DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING,

UNIVERSITY OF STAVANGER, NORWAY

Comprehensive Python Module for Computing and Visualizing Dynamic Time Warping Alignment: DTWPy

Hafiz Muhammad Gulzar

hmgulzar88@gmail.com

Student no. 228839

15-06-2015

This thesis is dedicated to my parents.

For their endless love, support and encouragement.

Acknowledgement

I would like to express my special appreciation and thanks to my supervisor Associate Professor Dr. Tomasz W. Wlodarczyk, you have been a tremendous mentor for me. I would like to thank you for encouraging my research and for allowing me to grow as a Computer Science graduate.

Thank you,

Hafiz Muhammad Gulzar

Table of Contents

List of figures

List of Tables

Abstract

Dynamic Time Warping (DTW) is a well-known technique used to determine alignment between two temporal sequences. DTW has been used in wide range of applications and it can be applied on any data which can be represented as linear sequence. Existing DTW libraries have out dated implementation of core DTW algorithm, which result in low performance or are inapplicable for big sequences. The aim of this thesis is to present, a comprehensive DTW library, encapsulating t implementation of DTW variants with recently proposed efficient algorithms. In this thesis I presented a python module for computing and visualizing DTW alignment: DTWPy. DTWPy has implementation of classical DTW, almost all the DTW variants proposed in literature to date, with recently proposed performance efficient algorithms i.e. fastDTW and DDTW. Algorithms implemented in DTWPy are tested thoroughly and give correct results. Correctness is verified by comparing it with existing R implementation. Furthermore, I have compared algorithms implemented, and performance of DTWPy with existing libraries. DTWPy have the most comprehensive implementation of DTW algorithms present in literature to date, and is applicable on large temporal sequence. The architecture of DTWPy is designed to be flexible, to scale and accommodate possible future improvements.

Chapter 1: Introduction

Dynamic time warping (DTW) is an approach used to determine the similarity between two time series by shrinking or expanding the selected time series. DTW [1] was introduced in 1960s, which gain its popularity when it was further explored in 1970s under the umbrella of speech recognition [2]. After that, improvements has been made on DTW and it has been used widely in many computing applications of different domains like signature matching, gesture recognition, data mining, computer vision, surveillance, chemical engineering, protein sequence alignment [3] , human motion recognition [4], and in word recognition [5].

Existing DTW libraries cover only a limited subset of the DTW spectrum, typically providing just the most basic functionality. A few libraries that cover most of step patterns and windowing functions contain outdated implementation of the core algorithm, which leads to reduced performance. R and Pandas[6] are the two most common tools used by data analysts. R provides one comprehensive DTW library, though it does not yet contain the latest performance improvements made possibly by current research. Pandas and Python in general, have multiple minor and incomplete DTW implementations. There is clearly a need for such a comprehensive library which contains performance improvements made by latest research work with all the DTW variants, DTW algorithm vary on the basis of step patterns, windowing functions and improvements made by research work.

Step patterns also referred as dynamic programming (DP) formulation and local constraints in literature. Local constraints determine the contribution of adjacent cell to determine the warping path between temporal sequences, whereas windowing functions limit overall search space to calculate DTW alignment. In this thesis I presented a comprehensive DTW module DTWPy, for computing and visualizing DTW alignments. DTWPy has Implemented classical DTW [2], with latest efficient core DTW algorithms i.e. fastDTW [7] and DDTW [8]. DTWPy is fully equipped with almost all the step patterns proposed in literature to date, and most commonly used windowing functions. DTWPy has implementation of step patterns classified by Sakoe-Chiba [2], Rabiner-Myers [5], with well know commonly used step patterns e.g. symmetric1 and symmetric2. In windowing functions DTWPy implemented Itakura parallelogram [9] SCband [2] and Paliwal window[10].

Moreover, I have tested the correctness, compared performance and DTW variants covered by DTWPy with R package [11] and mlpy[12]. From the results it can be concluded that Algorithms implemented by DTWPy, determine optimal alignment between two temporal sequences and DTWPy is the most comprehensive implementation of core DTW algorithms and DTW variants to date. The architecture of DTWPy is designed flexible so that possible future research work improvements can be incorporated.

1.1 Organization of this thesis

Chapter 2 gives literature review in detail, classification of DTW algorithms, constraints which have to follow in order to get optimal warping path between two time series. In addition to that, it describes classical DTW and latest research work done in DTW domain to improve complexity of DTW algorithms. In chapter 3, I discussed the usage of DTWpy, algorithms, step patterns, windowing functions, and alignment visualization which are implemented by DTWPy. Chapter 4 describes the correctness of DTWPy and compares performance with R package. Finally in Chapter 5 conclusion will be summarized and significance of DTW is evaluated.

Chapter 2: Literature Review

Dynamic Time warping is an algorithm, used to calculate the alignment between two temporal sequences (query and a template). In the beginning it was only used for speech recognition. Speech recognition typically referred to conversion of spoken words to textual words. In speech recognition audio data is converted into templates, and to run the actual temporal signal against each template by applying some constraints, to find the best match. Best match template always have the minimum distance measure from query (input). The use of DTW is not limited to speech recognition; in fact it can be applied on any type of temporal data, which can be represented in linear sequence. Initially proposed DTW algorithm (referred as classical DTW in this thesis) has quadratic time and space complexity. Let's understand of classical DTW algorithm in detail.

2.1 Classical DTW Algorithm

Classical DTW has quadratic time and space complexity and is explained eminently in [8][13][3][7][14] literature. As mentioned earlier, DTW can be applied on any kind of temporal data which can be presented in linear sequence, which can vary in time and speed. To understand DTW let's assume we have two time series X and Y of length n and m respectively, where:

$$X = x_1, x_2, x_3, x_4, \dots, x_n \tag{1}$$

$$Y = y_1, y_2, y_3, y_4, \dots, y_m \tag{2}$$

Classical DTW uses dynamic programming approach to find the alignment between two time series which align the time series based on minimized distance. Dynamic Programming (also referred as DP) is a powerful approach which divides the big problem into smaller sub-problems. The result of smaller sub-problems is calculated and then aggregated to calculate the solution of the actual problem. Due to this property it is also known as *divide and conquer* approach, and *remember your past,* since result of sub-problem contributes to solve original problem. It can be achieved by using two different approaches i.e. top down and bottom up. In the top down approach, problem is solved by breaking it down into sub-problems. Sub-problems are then solved independently, and result of each sub problem is stored. This eventually contributed towards overall solution. On the other hand DTW is a step by step, bottom up approach; where the result of sub-problem is used to solve the given problem progressively. For example in DTW sub-problems $D(i, j)$ is solved first as shown in equation 7, and used progressively to calculate solution for $D(n, m)$ as shown in equation 3.

The first step to calculate DTW alignment between two time series, is to construct an n-by-m cost matrix where each (i^{th}, j^{th}) element corresponds to distance measured between x_i and y_j. Distance can be measure in by using different distance metrics e.g. simple Manhattan difference

$d(x_i, y_j) = |x_i - y_j|$, squared distance $d(x_i, y_j) = (x_i, y_j)^2$ or Euclidian distance, or any other[1] distance measure which describes the alignment between the time series. Based on cumulative distance for each path in the cost matrix, best match between time series can be found by using Euclidian distance which is one of the most common methods for distance calculation in DTW as shown in the following equation 3:

$$DTW(X, Y) = min\left\{\sum_{k=1}^{k} d(w_k)\right. \tag{3}$$

Where d the selected distance measure between two time series, and w_k is the cost matrix element that will be the k^{th} element of warping path W as shown in equation 4

$$W = w_1, w_2, w_3, w_4, \dots, w_k \tag{4}$$

The warping path can be found using dynamic programming formulation (also known as step patterns, local constraints), which determine the contribution of neighboring cells in the cost matrix to fill cell $D(i,j)$ as given in equation 7. DP formulation can be symmetric and asymmetric. In asymmetric formulation one of the points around the diagonal i.e. $D(i-1,j)$ or $D(i,j-1)$ is skipped or given more weight. Table 1 contains various examples of different symmetric and asymmetric DP algorithms used in DTW. However equation 7 can be termed as a symmetric formulation, since both points around the diagonal of the current point, are used with equal weights.

Research experiments reveals that symmetric formulation gives better results as compared to asymmetric in speech recognition [14]. A DP formulation e.g. equation 7, in this context, gives the cumulative distance for each point by taking sum of distance with the minimum of cumulative distances of the adjacent diagonal points. It fills the global cost matrix D by first filling the first column and first row of the matrix, in the following fashion as shown in equation 5,6 and 7 by initializing D(0,0) = 0

$$D(i,1) = D(i-1,1) + d(i,1) \tag{5}$$
$$D(1,j) = D(1,j-1) + d(1,j) \tag{6}$$
$$D(i,j) = d(i,j) + min[D(i-1,j), D(i-1,j-1), D(i,j-1)] \tag{7}$$

Once the global cost matrix has been filled with accumulated distances, the next step is to find the warping path between time series by using cost matrix. The warping path can found easily with the help of greedy approach, by backtracking the cost matrix. A greedy approach in algorithm theory makes the local optimal decisions, at the current position with assumption that it will give global optimal at the end. The warp path search starts from $D(n,m)$ and backtracks with the evaluation of all adjacent cells from left, down, diagonally to the bottom left. If any of

[1]Akila and Chandra[15] contains the review of different distance functions which can be used in DTW.

these adjacent cell contain the minimum values, it is then added to the start of the warping path until $D(1,1)$ is reached.

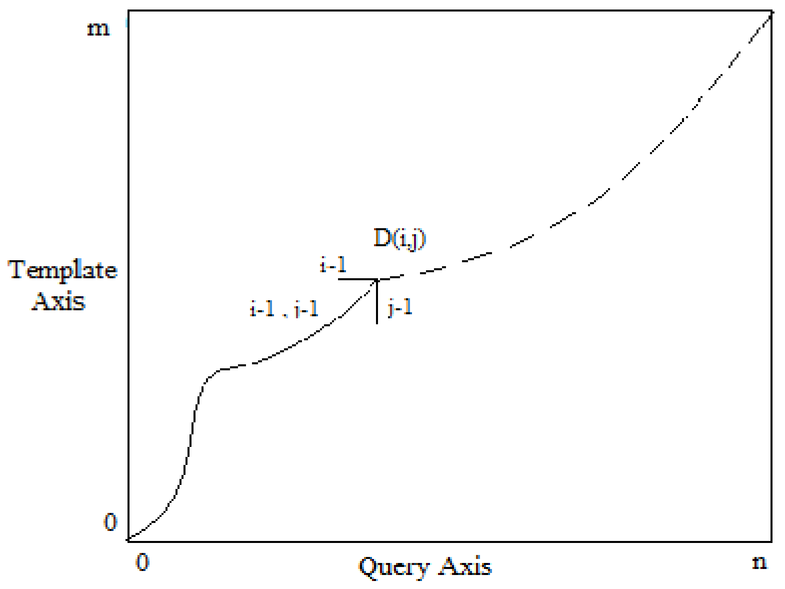


Figure 1: Dynamic Time Warping

Figure 1 shows the execution of DTW to calculate warping path between query and a template time series. DTW algorithm starts from point $D(0,0)$ till the end at the highest point i.e. $D(n,m)$. At any point $D(i,j)$, cumulative distance for each point is calculated by taking sum of distance $d(i,j)$ with the minimum of cumulative distances of all the successor points as shown in equation 7. However the number and contribution of successor points differ with respect to each DP formulation, Table 1, 2 and 3 has almost all the DP formulations also known as step patterns; I will further discuss step patterns in section 2.1.1.4 and 2.1.15 in detail.

For simplicity let's assume a symmetric step pattern, where all the adjacent points participate on equal basis. One DP formulation is evaluated for all the point, and cost matrix is filled with accumulated distance measures. It is now possible to find optimal warping path by backtracking from the $point(n,m)$, figure 1 shows possible warping path which can be obtained after backtracking the cost matrix. From the careful observations over optimal warping path obtained from the cost matrix, few improvements has been suggested in the literature and referred as constraints.

2.1.1 Constraints

In order to find optimal warping path, from all possible warping paths few constraints must be satisfied, while running the DTW algorithm. Constraints not only reduce the search space for warping path but also increase the performance of the algorithm. Constraints can be divided into two categories, local constraints and global constraints. Local constraints take care of the slope when a step is made to in local path; hence contribute towards calculating the accurate path. On the other hand global constraints reduce the search space for warping path, and improve overall efficiency of DTW algorithm. All the global and local constraints are described as follow:

2.1.1.1 Boundary constraint

Boundary condition focuses on head and tail of the warping path. It states that the first point of the warping point must be $w_1 = (1,1)$ and the last point should be $w_k = (n, m)$, where n and m represent the length of query and template time series respectively. The warping paths which do not satisfy boundary conditions are often referred as incorrect paths.

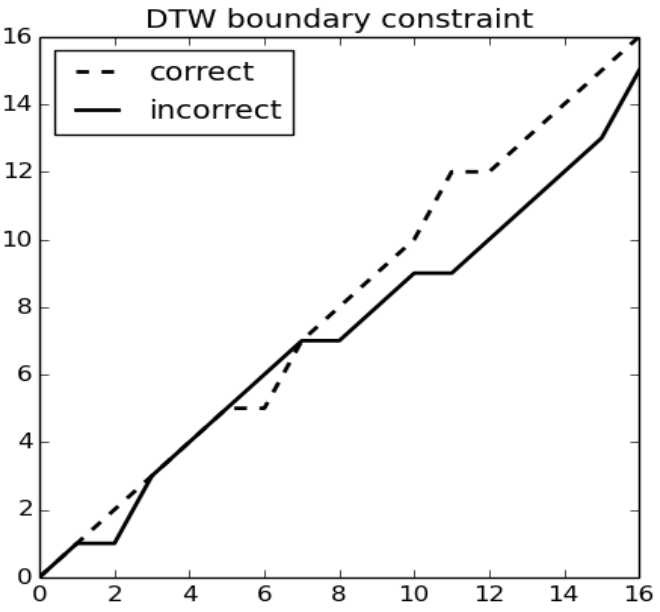


Figure 2: Boundary condition

Graphical representation of boundary constraint is shown in figure 2, where dashed line starts from the first cell of the cost matrix and end on the last cell $w_k = (n, m)$. On the other hand solid line starts from initial cell but ends at $w_k = (n, m - 1)$, which violate the boundary constraint, hence can be termed as incorrect warping path.

2.1.1.2 Continuity constraint

Continuity constraint makes sure that, a valid warping path must be continues. In other words it makes sure the contribution of every point in both template and query sequences. It can also be stated as; each cell in the cost matrix must the restricted to its neighboring cells, or the previous point of any point in the cost matrix should be $(i_k - 1, j_k), (i_k, j_k - 1), (i_k - 1, j_k - 1)$.

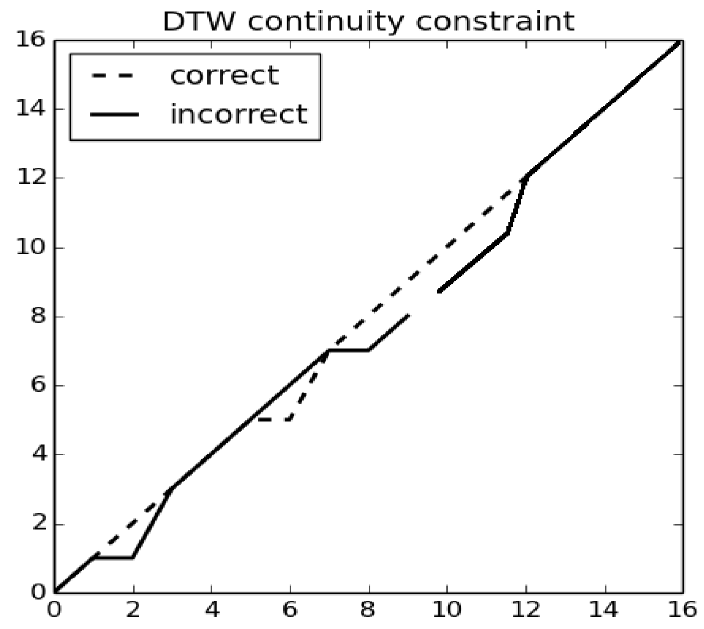


Figure 3: Continuty constraint

Figure 3 elaborates continuity constraint, it can be seen that path represented by solid line is fulfilling the boundary constraints but defying the continuity constraint. It is vital that an optimal path must obey all the constraints at once.

2.1.1.3 Monotonicity constraint

A valid warping path must be continues, have valid boundary points and must be monotonic in nature. Monotonicity constraint enforces that; the points of warping path must have increasing trend. It can be stated as a warping path cannot decrease in time, it can be straight or can increase i.e. for every $i_k \geq i_k - 1$ and $j_k \geq j_k - 1$.

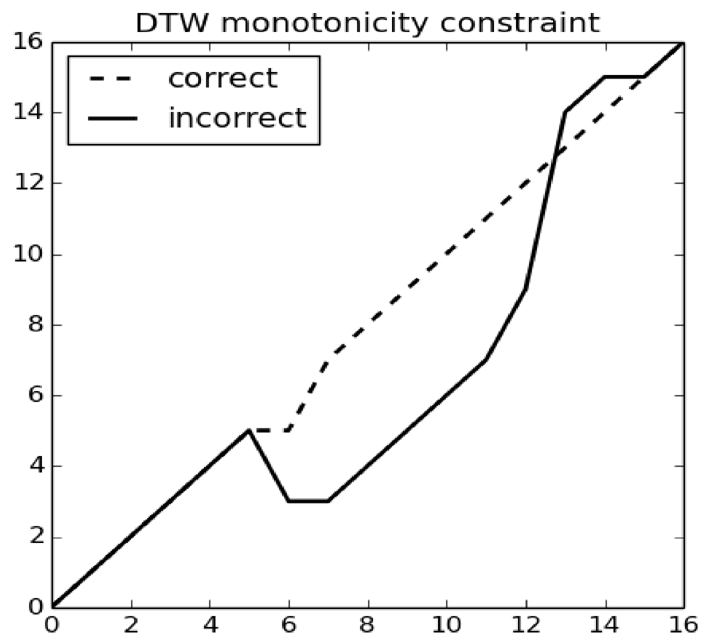


Figure 4: Monotonicity constraint

The warping path represented by solid line in the figure 4, fulfill all other DTW constraint but has decreases in time for a moment, which is enough to disregard. A valid warping path must fulfill all three above mentioned constraints i.e. boundary constraint, continuity constraint and monotonicity constraints. All three together, define the validity of warping path.

However there exist some other constraints but they are used either to increase overall performance of DTW algorithm, or define the contribution of adjacent cells. Let's discuss how constraints contribute to improve time and space complexity of the DTW algorithm.

2.1.1.4 Slope weighting

In order to determine the optimal path it is necessary to apply certain local continuity constraints on warping path, as mentioned above in continuity constraint, contribution of every legal point is vital. Itakura [9] proposed four types of weighting function which can be applied on the arc to make it biased towards the diagonal, further Sakoe-Chiba [2] suggested normalization function to put equal weight on all segments of local path.

To understand slope weighting let's update equation 7 to equation 8 (shown below) with the addition of slope weight ϖ, where ϖ is a positive real number. The value of ϖ effect the warping path, to make it more biased towards the diagonal [8].

$$D(i,j) = d(i,j) + min[D\varpi(i-1,j), D(i-1,j-1), D\varpi(i,j-1)] \tag{8}$$

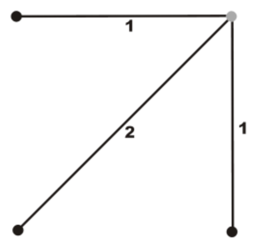
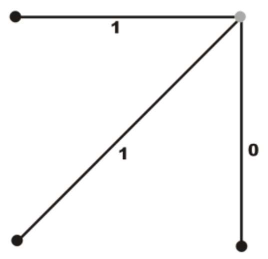
If we give more weight to cell, it means, that cell is more likely not going to be part of warping path. However selection of step pattern is not obvious, it depend on type of application where DTW will be applied. It is seen symmetric step patterns gives better results in word recognition then asymmetric, in the following section all the famous step patterns are discussed in detail.

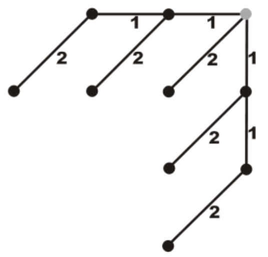
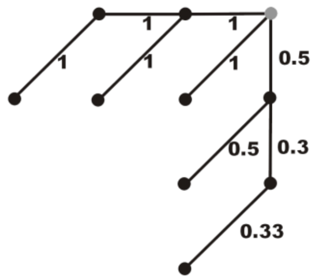
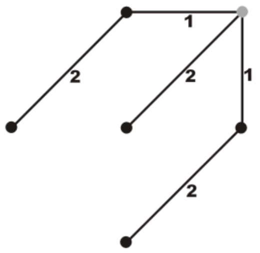
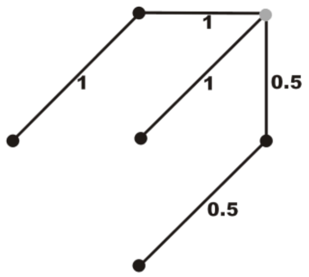
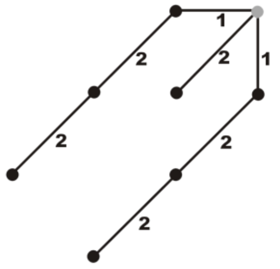
2.1.1.5 Step patterns

As mentioned earlier, first step of DTW algorithm is to fill the cost matrix with the distance according to dynamic programming formulation. In context of DTW dynamic programming formulation is referred as step pattern. In literature step patterns also referred as local continuity constraints and local constraints.

Step patterns define the value for $D(i,j)$ given the $D(i-1,j), D(i-1,j-1), D(i,j-1)$ and it is calculated progressively for all the cells in the cost matrix. In other words, it specify the allowed steps for the warping on each moment, we can take equation 7 as an example of step pattern where it determine the next point for the warping path.

In the literature[2][5][16] step patterns has been categorized into three different classifications. The most famous classification is of Sakoe and Chiba[2], classified step patterns based on their symmetry and slope constraints and proposed four step patterns, each step pattern has its symmetric and asymmetric formulation. Detailed description of Sakoe-Chiba step patterns is given in the following table 1.

Step Patterns Classified by Sakoe-Chiba			
Name	**DP-Algorithm**	**Graphical representation**	**Normalization**
symmetricP0	g[i,j] = min(g[i-1,j-1] + 2 * d[i ,j] , g[i ,j-1] + d[i ,j] , g[i-1,j] + d[i ,j])		N+M
asymmetricP0	g[i,j] = min(g[i ,j-1] + 0 * d[i ,j] , g[i-1,j-1] + d[i ,j] , g[i-1,j] + d[i ,j])		N

symmetricP05	g[i,j] = min(g[i-1,j-3] + 2 * d[i ,j-2] + d[i ,j-1] + d[i ,j], g[i-1,j-2] + 2 * d[i ,j-1] + d[i ,j] , g[i-1,j-1] + 2 * d[i ,j] , g[i-2,j-1] + 2 * d[i-1,j] + d[i ,j] , g[i-3,j-1] + 2 * d[i-2,j] + d[i-1,j] + d[i ,j])		N+M
asymmetricP05	g[i,j] = min(g[i-1,j-3] +0.33 * d[i ,j-2] +0.33 * d[i ,j-1] +0.33 * d[i ,j] , g[i-1,j-2] +0.5 * d[i ,j-1] +0.5 * d[i ,j] , g[i-1,j-1] + d[i ,j] , g[i-2,j-1] + d[i-1,j] + d[i ,j] , g[i-3,j-1] + d[i-2,j] + d[i-1,j] + d[i ,j])		N
symmetricP1	g[i,j] = min(g[i-1,j-2] + 2 * d[i ,j-1] + d[i ,j] , g[i-1,j-1] + 2 * d[i ,j] , g[i-2,j-1] + 2 * d[i-1,j] + d[i ,j])		N+M
asymmetricP1	g[i,j] = min(g[i-1,j-2] +0.5 * d[i ,j-1] +0.5 * d[i ,j] , g[i-1,j-1] + d[i ,j] , g[i-2,j-1] + d[i-1,j] + d[i ,j])		N
symmetricP2	g[i,j] = min(g[i-2,j-3] + 2*d[i-1,j-2]+2*d[i ,j-1]+d[i,j] , g[i-1,j-1] + 2*d[i ,j] , g[i-3,j-2] + 2*d[i-2,j-1]+2*d[i-1,j]+d[i,j])		N+M

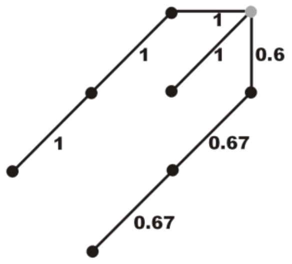
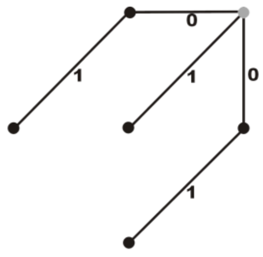
asymmetricP2	g[i,j] = min(g[i-2,j-3] +0.67 * d[i-1,j-2] +0.67 * d[i ,j-1] +0.67 * d[i ,j] , g[i-1,j-1] + d[i ,j] , g[i-3,j-2] + d[i-2,j-1] + d[i-1,j] + d[i ,j])		N

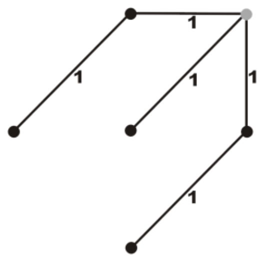
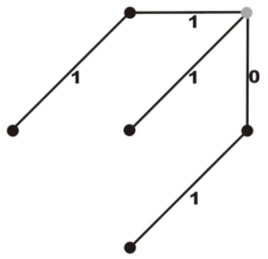
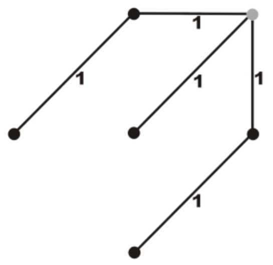
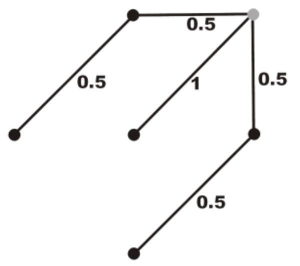
Table 1: Detail of step patterns classified by Sakoe-Shiba

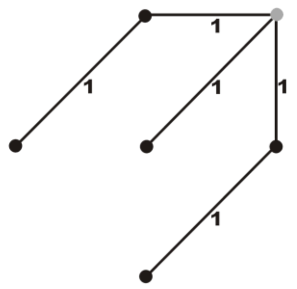
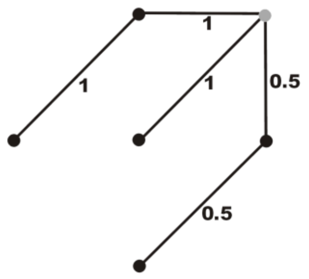
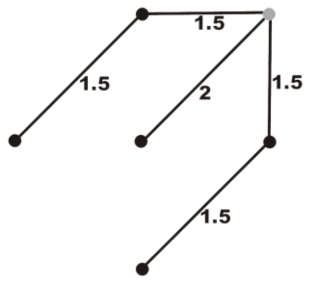
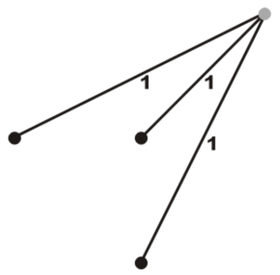
Table 1 have all the step patterns classified by Sakoe-Shiba, however it can be further categorized as symmetric-SC[2] step patterns and asymmetric-SC step patterns. It is worth noticing that all the asymmetric-SC step patterns have normalization factor *N*, where *N* is the length of query temporal sequence. It is due to the fact that in asymmetric-SC step patterns, cells on the right side (downside) of the diagonal has given less weight (shown as number on the arcs) as compared to those on the left side (upside).

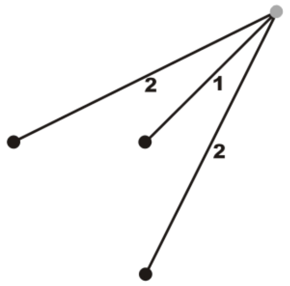
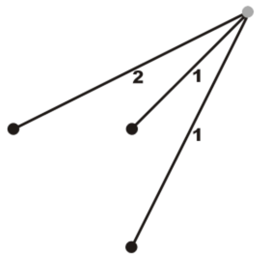
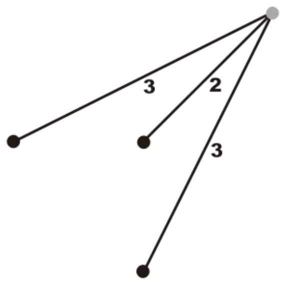
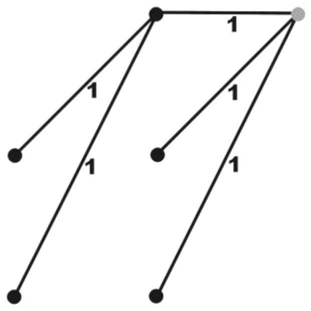
However Sakoe-Shiba classification is not the only classification of this kind. Rabiner and Juang[16] also classified different step patterns, unlike Sakoe-Chiba this classification is based on local continuity, slope weighting and smoothed. Following table 2 shows step patterns classified by Rabiner-Myers.

Step Patterns classified by Rabiner-Myers			
Name	**DP-Algorithm**	**Graphical representation**	**Normalization**
typeIa	g[i,j] = min(g[i-2,j-1] + d[i-1,j] + 0 * d[i ,j] , g[i-1,j-1] + d[i ,j] , g[i-1,j-2] + d[i ,j-1] + 0 * d[i ,j])		N/A

[2] SC stands for Sakoe-Shiba

typeIb	g[i,j] = min(g[i-2,j-1] + d[i-1,j] + d[i ,j] , g[i-1,j-1] + d[i ,j] , g[i-1,j-2] + d[i ,j-1] + d[i ,j])		N/A
typeIc	g[i,j] = min(g[i-2,j-1] + d[i-1,j] + d[i ,j] , g[i-1,j-1] + d[i ,j] , g[i-1,j-2] + d[i ,j-1] + 0 * d[i ,j] ,)		N
typeId	g[i,j] = min(g[i-2,j-1] + 2 * d[i-1,j] + d[i ,j] , g[i-1,j-1] + 2 * d[i ,j] , g[i-1,j-2] + 2 * d[i ,j-1] + d[i ,j] ,)		N+M
typeIas	g[i,j] = min(g[i-2,j-1] +0.5 * d[i-1,j] +0.5 * d[i ,j] , g[i-1,j-1] + d[i ,j] , g[i-1,j-2] +0.5 * d[i ,j-1] +0.5 * d[i ,j])		NA

typeIbs	g[i,j] = min(g[i-2,j-1] + d[i-1,j] + d[i ,j] , g[i-1,j-1] + d[i ,j] , g[i-1,j-2] + d[i ,j-1] + d[i ,j])		NA
typeIcs	g[i,j] = min(g[i-2,j-1] + d[i-1,j] + d[i ,j] , g[i-1,j-1] + d[i ,j] , g[i-1,j-2] +0.5 * d[i ,j-1] +0.5 * d[i ,j])		N
typeIds	g[i,j] = min(g[i-2,j-1] +1.5 * d[i-1,j] +1.5 * d[i ,j] , g[i-1,j-1] + 2 * d[i ,j] , g[i-1,j-2] +1.5 * d[i ,j-1] +1.5 * d[i ,j])		N+M
typeIIa	g[i,j] = min(g[i-1,j-1] + d[i ,j] , g[i-1,j-2] + d[i ,j] , g[i-2,j-1] + d[i ,j])		NA

typeIIb	g[i,j] = min(g[i-1,j-1] + d[i ,j] , g[i-1,j-2] + 2 * d[i ,j] , g[i-2,j-1] + 2 * d[i ,j])		NA
typeIIc	g[i,j] = min(g[i-1,j-1] + d[i ,j] , g[i-1,j-2] + 2 * d[i ,j] , g[i-2,j-1] + 2 * d[i ,j])		NA
typeIId	g[i,j] = min(g[i-1,j-1] + 2 * d[i ,j] , g[i-1,j-2] + 3 * d[i ,j] , g[i-2,j-1] + 3 * d[i ,j])		N+M
typeIIIc	g[i,j] = min(g[i-1,j-2] + d[i ,j] , g[i-1,j-1] + d[i ,j] , g[i-2,j-1] + d[i-1,j] + d[i ,j] , g[i-2,j-2] + d[i-1,j] + d[i ,j])		N

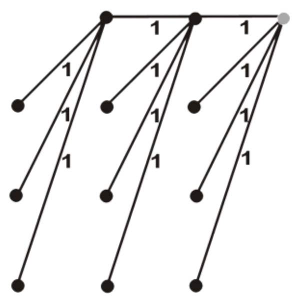
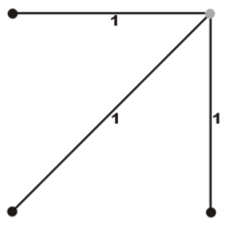
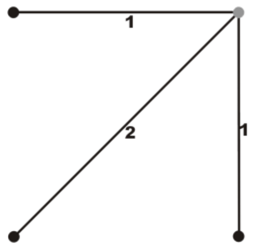
typeIVc	g[i,j] = min(g[i-1,j-1] + d[i ,j] , g[i-1,j-2] + d[i ,j] , g[i-1,j-3] + d[i ,j] , g[i-2,j-1] + d[i-1,j] + d[i ,j] , g[i-2,j-2] + d[i-1,j] + d[i ,j] , g[i-2,j-3] + d[i-1,j] + d[i ,j] , g[i-3,j-1] + d[i-2,j] + d[i-1,j] + d[i ,j] , g[i-3,j-2] + d[i-2,j] + d[i-1,j] + d[i ,j] , g[i-3,j-3] + d[i-2,j] + d[i-1,j] + d[i ,j])		N

Table 2: Detail of step patterns classified by Rabiner-Myers

It is worth noticing that in Rabiner-Myers classification they include smoothed factor in addition to Sakoe-Chiba classification.

Other (commonly used step patterns)			
Name	**DP-Algorithm**	**Graphical representation**	**Normalization**
symmetric1	g[i,j] = min(g[i-1,j-1] + d[i ,j] , g[i ,j-1] + d[i ,j] , g[i-1,j] + d[i ,j])		N/A
symmetric2	g[i,j] = min(g[i-1,j-1] + 2 * d[i ,j] , g[i ,j-1] + d[i ,j] , g[i-1,j] + d[i ,j])		N+M

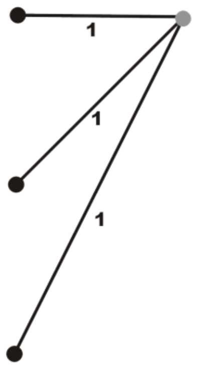
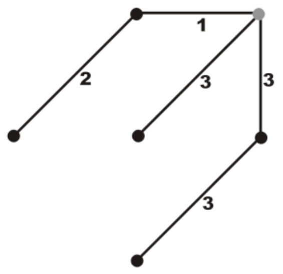
asymmetric	g[i,j] = min(g[i-1,j] + d[i ,j] , g[i-1,j-1] + d[i ,j] , g[i-1,j-2] + d[i ,j])		N
mori2006	g[i,j] = min(g[i-2,j-1] + 2 * d[i-1,j] + d[i ,j] , g[i-1,j-1] + 3 * d[i ,j] , g[i-1,j-2] + 3 * d[i ,j-1] + 3 * d[i ,j])		M

Table 3: Detail of commonly used step patterns

Table 3 gives the list of most commonly used step patterns present in the literature except mori2006[17]. As it can be seen in mori2006 step pattern, cells on right side have given weight double then diagonal and those of on the left side of diagonal.

2.1.1.6 Windowing functions

Windowing functions also referred as global constraints and bands in literature. Unlike local constraints i.e. mentioned in earlier section, global constraints define overall search space for warping path in cost matrix. Global constraints improve the running time of DTW algorithm from $O(nm)$ to $O(nk)$ where k is the window size. However it is fall in the category of quadratic time complexity.

Global constraints[14] allow only those points from the cost matrix which falls into warping window $|i_k - j_k| \leq \omega$, where ω is the positive integer represent the width of the window. In other words points which lie outside of the window are free from consideration. Local constraints contribute to find accurate warping path, whereas global constraints (windowing/band) speed up the calculation process by limiting the traversal of number of cells in the cost matrix. Among various windowing schemes Sakoe-Chiba Band[2] and Itakura Parallelogram [9] are widely used, Let's see few examples of global constraints as follow:

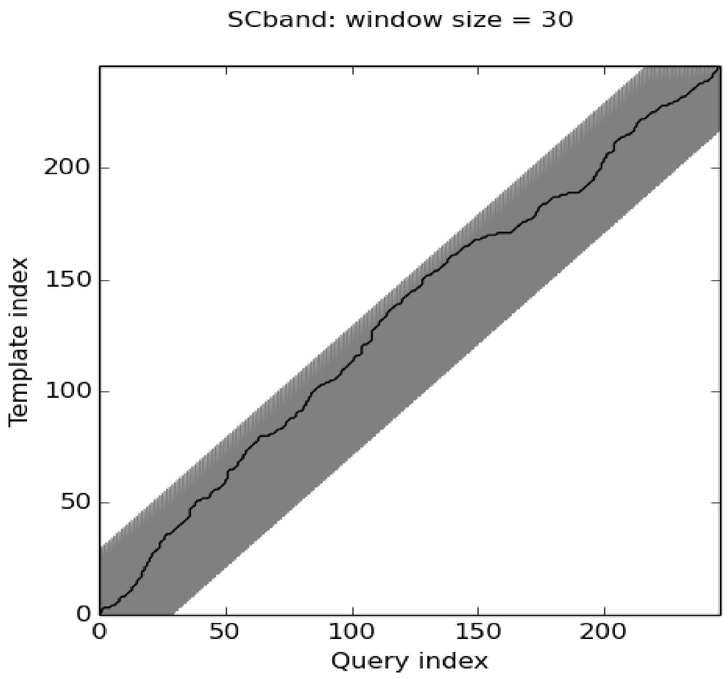


Figure 5: SCband

Sakoe-Shiba band (SCband) is one of the simplest and most commonly used bands, used to limit calculation of cells in the cost matrix. In the example figure we have used window size ω *=10*, so points/cells whose absolute difference is less then ω i.e. $|i_k - j_k| \leq 10$, will only be evaluated. However the value of size parameter can only be calculated by training method i.e. reducing the window size gradually and checking if it still gives optimal warping path.

Paliwal [10] suggested a modification over SCband, as it can be seen in figure 5, it is almost similar to SCband. Unlike SCband, Paliwal window adapt itself to length of the temporal sequence.

$$|i_k * J/I - j_k| \leq \omega \qquad \textbf{(9)}$$

In Paliwal window, cells are limited according to above mentioned equation 9, where $J \text{ and } I$ are lengths of reference and query sequences.

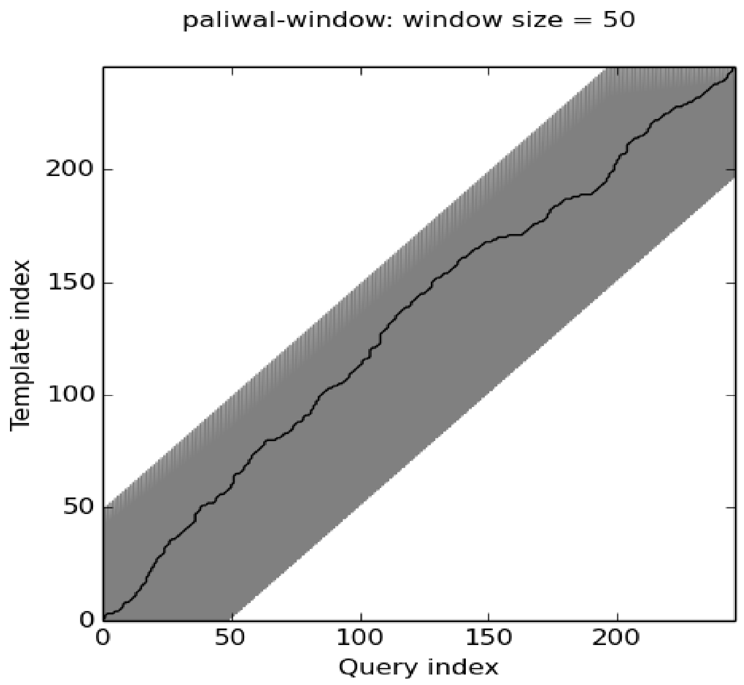


Figure 6: Paliwal window

Figure 6 illustrate a warping path calculated with in Paliwal window. However it can be seen from equation 9 that, the Paliwal window includes the boundary points in it only when the window size ω is greater than the difference[10].

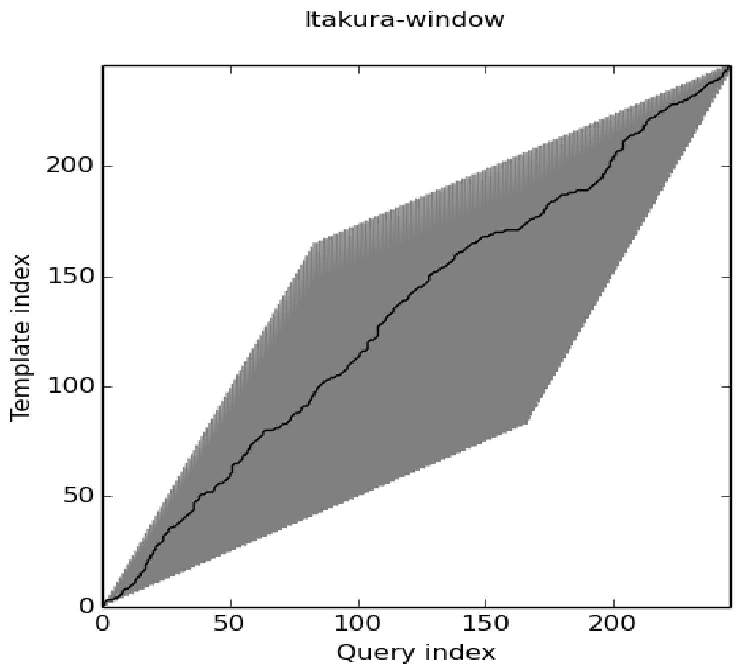


Figure 7: Itakura parallelogram

Itakura parallelogram [9] is one of the most common global constraint but it is not as simple as SCband. As it can be seen from the graphical representation of Itakura parallelogram, from the starting indices it is thin, so one can made an easy observation to set the boundary constraint for warping path very carefully.

DTW algorithm has been applied in various computing domains and has a wide range of applications. To make it more applicable many improvements has been proposed to classical DTW, which produced numerous variants of DTW algorithms. In the following section, I have discussed the classification of DTW algorithms.

2.2 Classification of DTW

As described earlier, DTW algorithms can be improved by implementing global constraints. However it is not the only way to improve the performance of DTW algorithms, to expand the application area of DTW algorithm, many attempts has been made to improve execution time of DTW. Other than global constraints, techniques like *data abstraction* and *indexing* also used to improvement successfully.

All the DTW algorithms present so far in literature classified on the basis of techniques used to improve DTW algorithms. Stan and Philip[7] described the classification of DTW algorithms, which is based on the techniques used to improve the complexity of DTW algorithms. According to classification proposed, improvements which have been made to DTW algorithms, fall into one of the following three categories.

2.2.1 Constraints based

Global constraints are used to speed up the DTW by means of restricting cells in the cost matrix as described above in constraints section. However, constraints improves the execution time by means of constant factor but overall it is still $O(N^2)$ examples includes [2][9][8] which implements windowing approach to find optimal warping path.

2.2.2 Data Abstraction based

The idea is to run DTW algorithm on low resolution time series data and scale up to full resolution cost matrix, in order to find the optimal warping path. Using this technique speed up the DTW algorithm by a constant factor but overall it is still $O(N^2)$ examples of such work includes *Iterative Deepening Dynamic Time Warping for Time Series* [18] and *Scaling up dynamic time warping for datamining applications* [19].

Data abstraction improves the DTW execution time but it do not guarantee optimal warping path due the fact that calculation of low resolution warping distance to higher resolution ignores the *local variations*.

2.2.3 Indexing based

Indexing approach to speed up DTW algorithms, applied on lower bound function (LBF) rather than applying on whole sequence. It improves the running time of DTW by limiting the number of executions but do not improve DTW algorithm directly, [20][21] describes this approach for DTW.

All three techniques mentioned above can be applied alone to improve DTW algorithm, or it can be used together. I will discuss fastDTW algorithm which returns warping path in linear time and space complexity, which uses the idea of constraints with data abstraction together.

2.3 Derivative Dynamic Time Warping

Derivative Dynamic Time Warping (DDTW)[8] is the modification of classical DTW and lies in the category of constraints based DTW algorithm. The modification is made on distance measure, as it's described in earlier section, any distance measure can be used which gives information about alignment of time series. If both time series $X \ and \ Y$ are identical to each other, the calculated distance between them should be zero. Keogh and Pazzani proposed to use square of distances of estimated derivatives that in DDTW instead of using Euclidian distance as given in equation 1 to construct n-by-m cost matrix, whereas derivative can be calculated as given in following equation 10:

$$D_x[Y] = \frac{(Y_i - Y_{i-1}) + ((Y_{i+1} - Y_{i-1})/2)}{2} \quad where \ \ 1 < i < m \tag{10}$$

Use of derivative instead of Euclidian distance is recommended to avoid problem of *singularities (single point on one time series maps onto large subsection of another time series)* to get optimal path. Problem of singularity can be produced easily by change in y-axis of one time series as depicted in the following figures:

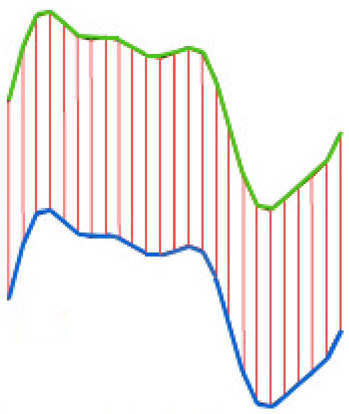


Figure 8: Alignment of two identical time series

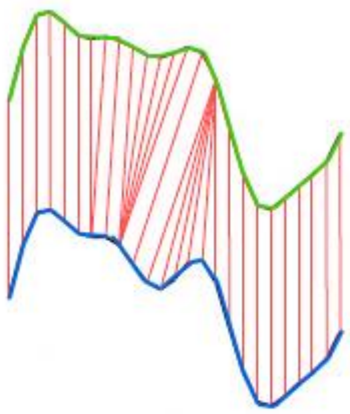


Figure 9: Alignment of two time series, with a slight change in local slop of one time series

Keogh and Pazzani further performed some experiments on different EEG, exchange rate and space shuttle datasets and shown that mean warping value calculated by DDTW is lesser then calculated by DTW. However execution time for DDTW is same as of DTW which is $O(N^2) \ where \ n = m$, with the additional overhead of constant factor due to calculation of derivatives.

In general DTW algorithm requires $O(N^2)$ for computing warping path, however few efforts has been made to improve execution time of DTW algorithm, among the improved versions of DTW, SparseDTW[22] calculates optimal warping path faster than constrained DTW algorithms[3]. fastDTW[7] (a multilevel approach) algorithm gives approximation to warping path in linear time and space complexity.

2.4 FastDTW

As discussed in earlier section, classical DTW has quadratic time and space complexity, which make it infeasible to apply classical DTW on large temporal sequence.

Therefor a lot of research work has been done to improve the complexity of DTW algorithm, which includes SparseDTW[22], Exact indexing of dynamic time warping[21], Faster Retrieval with a Two-Pass Dynamic-Time-Warping Lower Bound[23], and fastDTW[7]. Wang, Xiaoyue et al. [24] have the review about different versions of improved DTW algorithms.

FastDTW is a multilevel technique used by Stan & Philip for DTW algorithm which is influenced by graph bisection[7]. FastDTW use the concepts of constraints based DTW algorithm and data abstraction together and calculate approximate warping path distance between time series in linear time and space complexity. FastDTW algorithm solely comprise on following three main phases, *coarsening* which shrunk the time series, *projection* takes shrunken time series and calculate warping path at low resolution then project the warping path for high resolution, finally refinement of the projected path occur in *refinement* phase. Following is the brief overview of the phases.

2.4.1 Coarsening

Coarsening is the first step in multilevel approach, in coarsening time series is represented in minimal points, preserving the same curve as the original time series; this is done by taking the average of adjacent points in pairs or triplets depending upon the resolution factor[4] for the coarsening. Coarsening is called many times until the base condition is met, producing different low level resolutions for time series. Each low level resolution is then used in projection phase.

2.4.2 Projection

In projection phase, warping path is calculated for low level resolution and used to project warping path for next high level resolution. This means, each diagonal point in the low resolution path will be mapped to at least four points of high resolution path, and possibly more if it is not diagonal point provided that the resolution factor is two.

Projection phase creates a window, which includes projected path, calculated from low resolution path. The window which contain possible warping path for high resolution is then passed to refinement phase. It is worth noticing that output of the projection phase is a window, where the width of the window is determined by the radius parameter, which is then promoted to next phase for refinement.

[3] Constrained DTW algorithms are those which improve the execution time by implementing global constraints and restrict the DTW to limit the computations.
[4] Recommended resolution factor is two for coarsening.

2.4.3 Refinement

Refinement is the final phase of fastDTW algorithm; it takes a window as an input, containing possible warping path, calculated in previous phase. Refinement has nothing special, it is just like classical DTW, where we evaluate dynamic programming formulation for each cell in the window to calculate warping path. Calculated warp path, then used recursively for next resolution until path is calculated for the whole time series.

Classical DTW algorithm has quadratic time and space complexity for calculating warping path. This is because, in Classical DTW all the cells of NxM matrix is populated with the distance. However in fastDTW algorithm, number of cells is limited efficiently to N, by using data abstraction concept and with the use of global constraint.

2.4.4 Algorithm pseudocode

FastDTW algorithm uses recursive approach to determine the warping path between two time series. It takes a *radius* parameter with temporal sequences as input, which is used to apply a so called radius window. Radius window used to restrict the calculation for warping path in cost matrix, just like SCband and Itakura parallelogram.

```
# input:
# x - array - query time series
# y - array - reference time series
# radius - integer
# output:
# warping distance, warping path

def fastdtw(x, y, radius=1):
    min_time_size = radius + 2

    # Call classical dtw at lowest resolution
    if len(x)<min_time_size or len(y)< min_time_size:
        return dtw(x, y)

    # Coarsening phase
    x_shrinked = __reduce_by_half(x)
    y_shrinked = __reduce_by_half(y)

      distance, path = fastdtw(x_shrinked,y_shrinked,
      radius, dist, pattern)
    # Projection phase
    window = __expand_window(path, len(x),
             len(y),radius)

    #   Refinement phase
    return constrained_dtw(x, y,window)
```

Figure 10: The fastDTW algorithm

Pseudocode of fastDTW algorithm is shown in table 4, it can be seen that in refinement phase, variable *window* is passed as input; width of the widow depends on radius input parameter.

Chapter 3: The Python Module: DTWPy

DTW algorithms has been used in variety of applications, however DTW libraries has out dated implementation of DTW algorithms. There exists only one comprehensive library in R, which is not updated with current research made in DTW domain. DTW algorithms vary on the basis of step patterns and windowing functions.

Step patterns define the contribution of neighboring cells in path calculation, whereas windowing functions define overall search space for alignment path in DTW. Due to immense research, huge variety of step pattern, windowing functions and DTW algorithms has been proposed. This makes it arduous for time series analysts to use and test all the step patterns, windowing functions and algorithms in order to find best alignment between time series.

To overcome this limitation I have developed a comprehensive module in python DTWPy, which facilitate the use of different DTW algorithms with different step patterns and windowing functions. Abstract level view of DTWPy is shown in the following figure.

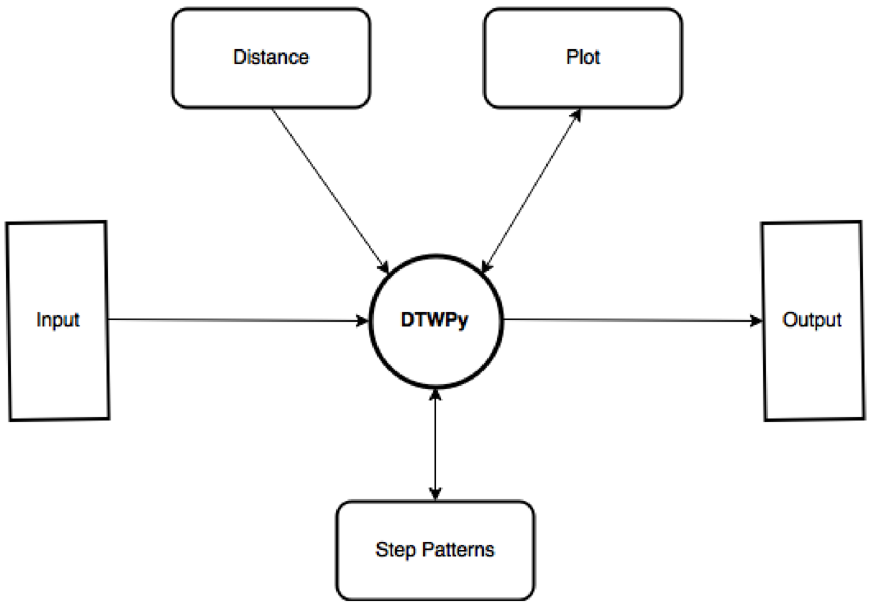


Figure 11: Architecture of DTWPy

DTWPy expose three major functions named according to the algorithm implemented i.e. DTW, fastDTW and DDTW for classical DTW, fastDTW and Derivative DTW respectively. Each function in DTWPy requires distance function as input parameter with other DTW parameters e.g. query and template time series, windowing functions.

3.1 Implementation Overview

DTWPy is implemented in Python programming language, and has dependency on numpy 1.9.2 [25] and matplotlib 1.4.3 [26]. Numpy is used to increase the performance of DTWpy, since numpy array data structure is much faster than python list. Matplotlib is used to equip DTWPy with plotting features.

DTWPy takes query and a template time series with distance function as compulsory parameters to calculate best warping path between them. However number and type of parameters depend on selected algorithm, which is described in detail in the following sections.

3.1.1 Algorithms

DTWPy has variety of algorithms, with the ability to fulfil demand of any application area. Existing implementations of DTW has only classical DTW, and lack in latest algorithms which have better performance than classical DTW. DTWPy implementation has algorithms which not only calculates warping path in linear time and space complexity, but also avoid the problem on singularity[5]. DTWPy following three different DTW algorithms which can be used with variety of different distance metrics, step patterns and global constraints algorithms.

3.1.1.7 Classical DTW

Classical DTW calculates optimal warping path and warping distance between two time series in $O(N^2)$ time and space complexity, I have explained classical DTW in earlier sections, here I will discuss how to use classical DTW in DTWpy to calculate warping path between two time series.

In DTWPy classical DTW algorithm takes three compulsory parameters as input i.e. a query time series, a template time series and distance metric, all input parameters are shown in the following:

- **x** – query time series
- **y** – template time series
- **dist** – distance function received form distancefunc module by passing the distance name
- windowtype – string indicating the window type, see window section for detail
- windowsize – integer value for size of the selected window type
- pattern – name of the step pattern, see step patterns section for possible step patterns
- normalized – set to true to get normalized warping distance
- distance_only – set to true to get only warping distance, faster

Example usage of classical DTW algorithm is shown in the following, overall code will remain same for other algorithm implementations, however function name is and parameters will vary according to algorithm.

[5] Problem of singularities is explained in section 2.3 under the heading of derivative DTW.

```
\''
   Classical DTW usage, in DTWPy
\''

from pandas.io.data import DataReader
from datetime import datetime
# import distance moduel of DTWPy module
from distance_measure import distancefunc
# import dtwpy module
import dtwpy
# import plot module of DTWPy
import plotDTWalignment

# get pandas dataframe
f = DataReader("F", "yahoo", datetime(2000, 1, 1),
    datetime(2012, 1, 1))
f_2008 = f[f.index.year == 2008]
f_2009 = f[f.index.year == 2009]
query = f_2008.Volume.values
template = f_2009.Volume.values

# get the distance function from distance measure
module
dist = distancefunc(name="manhattan")

# find warping distance and warping path between
query and template time series using classical dtw
with no window and symmetic1 step pattern

distance, path = dtwpy.dtw(query, template,
          dist=dist, windowtype=”scband”,
          windowsize=50,pattern="symmetric1",
          normalized=False,dist_only=False)
```

Figure 12: DTWPy classical DTW example

Figure 12 shows the example for using classical DTW, in DTWPy. All the algorithms implemented in DTWPy return warping path and warping distance. However path can be visualized by using the plot module i.e. sub module of DTWPy, discussed later in plotting section. Implementation code for classical DTW can be found in appendix.

3.1.1.8 Derivative DTW (DDTW)

DDTW calculates optimal warping path in $O(N^2)$ in addition to some constant factor, due the fact that it calculates the slope of the time series according to the equation 9 in this thesis. DDTW is preferred when there are variations in local slope, in one of the time series. Slope variation is referred as problem of singularities in literature, it is seen that problem of singularities is ignored by other DTW algorithms[8]. Classical DTW and DDTW share the same number and type of parameters in DTWPy as follow:

- **x** – query time series
- **y** – template time series
- **dist** – distance function received form distancefunc module by passing the distance name
- windowtype – string indicating the window type, see window section for detail
- windowsize – integer value for size of the selected window type
- pattern – name of the step pattern, see step patterns section for possible step patterns
- normalized – set to true to get normalized warping distance
- distance_only – set to true to get only warping distance, faster

```
# get the distance function from distance measure
module
dist = distancefunc(name=" euclidean")

# call to derivative DTW
distance, path = dtwpy.ddtw(query, template,
        dist=dist,windowtype=”paliwal”,
        windowsize=10,pattern="symmetricp0",
        normalized=False, dist_only=False)
```

Figure 13: Usage of DDTW

Figure 13 shows example usage for DDTW in DTWPy module, with Paliwal window type and Euclidian distance.

3.1.1.9 FastDTW

FastDTW calculates approximate warping path and warping distance in $O(N)$ time and space complexity, between two time series. It projects warping path from low resolution to high resolution and limits the calculation based on radius parameter. FastDTW implementation in DTWPy has slightly different parameter than classical DTW and DDTW. As discussed earlier, fastDTW uses so called radius window, Sakoe-Chiba window, Paliwal window[10] and Itakura parallelogram cannot be used with fastDTW, however DTWPy implementation, support usage of all step patterns with fastDTW algorithm.

FastDTW has four compulsory parameters as shown in bold font in the following:

- **x** – query time series
- **y** – template time series
- **dist** – distance function received form distancefunc module by passing the distance name
- **radius** – integer value implements so called radius window
- pattern – name of the step pattern, see step patterns section for names
- normalized – set to true to get normalized warping distance

```
# get the distance function from distance measure
module
   dist = distancefunc(name=" euclidean")

distance, path = fastdtw(x, y, radius=1,dist=dist,
           pattern="symmetricP05",
           normalized=False)
```

Figure 14: Usage of fastDTW

As shown in the snippet a special parameter *radius*, is passed to fastDTW which limit the contribution of neighboring cell, to calculate warping path and warping distance in linear time and space complexity. By default radius is set to 1, increase in radius will increase the search space for fastDTW.

3.1.2 Step Patterns in DTWPy

Step patterns are discussed in detail in section 2.1.1.5; hence in this section I will only discuss the usage of step patterns. DTWPy uses naming convention for step pattern according to classification group, as the same naming convention is used in R package[11]. DTWPy has almost all the step patterns present in literature to date, to use specific step pattern, name[6] of the step pattern have to pass in the algorithm, to get alignment accordingly. All the algorithms present in DTWPy module, support all the step patterns. However symmetric1 step pattern is implemented by default, following code snippet shows the usage of step pattern in DTWPy:

```
# All the step patterns are supported by all the
#algorithms implemented in DTWPy
   distance, path = dtwpy.ddtw(query, template,
   dist=dist,windowtype=”paliwal”,
   windowsize=10,pattern="symmetricP0",
   normalized=False, dist_only=False)
```

Figure 15: Usage of step patterns

[6] See table 1, 2, and 3 for names, DTWPy uses same names as given in tables.

3.1.3 Windowing in DTWPy

Windowing/global constraints used to minimize the number of calculations yet targeting the optimal path in DTW algorithms. Sakoe-Chiba band and Itakura parallelogram are the most common windows used to improve DTW algorithm as shown in Figure 5 and Figure 6 respectively. DTWPy gives the implementation of both windows; in addition DTWPy has implementation of Paliwal window which is a modification of Sakoe-Chiba window. All three can be used with any step pattern mentioned above, but only in Classical DTW or DDTW. This is due to the fact that, fastDTW uses its own radius window to optimize the running time of DTW algorithm, following code snippet shows the usage of all three windows in DTWPy.

```
# usage of scband

   distance, path = dtwpy.ddtw(query, template,
   dist=dist,windowtype="scband",
   windowsize=10,pattern="symmetricP0",
   normalized=False, dist_only=False)

# usage of Paliwal window

   distance, path = dtwpy.ddtw(query, template,
   dist=dist,windowtype="paliwal",
   windowsize=10,pattern="symmetricP0",
   normalized=False, dist_only=False)

# usage of Itakura parallelogram

   cost_matrix, distance, path = dtwpy.ddtw(query,
   template,
   dist=dist,windowtype="itakura",
   windowsize=10,pattern="symmetricP0",
   normalized=False, dist_only=False,cost_matrix=True)
```

Figure 16: Usage of windowing functions

3.1.4 Plotting in DTWPy

Matplotlib library is used in DTWPy, to facilitate DTW alignment visualization, between two time series. Plotting module in DTWPy uses warping path variable returned from DTWPy as it can be depicted in architecture figure 11. There are three types of plotting available in DTWPy. Visualization of warping path with query and template sequences as shown in figure 17 in the following.

Plotting module in DTWPy is kept separate, so new visualization features can be added in future.

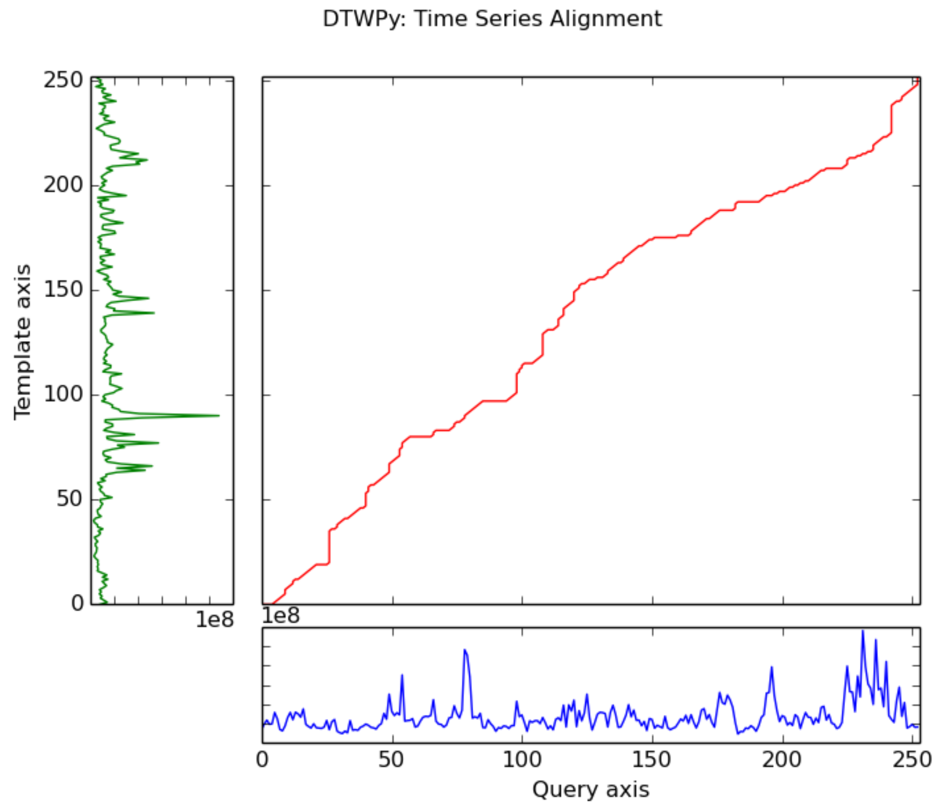


Figure 17: Alignment plot with query and template time series

It is very easy to plot alignment using DTWPy's plotting module, it require warping path with both time series, in order to visualize with query and template sequences.

```
# import plotDTWalignment
# to plot with query and template sequence
plotDTWalignment.plotwithQT(x, y, path,title)

# to plot with alignment with window
plotDTWalignment.plotalignment_with_window(window,path,title)

# to plot with simple alignment
plotDTWalignment.plotalignment(path, title="")
```

Figure 18: Usage of windowing functions

In the following chapter 4, I will describe the correctness and comparison of DTWPy module with other existing implementations.

Chapter 4: Results

As discussed in earlier sections, DTW has been used in wide range of applications and there is not any comprehensive implementation which covers all the DTW variants. However there exists a very few basic implementations of classical DTW e.g. mlpy[12].

4.1 DTW Package in R

Toni Giorgino developed a package of DTW algorithms in R [11]. The R package gives option to use different local constraints, and global constraints to run with classical DTW algorithm. The R package implemented well known symmetric1, symmetric2 and asymmetric with all types of step patterns (referred as slop constraints, local constraints and DP-recursion rules in literature) classified by Rabinar-Juang [16], Sakoe-Chiba [2], and Rabiner-Myers[5]. Moreover in global constraints R package includes Sakoe-Chiba band, and Itakura parallelogram [9]. In addition to most common distance functions the R package also has implementations to Levenshtein's distance and Needleman-Wunsch algorithm[27], which is used to calculate alignments of strings.

In R package the kernel of DTW function is written in C programming language to speed up the execution however, as it still has $O(N^2)$ time and space complexity, and not updated with recent research, which makes it limited to larger problems.

4.2 Mlpy

Mlpy[12] is a Python module for machine learning, have very basic and outdated implementation of DTW. It has implemented classical DTW with SCband and Itakura parallelogram windowing functions.

In the following section, table 4, 5 and 6 compare functionality provided by DTWPy, R and mlpy for DTW algorithms.

4.3 Correctness of DTWPy

Every warping path must satisfy continuity, monotonicity and boundary conditions, as described in chapter 2. Every step pattern, windowing function and algorithm in DTWPy is tested thoroughly. I have compared the results with R implementation, which is the only comprehensive implementation, exist after DTWPy. Figure 19, 20 and 21 shows warping path obtained by using step patterns classified by Sakoe-Shiba with the classical DTW, DDTW and fastDTW respectively. Following results verify the correctness of the implemented algorithms.

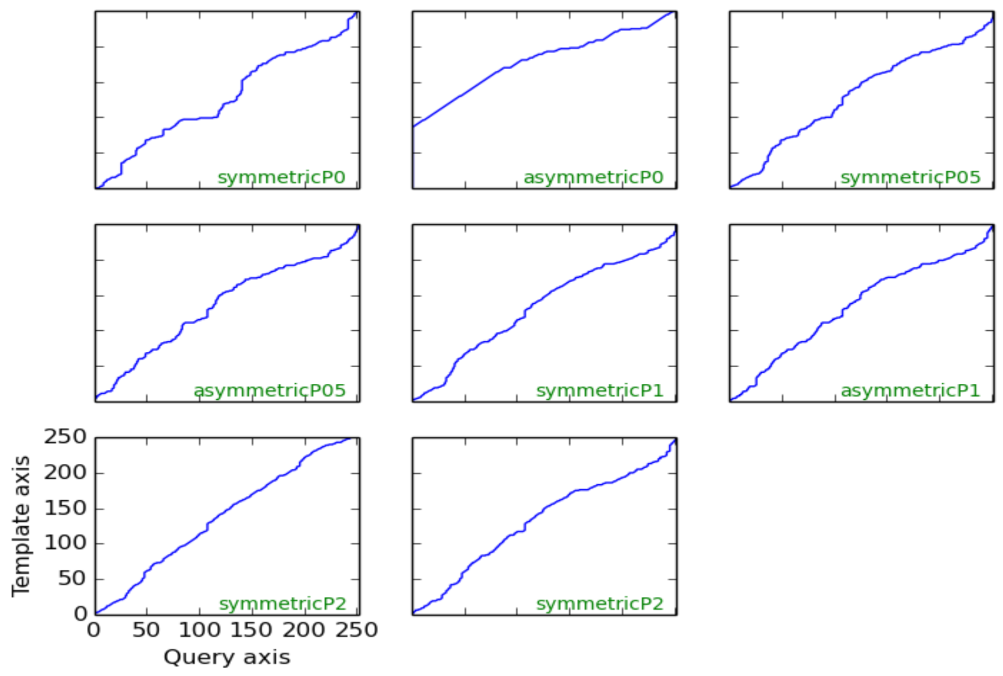


Figure 19: Classical DTW with Sakoe-Shiba step pattern

It can be seen easily that, warping paths in figure 19 and 20 are identical i.e. calculated by using classical DTW and fastDTW algorithm respectively. I used radius = 10 for fastDTW algorithm and it can be concluded that on radius = 10 fastDTW algorithm give optimal results, in linear time and space complexity.

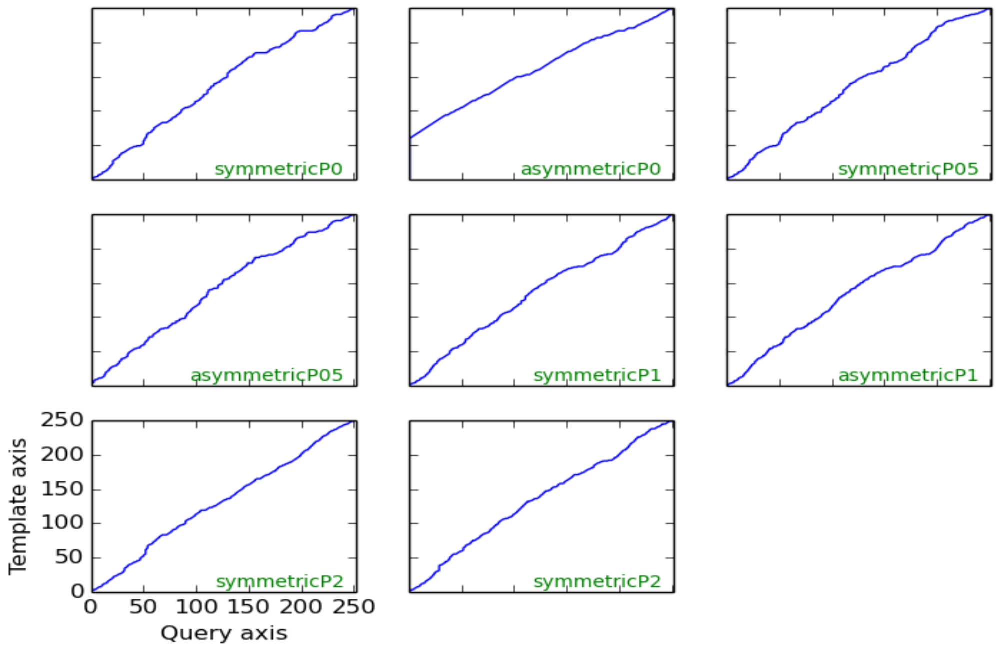


Figure 20: DDTW with Sakoe-Shiba step pattern

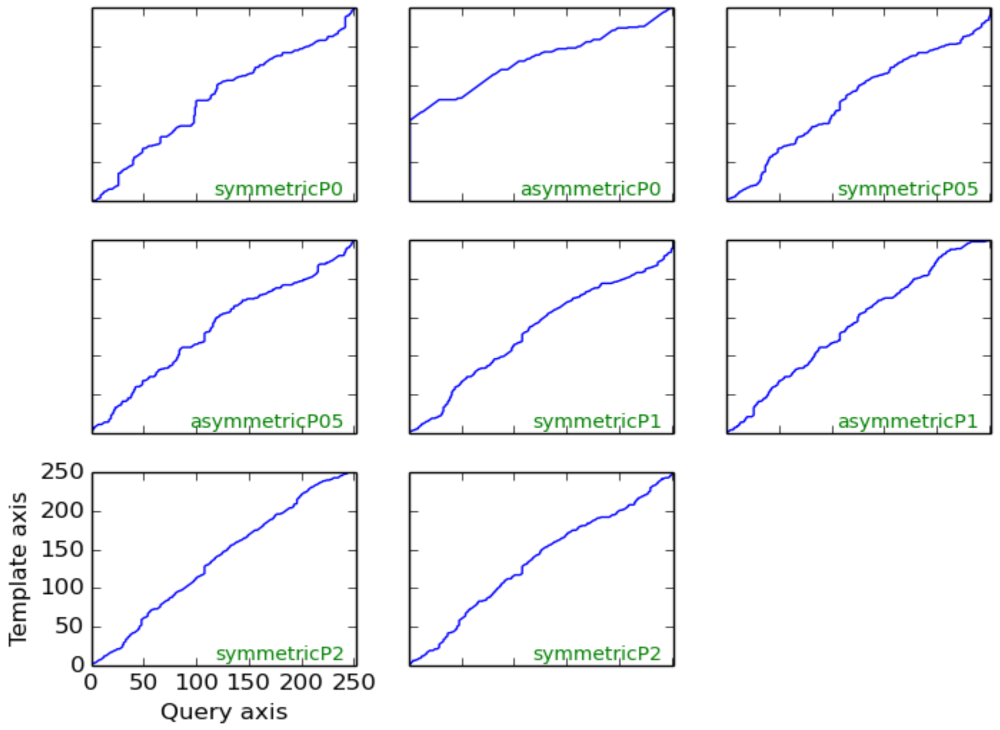


Figure 21: FastDTW with Sakoe-Shiba step pattern

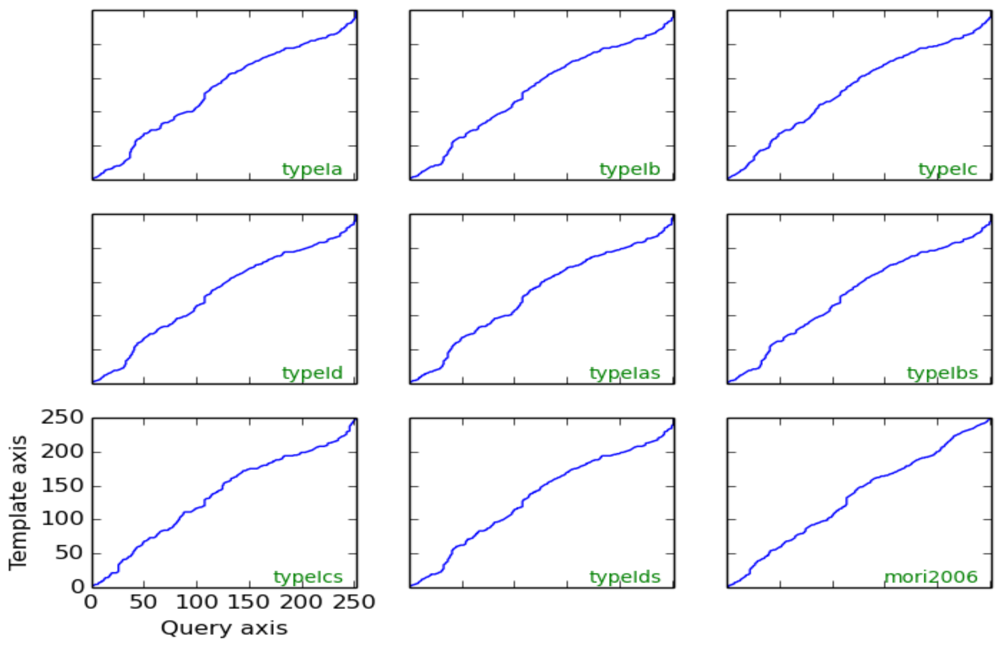


Figure 22: Classical DTW with Rabiner-Myers step patterns

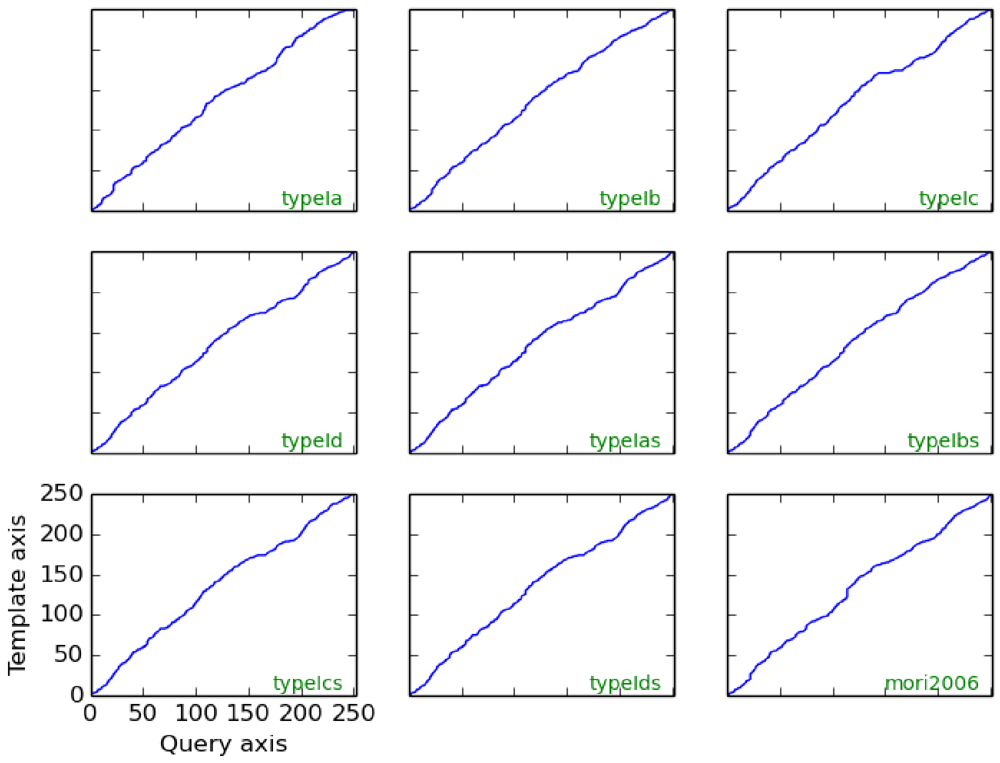


Figure 23: DDTW with Rabiner-Myers step patterns

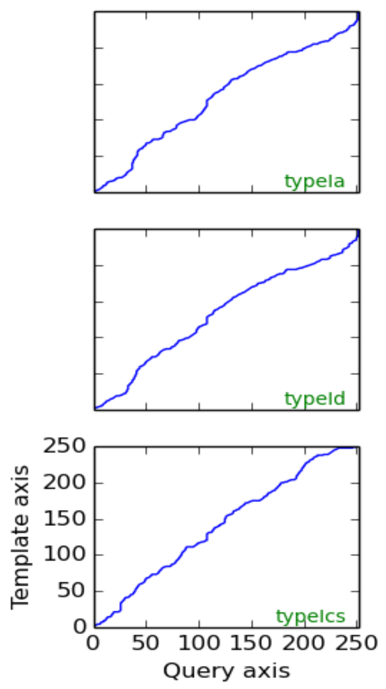


Figure 24: FastDTW with Rabiner-Myers step patterns

Figure 22, 23 and 24 shows warping paths obtained by using typeI step patterns classified by Rabiner-Myers with the classical DTW, DDTW and fastDTW respectively. All the algorithms implemented in DTWPy give correct results, if we compare with R implementation.

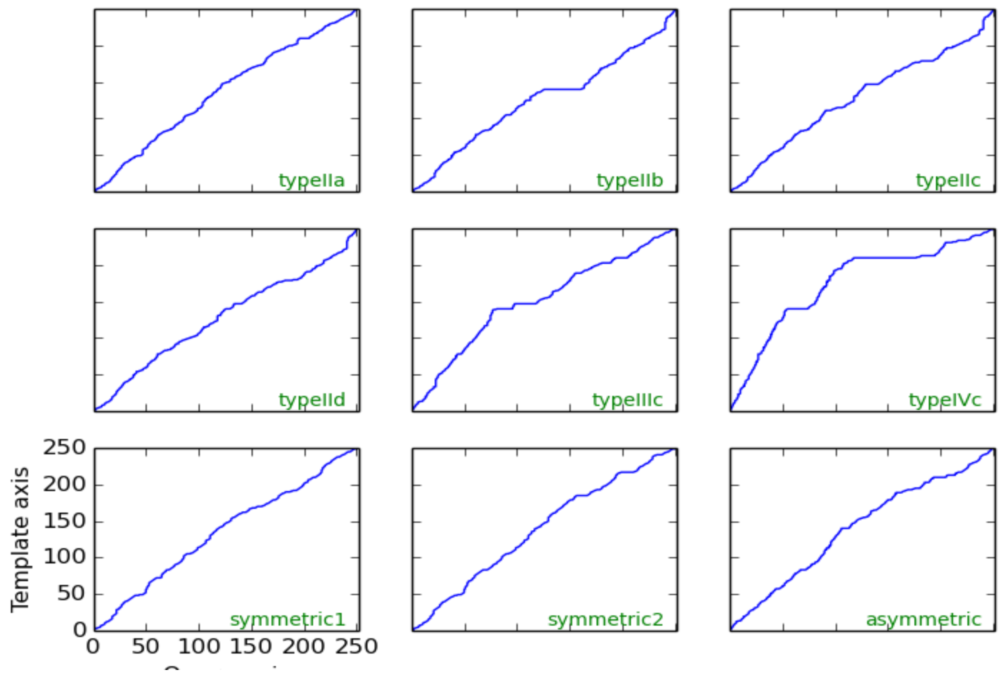


Figure 25: Classical DTW with Rabiner-Myers step patterns and other

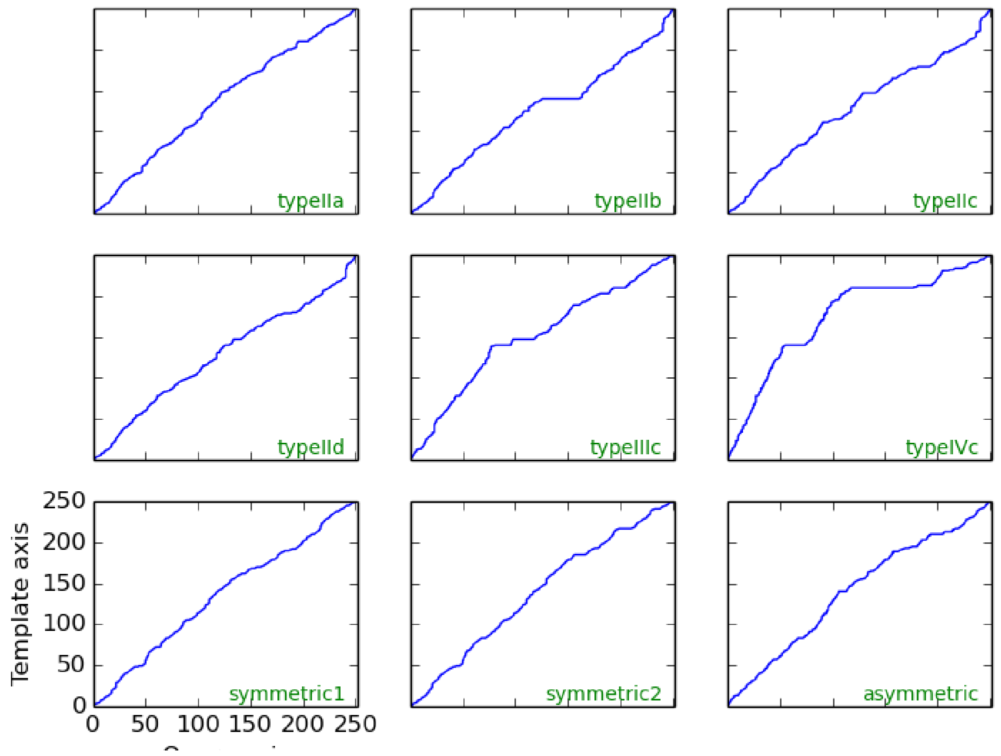


Figure 26: DDTW with Rabiner-Myers step patterns and other

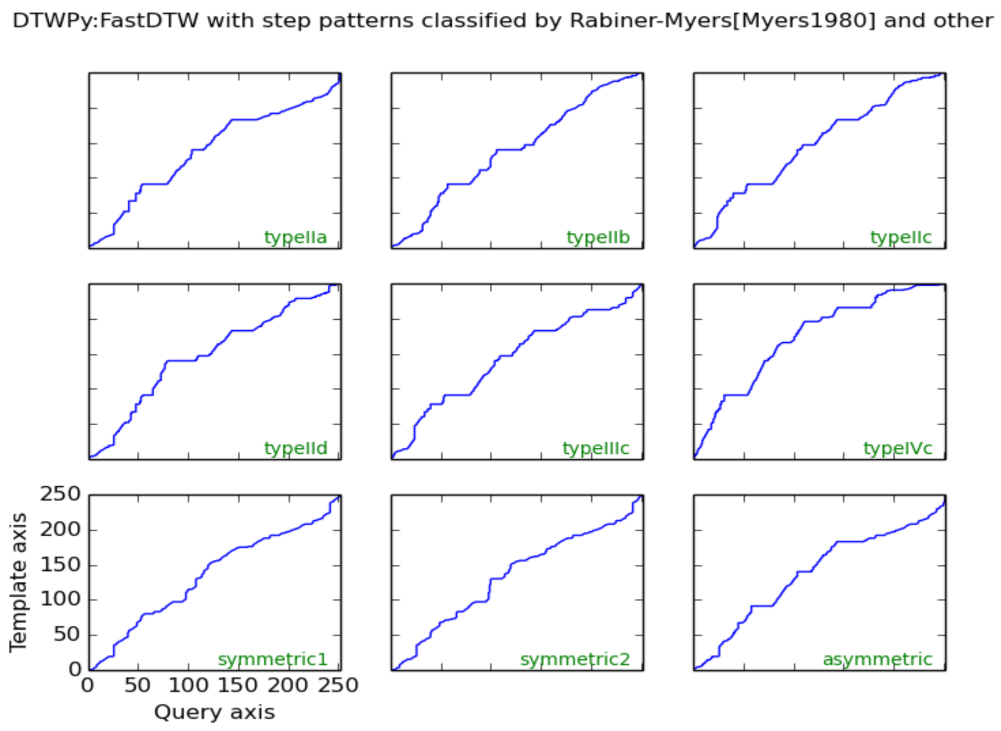


Figure 27: FastDTW with Rabiner-Myers step patterns and other

Figure 25, 26 and 27 shows the warping path obtained by using step patterns typeII, typeIII and typeIV classified by Rabiner-Myers whereas and most commonly used step patterns in the literature.

It is worth noticing that the alignments calculated by classical DTW and fastDTW are identical to each other. Whereas alignments calculated by using DDTW are different and paths are tend to be toward diagonal , it is due to the fact that DDTW uses derivatives instead of original values and determine the warping path accordingly.

4.4 Comparison of DTW implementations

Classical DTW algorithm calculates alignment in quadratic time and space complexity Mlpy and R package only provide implementation of classical DTW, which do not scale well for large time series, following table 4, 5 and 6 gives an overview of different implementations which I found the most comprehensive.

Algorithms	**DTWPy**	**R Package**	**Mlpy**
Classical DTW	Yes	Yes	Yes
FastDTW	Yes	No	No
Derivative DTW	Yes	No	No

Table 4: Implementation of DTW algorithms

As it can be seen from table 4, R package and Mlpy module for DTW gives implementation only for classical DTW. On the other hand DTWPy has the most comprehensive implementation of DTW algorithms. DTWPy leverages fastDTW and DDTW, which determines the alignment in linear time and space complexity, and gives better results when time series has local shifts in the slope[8] respectively.

Global Constraints	**DTWPy**	**R Package**	**Mlpy**
SCband	Yes	Yes	Yes
Itakura parallelogram	Yes	Yes	Yes
Slantedband[7]	No	Yes	Yes
Paliwal window	Yes	No	No

Table 5: Implementation of global constraints

To improve the performance of classical DTW Sakeo-Shiba, Iakura and Paliwal proposed SCband, Itakura parallelogram and Paliwal window[10] respectively. Global constraints reduce the search space of cost matrix, which result in slight improvement in performance. Slantedband (R package specific) is a modification over SCband, which is not much different from Paliwal window, and is not commonly used in literature. Table 5 shows the implementation of global constraints which can be used to improve performance of classical DTW. It clearly shows DTWPy has upper hand to all other DTW libraries, in terms of windowing functions.

[7] Slanted window is R package specific, have no confirmation about author.

Step Patterns	DTWPy	R Package	Mlpy
SC step patterns	Yes	Yes	No
Rabiner-Myers	Yes	Yes	No
Rabiner-Juang	No	Yes	No
symmetric1	Yes	Yes	No
symmetric2	Yes	Yes	Yes
asymmetric	Yes	Yes	No
mori2006	Yes	Yes	No

Table 6: Implementation of local constraints

Apart from global constraints, local constraints (step patterns) also play vital roles to find the optimal path in DTW[8], table 6 present overview of step pattern implemented by DTWpy, R package and Mlpy. In the coming section I will discuss about performance of DTWPy.

4.5 Performance Analysis

In section 4.3, I have discussed correctness i.e. DTWPy gives correct results on any combination of DTW algorithms. End results i.e. warping distance and warping path, of each step pattern with windowing functions have been verified with mlpy and R implementation of DTW. DTWPy is thoroughly tested. Moreover graphs in section 3.2 also conveys that warping path obtained by DTWPy fulfill all the compulsory constraints (described in section 2.2.1) which need to be met to get optimal warping path.

In addition to correctness I analyzed the execution time of algorithms DTWPy, in contrast to R package implementation, system specifications are listed in table 7 as follow:

System Specifications	
Operating System	Ubuntu 14.04 LTS
Operating System Type	64bit
Processor	Intel® Core™ i7-3612QM CPU @ 2.10GHz × 8
Memory	7.7 GiB
Swap	14.9 GiB
Model	Lenovo-G500s

Table 7: System specifications

[8] Described in section 3.4 in detail

To analyze the execution time on both environments R and Python, I have used Rprof and Cprofile respectively on the same dataset. At first I performed a test with classical DTW of R package and DTWPy module, the result is shown in figure 21, on the y axis time taken in seconds and on the x-axis I placed length of time series.

Several observations were made and it shows DTWPy has visible high execution time as compared to R, the main reason of this difference is that, kernel of R DTW package's is written in C programming language, and use matrix implementation under the hood, which make it faster than DTWpy. On the other hand DTWPy is implemented in python programming language. Python is referred as a scripting language and dynamically typed, which implies that python's code is interpreted rather than compiled. Therefore python's code is translated to machine language at run time, thus python has to pay overhead of translation with execution at run time.

However, dynamic typing in python makes development easier and flexible, which allows scientists to use development time more efficiently. This is because python has been used intensively for scientific purposes, and communities have been continuously growing. Summing-up, python is an efficient language for doing scientific research with code.

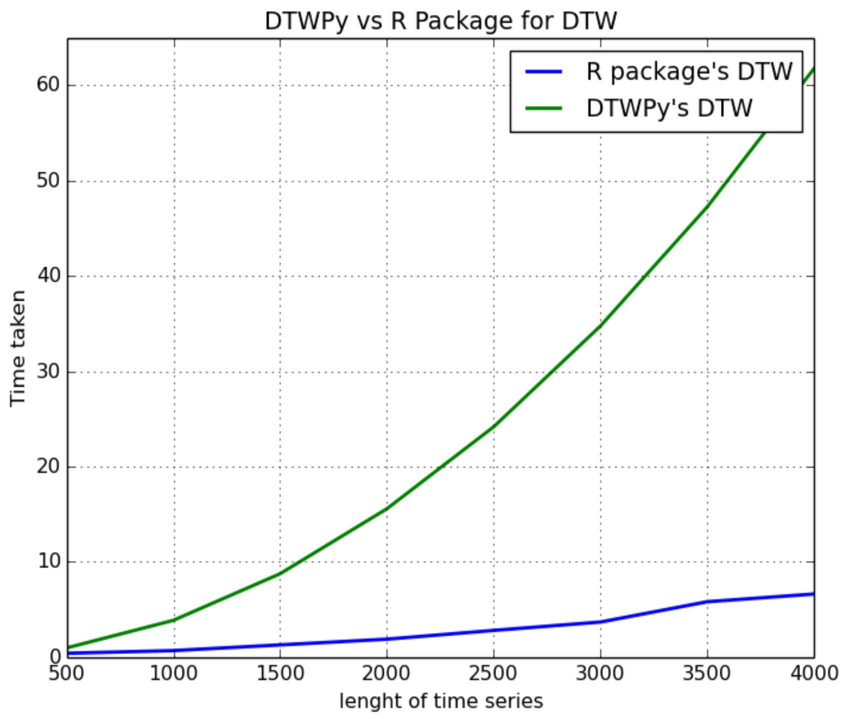


Figure 28: Execution time w.r.t lenght of time series

As in above comparison of classical DTW between DTWPy and R package has measureable difference besides the fact both algorithms share the same time and space complexity. It was quite compelling to do performance comparison of R implementation against fastDTW.

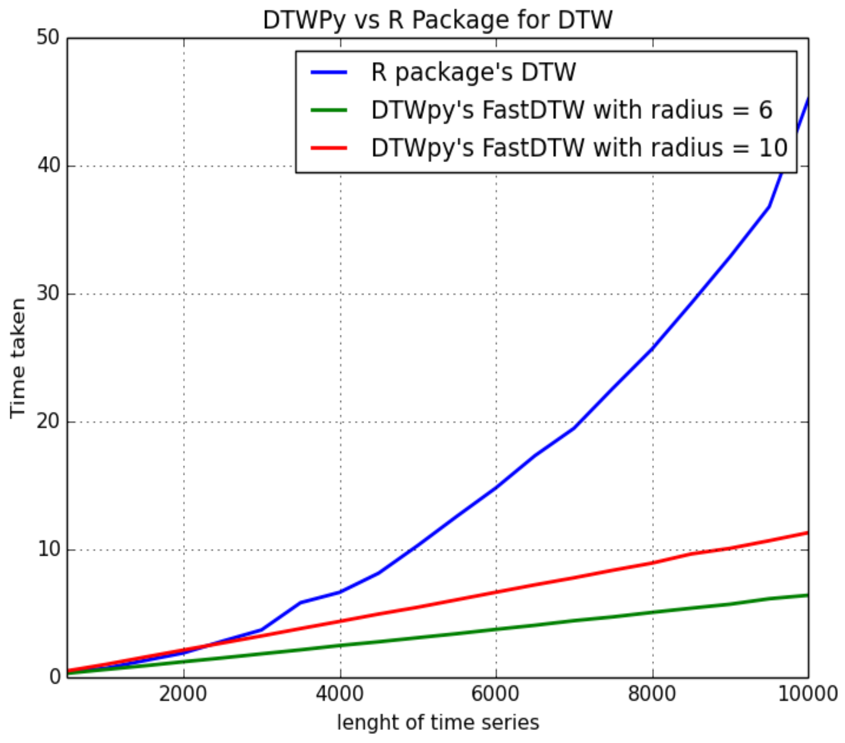


Figure 29: Execution time w.r.t length of time series

To compare the execution time of fastDTW implementation with R, I have taken time in seconds on y-axis and length of time series on x-axis. Same environment and dataset has been provided to both implementations. The result was quite convincing. As it can be seen from figure 22 with shorter length of time series R implementation of DTW has almost same execution time. But with the increase in length of time series, R implementation shows notable increase in execution time. It is due to the fact that fastDTW has linear time and space complexity whereas Classical DTW bears the quadratic time complexity to calculate warping path between two time series. It is worth noticing that even on radius= 10 fastDTW has clear difference in time, and on radius=10.

In short DTWPy covers all the DTW variants present in literature so far, and equipped with latest performance improvements made on DTW algorithms. In general DTWPy has advantage on all other implementations of DTW present so far.

Chapter 5: Conclusion

Dynamic Time warping is a famous technique to find optimal warping path between time series, and has significance use in different research disciplines. However, there does not exist any of such comprehensive library which covers maximum DTW variants. R DTW package facilitate the data analyst up to some extent, but it do not scale well for time series of big length. This is due to the fact that R only provides implementation of classical DTW which has (N^2) time and space complexity.

In this thesis, I presented DTWPy; a python module for computing and visualizing Dynamic Time Warping alignments, which have implementations of almost all local constraints and global constraints present in the literature to date. DTWPy have clear advantage over all existing DTW Implementations due to the fact that, unlike other implementations it is updated with recent possible improvements made on DTW algorithm in the form of fastDTW and DDTW. DTWPy architecture is designed in a way so that I can easily incorporate possible future improvements on DTW.

At the present time, most of the data analysts are using Pandas[6] data set ,and python for time series analysis. DTWPy is implemented in core python, which encourages python communities, working on time series analysis to use DTWPy without any development cost. DTWPy module can be used in any kind of application, keeping in mind the limitations of python and DTW algorithm.

.

Appendix

5.1 Distance measures

```
#distance_measure.py
def distancefunc(name="manhattan"):
    if name=="manhattan":
        return lambda x, y:abs(x - y)
    elif name == "euclidean":
        return lambda x,y:pow(x-y, 2)
    elif name == "canberra":
        return lambda x,y:(abs(x - y) / (x + y))
    return lambda x, y:abs(x - y)
```

5.2 Step patterns

```
# patterns.py
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import numpy as np

def symmetric1(ts_x, ts_y, window, dist, pattern,
    normalized):
    cost_matrix = np.zeros([len(ts_x), len(ts_y)])
    cost_matrix[:] = float('inf')
    for i, j in window:
        dt = dist(ts_x[i], ts_y[j])
        if i == j == 0:
            cost_matrix[i][j] = dt
        else:
            cost_matrix[i][ j] = min(cost_matrix[i][ j
           - 1] + dt, cost_matrix[i - 1][ j] + dt, cost_matrix[i -
           1][ j - 1] + dt)
    if normalized:return cost_matrix
    return cost_matrix


def symmetric2(ts_x, ts_y, window, dist, pattern, normalized):
    cost_matrix = np.empty([ts_x.size, ts_y.size])
    cost_matrix[:] = float('inf')
    for i, j in np.nditer(window):
        dt = dist(ts_x[i], ts_y[j])
        if i == j == 0:
            cost_matrix[i][j] = dt
        else:
            cost_matrix[i][ j] = np.minimum(cost_matrix[i][ j - 1] +
            dt, cost_matrix[i - 1][ j] + dt, cost_matrix[i - 1][ j -
            1] + 2 * dt)

    if normalized:cost_matrix[i][ j] = (cost_matrix[i][ j] /
        (len(ts_x) + len(ts_y)))
    return cost_matrix

def asymmetric(ts_x, ts_y, window, dist, pattern, normalized):
    cost_matrix = np.zeros([len(ts_x), len(ts_y)])
    cost_matrix[:] = float('inf')
    for i, j in window:
        dt = dist(ts_x[i], ts_y[j])
        if i == j == 0:
            cost_matrix[i][ j] = dt
        else:
            cost_matrix[i][ j] = min(cost_matrix[i - 1][ j - 2] + dt,
            cost_matrix[i-1][j] + dt, cost_matrix[i - 1][j - 1] + dt)

    if normalized:cost_matrix[i][j] = (cost_matrix[i][j] /
(len(ts_x)))
    return cost_matrix

def asymmetricP0(ts_x, ts_y, window, dist, pattern, normalized):
    cost_matrix = np.zeros([len(ts_x), len(ts_y)])
    cost_matrix[:] = float('inf')
    for i, j in window:
        dt = dist(ts_x[i], ts_y[j])
        if i == j == 0:
            cost_matrix[i][j] = dt
        elif i == 0:
            cost_matrix[i][j] = cost_matrix[i][j - 1] + dt
        elif j == 0:
            cost_matrix[i][j] = cost_matrix[i - 1][ j] + dt
        else:
            cost_matrix[i][j] = min(cost_matrix[i][j - 1],
            cost_matrix[i-1][j]+ dt, cost_matrix[i - 1][ j - 1] + dt)
    if normalized:cost_matrix[i][j]=(cost_matrix[i][j] / (len(ts_x)))
    return cost_matrix

def symmetricVelichkoZagoruyko(ts_x, ts_y, window, dist, pattern,
normalized):
    cost_matrix = np.zeros([len(ts_x), len(ts_y)])
    cost_matrix[:] = float('inf')
    for i, j in window:
        dt = dist(ts_x[i], ts_y[j])
        if i == j == 0:
            cost_matrix[i][ j] = dt  # 0#2 * dt #float('inf')  #
        else:
            cost_matrix[i][ j] = min(cost_matrix[i ][ j - 1] + dt,
            cost_matrix[i - 1][ j - 1] + dt * .001, cost_matrix[i –
            1][ j])
    return cost_matrix

def symmetricP1(ts_x, ts_y, window, dist, pattern, normalized):
    cost_matrix = np.zeros([len(ts_x), len(ts_y)])
    cost_matrix[:] = float('inf')
    for i, j in window:
        dt = dist(ts_x[i], ts_y[j])
        dt1 = dist(ts_x[i], ts_y[j - 1])
        dt2 = dist(ts_x[i - 1], ts_y[j])
        if i - 1 < 0:
            dt2 = float('inf')
        if j - 1 < 0:
            dt1 = float('inf')
        if i == j == 0:
            cost_matrix[i][ j] = dt  # 0#2 * dt #float('inf')  #
        else:
            cost_matrix[i][ j] = min(cost_matrix[i - 1][ j - 2] + 2 * 
dt1 + dt, cost_matrix[i - 1][ j - 1] + 2 * dt, cost_matrix[i - 2][ j 
- 1] + 2 * dt2 + dt)
    if normalized:cost_matrix[i][ j] = (cost_matrix[i][ j] / 
(len(ts_x) + len(ts_y)))
    return cost_matrix

def asymmetricP1(ts_x, ts_y, window, dist, pattern, normalized):
    cost_matrix = np.zeros([len(ts_x), len(ts_y)])
    cost_matrix[:] = float('inf')
    for i, j in window:
        dt = dist(ts_x[i], ts_y[j])
        dt1 = dist(ts_x[i], ts_y[j - 1])
        dt2 = dist(ts_x[i - 1], ts_y[j])
        if i - 1 < 0:
            dt2 = float('inf')
        if j - 1 < 0:
            dt1 = dist(ts_x[i], ts_y[j - 1])
        if i == j == 0:
            cost_matrix[i][j] = dt
        else:
            cost_matrix[i][j] = min(cost_matrix[i - 1][ j - 2] + dt1 
/ 2 + dt / 2, cost_matrix[i - 1][ j - 1] + dt, cost_matrix[i - 2][ j 
- 1] + dt2 + dt)
    if normalized:cost_matrix[i][ j] = cost_matrix[i][ j] / 
(len(ts_x))
    return cost_matrix

def symmetricP2(ts_x, ts_y, window, dist, pattern, normalized):
    cost_matrix = np.zeros([len(ts_x), len(ts_y)])
    cost_matrix[:] = float('inf')
    for i, j in window:
        dt = dist(ts_x[i], ts_y[j])
        dt1 = dist(ts_x[i], ts_y[j - 1])
        dt2 = dist(ts_x[i - 1], ts_y[j])
        dt3 = dist(ts_x[i - 1], ts_y[j - 2])
        dt4 = dist(ts_x[i - 2], ts_y[j - 1])
        if i - 1 < 0 :
            dt2 = float('inf')
        if j - 1 < 0 :
            dt2 = float('inf')
        if  i - 2 < 0:
            dt4 = float('inf')
        if j - 2 < 0:
            dt3 = float('inf')

        if i == j == 0:
            cost_matrix[i][j] = dt
        else:
            cost_matrix[i][j] = min(cost_matrix[i - 2][ j - 3] + dt1
* 2 + dt + dt3 * 2, cost_matrix[i - 1][ j - 1] + 2 * dt,
cost_matrix[i - 3][j - 2] + 2 * dt4 + 2 * dt2 + dt)
    if normalized:cost_matrix[i][j] = cost_matrix[i, j] / (len(ts_x)
+ len(ts_y))
    return cost_matrix

def asymmetricP2(ts_x, ts_y, window, dist, pattern, normalized):
    cost_matrix = np.zeros([len(ts_x), len(ts_y)])
    cost_matrix[:] = float('inf')
    for i, j in window:
        dt = dist(ts_x[i], ts_y[j])
        dt1 = dist(ts_x[i], ts_y[j - 1])
        dt2 = dist(ts_x[i - 1], ts_y[j])
        dt3 = dist(ts_x[i - 1], ts_y[j - 2])
        dt4 = dist(ts_x[i - 2], ts_y[j - 1])
        if i - 1 < 0 :
            dt2 = float('inf')
        if j - 1 < 0 :
            dt2 = float('inf')
        if  i - 2 < 0:
            dt4 = float('inf')
        if j - 2 < 0:
            dt3 = float('inf')

        if i == j == 0:
            cost_matrix[i][ j] = dt
        else:
            cost_matrix[i][ j] = min(cost_matrix[i - 2][ j - 3] + dt1
* (2 / 3) + dt * (2 / 3) + dt3 * (2 / 3), cost_matrix[i - 1][ j - 1]
+ dt, cost_matrix[i - 3][ j - 2] + dt4 + dt2 + dt)
    if normalized:cost_matrix[i][j] = cost_matrix[i][j] / (len(ts_x))
    return cost_matrix

def symmetricP05(ts_x, ts_y, window, dist, pattern, normalized):
    cost_matrix = np.zeros([len(ts_x), len(ts_y)])
    cost_matrix[:] = float('inf')
    for i, j in window:
        dt = dist(ts_x[i], ts_y[j])
        dt1 = dist(ts_x[i], ts_y[j - 1])
        dt2 = dist(ts_x[i - 1], ts_y[j])
        dt3 = dist(ts_x[i], ts_y[j - 2])
        dt4 = dist(ts_x[i - 2], ts_y[j])

        if i - 1 < 0 :
            dt2 = float('inf')
        if j - 1 < 0 :
            dt2 = float('inf')
        if  i - 2 < 0:
            dt4 = float('inf')
        if j - 2 < 0:
            dt3 = float('inf')

        if i == j == 0:
            cost_matrix[i][ j] = dt
        else:
            cost_matrix[i][j]=min(cost_matrix[i-1][j-3]
                 +dt3*2+dt1+dt,
                  cost_matrix[i - 1][ j - 2] + dt1 * 2 + dt,
                  cost_matrix[i - 1][ j - 1] + 2 * dt,
                  cost_matrix[i - 2][ j - 1] + 2 * dt2 + dt,
                  cost_matrix[i - 3][ j - 1] + 2 * dt4 + dt2 + dt)
    if normalized:cost_matrix[i][ j] = cost_matrix[i, j] / (len(ts_x)
          + len(ts_y))
    return cost_matrix

def asymmetricP05(ts_x, ts_y, window, dist, pattern, normalized):
    cost_matrix = np.zeros([len(ts_x), len(ts_y)])
    cost_matrix[:] = float('inf')
    for i, j in window:
        dt = dist(ts_x[i], ts_y[j])
        dt1 = dist(ts_x[i], ts_y[j - 1])
        dt2 = dist(ts_x[i - 1], ts_y[j])
        dt3 = dist(ts_x[i], ts_y[j - 2])
        dt4 = dist(ts_x[i - 2], ts_y[j])

        if i - 1 < 0 :
            dt2 = float('inf')
        if j - 1 < 0 :
            dt2 = float('inf')
        if  i - 2 < 0:
            dt4 = float('inf')
        if j - 2 < 0:
            dt3 = float('inf')
        if i == j == 0:
            cost_matrix[i][j] = dt
        else:
            cost_matrix[i][j] = min(cost_matrix[i - 1][ j - 3] + dt3
                   * (1 / 3) + dt1 * (1 / 3) + dt * (1 / 3) ,
                   cost_matrix[i-1][j-2]+dt1*(1 / 2) + dt * (1 / 2),
                   cost_matrix[i - 1][ j - 1] + dt,
                   cost_matrix[i - 2][ j - 1] + dt2 + dt,
                   cost_matrix[i - 3][ j - 1] + dt4 + dt2 + dt)
    if normalized:cost_matrix[i, j]=cost_matrix[i][j][0]/ (len(ts_x))
    return cost_matrix

def asymmetricItakura(ts_x, ts_y, window, dist, pattern, normalized):
    cost_matrix = np.zeros([len(ts_x), len(ts_y)])
    cost_matrix[:] = float('inf')
    for i, j in window:
        dt = dist(ts_x[i], ts_y[j])
        dt2 = dist(ts_x[i - 1], ts_y[j])
        if j - 1 < 0 :
            dt2 = float('inf')
        if i == j == 0:
            cost_matrix[i][ j] = dt
        else:
            cost_matrix[i][ j] = min(cost_matrix[i -1][ j - 2] + dt ,
                    cost_matrix[i - 1][ j - 1] + dt,
                    cost_matrix[i - 2][ j - 1] + dt2 + dt,
                    cost_matrix[i - 2][ j - 2] + dt2 + dt)
    return cost_matrix

def typeIa(ts_x, ts_y, window, dist, pattern, normalized):
    cost_matrix = np.zeros([len(ts_x), len(ts_y)])
    cost_matrix[:] = float('inf')
    for i, j in window:
        dt = dist(ts_x[i], ts_y[j])
        dt1 = dist(ts_x[i], ts_y[j - 1])
        dt2 = dist(ts_x[i - 1], ts_y[j])

        if i - 1 < 0 :
            dt2 = float('inf')
        if j - 1 < 0 :
            dt2 = float('inf')

        if i == j == 0:
            cost_matrix[i][ j] = dt
        else:
            cost_matrix[i][j] = min(cost_matrix[i - 2][ j - 1] + dt2,
                      cost_matrix[i - 1][j - 1] + dt,
                      cost_matrix[i - 1][ j - 2] + dt1)
    return cost_matrix

def typeIb(ts_x, ts_y, window, dist, pattern, normalized):
    cost_matrix = np.zeros([len(ts_x), len(ts_y)])
    cost_matrix[:] = float('inf')
    for i, j in window:
        dt = dist(ts_x[i], ts_y[j])
        dt1 = dist(ts_x[i], ts_y[j - 1])
        dt2 = dist(ts_x[i - 1], ts_y[j])

        if i - 1 < 0 :
            dt2 = float('inf')
        if j - 1 < 0 :
            dt2 = float('inf')

        if i == j == 0:
            cost_matrix[i][j] = dt
        else:
            cost_matrix[i][j] = min(cost_matrix[i-2][j-1]+ dt2 + dt ,
                     cost_matrix[i - 1][j - 1] + dt,
                     cost_matrix[i - 1][ j - 2] + dt1 + dt)
    return cost_matrix

def typeIc(ts_x, ts_y, window, dist, pattern, normalized):
    cost_matrix = np.zeros([len(ts_x), len(ts_y)])
    cost_matrix[:] = float('inf')
    for i, j in window:
        dt = dist(ts_x[i], ts_y[j])
        dt1 = dist(ts_x[i], ts_y[j - 1])
        dt2 = dist(ts_x[i - 1], ts_y[j])

        if i - 1 < 0 :
            dt2 = float('inf')
        if j - 1 < 0 :
            dt2 = float('inf')

        if i == j == 0:
            cost_matrix[i][j] = dt
        else:
            cost_matrix[i][j]=min(cost_matrix[i-2][j- 1] + dt2 + dt ,
                   cost_matrix[i - 1][ j - 1] + dt,
                   cost_matrix[i - 1][j - 2] + dt1)
    if normalized:cost_matrix[i][j] = cost_matrix[i, j] / (len(ts_x))
    return cost_matrix

def typeId(ts_x, ts_y, window, dist, pattern, normalized):
    cost_matrix = np.zeros([len(ts_x), len(ts_y)])
    cost_matrix[:] = float('inf')
    for i, j in window:
        dt = dist(ts_x[i], ts_y[j])
        dt1 = dist(ts_x[i], ts_y[j - 1])
        dt2 = dist(ts_x[i - 1], ts_y[j])

        if i - 1 < 0 :
            dt2 = float('inf')
        if j - 1 < 0 :
            dt2 = float('inf')
        if i == j == 0:
            cost_matrix[i][ j] = dt
        else:
            cost_matrix[i][j] = min(cost_matrix[i - 2][ j - 1] + dt2
            * 2 + dt  ,
                     cost_matrix[i - 1][j - 1] + dt * 2,
                     cost_matrix[i - 1][ j - 2] + dt1 * 2 + dt)
    if normalized:cost_matrix[i][j] = cost_matrix[i][ j] / (len(ts_x)
        + len(ts_y))
    return cost_matrix

def typeIas(ts_x, ts_y, window, dist, pattern, normalized):
    cost_matrix = np.zeros([len(ts_x), len(ts_y)])
    cost_matrix[:] = float('inf')
    for i, j in window:
        dt = dist(ts_x[i], ts_y[j])
        dt1 = dist(ts_x[i], ts_y[j - 1])
        dt2 = dist(ts_x[i - 1], ts_y[j])

        if i - 1 < 0 :
            dt2 = float('inf')
        if j - 1 < 0 :
            dt2 = float('inf')

        if i == j == 0:
            cost_matrix[i][ j] = dt
        else:
            cost_matrix[i][j]=min(cost_matrix[i-2][j-1]+dt2/2+dt/ 2 ,
                    cost_matrix[i - 1][j - 1] + dt,
                    cost_matrix[i - 1][ j - 2] + dt1 / 2 + dt / 2)
    if normalized:cost_matrix[i][j]= cost_matrix[i][ j] / (len(ts_x))
    return cost_matrix

def typeIbs(ts_x, ts_y, window, dist, pattern, normalized):
    cost_matrix = np.zeros([len(ts_x), len(ts_y)])
    cost_matrix[:] = float('inf')
    for i, j in window:
        dt = dist(ts_x[i], ts_y[j])
        dt1 = dist(ts_x[i], ts_y[j - 1])
        dt2 = dist(ts_x[i - 1], ts_y[j])

        if i - 1 < 0 :
            dt2 = float('inf')
        if j - 1 < 0 :
            dt2 = float('inf')

        if i == j == 0:
            cost_matrix[i][ j] = dt
        else:
            cost_matrix[i][ j]=min(cost_matrix[i-2][j-1] + dt2 + dt ,
                       cost_matrix[i - 1][ j - 1] + dt,
                       cost_matrix[i - 1][ j - 2] + dt1 + dt)
    if normalized:cost_matrix[i][j]=cost_matrix[i][ j] / (len(ts_x))
    return cost_matrix


def typeIcs(ts_x, ts_y, window, dist, pattern, normalized):
    cost_matrix = np.zeros([len(ts_x), len(ts_y)])
    cost_matrix[:] = float('inf')
    for i, j in window:
        dt = dist(ts_x[i], ts_y[j])
        dt1 = dist(ts_x[i], ts_y[j - 1])
        dt2 = dist(ts_x[i - 1], ts_y[j])

        if i - 1 < 0 :
            dt2 = float('inf')
        if j - 1 < 0 :
            dt2 = float('inf')

        if i == j == 0:
            cost_matrix[i][j] = dt
        else:
            cost_matrix[i][j] = min(cost_matrix[i - 2][j-1]+dt2 + dt,
                  cost_matrix[i - 1][j - 1] + dt,
                  cost_matrix[i - 1][j - 2] + dt1 / 2 + dt / 2)
    if normalized:cost_matrix[i][j] = cost_matrix[i][j] / (len(ts_x))
    return cost_matrix

def typeIds(ts_x, ts_y, window, dist, pattern, normalized):
    cost_matrix = np.zeros([len(ts_x), len(ts_y)])
    cost_matrix[:] = float('inf')
    for i, j in window:
        dt = dist(ts_x[i], ts_y[j])
        dt1 = dist(ts_x[i], ts_y[j - 1])
        dt2 = dist(ts_x[i - 1], ts_y[j])

        if i - 1 < 0 :
            dt2 = float('inf')
        if j - 1 < 0 :
            dt2 = float('inf')

        if i == j == 0:
            cost_matrix[i][ j] = dt
        else:
            cost_matrix[i][j] = min(cost_matrix[i - 2][ j - 1] + dt2
                 * 1.5 + dt * 1.5 ,
                 cost_matrix[i - 1][ j - 1] + dt * 2,
                 cost_matrix[i - 1][j - 2] + dt1 * 1.5 + dt * 1.5)
    if normalized:cost_matrix[i][j] = cost_matrix[i][j] / (len(ts_x)
            + len(ts_y))
    return cost_matrix
```

```
def typeIIa(ts_x, ts_y, window, dist, pattern, normalized):
    cost_matrix = np.zeros([len(ts_x), len(ts_y)])
    cost_matrix[:] = float('inf')
    for i, j in window:
        dt = dist(ts_x[i], ts_y[j])
        if i == j == 0:
            cost_matrix[i][ j] = dt
        else:
            cost_matrix[i][ j] = min(cost_matrix[i - 1][ j - 1] + dt,
                 cost_matrix[i - 2][j - 1] + dt,
                 cost_matrix[i - 1][ j - 2] + dt)
    if normalized:cost_matrix[i][j] =cost_matrix[i][ j] / (len(ts_x))
    return cost_matrix

def typeIIb(ts_x, ts_y, window, dist, pattern, normalized):
    cost_matrix = np.zeros([len(ts_x), len(ts_y)])
    cost_matrix[:] = float('inf')
    for i, j in window:
        dt = dist(ts_x[i], ts_y[j])
        if i == j == 0:
            cost_matrix[i][j] = dt
        else:
            cost_matrix[i][j] = min(cost_matrix[i-1][j-1]+dt,
                 cost_matrix[i - 2][ j - 1] + dt * 2 ,
                 cost_matrix[i - 1][ j - 2] + dt * 2)
    if normalized:cost_matrix[i][j] = cost_matrix[i][j] / (len(ts_x))
    return cost_matrix

def typeIIc(ts_x, ts_y, window, dist, pattern, normalized):
    cost_matrix = np.zeros([len(ts_x), len(ts_y)])
    cost_matrix[:] = float('inf')
    for i, j in window:
        dt = dist(ts_x[i], ts_y[j])
        if i == j == 0:
            cost_matrix[i][ j] = dt
        else:
            cost_matrix[i][j] = min(cost_matrix[i - 1][ j - 1] + dt,
                 cost_matrix[i - 1][ j - 2] + dt,
                 cost_matrix[i - 2][ j - 1] + dt * 2)
    if normalized:cost_matrix[i][j] = cost_matrix[i][j] / (len(ts_x))
    return cost_matrix

def typeIId(ts_x, ts_y, window, dist, pattern, normalized):
    cost_matrix = np.zeros([len(ts_x), len(ts_y)])
    cost_matrix[:] = float('inf')
    for i, j in window:
        dt = dist(ts_x[i], ts_y[j])
        if i == j == 0:
            cost_matrix[i][j] = dt
        else:
            cost_matrix[i][j] = min(cost_matrix[i-1][j - 1] + dt * 2,
                 cost_matrix[i - 1][ j - 2] + dt * 3 ,
                 cost_matrix[i - 2][ j - 1] + dt * 3)
    if normalized:cost_matrix[i][ j] = cost_matrix[i][ j] /
            (len(ts_x) + len(ts_y))
    return cost_matrix
def typeIIIc(ts_x, ts_y, window, dist, pattern, normalized):
    cost_matrix = np.zeros([len(ts_x), len(ts_y)])
    cost_matrix[:] = float('inf')
    for i, j in window:
        dt = dist(ts_x[i], ts_y[j])
        dt1 = dist(ts_x[i - 1], ts_y[j])

        if i - 1 < 0 :
            dt1 = float('inf')
        if i == j == 0:
            cost_matrix[i][ j] = dt  # 0#2 * dt #float('inf')  #
        else:
            cost_matrix[i][j] = min(cost_matrix[i - 1][ j - 2] + dt,
                 cost_matrix[i - 1][ j - 1] + dt,
                 cost_matrix[i - 2][j - 1] + dt1 + dt,
                 cost_matrix[i - 2][j - 2] + dt1 + dt)
    if normalized:cost_matrix[i][j] = cost_matrix[i][j] / (len(ts_x))
    return cost_matrix

def typeIVc(ts_x, ts_y, window, dist, pattern, normalized):
    cost_matrix = np.zeros([len(ts_x), len(ts_y)])
    cost_matrix[:] = float('inf')
    for i, j in window:
        dt = dist(ts_x[i], ts_y[j])
        dt1 = dist(ts_x[i - 1], ts_y[j])
        dt2 = dist(ts_x[i - 2], ts_y[j])

        if i - 1 < 0 :
            dt1 = float('inf')
        if i - 2 < 0 :
            dt2 = float('inf')
        if i == j == 0:
            cost_matrix[i][ j] = dt
        else:
            cost_matrix[i][ j] = min(cost_matrix[i - 1][ j - 1] + dt,
                      cost_matrix[i - 1][j - 2] + dt,
                      cost_matrix[i - 1][ j - 3] + dt ,
                      cost_matrix[i - 2][ j - 1] + dt1 + dt,
                      cost_matrix[i - 2][ j - 2] + dt1 + dt,
                      cost_matrix[i - 2][j - 3] + dt1 + dt,
                      cost_matrix[i - 3][ j - 1] + dt2 + dt1 + dt,
                      cost_matrix[i - 3][j - 2]+ dt2 + dt1 + dt,
                      cost_matrix[i - 3][ j - 3] + dt2 + dt1 + dt )
    if normalized:cost_matrix[i][j] = cost_matrix[i][j] / (len(ts_x))
    return cost_matrix
```

```
def mori2006(ts_x, ts_y, window, dist, pattern, normalized):
    cost_matrix = np.zeros([len(ts_x), len(ts_y)])
    cost_matrix[:] = float('inf')
    for i, j in window:
        dt = dist(ts_x[i], ts_y[j])
        dt1 = dist(ts_x[i - 1], ts_y[j])
        dt2 = dist(ts_x[i], ts_y[j - 1])
        if i - 1 < 0 :
            dt1 = float('inf')
        if j - 1 < 0 :
            dt2 = float('inf')
        if i == j == 0:
            cost_matrix[i][j] = dt
        else:
            cost_matrix[i][j] = min(cost_matrix[i-2][j-1]+dt1*2 + dt,
                 cost_matrix[i - 1][ j - 1]+dt* 3,
                 cost_matrix[i - 1][ j - 2] + dt2 * 3 + dt * 3 )
    if normalized:cost_matrix[i][j] = cost_matrix[i][ j]/ (len(ts_y))
    return cost_matrix
```

5.3 Plotting module

```
#plotDTWalignment.py
import matplotlib.pyplot as plt
import numpy as np

def plotwithQT(x, y, path,title):
    xaxis = [x1[0] for x1 in path]
    yaxis = [y1[1] for y1 in path]

    plt.figure(0)
    # plt.axes([0, x.size, 0, y.size])
    ax1 = plt.subplot2grid((5, 5), (0, 0), rowspan=4)
    x1 = np.arange(0, y.size)
    ax1.set_ylabel('Template axis')
    plt.setp(ax1.get_xticklabels(), visible=False)
    ax1.plot(y, x1, color='green')

    ax1.set_ylim([0, y.size])

    ax3 = plt.subplot2grid((5, 5), (4, 1), colspan=4)
    ax3.plot(x)
    plt.setp(ax3.get_yticklabels(), visible=False)
    ax3.set_xlabel('Query axis')
    ax3.set_xlim([0, x.size])
    ax2 = plt.subplot2grid((5, 5), (0, 1), colspan=4, rowspan=4)
    ax2.plot(xaxis,yaxis, label=r"symmetric1",color='red')
    # setp( ax2.get_xticklabels(), visible=False)


    plt.legend(loc="upper left", bbox_to_anchor=[0, 1], ncol=1,
       shadow=True, title="Step Pattern", fancybox=True)
    ax2.get_legend().get_title().set_color("purple")
    ax2.set_xlim([0, x.size])
    ax2.set_ylim([0, y.size])
    plt.setp(ax2.get_xticklabels(), visible=False)
    plt.setp(ax2.get_yticklabels(), visible=False)

    plt.suptitle("DTWPy: Time Series Alignment"+title)
    plt.show()

def plotalignment_with_window(window,path,title):
    plt.figure(num=None, figsize=(5, 5), dpi=80, facecolor='w',
         edgecolor='k')
    xlim,ylim = max(path,key=lambda item:item[1])
    wx = [x[0] for x in window]
    wy = [y[1] for y in window]
    x = [x[0] for x in path]
    y = [y[1] for y in path]
    plt.plot(wx,wy,color="gray")
    plt.plot(x,y,color="black")
    plt.ylabel('Template index')
    plt.xlabel('Query index')
    plt.axis([0, xlim, 0, ylim])
    plt.suptitle("Itakura-window"+title)
    plt.show()

def plotalignment(path,title):
    plt.figure(num=None, figsize=(5, 5), dpi=80, facecolor='w',
        edgecolor='k')
    xlim,ylim = max(path,key=lambda item:item[1])
    x = [x[0] for x in path]
    y = [y[1] for y in path]
    plt.plot(x,y,color="black")
    plt.ylabel('Template index')
    plt.xlabel('Query index')
    plt.axis([0, xlim, 0, ylim])
    plt.suptitle("Itakura-window"+title)
    plt.show()

def plotdetailalignment(query,template,path, title=''):
    coef=5
    xaxis = [x1[0] for x1 in path]
    yaxis = [y1[1] for y1 in path]
    for i in range(len(template)):
        template[i] += template[i]+coef

    plt.figure(1)
    plt.plot(template, lw=3)
    plt.plot(query, lw=3)
    for i in range(0,max(len(xaxis),len(yaxis))):
        plt.plot()
    #for val in path:
        #plt.plot([val[0], val[1]],[template[val[0]],
query[val[1]]], 'k', lw=0.5)

    plt.axis('off')
    plt.show()
```

5.4 DTWPy

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# dtwpy.py

import patterns
import numpy as np
from memprof import *
def __derivation(ts):
        drts = []

        for i in range(1, len(ts) - 1):
            derived = (ts[i]-ts[i-1)+((ts[i+1]-ts[i-1])/2.0)/2.0
            drts.append(derived)

        return drts

def ddtw(x, y, dist=lambda a, b: abs(a - b), windowtype=None,
windowsize=None, pattern="symmetric1", normalized=False,
dist_only=False):
    x = __derivation(x)
    y = __derivation(y)
    len_x, len_y = len(x), len(y)
    window = ''
    if windowtype == "scband":
        window = __getSCwindow(windowsize, len_x, len_y)
    elif windowtype == "itakura":
        window = __getItakurawindow(len_x, len_y)
    elif windowtype == "paliwal":
        window = __getpaliwalwindow(windowsize, len_x, len_y)
    else:
        window = [(i, j) for i in range(len_x) for j in
            range(len_y)]
    D = __cost_matrix(x, y, window, dist, pattern, normalized)
    if dist_only:
        return D[len_x - 1][ len_y - 1]
    else:
        path = __backtrack(D, len_x - 1, len_y - 1)

    return (window,D[len_x - 1][ len_y - 1], path)

def dtw(x, y, dist=lambda a, b: abs(a - b), windowtype=None,
windowsize=None, pattern="symmetric1",
normalized=False,dist_only=False,cost=False):
    len_x, len_y = x.size, y.size
    np.empty([len_x, len_y], dtype=float)
    if windowtype == "scband":
        window = __getSCwindow(windowsize, len_x, len_y)
    elif windowtype == "itakura":
        window = __getItakurawindow(len_x, len_y)
    elif windowtype == "paliwal":
        window = __getpaliwalwindow(windowsize, len_x, len_y)
    else:
        window = [(i, j) for i in np.arange(len_x) for j in
np.arange(len_y)]
    D = __cost_matrix(x, y, window, dist, pattern, normalized)
    if dist_only:
        return D[len_x - 1][ len_y - 1]
    else:
        path = __backtrack(D, len_x - 1, len_y - 1)
    if cost == False:
        return (D[len_x - 1][ len_y - 1], path)
    return (window,D[len_x - 1][ len_y - 1], path)



def fastdtw(x, y, radius=1, dist=lambda x, y:abs(x - y),
pattern="symmetric1", normalized=False):
    min_time_size = radius + 2

    if len(x) < min_time_size or len(y) < min_time_size:
        return constrained_dtw(x, y, dist, pattern, normalized)

    x_shrinked = __reduce_by_half(x)
    y_shrinked = __reduce_by_half(y)
    _, path = fastdtw(x_shrinked, y_shrinked, radius, dist,
        pattern, normalized)
    window = __expand_window(path, len(x), len(y), radius)
    return constrained_dtw(x, y, dist, pattern, normalized,
window)

def constrained_dtw(x, y, dist, pattern, normalized,
window=None):
    len_x, len_y = len(x), len(y)
    if window is None:
        window = [(i, j) for i in range(len_x) for j in
range(len_y)]

    D = __cost_matrix(x, y, window, dist, pattern, normalized)
    path = __backtrack(D, len_x - 1, len_y - 1)

    return (D[len_x - 1][ len_y - 1], path)

def __cost_matrix(ts_x, ts_y, window, dist, pattern,
normalized):
    if pattern == "symmetric1":
        return patterns.symmetric1(ts_x, ts_y, window, dist,
pattern, normalized)
    elif pattern == "symmetric2" or pattern == "symmetricP0":
        return patterns.symmetric2(ts_x, ts_y, window, dist,
pattern, normalized)
    elif pattern == "asymmetric":
        return patterns.asymmetric(ts_x, ts_y, window, dist,
pattern, normalized)
    elif pattern == "symmetricP1":
        return patterns.symmetricP1(ts_x, ts_y, window, dist,
pattern, normalized)
    elif pattern == "asymmetricP0":
        return patterns.asymmetricP0(ts_x, ts_y, window, dist,
pattern, normalized)
    elif pattern == "asymmetricP1":
        return patterns.asymmetricP1(ts_x, ts_y, window, dist,
pattern, normalized)
    elif pattern == "symmetricP2":
        return patterns.symmetricP2(ts_x, ts_y, window, dist,
pattern, normalized)
    elif pattern == "asymmetricP2":
        return patterns.asymmetricP2(ts_x, ts_y, window, dist,
pattern, normalized)
    elif pattern == "symmetricP05":
        return patterns.symmetricP05(ts_x, ts_y, window, dist,
pattern, normalized)
    elif pattern == "asymmetricP05":
        return patterns.asymmetricP05(ts_x, ts_y, window, dist,
pattern, normalized)
    elif pattern == "asymmetricItakura":
        return patterns.asymmetricItakura(ts_x, ts_y, window,
dist, pattern, normalized)
    elif pattern == "typeIa":
        return patterns.typeIa(ts_x, ts_y, window, dist,
pattern, normalized)
    elif pattern == "typeIb":
        return patterns.typeIb(ts_x, ts_y, window, dist,
pattern, normalized)
    elif pattern == "typeIc":
        return patterns.typeIc(ts_x, ts_y, window, dist,
pattern, normalized)
    elif pattern == "typeId":
        return patterns.typeId(ts_x, ts_y, window, dist,
pattern, normalized)
    elif pattern == "typeIas":
        return patterns.typeIas(ts_x, ts_y, window, dist,
pattern, normalized)
    elif pattern == "typeIbs":
        return patterns.typeIbs(ts_x, ts_y, window, dist,
pattern, normalized)
    elif pattern == "typeIcs":
        return patterns.typeIcs(ts_x, ts_y, window, dist,
pattern, normalized)
    elif pattern == "typeIds":
        return patterns.typeIds(ts_x, ts_y, window, dist,
pattern, normalized)
    elif pattern == "typeIIa":
        return patterns.typeIIa(ts_x, ts_y, window, dist,
pattern, normalized)
    elif pattern == "typeIIb":
        return patterns.typeIIb(ts_x, ts_y, window, dist,
pattern, normalized)
    elif pattern == "typeIIc":
        return patterns.typeIIc(ts_x, ts_y, window, dist,
pattern, normalized)
    elif pattern == "typeIId":
        return patterns.typeIId(ts_x, ts_y, window, dist,
pattern, normalized)
    elif pattern == "typeIIIc":
        return patterns.typeIIIc(ts_x, ts_y, window, dist,
pattern, normalized)
    elif pattern == "typeIVc":
        return patterns.typeIVc(ts_x, ts_y, window, dist,
pattern, normalized)
    elif pattern == "mori2006":
        return patterns.mori2006(ts_x, ts_y, window, dist,
pattern, normalized)



def __backtrack(D, max_x, max_y):
    path = []

    i, j = max_x, max_y
    path.append((i, j))
    while i > 0 or j > 0:
        diag_cost = float('inf')
        left_cost = float('inf')
        down_cost = float('inf')

        if (i > 0) and (j > 0):
            diag_cost = D[i - 1][ j - 1]
        if i > 0:
            left_cost = D[i - 1][ j ]
        if j > 0:
            down_cost = D[i][ j - 1]

        if (diag_cost <= left_cost and diag_cost <= down_cost):
            i, j = i - 1, j - 1
        elif (left_cost < diag_cost and left_cost < down_cost):
            i = i - 1
        elif (down_cost < diag_cost and down_cost < left_cost):
            j = j - 1
        elif i <= j:
            j = j - 1
        else:
            i = i - 1

        path.append((i, j))

    path.reverse()
    return path



def __reduce_by_half(x):
    return [(x[i // 2] + x[1 + i // 2]) / 2 for i in range(0,
len(x), 2)]
```

```
def __expand_window(path, len_x, len_y, radius):
    path_ = set(path)
    for i, j in path:

        for a, b in ((i + a, j + b) for a in range(-radius,
radius + 1) for b in range(-radius, radius + 1)):
            path_.add((a, b))

    window_ = set()
    for i, j in path_:
        for a, b in ((i * 2, j * 2), (i * 2, j * 2 + 1), (i * 2
+ 1, j * 2), (i * 2 + 1, j * 2 + 1)):
            window_.add((a, b))
    window = []
    start_j = 0
    for i in range(0, len_x):
        new_start_j = None
        for j in range(start_j, len_y):
            if (i, j) in window_:
                window.append((i, j))
                if new_start_j is None:
                    new_start_j = j
            elif new_start_j is not None:
                break
        start_j = new_start_j

    return window
def __getSCwindow(windowsize, len_x, len_y):
    window = []
    for i in range(len_x):
        for j in range(len_y):
            if abs(i - j) < windowsize:
                window.append((i, j))
    return window
def __getItakurawindow(len_x, len_y):
    window = []
    for i in range(len_x):
        for j in range(len_y):
            if  ((j < 2 * i) and (i <= 2 * j) and (i >= len_x -
1 - 2 * (len_y - j)) and (j > len_y - 1 - 2 * (len_x - i))):
                window.append((i, j))
    return window

def __getpaliwalwindow(windowsize, len_x, len_y):
    window = []
    for i in range(len_x):
        for j in range(len_y):
            if abs(i*len_y/len_x - j) < windowsize:
                window.append((i, j))
    return window
```

References

[1] T. K. Vintsyuk, “Speech discrimination by dynamic programming,” *Cybernetics*, vol. 4, no. 1, pp. 52–57, Jan. 1968.

[2] H. Sakoe and S. Chiba, “Dynamic programming algorithm optimization for spoken word recognition,” *IEEE Trans. Acoust. Speech Signal Process.*, vol. 26, no. 1, pp. 43–49, Feb. 1978.

[3] P. Senin, “Dynamic time warping algorithm review,” *Inf. Comput. Sci. Dep. Univ. Hawaii Manoa Honol. USA*, pp. 1–23, 2008.

[4] J. Blackburn and E. Ribeiro, “Human Motion Recognition Using Isomap and Dynamic Time Warping,” in *Human Motion – Understanding, Modeling, Capture and Animation*, A. Elgammal, B. Rosenhahn, and R. Klette, Eds. Springer Berlin Heidelberg, 2007, pp. 285–298.

[5] C. S. Myers and L. R. Rabiner, “A Comparative Study of Several Dynamic Time-Warping Algorithms for Connected-Word Recognition,” *Bell Syst. Tech. J.*, vol. 60, no. 7, pp. 1389–1409, Sep. 1981.

[6] “Python Data Analysis Library — pandas: Python Data Analysis Library.” [Online]. Available: http://pandas.pydata.org/. [Accessed: 03-Jun-2015].

[7] S. Salvador and P. Chan, “Toward accurate dynamic time warping in linear time and space,” *Intell. Data Anal.*, vol. 11, no. 5, pp. 561–580, Jan. 2007.

[8] E. Keogh and M. Pazzani, “Derivative Dynamic Time Warping,” in *Proceedings of the 2001 SIAM International Conference on Data Mining*, 0 vols., Society for Industrial and Applied Mathematics, 2001, pp. 1–11.

[9] F. Itakura, “Minimum prediction residual principle applied to speech recognition,” *IEEE Trans. Acoust. Speech Signal Process.*, vol. 23, no. 1, pp. 67–72, Feb. 1975.

[10] K. K. Paliwal, A. Agarwal, and S. S. Sinha, “A modification over Sakoe and Chiba’s dynamic time warping algorithm for isolated word recognition,” *Signal Process.*, vol. 4, no. 4, pp. 329–333, 1982.

[11] T. Giorgino, “Computing and Visualizing Dynamic Time Warping Alignments in R: The dtw Package,” *J. Stat. Softw.*, vol. 31, no. 7, pp. 1–24, 2009.

[12] “mlpy - Machine Learning Python.” [Online]. Available: http://mlpy.sourceforge.net/. [Accessed: 08-Jun-2015].

[13] C. A. Ratanamahatana and E. Keogh, “Everything you know about dynamic time warping is wrong,” in *Third Workshop on Mining Temporal and Sequential Data*, 2004, pp. 22–25.

[14] D. J. Berndt and J. Clifford, “Using Dynamic Time Warping to Find Patterns in Time Series.,” in *KDD Workshop*, 1994, pp. 359–370.

[15] A. Akila and E. Chandra, “Slope finder—a distance measure for DTW based isolated word speech recognition,” *Int J Eng Comput Sci*, vol. 2, no. 12, pp. 3411–3417, 2013.

[16] L. R. B.-H. Juang, “Fundamentals of Speech Recognition Prentice Hall,” *Englewood Cli S*, 1993.

[17] A. Mori, S. Uchida, R. Kurazume, R.-I. Taniguchi, T. Hasegawa, and H. Sakoe, “Early Recognition and Prediction of Gestures,” in *18th International Conference on Pattern Recognition, 2006. ICPR 2006*, 2006, vol. 3, pp. 560–563.

[18] S. Chu, E. J. Keogh, D. M. Hart, M. J. Pazzani, and others, “Iterative Deepening Dynamic Time Warping for Time Series.,” in *SDM*, 2002, pp. 195–212.

[19] E. J. Keogh and M. J. Pazzani, “Scaling up dynamic time warping for datamining applications,” in *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2000, pp. 285–289.

[20] S.-W. Kim, S. Park, and W. W. Chu, “An Index-Based Approach for Similarity Search Supporting Time Warping in Large Sequence Databases,” in *Proceedings of the 17th International Conference on Data Engineering*, Washington, DC, USA, 2001, pp. 607–614.

[21] E. Keogh and C. A. Ratanamahatana, “Exact indexing of dynamic time warping,” *Knowl. Inf. Syst.*, vol. 7, no. 3, pp. 358–386, 2005.

[22] G. Al-Naymat, S. Chawla, and J. Taheri, “SparseDTW: A Novel Approach to Speed Up Dynamic Time Warping,” in *Proceedings of the Eighth Australasian Data Mining Conference - Volume 101*, Darlinghurst, Australia, Australia, 2009, pp. 117–127.

[23] D. Lemire, “Faster Retrieval with a Two-Pass Dynamic-Time-Warping Lower Bound,” *Pattern Recognit.*, vol. 42, no. 9, pp. 2169–2180, Sep. 2009.

[24] X. Wang, A. Mueen, H. Ding, G. Trajcevski, P. Scheuermann, and E. Keogh, “Experimental comparison of representation methods and distance measures for time series data,” *Data Min. Knowl. Discov.*, vol. 26, no. 2, pp. 275–309, Feb. 2012.

[25] “NumPy — Numpy.” [Online]. Available: http://www.numpy.org/. [Accessed: 04-May-2015].

[26] “matplotlib: python plotting — Matplotlib 1.4.3 documentation.” [Online]. Available: http://matplotlib.org/. [Accessed: 04-May-2015].

[27] S. B. Needleman and C. D. Wunsch, “A general method applicable to the search for similarities in the amino acid sequence of two proteins,” *J. Mol. Biol.*, vol. 48, no. 3, pp. 443–453, Mar. 1970.