Dictionary-Based Compression for Long Time-Series Similarity

Willis Lang, Michael Morse, Jignesh M. Patel
Computer Science and Engineering, University of Michigan
`{wlang, mmorse, jignesh}@eecs.umich.edu`

Abstract—Long time-series datasets are common in many domains, especially scientific domains. Applications in these fields often require comparing trajectories using similarity measures. Existing methods perform well for short time-series but their evaluation cost degrades rapidly for longer time-series. In this work, we develop a new time-series similarity measure called the Dictionary Compression Score (DCS) for determining time-series similarity. We also show that this method allows us to accurately and quickly calculate similarity for both short and long time-series. We use the well known Kolmogorov Complexity in information theory and the Lempel-Ziv compression framework as a basis to calculate similarity scores. We show that off-the-shelf compressors do not fair well for computing time-series similarity. To address this problem, we developed a novel dictionary-based compression technique to compute time-series similarity. We also develop heuristics to automatically identify suitable parameters for our method, thus removing the task of parameter tuning found in other existing methods. We have extensively compared DCS with existing similarity methods for classification. Our experimental evaluation shows that for long time-series datasets, DCS is accurate, and it is also significantly faster than existing methods.

Index Terms—Spatial databases and GIS, Database Management

1 Introduction

Applications of time-series data range from electrocardiogram (ECG) measurements in medicine, to facial recognition in biometrics, to particle tracking in physics. Increasingly, long time-series datasets and the need to analyze them have motivated the database community to create efficient and accurate similarity measures [4], [10], [11], [18], [48], [49], [55], [57]. These time-series similarity measures are useful for querying a database, classification, and clustering. Many of these methods are based on the dynamic programming (DP) paradigm where optimality in the data alignment is desired. This optimality is variable depending on the properties to be examined such as longest common subsequence or edit distance. Since many state-of-the-art methods use the DP method, the cost of comparing two time-series is quadratic in the length of the time-series. This makes the evaluation of similarity between two *long* time-series very expensive. Furthermore, many pruning techniques have been devised so that time-series similarity queries do not need to compute the similarity measures between the query and every time-series in the database [6], [25], [32], [49], [57], [58]. These techniques employ indexing methods to prune the search space but still require many sequence comparisons, and the amount of pruning is data dependent as we show in our experimental results. Clustering algorithms such as the well-known Unweighted Pair-Group Method with Arithmetic Mean (UPGMA) [50], where a quadratic-space dissimilarity matrix must be fully calculated, would benefit from a fast and accurate similarity measure [21]. In such cases, if time-series are to be clustered, no pruning can be done as all distances must be calculated. The quadratic nature of existing methods would make this computation extremely lengthy for longer time-series. Thus, there is a need for a more efficient similarity calculation method for *long* time-series data.

Similarity measures based on compression have been suggested for use with time-series [29]. This previous work has used the Kolmogorov Complexity as a foundation for their similarity score. These measures use off-the-shelf compression methods as a basis for computing a similarity score [13]. Techniques of this type differ in approach from similarity measures such as DTW, edit distance measures (EDR/ERP), longest common subsequence (LCSS), or Euclidean because this method does not use pairwise comparisons of points in the time-series, but instead use the compressed sizes of entire time-series data to estimate sequence similarity. As a scoring technique, this method can be used for querying, classification, or clustering just like the methods mentioned above. These methods rely on both off-the-shelf compressors and discretization of the input; we show that the compression-like strategy of our method, which uses neither of these, is better than the competing alternative. Problems with discretization include improper discretization due to the hard bucket limits as well as additional parameters for bucket sizes (this is discussed further in Section 3.1). Moreover, as is evident in [13], [29], [38] and in our own examination, while these methods work well on text data, their performance on time-series can be improved. As we show in this paper, our DCS method is more accurate than these existing methods.

In this paper, we propose a novel compression-based scheme for computing time-series similarity. Our technique, called the Dictionary Compression Score (DCS), is a method that computes time-series similarity in the native, continuous domain space by looking at sequence segments. These segments are then used to construct a dictionary, which in turn is used to compute a similarity score. The dictionary construction scheme bears some similarity to the method employed by Lempel-Ziv compression, which builds words from a distinct alphabet instead of time-series sequences. If the dictionary used by the DCS technique is constructed on one time-series and used to compress another time-series, the method can be used as a similarity score similar to existing techniques such as DTW, LCSS, or Euclidean.

Our results show that for long time-series, DCS is often 30% faster than lower bounded constrained DTW and an order of magnitude faster than constrained LCSS using FTSE [43] while providing competitively accurate similarity scores. DCS is comparable to time-series Bitmaps [36] in speed; however, DCS is more consistent in achieving high accuracy than Bitmaps over varying types of data. While DCS is designed for long time-series, it is also interesting to ask how it performs on short time-series. DCS accuracy on short time-series is competitive against constrained DTW, and LCSS, with no clear statistical winner.

Specifically, this paper makes the following contributions:

1) A novel *Continuous-Domain Dictionary Compression* method is developed that builds continuous time-series elements in the native space to compute "words" suitable for storage in a dictionary created by a Lempel-Ziv compression-like scheme. The DCS method is developed which uses the Continuous-Domain LZ Compression method as a scoring method for time-series similarity.
2) We provide a simple hill-climbing and parameter refinement method for parameter selection, essentially making DCS a parameter-free method.
3) Our experimental section shows that for long time-series, DCS is significantly faster than existing methods while just as accurate. For short time-series, this compression style method is competitive in accuracy while other compression-based systems are not. These results show that we have produced a compression-based scheme that can compete with the current state-of-the-art where prior compression-based schemes could not.

The remainder of this paper is organized as follows: Section 2 presents related work. Section 3, describes our DCS method. Section 4 contains experimental results, and Section 5 contains our concluding remarks.

2 Related Work

As is commonly done in time-series similarity, we assume that time is discrete throughout this paper. A time-series T is defined as a sequence T = (p_1, t_1), (p_2, t_2), ... , (p_n, t_n), where each p_i is a data point in a d-dimensional space that is sampled at time t_i. Each t_i is strictly greater than each t_{i-1}, and we assume that the sampling rates of time-series are equivalent.

A few examples of time-series from a labeled dataset are shown in Figure 1. These time-series come from the popular Cylinder-Bell-Funnel synthetic dataset [28]. This figure shows three different time-series, namely (a) the cylinder type, (b) the bell type, and (c) the funnel type. Given a query sequence, the task is to classify the query as either a cylinder, bell, or funnel.

A normalization scheme for time-series is developed in [19]. Using this method, a time-series R is normalized by evaluating $\forall\ i\ \in\ m:\ r_i = (r_i - \mu)/\sigma$ for all of the m elements of R, where μ is the mean of the data and σ is the standard deviation. This method is widely used in [10], [28], [41]. In this paper, all time-series are normalized as described above. Further, we assume that time series to be compared are of the same length.

There are many techniques for measuring the similarity between two time-series. For example, the simple Euclidean distance [2] between two time-series sums the Euclidean distance between corresponding elements of each time-series. The Dynamic Time Warping measure (DTW) was first introduced in [4] and extensively developed further by [28] [46] [25] [30] to name a few. Unlike Euclidean, DTW does not require time-series to be of equal lengths and it allows for time shifting between time-series by repeating some sequence elements. There are several techniques based on the edit distance, including the Edit distance with Real Penalty (ERP) [10] and the Edit Distance on Real sequences (EDR) [11]. The Longest Common Subsequence (LCSS) technique is presented in [55] and most recently, the Sequence Weighted Alignment (Swale) method that combines elements of the LCSS and EDR methods is presented in [43]. A recent examination of these state-of-the art methods is presented by Ding et al. [15].

Dimensionality reduction techniques have been extensively studied, including the Discrete Fourier Transform [2], the Singular Value Decomposition [35], Discrete Wavelet Transform [8], [45], Piecewise Aggregate Approximation [27], [56], and the Adaptive Piecewise Constant Approximation [31]. Indexing of time-series is studied in [18], which proposes a lower bounding technique that guarantees no false dismissals. Indexing techniques for the DTW measure have been proposed, including [25], [32], [49], [57], [58]. Since this paper discusses a fundamentally different style of time-series analysis, we do not discuss these methods further but we will compare against some of them in our experimental evaluation.

Time-series similarity can be calculated using compression techniques [29]. These methods approximate the Kolmogorov Complexity by compressing time-series R concatenated with time-series S. Recent work [12], [13], [29], [38], [39], [44] has shown that the Kolmogorov complexity is a effective basis for developing accurate similarity measures. This work dealt with discrete data or discretizing continuous data and applying off-the-shelf compressors to approximate the Kolmogorov complexity. The intuition behind the scheme is that if R and S are similar, the compressed size of the concatenated file should be smaller than R and S compressed alone, since the compression method has been "warmed up" with R before S is compressed. Our methods use a dictionary similar to that used by common compressors and we develop a scoring method that utilizes the mechanics of a compression technique.

Another large body of work that is similar to the methods we discuss here is that of Vector Quantization (VQ) [20]. Both methods are based on lossy compression. Our methods similarly transform an input vector into smaller blocks or segments. VQ can be used for classification as described in [34]. However, while VQ maps fixed-length segments into a finite (and thus discrete) set of possible values, our method stores each *variable-length* segment extracted from the input

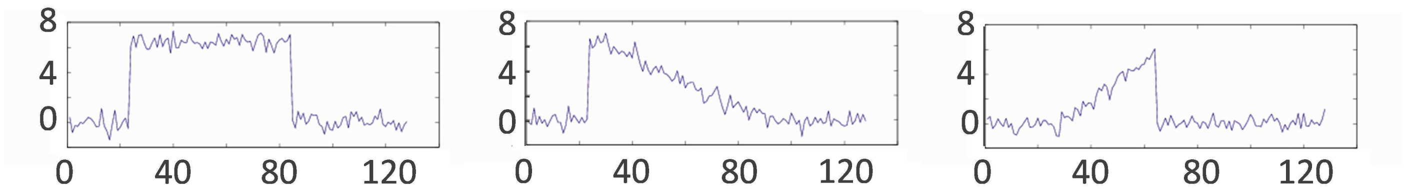


Fig. 1. Three examples from the Cylinder-Bell-Funnel Dataset.

vector and so we do not have a finite or discrete set of mapping targets. Further, our variable-length segments are determined by a Lempel-Ziv like algorithm that lengthens segments based on frequency. Details on this can be found in Section 3.2.

While we have discussed different methods of similarity measure, here we discuss various applications of these similarity measures. One use of these similarity measures is in data clustering. Given a set of time-series, a dissimilarity matrix can be calculated and clustering algorithms can be applied such as UPGMA [50] or Partitioning Around Medoids (PAM) [24].

In terms of database querying, the similarity measures can be used to search a time-series database against a single time-series query producing a ranked list of matching results. This use naturally also leads to the use of these similarity measures for classification using the Nearest-Neighbor method. For the purpose of classification, other methods such as Random Forest [7] and Support Vector Machines (SVM) [9] can be used. But, these methods do not provide an explicit similarity measure like DTW, EDR/ERP, LCSS, etc. and cannot be used for any of the other uses of these similarity measures as mentioned above.

2.1 Time-series Bitmaps

Time-series Bitmaps were introduced in [36]. This tool is used to visualize time-series in the form of thumbnail images. These bitmap images can also be used for time-series similarity. The similarity between time-series Bitmaps and our method lie in the calculation of short segments within each time-series and their frequency. However, the way these segments are created, counted, and analyzed are very different in both methods. For a given alphabet size and desired segment length, Bitmaps discretize the time-series, enumerate all possible words, and finally calculate the frequency of each within a time-series. We compare DCS with Bitmaps in our experimental evaluation.

2.2 Kolmogorov Complexity

Kolmogorov complexity is the measure describing the minimum amount of information required to produce a specific item of data. Li et al. [38] defines it as: $K(x)$ of a string x is the length of the shortest binary program to compute x on an appropriate universal computer - such as a universal Turing machine. Further, $K(x|y)$ is the length of the shortest program to generate x given y, an additional input. Finally, $K(x, y)$ is the length of the shortest program that generates x and y and a delimiter. Kolmogorov complexity is in general, incomputable [38], [42]. We rely on the idea that two similar strings of data will have very similar Kolmogorov complexities. Further, if x and y are similar, we should expect $K(x|y)$ to be small; if $x = y$, $K(x|y) = 0$. The main approach for approximation is to substitute $K(x|y)$ with $C(x|y)$ where function C is a compressor and we can measure the length of the compressed data. Since $K(x|y)$ is the length of the shortest program that generates x given y, the approximation $C(x|y)$ is the length of the code that can regenerate x given y. Interested readers can refer to Vereshchagin for details of Kolmogorov functions [53].

2.2.1 *Off-the-Shelf Compression and Similarity*

Since the relationship between the Kolmogorov complexity and compressibility of data are quite related [39], it is unsurprising to find a large body of work studying the approximation of $K(x)$ using current compression techniques [12], [13], [29], [38], [44]. These methods utilize off-the-shelf compressors to approximate Kolmogorov complexity and use it to determine similarity between data. In our study, we have reached the same conclusion that these prior methods work well for text and discrete data. While these methods are arguably not suited for short time-series analysis [29], for the purpose of completion, we tested how our method works on short time-series. Interestingly, we see competitive results for short time-series against classical dynamic programming methods. Our work does not rely on off-the-shelf products but instead augments the Lempel-Ziv algorithm to produce a more accurate similarity measure.

2.3 Lempel-Ziv

Lempel-Ziv (LZ) has been well studied since its development in 1977 [59]. The algorithmic complexity of LZ is linear [3]. In its raw form, it scans along a sequence of data and in one pass simultaneously generates a dictionary and compresses the data. This is an elegant and effective method of compression, as its compressed form can be disseminated and uncompressed without ever requiring knowledge of the contents of the dictionary. We use the LZ algorithm as shown in Algorithm 1.

2.3.1 *Lempel-Ziv Shortcomings*

As a compressor, the compression ratios achievable by LZ are generally weaker than other commonly used compressors [40]. Its compression is based on its elongation of items in the dictionary (we will refer to these as words). Since the algorithm only makes one pass of the data, the recognition of repetitive pieces of the data is limited. Also, the greedy nature of word matching is not guaranteed to be optimal. Further passes through the data would obviously provide the algorithm with more knowledge of the highly repetitive, lengthy regions that allow for best compression. Furthermore, the left to right

Algorithm 1 Lempel-Ziv Algorithm

```
1: Input: $S = s_0 ... s_n$
2: Initialize $dictionary$: set of singleton alphabet symbols
3: $w$ = NULL
4: while $|S| > 0$ do
5:     $c$=$s_0$
6:     if $wc \in dictionary$ then $w$ = $wc$
7:     else add $wc$ to $dictionary$; output code for $w$; $w$ = $c$
8:     end if
9:     $S = s_1 ... s_n$
10: end while
11: output code for $w$
```

mechanism only gives better compression to the latter parts of the input data as the dictionary becomes more populated. This means that the earlier portions of the data will be poorly compressed with smaller dictionary words. However, as it is linear in time complexity with respect to the input data, it is faster than many other compression techniques.

3 The DCS Method

In this section, we describe our DCS method. We begin with an overview of the *Continuous-Domain Dictionary Compression*. Then, we describe a method for constructing a dictionary for time-series datasets in the continuous domain.

3.1 Continuous versus Discrete

Before discussing the method itself, we motivate the benefits of a non-discretizing method for time-series compression. Figure 2 shows that discretization of continuous data may miss proximity relationships that a more flexible interval method may capture. Notice that the three data points in buckets E and F. E contains two relatively distant points while E and F contain two points that are close together. An interval-based method discretizes the continuous value domain into intervals that are all of a uniform length. The method determines a score for two particular values based on whether or not they map to the same interval. As seen in the figure, two points that fall on opposite sides of an artificially imposed boundary are unnecessarily penalized. While working in the continuous domain requires extra overhead in terms of data structures such as interval trees, it allows a more accurate view of the relationships between time points.

3.2 Algorithm Overview

In this section we discuss the details of our work, starting with a description of terms that we use. As the LZ algorithm is used as a compressor for text data, the input is a string. It inputs substrings, i.e. "words", into its dictionary while it scans through the characters. For time-series, the parallels are as such: the time-series can be thought of as the string, "segments" are analogous to "words", and time points are similar to characters.

The LZ algorithm constructs a dictionary by attaching a new character c from the string that it is compressing to a word w that is already in the dictionary and checking to see if that word is present. If it is, c is added to the word. This is the same

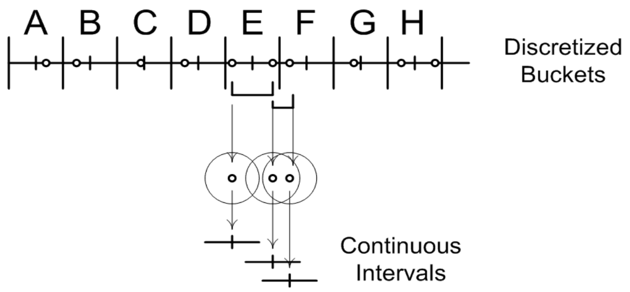


Fig. 2. Hard discretization quickly buckets data points into partitions. This example shows that 2 data points that are close together are partitioned into E and F while a more distant point is also bucketed into E. Our continuous compression method uses an ϵ parameter to create intervals which distinguish nearby points more flexibly. Here the middle interval intersects both the first and last interval. The first and last intervals are non-intersecting which maintains ϵ integrity. Searches for nearby time points are facilitated by interval trees.

Algorithm 2 Build and Compress Phases

```
1: Input: $X = x_0 ... x_n, Y = y_0 ... y_n$
2: Output: $score$
3: $Dict_X$ =BUILD DICTIONARY for $X$
4: COMPRESS $Y$ using the largest segments in $Dict_X$
5: OUTPUT compressed size of $Y$
```

intuition behind the way that the dictionary is constructed for our Continuous-Domain Dictionary Compression.

In a number of other time-series similarity scoring techniques including the popular LCSS [55], EDR [11], and LCSS-FTSE [43] measures, an ϵ factor is used to find matching elements. In these schemes, if two time-series elements $r_i \in R$ and $s_j \in S$ are within ϵ of each other, i.e. if abs($r_i - s_j$) $< \epsilon$, r_i and s_j are said to match. Otherwise, they are said to not match. We also use this ϵ criteria to determine matching time-series components. Continuous-Domain Dictionary Compression constructs a dictionary for a time-series R by attempting to attach a new time point r_i from time-series R to a set of previous elements $r_{i'}$,...,r_{i-1} which constitute a segment already in the dictionary. The time point r_i is matched with any elements in the dictionary that are within a distance of ϵ, i.e. any elements that fall between $r_i - \epsilon$ and $r_i + \epsilon$. Once the dictionary for R is constructed, a similarity score between time-series R and S can be obtained by matching the segments in the dictionary with segments of S. We retain the greedy method of matching segments in the dictionary based on the length of the segment to retain the speed benefits. This greedy approach of dictionary segment matching is simple and effective as shown by our experiments. The overall scheme is summarized by Algorithm 2.

Scoring measures such as Euclidean distance, DTW, and LCSS calculate the similarity between two time-series by comparing both sets of time points from the two series simultaneously. Our method decouples the analysis of the two series by looking at them sequentially. In this way, every build phase can also be thought of as a *training* phase.

Algorithm 3 Build phase

```
1: Input: $S = s_0...s_n$
2: Output: $Dict_X$
3: Initialize $dictionary$: empty dictionary hash
4: $prev = NULL$; $loc = 0$; $Y = S$
5: while $|Y| > 0$ do
6:    $segment$ = longest dictionary item that is prefix of Y
7:    $i = |segment|$; $loc = loc + i$
8:    if $prev \neq NULL$ and $|prev| > 0$ then
9:       $prev = APPEND(prev, y_0)$
10:      while $|prev| > 0$ do
11:         $ADD(dictionary, prev)$; $prev = prev_{0...(|prev|-1)}$
12:      end while
13:   end if
14:   $prev = y_0...y_{i-1}$; $Y = y_i...y_{|S|}$
15: end while
16: OUTPUT $dictionary$
```

Algorithm 4 Compression phase

```
1: Input: $Dict_X, S = s_0...s_n$
2: Output: $Compress_X(S)$
3: $Compress_X(S) = 0$; $loc = 0$; $Y = S$
4: while $|Y| > 0$ do
5:    $segment$ = longest dictionary item that is prefix of Y
6:    $i = |segment|$; $loc = loc + i$; $Y = y_i...y_{|S|}$
7:    $Compress_X(S) = Compress_X(S) + 1$
8: end while
9: OUTPUT $Compress_X(S)$
```

Using the LZ dictionary framework allows our method to self adjust to different levels of similarity. For example, if the data that is used to create a dictionary is highly repetitive with repeated structure, the dictionary will contain longer segments. This allows our method to examine the target data for longer repeated *phrases* or *motifs* and identify similarity at a high level. However, if the input to the build process is highly random, with few repeating phrases, the Compression stage will simply look for two length segments that indicate similarity at a lower alphabet level.

3.2.1 Building the Dictionary

We present our method for building a dictionary from a time-series in Algorithm 3. For reasons explained below in Section 3.2.2, we do not explicitly load singleton time-points into the dictionary. Thus, the length of the shortest segment in our dictionary is 2. In addition to adding the segment $prev$ to the dictionary, all prefixes of the segment are also added (Lines 13-16). The reason for this is discussed below and illustrated in Section 3.2.5. We now discuss the parameters that allow our method to compress continuous data.

The first key idea is the dependency on the ϵ value for defining similarity between elements of two time-series R and S. As mentioned in the previous section, since we are working in the continuous domain, there is no alphabet for the time-series. The data elements now lie in the real number domain, so the issue of matching segments in the dictionary to a prefix of a time-series must differ from previous approaches.

In Figure 3, we demonstrate how the ϵ factor allows for segment matching. Here, a segment in the dictionary of length 5 was created to span time points $(t+4)...(t+8)$. We set an epsilon factor of 0.5 which allows other time points to match those in this dictionary segment within a Euclidean distance of 0.5. We depict a static ϵ value in Figure 3, which means that at each time point we rigidly allow an ϵ shift. Through experimentation, we have found that an average ϵ value for a given segment works just as well, and in some cases even better. For example, given an ϵ of 0.05, a rigid ϵ constraint forces each time point in the segment to satisfy the bounds. However, the average ϵ constraint would simply require that the average deviation over all time points in the segment be within ϵ. The intuition behind this type of shifting is to be resistant to noise factors at sporadic time points that would have made this segment a reject.

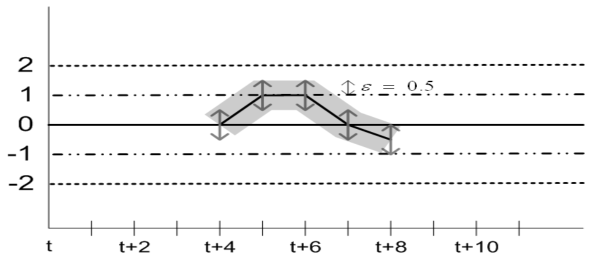


Fig. 3. The matching region for a segment in the dictionary where the $\epsilon = 0.5$. Depicted is the static epsilon technique where each data point is allowed a deviation of ϵ in Euclidean distance away from it at a given time point.

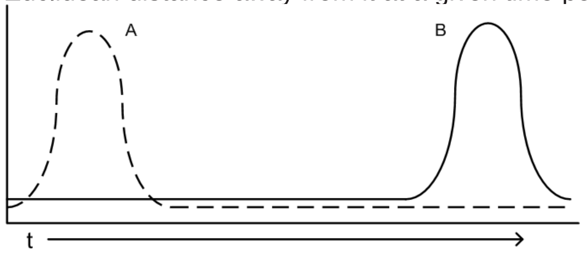


Fig. 4. The δ factor is used to control dictionary segments from being improperly used. Here, segments that are produced by the peak of time-series A could be used during the compression of the peak of target series B. Further, both time-series have flat intermediary regions that are within small ϵ. The compression sizes would be quite similar and erroneously give favorable similarity.

The second parameter, δ, determines the time warping sensitivity. Other methods employ a δ factor to restrict the time warping of a matching scheme, such as the Sakoe-Chiba band [48].

Figure 4 illustrates the significance of restriction. In this figure, the two curves are quite different since time-series A contains a peak early in the time-series and time-series B contains a peak at the end of the time-series. The similarity measure calculated by most existing techniques should reflect this difference. In our case, the dictionary segments built from any location such as the peak in time-series A could match any other location in B regardless of time warping. In this example, if the dictionary that is produced from the peak of time-series A is used to compress the peak of time-series B, a high compression ratio could be achieved. Thus, we use the δ factor to restrict dictionary segments to match a prefix. Our warp restriction is primarily used to increase accuracy.

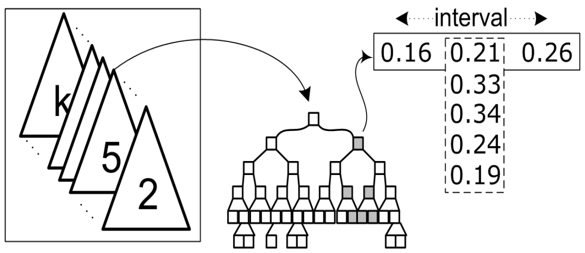


Fig. 5. A tree is built for each size of segment. If $\epsilon = 0.1$, then we can see the +/- 0.05 that is applied to the first element of the segment. This interval (specific segment) is then inserted into the tree. Since intervals in each tree are equal to ϵ, the traversal required is more limited than an interval tree with unrestricted interval sizes. An example traversal of the tree is shown by the shaded nodes.

The δ value to limit time warping is not used in the dictionary build phase. It is ignored to allow for longer prefixes to be matched and thus longer segments to be constructed in the dictionary. The dictionary build phase also adds all subsequent prefixes of a given segment into the dictionary as described in section 3.2.1, so that the segment can be matched with the δ restriction.

3.2.2 Implicit Singletons in the Dictionary

Note that unlike traditional dictionary building for LZ compression, we do not initialize the dictionary with the entire alphabet of singletons. The simple reason being that our alphabet is not discrete and finite. As described above, we search for increasingly longer segments to match the prefix. Since our shortest segment length is 2, we start looking in the dictionary for segments of length 2 to match the prefix of the sequence. If none is found, implicitly, a singleton is matched. This will be further illustrated in Section 3.2.5.

3.2.3 Dictionary Implementation

The implementation of the dictionary is crucial to the overall performance of DCS. In the worst case, we must probe the dictionary $O(n)$ times during the entire build/compression, and our dictionary contains $O(n)$ items. Without a good dictionary implementation, our method cost is $O(n^2)$. The best solution is to use an interval tree that allows $O(log(n))$ time for additions. While this solution does not guarantee a worst case solution better than $O(n^2)$, we show in Section 3.2.6 that this worst case is pessimistic in practice.

The dictionary is implemented using an interval tree. The details of an interval tree can be found in [14]. The interval tree is used in the following way: for each segment, we use the interval created by the epsilon bounds for the first value of the segment as the key for the tree. This is depicted in Figure 5. In the figure, we show that trees are constructed for each segment size, which keeps all the segments of a particular size x together in an interval tree. In the figure, we show an example where a segment of length 5 has an interval of 0.16 to 0.26. Thus, if we have a query interval of 0.11 to 0.21, our tree traversal would find this interval and examine the segment to which it is bound to see if the rest of the values satisfy the ϵ criteria. As an example, later nodes in the traversal are shaded to show the nodes/segments that would have to be examined.

3.2.4 Scoring

The largest prefix of the target data sequence is matched with entries in the dictionary until all elements of the target data sequence have been examined. The details are shown in Algorithm 4. The greedy nature of the algorithm for segment matching makes it fast, but also means that it may not produce the best possible alignment (just as some compressors outperform LZ). However, like LZ, we show that our method produces good alignments in practice.

In our method, each dictionary match is valued equally toward the final similarity score. For example, if the target data sequence is fully exhausted by the prefix matching and 100 segments in the dictionary are used (including singleton segments as described in the building phase), then the return measure from the compression phase is 100. The count is incremented by one at each iteration regardless of prefix length matched. The result is that for a sequence of length n, if all segments used during the compression were of length 2, the return value would be 0.5n. Our score mimics the length of compressed data given by an LZ compressor.

While Cilibrasi and Vitanyi [13] try to approximate the Kolmogorov Complexity ($K(x|y)$) of a string x given a string y using a compressor to find the compressed size of string y concatenated with string x, $NCD(x, y) = (C(xy)-C(x))/C(y)$, our method behaves differently in a subtle manner. Since we do not add any new elements to the dictionary after examining the first time-series y when scoring x, we do not find as near an approximation to the Kolmogorov complexity of $x|y$ (the minimum amount of information needed to communicate x given y). We do however, find a more accurate measure of how well the elements of y can match the elements of x. This is because the dictionary only contains elements from y. Since this is our ultimate goal, it is a meaningful distinction.

Our DCS(x,y), like DTW, is not a distance metric, it is a similarity score. The smaller the DCS(x,y), the more similar x and y are to each other than any other pair within the set of data. Notice that if $x = y$, we do not achieve $K(x|y) = 0$ which is required for a score to be classified as a metric.

3.2.5 Example

Figure 6 is an example of building the dictionary on a source data series as well as compressing two target data series. The first time-series is used to build a dictionary by calling Algorithm 3. Recall that singleton segments are not explicitly loaded into the dictionary. For all segments added to the dictionary, all prefixes of the segment are also included. This is shown with the segment (0.30, 0.20, 0.90) where its length 2 prefix is added. The need for this step is illustrated by target 1 where it matches a segment of length 3. The only way this could have occurred is if its prefix of length 2 was added because none of the other segments of length 2 were valid. Time of entry into the dictionary is also included. For example, the segment (0.90, 1.00):5 was created when the algorithm reached time point 5. When we compress using Algorithm 4, we also obey the ϵ and δ constraints. The δ constraint is

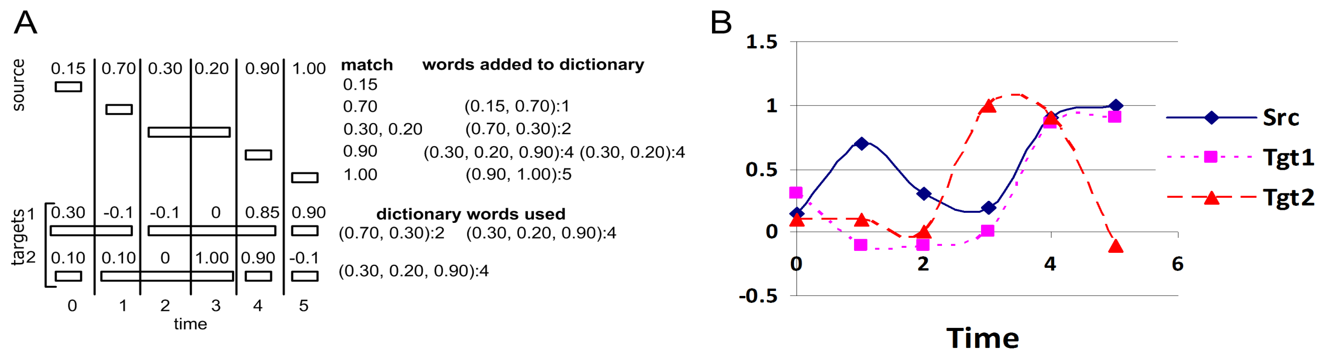


Fig. 6. Building and compressing time-series of length 6 with $\epsilon = 0.5, \delta = 2$. Time-series **Src, Tgt1, Tgt2** in panel B correspond to the time-series **source, target 1, target 2** in panel A respectively. Panel A shows the results of building the **Src** time-series from panel B. In panel A, the dictionary items added are shown on the right of the 'match' items. Dictionary items also store the time point at which they were created – $(< segment >) :< time\ of\ creation >$. In a Lempel-Ziv manner, time-series segments in the dictionary are built from **Src** and Time-series **Tgt1, Tgt2** are compressed using this dictionary. Segment $(0.30, -0.1) : 2$ matches $(0.70, 0.30) : 0$ since the it is within $\epsilon = 0.5$ and $\delta = 2$. The number of dictionary items used to compress the time-series is the DCS score. **Target 1** has a better score of 3 than **target 2** with a score of 4 and so **Src** is more similar to **Tgt1** in panel B.

utilized in the last time-series, we could not use the segment (0.30, 0.20):4 to match the prefix (0.10, 0.10) in target 2 because the segment was created at time point 4 while we are matching at time point 1. The temporal difference is greater than the δ of 2. This forces us to use an implicit singleton in the dictionary as the first match. Had we used the 2 length segment, we would end up using 3 segments to compress target 2 rather than the 4 shown. By visual inspection, we can see that the second time-series (target 1) is more similar to the first time-series (source). In comparison, the DCS score is 3 for target 1 and 4 for target 2. This would translate to $C(target1|source) = 3$ and $C(target2|source) = 4$. Since $C(target1|source) < C(target2|source)$, we would conclude that target 1 is more similar to the source time-series that target 2.

To conclude the discussion of the DCS algorithm, we summarize our method. *Time-series are built into interval tree dictionaries in their native domain in one pass (LZ-style). Compression of query time-series are done in a single pass to give a "compressed" size of the query time-series. This compressed size (similarity score) can be compared against the size given when using other time-series dictionaries.*

3.2.6 Algorithmic Cost Analysis

As mentioned earlier, LZ has linear complexity. In Algorithm 3 and 4, the worst case complexity is $O(n \times DictAccess)$. The algorithm is linear with respect to the dictionary access complexity, as we must examine each time point of the time-series a constant number of times. Each of these examinations is a probe of the dictionary: both a search and an addition. The reason there is an addition is because in our technique, all prefixes of a segment are added into the dictionary. Therefore, the constant number is 2. We extend the LZ algorithm to handle continuous data to retain its linear nature. However, since we are doing interval searches, constant dictionary time access is impossible.

Since we are using the interval tree, each addition into the dictionary costs $O(log(n))$ time. Each probe into the dictionary costs $O(log(n) + m)$ where m are the matches returned. Nodes can be examined in constant time if the maximum segment length allowed in the dictionary is set to some constant. This is generally done in LZ compression since the dictionary size is generally bounded. Further, we do not need to traverse the tree to recover *all* matches, once we find that there is a match, we stop the traversal and try to find a segment of longer length. Since m is bounded by $O(n)$, the algorithm has a worst case complexity of $O(n^2)$.

Further analysis and empirical results reveal that the $O(n^2)$ bound is very pessimistic in practice, as described below. For simplicity, if we assume that two time-series S and T are both of length n and the data is one dimensional, we can analyze the bound for m. Our two time-series are normalized around 0 and scaled by the standard deviation individually.

Now assuming that the data points are normally distributed and follow a conditional density function of random variable X as described by $\Phi(z)$ in Equation 1. The probability that one of our data values, drawn from the distribution of Z, falls in an interval $(a, b]$ is given by $P[a < Z \leq b]$ in Equation 2. Given $s_i \in S$ and $t_j \in T$, then the time point t_j falls within the interval given from $(s_i - \epsilon, s_i + \epsilon]$ with the probability $P[N(s_i) - \epsilon < N(t_j) \leq N(s_i) + \epsilon]$ in Equation 3. Now we can calculate the expected number of segments that are returned from the interval tree; $E[m|s_i]$ in Equation 4. The total number from all the queries on the interval tree is given by $E[m]$ in Equation 5.

$$\Phi(z) = \frac{1}{\sqrt[2]{2\pi}} \int_{-\infty}^{z} e^{-u^2/2}\, du \qquad (1)$$

$$P[a < Z \leq b] = \Phi(b) - \Phi(a) \qquad (2)$$

$$\begin{aligned} P[N(s_i) - \epsilon < \quad & N(t_j) \quad \leq N(s_i) + \epsilon] \\ = \quad \Phi(N(s_i) + \epsilon) - & \quad \Phi(N(s_i) - \epsilon) \end{aligned} \qquad (3)$$

To find the average number of intervals examined during a compression, we randomly generate 800, 2000 length sequences of values between 0 and 1. We then normalized each sequence by the mean and standard deviation. Using an $\epsilon = 0.5$ as in [54], we find that the average value for m is $0.06n$ when compressing a time-series with a dictionary of trees. That is, for each value in time-series T, we examine on average 6 percent of the values in time-series S that built the dictionary. As a result, the average cost of $0.06n^2$. This approximately translates to about a 15X improvement over N^2 time algorithms which is backed up by our long trajectory empirical results.

$$E[m|s_i] = n(\Phi(N(s_i)+\epsilon) - \Phi(N(s_i)-\epsilon)) \quad (4)$$

$$E[m] = n^2 \sum_{i=1}^{n} \frac{1}{n}(\Phi(N(s_i)+\epsilon) - \Phi(N(s_i)-\epsilon)) \quad (5)$$

Note that our analysis here is similar to that of [43] as both methods use an ϵ parameter in similar ways. However, an astute reader will have noticed that our average m is well below the figure reported in [43] because of the fact that we discontinue the dictionary probe as soon as a matching segment is found. As described earlier, the space complexity of our algorithm is linear. This still holds for continuous data as now in addition to the pointers for the segment, we store the epsilon interval for each node which is constant space.

3.3 Usage in Classification and Clustering

As with other similarity measures, our method can also be used for classification. For classifying continuous time-series data, we simply use the DCS and the *Continuous-Domain Dictionary Compression* method. For classification, given a set of training time-series with labels, DCS dictionaries can be built for each and its source label is attached to the dictionary. Given an incoming, unlabeled time-series, each of the database DCS dictionaries are used to compress the unlabeled series and a score is generated. The dictionary that gives the best score confers its label to the unlabeled time-series. This method is the nearest neighbor classification technique [29], [33], [52], where an unlabeled series is assigned its class label of its nearest neighbor in the training set. It has been argued that nearest neighbor classification is rather robust in its ability to resist the effect of noise. Also, given that the purpose of this classification is to highlight the effectiveness of the similarity measure, nearest neighbor is the most suitable because of its simplicity.

This idea can be similarly used for clustering methods such as UPGMA. Given time-series that we may wish to cluster, an N^2 dissimilarity matrix must be given as input. If a new trajectory is introduced to the database, the existing database dictionaries can be reused to compress the new time-series to create the pairwise scores. This feature of our method is unique from existing methods and provides time performance improvements for similarity score calculation.

3.4 Parameter Identification

Like many of the parameterized DTW methods such as the Sakoe-Chiba Band [48] or LCSS-FTSE [43], DCS requires

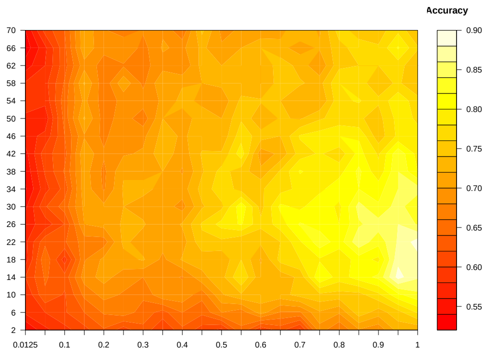


Fig. 7. ϵ/δ parameter grid and the leave-one-out DCS accuracy of the training set. δ parameter on the vertical axis, ϵ parameter on the horizontal axis.

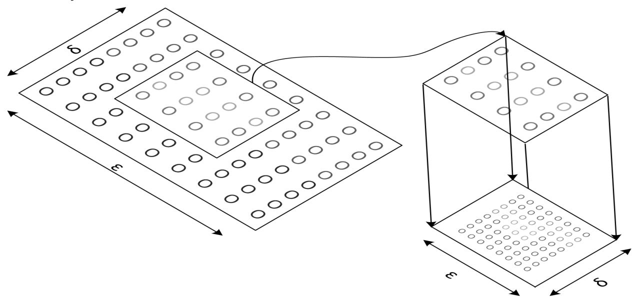


Fig. 8. ϵ/δ parameter grid with granularity refinement.

parameters to be set to produce optimal results. We train the parameters using some training set, to determine the optimal ϵ and δ (time warping) parameters. We note this framework we describe below can be applied to other similarity measures, such as Sakoe-Chiba Band or LCSS-FTSE. This is done simply by selecting each training time-series and calculating the DCS against all the other training time-series in the leave-one-out classification evaluation manner. Leave-one-out is often used for cross-validation and classification error estimation, but here we use it to discover suitable parameters [17]. Basically, at each "fold", for a set of size n, we choose $n-1$ time-series to train with and try to classify the remaining time-series. We repeat this procedure n times, allowing each of the time-series to be left out once and determine how many times (out of n) we correctly classified. Now for each full leave-one-out process, we choose a different parameter set. In our studies, we start with an $\epsilon = 0.05$ and increment 0.05 until $\epsilon = 1$. At an $\epsilon = 1$, we have hit the point where we allow time points to be one standard deviation apart. Since having an $\epsilon = 0$ would not allow any matching, we add $\epsilon = 0.0125$ to represent the 'exact' time point match. For δ, we did a search in $0.001m \leq \delta \leq 0.035m$ where m is the length of a time-series in increments of $0.002m$.

After we have calculated the leave-one-out accuracy in the training set for all ϵ/δ pairs, what we get a grid with ϵ as one dimension, and δ as the other dimension. For example, Figure 7 shows an ϵ/δ grid for the Synthetic Lightning EMP dataset [23]. This dataset was one of the long time-series datasets we ran experiments on. Details of this dataset can be found in Section 4.1. While we use the Synthetic Lightning EMP dataset here, these methods and results also hold with all the time-series we use in Section 4. The horizontal axis

Algorithm 5 Parameter Identification Heuristic for Leave-one-out classification (LOOC) on training data.

```
Input: a grid of LOOC accuracies, $\epsilon$ along x-axis, $\delta$ along y-axis
Output: $epselect$ is x coordinate on grid and $delselect$ is y coordinate on grid
if best accuracy is unique in grid then
    $epidx$ = x coordinate for best accuracy
    $delidx$ = y coordinate for best accuracy
else
    $epidx$ = x where sum of top-k accuracies in column x is max
    $delidx$ = y where ($epidx$,y) is max in column $epidx$
end if
$epselect$ = $epidx$; $dlo$ = $delidx$; $dhi$ = $delidx$
$step$ = $FALSE$; $peak$ = grid($epselect$, $delidx$)
while $TRUE$ do
    if grid($epselect$,$dlo$) $\neq peak$ AND $step$ then BREAK end if
    if grid($epselect$, $dlo$) $\neq peak$ then
        $step$ = $TRUE$; $peak$ = grid($epselect$, $dlo$)
    end if
    $dlo$ = $dlo$ - 1
end while
$step$ = $FALSE$; $peak$ = grid($epselect$, $delidx$)
while $TRUE$ do
    if grid($epselect$,$dhi$) $\neq peak$ AND $step$ then BREAK end if
    if grid($epselect$, $dhi$) $\neq peak$ then
        $step$ = $TRUE$; $peak$ = grid($epselect$, $dhi$)
    end if
    $dhi$ = $dhi$ + 1
end while
$delselect$ = ($dlo + dhi$)/2
```

plots δ and the vertical axis plots ϵ. The figure shows that there is a clear gradient in the leave-one-out accuracy in the training data. Further study of the grid data indicated that the ϵ parameter was more influential in the end accuracy of the calculated similarity measure than the δ parameter. Thus, we developed the technique found in Algorithm 5. The algorithm is based on hill climbing. First, we search for a unique maximum accuracy within this grid. If this is found, than we pick the ϵ value at which this grid point is found. If there are multiple maximum accuracies, we look at each ϵ column in the grid and calculate the sum of the top-k values within the column. In this tie situation, the ϵ column with the best top-k sum is the ϵ value we choose. We use the top 75th percentile in each column to retain within a standard deviation assuming N(0,1). Next, given this ϵ value, we look at the maximum accuracy in the column and find the most stable position within this peak. Thus we shift the δ value such that we move into the middle of plateau. In this way, we either choose the max peak, or we choose the best ϵ ridge and then slide along it to find the most stable maximum position. In Figure 7, there is one lone peak, at $\epsilon = 1.0$ and $\delta = 22$. After the δ stabilization, this shifts slightly to $\delta = 20$.

A further heuristic is to develop the ϵ/δ grid on a coarse grain scale and iteratively examine finer granularity of the grid. The manner in which this is done factors into the amount of time this heuristic takes. For instance, one could take a coarse grain grid in both ϵ and δ dimensions of the grid. If this is done such that only one deeper grain is examined, this would be a 75% savings in grid computation in the original iteration if the granularity doubles in each dimension. Figure 8 shows a schematic illustrative example. This example shows granularity refinement after Algorithm 5 has found an ϵ and δ pair. Looking in a refined fixed region around the coarse grain parameter pair allows us to look for maxima within the region. In this manner, it is possible to increase the efficiency of the parameter exploration in the training data.

4 Experimental Results

In this section, we evaluate the performance and accuracy of the DCS method. All experiments are run on a machine with an Intel 2.4Ghz Core 2 Duo processor with 2GB of DDR2 667 Mhz memory and SATA hard disks. The operating system was Fedora Core 6 and methods are implemented in C++ (except those where MATLAB code for comparison methods was provided). For our results, statistical significance is calculated with the Wilcoxon Matched Pairs Signed Ranks Test and a p-value cutoff of 0.01. We use the Wilcoxon test because it does not make the same normal distribution assumptions as the Student's t-test. Since we cannot make any assumption about the distribution of the accuracy results, the Wilcoxon test is more robust. Datasets are normalized as described in Section 2. All response times shown are the average over triple replicate runs.

A standard way to test similarity measures is to use it to classify data, where the results of the classification are known. This method has been used extensively in previous works [1], [11], [29], [43], [55], and we adopt this method here. We employ Nearest Neighbor and Leave-one-out style classification in our study. The effectiveness of the similarity measure is then measured using a single *accuracy* measure, which is computed as $|correct|/|testset|$ [1]. Improvement percentages are reported as $|(DCS - Method)|/Method$.

For our comparisons, we ran datasets over DTW-Sakoe-Chiba 3% warp restriction (DTW-SC3), LCSS, and the simple Euclidean (Eucl) measure. Lower bounding as described in [25] was applied to DTW-SC3 (DTW-SC3-LB for 3%). In addition to DTW-SC3-LB, we also wanted to see how a larger band would fare in accuracy and speed performance so a 10% band with lower bounding was also tested (DTW-SC10-LB). The DTW-SC is described as a special case of DTW-rk [1]. While the 3% warping band is shown to be a general sweet spot in band size [47], we also used our parameter exploration scheme to search for a band in the 1% to 10% range. We found in most cases, the heuristic band size resulted in the same accuracy as the 3% warping band. Also, the Sakoe-Chiba band measure shows no statistically significant difference in accuracy over the DTW-rk results posted in [1]. While we did not implement an index for DTW, the run time saved by implementing an index can be equated to the run time to lower bound DTW which is the Euclidean run time. Since the Euclidean run time is orders of magnitude faster than DTW in our experiments, this lower bounding time is negligible. For LCSS, our results are produced using LCSS-FTSE [43] to speedup the computation of LCSS. LCSS-FTSE was run with ϵ parameter set to 0.25 and run with the same 10% warp restricting band as that of DTW-SC10-LB.

We also compare against time-series Bitmaps (BM) as described in [36]. However, as discussed above, this method uses

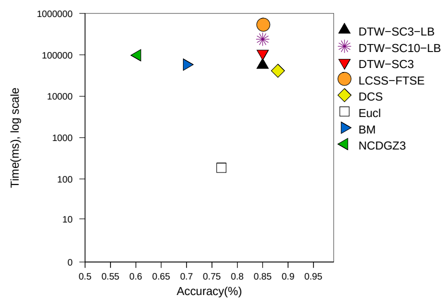


Fig. 9. Results of F2 classification by nearest neighbor, data from [23].

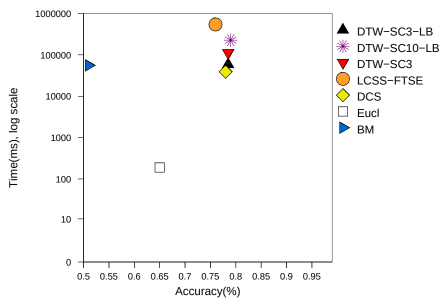


Fig. 10. Results of F6 classification by nearest neighbor, data from [23].

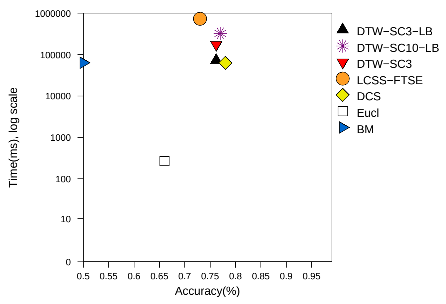


Fig. 11. Results of F7 classification by nearest neighbor, data from [23].

rigid discretization while we use dynamic intervals to compare time points. Further, Bitmaps rigidly enumerate and examine all possible "word" segments while DCS lengthens segments as our LZ-style algorithm examines prevalent "words". Other differences include the parameters required for DCS compared to Bitmaps. The Bitmap parameters include two parameters for the SAX discretization [41], one parameter for the alphabet size (up to 4), and another parameter for the size of the Bitmap. We use an alphabet size of 3 and a 64 cell Bitmap, as recommended by the authors. The 'N' and 'n' parameters are detailed on a per experiment basis. MATLAB code for SAX was run to discretize the data for this method.

We also compare against NCD. While [13], [29] show that using off-the-shelf compressors to approximate the Kolmogorov Complexity may work for discrete data such as text and discretized time series, we wanted to evaluate how well they do on time-series directly versus our method. We utilized the NCD measure as described by [13] and the gzip compressor. The details of NCD are described in section 2.2, 2.2.1, 3.2.4. Conversion of the time-series data to a discrete representation can be done via the SAX representation [41]. This representation carves up the assumed normal distribution of data points such that discretization buckets have uniform density. Given that this type of discretization requires a parameter to denote the size of the resulting symbol alphabet, we ran the same procedure over 3 values: 3, 10, 20. In other words we discretized the time-series to 3, 10, and 20 symbol alphabets and then ran gzip to compress these time-series representations. A key problem with this technique is that for the compression approximation to be as close as possible to the Kolmogorov Complexity, the compressor must achieve the highest possible levels of compression on the data [29]. Recently, an interesting finding has been that: *"It is interesting to note that PPM generally performs better than LZ-type coding in terms of the compress ratio. However, comparing the results shown in the first and second rows (figure), LZ78-based approach performs better in terms of classification accuracy... [40]"*. Of course, this introduces another disadvantage to the method because all possible compressors must be tested and even choosing the best compressor does not guarantee the best accuracy. We applied gzip because it is one of the most widely used and effective compressors. Here NCD using gzip and x characters is denoted NCDGZ(x).

Time results for NCD and Bitmap are end-to-end and included the time to discretize the unlabeled TEST data (we

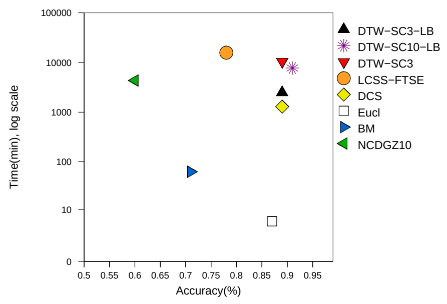


Fig. 12. Results of synthetic lightning (RS) classification by nearest neighbor using data from [23].

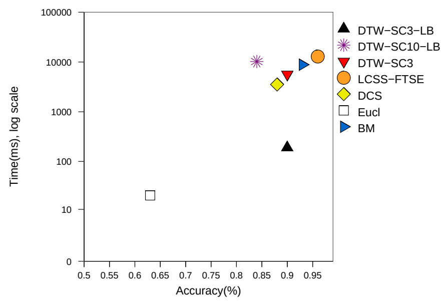


Fig. 13. Results of Surveillance classification by nearest neighbor using data from [36].

exclude TRAINING data processing time). Time results for our DCS measure are calculated for the time taken to compress a time-series with a dictionary since for classification the dictionaries can be prebuilt. Since both building a dictionary and compressing the time series both pass through the time-series once, dictionary build time is approximately the same as compression time. Parameter exploration time is also not included as it can also be precomputed. Parameter exploration time is approximately two times the dictionary build time on all the training time-series (since this is basically leave-one-out analysis on the training data and building and compression are approximately the same speed) multiplied by the parameter space size.

4.1 Long Time-Series

In this section we compare different time-series similarity measures on long time-series. Here we deem time-series longer than 1000 time points to be "long". The goal here is to evaluate the effectiveness of DCS against existing methods for long time-series. In these experiments, we use a symmetric DCS calculation, which means that for a set of time-series, the

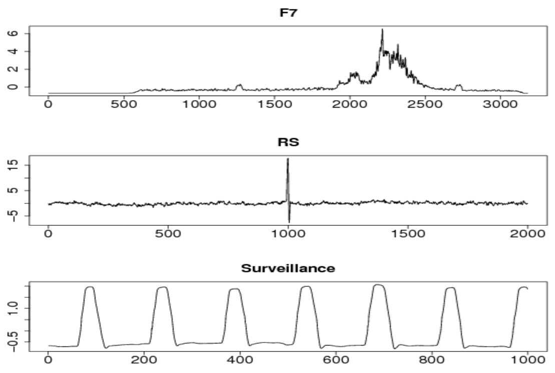


Fig. 14. Examples of the long time-series data that was analyzed. Class 6 from F7, class 1 from RS, class 2 from Surveillance.

similarity matrix calculated is symmetric. For example, in a one-sided DCS calculation, the similarity between query time-series Q and database time-series T is $DCS_T(Q)$ where is represents the DCS when Q is compressed with the dictionary built from T. A symmetric DCS score between two time-series is $DCS_T(Q) + DCS_Q(T)$. This method resembles the reciprocal best-hit method used for biological sequence comparisons [22].

4.1.1 Datasets

We use a number of long time-series datasets. A few of the datasets are provided from [23]. These are all longer than 2000 time points which is 3 times longer than the longest time-series in [1](examined in Section 4.2). Another long time-series dataset came from [36].

The first dataset also comes from [23]. This data is of lightning data collected from the Fast On-orbit Recording of Transient Events (FORTE) satellite. There are seven types of lightning, each with distinguishing characteristics. The interested reader can learn more about this data from [16]. The three datasets derived have 2, 6, and 7 classes; these datasets are called F2, F6, and F7, respectively. The first two, F2 and F6 contain the exact same time-series but with extra partitioning. F7 contains all of those and also an additional class of lightning measurement. They are not partitioned into training and test sets. Each time-series is 3181 time points long, F2 and F6 have 121 time-series each, and F7 has 143 time-series. The next dataset from [23] is a 2000 time point long, two class, synthetic time-series of lightning's electromagnetic pulse (EMP) measurements. We denote this set RS. This dataset has clearly partitioned training and test sets with 18000 test time-series and 2000 training time-series.

We used one dataset from [36], the Surveillance dataset, as the ECG dataset from that source is overly repetitive. ECG waveforms are usually analyzed over a small number of periods [5], [51]. We analyze a shorter ECG dataset in Section 4.2. The Surveillance dataset is 1000 time points long with four classes of 20 time-series. This dataset has no training/test partitions.

Since no training/test sets exist for all the long time-series except RS, we use the well known leave-one-out classification evaluation method to compare all methods [17]. As described above, leave-one-out simply takes a dataset of size n and takes each individual time-series out and tries to classify it against the remaining $n-1$ time-series. While the leave-one-out method is robust, it involves a very large number of runs. In our case, it required us to relearn the ϵ/δ parameters at each fold of the classification (on the other $n-1$ time-series). This relearning itself is a leave-one-out method as we described in Section 3.4. For the long time-series, we relearned at every fold by applying our parameter exploration method over a coarse grained parameter space: $0.1 \leq \epsilon \leq 1.0$ with 0.1 increments and $\delta = 0.011m, 0.021m, 0.031m$ for length m time-series. If leave-one-out is employed, classification time reported is the time to classify all n time-series of each fold. If a clear TRAIN/TEST partition is available, classification time is the time to classify the entire TEST partition.

The Bitmap method utilizes subsequences of time-series for measuring similarity. Since the authors do not provide a specific method for parameter discovery, we used the suggestions that are found in [26]. The SAX method that they utilize has two parameters 'N' and 'n'. Parameter 'N' is suggested to be two times the length of an interesting section of the time-series. Parameter 'n' defines the aggregation of time-points. For the F-series data, we used N=3181 and n=1 since this data was not periodic. Similarly for RS, we used N=2000 and n=1. For Surveillance, we used parameters (N=1000 and n=1). Correspondence with the authors revealed that Bitmaps are not very sensitive to changes in the parameter settings. This was experimentally found to be true as changes in the accuracy would change within approximately 5%.

For the NCD method, our implementations also utilized SAX for discretization. Our parameters are N=length and n=1 for all long time-series.

4.1.2 Results

The first set of results we discuss is the F-series datasets. These results are found in Figures 9, 10, and 11. Note that the NCD results for F6 and F7 are missing because there are no results from the NCD algorithm that gave us greater than 50% accuracy. As shown in Figures 9 to 11, DCS leads in two of the three time-series datasets. DCS is also relatively fast; only significantly slower than Euclidean. We also note that the speed advantage for DTW-SC3-LB over DTW-SC10-LB is around $3-4X$ and DCS is $27\%, 33\%, 13\%$ faster than DTW-SC3-LB in F2, F6, and F7 respectively. This means that DCS provides classification accuracy and speed performance just as high as DTW-SC3-LB even though our DCS results are not employing any pruning methods. We also notice that DTW-SC10-LB has a slightly higher accuracy than DTW-SC3-LB on F7. The heuristic band size found by our hill climbing method gives classification accuracies that match the 3% band.

On the much larger RS dataset (Figure 12), with its 18000 test time-series, DCS again outperforms DTW-SC3-LB in speed but this time by an even larger 48%. Both DTW-SC3-LB and DCS have identical accuracies of 89% on the RS dataset. The 10% band turned out to be the band selected by the hill climbing method and gave a higher accuracy of 91% but was slower than DTW-SC3-LB and DCS (by around 3X and 7X respectively). In this dataset, Euclidean provides

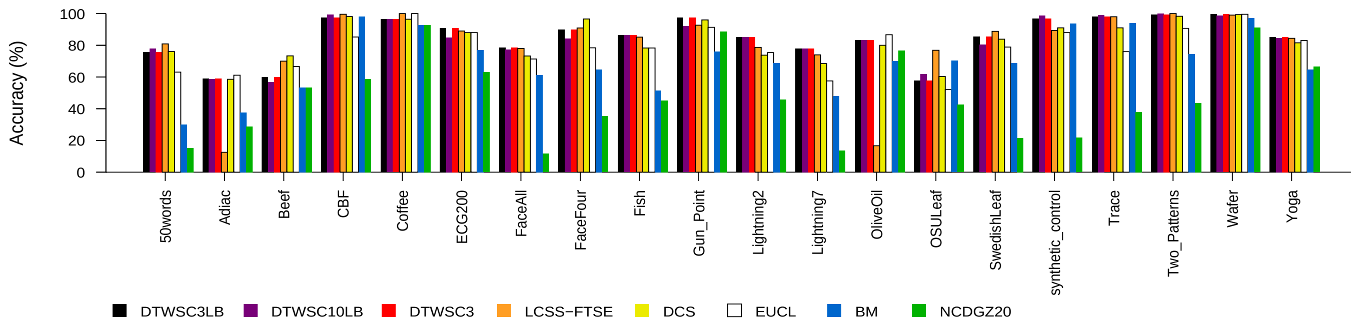


Fig. 15. Accuracy over 20 short time-series datasets.

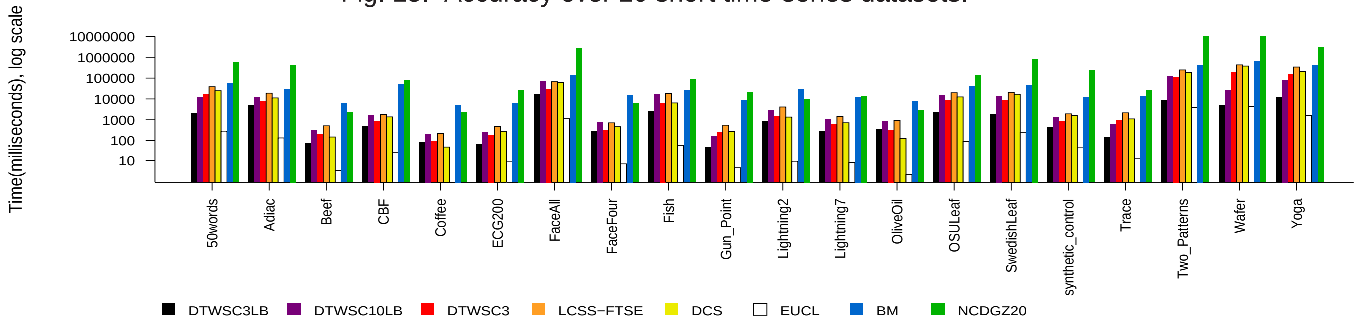


Fig. 16. Speed performance over 20 short time-series datasets.

the fastest calculation while giving a competitive accuracy suggesting that this dataset has little time warping variance. The poor accuracy of other compression based schemes, Bitmap and NCD, for these datasets overshadow whatever speed performance advantages they may have.

The Surveillance dataset results (Figure 13), show that DCS is slower and less accurate than DTW-SC3-LB for this dataset. However, we would like to note that the difference in accuracies between DCS and DTW-SC3-LB is only two classifications(out of 80). Again, the heuristic band found provided the same accuracy as the 3% band which is unsurprising given the findings in [47]. Also, as shown in Figure 14, the graphical representations of the time-series in these long datasets, Surveillance is a repetitive time-series with 6 complete periods while other datasets are non-repetitive. This may help explain a number of interesting points about the results in Figure 13. First, this is one reason why DCS is less accurate than DTW-SC3-LB. Second, since the Bitmap method is discretizing, this helps explain why it does much better on this dataset than the others. Since its word blocks are constant length and fixed alphabet, patterns in reoccuring periodicity is easier to deal with than a non-repetitive dataset. With a variable length dictionary word and no alphabet, DCS overcomes this limitation. Third, when we compare the speed benefits of lower bounding in F2, F6, F7, and RS to Surveillance, we see that a repetitive dataset aids in much better pruning for DTW-SC3.

To summarize, Figures 9 to 13 show that DCS calculates a fast and accurate similarity score that can be used to search for similar time-series. DCS provides the same accuracy as the next leading method DTW-SC3-LB. DCS also provides a faster similarity score than DTW-SC3-LB for non-repetitive long time-series. Different methods may be better suited for different types of datasets; DCS performs well for non-repetitive time-series while DTW-SC-LB gets the most pruning benefit when the data is repetitive. While other compression-based schemes do not compete well with DTW-SC3-LB, we show that a compression-based scheme can be developed to produce high accuracy while retaining its speed advantages.

4.2 Short Time-Series

As stated, DCS is designed with long time-series analysis in mind. However, for completeness, we briefly look at DCS performance when analyzing short time-series.

4.2.1 Datasets

Our short time-series data is from [1]. It consists of 20 time-series datasets of varying length and class cardinality. Table 1 lists all the time-series in this collection along with number of classes and length. The sources of the time-series range from motion capture (GunPoint), to OCR word recognition (50Words), to electrocardiogram measurements (ECG200). Lightning 2 and 7 are down sampled by 5X and 10X respectively from the long F2 and F7 and the DCS results in the next section illustrate how DCS is more suitable for long time-series similarity.

For DCS, the scores calculated are one-sided (non-symmetric) scores to simulate database queries as described above. Parameter exploration was analyzed using the methods described in Section 3.4. DTW warp restriction is limited to the narrow median band of 3% and the classic 10% band (optimal warp bands found at [1] show no statistical advantage over the 3% band). Bitmap times included the discretization time for the query TEST set only, this was done with the MATLAB SAX code from the SAX authors. SAX parameters used are $N = 32$ and $n = 8$. NCD time included discretization and compression steps as these would be required *per query*.

TABLE 1
Characteristics of the short time-series datasets.

Dataset	Classes/ Length	|Train|: |Test|	Dataset	Classes/ Length	|Train|: |Test|
50Words	50/270	450:455	**Adiac**	37/176	390:391
Beef	5/470	30:30	**CBF**	3/128	30:900
Coffee	2/286	28:28	**ECG200**	2/96	100:100
FaceAll	14/131	560:1690	**FaceFour**	4/350	24:88
FISH	7/175	175:175	**GunPoint**	2/150	50:150
Lightning2	2/637	60:61	**Lightning7**	7/319	70:73
OliveOil	4/570	30:30	**OSULeaf**	6/427	200:242
SwedishLeaf	15/128	500:625	**Synthetic**	6/60	300:300
Trace	4/275	100:100	**TwoPattern**	4/128	1000:4000
Wafer	2/152	1000:6174	**Yoga**	2/426	300:3000

4.2.2 Results

Figure 15 and 16 show the classification accuracy and speed (for all the time-series in the TEST sets) over the 20 short time-series (these results are also available in tabular form at [37]). In these figures, NCDGZ3 and NCDGZ10 are omitted since most their accuracies are less than 0.5. For rigor, we computed the statistical significance (see Section 4) and found no statistically significant advantage in accuracy between DCS and DTW-SC and LCSS-FTSE over all 20 datasets. DCS is statistically more accurate than NCD, Bitmaps, and Euclidean on these 20 datasets (DCS is always more accurate than NCD and only less accurate than Bitmaps and Euclidean on 3 and 6 datasets respectively, see the appendix in [37]). With the discretized constant length words, Bitmaps is unable to consistently provide accurate scores for time-series of such short length. Similarly, NCD's reliance on a discretized time-series also makes it inaccurate.

We notice that while DCS is more accurate than Euclidean for the long versions of Lightning F2 and F7 time-series, when the time-series' are down sampled and shorter in Lightning2 and Lightning7, DCS' accuracy advantage is not clear. This is because DCS' accuracy is dependent on the size of the dictionary it builds and thus the length of the time-series.

DCS' computational overhead made is slower than both DTW-SC3 and DTW-SC3-LB as well as Euclidean. DCS was faster than DTW-SC10-LB in 11 of the 20 datasets; there is no statistical significant difference.

To summarize, while DCS is designed for long time-series similarity scoring, we have shown that for short time-series, it can still provide competitive accuracies. While compression based NCD and Bitmaps schemes cannot handle such short data, our method overcomes their limitations.

5 Conclusions and Future Work

In this paper we have proposed a new compression-based similarity scoring method called DCS. DCS uses a compression technique called Continuous-Domain Dictionary Compression, which permits the computation of a similarity score in the native space of time-series. In this way, our method is different from existing compression-based techniques that utilize off-the-shelf compressors to calculate similarity. We have also shown that while methods using off-the-shelf compressors are shown to not fare well for short time-series similarity, our compression method does well enough to make them competitive with existing techniques. For short time-series, we have shown that the DCS accuracy is competitive with those of the other dynamic programming methods.

However, the true strength of our technique lies in the novel nature of building a dictionary on a continuous time-series and compressing another for a relative similarity measure, which allows efficient and effective comparison of long time-series. In the long time-series tests, we have shown that DCS provides similarity scores just as accurate as that of DTW-SC3-LB. Experimental results show DCS is faster than lower bounded DTW (DTW-SC3-LB) on long time-series even though DCS does not prune and scans the entire database. For certain applications such as UPGMA clustering, the input quadratic space dissimilarity matrix must be calculated in a timely manner and the fastest accurate measure should be used. DCS provides a more stable classification accuracy across different types of data than compression-based Bitmaps and NCD.

As part of future work, we plan on exploring additional methods to speed up the parameter selection (though this may have a limited impact, given that it is a one time cost), and developing pruning and indexing techniques for time-series comparisons based on the DCS framework.

Acknowledgements

We would like to thank the reviewers of this paper for their constructive comments on a previous version of this manuscript. This research was funded in part by funding from the National Science Foundation under grants IIS-0929988 and DBI-0926269.

References

[1] The UCR Time Series Classification/Clustering Homepage. "www.cs.ucr.edu/∼eamonn/time_series_data/".

[2] R. Agarwal, C. Faloutsos, and A. R. Swami. Efficient Similarity Search in Sequence Databases. In *FODO*, pages 69–84, 1993.

[3] R. Agarwal, K. Gupta, S. Jain, and S. Amalapurapu. An Approximation to the Greedy Algorithm for Differential Compression. *IBM Journal of Research and Development*, 50(1), 2006.

[4] D. Berndt and J. Clifford. Using Dynamic Time Warping to Find Patterns in Time Series. In *AAAI-94 Workshop on Knowledge Discovery in Databases*, pages 359–370, 1994.

[5] B. Boucheham. Matching of quasi-periodic time series patterns by exchange of block-sorting signatures. *Pattern Recognition Letters*, 29, 2008.

[6] T. Bozkaya, N. Yazdani, and Z. Ozsoyoglu. Matching and Indexing Sequences of Different Lengths. In *CIKM*, 1997.

[7] L. Breiman. Random forests. *Machine Learning*, 45:5–32, 2001.

[8] K. Chan and A.-C. Fu. Efficient Time Series Matching by Wavelets. In *ICDE*, pages 126–133, 1999.

[9] C. Chang and C. Lin. LIBSVM: a library for support vector machines. 2001.

[10] L. Chen and R. Ng. On the Marriage of Lp-norms and Edit Distance. In *VLDB*, pages 792–803, 2004.

[11] L. Chen, M. T. Özsu, and V. Oria. Robust and Fast Similarity Search for Moving Object Trajectories. In *SIGMOD*, pages 491–502, 2005.

[12] X. Chen, B. Francia, M. Li, B. McKinnon, and A. Seker. Shared Information and Program Plagiarism Detection. *IEEE Transaction on Information Theory*, 50(7), 2004.

[13] R. Cilibrasi and P. M. Vitanyi. Clustering by Compression. *IEEE Transactions on Information Theory*, pages 1523–1545, 2005.

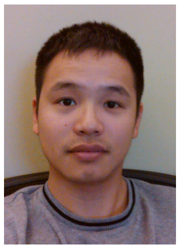
[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2001.

[15] 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. *VLDB*, 2008.

[16] D. Eads, D. Hill, S. Davis, S. Perkins, J. Ma, R. Porter, and J. Theiler. Genetic Algorithms and Support Vector Machines for Time Series Classification, 2002.

[17] A. Elisseeff and M. Pontil. Leave-one-out Error and Stability of Learning Algorithms with Applications. *Advances in Learning Theory: Methods, Models and Applications*, Vol.190, 2003.

[18] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast Subsequence Matching in Time-Series Databases. In *SIGMOD*, pages 419–429, 1994.
[19] D. Goldin and P. Kanellakis. On Similarity Queries for Time-Series Data: Constraint Specification and Implementation. In *Constraint Programming*, pages 137–153, 1995.
[20] R. M. Gray. Vector Quantization. *IEEE ASSP*, 1984.
[21] I. Gronau and S. Moran. Optimal Implementations of UPGMA and other Clustering Algorithms. *Information Processing Letters*, 2007.
[22] A. Hirsh and H. Fraser. Protein Dispensibility and Rate of Evolution. *Nature*, 2001.
[23] C. Jeffery. Synthetic Lightning EMP Data, 2005. http://public.lanl.gov/eads/datasets/.
[24] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley-Interscience, 1990.
[25] E. Keogh. Exact Indexing of Dynamic Time Warping. In *VLDB*, pages 406–417, 2002.
[26] E. Keogh. Tutorial: Mining Shape and Time Series Databases with Symbolic Representations. *SIGKDD*, 2007.
[27] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra. Dimensionality Reduction for Fast Similarity Search in Large Time Series Databases. *KAIS*, 3(3):263–286, 2000.
[28] E. Keogh and S. Kasetty. The Need for Time Series Data Mining Benchmarks: A Survey and Empirical Demonstration. In *SIGKDD*, pages 102–111, 2002.
[29] E. Keogh, S. Lonardi, and C. A. Ratanamahatana. Towards Parameter-Free Data Mining. In *International Conference on Knowledge Discovery and Data Mining*, 2004.
[30] E. Keogh and M. Pazzani. Scaling Up Dynamic Time Warping to Massive Datasets. In *PKDD*, pages 1–11, 1999.
[31] E. J. Keogh, K. Chakrabarti, S. Mehrotra, and M. J. Pazzani. Locally Adaptive Dimensionality Reduction for Indexing Large Time Series Databases. In *SIGMOD*, pages 151–162, 2001.
[32] S.-W. Kim, S. Park, and W. W. Chu. An Index-Based Approach for Similarity Search Supporting Time Warping in Large Sequence Databases. In *ICDE*, pages 607–614, 2001.
[33] J. M. Kleinberg. Two algorithms for nearest-neighbor search in high dimensions. pages 599–608, 1997.
[34] T. Kohonen. Improved versions of learning vector quantization. *Proceedings of the International Joint Conference on Neural Networks*, 1990.
[35] F. Korn, H. Jagadish, and C. Faloutsos. Efficiently Supporting Ad Hoc Queries in Large Datasets of Time Sequences. In *SIGMOD*, pages 289–300, 1997.
[36] N. Kumar, N. Lolla, E. Keogh, S. Lonardi, and C. A. Ratanamahatana. Time-series Bitmaps: A Practical Visualization Tool for working with Large Time Series Databases. *5th SIAM International Conference on Data Mining*, 2005.
[37] W. Lang, M. D. Morse, and J. M. Patel. Dictionary-Based Compression for Long Time-Series Similarity. 2008. http://cs.wisc.edu/∼wlang/compress_extended.pdf.
[38] M. Li, X. Chen, X. Li, B. Ma, and P. Vitanyi. The Similarity Metric. In *14th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2003.
[39] M. Li and P. Vitanyi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer, 1997.
[40] M. Li and Y. Zhu. *Image Classification Via LZ78 Based String Kernel: A Comparative Study*. Springer Berlin, 2006.
[41] J. Lin, E. Keogh, S. Lonardi, and B. Chiu. A Symbolic Representation of Time Series, with Implications for Streaming Algorithms. *DMKD*, 2003.
[42] A. B. Matos. Kolmogorov Complexity in Multiplicative Arithmetic. *DCC-FCUP Technical Report*, 2005.
[43] M. Morse and J. M. Patel. An Efficient and Accurate Method for Evaluating Time Series Similarity. *SIGMOD*, 2007.
[44] H. H. Otu and K. Sayood. A New Sequence Distance Measure for Phylogenetic Tree Construction. *Bioinformatics*, 19(16):2122–2130, 2003.
[45] I. Popivanov and R. Miller. Similarity Search Over Time Series Data Using Wavelets. In *ICDE*, page 212, 2001.
[46] C. Ratanamahatana and E. Keogh. Making Time-series Classification More Accurate Using Learned Constraints. In *SIAM International Conference on Data Mining*, 2004.
[47] C. Ratanamahatana and E. Keogh. Three Myths about Dynamic Time Warping. In *SIAM International Conference on Data Mining*, 2005.
[48] H. Sakoe and S. Chiba. Dynamic Programming Algorithm Optimization for Spoken Word Recognition. *IEEE Trans. Acoustics, Speech, and Signal Proc.*, Vol. ASSP-26(1):43–49, 1978.
[49] Y. Sakurai, M. Yoshikawa, and C. Faloutsos. FTW: Fast Similarity Search under the Time Warping Distance. In *PODS*, pages 326–337, 2005.
[50] P. H. E. Sneath and R. R. Sokal. Numerical Taxonomy. 1973.
[51] T. Syeda-Mahmood, D. Beymer, and F. Wang. Shape-based Matching of ECG Recordings. *EMBS*, 2007.
[52] K. Ueno, X. Xi, E. Keogh, and D.-J. Lee. Anytime Classification Using the Nearest Neighbor Algorithm with Applications to Stream Mining. pages 623–632, 2006.
[53] N. K. Vereshchagin and P. M. Vitanyi. Kolmogorov's Structure Functions and Model Selection. *Transactions on Information Theory*, 15(12), 2004.
[54] M. Vlachos, M. Hadjieleftheriou, D. Gunopulos, and E. Keogh. Indexing Multi-Dimensional Time-Series with Support for Multiple Distance Measures. In *SIGKDD*, pages 216–225, 2003.
[55] M. Vlachos, G. Kollios, and D. Gunopulos. Discovering Similar Multidimensional Trajectories. In *ICDE*, pages 673–684, 2002.
[56] B.-K. Yi and C. Faloutsos. Fast Time Sequence Indexing for Arbitrary Lp Norms. In *VLDB*, pages 385–394, 2000.
[57] B.-K. Yi, H. V. Jagadish, and C. Faloutsos. Efficient Retrieval of Similar Time Sequences Under Time Warping. In *ICDE*, pages 201–208, 1998.
[58] Y. Zhu and D. Shasha. Warping Indexes with Envelope Transforms for Query by Humming. In *SIGMOD*, pages 181–192, 2003.
[59] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.



Willis Lang is currently a PhD graduate student at the University of Wisconsin-Madison. He has research interests in database energy management, spatial data management, and bioinformatics. He has an MSc'08 from the University of Michigan and a BMath'06 from the University of Waterloo.

Michael Morse is currently a database scientist with the MITRE Corp. He has research interests in time series similarity, spatial and temporal data management, and schema integration. He completed his PhD in 2007 at the University of Michigan.

Jignesh M. Patel is an Associate Professor at the University of Wisconsin-Madison. He received his PhD from the University of Wisconsin-Madison in 1998. He is the recipient of an NSF Career Award and multiple IBM Faculty Awards. He has served on a number of Program Committees including SIGMOD, VLDB and ICDE. He has also served as the VLDB 2009 Core Database Technology PC Chair, as Vice-Chair for IEEE ICDE 2005, as an Associate Editor for the Systems and Prototype section of ACM SIGMOD Record, and as an Associate Editor for the IEEE Data Engineering Bulletin. He is a member of the ACM and the IEEE.