Matrix Profile

Ekaterina Miller

Summer 2020

1 Introduction

In the paper "Matrix Profile I: All Pair Similarity Joins for Time Series", the authors proposed a fast solution for all-pair-similarity-search. The basic idea of all-pair-similarity-search is as follows: Given a collection of data objects, retrieve the nearest neighbor for every object. Different classification algorithms (like KNN) or clustering algorithms (like k-means) are used to achieve it.

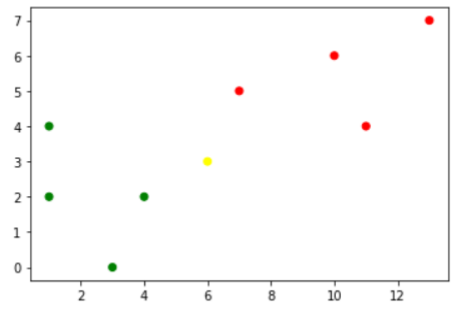


Figure 1:

For example, in **Figure 1** we have a set of data, that splits into two groups (green and red), and our goal is to find out to which group a new data point (yellow) belongs. To solve this problem, we will need to measure the distance from yellow point to every point in the dataset **Figure 2 left**, and find (for example 3) smallest distances **Figure 2 right**. The class with most smallest distances will claim the data point.

The same idea can be used for time series (the data recorded over time). In **Figure 3** we have three time series. The first two are similar to each other, but the third one has an anomaly. We can identify this type of anomalies by measuring the distance between time series.

There are two common problems in the time domain: time series motive and time series discord. **Time series motifs** are pairs of individual time series, or subsequences of a longer time series, which are very similar to each other. **Time series discords** are pairs of individual time series or subsequences of a longer

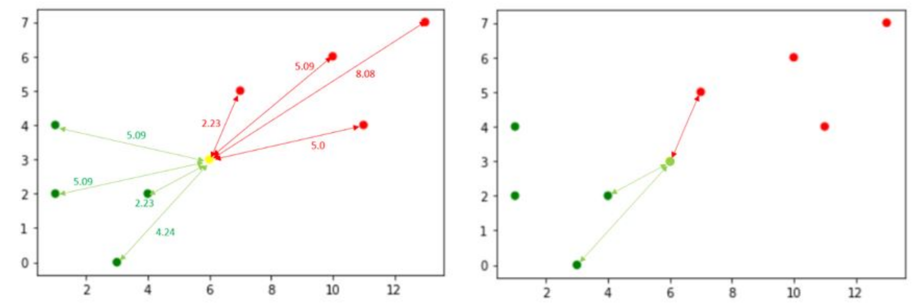


Figure 2:

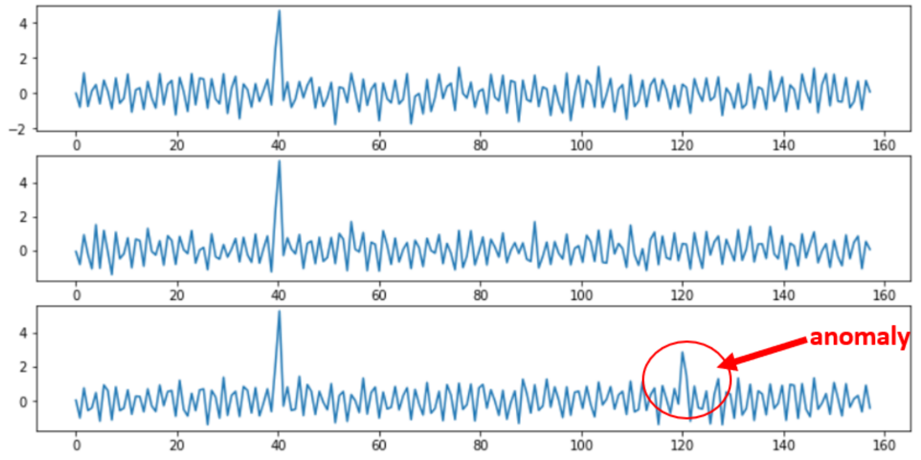


Figure 3:

time series, that is maximally different to all the rest subsequences of a longer time series.

In **Figure 4**, the time series motifs are showing in red and the time series discord is showing in orange. By taking the first "pick" as a query, sliding it across time series, and measuring the distance between quire and each slice of time series, we will be able to identify motifs (other "picks") and discord ("orange fall"). For motifs, the distance will be minimum and for discord, the distance will be maximum. This approach can be used in other domains. For example, the text can be represented as a time series, if we replace each letter with its ASCII code.

In **Figure 5**, the same words correspond to the same shapes of "picks" on the graph. The common applications in the text domain are Community discovery, Duplicate detection, Collaborative filtering, Clustering, and Query refinement. **Community discovery** is a process of detecting communities,

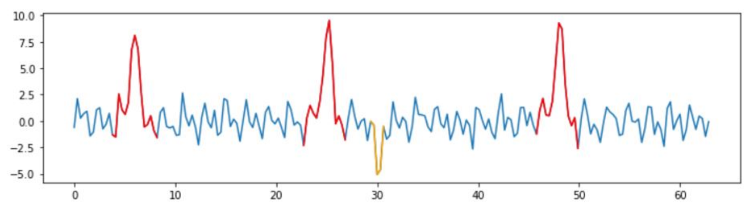
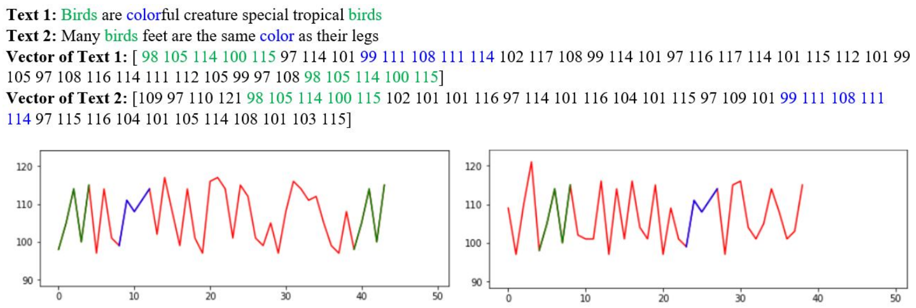


Figure 4:



Text 1: Birds are colorful creature special tropical birds
Text 2: Many birds feet are the same color as their legs
Vector of Text 1: [98 105 114 100 115 97 114 101 99 111 108 111 114 102 117 108 99 114 101 97 116 117 114 101 115 112 101 99 105 97 108 116 114 111 112 105 99 97 108 98 105 114 100 115]
Vector of Text 2: [109 97 110 121 98 105 114 100 115 102 101 101 116 97 114 101 116 104 101 115 97 109 101 99 111 108 111 114 97 115 116 104 101 105 114 108 101 103 115]

Figure 5:

such that the words within a community are often used together. **Duplicate detection** is a process of detecting similar documents. The plagiarism check will fall into this category. **Collaborative filtering** is a method of making automatic predictions about the interests of a user by collecting preferences or taste information from many users. **Clustering** is the grouping of text documents into groups containing similar documents. **Query refinement** is the process of refining (or changing, or narrowing down) a search engine query.

2 Definitions and Notations

Let's say that we have a time series $T = \{t_1, t_2, t_3, \ldots, t_n\}$ and query $Q = \{q_1, q_2, q_3, \ldots, q_m\}$, where $m < n$. Our goal is to find the "best fit" for Q in T. The straightforward solution will include taking all subsequence $T_{i,m}$ of the series T and creating a distance profile $D = \{d_1, d_2, d_3, \ldots, d_{n-m+1}\}$, where d_i is the Euclidean distance between the z-normalized subsequences.For example, if an all-subsequence set $A = \{T_{1,m}, T_{2,m}, T_{3,m}, \ldots, T_{n-m+1,m}\}$, $A[i] = T_{i,m}$, then d_i is the Euclidean distance between $A[i]$ and Q. The minimum of those distances will represent the best fit. Figure 6 demonstrates this concept.

If we graph the distance profile **Figure 7**(bottom), then the minimum values (valleys on the graph) will represent the best fit. The maximum values of the distance profile will represent the discord.

To generalize the problem, let's look at two all-subsequences set A and B

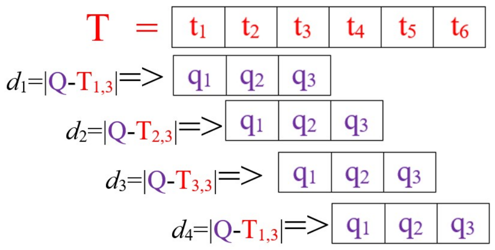


Figure 6: Creating a Distance Profile

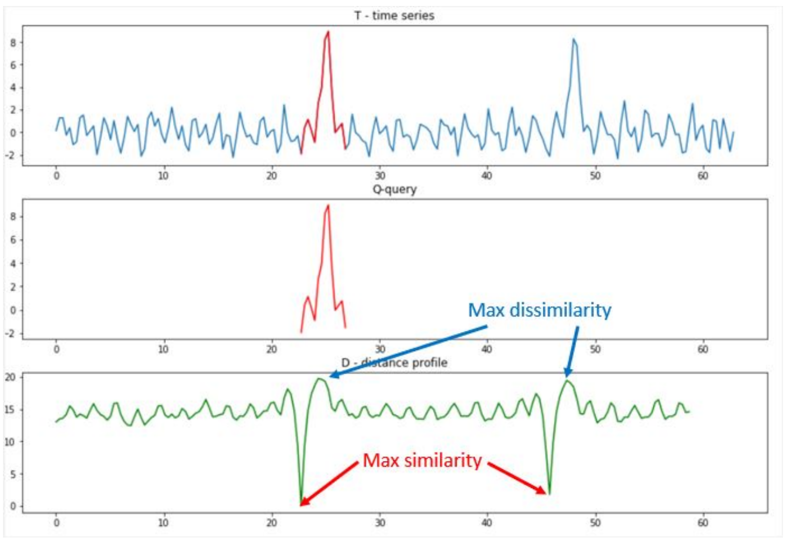
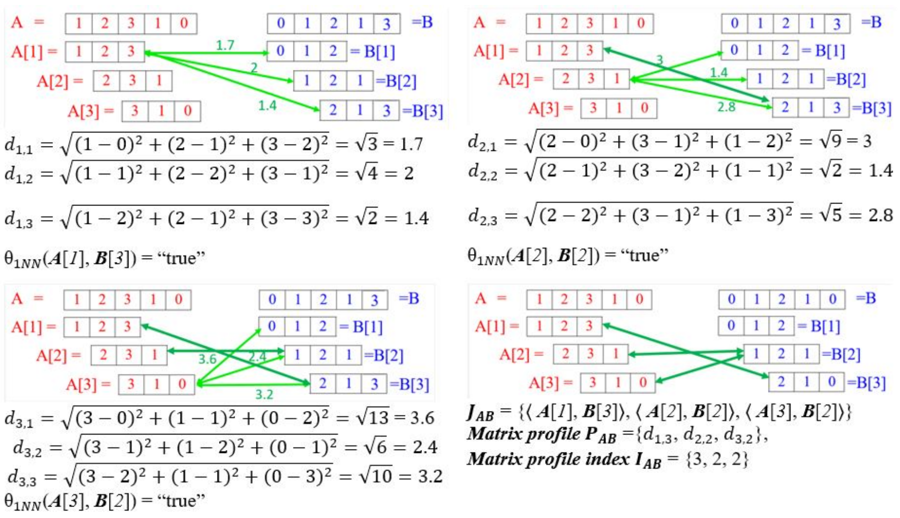


Figure 7: Graph of a Distance Profile



$d_{1,1} = \sqrt{(1-0)^2 + (2-1)^2 + (3-2)^2} = \sqrt{3} = 1.7$

$d_{1,2} = \sqrt{(1-1)^2 + (2-2)^2 + (3-1)^2} = \sqrt{4} = 2$

$d_{1,3} = \sqrt{(1-2)^2 + (2-1)^2 + (3-3)^2} = \sqrt{2} = 1.4$

$\theta_{1NN}(\boldsymbol{A}[1], \boldsymbol{B}[3])$ = "true"

$d_{2,1} = \sqrt{(2-0)^2 + (3-1)^2 + (1-2)^2} = \sqrt{9} = 3$

$d_{2,2} = \sqrt{(2-1)^2 + (3-2)^2 + (1-1)^2} = \sqrt{2} = 1.4$

$d_{2,3} = \sqrt{(2-2)^2 + (3-1)^2 + (1-3)^2} = \sqrt{5} = 2.8$

$\theta_{1NN}(\boldsymbol{A}[2], \boldsymbol{B}[2])$ = "true"

$d_{3,1} = \sqrt{(3-0)^2 + (1-1)^2 + (0-2)^2} = \sqrt{13} = 3.6$

$d_{3,2} = \sqrt{(3-1)^2 + (1-2)^2 + (0-1)^2} = \sqrt{6} = 2.4$

$d_{3,3} = \sqrt{(3-2)^2 + (1-1)^2 + (0-3)^2} = \sqrt{10} = 3.2$

$\theta_{1NN}(\boldsymbol{A}[3], \boldsymbol{B}[2])$ = "true"

$\boldsymbol{J_{AB}} = \{\langle \boldsymbol{A}[1], \boldsymbol{B}[3]\rangle, \langle \boldsymbol{A}[2], \boldsymbol{B}[2]\rangle, \langle \boldsymbol{A}[3], \boldsymbol{B}[2]\rangle\}$

Matrix profile $\boldsymbol{P_{AB}} = \{d_{1,3}, d_{2,2}, d_{3,2}\}$,

Matrix profile index $\boldsymbol{I_{AB}} = \{3, 2, 2\}$

Figure 8:

and two subsequences $A[i]$ and $B[i]$. The authors introduced the following definitions:

1NN-join function: given two all-subsequences sets A and B and two subsequences $A[i]$ and $B[j]$, a *1NN*-join function $\theta_{1nn}(A[i], B[j])$ is a Boolean function which returns "true" only if $B[j]$ is the nearest neighbor of $A[i]$ in the set B.

Similarity join set: given all-subsequences sets A and B, a similarity join set $J_A B$ of A and B is a set containing pairs of each subsequence in A with its nearest neighbor in B: $J_{AB} = \{(A[i], B[j]) | \theta_{1nn}(A[i], B[j])\}$. We denote this formally as $J_{AB} = A \bowtie 1nnB$.

A matrix profile (or just profile) P_{AB} is a vector of the Euclidean distances between each pair in J_{AB}.

A matrix profile index I_{AB} of a similarity join set J_{AB} is a vector of integers where $I_{AB}[i] = j$ if $\{A[i], B[j]\} \in J_{AB}$.

Figure 8 should help with those definitions. Matrix profile and matrix profile index can be used to find the answer to many time series data mining tasks. However, they are too expansive to compute to be practical. The complexity of calculating distance profile and as a result matrix profile is $O(n^2 * m)$. To put it in perspective, think about an example: assume we want to record a sound at 30kHz. It means we will get 30,000 data points per second. If our recording lasts 10 seconds, it will give us 300,000 data points. In this case, the length of the time series is n = 300,000. We would like to sample a one-second query, that will contain m= 30,000 data points. The calculation of the distance profile will take $O(n^2 \cdot m) = 300,000^2 \cdot 30,000 = 2.7 \times 10^15$ operations. Assuming our computer can execute 1,000,000 operations per second. It will take 85.6 years to calculate the matrix profile. As you can see we will need a better algorithm.

3 MASS algorithm

The matrix profile is the minimum value of all the distance profiles. Finding a fast way of computing a distance profile will allow us to compute the matrix profile. The Mueen's ultra-fast Algorithm for Similarity Search (MASS) does just this. The algorithm calculates the z-normalized sliding Euclidean distance between query Q and time series T. It relies on sliding dot product between Q and T and the Convolution theorem. The Convolution theorem stated that the convolution of two sets in the time domain can be calculated be taken the inverse Fourier Transform of the product of Fourier transform of each set. It can be written as

$$A * B = \mathcal{F}^{-1}\{\mathcal{F}(A) \cdot \mathcal{F}(B)\} \tag{1}$$

where $\mathcal{F}$ is the Fourier transform, $\mathcal{F}^{-1}$ is the inverse of Fourier transform, $\cdot$ is the element-wise multiplication, and $*$ is the convolution. Notice that the sliding dot product of the set A and B is the same as the convolution of A and the reverse of B. The python has a function *fftconvolve*. This function calculates the convolution of two sets using the Convolution Theorem. By default, it returns 'full' convolution. Since we need the inner dot product, we can use the mode = 'valid'. The implementation of the sliding dot product is showing bellow.

```
import numpy as np
from scipy import signal

def SlidingDotProduct(T, Q):
    Qr = Q[::-1] #reversing using list slicing
    dot =np.array(signal.fftconvolve(T ,Qr), mode='valid)
    return QT
```

Now we need the formula to calculate sliding z-normalized Euclidean distance using **SlidingDotProduct**. By definition, for the vector $X = [x_1, x_2, \dots, x_n]$ the z-normalise vector $X' = [x_1', x_2', \dots . x_n']$ can be calculated as $x_i' = \frac{x_i - \mu}{\sigma}$, where $i = \overline{0, \mathrm{n}-1}$, μ is the mean of X and $\mu = \frac{\sum_{i=0}^{n-1} x_i}{n}$, and σ is the standard deviation $\sigma = \sqrt{\frac{\sum_{i=0}^{n-1}(x_i - \mu)^2}{n}}$. Let's find the z-normalized Euclidean distance $D[i$ between query Q and subsequence $T_{i,m}$ of the time series T.

$$D[i] = \sqrt{\sum_{j=0}^{m-1} (t_j' - q_j')^2} \tag{2}$$

where t_j' and q_j' are z-normalized elements of the sub subsequence $T_{i,m}$ and query Q respectfully. Using the definition of the z-normalised vector above, (20 can be written as:

$$D[i] = \sqrt{\sum_{j=0}^{m-1} \left(\frac{t_j - \mu_T}{\sigma_T} - \frac{q_j - \mu_Q}{\sigma_Q} \right)^2} \tag{3}$$

where μ_T is the mean of the $T_{i,m}$, σ_T is the standard deviation of $T_{i,m}$, μ_Q is the mean of the Q, σ_Q s the standard deviation of Q. Using the formula for the square of the binomial $(a-b)^2 = a^2 - 2ab + b^2$ and property of the summation notation $\sum(a \pm b) = \sum a \pm \sum b$, the equation 3 can be rewritten as

$$D[i] = \sqrt{\sum_{j=0}^{m-1}\left(\frac{t_j - \mu_T}{\sigma_T}\right)^2 - 2\sum_{j=0}^{m-1}\frac{(t_j - \mu_T)(q_j - \mu_Q)}{\sigma_T \sigma_Q} + \sum_{j=0}^{m-1}\left(\frac{q_j - \mu_Q}{\sigma_Q}\right)^2} \tag{4}$$

Remember, $\sigma_T = \sqrt{\frac{\sum(t_j - \mu_T)^2}{m}}$. By squaring the both sides of this equation we will get ${\sigma_T}^2 = \frac{\sum(t_j - \mu_T)^2}{m}$. Now solve it for m, we will get $m = \frac{\sum(t_j - \mu_T)^2}{{\sigma_T}^2}$. Similar $\frac{\sum(q_j - \mu_Q)^2}{{\sigma_Q}^2} = m$. We can replace first and last term of the equation (4) with m. In addition, we can get rid of parentheses in the numerator of the middle term. As a result (4) can be written as

$$D[i] = \sqrt{2m - 2\sum_{j=0}^{m-1}\frac{(t_j q_j - t_j \mu_Q - \mu_T q_j + \mu_T \mu_Q)}{\sigma_T \sigma_Q}} \tag{5}$$

Applying the property of the summation notation $\sum(a \pm b) = \sum a \pm \sum b$, equation (5) can be written as

$$D[i] = \sqrt{2m + \frac{-2\sum_{j=0}^{m-1} t_j q_j + 2\sum_{j=0}^{m-1} t_j \mu_Q + 2\sum_{j=0}^{m-1} \mu_T q_j - 2\sum_{j=0}^{m-1} \mu_T \mu_Q}{\sigma_T \sigma_Q}} \tag{6}$$

We can simplify $\sum_{j=0}^{m-1} t_j \mu_Q$ and $\sum_{j=0}^{m-1} \mu_T q_j$ as follow:

$$\sum_{j=0}^{m-1} t_j \mu_Q = \mu_Q \cdot \sum t_j = \mu_Q \cdot \frac{\sum t_j}{m} \cdot m = \mu_Q \cdot \mu_T \cdot m$$

$$\sum_{j=0}^{m-1} \mu_T q_j = \mu_T \cdot \sum q_j = \mu_T \cdot \frac{\sum q_j}{m} \cdot m = \mu_Q \cdot \mu_T \cdot m$$

$\sum_{j=0}^{m-1} t_j q_j = QT$ is the inner dot product and

$$\sum_{j=0}^{m-1} \mu_T \mu_Q = \mu_T \mu_Q \sum_{j=0}^{m-1} 1 = \mu_T \cdot \mu_Q \cdot m$$

With this in mind we can rewrite (6) as

$$D[i] = \sqrt{2m + \frac{-2QT + 2m\mu_Q\mu_T + 2m\mu_Q\mu_T - 2m\mu_Q\mu_T}{\sigma_T \sigma_Q}} \tag{7}$$

Cancel like terms:

$$D[i] = \sqrt{2m + \frac{-2QT + 2m\mu_Q\mu_T}{\sigma_T\sigma_Q}} \tag{8}$$

Multiply numerate and denominator by m and factor out $2m$, we can rewrite (8) as

$$D[i] = \sqrt{2m\left(1 - \frac{QT - m\mu_Q\mu_T}{m\sigma_T\sigma_Q}\right)} \tag{9}$$

The authors use this formula for the implementation of the MASS function. The formula requires the calculation of the means and standard distributions for Q and each subsequence of T.

```
#********************ComputeMeanStd(T, Q)************************
#This function calculates means of Q and each subset of T and
#the standard deviation of Q and each subset of T
def ComputeMeanStd(T, Q):
    n=len(T)
    m= len(Q)
    size = n-m+1
    muT=np.zeros(size)  #the means of the subsets of T size m
    muQ=np.mean(Q)      #the mean of Q
    sdT=np.zeros(size)  #the standard deviations of the subsets
    sdQ=np.std(Q)
    for i in range(0, n-m+1):
        x = T[i: i + m]       #create subsets of T
        muT[i]=np.mean(x)   #calculate means for each subset
        sdT[i] = np.std(x)  #standard deviations for each subset
    return muQ, sdQ, muT, sdT

#*********************MASS(T, Q)*******************************
#This function calculates the Z-Normalized Euclidean distance
#between Q and each subset of T
def MASS(T, Q):
    n=len(T)
    m= len(Q)
    QT=np.array(signal.fftconvolve(T ,Q[:: -1], mode='valid'))
    mQ, sQ, mT, sTm = ComputeMeanStd(Q, T)
    D=np.zeros(n-m+1)
    for i in range(0, n-m+1):
        try:
            #formula from the paper
            D[i]=math.sqrt(2*m*(1-(QT[i]-m*mQ*mT[i])/(m*sQ*sTm[i])))
        except ValueError: #catch rounding error
            D[i] = math.sqrt(2 * m * (1 - (QT[i] -
            m * mQ * mT[i] - 0.000000000000001) / (m * sQ * sTm[i])))
```

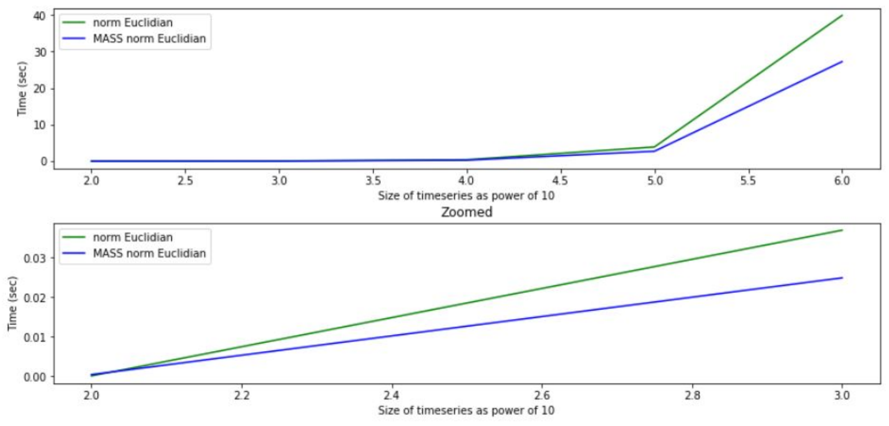


Figure 9: Time complexity of the calculating distance profile for z-normalized Euclidean distance with MASS function compare to Euclidean formula

```
    return D
```

If we test this function, we can see that it outperforms the sliding z-normalize Euclidian distance. The theoretical complexity of the Fast Fourier transform is $O(nlogn)$. However, if we put the data from our experiment into the table, we can see that (for $n > m$) when size gets multiply by 10, the time gets multiply by 10 too. So the actual complexity of the algorithm is $O(n)$. It can be explained

Value of n	10^2	10^3	10^4	10^5	10^6
Time Required	4.6×10^{-4}	2.5×10^{-2}	2.6×10^{-1}	2.7	2.7×10

Table 1: The time it takes to compute MASS for m=100 and varies n

by the fact that the python fft algorithm is highly optimized.

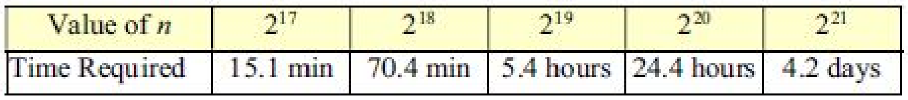
4 The STAMP Algorithm

The authors use MASS function in **S**calable **T**ime series **A**nythime **M**atrix **P**rofile(**STAMP**) algorithm. The algorithm finding the Matrix Profile P_{AB} and Matrix Profile Indexes I_{AB} for two time series T_A and T_B, and subsequence length m. The idea of the algorithm is to cut B into subsequence $B_{i,m}$, find the Distance Profile D between each $B_{i,m}$, and T_A, and find Matrix Profile P_{AB}, as the minimum of those Distance Profiles, and Profile Indexes I_{AB} for each subsequent $B_{i,m}$. The python code for this function provided bellow.

```
def STAMP(Ta, Tb, m):
    na=Ta.shape[0]
    size=na-m+1
    Pab=np.zeros(size)
    Iab=np.zeros(size, dtype='int')
    idxes=np.arange(0, size)
    for idx in idxes:
        D=MASS(Ta[idx:idx+m], Tb)
        Pab[idx]=np.amin(D)
        Iab[idx]=np.argmin(D)
    return Pab, Iab
```

5 Time complexity

The theoretical complexity of the algorithm for calculating the distance profile with MASS function is $O(nlogn)$. For matrix profile we need to calculate the distance profile n-m+1 times, that is O(n) complexity. The total complexity of the STAMP algorithm is $O(n^2logn)$. Simulate to our experiment, the authors

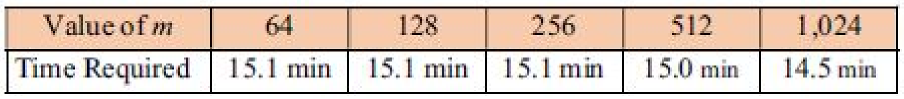


Value of *n*	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}
Time Required	15.1 min	70.4 min	5.4 hours	24.4 hours	4.2 days

Figure 10: The time required to calculate P_{AB} for $m = 256$ and varying n.

data (Figure 10) shows an even better performance of O(n2) complexity.

As the author showed Figure 10, the time complexity of the algorithm is independent of m. This makes it very officiant for self-similarity search.



Value of *m*	64	128	256	512	1,024
Time Required	15.1 min	15.1 min	15.1 min	15.0 min	14.5 min

Figure 11: The time required to calculate P_{AB} for $n = 2^{17}$ and varying m