part/portion of the time series T, handled by an individual core. Then, the partial kNN MPs from different cores are merged and the global MP is produced. The MP P_T^{core} and index profile I_T^{core}, obtained from each core are saved in P_T^{All} and I_T^{All} respectively (Line 20). These matrices are concatenated in P_T^{concat} and I_T^{concat} respectively (Line 21). The concatenated MP P_T^{concat} is then sorted column wise to obtain the sorted MP P_{sort} and sorted indexes $indx_{sort}$. They are then used to obtain the sorted index profile I_{sort} (Line 26). Finally, the top kNN rows of P_{sort} and I_{sort} are returned in P_T^{Final} and I_T^{Final} respectively in Line 27.

The function *"knn_MP_Parallel ()"* is similar to Algorithm 4, except that *"knn_MP_Parallel ()"* takes a portion of T that starts from $\mathfrak{st}$ and ends to $\mathfrak{ed}$, and for each subsequence of this portion finds the kNN matches among all subsequences of T.

6 Experimental Evaluation

In this section, we evaluate the performance of our proposed kNN MP technique and illustrate its utility for motif and anomaly detection from time series.

6.1 Setup

In our experiments, we have used several datasets. The first dataset is *chemometric data*, representing protein rate measured on 10 different products. The second dataset is *accelerometer data* obtained by attaching accelerometer at the neck of 13 sheep. The third dataset is *seismic data* and the fourth dataset is a synthetic *random walk* dataset. We have also used some datasets from the UCR time series data mining archive [19] (see the list in Table 2).

The multi-core experiments were performed on a computational server having 36 physical cores. The other experiments were performed on an off-the-shelf computer, having *Intel(R) Core(TM) i7-8850H CPU @ 2.60 GHz* processor with 32 GB RAM[3].

6.2 Utility Experiments

We illustrate the utility of kNN MP for knowledge discovery in three case studies including chemometric, accelerometer, UCR archive, and Yahoo anomaly detection dataset.

6.2.1 Case study: chemometric data

The experiments were performed with the dataset of 4075 spectrum, each having 680 dimensions. These spectrum represents the protein rates, measured on 10 different products: rapeseed (CLZ), corn gluten (CNG), sun flower seed (SFG), grass silage (EHH), full fat soya (FFS), wheat (FRG), sun flower seed (SFG), animal feed (ANF), soyameal (TTS), maïs (PEE), milk powder and whey (MPW). The complete data can be imagined as a matrix of 4075×680, where each row represents a time series of 680 elements. To build the kNN MP on a big time series, we concatenated the 4075 individual time series together and applied our kNN MP algorithm on it. We created the kNN MP by considering the subsequence length $m = 40$. Fig. 4 shows the $1NN$ vs $2NN$ and $4NN$ MPs for the chemometric time series. Note that we have only plotted a small part (*i.e.*, 640 elements) of the whole MP. A low threshold is defined to obtain the motifs (shown as dotted black line at the bottom of the curve).If we consider only the curve for $1NN$ (blue color) then there are several subsequences, which can be taken as motifs. We consider these motifs as *weak motifs*. We can detect the strong motifs by checking their existence in *kNN* MP. If a subsequence appears as motif in kNN MP, then it has at least k similar subsequences in the time series.

In Fig. 4, we see some *strong motifs* (marked as green circles) by considering $k = 2$ (top image) and $k = 4$ (bottom image). The motifs that appear in $4NN$ MP are stronger than those detected by using $2NN$ MP because they are repeated in higher values of k. There are some *weak motifs* (encircled by red color) that are shown in Fig 4 which appear only in *1NN* MP but not in either $2NN$ or in $4NN$ MPs. Another nice illustration is also shown in Fig. 5 where we can see the cases of *strong motifs* and *weak motifs* by considering the $1NN$ vs $2NN$ MP and $4NN$ MP.

The discords are the subsequences whose distance with other subsequences is high. Let's consider Fig. 6 that shows the sorted distances of the matches of three individual subsequences. The distances are shown along $Y-axis$ and the subsequences

[3]We have shared the code, datasets and instructions in : https://github.com/tanmayGIT/kNN_Matrix_Profile

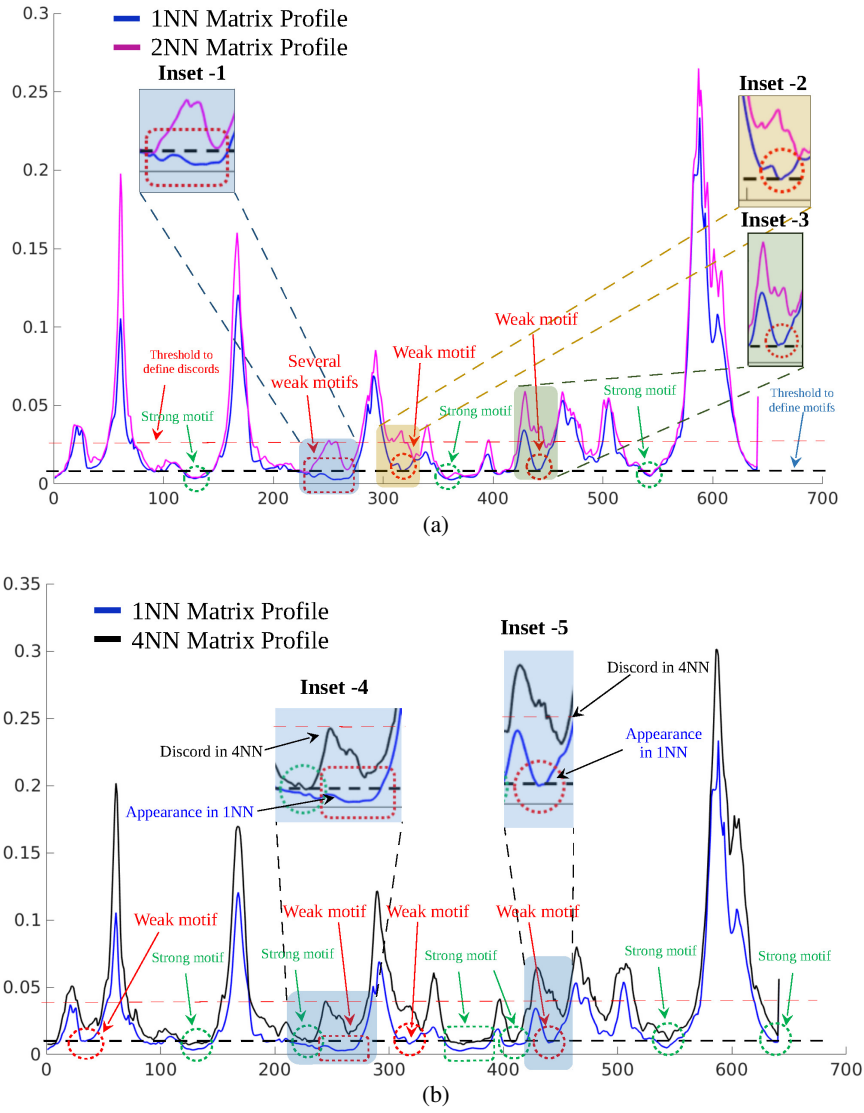


Figure 4: The extracted motifs and discords are illustrated in the plot : (a) The $1NN$ vs $2NN$ MP and (b) $1NN$ vs $4NN$ MP of a part of the time series is plotted (please see the online color version for better visibility).

with whom the distances are computed are shown along $X - axis$. The distance values plotted in the first curve (shown in green) have high values. The points in the third curve (shown by pink color in Fig. 6) represent the sorted distance values of a subsequence. They are mostly below the defined outlier (discord) threshold. So, if we consider even $4NN$ (or even more) MP then in no way this particular subsequence

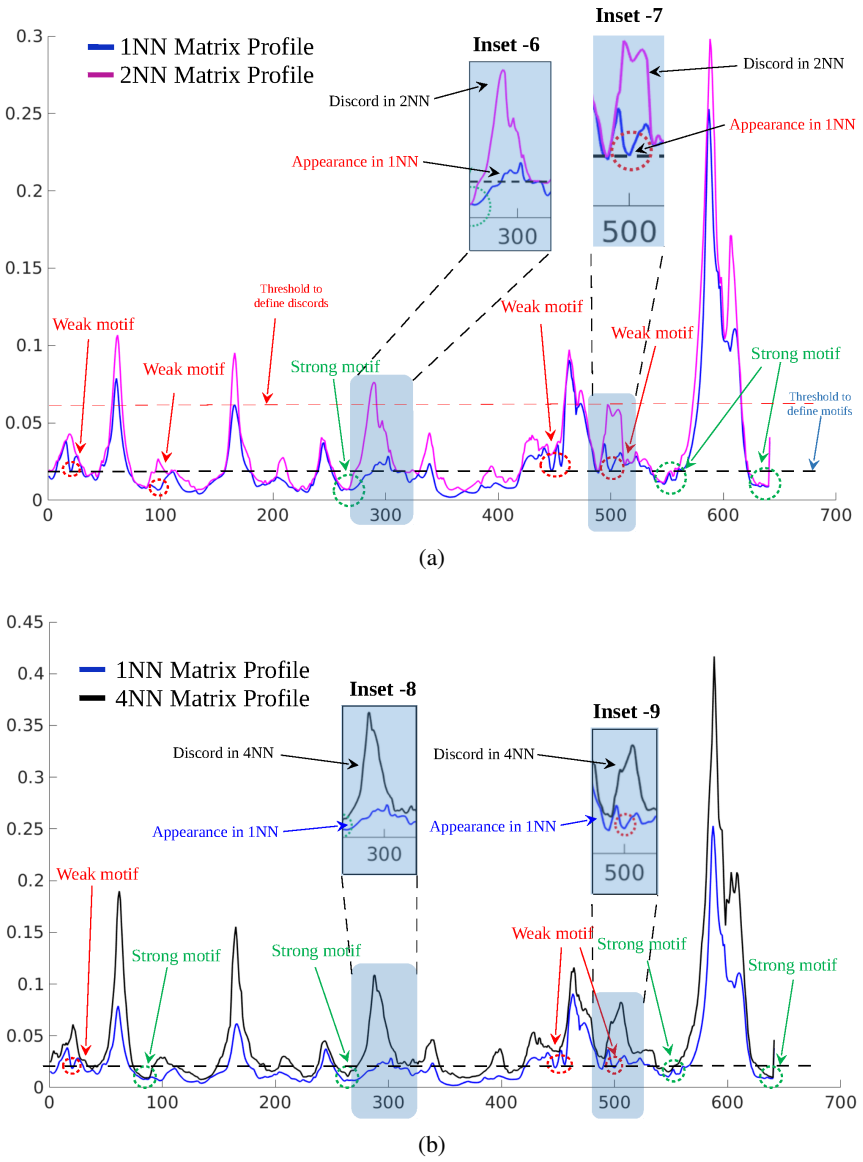


Figure 5: The extracted motifs and discords are illustrated in the plot : (a) The $1NN$ vs $2NN$ MP and (b) $1NN$ vs $4NN$ MP of a part of the time series is plotted (please see the online color version for better visibility).

will be detected as outlier (this is normal as this subsequence has close matches). But in the case of third curve, the top two distance values are less than the defined discord threshold, and the other distance values are more than the threshold. So, from the nature of the curve it can be depicted that this particular subsequence has two

close matches, but it is highly different than all other remaining subsequences. Thus, if we consider $1NN$ and $2NN$ then this particular subsequence will not be detected as outlier. But, if we consider $3NN$, $4NN$ and more then it will be detected as an outlier. Logically, this subsequence should be detected as outlier as it has only two very close neighbors (which can be outliers), and all of it's other neighbors are very different.

This scenario is confirmed by kNN MP in Fig. 7 (top). The 1^{st} subsequence has many matches shown as pink color (Fig. 7 top), this is why $1NN$, $2NN$, $3NN$ MPs (Fig. 7 bottom) show lower value at the index of this subsequence. On the other hand, the 3^{rd} subsequence (shown in green color) has no matches, hence the $1NN$, $2NN$, $3NN$ MPs show high values for the 3^{rd} subsequence. But, for the case of 2^{nd} subsequence, it has a close match (shown in red color). Hence these two subsequences (which are marked in red color) would closely match with each other. Accordingly the $1NN$ MP (shown in Fig. 7 bottom) shows a very low value for these subsequences (follow the pink colored curve).

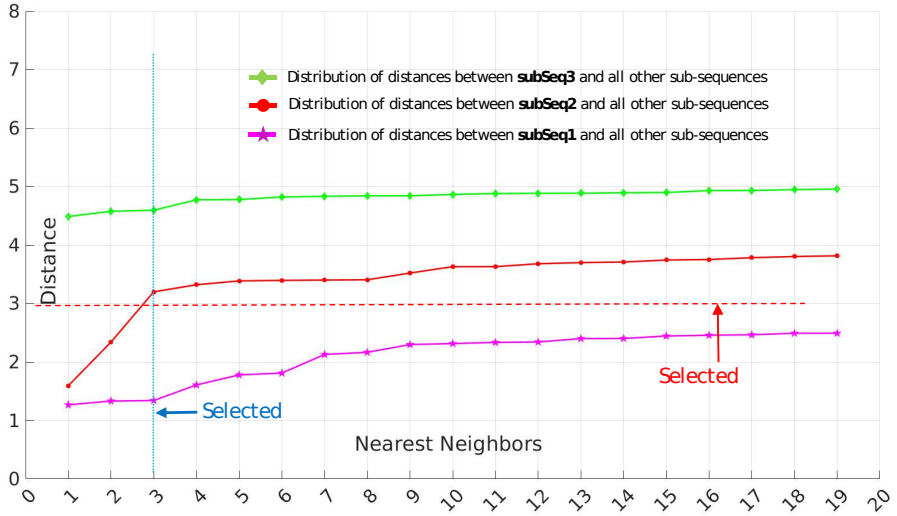


Figure 6: One special case of outliers detection (please see the online color version for better visibility)

It can be seen from Fig. 7 (top) that these two red colored subsequences should be considered as outliers as they are far from all other subsequences. In fact, they are $3NN$ discords, and their detection is possible only if we compute kNN MP, for $k \geq 3$. If we calculate $2NN$ MP then each of these red colored subsequences would have the blue colored subsequence as their $2nd$ nearest neighbors (blue colored subsequence is slightly different than the red ones). Hence to detect them as outliers, we need to calculate $3NN$ MP (shown by blue color in Fig. 7 (bottom)).

Now let's again consider the example of $1NN$ vs $2NN$ and $1NN$ vs $4NN$ MPs of *chemometric dataset*. To find the discords, a relatively high threshold is taken

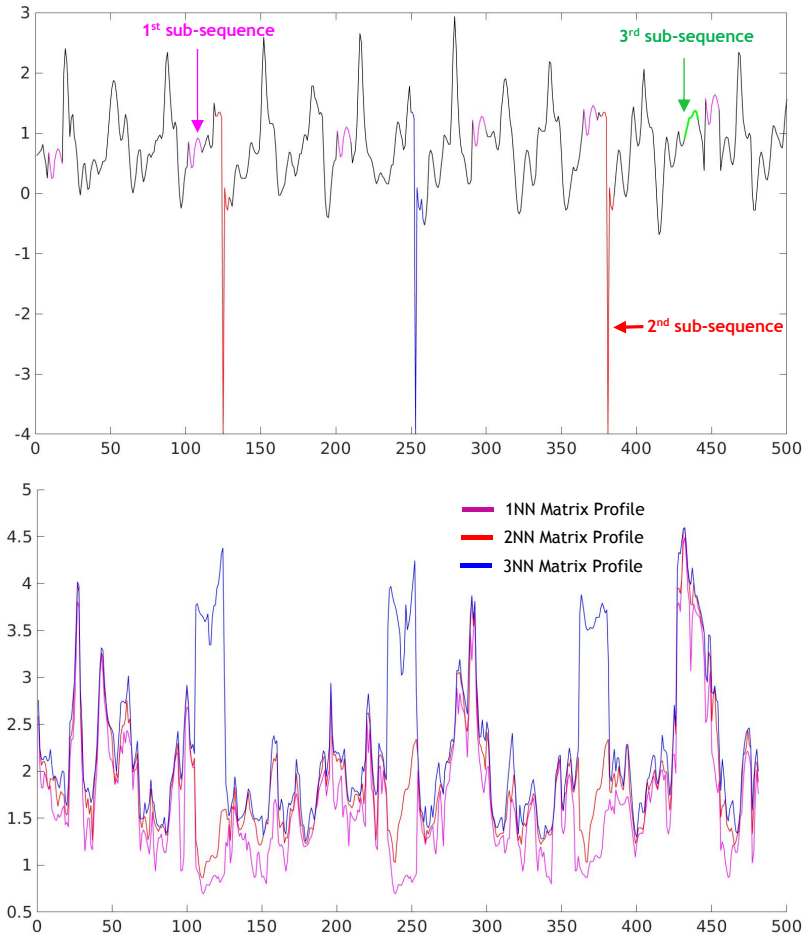


Figure 7: Special case of the outliers presence is depicted by considering a toy time series (top). The 1NN, 2NN and 3NN MPs are shown for the time series (bottom) (please see the online color version for better visibility)

(shown as dotted red line in Fig. 4). If we consider this threshold then the discords which are detected in $1NN$, should also be repeated (exactly at the same index) in $2NN$ as well as in $4NN$ MP. In Fig. 4 (top) there are several instances shown in Inset-1, Inset-2 and Inset-3, where we have zoomed over some portions of $1NN$ and $2NN$ MPs. It can be seen in these instances that the distance value is low (sometimes downward) in $1NN$ MP, but at these locations, the distance value is high in $2NN$ MP. Similar characteristics are also observed in Inset-4, Inset-5 for the case of $1NN$ vs $4NN$ MPs in Fig. 4 (bottom image).

6.2.2 kNN similarity search : case study on accelerometer data

This real world dataset corresponds to more than 8000 time series which have been measured by attaching accelerometer at the neck of 13 sheep. Acelerometers captured 3-axial acceleration at a constant rate of 100Hz. Each of the three axial acceleration gives a different information for the zoologist, but for the simplicity and to show the interest of proposed method, here we only consider X axis data. The accelerometer data are manually labeled into one of six activities: STANDING-GRAZING, STANDING-EATING BRUSH, STANDING-RUMINATING, WALKING, RUNNING, STANDING IMMOBILE. The sensor signals were pre-processed and for each activity of interest, a time series of 5 seconds (500 elements per time series) were constituted. By this manner, a dataset with 8532 time series is obtained where each of these time series is manually labeled.

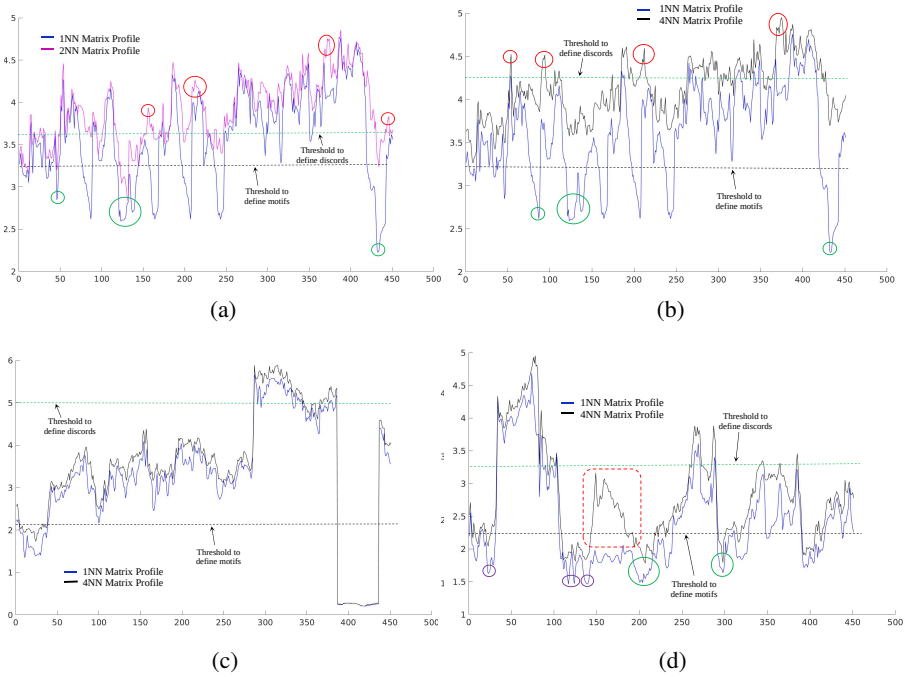


Figure 8: (a) The 1NN and 2NN MPs are plotted for a specific portion of the time series, having the label as: WALKING. (b) The 1NN and 4NN MPs are plotted of the same portion of the time series. (c) The 1NN and 4NN MPs (which follow very similar trajectories) are plotted for another portion of the time series, having labeled as WALKING. (d) The 1NN and 4NN MPs are plotted for a randomly chosen portion of the time series, having labeled as: RUNNING. (please see the online color version for better visibility)

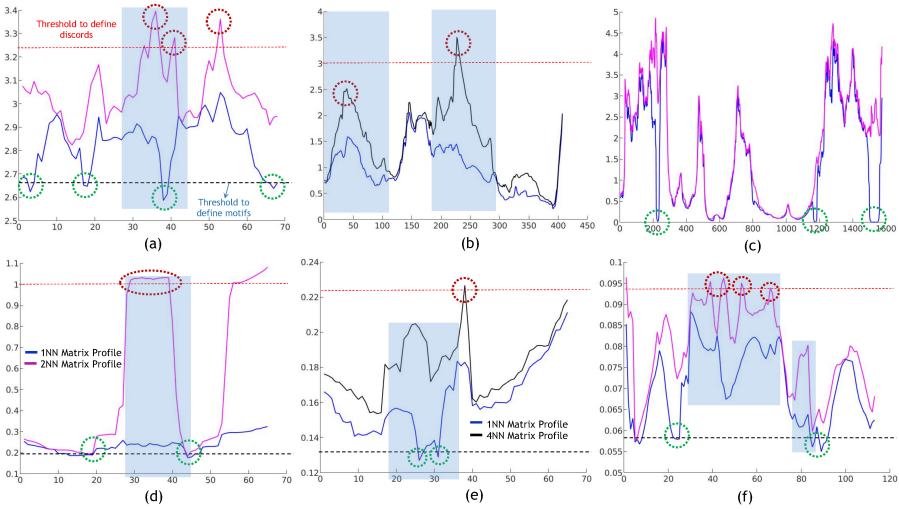


Figure 9: The results of UCR dataset : **(a)** FacesUCR dataset: the motifs and discords are shown by green and red circles respectively; **(b)** Beef dataset: presence of motifs in $4NN$ MP are not supported by $1NN$ MP; **(c)** CinCECGTorso dataset: the highlighted zone shows the contrary nature of $1NN$ vs $2NN$ MPs; **(d)** BME dataset: the highlighted zone shows the contrary nature of $1NN$ vs $2NN$ MPs; **(e)** BME dataset : some (weak) motifs in $1NN$ MP are not present in $4NN$ MPs; (f) Adiac dataset: the highlighted zones show the contrast between $1NN$ vs $2NN$ MPs where the weak motifs detected in $1NN$ MP are not present in $2NN$ MP and the discords detected in $2NN$ MP are not present in $1NN$ MP.

Here we create a big time series by concatenating all the 8532 time series and the subsequence length (m) is taken as 50. The total number of subsequences obtained from the concatenated long time series is $(500 \times 8531) - 50 + 1 = 42,65,451$. An interesting part of the kNN MP for the case of $1NN$ vs $2NN$ MP and $1NN$ vs $4NN$ MP is shown in Fig. 8a and Fig. 8b. It can be seen from these figures that at several locations the distance values in $1NN$ MP are less than the defined threshold for motif, but not in the $2NN$ MP (see the green circles). So these low points can not be considered as strong motifs. For the case of detecting outliers, several points are marked in Fig. 8a and Fig. 8b, where some outliers can be detected only in $2NN$ or $4NN$ (see the red circled points in Fig. 8a and Fig. 8b).

In many cases, the $1NN$ and $4NN$ MPs follow almost the same trajectory and such an example is shown in Fig. 8c. Another portion of the complete MP is shown in Fig. 8d in which we illustrate the $1NN$ and $4NN$ MPs, and several interesting cases (by green and violet encircles). The points marked by green circles are *strong motifs* where both of the $1NN$ and $4NN$ MPs satisfy the defined threshold, whereas the points marked by violet circles are *weak motifs*. The region marked by dotted red rectangle shows an interesting situation where the $4NN$ MP shows a completely different trajectory (upward) than $1NN$ MP (downward). This can mainly happen

when a particular subsequence has the 1^{st} nearest neighbor with whom it has small distance, but its distance with other neighbors is high (see Fig. 6). Such subsequences are usually anomalies.

6.2.3 Case study: UCR repository

Here we show some interesting results for kNN MP by using datasets from the UCR time series archive [19] (see the list of datasets in Table 2). To create a single big time series from each dataset, we have sequentially concatenated the individual time series from training and testing set. Fig. 9 shows a portion of the generated MP for different datasets. As seen in Fig. 9 (a), (b), (d), (e), (f), at some places the $1NN$ MP shows different trajectory than kNN (*i.e.*, $2NN$ and $4NN$) MPs where the detected weak motifs (shown in dotted green circle) in $1NN$ MP are absent in kNN MP. There are also discords (shown in dotted red circle) detected in kNN MPs, but absent in $1NN$ MP.

From these plots, we can visualize that the curves for $1NN$ MP and $2NN$ or $4NN$ MPs don't follow the same trajectory. Hence, the detection of motifs and discords from $1NN$ MP could be wrongly validated if we don't verify their presence in kNN (*i.e.*, $2NN...kNN$) MPs. From Fig. 9 (c) also, we observe that there are detected motifs present in only $1NN$ MP, but not in the other MPs. Probably these motifs are resulted due to the presence of noise and should be considered with precaution. Hence, from these experiments with UCR datasets, we can clearly visualize the usefulness of kNN MP over $1NN$ MP for the detection followed by confirmation of motifs and discords.

Table 2: Dataset details from UCR archive

Dataset Name	Size of training set	Size of testing set	Time series length
FacesUCR	200	2050	131
Beef	30	30	471
CinCECGTorso	40	1380	1639
BME	30	150	128
Adiac	390	391	176

6.2.4 Case study: Yahoo anomaly detection dataset

In this section, we show some interesting illustrations to depict the usefulness of kNN MP, using the Yahoo time series dataset that includes labeled anomalies[20]. This dataset contains several files (around 370), among them one part is based on real data (around 95), based on production traffic in some Yahoo services, whereas the other part contains synthetic (i.e., simulated) data. The anomalies in the simulated data were algorithmically generated, and those in the real-traffic data were manually labeled by Yahoo experts. The dataset is divided in 4 benchmarks, which are named as: "A1Benchmark-Real", "A2Benchmark-Synthetic", "A3Benchmark-Synthetic" and "A4Benchmark-Synthetic".

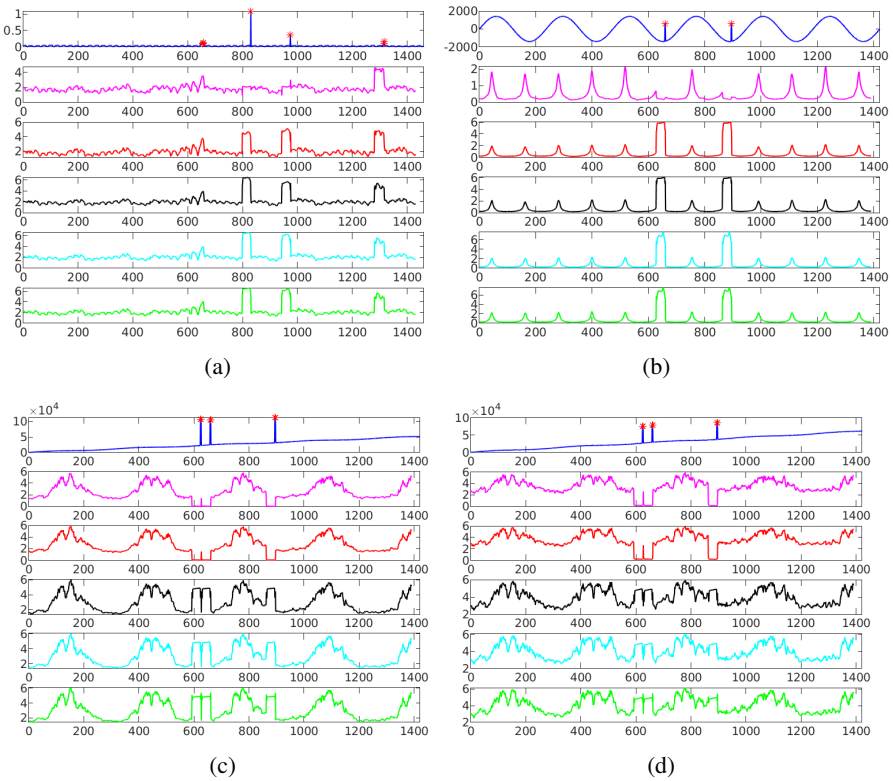


Figure 10: The usefulness of the kNN MP technique is shown using Yahoo dataset. In each figure, the time series is plotted at the top followed by $1NN$, $2NN$, $3NN$, $4NN$, and $5NN$ MPs, plotted subsequently (the subsequence length (m) is taken as 32). The outlier elements/points, obtained from the available ground truth of the dataset are marked by the red star at the topmost plot of each figure (please see the online color version for better visibility).

In Fig. 10a to Fig. 10d, we depicted some interesting examples of anomaly detection by *kNN* MPs. In each figure, the time series is plotted at the top, followed by $1NN$, $2NN$, $3NN$, $4NN$, and $5NN$ MPs. We see that the *kNN* MPs find more relevant anomalies than $1NN$ MP. In Fig. 10a, for instance, the *1NN* MP (see the second plot) is not able to detect the second and third anomalies which are marked by red-colored star on the time series plot, wheeras the *2NN*, *3NN*, *4NN* and *5NN* MPs (see the third until sixth plots in Fig. 10a) are able to detect them. The plausible reason is that these anomalies which are not detected in *1NN* MP have similar subsequences (anomalies) in the time series, thus the distance values of these anomalies to their $1NN$ are not high enough (see Fig. 6, Fig. 7 and it's corresponding discussions for further explanation). Whereas, the *2NN*, *3NN*, *4NN* and *5NN* have high

distance values with these specific subsequences (i.e. subsequences where the outliers are present in the time series), hence they can be detected by *kNN* MPs. A very similar characteristic is also visible in Fig. 10b.

Some more nice examples can be seen in Fig. 10c and Fig. 10d. There are three outlier elements in the time series (shown in the topmost plot of Fig. 10c) which have appeared as unusual spikes. If we observe carefully, then we can see that these three spikes (or their corresponding subsequences) look very similar to each other. Hence, the *1NN* of these anomalies is a close match among two other similar anomalies. Even the *2NN* of these anomalies will be a close match corresponding to other similar anomalies. Hence, it is not possible to detect them in *1NN* (see the second plot in Fig. 10c) and *2NN* MPs (see the third plot in Fig. 10c). But, if we compute *3NN*, *4NN*, and *5NN* (see the fourth, fifth and sixth plots respectively in Fig. 10c) MPs, then these outliers can be detected because their corresponding subsequences will have a high distance value to their third, fourth and fifth nearest neighbors. A very similar characteristic is also visible in Fig. 10d.

Table 3: The outlier detection accuracy of various kNN MP based on 3 high thresholds on the Yahoo dataset ("A1Benchmark-Real")

	Accuracies		
kNN MP	**Threshold (95%)**	**Threshold (90%)**	**Threshold (85%)**
1NN	0.317	0.413	0.469
2NN	0.349	0.485	0.556
3NN	0.386	0.509	0.584
4NN	0.439	0.522	0.630
5NN	0.458	0.553	0.653
6NN	0.490	0.566	0.673
7NN	0.500	0.610	0.686
8NN	0.509	0.622	0.698
9NN	0.522	0.629	0.704
10NN	0.542	0.643	0.720

We have also performed experiments to evaluate the accuracy of *kNN* MPs in comparison with *1NN* MP for anomaly detection by using all the labeled benchmarks from the Yahoo dataset. In our experiments, we generate the kNN matrix profiles for $1 \leq k \leq 10$. We consider a subsequence as a *discord* if its value in the matrix profile is higher than a predefined threshold. To automatically calculate the threshold for the detection of discords, we adopt a simple way by taking 95%, 90% and 85% of the maximum value of *1NN* MP. In this manner, for each time series, we can obtain 3 individual thresholds and based on these 3 threshold values, we have detected the discords of $1NN, 2NN, 3NN, ..., 10NN$ MP.

For each labeled outlier, we look within a horizontal window of size $2m$ to find any occurrence of the outlier among the detected discords in each of the $1NN, 2NN, 3NN, ..., 10NN$ MP, where m $(= 32)$ represents the subsequence

Table 4: The outlier detection accuracy of various kNN MP based on 3 high thresholds on the Yahoo dataset ("A2Benchmark-Synthetic")

	Accuracies		
kNN MP	**Threshold (95%)**	**Threshold (90%)**	**Threshold (85%)**
1NN	0.374	0.512	0.648
2NN	0.606	0.794	0.835
3NN	0.711	0.850	0.913
4NN	0.803	0.863	0.934
5NN	0.857	0.948	0.991
6NN	0.876	0.963	0.992
7NN	0.944	0.988	0.997
8NN	0.978	0.993	0.997
9NN	0.989	0.992	0.997
10NN	0.991	0.993	0.997

length, used for MP computation. For example, let's say in any particular time series T, there is a labeled outlier in the ground truth at the l^{th} location. Now, we look within the range of $[(l-m)$ to $(l+m)]$ positions in $1NN, 2NN, 3NN, ..., 10NN$ MP to find the existence of any detected (based on the chosen threshold) discords. If we find a discord within this range, then we consider it as a success, otherwise it is considered as failure in detection. Simply speaking, for each labeled anomaly, we consider it as detected, if its subsequence overlaps with one of the detected discords in the matrix profile. Thus, the accuracy of anomaly detection by a matrix profile is measured as the fraction of detected anomalies over the total number of anomalies. In this manner, we have computed the average accuracies of outlier detection over Yahoo time series.

Tables 3 and 4 show the accuracy of $1NN$ and kNN matrix profiles (for $k = 1, 2, ..., 10$), for *A1Benchmark-Real* and *A2Benchmark-Synthetic* benchmarks from the Yahoo dataset. The results show that the accuracy can significantly be improved by using kNN MP. As seen in Table 3, for the *A1Benchmark-Real* benchmark, the accuracy increases from 41% with $1NN$ MP to 64% with $10NN$ MP (when the threshold is taken as 90%) and 46% with $1NN$ MP to 72% with $10NN$ MP (when the threshold is taken as 85%).

The accuracy gain is even more impressive for the *A2Benchmark-Synthetic* benchmark. As seen in Table 4, the accuracy increases from 37% with $1NN$ MP to 99% with $10NN$ MP (when the threshold is taken as 85%). The accuracy also increases from 51% with $1NN$ MP to 99% with $10NN$ MP (when the threshold is taken as 90%) and 64% with $1NN$ MP to 99% with $10NN$ MP (when the threshold is taken as 85%). The reason for this significant accuracy gain is that some of the (ground truth) anomalies have similar subsequences in the dataset (which can be other anomalies). This is why these anomalies do not appear as discords in the $1NN$ MP, as the distance value to their nearest neighbor is low. However, their distance value to other subsequences may be high (*e.g.*, to 2NN, 3NN, etc), this is why they can be detected by kNN MPs.

6.3 Scalability of kNN similarity search

In this section, we study the scalability of our solution for building the kNN MP, by varying several parameters such as k, number of cores, length of time series (n) and subsequence length (m). In our experiments, we used several datasets. The first one is

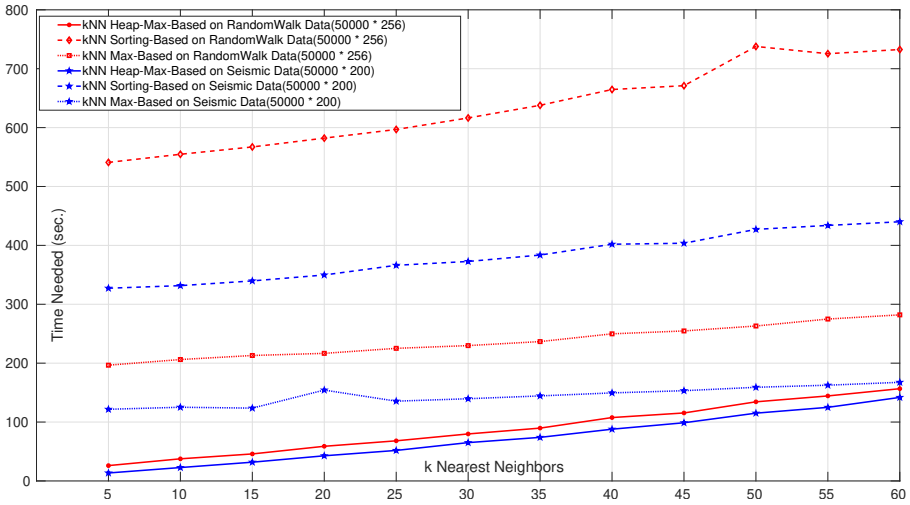
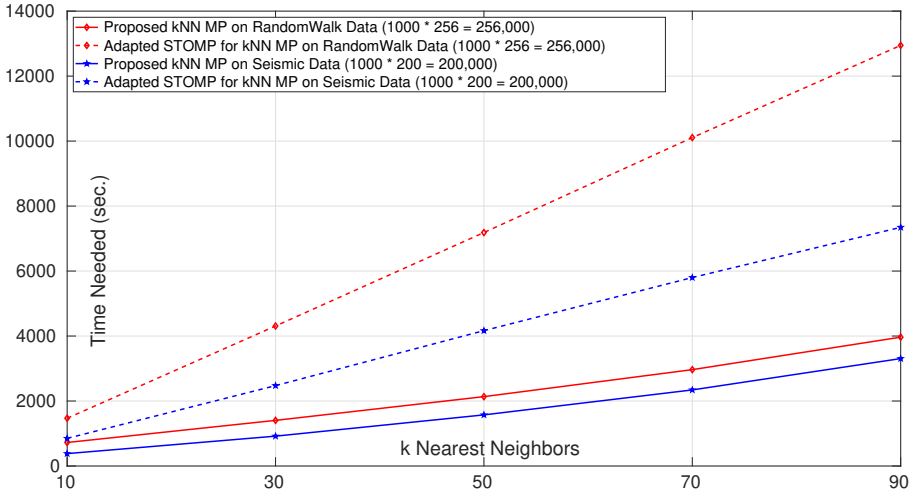


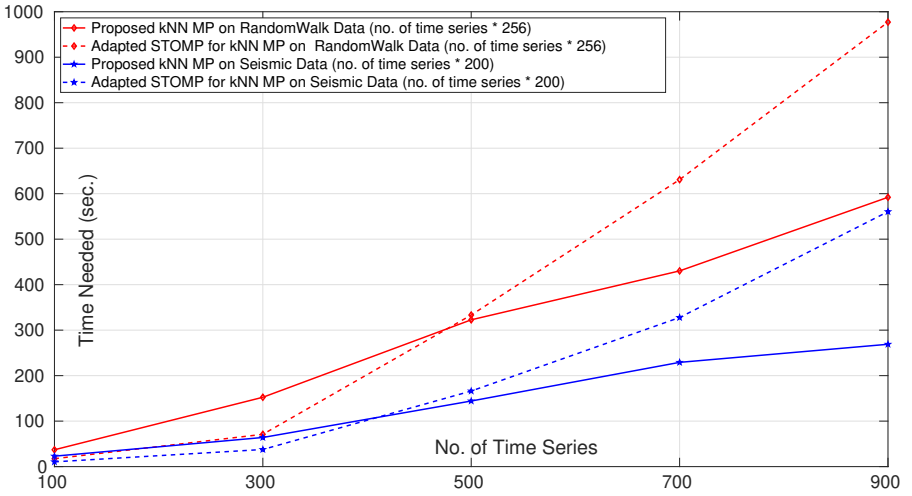
Figure 11: The execution time for computing kNN MP, with increasing the number k. The time is shown for three different proposed approaches, *i.e.*, sort based (refer to Section 4.2.1), max based (refer to Section 4.2.2), and heap-max based (refer to Section 4.2.3) (please see the online color version for better visibility).

called as *seismic* dataset, obtained from the domain of seismology, containing 50000 individual time series. The length of each time series is 200. By concatenating all time series, we obtained a big time series of 1 million values. The second dataset is called *random walk* dataset, having 50000 individual time series. The length of each time series is 256. We also concatenated the time series of this database to create a big time series. The initial three experiments (Fig. 12a, Fig. 12b, Fig.13a) were performed by using these two datasets. For the experiments on the number of cores in (Fig. 13b), we have used the *Hyper-spectral data of protein levels* and *accelerometer* datasets (which are are used in other experiments, mentioned in Section 6.2.1 and 6.2.2). The main reason of choosing these datasets is the higher length of their time series (680 and 500 respectively). As like the previous datasets, here also we have concatenated all the time series to build a big time series. In our experiments the default value for k is taken as 10.

For the first experiment, we incremented the value of k to study its effect on the time required for computing kNN MP. As seen in Fig. 12a, with increasing k, the required computational time increases linearly for both datasets. But, the computational time doesn't increase drastically for higher values of k, and the proposed



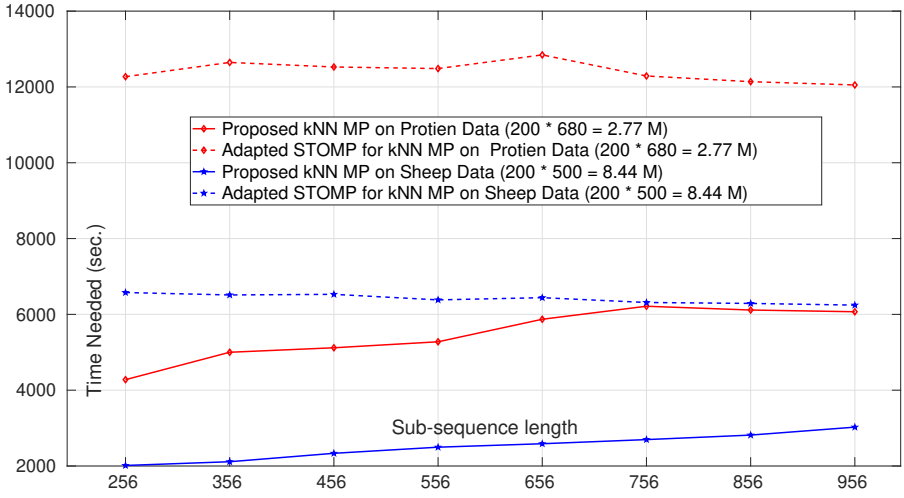
(a)



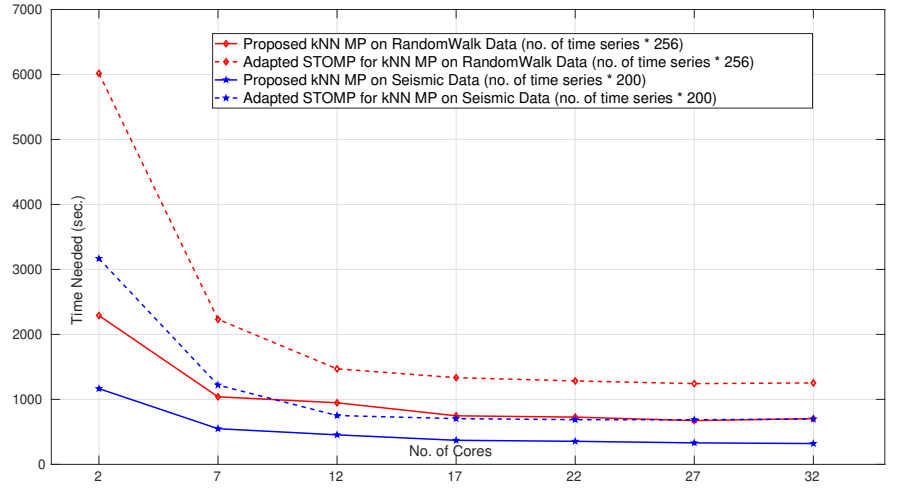
(b)

Figure 12: The execution time of our proposed algorithm and that of STOMP on random-walk, seismic, protein and sheep datasets: (a) The variation of execution time with increasing k for generating kNN MP. The colors red and blue represent the plot for random-walk and seismic datasets respectively. (b) The variation of execution time with increasing the length of time series (x-axis is representing the number of individual time series, which are concatenated to generate a single big time series) (please see the online color version for better visibility).

algorithm is able to give output of high kNN values with small change in computational time. From the second experiment shown in Fig. 12b, it can be seen that the



(a)



(b)

Figure 13: The comparative performance of our proposed algorithm and the technique by Yeh et.al [2] on random-walk, seismic, protein and sheep datasets: (a) The variation of computational time with increasing the length of subsequence (m) for the protein and sheep datasets respectively. (b) The computational time with increasing the number of cores for the random-walk and seismic datasets respectively (please see the online color version for better visibility).

computational time increases with increasing time series length. In Fig. 13a, it can be seen that computational time is almost linear with increasing the subsequence length.

Fig. 13b shows the evolution of the computational time with increasing the number of cores. As seen, the computational time firstly decreases significantly with the increase of number of cores for both datasets, but after a certain number of cores, it gets constant (after 17 cores).

6.3.1 Comparison with adapted STOMP

We have compared the computational time of STOMP algorithm by Yeh et.al [2] with our proposed technique for kNN MP. In general, the STOMP algorithm is designed to compute $1NN$ MP. We adapted it for kNN MP by applying it in k iterations, such that the best matches obtained in each iteration are excluded before the next iteration. Fig. 12 and 13 show the computational time of our proposed algorithm and STOMP algorithm by using the previously mentioned *seismic* and *random walk* datasets. The variation of computational time is depicted by increasing *k* by using 1000 individual time series from each dataset (seismic and random walk datasets, shown in Fig. 12a). It can be seen that our algorithm outperforms significantly the STOMP technique for computing kNN MP.

The second experiment, shown in Fig. 12b is performed to compare the computational time with increasing the time series length. The *x-axis* in the plot represents the number of individual time series which are concatenated to generate the single big time series to calculate *kNN MP*. In this experiment, our proposed technique has outperformed the STOMP algorithm. In the third experiment (shown in Fig. 13a), where the comparison is performed by increasing the subsequence length, our proposed technique has also better execution time than the STOMP algorithm, for computing kNN matrix profile.

7 Conclusion

In this paper, we proposed an efficient technique for computing *kNN MP*. Our technique is fast, simple and parallelizable on multiple cores of an off-the-shelf computer. Using several real and synthetic datasets, we illustrated the effectiveness of generating *kNN MP* by our technique. Our results show how qualitatively the *kNN* MP is useful for knowledge discovery compared to the *1NN* MP. For example, they show that the accuracy of anomaly detection can be improved significantly, *e.g.*, from 37% with $1NN$ MP to 99% with $10NN$ MP for *A2Benchmark-Synthetic* of the Yahoo dataset.

As a future work, we plan to extend our parallel technique for GPUs. Moreover, we would like to extend the *kNN MP* for the case of multi-dimensional data. Another possible research direction is to develop automatic methods for choosing the best values for kNN matrix profile parameters (such as the subsequence length or k) for anomaly/motif detection from a given dataset. Actually, these parameters should be mainly chosen by the experts of the domain.

Acknowledgment

We greatly acknowledge the funding from *Safran Data Analytics Lab*. The authors are grateful to Inria Sophia Antipolis - Méditerranée "Nef" computation cluster for providing resources and support.

References

[1] Yeh, C.-C.M., Herle, H.V., Keogh, E.J.: Matrix Profile {III:} The Matrix Profile Allows Visualization of Salient Subsequences in Massive Time Series. In: Proceedings of the International Conference on Data Mining (ICDM), pp. 579–588 (2016)

[2] Yeh, C.C.M., Zhu, Y., Ulanova, L., Begum, N., Ding, Y., Dau, H.A., Zimmerman, Z., Silva, D.F., Mueen, A., Keogh, E.: Time series joins, motifs, discords and shapelets: a unifying view that exploits the matrix profile **32**(1), 83–123 (2018)

[3] Zhu, Y., Yeh, C.-C.M., Zimmerman, Z., Kamgar, K., Keogh, E.: Matrix Profile XI: SCRIMP++: Time Series Motif Discovery at Interactive Speeds. In: Proceedings of the International Conference on Data Mining (ICDM), pp. 837–846 (2018)

[4] Sinha, S.: Discriminative motifs. In: Proceedings of the Sixth Annual International Conference on Computational Biology, pp. 291–298 (2002)

[5] Balasubramanian, A., Wang, J., Prabhakaran, B.: Discovering Multidimensional Motifs in Physiological Signals for Personalized Healthcare. J. Sel. Topics Signal Processing **10**(5), 832–841 (2016)

[6] Yagoubi, D.E., Akbarinia, R., Kolev, B., Levchenko, O., Masseglia, F., Valduriez, P., Shasha, D.E.: ParCorr: efficient parallel methods to identify similar time series pairs across sliding windows. Data Mining and Knowledge Discovery **32**(5), 1481–1507 (2018)

[7] Rakthanmanon, T., Campana, B.J.L., Mueen, A., Batista, G.E.A.P.A., Westover, M.B., Zhu, Q., Zakaria, J., Keogh, E.J.: Searching and mining trillions of time series subsequences under dynamic time warping. In: Yang, Q., Agarwal, D., Pei, J. (eds.) ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 262–270 (2012)

[8] Mueen, A., Keogh, E.J., Young, N.E.: Logical-shapelets: an expressive primitive for time series classification. In: Apté, C., Ghosh, J., Smyth, P. (eds.) ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Diego, CA, USA, August 21-24, 2011, pp. 1154–1162 (2011)

[9] Mueen, A., Hamooni, H., Estrada, T.: Time series join on subsequence correlation. In: Kumar, R., Toivonen, H., Pei, J., Huang, J.Z., Wu, X. (eds.) IEEE International Conference on Data Mining, ICDM 2014, Shenzhen, China, December 14-17, 2014, pp. 450–459 (2014)

[10] Yeh, C.M., Zhu, Y., Ulanova, L., Begum, N., Ding, Y., Dau, H.A., Silva, D.F., Mueen, A., Keogh, E.J.: Matrix profile I: all pairs similarity joins for time series: A unifying view that includes motifs, discords and shapelets. In: Bonchi, F., Domingo-Ferrer, J., Baeza-Yates, R., Zhou, Z., Wu, X. (eds.) IEEE 16th International Conference on Data Mining, ICDM 2016, December 12-15, 2016, Barcelona, Spain, pp. 1317–1322 (2016)

[11] Zhu, Y., Zimmerman, Z., Senobari, N.S., Yeh, C.M., Funning, G.J., Mueen, A., Brisk, P., Keogh, E.J.: Matrix profile II: exploiting a novel algorithm and gpus to break the one hundred million barrier for time series motifs and joins. In: Bonchi, F., Domingo-Ferrer, J., Baeza-Yates, R., Zhou, Z., Wu, X. (eds.) IEEE 16th International Conference on Data Mining, ICDM 2016, December 12-15, 2016, Barcelona, Spain, pp. 739–748 (2016)

[12] Zhu, Y., Yeh, C.-C.C.M., Zimmerman, Z., Keogh, E.J.: Matrix Profile {XVII:} Indexing the Matrix Profile to Allow Arbitrary Range Queries. In: International Conference on Data Engineering (ICDE), pp. 1846–1849 (2020)

[13] Nakamura, T., Imamura, M., Mercer, R., Keogh, E.J.: MERLIN: parameter-free discovery of arbitrary length anomalies in massive time series archives. In: Plant, C., Wang, H., Cuzzocrea, A., Zaniolo, C., Wu, X. (eds.) 20th IEEE International Conference on Data Mining, ICDM 2020, Sorrento, Italy, November 17-20, 2020, pp. 1190–1195 (2020)

[14] Mercer, R., Alaee, S., Abdoli, A., Singh, S., Murillo, A.C., Keogh, E.J.: Matrix profile XXIII: contrast profile: A novel time series primitive that allows real world classification. In: Bailey, J., Miettinen, P., Koh, Y.S., Tao, D., Wu, X. (eds.) IEEE International Conference on Data Mining, ICDM 2021, Auckland, New Zealand, December 7-10, 2021, pp. 1240–1245 (2021)

[15] Zimmerman, Z., Kamgar, K., Senobari, N.S., Crites, B., Funning, G.J., Brisk, P., Keogh, E.J.: Matrix profile XIV: scaling time series motif discovery with gpus to break a quintillion pairwise comparisons a day and beyond. In: Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019, pp. 74–86 (2019)

[16] Zimmerman, Z.P.: Breaking computational barriers to perform time series pattern mining at scale and at the edge. PhD thesis, University of California, Riverside, https://escholarship.org/content/qt51z7d647/qt51z7d647.pdf (2019)

[17] He, Y., Chu, X., Wang, Y.: Neighbor profile: Bagging nearest neighbors for unsupervised time series mining. In: 36th IEEE International Conference on

Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020, pp. 373–384 (2020)

[18] Zhu, Y., Zimmerman, Z., Senobari, N.S., Yeh, C.-C.M., Funning, G., Mueen, A., Brisk, P., Keogh, E.J.: Matrix Profile {II:} Exploiting a Novel Algorithm and GPUs to Break the One Hundred Million Barrier for Time Series Motifs and Joins. In: Proceedings of the International Conference on Data Mining (ICDM), pp. 739–748 (2016)

[19] Dau, Hoang Anh and Keogh, Eamonn and Kamgar, Kaveh and Yeh, Chin-Chia Michael and Zhu, Yan and Gharghabi, Shaghayegh and Ratanamahatana, Chotirat Ann and Yanping and Hu, Bing and Begum, Nurjahan and Bagnall, Anthony and Mueen, Abdullah and Batista, Gustavo: The UCR Time Series Classification Archive. https://www.cs.ucr.edu/~eamonn/time_series_data_2018/ (2018)

[20] Laptev Nikolay and Amizadeh Saeed and Billawala Youssef: A Benchmark Dataset for Time Series Anomaly Detection. https://yahooresearch.tumblr.com/post/114590420346/a-benchmark-dataset-for-time-series-anomaly

Appendix: I Fast Calculation of Distance

Mueen et. al proposed a technique, known as *Mueen's Algorithm for Similarity Search (MASS)* [9] [8] for the fast calculation of z-normalized *Euclidean Distance* between query subsequence and the subsequence of target time series, by exploiting *Fast Fourier Transform (FFT)*. Let us first explain Algorithm 6 that generates the dot product of a subsequence (q), and the subsequences of a target time series (T).

Algorithm 6: SLIDINGDOTPRODUCT

Input: A query subsequence (q), A target time series (T)
Output: The dot product (qT) between single query subsequence and all the target subsequences

1 $n \leftarrow$ no. of elements in T; $s \leftarrow$ no. of elements in q
2 $T_a \leftarrow$ double the length of T by appending n number of zeros at the end
3 $q_a \leftarrow$ reverse the elements in q so that the last element become first and vice versa
4 $q_{ra} \leftarrow$ append $(2n - m)$ zeros at the end of q_r
5 $q_{raf} \leftarrow$ do $FFT(q_{ra})$; $T_{af} \leftarrow$ do $FFT(T_a)$
6 $M \leftarrow$ elementwise multiplication of q_{raf} and T_{af} *// as both q_{raf} and T_{af} are of the same size, so we can easily multiply element to elements*
7 $qT \leftarrow InverseFFT(M)$
8 **return** P_T

In Line 4, both vectors q and T are made to be of the same length (see Section Appendix: I.1) by appending the required amount of zeros $(2n - m)$ to the

reversed query so that like T_a, q_{ra} will have $2n$ elements. This is done to facilitate element wise multiplication in frequency domain. In Line 5, the Fourier transform of q_{ra} and T_a is performed to transform time domain signals into frequency domain signals. Then in Line 6, an element wise multiplication is performed between two complex valued vectors, followed by inverse FFT on the resultant product.

Appendix: I.1 Relation between convolution in time domain with frequency domain

The time domain convolution of two signals $T = [t_1, t_2, ..., t_n]$ and $q = [q_1, q_2, ..., q_m]; m << n$ can be calculated by sliding q upon T. For implementation, we would need $\frac{m}{2}$ number of zeros padding at the beginning and at the end of original T vector. The convolution between these two vectors is represented by $(T * q)$ which is a vector $\mathbf{c} = [c_1, c_2, ..., c_{n+m-1}]$ denoted as : $\mathbf{C}_p = \sum_u T_u \; q_{p-u+1}$ where $u = [\max(1, p-u+1)].....[\min(n, m)]$ and u ranges over all the sub-scripts for T_u and q_{p-u+1}.

The convolution in time domain can be quickly calculated by element-wise multiplication in frequency domain by taking *Fourier Transform* of two signals, multiplying them element-wise and then applying inverse Fourier transform. Performing full convolution (in both time and frequency domains) between two $1D$ (same for $2D$ also) signals of size n and m results in an output of size $(n + m - 1)$ elements. Usually, the two signals are made of same size by padding zeros at the end to facilitate the multiplication operation.

Appendix: II Fast Calculation of Mean and Standard Deviation :

The fast calculation of mean (μ) and standard deviation (σ) of a vector of elements (x) is proposed by Rakthanmanon et.al [7]. The technique needs only one scan through the sample to compute the mean and standard deviation of all the subsequences. The mean of the subsequences can be calculated by keeping two running sums of the long time series which have a lag of exactly m values.

$$\mu = \frac{1}{m}\left(\sum_{i=1}^{k} x_i - \sum_{i=1}^{k-m} x_i\right) \quad \sigma^2 = \frac{1}{m}\left(\sum_{i=1}^{k} x_i^2 - \sum_{i=1}^{k-m} x_i^2\right) - \mu^2 \quad (2)$$

In the same manner, the sum of squares of the subsequences can also be calculated which are used to compute the standard deviation of all the subsequences by using Equations 2.

Appendix: III Brief description of MASS algorithm :

The *MASS* algorithm is mentioned in Algorithm 7. In Line 1 of this algorithm, the sliding dot product is calculated by using Algorithm 6.

The z-normalized Euclidean distance (D_i) is calculated between two time series subsequences q and T_i by using the dot product between them (qT_i). The formula to

Algorithm 7: MASS $(q, \mu_q, \sigma_q, T, \mu_T, \sigma_T)$

Input: A query subsequence (q), mean of q (μ_q), standard deviation of q (σ_q), Target time series (T), mean of T (μ_T), standard deviation of T (σ_T)

Output: Distance profile (D), Dot product (qT)

1 $qT \leftarrow SlidingDotProducts(q, T)$ *//see Algorithm 6*

2 $D \leftarrow CalculateDistanceProfile(qT, \mu_q, \sigma_q, \mu_T, \sigma_T)$ *// see Equation 3*

3 **return** qT, D

calculate the distance (D_i) is shown below.

$$D[i] = \sqrt{2m\left(1 - \frac{qT_i - m\mu_q\,\mu_{T_i}}{m\sigma_q\,\sigma_{T_i}}\right)} \tag{3}$$

In this Equation 3, m is the subsequence length, μ_q is the mean of query sequence q, μ_{T_i} is the mean of T_i, σ_q is the standard deviation of q and σ_{T_i} is the standard deviation of T_i.

Appendix: IV Brief description of STAMP algorithm :

The *Scalable Time Series Anytime MP (STAMP)* algorithm, proposed by Yeh et.al [2] (outlined in Algorithm 8) calculates the closest match (*1NN*) of every subsequence in a time series T, based on the calculated distance (called as *distance profile*) between any particular subsequence with all the remaining subsequence in T.

The pseudo-code is mentioned in Algorithm 8. A for loop is used in Line 6 to chop each subsequence (consider as *query* for better understanding). Then, a distance vector $(Dist_{cutQuery})$ is computed by calculating the distances between all other subsequences in T and the *query*. In each iteration, the smallest distance and its corresponding index are kept in P_T and I_T vectors.

Appendix: IV.1 Two independent time series matching using STOMP

In Section 4.1.1, we have explained the general principal of *STOMP* algorithm for the computation of MP for a single time series. In case of two independent time series, the basic *STOMP* algorithm needs to be marginally modified. The pseudo-code of modified *STOMP* algorithm, named as *IndependentSTOMP* is mentioned in Algorithm 9.

While called, this algorithm calculates the distances from 2 query subsequence until the last query subsequences *i.e.*, $(Idxs)$ (see Line 2) because before calling this algorithm, we have already computed the distances between 1^{st} query subsequence and target subsequence t. Now let's concentrate on the principal part of *STOMP*. Remind that the dot product profile (QT) of any particular subsequence, can be derived from the dot product profile of it's previous subsequence (see Section 4.1.1). So, in

Algorithm 8: STAMP(T, m)

Input: The user given time series T, subsequence length m
Output: The MP P_T and associated matrix profile index I_T

$n_T \leftarrow length(T)$ *// get the no. of elements in T*
$P_T \leftarrow$ Initialize this $1D$ vector with inf
$I_T \leftarrow$ Initialize this $1D$ vector with zeros
$Idxs \leftarrow (n_T - m + 1)$ *// total number of possible subsequences*
$\mu_T, \sigma_T \leftarrow ComputeMeanStd(T)$ *// see Equation 2*
for $i \leftarrow 1$ ***to*** $Idxs$ **do**
  $cutQuery \leftarrow T[i \text{ to } (i + m - 1)]$ *// get query subsequence by chopping T from index i to $(i + m - 1)$*
  $Dist_{cutQuery} \leftarrow MASS(cutQuery, \mu_T[i], \sigma_T[i], T, \mu_T, \sigma_T)$ *// calculate the distance between query subsequence and the other subsequences in T*
  $P_T \leftarrow computeElementwiseMin(Dist_{cutQuery}, P_T)$ *// perform element-wise minimum of values from P_T and $Dist_{cutQuery}$ and store the minimum value in P_T*
  $I_T \leftarrow i$ *//update I_T at the indexes where element-wise minimum operation replaces the previously stored value in P_T with the new value from $Dist_{cutQuery}$*
return P_T and I_T *// return the P_T and I_T array*

Algorithm 9: INDEPENDENTSTOMP(t, μ_t, σ_t, $tqSingleVal$, Q, QT, μ_Q, σ_Q)

Input: One target subsequence (t), Mean of t (μ_t), Standard deviation of t (σ_t), Dot product value between first query subsequence and t ($tqSingleVal$), Query time series (Q), Existing dot product between all query subsequences and the previous target subsequence (QT), Mean of whole query time series Q (μ_Q), Standard deviation of Q (σ_Q)
Output: A MP P_Q and associated MP index I_Q

1 $Idxs \leftarrow (n_Q - m + 1)$ *// total number of possible subsequences*
2 **for** $j \leftarrow 2$ ***to*** $Idxs$ **do**
3 $QT[j] \leftarrow QT[j-1] - (Q[j-1] \times t[1]) + (Q[j+m-1] \times t[1+m-1])$
4 $QT[1] \leftarrow tqSingleVal$
5 $Dist_{cutQuery} \leftarrow CalculateDistanceProfile(QT, \mu_t, \sigma_t, \mu_Q, \sigma_Q)$ *// calculate the distance profile*
6 **return** $Dist_{cutQuery}$ and QT *// return the $Dist_{cutQuery}$ and QT array*

Line 3 we repetitively calculate the dot product profile between each query subsequence (from 2^{nd} subsequence onward) and the target subsequence (t). The 1^{st} term *i.e.* ($QT[j-1]$) at the right hand side of Line 3 holds the dot product profile of the previous target subsequence (in comparison with t) and j^{th} query subsequence. The $Q[j-1]$ in 2^{nd} term represents the 1^{st} element of the previous query subsequence

and $t[1]$ represents the 1^{st} element of the target subsequence (t). In the same manner, $Q[j+m-1]$ and $t[1+m-1]$ terms represent the last element of j^{th} query subsequence (current one) and the last element of the subsequences t. In Line 4, the dot product value between the first query subsequence $(j=1)$ and the specific target subsequence (t) is copied in $QT[1]$. This value is taken as input in this algorithm (*i.e.*, $tqSingleVal$) The distance profile is calculated in Line 5 by using the dot product profile (QT), mean (μ_t) and STD (σ_t) value of target subsequence t, mean (μ_Q) and STD (σ_Q) vector of query time series. Finally the distance vector $(Dist_{cutQuery})$ and the dot product vector (QT) are returned as the output from the algorithm.

The time complexity of classical *STOMP* is $\mathcal{O}(n)^2$ which is $\mathcal{O}(\log n)$ improvement over *STAMP* algorithm. This improvement is highly useful for computing MP over big time series.