Pair-wise Correlations Analysis among Multiple Time Series Data

Master Project

Institut für Informatik
der
Universität Zürich

Andris Prokofjevs
Nicoletta Farabullini

Supervisors
AmirReza Alizade Nikoo, Sven Helmer

Zürich
30.07.2021

Contents

1 Introduction

Intensive Care Unit (ICU) patients require a fast and sophisticated analysis of their condition in real time. To do so, it is paramount to compare patient data across time series in order to find outliers in the data leading to potential threats. Similarity search in time series for ICU patients is of much interest as it can immediately detect changes with respect to the physical state of the patient and potential red flags. The goal of this project was to compute pairwise Pearson Correlation Coefficients in a dataset of multiple time series. To illustrate this process, let's take look at a time series x, i.e. a sequence of values ordered according to the time of generation: $x = x_1, x_2, \ldots, x_n$ as illustrated in Figure 1.

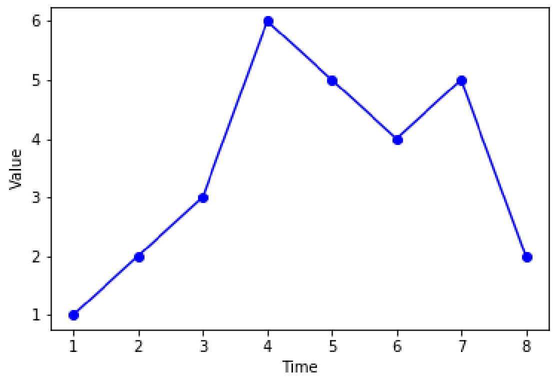

Figure 1: Value VS Time for time series

A correlation serves an indication of the relationship between two series and subsequent assessment of whether they are moving in the same direction. This method is efficient as, compared to other approaches such as simple Euclidean distance calculations, it is scale invariant and unaffected by the addition or subtraction of constants across the time series. Direct calculations (Naive Pearson and Adapted Pearson) as well as dimensionality reduction approaches (PAA, DFT, DWT) were used. Direct computations of Pearson correlations are the most straight forward methods to observe similarities among series. However, to speed up outputs production dimensionality reduction methods have also been developed.

We subsequently performed an analysis to compare run times and performance across all five methods. In order to generate results in the most transparent form, almost all of the code was written using basic Python, with the two exceptions being in the DFT module, where a couple of functions from the math and cmath libraries had to be used due to the presence of the complex numbers.

Our results showed that algorithms behave differently based on which values are changed. In other words, depending on the specific parameter settings algorithms can perform better or worse than others. In particular, while Naive and Adapted Pearson follow a somewhat monotonic pattern that is independent of the changing of parameters values, the dimensionality reduction methods show a decreasing-then-increasing behavior. Sweet spots values were found by targeting the change in behavior for PAA, DFT and DWT with respect to their unique parameters.

2 Pearson correlation

Similarity search in time series is evaluated by computing the Pearson Correlation Coefficient, a statistical measure of correlation between two time series that ranges between +1 and -1: +1 represents a total positive correlation, 0 no correlation, and -1 total negative correlation. A positive correlation is obtained when two series increase and/or decrease at the same pace as shown in Figure 2:

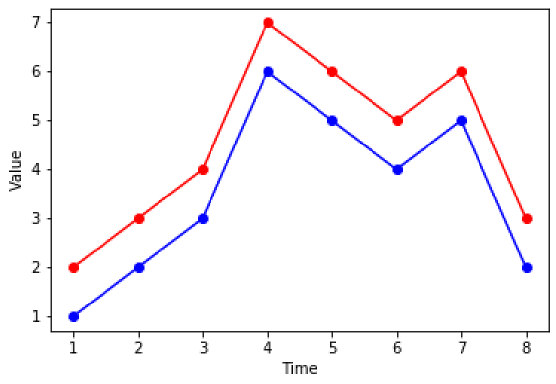

Figure 2: Positive Correlation

A negative correlation is obtained when two series inversely increase and/or decrease as shown in Figure 3:

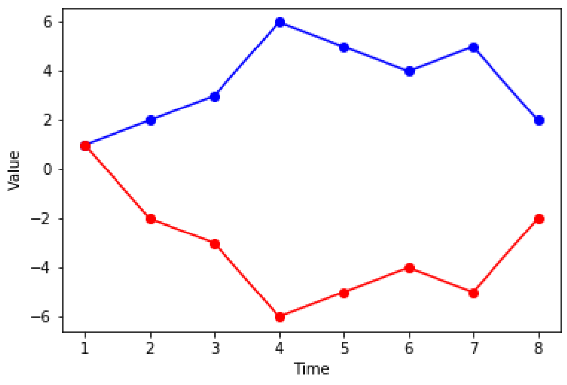

Figure 3: Negative Correlation

A zero correlation is obtained when two series show completely unrelated behaviors as shown in Figure 4:

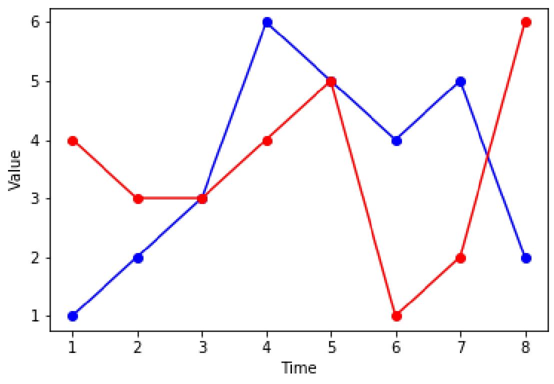

Figure 4: Zero Correlation

Several techniques to calculate the Pearson Correlation Coefficient have been developed. Sections 2 illustrates direct methods and Section 3 explicates dimensionality reduction techniques.

2.1 Naive Pearson

Pearson Correlation measures how one series differ from the other. In its naive approach, this coefficient is directly proportional to the covariance and inversely proportional to the variance. Mathematical derivations of these quantities begin with the definition of the mean:

$$\bar{x} = \frac{1}{n} \cdot \sum_{i=1}^{n} x_i \tag{1}$$

$$var(x) = \frac{1}{n} \cdot \sum_{i=1}^{n} (x_i - \bar{x})^2 \tag{2}$$

$$cov(x,y) = \frac{1}{n} \cdot \sum_{i=1}^{n} [(x_i - \bar{x}) - (y_i - \bar{y})] \tag{3}$$

$$corr(x,y) = \frac{cov(x,y)}{\sqrt{var(x) \cdot var(y)}} = \frac{\sum_{i=1}^{n} [(x_i - \bar{x}) - (y_i - \bar{y})]}{\sqrt{\sum_{i=1}^{n} (x_i - \bar{x})^2 \cdot \sum_{i=1}^{n} (y_i - \bar{y})^2}} \tag{4}$$

Equation (4) clearly shows that as both series move in the same direction (i.e. if their values either linearly increase or decrease), the correlation will be positive. On the other hand, if they diverge, the coefficient will be negative.

Even though this method is sufficient to draw the right conclusions, it can be computationally expensive. In fact, both variance and covariance include both series in their respective equations, which is not ideal from a computational standpoint. Adapted Pearson shows a re-arrangement of these equations such that most of the series parameters are computed separately and thus allows to pre-calculate multiple parameters that are then saved and reused.

2.2 Adapted Pearson

Adapted Pearson modifies the Naive Pearson approach to separate calculations for the two series. By applying summations rules to Equation (4) and performing some simplifications, it is possible to derive a variant of Naive Pearson, which only has one parameter being dependent on both series. These new parameters are labeled s_1 to s_5:

$$s_1 = \sum_{i=1}^{n} x_i \tag{5}$$

$$s_2 = \sum_{i=1}^{n} x_i^2 \tag{6}$$

$$s_3 = \sum_{i=1}^{n} y_i \tag{7}$$

$$s_4 = \sum_{i=1}^{n} y_i^2 \tag{8}$$

$$s_5 = \sum_{i=1}^{n} x_i \cdot y_i \tag{9}$$

$$corr(x, y) = \frac{n \cdot s_5 - s_1 \cdot s_3}{\sqrt{(n \cdot s_2 - s_1^2) \cdot (n \cdot s_4 - s_3^2)}} \tag{10}$$

Because s_5 is the only parameter requiring both time series, most of the values needed in the correlation equation (Equation (10)) can be computed treating x_i and y_i separately, suggesting an improved computational time.

However there may be scenarios where the Pearson Correlation Coefficient is meaningful for a specific task only if above a certain threshold value. Direct methods have no way to make any preliminary filtering of the data to avoid performing unnecessary calculations. Dimensionality reduction techniques aim at speeding up the computational process by making preliminary approximated calculations and subsequent filtering of the data.

3 Dimensionality reduction

Time series can be computationally really expensive as well as unnecessary to analyse as an entire series if there are initial conditions that would filter values out. Dimensionality reduction techniques have been proven to improve processing time by means of approximations and filtering using the Euclidean distance.

3.1 Euclidean distance

All three dimensionality reduction techniques implemented in this work check whether the Euclidean distance between the two time series passes a set threshold.

$$d(\hat{x}, \hat{y}) = \sqrt{\sum_{n=1}^{n} (\hat{x}_i - \hat{y}_i)^2} \tag{11}$$

Where $\hat{x}$ and $\hat{y}$ represent normalized versions of the original dataset. In order to determine if the exact correlation needs to be computed, the Euclidean Distance must pass the following condition [AML10]:

$$d(\hat{x}, \hat{y}) \leq \sqrt{2 \cdot m \cdot (1 - T)} \tag{12}$$

Where T is the threshold value, m the length of the sequence, and $\hat{x}$ is the normalized sequence of x:

$$\hat{x} = \frac{x - \mu_x}{\sigma_x} \tag{13}$$

Doing so will determine whether the correlation is set above the threshold [AML10]:

$$corr(x, y) \geq T \tag{14}$$

Now the Pearson correlation coefficient can be calculated based on the Euclidean distance [AML10]:

$$corr(x, y) = 1 - \frac{1}{2 \cdot m} \cdot d^2(\hat{x}, \hat{y}) \tag{15}$$

Setting a threshold condition as a prerequisite to compute the correlation leads to the absence of false negatives [AML10] and allows for filtering of the dataset, thus improving the speed of run time. Three dimensionality reduction methods were analysed in this work: PAA (Piecewise Aggregate Approximation), DFT (Direct Fourier Transform), and DWT (Direct Wavelet Transform):

3.2 PAA

Piecewise Aggregate Approximation (PAA) is a dimensionality reduction method that approximates a time series $x = x_1, x_2, \ . \ . \ . \ , x_n$ by dividing it into N subsections and calculating the respective mean values, obtaining a new sequence $\bar{X} = \bar{x_1}, \bar{x_2}, \ . \ . \ . \ , \bar{x_n}$. For convenience it is assumed that N is a factor of length of the input series n (in sliding window scenario, series n translates to a window size) [EKM01].

$$\bar{x_i} = \frac{N}{n} \cdot \sum_{j=\frac{n}{N}\cdot(i-1)+1}^{\frac{n}{N}\cdot i} \hat{x_j} \tag{16}$$

The same method is applied to a second series with output $\bar{Y_i}$. $\bar{X_i}$ and $\bar{Y_i}$ are then used to check for the threshold condition (Equation (11)-(14)). If such condition is passed, results are used to calculate the Pearson Correlation Coefficient (Equation (15)). If the threshold condition is not passed, the code will skip the calculation and output None.

Transferring this mathematical concept to a graphical representation in Figure 5, it is possible to observe how the PAA behaves when applied to an approximated X' sequence as an approximation of linear box basis functions.

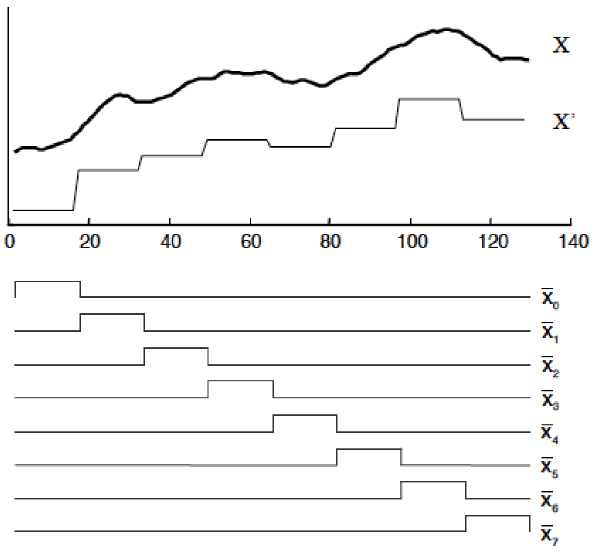

Figure 5: PAA applied to X' [EKM01]

3.3 DFT

The Discrete Fourier Transform (DFT) is one of the most popular techniques in signal processing. It takes a sequence as input and outputs another sequence with complex numbers of same length as the original [AML10].

$$X_f = \frac{1}{m} \cdot \sum_{k=0}^{m-1} x_i \cdot \exp(\frac{-2 \cdot \pi \cdot i \cdot f \cdot k}{m}) \tag{17}$$

Where,
i = the imaginary number $\sqrt{-1}$,
f = coeff_num = data point index of output sequence,
k = data point index of input sequence,
m = sequence length

Equation(17) is applied to two normalized sequences with outputs X_f and Y_f. The two new sequences are then used to calculate the Euclidean distance by separating the real and imaginary components [AML10]:

$$D = \sqrt{\sum_{i=0}^{m} (\bar{x}_{i,real} - \bar{y}_{i,real})^2 + (\bar{x}_{i,im} - \bar{y}_{i,im})^2} \tag{18}$$

If D passes the threshold condition (Equation(14)), a new Euclidean distance is calculated directly from the normalized sequence Equation (13) through Equation (11) and results are used to calculate the Pearson Correlation Coefficient

(Equation (15)). If D does not pass the threshold condition, the code will skip the calculation and output None.

Transferring this mathematical concept to a graphical representation in Figure 6, it is possible to observe how the DFT behaves when applied to an approximated X' sequence as a combination of four Fourier bases waves.

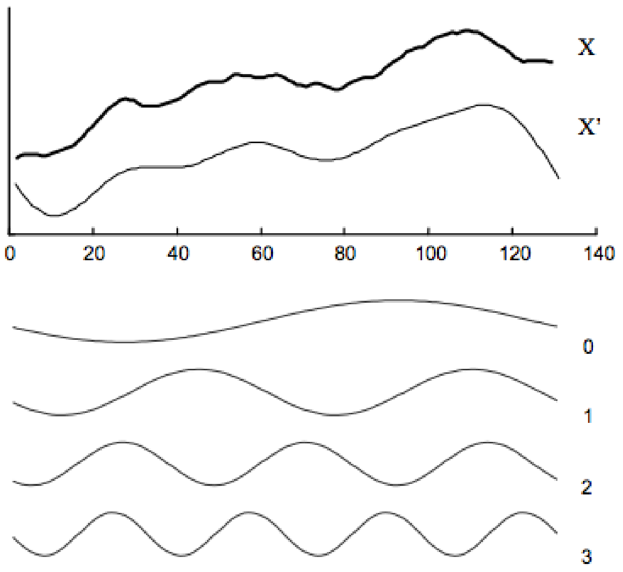

Figure 6: DFT applied to X' [EKM01]

3.4 DWT

The Discrete Wavelet Transform (DWT) used in this project is the Haar transform (H). Given a sequence $\vec{x} = x_1, x_2, \ldots, x_n$, H($\vec{x}$) = ($x_{0,0}$ d_1 d_2 ... d_{2^i+j} d_{2^i+j+1} ... d_{n-1}) where xi, j and d_{2^i+j} are defined as [pCcF99]:

$$x_{i,j} = \frac{x_{i+1,2j} + x_{i+1,2j+1}}{\sqrt{2}} \tag{19}$$

where i represents the vertical level index of the hierarchy, ranging from 0 to $\log_2(n)$ and j the horizontal level index of the hierarchy ranging from 0 to n - 1 [pCcF99]. A graphical representation can be viewed in Figure 7:

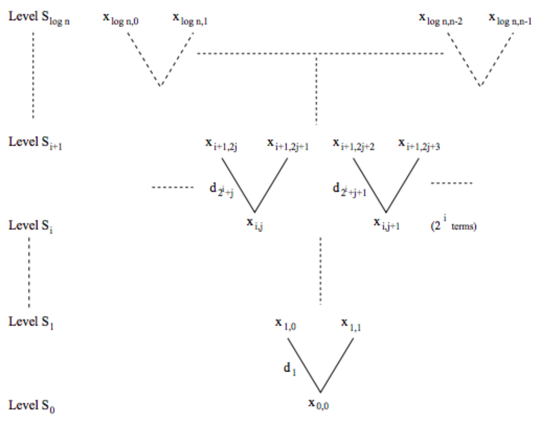

Figure 7: Hierarchy levels in DWT [pCcF99]

$$d_{x,2^i+j} = \frac{x_{i+1,2j} - x_{i+1,2j+1}}{\sqrt{2}} \tag{20}$$

Where denominator is changed from 2 in original formula by [pCcF99] to $\sqrt{2}$ following the authors note that such replacement of scaling factor serves as the normalization step in order to preserve the Euclidean distance between Haar and time domains. Equation(19)-(20) are applied to two normalized sequences $\bar{x}_{i,j}$ and $\bar{y}_{i,j}$. The Euclidean distance is calculated using the information from all previous hierarchy levels up to the one defined by the user. The more levels are taken into account, the more accurate the Euclidean distance and subsequently the correlation estimate will be. The Euclidean distance for each predefined hierarchy level h_c is then calculated [pCcF99]:

$$D = \sqrt{\frac{\sum_{i=0}^{\log_2(n)} \sum_{j=0}^{n-1} (\bar{x}_{i,j} - \bar{y}_{i,j})^2}{2}} \tag{21}$$

If D passes the threshold condition (Equation(14)), a new Euclidean distance is calculated directly from the normalized sequence and results are used to calculate the Pearson Correlation Coefficient. If D does not pass the threshold condition, the code will skip the calculation and output None.

Transferring this mathematical concept to a graphical representation in Figure 8, it is possible to observe how the DWT behaves when applied to an approximated X' sequence as a combination of eight Haar wavelet bases.

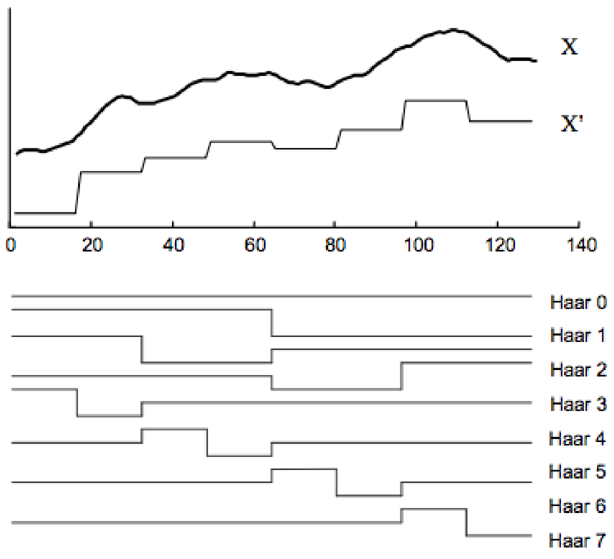

Figure 8: DWT applied to X' [EKM01]

These 5 algorithms were implemented inside our work and their performance was evaluated using a sample dataset.

4 Implementations

The algorithms implementation inside the project was structured into different modules, each performing a specific task. This was done to avoid repeating functions when functions were used in multiple algorithms, performing clear run time measurements, and creating the sample data set only once. All modules were then called in the call_testing_pearson.py file. Figure 9 provides a general overview of the code structure, where numbers are an indication of execution order.

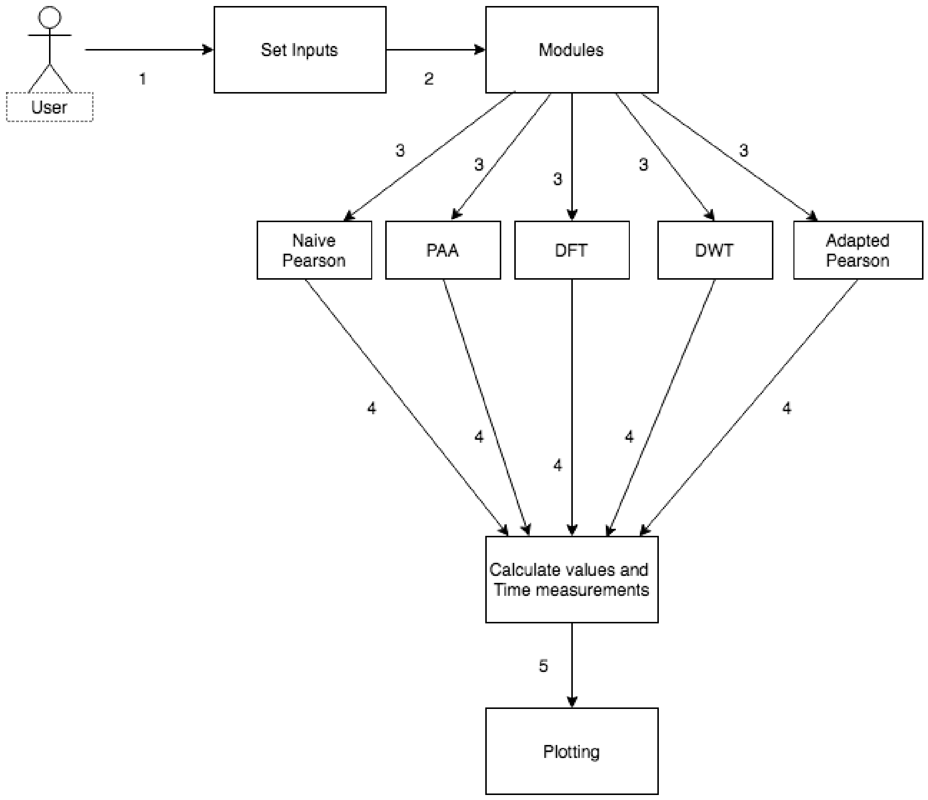

Figure 9: Simplified Project Schema

The same sample data set was used for all algorithms. This sample data set was created to simulate a real-life scenario of receiving measurements from an ICU patient. The individual entries were created at random, however the number of records and sequences were explicitly defined.

Table 1: Sample Data set fragment

-0.02	-0.68	-1.54	-1.94	-2.34	-1.75	-1.29	-1.47	-1.89	-1.71
-0.66	-1.52	-1.92	-1.46	-1.88	-1.42	-1.21	-1.94	-2.02	-2.56
-0.86	-1.26	-0.8	-0.62	-0.41	-0.88	-1.42	-2	-1.18	-0.52
-0.4	-0.8	-1.22	-1.01	-1.09	-0.24	0.58	-0.14	-0.95	-1.11
-0.4	0.19	0.65	0.18	1.03	0.99	1.37	1.77	1.94	1.63

This dataset was then used to simulate the data streaming, in which new data is added and old data removed. This method was implemented by means of a sliding window scenario.

4.1 Sliding Window and Iterations implementation

In order to create a data streaming scenario where new data is continuously fed to the system and old data removed, a much larger version of the dataset in Table 1 was used to simulate sliding windows. As an example, a partioning with window size of 4 elements and a step size of 2 is illustrated in Figures 10 to 12.

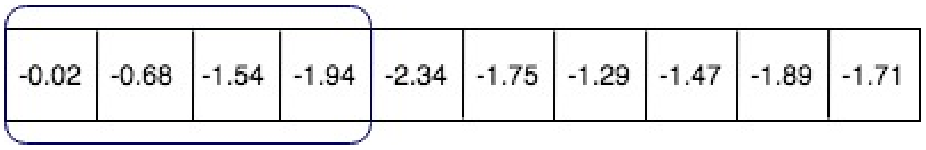

Figure 10: Sliding Window Schema at t_0

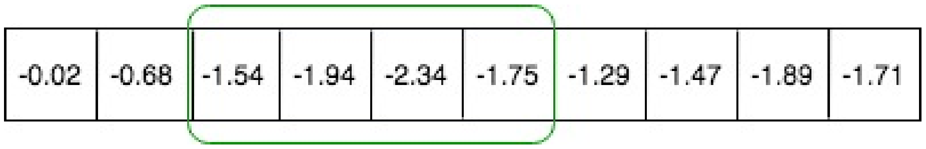

Figure 11: Sliding Window Schema at t_1

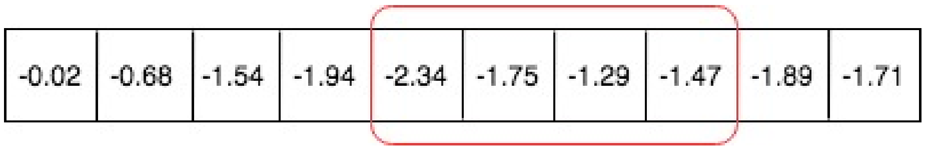

Figure 12: Sliding Window Schema at t_2

Qualitatively speaking, the window color changes with respect to the time, i.e. the blue window represents the examined window at time t_0, the green at time t_1, and the red at time t_2.

In general, each new window is obtained by following a First-In-First-Out approach: starting from the previous window, the number of elements removed corresponds to the step size and are taken from the front, a respective number of elements is added to the back. The amount of elements added always corresponds to the amount removed. Applying this logic to Figures 10, 11 and 12, the first window is set with 4 elements and all coefficients are calculated for that window. Afterwards, two more data points are added to the sequence and the first two data points of the initial window are removed.

This extension and reduction method requires data to be segmented in multiple windows, which in turn mandates that correlations for window pairs have to be calculated sequentially. The use of iterations satisfies this requirement. Two functions were created: one for the initial window and another for any following one. Respectively, as shown in Listing 1 function sliding_window_first_run(dataset, window_size) creates the first window from the data set, whereas function sliding_window_fun(dataset, window_size, step_size, iteration) in Listing 2 outputs all windows elements to add and remove from any window after the initial one is created. The former was used during the first ($0th$) iteration, whereas the latter in all other iterations following the first one. For efficiency reasons, all of the initial windows values were calculated first and then the sliding window technique was applied by looping over the rest of the data set. Example schema of code structure for all implemented algorithms is illustrated in Listing 3

```python
def sliding_window_first_run(dataset, window_size):
    # start by setting window as empty list
    window = []
    # fill window with first batch of elements
    window = dataset[0:window_size]
    return window
```

dataset is given by user or created using parameters that the user inputed.
window_size is defined by user's input

Listing 1: sliding_window_first_run function

```python
def sliding_window_fun(dataset, window_size, step_size, iteration):
    # start by setting window as empty list
    window_step = []

    # detect first element to extend
    start_extension = window_size + step_size * (iteration - 1)
    # detect last element to extend
    stop_extension = start_extension + step_size
    # detect first element to delete
    start_reduction = start_extension - window_size
    # detect last element to delete
    stop_reduction = stop_extension - window_size

    # select the elements that have to be extended
    window_extension = dataset[start_extension:stop_extension]
    # select the elements that have to be deleted
    window_reduction = dataset[start_reduction:stop_reduction]

    window_step = [window_extension, window_reduction]

    return window_step
```

dataset is given by user or created using parameters that the user inputed.
window_size and step_size are defined by the user's input
iteration is generated and passed to this function from general iterations loop (see Listing 3)

Listing 2: sliding_window_fun function

```python
# number of sequences (variable m in parameter input)
dataset_n_col = len(dataset)
# final Pearson correlation coefficients list
output = []
# k is the number of iterations
for k in range(0, (length_of_a_row-window_size) / step_size):
    # define windows and pre-calculate values
    for i in range(0, dataset_n_col):
        if k == 0:
        # define first window with sliding_window_first_run
        else:
        # define first window with sliding_window_fun

        # calculate all windows and values which can be
        # calculated for each sequence separately

        # append everything to list
    for i in range(0, dataset_n_col):
        # extract values for the ith operation
        for j in range(i + 1, dataset_n_col):
            # extract values for the jth operation

            # calculate all remaining variables, which can
            # only be computed with data from both sequences

            # calculate Pearson correlations

            # append results to list
```

It is important to note that values have to be large enough to perform all calculations. For example, if window size was set to 2, variance in Naive Pearson would be 0, which would lead the algorithm to crash. Additionally, most calculations are independent of the iteration value, however this is not the case for the mean.

Listing 3: General code structure

Another important difference between first and all subsequent iterations apart from different method of window computation is mean computation. In first iteration (k = 0) the standard mean calculation formula is used. However in the following iterations (k > 0), the mean function is modified to improve run time performance. Exact function is to be found in Listing 4.

```python
def calc_mean_seq(window_size, mean_val, reduced, extend)
    avg = mean_val - sum(reduced)/window_size + \\
          sum(extend)/window_size
    return avg
```

To compute mean from shifted window, it is sufficient to separately sum the "extend" and "reduced" elements, respectively by adding and subtracting them by the previously calculated mean values, and dividing by the length of the window size.
mean_val is the mean value of the previous window from previous iteration
reduced and extend are values generated by sliding_window_fun function (Listing 2)

Listing 4: Reduced computation time mean function

To sum up the differences between the $0th$ iteration and any following one:

1. if k = 0: first window is defined using sliding_window_first_run(dataset, window_size), the means are calculated with the standard formula;

2. if k > 0: windows are defined using sliding_window_fun(dataset, window_size, step_size, iteration) by extracting individual "extend" and "reduced" components, the means are calculated with the calc_mean_seq(window_size, mean_val, reduced, extend) formula.

Final results for run time[1] and Pearson Correlations Coefficients are appended to an output list. This general logic is then adapted to each different algorithm. In the following subsections, we describe specific implementations of different algorithms with respect to the structure presented in Listing 3

4.1.1 Naive Pearson implementation

Within the first i loop, mean values for each kth iteration are calculated. These are then extracted in the nested i and j loop and remaining calculations of variance, covariance and correlation itself are performed. Before executing Naive Pearson or any other algorithm it is important to check that step_size is smaller or equal to window_size.

4.1.2 PAA implementation

Within the first i loop X_{bar} values from normalized sequences for each kth iteration are calculated as well as a check for N to be less of equal to the window length and being a factor of window_size. Normalised window and X_{bar} values are appended to the list. These are then extracted in the nested i and j loop where Euclidean distance calculations, threshold checks, and final Pearson correlation calculations are performed.

4.1.3 DFT implementation

Within the first i loop X_f values from normalized sequences for each kth iteration are calculated. Normalised window and X_f values are appended to the list. These are then extracted in the nested i and j loop where euclidean distance calculations, threshold checks, and final Pearson correlation calculations are performed. It is important to note that coeff_num should be less or equal to window_size.

4.1.4 DWT implementation

Within the first i loop X_w values from normalized sequences for each kth iteration are calculated. Normalised window and X_w values are appended to the list. These are then extracted in the nested i and j loop where euclidean distance calculations, threshold checks, and final Pearson correlation calculations are performed. Since hierarchy_level in DWT implementation is of central importance, it is important that it is bigger or equal to the maximum amount of times that a window_size can be divided by 2 without decimal, thus math.log(window_size, 2) >= hierarchy_level.

[1]More on run time calculations in Section 4.3

4.1.5 Adapted Pearson implementation

Within the first i loop, s_1 and s_2 values for both sequences are calculated using the sequence_sum_first_run function for the $0th$ k iteration. The function is described in Listing 5:

```python
def sequence_sum_first_run(dataset):
    s_1 = sum(dataset)
    s_2 = sum([x ** 2 for x in dataset])
    output = [sequence_sum1, sequence_sum2]
    return output
```

Listing 5: sequence_sum_first_run function

For any other kth iteration, an optimized function sequence_sum described in Listing 6 was used:

```python
def sequence_sum(prev_sum, reduced, extend):
    # add and subtract "reduce" and "extend" parameters
    s_1 = prev_sum[0] - sum(reduced) + sum(extend)
    s_2 = prev_sum[1] - sum(x ** 2 for x in reduced) + \\
          sum(x ** 2 for x in extend)
    output = [s_1, s_2]
    return output
```

This function uses similar reduced and extend logic as mean calculations mentioned in 4

Listing 6: sequence_sum function

Window elements, s_1, and s_2 values are appended to a list. These are then extracted in the nested i and j loop, where s_5 and final Pearson correlation calculations are performed.

In general, avoiding repetitiveness is of great support in improving the run time. An initial indication of how run times compare for each method is the time complexity.

4.2 Time complexity

Time complexity calculations provide an indication of how fast each algorithm will run. A lower time complexity is indicative of a better run time performance. To evaluate each algorithm, the following rules were applied:

- a simple expression or if statement outside of any loop will have time complexity of O(1)
- a loop will will repeat itself as many times as the length of the input n (e.g. n), hence the time complexity is O(n)
- two nested loops will repeat themselves as many times as the multiplication of length of the inputs (e.g. n*m), hence the time complexity is O(n*m)
- a recursive function will repeat itself until the exiting condition is met, hence the time complexity is O(log(n))

- if a script includes expressions, if statements, and loops, the higher time complexity value is predominant
- if there are two or more independent predominant expressions, their time complexities are summed together.

These rules were applied to all of the five algorithms to calculate their time complexities. For simplicity in the notation, the length of the data set sequence is labeled as m and the number of records in the data set as n. Additionally, step size is usually smaller than window size but it was still included in the time complexity result as this fact cannot be taken as a universal assumption.

4.2.1 Naive Pearson

Naive Pearson contains an outer k for loop ranging from 0 to number of windows, and 2 inner loops: an independent i loop of complexity O(m*(window_size+step_size)), and a nested i and j loop of complexity O(m^2 * window_size). Combining these, the final time complexity is: O(m * k * (step_size + m * window_size)).

4.2.2 Adapted Pearson

Adapted Pearson contains an outer k for loop ranging from 0 to number of windows, and 2 inner loops: an independent i loop of complexity O(m * (step_size + window_size)), and a nested i and j loop of complexity O(m^2 * window_size). Combining these, the final time complexity is: O(m * k * (step_size + m * window_size)).

4.2.3 PAA

PAA Pearson contains an outer k for loop ranging from 0 to number of windows, and 2 inner loops: an independent i loop of complexity O(m * (step_size + window_size), and a nested i and j loop of complexity O(m^2 * (window_size + N)), where K = window_size / N . Combining these complexities, the final time complexity is: O(k * m * (m * window_size + N + step_size).

4.2.4 DFT

DFT Pearson contains an outer k for loop ranging from 0 to number of windows, and 2 inner loops: an independent i loop of complexity O(m * (window_size * coeff_num + step_size)), and a nested i and j loop of complexity O(m^2 * (window_size + coeff_num). Combining these, the final time complexity is: O(k * m (m * (window_size + coeff_num) + (window_size * coeff_num + step_size))).

4.2.5 DWT

DWT Pearson contains an outer k for loop ranging from 0 to number of windows, and 2 inner loops: an independent i loop of complexity O(m * window_size + step_size), and a nested i and j loop of complexity O(m^2 * hierarchy_level * window_size). Combining these last two complexities with the k loop, the final time complexity is: O(k * m (step_size + m * hierarchy_level * window_size)).

4.2.6 Conclusions on Time complexity

Based on these time complexity calculations, a prediction can be made regarding the impact of different input values on the performance. Summation of findings for each algorithm:

- Naive Pearson: O(m * k * (step_size + m * window_size))
- Adapted Pearson: O(m * k * (step_size + m * window_size))
- PAA: O(k * m * (m * window_size + N + step_size)
- DFT: O(k * m (m * (window_size + coeff_num) + (window_size * coeff_num + step_size)))
- DWT: O(k * m (step_size + m * hierarchy_level * window_size))

Even though Naive and Adapted Pearson display analogous time complexities, their internal math and implementations differ. Section 5.2.1 will go into more details regarding this matter. In addition, depending on the values for N, coeff_num and hierarchy_level, the dimensionality reduction techniques perform differently and can be either better or worse than Adapted Pearson.

To look at more accurate results, run time calculations were performed and visualisations were created.

4.3 Run time calculations

Run time computations were used to calculate the algorithms performance. The time library was imported in each module and the time.perf_counter() function was used to measure the time difference between two consecutive calls of the function; it returns a float value of time in seconds. This was the function of choice as it measures elapsed time with the highest resolution available, which is of use in the "initial_and_incremental" execution mode, which is explained in Section 5.1. However comparing this function to other possibilities given in Python, high time resolution is a trade off as we loose the ability to distinguish between system and user CPU time. Sample code similar to Listing 3, but expanded with time measurement points can be found in Listing 7:

```
# number of sequences (variable m in parameter input
dataset_n_col = len(dataset)
# final Pearson correlation coefficients list
output = []
# k is the number of iterations
for k in range(0, (length_of_a_row-window_size) / step_size):
    # define windows and pre-calculate values
    for i in range(0, dataset_n_col):
        if k == 0:
        ẽxtbf initial
            start_initial = time.perf_counter()
            # calculate all windows and values which can be
            # calculated for each sequence separately
            end_initial = time.perf_counter()
            output.append([end_initial - start_initial, "
    initial_time"])
        else:
            start_incremental = time.perf_counter()
             # calculate all windows and values which can be
            # calculated for each sequence separately
            end_incremental = time.perf_counter()
            output.append([end_incremental - start_incremental, "
    incremental_time"])
        # append everything to list
    for i in range(0, dataset_n_col):
        # extract values for the ith operation
        for j in range(i + 1, dataset_n_col):
            # extract values for the jth operation
            # calculate all remaining variables, which can
            # be only be computed with data from both sequences
            start_calculated_sequence_objects = time.perf_counter()
            # calculate Pearson correlations
            end_calculated_sequence_objects = time.perf_counter()
            output.append([pearson_corr,
    calculated_sequence_objects_time, "
    calculated_sequence_objects_time"])
```

Listing 7: General code structure expanded with time measurement points

Qualitative description of Listing 7 is as follows:

1. At the $0th$ iteration (when k is 0), the run time count was started and saved under the start_initial variable. Once all required calculations in initial run were done, time was saved under end_initial. The difference between these two variables represents the time taken for a single iteration in the initial window calculations

2. At any k subsequent iteration, the run time count was taken and saved under the variable start_incremental variable. Similarly to start_initial, after incremental calculations were done, time was saved under end_incremental. The difference between start_incremental and end_incremental represents the time for a single iteration of incremental calculations

3. At all k iterations, the run time count was taken and saved under the

start_calculated_sequence_objects variable, right before the values that require inputs from both sequences were calculated. After these calculations are done run time was saved under the end_calculated_sequence_objects variable. The difference of these two represents the time for single iteration of calculations that require input from both sequences.

The difference between the start and end values were appended with the correlation coefficients and passed to the testing_pearson.py file for further processing and saved into .csv file.

It is important to mention that the sum of single initial, single incremental and calculate_sequence_objects times is not equal to overall execution time. Approximate overall execution time for any given set of inputs should be calculated as follows:

single initial time * number of sequences in input dataset (dataset_n_col)
+
single incremental time * number of sequences in input dataset (dataset_n_col) * k-1
+
single calculated_sequence_objects time * number of sequences in input dataset (dataset_n_col) squared * k.

The expectation was that time would occasionally vary between the runs with same parameters, thus each measurement was repeated 20 times and the average of the overall performance time was taken. All of these values were compiled into a data frame and exported in a csv file to be used for the experimental evaluation.

5 Experimental evaluation

Each algorithm implementation, tests, and run time measurements were implemented inside a different module. Functions used in more than one algorithm were collected in a separate file named common_functions.py. All separate algorithms were called from the testing_pearson.py module, which wraps algorithm functions into another that handles iterations and averaging of time measurements. These wrapped functions are then called from module call_testing_pearson.py, which reads user inputs, creates or imports dataset (depending on user inputs), passes necessary variables to wrapped functions in testing_pearson.py, and handles .csv export as well as visualisations if defined so by user input in command line. Figure 13 displays a visual representation of the project architecture.

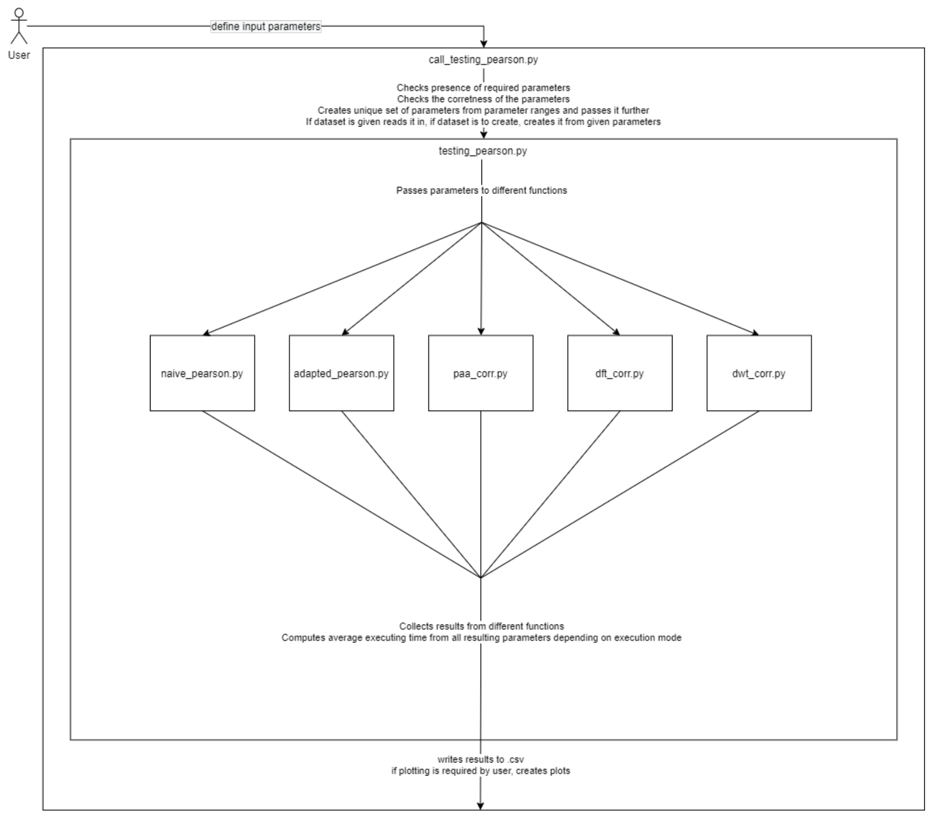
User
define input parameters
call_testing_pearson.py
Checks presence of required parameters
Checks the corretness of the parameters
Creates unique set of parameters from parameter ranges and passes it further
If dataset is given reads it in, if dataset is to create, creates it from given parameters
testing_pearson.py
Passes parameters to different functions
naive_pearson.py
adapted_pearson.py
paa_corr.py
dft_corr.py
dwt_corr.py
Collects results from different functions
Computes average executing time from all resulting parameters depending on execution mode
writes results to .csv
if plotting is required by user, creates plots

Figure 13: Integrated Project Schema

5.1 Command line initiation

All functions directly implementing the algorithms in Sections 2 and 3 require a set of inputs to be provided by the user. Thus a set of input parameters has to be explicitly declared from command line at the beginning of each simulation in order to establish initial values, what parameters are steady and/or variable, whether a data set is provided or has to be generated, and if plots need to be created:

- create_dataset: a boolean parameter - False if dataset is provided by the user, in this case only the path_to_file has to be specified. True if it has to be generated by the code and the following additional parameters need to specified[2]:

[2] "min", "step" and "max" for all parameters represent a range of values, which will be tested in combination with all other defined parameters and their ranges. Step parameter represents the increase step inside the defined range. In Python this logic would be expressed as range(min, max, step). If the a value obtained from the summation of a previous one with the step is bigger than the max value, it will be ignored

- n_min: an integer parameter - minimum number of records in dataset

  - n_step: an integer parameter - incremental number of records in dataset

  - n_max: an integer parameter - maximum number of records in dataset

  - m_min: an integer parameter - minimum number of sequences in dataset

  - m_step: an integer parameter - incremental number of sequences in dataset

  - m_max: an integer parameter - maximum number of sequences in dataset

- plotting: a boolean parameter - True if the user requires plots to be generated, False otherwise
- window_size_min: an integer parameter - minimum window size value
- window_size_step: an integer parameter - incremental window size value
- window_size_max: an integer parameter - maximum window size value
- step_size_min: an integer parameter - minimum step size value
- step_size_step: an integer parameter - incremental step size value
- step_size_max: an integer parameter - maximum step size value
- N_min: an integer parameter - minimum N value for PAA
- N_step: an integer parameter - incremental N value for PAA
- N_max: an integer parameter - maximum N value for PAA
- hierarchy_level_min: an integer parameter - minimum hierarchy level value for DWT
- hierarchy_level_step: an integer parameter - incremental hierarchy level value for DWT
- hierarchy_level_max: an integer parameter - maximum hierarchy level value for DWT
- coeff_num_min: an integer parameter - minimum coeff_num value for DFT
- coeff_num_step: an integer parameter - incremental coeff_num value for DFT
- coeff_num_max: an integer parameter - maximum coeff_num value for DFT
- threshold_min: a float parameter - minimum threshold value
- threshold_step: a float parameter - incremental threshold value
- threshold_max: a float parameter - maximum threshold value

- output_file_name: a string parameter - name of the output file without including extension[3]
- iterations: an integer parameter - number of measurement repetitions in order to get the averaged time results.[4]
- execution_mode: a string input - to define execution mode. ”full” mode computes the simulation in full, whereas ”initial and incremental” computes only the first iteration (k=1) in Listing 7 and averages all 3 types of time measurements obtained this way.[5] Execution mode was created for fast debugging and initial results exploration.

All input values have to be executed as parameters of call_testing_pearson.py file in command line: this script is the only file that needs to be called from command line. Examples of command line inputs are presented in Listings 8 and 9.

```
python call_testing_pearson.py create_dataset=False plotting=False
window_size_min=4 window_size_step=2 window_size_max=5
step_size_min=2 step_size_step=1 step_size_max=3 N_min=4
N_step=1 N_max=5 threshold_min=0.7 threshold_step=0.1
threshold_max=0.8 hierarchy_level_min=1 hierarchy_level_step=1
hierarchy_level_max=2 path_to_file="imported_file_path"
output_file_name="exported_file" iterations=20 coeff_num_min=0
coeff_num_step=1  coeff_num_max=10
execution_mode= "initial_and_incremental"
```

Listing 8: Example call command using ”initial_and_incremental” execution mode and no dataset creation

```
python call_testing_pearson.py create_dataset=True plotting=False
window_size_min=4 window_size_step=2 window_size_max=5
step_size_min=2 step_size_step=1 step_size_max=3 N_min=4 N_step=1
N_max=5 threshold_min=0.7 threshold_step=0.1 threshold_max=0.8
hierarchy_level_min=1 hierarchy_level_step=1 hierarchy_level_max=2
n_min=200 n_step=100 n_max=201 m_min=200 m_step=100 m_max=201
output_file_name="exported_file" iterations=20 coeff_num_min=0
coeff_num_step=1  coeff_num_max=10 execution_mode= "full"
```

Listing 9: Example call command using ”full” execution mode and dataset creation with required additional parameters

[3]extension of the output file is always .csv

[4]As stated in 4.3 our standard value here were 20 iterations (repetitions)

[5]The value for initial time is computed only once.
The value for incremental is computed a number of times proportional to the number of sequences in the dataset (dataset_n_col) and an average value is taken.
Finally, calculated_sequence_objects time is calculated dataset_n_col squared number of times and the final result is the average of calculated_sequence_objects.
Averaging is performed inside of testing_pearson.py script

window_size	step_size	N
1024	100	16

coeff_num	threshold	n
3	0.95	4000

m	hierarchy_level	time_naive
10	4	0.761

time_naive_initial	time_naive_incremental	time_naive_calculated_sequence_objects
87.610	2386.284	757434.164

time_adapted	time_adapted_initial	time_adapted_incremental
0.111	1690.664	11399.715

time_adapted_calculated_sequence_objects	time_PAA	time_PAA_initial
97331.595	0.126	3861.279

time_PAA_incremental	time_PAA_calculated_sequence_objects_adapted	time_DFT
113748.984	6462.629	0.691

time_DFT_initial	time_DFT_incremental	initial_dft_fun_time
19979.974	581882.695	16570.969

incremental_dft_fun_time	time_DFT_calculated_sequence_objects	time_DWT
481079.585	88498.879	0.316

time_DWT_initial	time_DWT_incremental	time_DWT_calculated_sequence_objects
9856.154	290292.349	14496.985

Table 2: Example output csv table

The input values order is not fixed and can be re-arranged. Once these values are entered, call_testing_pearson.py calculates the results and outputs a csv file with time measurements for each algorithm. An example of a csv output file with sample values for one row is shown in Table 2:

These results are used to create visualizations and draw conclusions regarding the performance of each algorithm.

5.2 Results

Depending on parameters values, the algorithms show different behaviors. Therefore, various analyses were performed by investigating different parameters with respect to increasing numbers in their values. In some cases, all algorithms were affected by the changing parameters whereas in others only one approach was. In the latter case, "sweet spots" were extracted for each analysis. The expectation is that for an increasing number of sequences and records[6] in dataset as well as for increasing window size, and step size[7] all algorithms will increase in execution time as these parameters are present in all of their time complexities. On the other hand, for an increasing threshold, only dimensionality reduction methods will be subject to change. In fact, we expect the execution time to decrease with the increase of threshold value, since more sequence combinations will be "filtered out" before the exact correlation is calculated; thus saving computation time. Finally, for N, coeff_num, and hierarchy level, only the respective algorithms that include these parameters will show a difference in behavior. For these parameters we expect the time to initially decrease as these variables increase in values. This phenomenon is due to the fact that higher values help filtering out correlations ahead of exact calculations thanks to the threshold. However with an increase of these parameters, mathematical overhead to calculate required variables for the filtering increases as well. Our consequent expectation is that the negative effects will offset the time savings achieved by the filtering; thus, "sweet spots" are expected.

5.2.1 Increasing number of sequences (m)

An increasing number of sequences leads to a higher run time. The slowest algorithm is the Naive Pearson, followed by DFT, Adapted Pearson, DWT, and PAA. Their behavioral pattern resembles an exponential pattern as illustrated in Figure 14.

[6]Important to note, that we expect an increase in execution time for increased number of records only in our simulation setting for overall execution time in "full" mode. In a real life scenario, setting the number of records should not play a role. The reason for this is because records are screened by the medical equipment and entered into the system immediately

[7]we expect step size to have a negative effect on execution time in a real life setting as a bigger step size requires more adjustments and mathematical calculations. However in an experimental setting overall time might decrease, since the increase in the step size reduces the number of iterations needed to slide the window over the complete dataset. Thus, this effect should have a positive impact on the overall execution time

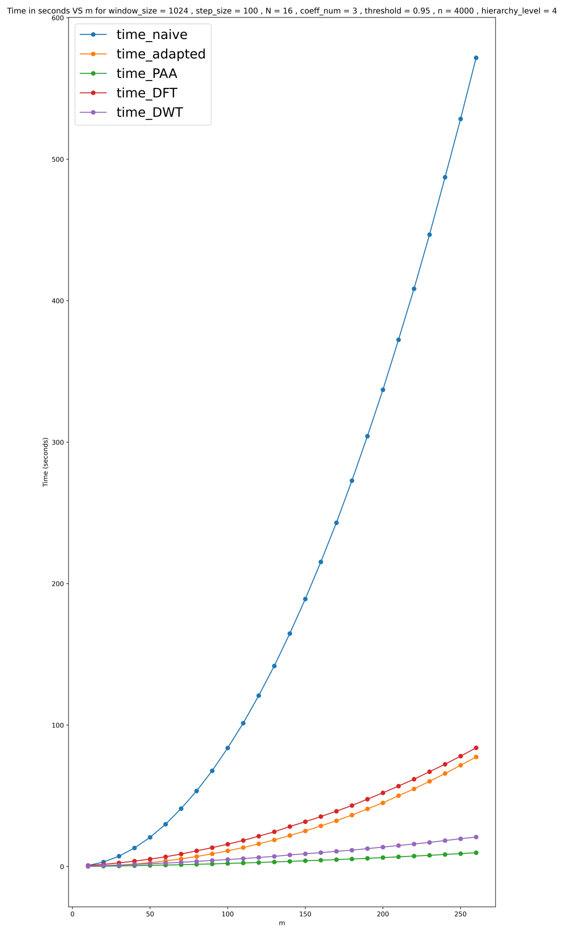

Figure 14: Increasing number of sequences

There is a solid difference in performance between the Naive Pearson and the other algorithms, where the former is noticeably slower. This is due to the fact that it is heavily dependent on the number of sequences, which can also be observed from its time complexity. Since Adapted Pearson has the same complexity as Naive Pearson, one may expect them to have comparable performance; however, due to the independency of the sequences in Adapted Pearson, this algorithm performs far better. In fact, the remaining four behave similarly in pairs: DFT and Adapted display close values, likewise DWT and PAA. It is worth noting that DFT results in this plot are not competitive with the other two dimensionality reduction methods; this behavior is also present in other plots and will be explained in Section 5.2.8.

5.2.2 Increasing number of records (n)

An increasing number of records leads to a higher run time. The slowest algorithm is the Naive Pearson, followed by DFT, Adapted Pearson, DWT, and PAA. Figure 15 shows the resulting plot:

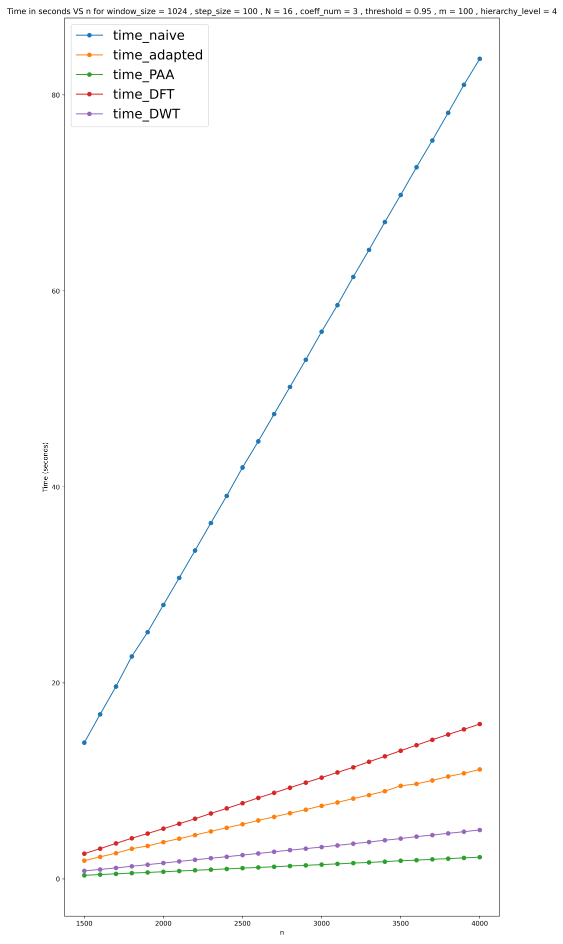

Figure 15: Increasing number of records

The algorithms follow an incremental linear pattern with Naive significantly diverging from the others. The reason why Adapted Pearson is not as affected as Naive is the same as described in the comments to Figure 14.

5.2.3 Increasing window size

An increasing window size leads to a higher run time in all algorithms. The algorithm that is the most affected by the increasing of this factor is Naive Pearson. The others' performance is interchangeable. With a larger window size, PAA performs better thanks to the N factor, whereas DWT is affected by the hierarchy levels which can improve or worsen the performance based on the window size. DFT behavior will be further analysed in the coeff_num section. Figure 16 shows the resulting graph on a small scale:

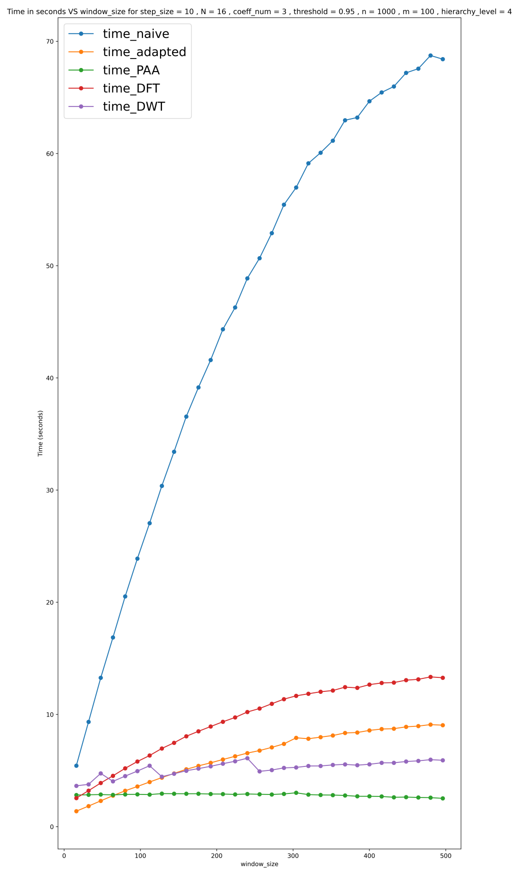

Figure 16: Incremental window size (smaller window sizes)

On a bigger window size scale, the algorithm behaviors are even more defined. The slowest is Naive Pearson, followed by DFT, Adapted Pearson, DWT, and finally PAA. Compared to Naive Pearson, the remaining algorithms' change in performance is not incredibly noticeable. These observations can be viewed in Figure 17:

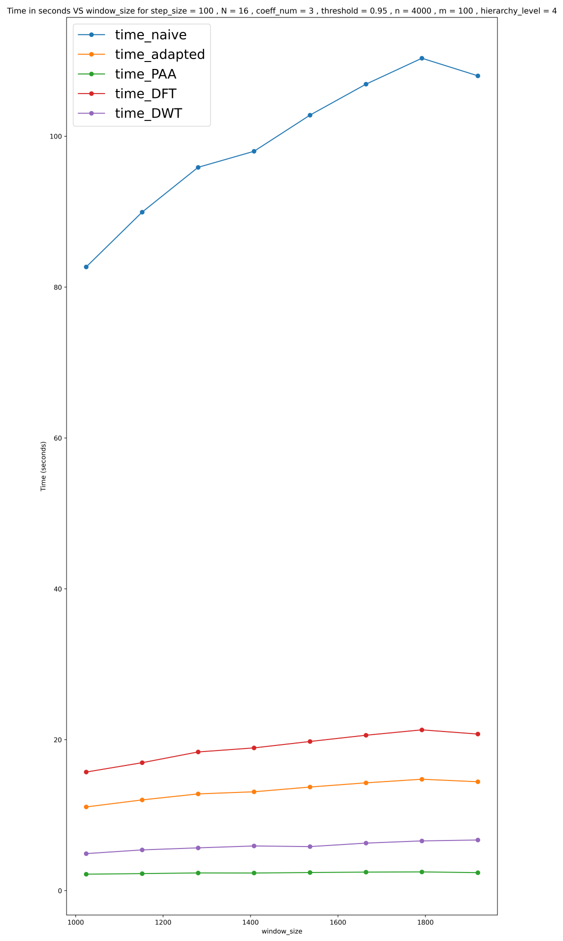

Figure 17: Increasing window size

Within each window size, it was also important to analyse the impact of an increasing step size as it is an essential component of the sliding window design.

5.2.4 Increasing step size

A higher step size reduces the number of computations needed and hence decreases the overall run time. Looking at the time complexities, a higher step size will asymptotically approach window size and thus reduce the complexity. The slowest algorithm is Naive Pearson, followed by DFT, Adapted Pearson, DWT, and finally PAA. The algorithms behaviors are asymptotically decremental as shown in Figure 18:

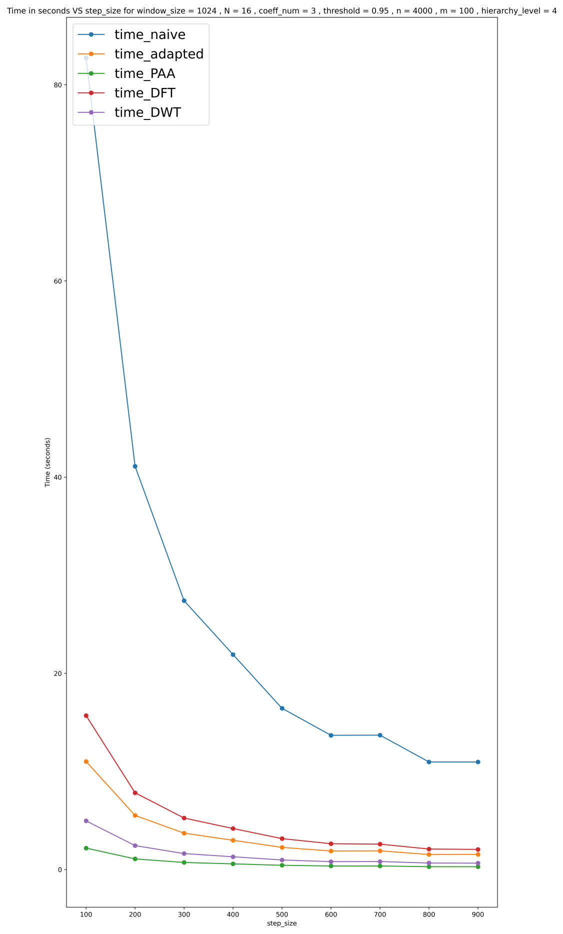

Figure 18: Increasing step size

This result matches our expectation regarding the impact of step_size on execution time in "full" mode. However, to test the impact in real life scenario, further analyses were performed. Because of scales differences, Naive and Adapted Pearson were separated from the dimensionality reduction methods.

With increasing step size, incremental time for Adapted Pearson will increase and thus be more affected compared to Naive Pearson as demonstrated in Figure 19:

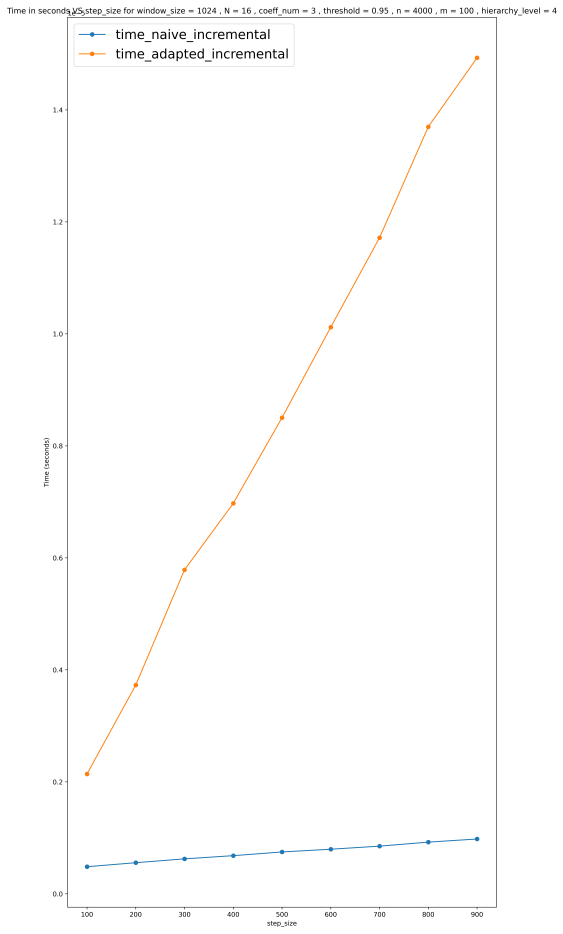

Figure 19: Increasing incremental step size for Naive and Adapted Pearson

DFT performs the slowest out of the three dimensionality reduction techniques, followed by DWT and PAA. However, within their run time values, their behavior is linear. Figure 20 shows the output graph:

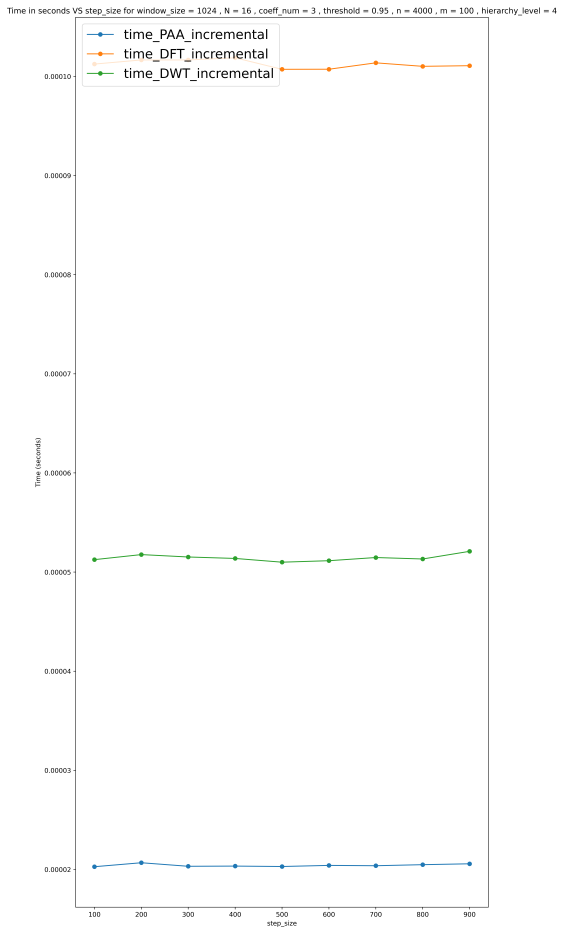

Figure 20: Increasing incremental step size for dimensionality reduction techniques

This analysis shows that, even though Naive Pearson is the slowest performing algorithm in regards to overall time, Adapted Pearson is subject to the most effect with respect to incremental time, which is the most relevant metric for real world scenarios. This is due to its higher dependency from step size relative to window size. On the other hand, dimensionality reduction methods show a somewhat steady behavior. However, the situation became inverted when the effect of increasing threshold values was examined.

5.2.5 Increasing threshold

Dimensionality reduction techniques perform efficiently if the threshold is high enough to filter out part of the data. Naive and Adapted Pearson are not affected by an increasing threshold value as it is not part of their algorithms. For the remaining algorithms, a low threshold is not useful as almost (or all) Euclidean distance values will pass the condition. This fact leads to a higher computational time than Adapted Pearson as the threshold check has to be performed in addition to the Pearson correlation calculations. In other words, as threshold values are low, filtering will not play such a big role, leading to slower performances. With higher threshold values, all dimensionality reduction techniques decrease with respect to their computation time. In fact, as the threshold value approaches its maximum (1), PAA, DFT, and DWT all eventually fall below Adapted Pearson. Going into more details, PAA filters values out more efficiently than the other two[8]. For the same reason, DWT performs better than DFT. These observations can be confirmed by looking at Figure 21.

The next sections 5.2.6, 5.2.7 and 5.2.8 explore specific dimensionality reduction parameters. It will be observed that in all of them the increasing of each parameter does not have a direct relationship with the increasing and/or decreasing in performance of the algorithm, rather it displays sweet spots. The reasoning behind this pattern is related to the fact that initially threshold helps speeding up the process; however, with higher values, the threshold check is not useful anymore and increases computational time as the overhead from dimensionality reduction becomes bigger. The individual components for each method that were examined are: N, hierarchy level, and coeff_num.

[8]Unit test results produced with the example inputs: create_dataset=True plotting=False window_size_min=1024 window_size_step=128 window_size_max=2048 step_size_min=100 step_size_step=10 step_size_max=101 N_min=16 N_step=2 N_max=17 threshold_min=0.8 threshold_step=0.05 threshold_max=1 hierarchy_level_min=4 hierarchy_level_step=1 hierarchy_level_max=5 n_min=4000 n_step=100 n_max=4001 m_min=100 m_step=100 m_max=101 coeff_num_min=3 coeff_num_step=1 coeff_num_max=4 iterations=20 execution_mode=full output_file_name="unit_test_results" show that out of first 1024650 calculations, PAA correctly filtered out 982778 calculations (95,9%), DFT - 393091 (38,3%) and DWT - 880961 (86%)

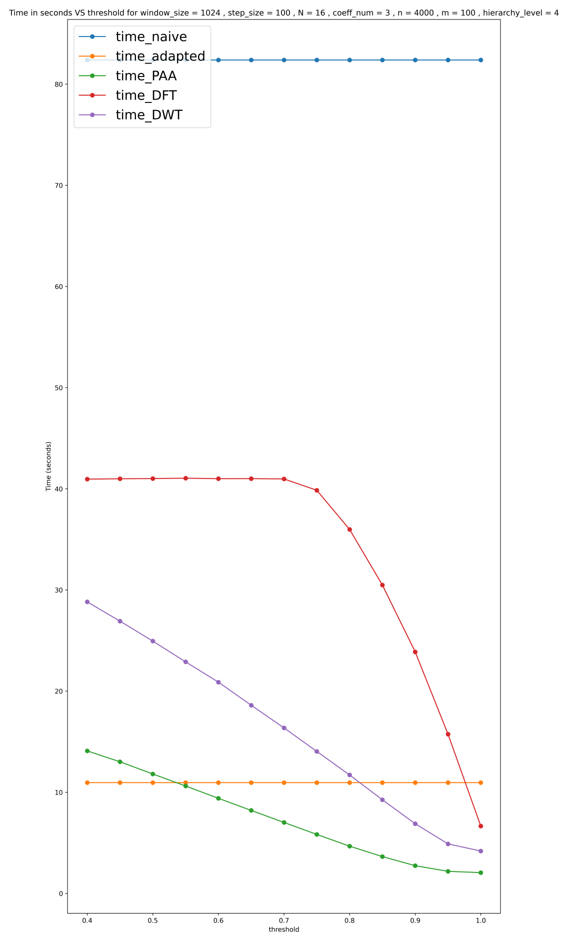

Figure 21: Increasing threshold

5.2.6 Increasing N

The N parameter is only present in the PAA calculations. Naive Pearson, Adapted Pearson, DFT, and DWT are not affected as N is not part of their algorithms and time complexities. More details regarding PAA behavior with increasing N can be viewed in Figure 22:

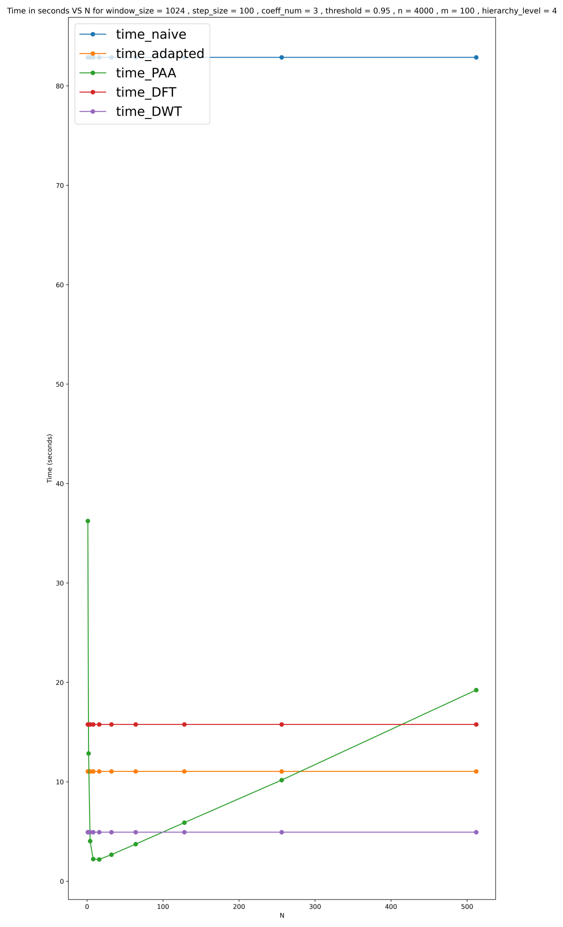

Figure 22: Increasing N

PAA behavior changes with respect to N, first decreasing and then increasing. The sweet spot is found around 10. This leads to the conclusion that N is neither consistently directly nor inversely proportional to the PAA run time. Consequently, choosing too low or too high of a N value will not lead to an optimal performance. The second best performing dimensionality reduction method in our work was DWT, where hierarchy level played a pivoting role.

5.2.7 Increasing hierarchy level

The hierarchy level parameter is only present in the DWT calculations. Naive Pearson, Adapted Pearson, PAA, and DFT are not affected as hierarchy level is not part of their algorithms and time complexities. More details regarding DWT behavior with increasing hierarchy level can be viewed in Figure 23:

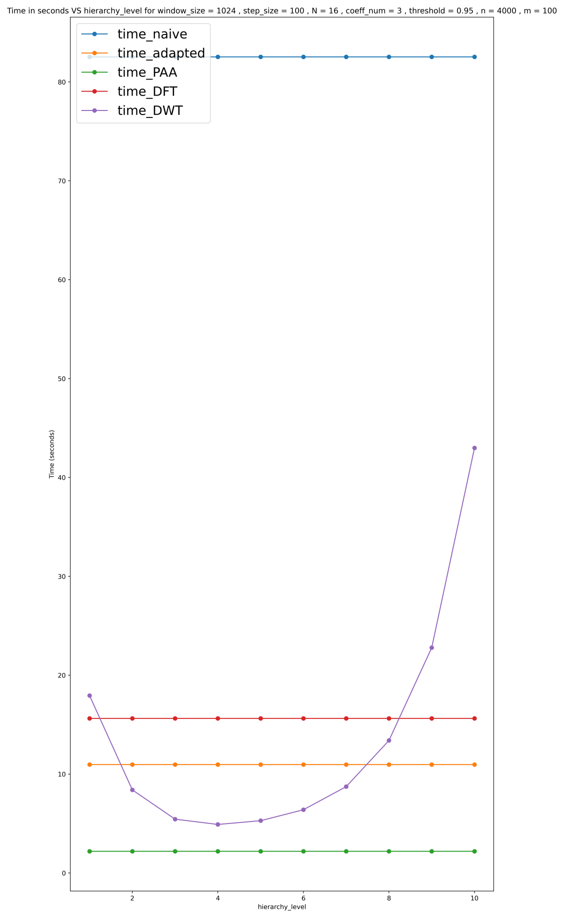

Figure 23: Increasing hierarchy level

DWT behavior changes with respect to the hierarchy level, first decreasing and then increasing. The sweet spot is found around 4. This leads to the conclusion that the hierarchy level is neither consistently directly nor inversely proportional to the DWT run time. Consequently, choosing too low or too high of a hierarchy level value will not lead to an optimal performance.

The slowest of the three dimensionality reduction techniques in our project was DFT. Our explanation behind this phenomenon will be explained in the next section, where the impact of coeff_num is examined.

5.2.8 Increasing coeff_num

The coeff_num parameter is only present in the DFT calculations. Naive Pearson, Adapted Pearson, PAA, and DWT are not affected as coeff_num is not part of their algorithms and time complexities. More details regarding DFT behavior with increasing coeff_num can be viewed in Figure 24:

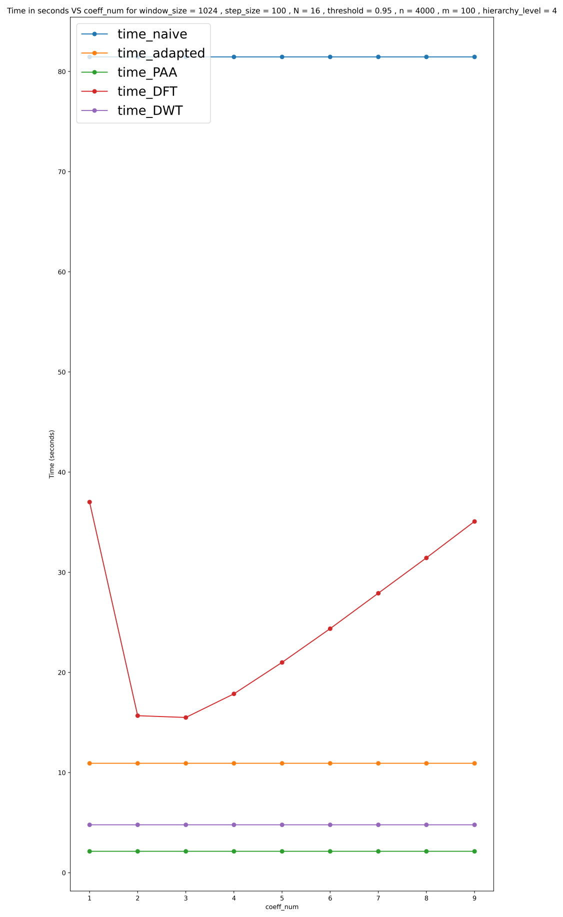

Figure 24: Increasing Coeff_num

DFT behavior changes with respect to coeff_num, first decreasing and then increasing. The sweet spot is found between 2 and 3. This leads to the conclusion that coeff_num is neither consistently directly nor inversely proportional to the DFT run time. Consequently, choosing too low or too high of a coeff_num value will not lead to an optimal performance. It is important to note that, even with the most efficient values, we still have high degree of false positives.

Further analyses were performed to deeper investigate DFT behavior. It can be observed in Figure 25 that initial and incremental times of specific DFT function execution comprises a substantial amount (on average 59%) of the overall initial and incremental execution time. The output values of these two, in our opinion, are largely due to the run time used to process the DFT function, whereas calculating the sequence objects consumes much less.

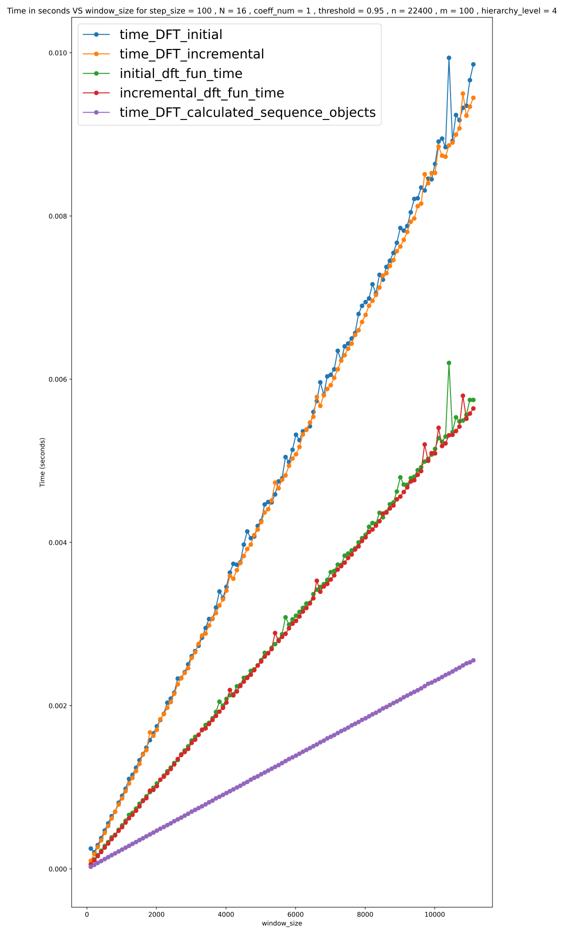

Figure 25: Increasing Coeff_num

Going into more details in regards to what portions of the DFT function are more expensive, several parameters were investigated and shown in Figure 26.

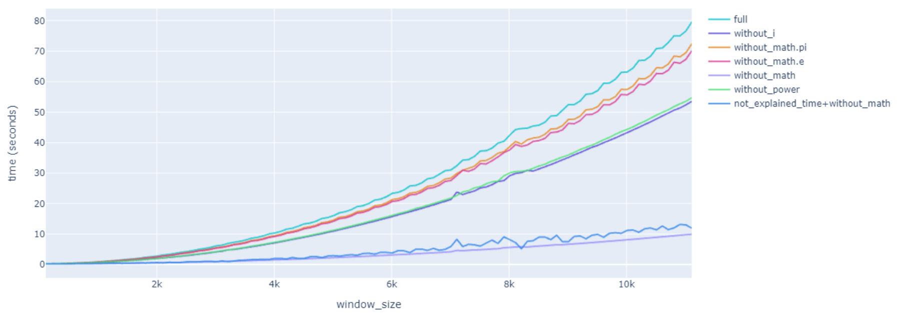

Figure 26: Increasing Coeff_num

The legend labels represent the removal of individual components of Equation (17):

- full is the time when all equation components are included.
- without i: removal of i from equation
  $\sum$ x$_i$ += dataset[k]*math.e**((-2 * math.pi * f * k) / window_size)
- without math.pi: removal of pi from equation
  $\sum$ x$_i$ += dataset[k]*math.e**((-2 * i * f * k) / window_size)
- without math.e: removal of e from equation
  $\sum$ x$_i$ += dataset[k]*1**((-2 * math.pi * i * f * k) / window_size)
- without math: every mathematical operation is removed
  $\sum$ x$_i$ += dataset[k]
- without power: power is replaced with multiplication
  $\sum$ x$_i$ += dataset[k]*math.e*((-2 * math.pi * i * f * k) / window_size)
- not explained time: is just full - all of the other times that are included in the computation.

The analysis shows "not explained time" and "without math" are almost identical. Consequently, removing only one of the mathematical operations does not drastically affect the performance, however their comprehensive need indeed makes the function slower. Power function and irrational number component also cause delays in performance, singularly having more effect than only one of the previously mentioned parameters. The combined time from power (full - without power) and i (full - without i) takes on average 63% of total DFT function execution time. Power and i take 37.17% (0.63*0.59) of the whole execution time of DFT. If we adjust DFT in Figure 24 to consume 37.17% less time, its performance line would be placed just under the Adapted Pearson line.

Aside from these parameters, DFT performs below expectations because of the methods used to implement it. In fact, when checking for the threshold parameter, coeff_num is used instead of window size, which can lead to the presence of false negatives. In addition, the current implementation passes a critical amount of false positives which would need a second round of checking. These and other additional edits (including adaptation of the code to remove the need for sequence normalization) as described in [ZS02] would make DFT competitive with the other two dimensionality reduction methods.

5.3 Unit Testing

Some basic tests were implemented to check validity of the data, pass only the appropriate values and pinpoint discrepancies in the results. In case that the algorithms had bugs or were not performing as expected, error messages were outputted. The code for these tests can be found in the unit_test.py file.

6 Conclusions and Future Work

Intensive Care Unit (ICU) patients require real-time fast and sophisticated analysis of their condition. For this reason, it is paramount to compare patient data across time series in a fast and reliable fashion. To achieve this goal, pairwise Pearson Correlation coefficients computations can be used. In this project, we investigated different methods to perform this calculation: Naive Pearson, Adapted Pearson, PAA, DFT, and DWT. We performed several analyses to compare run times and performance across all five methods by implementing them in Python.

Our results showed that algorithms may behave differently based on which values are changed:

- Increasing number of sequences (m): all algorithms show exponential increase
- Increasing number of records (n): all algorithms show linear increase
- Increasing window size: all algorithms show an increase in run time
- Increasing step size: all algorithms show an asymptotically decrease in run time. However if looking at real life scenarios when only incremental time is applicable, time of dimensionality reduction techniques stays stable, whereas for Naive Pearson and Adapted Pearson rises.
- Increasing threshold: dimensionality reduction methods perform better with a higher threshold value
- PAA - Increasing N: sweet spot found around 10 with our parameters set. Different parameters will lead to a different sweet spot value.

- DFT - Increasing coeff_num: sweet spot found between 2 and 3 with our parameters set. Different parameters will lead to a different sweet spot value.
- DWT - Increasing hierarchy level: sweet spot found around 4 with our parameters set. Different parameters will lead to a different sweet spot value.

For the future, the DFT implementation in the code should be optimized such that it is competitive with the other two dimensionality reduction methods. For Adapted Pearson, it would be beneficial to also show only one change for one incremental window in the visualizations. Furthermore, we believe that real-life implementations should be implemented in one of the lower level languages (e.g. C) to perhaps further improve general run time and performance (even though relationships among algorithms will remain the same). Additional nice-to-haves would be to implement a hyper-parameters choice strategies (e.g. grid search) to automate the process of selecting the most optimal parameters (e.g. the sweet spots) for the inputs as well as a User Interface so that users can perform their own analyses interactively and without having to run a command line based computer program.

References

[AML10] Abdullah Mueen S. N., Liu J.: Fast approximate correlation for massive time-series data. *ResearchGate* (2010). URL: `https://www.researchgate.net/publication/221214441_Fast_approximate_correlation_for_massive_time-series_data`.

[EKM01] Eamonn Keogh Kaushik Chakrabarti M. P., Mehrotra S.: Dimensionality reduction for fast similarity search in large time series databases. *springer.com* (2001). URL: `https://link.springer.com/article/10.1007/PL00011669`.

[pCcF99] pong Chan K., chee Fu A. W.: Efficient time series matching by wavelets. *semanticscholar.org* (1999). URL: `https://www.semanticscholar.org/paper/Efficient-time-series-matching-by-wavelets-Chan-Fu/c01f6db7dd67125d69a4fda7c2a79efc191777fe`.

[ZS02] Zhu Y., Shasha D.: Statstream: Statistical monitoring of thousands of data streams in real time. *ScienceDirect* (2002). URL: `http://www.vldb.org/conf/2002/S10P04.pdf`.