Efficient Dynamic Time Warping for Big Data Streams

Rafael M. Martins
Dept. of Computer Science and Media Technology
Linnaeus University
Växjö, SE
rafael.martins@lnu.se

Andreas Kerren
Dept. of Computer Science and Media Technology
Linnaeus University
Växjö, SE
andreas.kerren@lnu.se

***Abstract*—Many common data analysis and machine learning algorithms for time series, such as classification, clustering, or dimensionality reduction, require a distance measurement between pairs of time series in order to determine their similarity. A variety of measures can be found in the literature, each with their own strengths and weaknesses, but the Dynamic Time Warping (DTW) distance measure has occupied an important place since its early applications for the analysis and recognition of spoken word. The main disadvantage of the DTW algorithm is, however, its quadratic time and space complexity, which limits its practical use to relatively small time series. This issue is even more problematic when dealing with streaming time series that are continuously updated, since the analysis must be re-executed regularly and with strict running time constraints. In this paper, we describe enhancements to the DTW algorithm that allow it to be used efficiently in a streaming scenario by supporting an *append* operation for new time steps with a linear complexity when an exact, error-free DTW is needed, and even better performance when either a Sakoe-Chiba band is used, or when a *sliding window* is the desired range for the data. Our experiments with one synthetic and four natural data sets have shown that it outperforms other DTW implementations and the potential errors are, in general, much lower than another state-of-the-art approximated DTW technique.**

***Index Terms*—time series, dynamic time warping, streaming**

I. Introduction

A time series is a dynamic sequence of (potentially multidimensional) elements that changes with time, i.e., the value of each step of a time series is an observation obtained chronologically. Time series usually consist of large and complex data sequences, and the interest for them arises from their presence in various domains of knowledge in science, engineering, business, healthcare, to name just a few [1]–[4].

Many common data analysis and machine learning algorithms for time series, such as classification [1], clustering [4], or dimensionality reduction [5], require a distance measurement between pairs of time series in order to determine their similarity. A variety of measures can be found in the literature, from the common Euclidean distance to the more sophisticated so-called *elastic* similarity measures [6], [7], each with their own strengths and weaknesses in terms of performance and speed, which dictate in which application scenarios they may be successfully applied [8].

Since its early applications on the analysis and recognition of spoken word [9], the Dynamic Time Warping (DTW) distance measure has become an important tool for time series analysts and has been a constant subject of important research in the area [10]. Its need comes from the limitations of other distance measures when it comes to matching natural time series that are inherently similar in shape, i.e., represent the same underlying phenomenon, but differ in both total length and the length of their composing patterns. For example, consider the problem of comparing the audio recordings of two different persons speaking the same list of words. While the two time series will have similarities, each person has their own speech pace and rhythm, different letters or syllables might be highlighted, and different accents will add complex, non-linear disparities. The use of DTW compensates for these differences by searching for the best non-linear matching for a pair of time series before computing the distance between them. It has been demonstrated that DTW significantly improves the results of different learning techniques for time series [2], [3].

The main disadvantage of the DTW algorithm is, however, its time and space complexity of $O(N^2)$, which limits its practical use to the analysis of time series that contain, at most, a few thousand time steps [11]. This issue is even more problematic when dealing with time series that are continuously updated, since the entire data set is not available at the beginning of the process, and the analysis must be re-executed regularly and with strict running time constraints. Such a scenario of continuously monitoring the similarities of multiple data streams can be found in many application domains, such as financial analysis [12] or biological data [13].

According to Zhu and Shasha [12], in any system that manages data streams with a sufficiently good performance: (a) streams are updated through the insertion of new elements (with little to no change of previous steps); (b) streams are treated as never-ending sequences of events, not as sets of elements; (c) one-pass algorithms must be used, due to the sheer size of streams; and (d) sacrificing accuracy for speed is acceptable. In this paper, we describe a set of enhancements to the DTW algorithm, collectively called `SlideDTW`, that follow these guidelines and allow it to be used efficiently in a streaming scenario. The proposed incremental computation of DTW supports an *append* operation for new time steps with a complexity of $O(N)$ when an exact, error-free DTW

In Proceedings of the IEEE International Conference on Big Data (Big Data '18): Workshop on Real-time & Stream Analytics in Big Data & Stream Data Management, pages 2924-2929, Seattle, USA, 2018. IEEE. DOI: 10.1109/BigData.2018.8621878

is needed, or a constant complexity when either a Sakoe-Chiba band is acceptable, or when a *sliding window* is the desired range for the data. While the latter two situations result in approximate computations and potentially incur on lower acccuracy, our experiments with one synthetic and four natural data sets have shown that our proposed techniques outperform other implementations (in terms of speed) and the errors are, in general, much lower than another state-of-the-art approximated DTW technique.

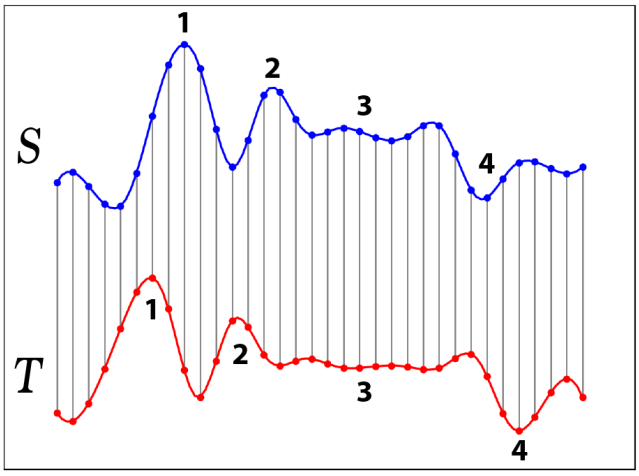
The rest of this paper is organized as follows: in Section II we introduce some of the DTW background needed to clearly understand the proposed improvements; in Section III we describe in details the `SlideDTW` algorithms for implementing an incremental DTW that supports the *append* operation—and the computation of a DTW with a sliding window—with high performance; in Section IV we discuss some of the related work and how they compare to our approach; in Section V we show the results of our experiments on the performance and accuracy of the described approach; and in Section VI we offer some final remarks and ideas for future work.

II. Background

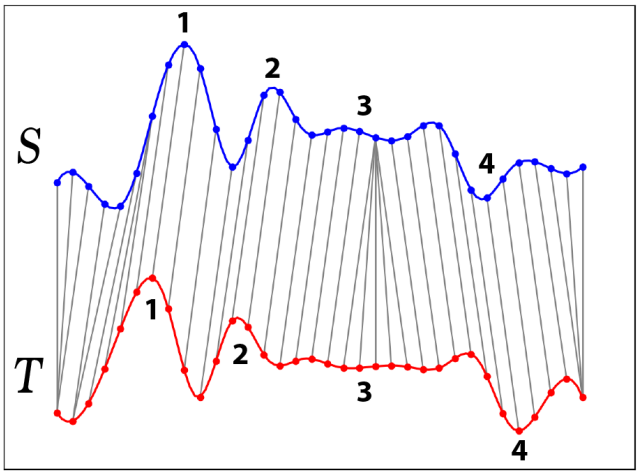
Arguably the simplest way to compute the similarity between two time series S and T is to use the well-known Euclidean distance [14]. In this case, each time series is seen as a vector of real values (its time steps), and the distance measure can be computed in a straightforward manner. In case the time series are multivariate, i.e., each time step i (S_i, T_i) is a vector of measurements, a *local* distance measure δ is necessary in order to compare pairs of multivariate time steps. Again, it is possible (and common) to use the Euclidean distances between multivariate time steps for the *local* comparisons, but any other distance measure between two vectors may be used. This process is summarized in Equation 1.

$$E(S,T) = \sqrt{\sum_i \delta(S_i, T_i)^2} \tag{1}$$

Comparing the time steps of two time series in this way leads to an arrangement similar to the one shown in Figure 1(a), where the vertical lines connect the time steps that are compared with δ. A quick visual inspection of the two time series in Figure 1 is enough to realize that, although they may not match perfectly, there are many similarities in their shapes, namely two consecutive peaks (1 and 2) and a valley (4), separated by a period of average values (3). These similar features are not captured when applying the Euclidean distance to this pair of time series and comparing their steps linearly. Take, for example, Peak 1 from time series T: it is directly compared with some of the lowest values of time series S, which come right before the rise that leads to the equivalent Peak 1 from S. While it may seem like shifting T to the right would fix this issue, that would make the misalignment from Valley 4 even worse, since it is already shifted to the right (when compared to its counterpart from S). These and other similar problems that can be observed in the other features happen due to the two time series being *non-linearly* shifted from each other, i.e., not only their features are not aligned, but their misalignment varies along time.



(a) Euclidean Distance



(b) DTW Distance

Figure 1: Comparison between the Euclidean and the DTW distances of two time series S and T, regarding the matching of steps for comparison.

A. Dynamic Time Warping (DTW)

The use of DTW to compare two time series aims to solve exactly that problem: to find an optimal non-linear alignment between the time series, so that their comparison fits as best as possible their matching features. Figure 1(b) shows an example of the result of DTW when applied to S and T. It can be observed from Figure 1(b) that matching features between S and T are now being compared correctly. Although the final value of the distance is still computed using δ between pairs of time steps, one from each of the series, as shown in Equation 2, now the algorithm searches for different possible

pairings that may be better than the default ones from the Euclidean distance. Notice that in Equation 2, as opposed to Equation 1, the indexes of the time steps being compared (i, j) are potentially different at each execution of δ.

$$DTW(S,T) = \sum_{i,j} \delta(S_i, T_j) \tag{2}$$

The question then is how to find the optimal sequence of pairs (i, j)—called the warping path w—to apply δ to in Equation 2 and get the final optimal DTW distance. Algorithm 1 describes a recursive procedure to obtain w and compute $DTW(S,T)$, using the notation $S_{\rightarrow i} = [S_1, \ldots, S_i]$ (the subsequence of S from the 1^{st} to the i^{th} time step).

Input: Time series S, T; indexes i, j
Result: $DTW(S_{\rightarrow i}, T_{\rightarrow j})$

$$DTW(S_{\rightarrow i}, T_{\rightarrow j}) = \delta(S_i, T_j) + \begin{cases} 0, & \text{if } i = j = 1 \\ DTW(S_{\rightarrow i}, T_{\rightarrow j-1}), & \text{if } i = 1 \\ DTW(S_{\rightarrow i-1}, T_{\rightarrow j}), & \text{if } j = 1 \\ \mathbf{min}\{ & \\ \quad DTW(S_{\rightarrow i-1}, T_{\rightarrow j-1}), & \\ \quad DTW(S_{\rightarrow i-1}, T_{\rightarrow j}), & \\ \quad DTW(S_{\rightarrow i}, T_{\rightarrow j-1})\}, & \text{otherwise} \end{cases}$$

Algorithm 1: Recursive algorithm for DTW

In summary, we start the recursion by running $DTW(S,T) = DTW(S_{\rightarrow i}, T_{\rightarrow j})$, for $i = |S|$ and $j = |T|$, i.e., we want the distance for the full sequences, up to their last elements (the pair (i, j) formed by the last two elements of S and T will always be the last pair of w). Then, recursively, we choose the *previous* pair of w by finding which pair among $\{(i-1, j-1), (i-1, j), (i, j-1)\}$ results in the minimum DTW distance for the corresponding subsequences. In case either i or j equals 1 (meaning the first element of a time series was reached), the selection is trivial. The pair $(1, 1)$ ends the recursion and is always the first element of w. The final value of the DTW distance results in the sum of $\delta(S_i, T_j)$ for all $(i, j) \in w$, as described in Equation 2.

III. Incremental DTW for Big Data Streams

In this section, we describe some simple mechanisms to allow for efficient on-the-fly DTW computations of streaming time series, which we collectively call `SlideDTW`. In order to explain these mechanisms, we must first introduce an alternative way to compute the DTW between two time series, by means of the *Accumulated Cost Matrix* ($D_{S,T}$) between two time series S and T. The procedure is described in Algorithm 2 and illustrated in Figure 2 with a simple (artificial) example.

The cells of the first row and first column of the $D_{S,T}$ matrix are computed in a straightforward manner, by accumulating

Input: Time series S, T
Result: Accumulated Cost Matrix $D_{S,T}$
D = empty $|S| \times |T|$ matrix;
$D(1,1) = \delta(S_1, T_1)$;
for $i \leftarrow 2$ **to** $|S|$ **do**
  $D(1,i) = \delta(S_i, T_1) + D(S_{i-1}, T_1)$;
end
for $j \leftarrow 2$ **to** $|T|$ **do**
  $D(j,1) = \delta(S_1, T_j) + D(S_1, T_{j-1})$;
end
for $i \leftarrow 2$ **to** $|S|$ **do**
  for $j \leftarrow 2$ **to** $|T|$ **do**
    $D(i,j) = \delta(S_i, T_j) +$
    $min(D(S_{i-1}, T_{j-1}), D(S_{i-1}, T_j), D(S_i, T_{j-1}))$;
  end
end

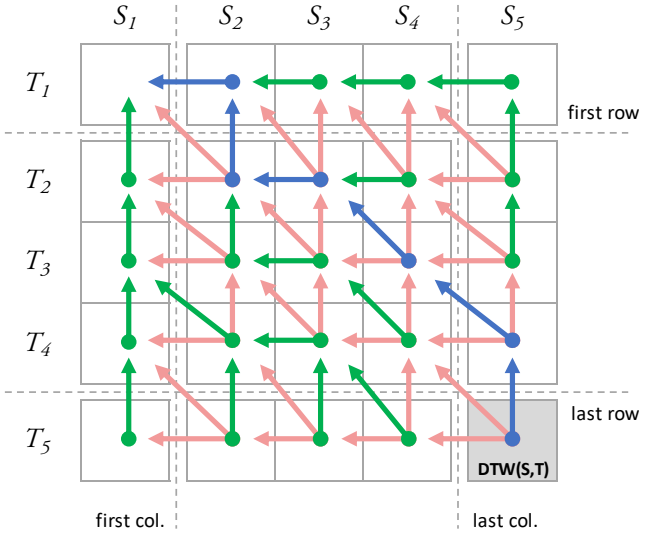
Algorithm 2: Accumulated Cost Matrix $D_{S,T}$

the costs of previous cells of the same row or column. These are computed by the first two `for`'s in Algorithm 2 and illustrated in Figure 2 (a) by the single arrows coming out of each cell from the first row and column, pointing to the previous cell. The direction of the arrows in Figure 2 (a) indicate *dependency*, i.e., which cells must be calculated before the one which originates the arrow. For example: cell (S_1, T_2) *depends* only on cell (S_1, T_1), while cell (S_2, T_2) *depends* on cells (S_2, T_1), (S_1, T_2), and (S_1, T_1). All the rest of the cells are computed as an accumulation of the minimum DTW cost of all their dependencies.

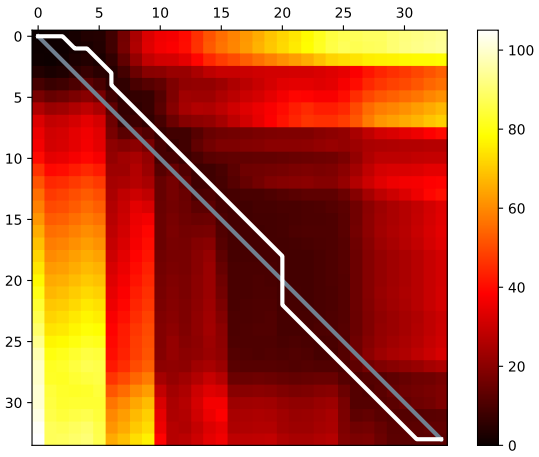
In Figure 2 (a), the green (or blue) arrows represent the previous cell that was "chosen" (i.e., they satisfied the **min** constraint), while the red arrows point to the previous cells who were considered, but not chosen (due to not being the **min**). After all the cells are computed, as a side effect, the last cell of the last column (or row) of $D_{S,T}$ stores the final DTW value between S and T, i.e., $DTW(S,T) = D_{S,T}(|S|, |T|)$, which is the same as computed with the recursive version of DTW shown in Algorithm 1. The warping path w, illustrated in Figure 2 (a) by the blue arrows, can be extracted by starting from $D_{S,T}(|S|, |T|)$ and back-tracking through $D_{S,T}$ by following the "chosen" cells.

To illustrate the results of Algorithm 2, the $D_{S,T}$ for time series S and T from Figure 1 is shown in Figure 2 (b), along with the warping paths for the DTW (white) and the Euclidean distance (gray). It is possible to visualize how the path of the Euclidean distance crosses high values (e.g., the regions around cells 5, 10, or 25), while the DTW searches for paths that pass through lower-valued cells.

Considering Figure 2 (a), one important observation can be made: the computation of each row of the $D_{S,T}$ matrix depends only on the previous row, and the computation of each column of $D_{S,T}$ depends only on the previous column. If one additional time step is *appended* to one of the time series (or both), we need only the last row and column of the



(a) Simple example to illustrate the computation of $D_{S,T}$



(b) $D_{S,T}$ and warping paths for S and T from Figure 1

Figure 2: Examples of Accumulated Cost Matrix ($D_{S,T}$) and warping paths. (a) Simple example to illustrate the computation of $D_{S,T}$ and the choice of the warping path starting with the last cell (blue arrows). (b) The $D_{S,T}$ for the time series from Figure 1, with the warping paths for DTW (white) and Euclidean distance (gray).

$D_{S,T}$ to compute the DTW between the updated time series. This observation leads to the definition of Algorithm 3, an incremental computation of DTW for big data streams. The procedure is executed every time new time steps s and t are *appended* to S and T respectively, i.e., when new data is added to the stream, and it efficiently updates the value of $DTW(S,T)$ to reflect the new data.

The input for Algorithm 3 includes the previous state of the two time series S and T (before the new steps are appended), the two new steps s and t, and the last row (R^Ω) and last column (C^Ω) of the accumulated cost matrix $D_{S,T}$. Initially, two new empty lists are generated; these will function as the new row (R) and column (C) that would normally be appended

Input: S, T, R^Ω, C^Ω; new time steps s, t
Result: S, T, R^Ω, C^Ω; $DTW(S,T)$
R, C = empty lists of size $|S|+1$ and $|T|+1$;
$R_1 \leftarrow \delta(t, S_1) + R^\Omega_1$;
for $i \leftarrow 2$ **to** $|S|$ **do**
  $R_i \leftarrow \delta(t, S_i) + \mathbf{min}(R_{i-1}, R^\Omega_{i-1}, R^\Omega_i)$;
end
$C_1 \leftarrow \delta(s, T_1) + C^\Omega_1$;
for $j \leftarrow 2$ **to** $|T|$ **do**
  $C_j \leftarrow \delta(s, T_j) + \mathbf{min}(C_{j-1}, C^\Omega_{j-1}, C^\Omega_j)$;
end
$DTW(S,T) \leftarrow \delta(s,t) + \mathbf{min}(R_{|S|}, C_{|T|}, R^\Omega_{|S|})$;
$R_{|S|+1} \leftarrow C_{|T|+1} \leftarrow DTW(S,T)$;
$R^\Omega \leftarrow R$; $C^\Omega \leftarrow C$;
$S_{|S|+1} \leftarrow s$; $T_{|T|+1} \leftarrow t$;

Algorithm 3: Update DTW by *appending* new time steps

to $D_{S,T}$. The new row and column are computed in a very similar way as in $D_{S,T}$, i.e., by accumulating the minimum value of the costs of the previous steps (their *dependencies*). In order to do that, as observed earlier, we do not need to store the entire $D_{S,T}$ matrix, but only its last row and last column, in this case R^Ω and C^Ω respectively. Note that, in the first execution of the algorithm, in case S and T are empty, R^Ω_1 and C^Ω_1 should be taken as $\inf$ (the rest of the algorithm remains unchanged). The final step is to compute the new $DTW(S,T)$ based on the last elements of R and C, which is then appended to the end of the new lists. The third element of the **min** operator is simply the current $DTW(S,T)$, i.e., the last element of either R^Ω or C^Ω (using $C^\Omega_{|T|}$ would not change the result). Finally, R and C replace the current R^Ω and C^Ω, which are discarded, and s and t are appended to S and T.

The described procedure works for appending one single time step at a time to each time series; in case more than one step must be appended to the stream at a time, the algorithm can simply be repeated as many times as needed (which would be the same as computing several new rows and columns of the accumulated cost matrix $D_{S,T}$).

It is important now to discuss the differences between Algorithm 2 and Algorithm 3 and why the latter is more suitable for streaming data than the former. One important use case for the real-time analysis of streaming time series is to re-compute the distance between pairs of time series every time new data arrives. Using the default implementation of DTW as described in Algorithm 2 (which is the one found in most software libraries) will lead to a quadratic time and space complexity of $O(nm)$ as the data arrives from the stream. Every update of the DTW for each pair of time series (S,T) requires the entire computation of the $D_{S,T}$ matrix in two nested *for* loops. As can be seen in the quantitative results shown in Section V, such a performance constraint can quickly become infeasible for big data streams. On the other hand, the

incremental approach described in Algorithm 3 removes that constraint and brings the complexity in this use case down to $O(n+m)$; the results are now obtained from performing two separate (instead of nested) *for* loops. The space complexity is also reduced to $O(n+m)$, since only two additional lists (instead of an entire matrix) must be stored in memory.

There are, however, two important disadvantages that must be highlighted for the approach described in Algorithm 3. The first one is that, for each $DTW(S,T)$ between each pair of time series, their corresponding R^{Ω} and C^{Ω} must be stored at all times in-between executions of the algorithm. In both cases, the entire series S and T must be stored permanently and reused for each time step, but for Algorithm 2 the entire matrix is recomputed and discarded at each step, which allows temporary memory to be used. This is not the case for Algorithm 3.

The second point is related to the first: the *accuracy* of the final $DTW(S,T)$ computed from Algorithm 3 depends on the user-defined space reserved for storing R^{Ω} and C^{Ω} (which we will call σ). As long as there is space available to store the two complete lists (i.e., $\sigma \geq |R^{\Omega}| + |C^{\Omega}|$), the computed $DTW(S,T)$ is error-free, since its computation mirrors exactly the computation of $D_{S,T}$. When the space limit is reached, however, then old time steps must be discarded in order to make space for new ones, which will have an effect on the accuracy of the results. Such a limitation is very similar to defining a Sakoe-Chiba constraint, as discussed in Section IV, with a band width of σ, and shares its strengths and weaknesses. The effects of this are illustrated and discussed further in Section V.

A. Incremental DTW with a Sliding Window

While Algorithm 3 provides a way to compute, in real time, an accurate DTW for streams considering them in their entirety (i.e., the distance between two streams considering all time steps since the beginning of the data collection), sometimes another use case is relevant: to obtain the DTW distance between two streams considering only a *sliding window* of λ time steps [15]. Examples of applications for this are speech recognition [16] and text mining [17].

With any standard implementation of DTW (such as the ones introduced in Sections II and IV), the workflow is simple: for a window length of λ, after new time steps are appended to the streams, discard older time steps and only consider the most recent ones, such that

$$DTW(S,T) = DTW(S_{\lambda\rightarrow}, T_{\lambda\rightarrow}), \tag{3}$$

where $S_{\lambda\rightarrow} = (S_{i-\lambda}, \ldots, S_i)$ and $T_{\lambda\rightarrow} = (T_{j-\lambda}, \ldots, T_j)$, then re-compute $DTW(S,T)$ normally.

The same is not true, however, for Algorithm 3, due to its reliance on accumulated values in a similar way to the $D_{S,T}$. As can be verified from Figure 2 (a), a change in the first row or column (which is the case when an older time step is discarded) causes a cascade of changes that affects all other rows and columns, potentially forcing the re-computing of the entire $D_{S,T}$ and resulting in an entirely new warping path. In order to achieve the gains offered by Algorithm 3, such cost is prohibitive.

We propose DTW_{λ} (Equation 4) to improve on this and provide an approximation of the computation of DTW between two time series S and T for a *sliding window* of length λ:

$$DTW_{\lambda}(S,T) = DTW(S,T) - DTW(S_{\rightarrow\lambda}, T_{\rightarrow\lambda}), \tag{4}$$

where $S_{\rightarrow\lambda} = (S_1, ..., S_{i-\lambda})$ and $T_{\rightarrow\lambda} = (T_1, ..., T_{j-\lambda})$.

Since the full DTW for two time series of the same length i is computed as the accumulated cost of the warping of all i time steps, we assume that the DTW for the window length of $\lambda < i$ (i.e., the last λ time steps) is approximately the value of the full $DTW(S,T)$ minus the value of the DTW for the first $i-\lambda$ time steps. Conveniently, when using Algorithm 3, $DTW(S_{\rightarrow\lambda}, T_{\rightarrow\lambda})$ is already computed as a partial step to $DTW(S,T)$. All we need to do is to augment Algorithm 3 to store the last λ values obtained from $DTW(S,T)$ in a list

$$\Lambda = \big[DTW(S_{\rightarrow\lambda}, T_{\rightarrow\lambda}), \ldots, DTW(S,T)\big].$$

Whenever new time steps are added to S and T, the oldest (Λ_1) is removed and a new one is inserted at the end of the list, so that $\Lambda_{\lambda} \leftarrow DTW(S,T)$. Finally, $DTW_{\lambda}(S,T) = DTW(S,T) - \Lambda_1$.

The quantitative results, as discussed in Section V-C, confirm that the value is not exact, but yields—in most cases—a very good approximation of the values that a fully quadratic DTW computation would give. Coupled with the significantly better performance, it stands out as a potentially useful tool for time series analysts.

IV. Related Work

In this section, we discuss existing works related to modifications of the original DTW algorithm regarding performance improvements and support for streaming.

A. DTW Performance Improvements

In its raw form, the procedure described in Algorithm 1 is exponential, due to a recursive re-evaluation of entire candidate paths for every new pair appended to the optimal warping path. This is usually brought down to a quadratic complexity using dynamic programming, as described in Section III, which is still prohibitive when dealing with big streams of time series. One common way to deal with this is to add global constraints to the search for candidate paths, with techniques such as Itakura's Parallelogram [18] or the Sakoe-Chiba band [9]. With a Sakoe-Chiba band of constant width σ, warping paths never deviate more than σ steps from the diagonal of the $D_{S,T}$, which not only brings down the complexity to $O(\sigma N)$, but may also yield better results, for example, in machine learning tasks [19].

Silva and Batista [20] observed that the Euclidean distance works as an upper bound of DTW—the DTW distance between two time series is never *larger* than the Euclidean distance—and developed `PrunedDTW` as an exact, error-free speed-up of the original technique. The DTW computation remains the same as the original, but it is *pruned* to avoid extra

unnecessary computations when the upper bound is reached (i.e., the optimal solution cannot be improved by following a pruned path). As a disadvantage, in order to set its upper bound, `PrunedDTW` requires the preliminary computation of the Euclidean distance between the two *full* time series. In a streaming scenario, the full time series are not available beforehand, so the Euclidean distance must be re-computed at each step to ensure the upper bound is correct (as we have done in the experiments in Section V). Although the computation of the Euclidean distances at each step might be accelerated using a similar method as ours (by re-using the components of the last computation), in order to maintain its error-free results the entire $D_{S,T}$ matrix would need to be re-computed due to the change in the upper bound value.

When an exact DTW is not absolutely necessary, some techniques have been proposed to approximate its value with faster computation times by using data abstraction [21], [22]. `FastDTW` [11] estimates the DTW using a multi-level approach, starting from a solution to sampled-down, coarse versions of the time series, and recursively refining that solution to higher-resolution versions. The refinement is done by locally adjusting the coarse-level warping path, looking for solutions in the *neighborhood* of the simplified warping path (according to a *radius* parameter), until a final solution is found to the original time series. While the performance gain is significant, since the complexity of `FastDTW` is $O(N)$, the procedure only works well in cases when the optimal warping path is composed of nodes that are located near the initial coarse-level solutions; otherwise, the solutions can be far from the optimal (as our experiments in Section V show).

Another example use case for approximated DTW is when, among a full set of time series, only the *nearest* ones to a certain reference S are needed, i.e., when searching for nearest neighbors or querying large databases [23]. In such cases, an approximated DTW can be useful for discarding time series that are below a threshold of similarity, leaving only a small *filtered* subset of candidate series where the full DTW is then computed. A well-known example is LB_{Keogh}, where upper (U) and lower (L) bounds are defined around a time series S according to a width b, such that

$$\begin{aligned} E_i &= \big[\mathbf{max}\{1, i-b\}, \mathbf{min}\{|S|, i+b\}\big], \\ U_i(S) &= \mathbf{max}\{S_j | j \in E_i\}, \\ L_i(S) &= \mathbf{min}\{S_j | j \in E_i\}. \end{aligned} \tag{5}$$

An Euclidean-like local distance δ is then computed for each step T_i of a candidate series T, such that $\delta = 0$ if T_i is within the bounds ($L_i(S) < T_i < U_i(S)$), or it is the Euclidean distance between T_i and the closest bound (either U_i or L_i) otherwise. LB_{Keogh} only gives a rough estimation of the real DTW value, which is fast and works well enough as a filter before the exact DTW is applied but not as a replacement for the DTW itself.

B. Support for Streaming

In order to deal with the problem of monitoring, processing, and comparing multiple data streams, Zhu and Shasha [12] proposed StatStream, a system that uses a sliding window to compute different statistics about streaming time series, including correlation (which can be seen as the opposite of a distance/dissimilarity). They reported being able to monitor in real-time up to $10,000$ streams, but the system does not support DTW.

The problem of reducing the complexity of DTW for streaming cases has been considered recently in the work of Oregi et al. [8]. The authors propose ODTW (On-line DTW), which also includes a form of incremental computation of DTW, complemented by a mechanism for weighting down the pairs that compose the warping path according to their age—older pairs of time steps have less influence in the final value of the distance. Even though the authors present their technique as a generalization of the DTW (i.e., it can be exact with the correct parameter settings), the results are, in general, different than the original DTW, which may or may not be desirable for the analyst. Additionally, the mechanism of weighting down the older pairs from the warping path is a parameterized setting, and it has not been applied for computing the DTW with a sliding window, as we proposed.

Khan et al. [24] also used an incremental implementation of DTW in the context of matching a series of user's gestures with a reference series of gestures from an expert, for the domain of physical therapy. The authors' implementation differs from ours in that the user's series are constantly growing, while the expert's series is static, so the matching does not happen between two streams. Also, there is no support for re-using previous computations in order to compute the DTW between a sliding window of time steps.

V. Evaluation

We start by evaluating the techniques described in Section III using five data sets, one synthetic and four natural, according to two criteria: *performance* and *accuracy*. The *performance* experiments are intended to show that the reduced complexity of the incremental DTW is indeed observed in practice, while the *accuracy* experiments measure the magnitude of the errors of the results when compared to the full, exact DTW results (if applicable). For both measures, a smaller value is better.

Our described incremental DTW techniques are compared to the following other DTW techniques: (a) the `DTW`, implemented according to Algorithm 2 using Python and optimized with Numba's just-in-time compiler; (b) the `FastDTW` (described in Section IV), implemented in Cython[1]; and (c) the `PrunedDTW` (described in Section IV), re-implemented from the original code in Python and also optimized with Numba's just-in-time compiler.

The five data sets used were chosen for the paper after a pilot run of the experiments together with a set of 80 additional data sets (taken from the UCR Time Series Classification Archive [25]), during which we identified a few different profiles of results, i.e., groups of data sets for which the results

[1] https://github.com/slaypni/fastdtw

behaved similarly. This final set of 5 data sets exemplifies each of the observed profiles.

1) **Random:** 100 uniformly-sampled random time series with 600 time steps each.
2) **Tweets:** 100 time series extracted from the Nordic Tweet Stream project [26], in the period of May 2016. Each time series represents the number of tweets that included one of the 100 most popular hashtags each hour of the period.
3) **Earthquakes:** 322 time series with 512 time steps each, obtained from the UCR Time Series Classification Archive [25]. The data comes originally from Northern California Earthquake Data Center, and each time step is an averaged reading of seismic activity for one hour.
4) **Ham:** 109 time series with 431 time steps each, obtained from the UCR Time Series Classification Archive [25]. Food spectrograph measurements of dry-cured hams for the classification of food types, with applications in food safety and quality assurance.
5) **BirdChicken:** 20 time series with 512 time steps each, obtained from the UCR Time Series Classification Archive [25]. The time series are descriptors of shapes (outlines) of images of chickens and birds mapped into 1-D series of distances to the center, designed to be used in classification problems.

For each data set, the first step was to extract a random sample S of pairs of time series (among all N time series available in each data set) to use for each experiment, i.e. $S = \{(s_i, s_j)\}, 0 \leq i, j < N, i \neq j$. The reason for this initial sampling is that we want to measure the accuracy and speed of the DTW computations, which is an operation that is applied to pairs of time series. We could have simply compared all pairs of time series of each data set, but by choosing a fixed set of random pairs from each data set, we ensured that the population of time series from each data set was represented in a balanced way while we maintained a feasible running time for the experiment. For this paper, we used $\|S\| = 50$.

A. Experiment 1: Performance

Figure 3 shows the results of the first evaluation: the performance of each DTW implementation (y axis), averaged on 50 runs, against the number of time steps appended to the time series (x axis). To simulate a streaming situation, we start with empty time series and iteratively append one time step at a time, computing the running time at each iteration.

As expected, our described technique (`SlideDTW`) behaves similarly in all five data sets: the running time increases very slowly as the series grow, since it directly supports the *append* operation in linear time $O(n+m)$. Neither of the other tested implementations (`FastDTW`, `PrunedDTW`, or `DTW`) support such an *append* operation, which means that the running times grow much more steeply, linearly for `FastDTW` and quadratically for `PrunedDTW` and `DTW`. All the techniques showed consistent behavior among all tested data sets.

B. Experiment 2: Accuracy

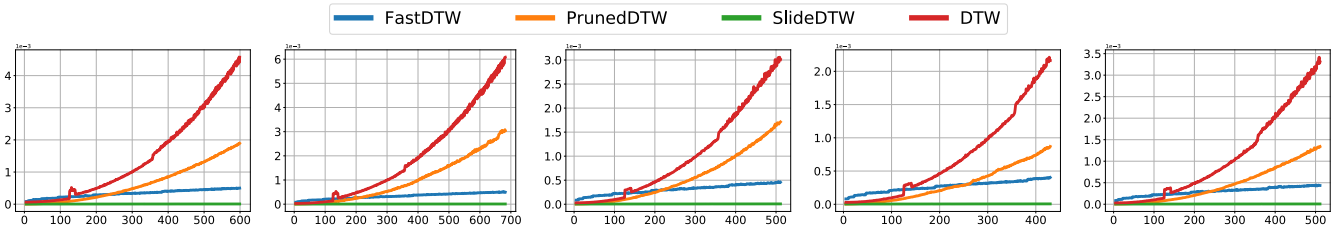
As discussed in Section III, `SlideDTW` is error-free as long as R^{Ω} and C^{Ω} are stored in their entirety; when some user-specified space limit is reached, older time steps must be discarded, which will lead to decreased accuracy as new time steps arrive. In this context, *accuracy* is considered to be the absolute difference between the value computed by `SlideDTW` (or `FastDTW`) and the value computed by `DTW` at each iteration (which is considered to be the optimal value).

In order to comprehend the magnitude of the accuracy problems the user has to deal with, we re-ran the experiments on each of the five data sets with four different imposed space limits: N (the full time series), $N/2$, $N/4$, and $N/8$. The plots in Figure 4 show the accuracy of `SlideDTW` with each constraint, plus the accuracy of `FastDTW` for comparison. As with the previous figure, the x axis indicates number of appends, while the y axis represents the error observed at each append, averaged from 50 runs. Note that the running time (performance) of the space-constrained experiments are not shown, because they follow the same results as Figure 3.

While it could be argued from Figure 3 that the differences in running times for `SlideDTW` and `FastDTW` are not significant, the accuracy results paint a different picture. As it can be seen from Figure 4, as new time steps are appended to the time series, in most cases the errors for the `FastDTW` implementation increase very fast, while the errors for `SlideDTW` are significantly lower. Different than the first experiment, however, the results are not consistent among all data sets. The best results are obtained from data sets *Random* and *Earthquakes*, where the errors are either non-existent or very close to zero. For time series with this profile, the choice between the two implementations appears to be clear. Slightly worse results were obtained from data sets *Twitter* and *Ham*, where it is possible to observe that, as the space constraint gets smaller, the errors begin to appear. Even in their worst cases, however, when the space is constrained to be equal to one eighth of the total length of the series (purple lines), they are still significantly lower than `FastDTW`. Finally, in the results observed with the data set *BirdChicken*, the `SlideDTW` technique is still superior in most cases, losing only in the smallest space constraint ($N/8$).

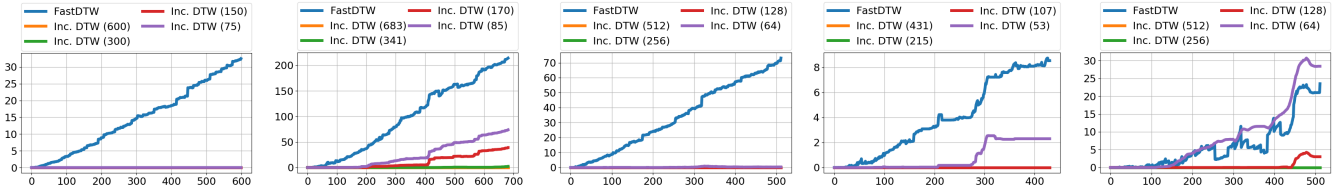
C. Experiment 3: Sliding Window

The third set of experiments were designed to test how the different DTW implementations behave when a sliding window is used, i.e., when only the last λ steps of each time series should be used to compute the DTW between them (as described in Section III-A). The setup for these experiments is slightly different than previous ones. For each run of the experiment, the window length λ was fixed; next, all time steps of each time series were appended, one at a time, sliding the window as described in Section III-A. After each append, the *accuracy* was recomputed by comparing the obtained value of `SlideDTW` (or `FastDTW`) at that step with the value of a newly computed `DTW` that considers only the last λ steps, i.e., a standard DTW computation as if the time series involved



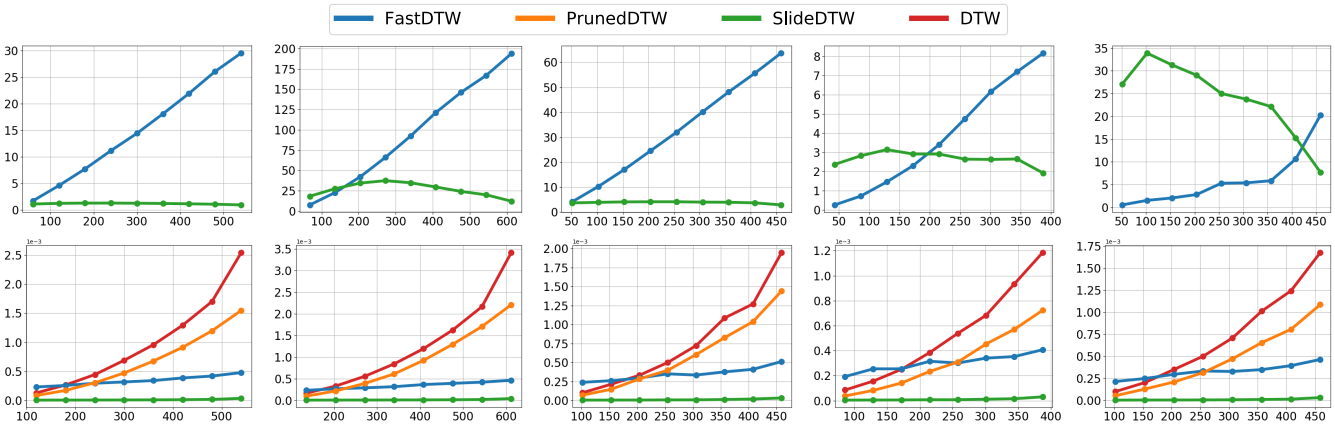
(a) Random (b) Twitter (c) Earthquakes (d) Ham (e) BirdChicken

Figure 3: Experiment 1 – Performance, in milliseconds (y axis) per series length (x axis). Average for 50 pairs.



(a) Random (b) Twitter (c) Earthquakes (d) Ham (e) BirdChicken

Figure 4: Experiment 2 – Accuracy, in difference from DTW (y axis) per series length (x axis). Average for 50 pairs.



(a) Random (b) Twitter (c) Earthquakes (d) Ham (e) BirdChicken

Figure 5: Experiment 3 – Sliding window. First row: *accuracy*, measured in difference from DTW (y axis) per window length (x axis). Second row: *performance*, measured in milliseconds (y axis) per window length (x axis). All results averaged for 50 different pairs of series.

were only λ-steps long. After all steps were appended, each of these partial accuracy scores were averaged, resulting then in the accuracy for the corresponding window length λ. As before, this was repeated for 50 different random pairs of time series for each data set, and the results were again averaged. For example, in Figure 5 (top row), in the 100 mark of the x axis, the y value represents the average accuracy obtained when a pair of time series is appended, one step at a time, with a window of fixed length $\lambda = 100$. The performance values (Figure 5, bottom row) are measured as the average time for computing the DTW for one window of each length; this is the most relevant for a streaming situation, since you want to recompute your DTWs and update your data as fast as possible as the data comes in. This is different than measuring the total running time for computing all windows, since that changes depending on the window length: smaller windows are fast to compute, but you need to compute many of them; larger windows are slow, but you need to compute few of them. We did not measure that as it is less important in a streaming scenario.

The results reflect the observations made in Section III: computing a sliding window by reusing and adapting the accumulated values from the `SlideDTW` is very fast, but it is an approximation and may not yield optimal results when the length of the sliding window is too small. In some data sets the `FastDTW` implementation works better for small window lengths (e.g., $\lambda \leq 150$ for *Twitter*, or $\lambda \leq 200$ for *Ham*), but, as the window length increases, the magnitude of the errors

from FastDTW also increases (following the results shown in Figure 4), while the errors for SlideDTW decrease and move towards zero as λ approaches N. For scenarios involving big data streams, where the amount of data for each time series is very large and the sliding windows are big, SlideDTW shows more promise. However, as with Experiment 2 (Section V-B), the results are again not consistent throughout all data sets. The results for the *BirdChicken* data set (Figure 5(e)), for example, are the worst among all the tested, with the SlideDTW implementation only gaining an edge against FastDTW around $\lambda \geq 425$. This shows that, depending on the profile of the time series, the accuracy of SlideDTW may vary, warranting the need for more research in the direction of identifying aspects of time series that may help the analyst preview the quality of the output and choose which technique to use.

Performance scores remain unchanged when compared to Experiment 1 (Figure 3), i.e., SlideDTW is still the fastest when compared to other implementations, even after the inclusion of the operations that adapt the time series at each append to approximate the window-based DTW.

VI. Conclusion

In this paper, we described a set of enhancements to the DTW algorithm, collectively called SlideDTW, that allow it to be used efficiently in a streaming scenario, with a linear complexity of $O(N)$ when an exact, error-free version of the DTW is needed, or a constant complexity when a user-defined constraint is added (in a similar way to a Sakoe-Chiba band). SlideDTW supports the use of a *sliding window* with a constant length as the desired range of analysis for the data. Our experiments with synthetic and natural data sets have shown that the performance improves dramatically over previous implementations, and the potential errors incurred by the approximations are, in general, much lower than another state-of-the-art approximated DTW technique.

As future work, we intend to investigate the different "profiles" of time series that were detected during the experiments, in order to understand why they happen and how to improve the performance for some of the worse results, and additionally investigate the effects of the described techniques in other activities that depend on pairwise similarities, such as clustering, classification, dimensionality reduction, or visualization.

References

[1] A. Bagnall, J. Lines, A. Bostrom, J. Large, and E. Keogh, "The great time series classification bake off: A review and experimental evaluation of recent algorithmic advances," *Data Mining and Knowledge Discovery*, vol. 31, no. 3, pp. 606–660, 2017.

[2] H. Ding, G. Trajcevski, P. Scheuermann, X. Wang, and E. Keogh, "Querying and mining of time series data: Experimental comparison of representations and distance measures," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1542–1552, Aug. 2008.

[3] E. Keogh and S. Kasetty, "On the need for time series data mining benchmarks: A survey and empirical demonstration," *Data Mining and Knowledge Discovery*, vol. 7, no. 4, pp. 349–371, 2003.

[4] S. Aghabozorgi, A. S. Shirkhorshidi, and T. Y. Wah, "Time-series clustering — A decade review," *Information Systems*, vol. 53, pp. 16–38, 2015.

[5] J. P. Cunningham and M. Y. Byron, "Dimensionality reduction for large-scale neural recordings," *Nature Neuroscience*, vol. 17, no. 11, p. 1500, 2014.

[6] E. S. Ristad and P. N. Yianilos, "Learning string-edit distance," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 5, pp. 522–532, 1998.

[7] E. J. Keogh and M. J. Pazzani, "Derivative dynamic time warping," in *Proceedings of the 2001 SIAM International Conference on Data Mining*. SIAM, 2001, pp. 1–11.

[8] I. Oregi, A. Pérez, J. Del Ser, and J. A. Lozano, "On-line dynamic time warping for streaming time series," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2017, pp. 591–605.

[9] H. Sakoe and S. Chiba, "Dynamic programming algorithm optimization for spoken word recognition," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 26, no. 1, pp. 43–49, 1978.

[10] M. Müller, "Dynamic time warping," *Information Retrieval for Music and Motion*, pp. 69–84, 2007.

[11] S. Salvador and P. Chan, "Toward accurate dynamic time warping in linear time and space," *Intelligent Data Analysis*, vol. 11, no. 5, 2007.

[12] Y. Zhu and D. Shasha, "StatStream: Statistical monitoring of thousands of data streams in real time," in *Proceedings of the 28th International Conference on Very Large Databases*, ser. VLDB '02. Elsevier, 2002, pp. 358–369.

[13] P. Capitani and P. Ciaccia, "Warping the time on data streams," *Data & Knowledge Engineering*, vol. 62, no. 3, pp. 438–458, 2007.

[14] B.-K. Yi and C. Faloutsos, "Fast time sequence indexing for arbitrary Lp norms," in *Proceedings of the 26th International Conference on Very Large Data Bases*, ser. VLDB '00. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, pp. 385–394.

[15] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ser. PODS '02. New York, NY, USA: ACM, 2002, pp. 1–16.

[16] G. Kang and S. Guo, "Variable sliding window DTW speech identification algorithm," in *Proceedings of the Ninth International Conference on Hybrid Intelligent Systems*, ser. HIS '09, vol. 1, Aug. 2009, pp. 304–307.

[17] M. Matuschek, T. Schlüter, and S. Conrad, "Measuring text similarity with dynamic time warping," in *Proceedings of the 2008 International Symposium on Database Engineering & Applications*, ser. IDEAS '08. New York, NY, USA: ACM, 2008, pp. 263–267.

[18] F. Itakura, "Minimum prediction residual principle applied to speech recognition," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 23, no. 1, pp. 67–72, 1975.

[19] C. A. Ratanamahatana and E. Keogh, "Everything you know about dynamic time warping is wrong," in *Proceedings of the Third ACM SIGKDD Workshop on Mining Temporal and Sequential Data*, ser. KDD/TDM '04. New York, NY, USA: ACM, 2004.

[20] D. F. Silva and G. E. A. P. A. Batista, "Speeding up all-pairwise dynamic time warping matrix calculation," in *Proceedings of the 2016 SIAM International Conference on Data Mining*, 2016.

[21] 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*, ser. KDD '00. New York, NY, USA: ACM, 2000, pp. 285–289.

[22] S. Chu, E. Keogh, D. Hart, and M. Pazzani, "Iterative deepening dynamic time warping for time series," in *Proceedings of the 2002 SIAM International Conference on Data Mining*. SIAM, 2002, pp. 195–212.

[23] E. Keogh and C. A. Ratanamahatana, "Exact indexing of dynamic time warping," *Knowledge and Information Systems*, vol. 7, no. 3, pp. 358–386, 2005.

[24] N. M. Khan, S. Lin, L. Guan, and B. Guo, "A visual evaluation framework for in-home physical rehabilitation," in *Proceedings of the 2014 IEEE International Symposium on Multimedia*, ser. ISM '14, Dec. 2014, pp. 237–240.

[25] Y. Chen, E. Keogh, B. Hu, N. Begum, A. Bagnall, A. Mueen, and G. Batista, "The UCR time series classification archive," Jul. 2015, www.cs.ucr.edu/~eamonn/time_series_data/.

[26] M. Laitinen, J. Lundberg, M. Levin, and R. Martins, "The nordic tweet stream: A dynamic real-time monitor corpus of big and rich language data," in *DHN*, 2018.