2011 11th IEEE International Conference on Data Mining

Time Series Epenthesis: Clustering Time Series Streams Requires Ignoring Some Data

Thanawin Rakthanmanon Eamonn J. Keogh Stefano Lonardi Scott Evans[§]
Department of Computer Science and Engineering
University of California, Riverside
[§]GE Global Research
{rakthant, eamonn, stelo} @cs.ucr.edu, [§]evans@ge.com

***Abstract*—Given the pervasiveness of time series data in all human endeavors, and the ubiquity of clustering as a data mining application, it is somewhat surprising that the problem of time series clustering from a single stream remains largely unsolved. Most work on time series clustering considers the clustering of *individual* time series, e.g., gene expression profiles, individual heartbeats or individual gait cycles. The few attempts at clustering time series *streams* have been shown to be objectively incorrect in some cases, and in other cases shown to work only on the most contrived datasets by carefully adjusting a large set of parameters. In this work, we make two fundamental contributions. First, we show that the problem definition for time series clustering from streams currently used is inherently flawed, and a new definition is necessary. Second, we show that the Minimum Description Length (MDL) framework offers an efficient, effective and essentially parameter-free method for time series clustering. We show that our method produces objectively correct results on a wide variety of datasets from medicine, zoology and industrial process analyses.**

Keywords—time series; clustering; MDL

I. INTRODUCTION

Time series data is pervasive across almost all human endeavors, including medicine, finance, science, and entertainment. As such it is hardly surprising that it has attracted significant attention in the research community [1][3][26][21]. Given the ubiquity of clustering both as a data mining application in its own right and as a subroutine in other higher-level data mining applications (i.e., summarization, rule-finding, etc.), it is surprising that the problem of time series clustering from a *single* time series stream remains largely unsolved. Most work on time series clustering considers the clustering of individual time series, say, gene expressions or extracted signals such as individual heartbeats. The few attempts at clustering the contents of a *single* time series stream have been shown to be objectively incorrect in some cases [16], and in other cases shown to work only on the most contrived datasets by carefully adjusting a large set of parameters. In this work, we make two fundamental contributions. First, we show that the problem definition for time series clustering from streams currently used is inherently flawed. Any meaningful algorithm must avoid trying to cluster *all* the data. In other words, the subsequences of a time series should only be clustered if they are clusterable. This seems to open up a "chicken and egg" paradox. However, our second contribution is to show that the Minimum Description Length (MDL) framework offers an efficient, effective and essentially parameter-free solution to this problem.

We begin by giving the intuition behind the fundamental observation, that clustering of time series from a single stream of data requires ignoring some of the data.

A. Why Clustering Time Series Streams requires Ignoring some Data

The observation motivating our efforts to cluster time series is that any attempt that insists on trying to explain *all* the data is doomed to failure. Consider one of the most obviously "clusterable" time series data sources: motion-captured sign language, such as American Sign Language (ASL). There has been much recent work on nearest-neighbor classification of such data, with accuracies greater than 90% frequently reported [1]. This suggests that a long data stream of ASL might be amenable to clustering, where each cluster maps to a distinct "word" or "phrase."

However, all such data contains Movement Epenthesis (ME) [7][26]. During the production of a sign language sentence, it is often the case that a movement segment needs to be inserted between two consecutive signs to move the hands from the end of one sign to the beginning of the next. These ME segments can be as long as—or even longer than—the true signs, and are typically not performed with the precision or repeatability of the actual words, since they have no meaning. Recent sophisticated sign language recognition systems for continuous streams have begun to recognize that "*automated sign recognition systems need a way to ignore or identify and remove the movement epenthesis frames prior to translation of the true signs*" [26].

What we observed about ASL as a concrete and intuitive example matches our experience with dozens of other datasets, and indicates that this is a pervasive phenomenon. We believe that almost all datasets have sections of data that do not represent a discrete underlying behavior, but simply a transition between behaviors or random drifts where no behavior is taking place. In many datasets, such sections constitute the *majority* of the data. If we are forced to try to model these in our clusters, they will swamp the true significant clusters. We can best demonstrate this effect, and hint at our proposed solution by an experiment on a discrete analogue of time series, in this case English *text*.

We emphasis that this is *just a expository example*, and if we were really assigned to cluster such text data we could do better than the attempt shown below.

Consider the following string *D*, which from left to right mentions three versions of the name David (English, Persian, Yiddish) and three versions of the name Peter (English, Croatian, Danish). Note that all names have five letters each.

David enjoined Peter who identified Davud son of Petar friend to Dovid and Peder, to do what...

1550-4786/11 $26.00 © 2011 IEEE
DOI 10.1109/ICDM.2011.146

IEEE computer society

Here the words between the names are *exactly* the epenthesis previously referred to. To make it more like our time series problem, we can strip out the punctuation and spacing, leaving us:

davidenjoinedpeterwhoidentifieddavudsonofpetarfriendtodovidandpedertodowhat

The discrete analogue of the clustering algorithm in [9] would begin by extracting all the subsequences of a given fixed length. Let us assume for simplicity the length five is used, and thus the data is transformed into:

david
avide
viden
idenj
...
owhat

In Figure 1 we show representative clusters for two values of K, if we perform partitional clustering as in [9] on this extracted data.

Three clusters of equal diameter when $K = 20$

{**whoid**, **davud**, **njoin**, **dovid**, **david**}
{**ified**, **frien**, **oined**, **oiden**, **viden**, **vidan**}
{**todow**, **todov**, **sonof**}

Three clusters of equal diameter when $K = 40$

{**avuds**, **ovida**, **avide**}
{**davud**, **dovid**, **david**}
{**nofpe**, **nedpe**, **andpe**}

Figure 1. Representative partitional clusters from dataset D for two settings of K.

Note that while the cluster of the name variants of David is discovered, we find that under any setting of K there are equally significant *meaningless* clusters, for example {nofpe, nedpe, andpe}. This is in spite of the fact that this can be considered a particularly easy task. Exactly 40% of the signal consists of data we hope to recover, and we deliberately avoided name variants of different lengths (i.e., pieter, pere). In more realistic settings we expect much less of the data to contain meaningful signals. Note also that the problem is not mitigated by using other clustering variants. The problem is inherent in the false assumption that a clustering of a single stream that must explain *all* such data could ever produce meaningful results [16].

B. How MDL Can Help

In contrast to the previous section, it is instructive to see what our proposed algorithm will do in this case. While the details of our algorithm are not introduced until Section IV, we can still outline the basic intuition here.

The original string D has a bit-level representation whose length we denote as $DL(D)$. Our algorithm can be imagined as attempting to losslessly compress the data by finding repeated structure in it. As there is little *exactly* repeated structure, we must find approximately repeated structure and encode the differences. For example, if we find the approximately repeated versions of the name "david", we can think of one version as being a model or *hypotheses* for the data, and encode only the difference between the other occurrences:

H_1 = {1:**david**}

1___enjoinedpeterwhoidentified1___u_sonofpetarfriendto1_o___andpedertodowhat

In terms of MDL we can see **david** as a partial hypothesis H_1 or description of the data. This model has some size, which is simply the length in bits of the word $DL(H_1) = DL(\textbf{david})$. In addition, the size of the remaining data was both *reduced* by factoring out the common structure and (slightly) *increased* by the overhead of the pointers to the dictionary, etc[1]. When encoded with the hypothesis, the length (in bits) of the description of the data is given as $DL(D \mid H_1)$. The total cost of both the hypothesis and the data encoded using the hypothesis is just $DL(H_1) + DL(D \mid H_1)$..

Because this sum is less than the length of the original data $DL(D)$, we feel that we are making progress. Perhaps, however, there is more structure we can exploit. A brief inspection of the data suggests another model, H_2, that exploits *both* repeated names:

H_2 = {1:**david** 2:**peter**}

1___enjoined2___whoidentified1___u_sonof2___a_friendto1_o___and2__d__todowhat

Because $DL(H_2) + DL(D \mid H_2) < DL(H_1) + DL(D \mid H_1)$, we prefer this new hypothesis as a model of the data.

Are we now done? We can try other hypotheses. For example, we could consider the hypothesis H_3 = {1:**david** 2:**peter** 3:**ono**}, attempting to exploit the two occurrences of a pattern "o*o" (i.e.,..**sonof**.. and ..**to** d**o**..). However, because this pattern is short, and only has two occurrences, we cannot break even with the cost of the overhead:

$$DL(H_2) + DL(D \mid H_2) < DL(H_3) + DL(D \mid H_3)$$

Because we cannot find any other hypotheses that produce a smaller model, we invoke the MDL principle to claim that H_2 = {1:**david** 2:**peter**} is the best model of the data D. Here *best* means something beyond simply achieving the greatest compression. We can claim that MDL approach has achieved the most parsimonious explanation of the data, recovering the true underlying structure [8][12][14][18]. In at least this case, where the sentence was contrived as an excuse to use two names trice, MDL *did* recover the true underlying structure.

Note that while our informally stated algorithm does manage to recover the two embedded clusters, it *does not* attempt to explain all of the data. This is a critical observation, in order to cluster a single stream of data, be it discrete or real-valued, we *must* be able to represent and rank solutions that ignore some of the data.

II. Related Work

The tasks of clustering *multiple* time series streams, or many individual time series (i.e., gene expressions) have received significant attention, but the solutions do not inform

[1] In this toy example, we are deliberately glossing over the concrete details of how the pointers are represented and how the amount compression achieved is measured, etc. [18]. We will formalize these details in Section III.

the problem we consider here, the task of clustering a *single* time series stream. The most commonly referenced technique for clustering a *single* time series stream is presented in [9] as a subroutine for rule discovery in time series. In essence the method slides a fixed length window across the stream, extracting all subsequences which are then clustered with *K*-Means. The reader may have already spotted a flaw here; the algorithm tries to explain *all* the data. In [16] (and follow-up works by more than twenty other authors [4][5][10]), it was shown that this method can only produce cluster centers that are sine waves, and the output of the algorithm is essentially *independent of the input*. Note that even if the algorithm did not have these fatal flaws, it assumes the cluster all have equal length, and that we know the correct value of *K*. As we shall show, our method requires neither assumption.

Since the problem with [9] was pointed out in 2005 [16], at least a dozen solutions have been proposed. In Section V.*C* we show that the most referenced of these works [5] does not produce objectively correct results, even after extensive parameter tuning by the original authors on a relatively simple problem.

While there have been some efforts to use MDL with time series [22][24], they all operate on a quantized representation of the data. This has the disadvantage of requiring three parameters (cardinality, dimensionality and window size), eliminating the greatest advantage of MDL, its intrinsically parameter-free nature.

While MDL has had surprisingly little impact in data mining, it is a tool of choice for many bioinformatics problems. For example, working with RNA data, Evans et. al. have proposed a method using data compression and the MDL principle that is capable of identifying motif sequences, some of which were discovered to be miRNA target sites implicated in breast cancer [12]. Moreover, the authors showed the generality of their ideas by applying them, unmodified, to the problem of network traffic anomalies [13]. There is also a significant work on using MDL to mine graphs [14][21], dating back to classic work by Cook et al. [8].

Finally, we note that the task was informed by, and may have implications for many other time series problems, including time series segmentation[2] [3]. To see why, let us revisit the technique of text analogy. It is not obvious how one should segment the three concatenated words "hisabasiais". Perhaps the best we could do is to exploit the known frequencies of bigrams and trigrams, etc. In fact, most time series segmentation algorithms essentially do the real-valued equivalent of this [3]. However, if we see another such triplet of three concatenated words from later in the same stream, for example "withoutabasiais", we can immediately see that "abasia" must be a word[3].

[2] The phrase "*time series segmentation*" is unfortunately overloaded. It can mean approximating the data with the smallest number of piecewise polynomial segments for a given error threshold, or as here; extracting small, discrete, *semantically* meaningful segments of data [3].

[3] *Abasia* is the inability to walk due to impaired muscle coordination.

III. Background and Notation

A. Definitions and Notation

We begin by defining the data type of interest, *time series*:

Definition 1: A *time series T* is an ordered list of numbers. $T = t_1, t_2, \ldots, t_m$. Each value t_i can be any finite number (e.g., for two-byte values they could be integers in range [-32,768, 32,767]) and m is the length of time series T.

Before continuing, we must make and justify a choice. The MDL technique that is at the heart of our algorithm requires *discrete* data, but most time series datasets use four or eight bytes per value, and are thus real-valued. Our solution is simply to cast the *real-valued* numbers into a reduced cardinality version. Does such a reduction lose meaningful information? To test this, we did one nearest-neighbor classification on eighteen public time series datasets, for cardinalities from the original four bytes down to a single bit. Figure 2 shows the results. As we can see, we can drastically reduce cardinality without reducing accuracy. The original four-byte cardinality is typically a by-product of file format convention or hardware specification, and not a claim as to the intrinsic cardinality of the data.

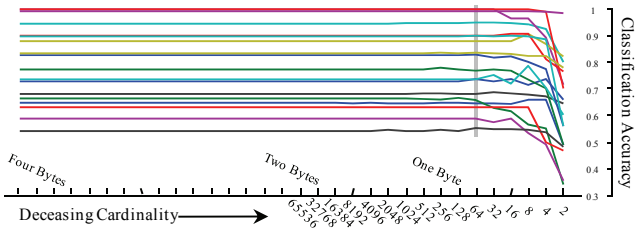

Figure 2. Classification accuracy on 18 time series datasets as a function of the data cardinality. Even if we reduce the cardinality of the data from the original 4,294,967,296 to a mere 64 (vertical bar), the accuracy does not decrease.

We note that there may be other things we *could* have done. For example, the MML framework [25] which is closely related to MDL would allows us to work in original continuous space. However, we choose MDL because it is more familiar and it allows for a more intuitive explanation of our algorithms. Likewise, we have at least a dozen choices of how to discretize the time series (adaptive binning, uniform binning, SAX etc) however, after testing all published algorithms and finding it made little or no difference, we settled on the simple idea shown below in Definition 3.

Based on the observations in Figure 2, we will simply use 64-value (6-bit) cardinality in the rest of this work.

While the source data is one long time series, we ultimately wish to cluster it into sets of shorter *subsequences*:

Definition 2: A *subsequence* $T_{i,k}$ of a time series T is a short time series of length k which starts from position i. Formally, $T_{i,k} = t_i, t_{i+1}, .., t_{i+k}$, $1 \leq i \leq m-k$.

As we previously noted, we are working in a space of reduced cardinality. Because comparing time series with different offsets and amplitudes is meaningless [16], we must

(slightly) adapt the normalization process for our *discrete* representation:

Definition 3: A *discrete normalization* function *DNorm* is a function to normalize a real-valued subsequence T into b-bit discrete value of range $[1,2^b]$. It is defined as followings:

$$DNorm(T) = round\left(\left(\frac{T - min}{max - min}\right) * (2^b - 1)\right) + 1$$

where *min* and *max* are the minimum and maximum value in T, respectively.

Based on the results in Figure 2, b is fixed at 6 for all experiments. We need to define a distance measure; we use the ubiquitous *Euclidean distance* measure:

Definition 4: The distance between two subsequences $T_{i,k}$ and $T_{j,k}$ is the *Euclidean distance* (ED) between $T_{i,k}$ and $T_{j,k}$. Both subsequences must be in the same length. Hence, it is:

$$Dist(T_{i,k}, T_{j,k}) = \sqrt{\sum_{l=0}^{k-1} \left(t_{i+l} - t_{j+l}\right)^2}$$

As we shall see later, the Euclidean distance is *not* general enough to support clustering from time series streams; nevertheless, it is still a useful subroutine to speed up our more general measures.

For both the full time series T and any subsequences derived from it, we are interested in knowing how many bits are necessary to represent it. Normally the number of bits depends solely on the data format, which is typically a reflection of some arbitrary choices of hardware and software. In contrast, we are interested in knowing the *minimum* number of bits to exactly represent the data. In the general case, this number is not calculable, as it is the Kolmogorov complexity of the time series [19]. However, there are numerous ways to approximate this, using Huffman coding, Shanon-Fano coding, etc. Because entropy is a lower bound on the average code length from any such encoding, we can use the *entropy* of the time series as its description length:

Definition 5: The *entropy* of a time series T is defined as:

$$H(T) = -\sum_t P(T = t) \log_2 P(T = t)$$

where $P \log_2 P$ is defined as 0, if $P = 0$.

We can now define the *description length* of a time series.

Definition 6: A *description length DL* of a time series T of length m is the total number of bits required to represent it, that is $DL(T) = m*H(T)$.

The *DL* of a time series using entropy clearly depends on the data itself, not just arbitrary representational choices. Figure 3 shows four time series, which all require 250 bytes to characterize in the original representation, but which have differing *entropies* and thus different description lengths.

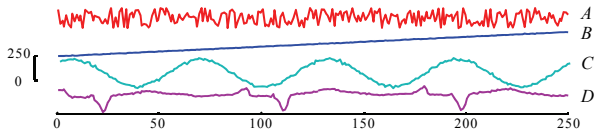

Figure 3. Four time series of length 250 and with a cardinality of 256. Naively all require 250 bytes to represent, but they have different *description lengths*.

The reader may have anticipated the following observation. While the (slightly noisy) straight line B has a high entropy, we would subjectively consider it a simple shape. It *is* simple given our belief (hypothesis) that it is a slightly corrupt version of a straight line. If H is this hypothesis, then we can consider instead the entropy of a time series B', which as shown in Figure 4, is simply B encoded using H, and written as $B' = (B \mid H)$. As a practical matter, to use H to encode B, we simply subtract H from B to get a difference vector B', and encode this simpler vector B'.

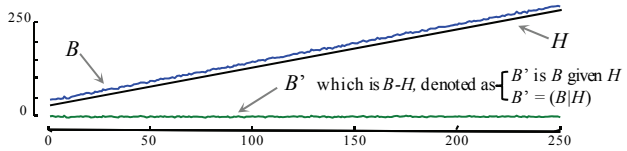

Figure 4. Time series B can be represented exactly as the sum of the straight line H and the difference vector B'.

While the vector B' is also of length **250**, it has only 10 unique values, all of which are small in magnitude, thus its entropy rate is only **2.51** bits. In contrast, B has 172 unique values and an entropy rate of 7.29. Note that if we are given only B', we cannot reconstruct B; we also need to know the slope and mean of the line. Thus, when reporting the overall number of bits for B', we must also consider the number of bits it takes to encode the hypothesis (the line H). We can encode the line simply by recording the heights' **two** locations, the first and last points[4], each of which requires a single **byte**. Thus, the number of bits required to represent B using our hypothesis is:

$$DL(B) = DL(H) + DL(B \mid H) = (2 * 8) + (250 * 2.51) = 643.5 \text{ bits}$$

which is significantly less than the 1,822 bits required for the naive encoding of B without any hypothesis.

Note the straight line would not help in reducing the number of bits required to represent time series C, but using a sine wave as the hypothesis *would* significantly help. This observation inspired one of the principle uses of MDL, model section [23]. Statisticians use this principle to decide if some noisy observations suggest an underlying physical model is produced by, say, a piecewise linear model as opposed to a sinusoidal model. However, our work leverages off a simple but unexploited observation. The hypotheses are not limited to well-defined functions such as sine waves, wavelet basis functions, polynomial models, etc. The hypothesis model can be *any arbitrary time series*. We will see how this observation can be exploited in detail later, but

[4] If we know the time series is z-normalized, we only need one byte to record the line.

in brief: if *k* subsequences of a stream truly form a cluster, then it should be possible to store them in less space by encoding them as a set of difference vectors to the mean of all of them. Thus, we have a potential test to guide our search for clusters.

Having seen this intuition, we can now formalize the notion of *hypothesis* as it pertains to our problem:

Definition 7: A *hypothesis H* is a subsequence used to encode one or more other subsequences of the same length.

As a practical matter, the encoding we use is the one visualized Figure 4, we simply subtract hypothesis *H* from the target subsequence(s) and encoded the difference vector(s). We could encode the difference vector(s) with, say, Huffman encoding, but as we noted in Definition 5, we really only care about the *size* of the encoding, so we simply measure the entropy of the difference vector(s) to get a lower bound of the size of encoding.

A necessary (but not sufficient) condition to place two subsequences *H* and *B* into the same cluster is:

$$DL(B) > DL(B|H)$$

This inequality requires that the subsequence *B* takes fewer bits to represent when *H* is used as a basis to encode it, encoding the intuition that the two subsequences are related or similar.

We can hint at the utility of thinking about our data in terms of hypothesis encoding by revisiting our text example. When a clustering text stream, would it be better to merge **A** or **B**?

A = {david, dovid}, **B** = {petersmith, petersmidt}

The first case allows a tight cluster of two short words, is that better than a looser but longer cluster B? The problem is exacerbated when we consider the possibility of clusters with more than two members: how would we rank the relative utility of the tentative cluster **C** = {bob, rob, hob}?

Normally, clustering decisions are made by considering *Euclidean distance* (or its text counterpart, *Hamming distance*); however, Euclidean distance only allows meaningful comparisons when all the subsequences are the same length. The solution for text, to use the length-normalized Hamming distance, *cannot* be generalized here. The reason is subtle and underappreciated, suppose we have two subsequences of length *k* that are distance *d* apart. If we truncate the end points and measure the distance again, we might find it has increased! This is because we should only compare z-normalized time series when using Euclidean distance[5], and after (re)Z-normalizing the slightly shorter subsequences, we may find they have grown further apart. Thus, the z-normalized Euclidean distance function is not linear in length and is not even monotonic.

[5] The solution of not normalizing the time series would mitigate this problem, but measuring the Euclidean distance between two time series with different offsets or amplitudes produces meaningless results [16].

We have already hinted at the fact that the *DL* function can use extra information, by using "*given*", i.e., $DL(B'|H)$ is the *DL* of B' given *H*. We can now formalize this notion:

Definition 8: A *conditional description length* of a subsequence *A* when a hypothesis *H* is given is

$$DL(A|H) = DL(A - H)$$

Recall from Figure 3 and Figure 4 that the *DL* of a subsequence depends on the structure of the data. For example a constant line has a very low *DL*, whereas a random vector has a very high *DL*. If *A* and *H* are very similar, their difference (*A*-*H*) will be close to a constant line and thus have a tiny *DL*. In essence then, the *DL* function gives us a parameter-free test to see if two subsequences should be clustered together.

We generalize the notion of *DL* to multiple sequences next. We can apply the same spirit by using a hypothesis to calculate the minimum number of bits required to keep a cluster. We call this *description length of a cluster*:

Definition 9: A *Description Length of a Cluster* (*DLC*) **C** is the number of bits needed to represent all subsequences in **C**. In this special case, *H* is the center of the cluster. Hence, the description length of cluster **C** is defined as:

$$DLC(\mathbf{C}) = DL(H) + \sum_{A \in \mathbf{C}} DL(A|H) - max_{A \in \mathbf{C}} DL(A|H)$$

The above *DLC* gives us a primitive to measure the reduction in bits achieved by encoding data with a hypothesis. Our clustering algorithm is essentially a search algorithm, and there are three operators that avail of the *DLC* definition to test how many *bits* a particular choice can save. Thus, these three operators fall under the umbrella definition of *bitsave*:

Definition 10: A *bitsave* is the total number of bits saved after applying an operator that creates a new cluster, adds a subsequence to an existing cluster, or merges two existing clusters together. It is the difference in the number of bits before and after applying a given action:

bitsave = *DL*(*Before*) - *DL*(*After*)

In detail, the *bitsave* for each operator is defined as following:

1) Creating a new cluster **C'** from subsequences *A* and *B*

bitsave = *DL*(*A*) + *DL*(*B*) - *DLC*(**C'**)

2) Adding a subsequence *A* to an existing cluster **C**

bitsave = *DL*(*A*) + *DLC*(**C**) - *DLC*(**C'**)

where **C'** is the cluster **C** after including subsequence *A*.

3) Merging cluster $\mathbf{C_1}$ and $\mathbf{C_2}$ to a new cluster **C'**.

bitsave = *DLC*($\mathbf{C_1}$) + *DLC*($\mathbf{C_2}$) - *DLC*(**C'**)

Note that, as we discussed earlier, we do *not* use Euclidean distance to make decisions about which subsequences to place into which clusters. We use only use Euclidean distance in two subroutines: motif discovery and finding the closest subsequence from a given cluster center. More details are in the following sections.

IV. CLUSTERING ALGORITHM

Having introduced the necessary notation, we are finally in a position to introduce our algorithm. We begin by giving a simple text and visual intuition in the next section, and follow by giving detailed and annotated pseudo code in Section IV.*B*.

A. The Intuition behind Stream Clustering

Recall that our input is a single time series like the one shown in Figure 5.*bottom* and our required output is a set of clusters -- possibly of different lengths and sizes. Recall that the union of all the subsequences in this set of clusters may only cover a fraction of the input time series. Indeed, for pathological cases we are given a pure noise time series, we want our algorithm to return a *null* set of clusters. In Figure 5 we show our running example. It contains the interwoven calls of two very different species of birds.

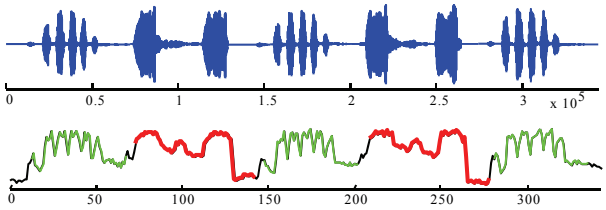

Figure 5. Two interwoven bird calls featuring the *Elf Owl*, and *Pied-billed Grebe* are shown in the original audio space (*top*), and as a time series extracted by using MFCC technique and then clustered by our algorithm (*bottom*).

Our proposed clustering algorithm is a bottom-up greedy search over the space of clusters. For the moment, we will ignore the computational effort that it requires and simply explain *what* is done, leaving the *how* it is (efficiently) done for the next section.

Our algorithm is an iterative merging algorithm similar in spirit to an agglomerative clustering algorithm [15]. However, the differences are telling and worth enumerating:

- Our algorithm typically stops merging before explaining all the data, thus producing a *partitioning* a subset of the data, not producing a *hierarchy* of all the data.
- Agglomerative clustering algorithms are typically implemented such that they require quadratic space, our algorithm has only linear space requirements[6].
- Most critically, agglomerative clustering algorithms assume the K items of a *fixed* dimensionally (subsequence length) to be clustered are inputs to the algorithm. However, we do not know how many items will ultimately be clustered, or even how long the items will be.

Similar to agglomerative clustering, we have a search problem that uses operators, in our case, *create*, *add*, and *merge* (Definition 10). When the algorithm begins, only *create* is available to us.

[6] Linear space agglomerative clustering algorithms *do* exist, but require highly multiply redundant calculations to be performed, and are thus rarely used due to their lethargy.

We begin by finding the best initial pair of subsequences to combine so that we may *create* a cluster of two items. To find this best pair, we treat one as a hypothesis and see how well it encodes the other (Definition 8). The pair that reduces the bit cost the most is the pair of choice. This is shown in Figure 6 as **Step 1**.

There is a potential problem here. Even if we fix the length of subsequences to consider to a constant s, the number of candidate pairs to consider is quadratic in the length of the time series, approximately $O((m-s)^2/2)$. Furthermore, there are no known shortcuts that let us search this space in subquadratic time.

The solution to this problem is to note that Euclidean distance and conditional description length are highly correlated where either of them is small. We can leverage off this fact because there exist very fast algorithms to find the closest pair of subsequences (which are known as *time series motifs* [20]) under Euclidean distance. So rather than a brute force search using the conditional description length, we do a fast motif search and then test the motif pair's conditional description length.

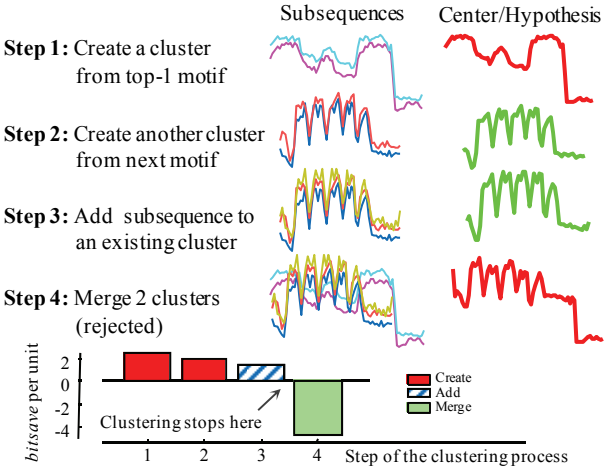

Figure 6. A trace of our algorithm on the bird call data shown in Figure 5.*bottom*.

In the next stage of the algorithm, there are two operators available to us. We can either *add* a third item to our existing cluster of size two, or we can *create* a new cluster, possibly of a different length. In Figure 6 in **Step 2**, we can see that in this case our scoring functions suggest creating a new cluster is the better option in.

In the subsequent phase of the algorithm, it happens that *all* operators are available to us: we could try to *create* a new cluster, we could *merge* our two existing clusters, or we could *add* a subsequence to one of our two clusters. As we can see in Figure 6, **Step 3**, the last option is chosen.

In the next iteration, the cheapest operator was to merge our two existing clusters as shown in **Step 4**. However, doing this does not decrease the size of the representation—it *increases* it. As such, our algorithm terminates after returning the two clusters it had created up to **Step 3**. The only other way that our algorithm can terminate is if it simply runs out of data to cluster.

B. Our algorithm in detail

As we noted in the last section, our algorithm is a bottom-up search algorithm. The input is a single time series, and the output is a set of clusters of subsequences. Our algorithm can cluster subsequences of different lengths, and it does not require the number of clusters to be specified.

There are three operators in our search algorithm: *create*, *add*, and *merge*. In each step we do all (legal) operations and choose the operator which maximizes the number of bits saved as measure by *bitsave* (Definition 10). The current clusters are updated with respect to that choice.

The algorithm can terminate in just two ways; either the best possible choice cannot save any bits, or all data is used up.

Most attempts to cluster time series [5][9][10] suffer from a surfeit of parameters. Our algorithm allows essentially none. However, if we allow subsequences that are too short, we can get pathological results in some cases. For example, there are only two possible z-normalized subsequences of length two. Moreover, a user may wish to bias the algorithm towards certain clustering. For example, for electrical power demand load we may be interested in weekly *or* daily patterns. Thus, as shown in TABLE I, we allow the user the option of suggesting an approximate length *s*.

The algorithm begins by initializing the cluster set to *empty*, then it enters a loop until no more bits can be saved (line 2) or it runs out of data. Within each iteration the loop, we perform three operators *create* (line 4-9), *add* (line10-15), and *merge* (line16-22), and we keep the results of the most parsimonious operator.

For the *create* process, we call a subroutine to find time series motifs under Euclidean distance using the fastest currently known technique [20]. Because we do not know how long the subsequences in the cluster should be, the algorithm runs `MotifDiscovery` multiple times on different lengths of motif (line 5). If the new cluster is created, then the number of bits saved is calculated (line 7). The temporary version of updated clusters are kept (line 8) and used if the algorithm eventually chooses to *create* this cluster (line 24). Recall that the details of function `ComputeBitsave` are provided in Definition 9 and 10.

It is possible to *add* a subsequence into an existing cluster (line 10-15). We first find the most similar subsequence in the input time series with respect to the center of a given cluster (line 11); we can achieve this task by using any nearest-neighbor search algorithm [11], including brute force search. After the search, the cluster is updated to include that nearest subsequence (line 12), the number of bits saved is calculated, and the temporary clusters are recorded (line 13-14).

For our last operator, any pair of clusters is allowed to *merge* (line 18); we then compute the number of bits saved for each pair, and record the temporary cluster.

After the algorithm measures the number of bits saved from all possible choices, the final cluster is updated with respect to the choice that maximizes the number of bits saved (line 23-24).

TABLE I. MAIN TIME SERIES STREAM CLUSTERING ALGORITHM

Input: *ts* : time series,
s : approximate length
Output: *cluster* : final cluster of subsequences

```
1   cluster = {}
2   while bitsave>0
3     bitsave=-∞, i=0
      // create new cluster
4     for len = s to 2s
5       (A,B)  = MotifDiscovery(ts,len)
6       C'     = CreateCluster(a,b)
7       bs(++i)= ComputeBitsave(C',A,B)
8       cluster'(i) = clusterU{C'}
9     end for
      // add subsequence to an existing cluster
10    for C ∈ cluster
11      A  = NearestNeighbor(ts,C)
12      C' = AddToCluster(C,A)
13      bs(++i)= ComputeBitsave(C',C,A)
14      cluster'(i) = clusterU{C'}-{C}
15    end for
      // merge 2 clusters
16    for C1 ∈ cluster
17      for C2 ∈ cluster and C1~=C2
18        C' = MergeClusters(C1,C2)
19        bs(++i)=ComputeBitsave(C',C1,C2)
20        cluster'(i)=clusterU{C'}-{C1}-{C2}
21      end for
22    end for
      // update the result
23    [bitsave ind] = max(bs);
24    cluster = cluster'(ind);
25  end while
```

We have glossed over an important detail: the two items being combined by the merge/add operators may be of different lengths. To allow this critical flexibility, we use a simple data structure to record a cluster. For any given cluster `C`, `C.size` records the number of subsequences in the cluster, `C.cen` is the center of the cluster, C.seq is a set of subsequences in the cluster, and `C.shift` is a set of shift positions (i.e., offsets) of each subsequence in C when it aligned to the `C`.cen. Note that to compute the conditional description length (Definition 8), a subsequence and its hypothesis must be of the same length.

TABLE II shows how a new cluster can be created from two subsequences of the same size. Because those two subsequences are from motif discovery under Euclidean distance, their align position is set to 0 (line 4). The center of new cluster is the average of those two subsequences.

When we want to add a subsequence *A* to an existing cluster `C`, the new center is created by the weighted average of the current center and the subsequence (line 1 of TABLE III). Because *A* is the nearest neighbor of `C.cen`, no offset alignment is needed for *A*.

TABLE IV shows how two clusters of different lengths can be merged into the same cluster. The new cluster contains all subsequences from both clusters (line 1-2). Because two clusters may be different lengths, we need to align them before finding the new center. As the Figure 6 example shows, **Step 4** merges two clusters from **Step 1** and **Step 3**. The red center in **Step 1** is longer than the green center in **Step 3**. We align the red center at all possible offsets (line 6), and then create a new center by averaging

two current centers. Parts of the new centers are created by weighted averaging from all (one *or* two) centers that cover that part (line 7-9). To make a decision among all possible offsets, MDL plays an important role again; at each offset, *bitsave* is calculated (line 11), and we choose the offset which can save the maximum number of bits (line 22-23). Similar to the code in line 6-13, which evaluates all offsets when $\mathbf{C_1}$.cen moves *into* $\mathbf{C_2}$.cen, the code in line 14-21 evaluates the inverse when $\mathbf{C_1}$.cen moves *out of* $\mathbf{C_2}$.cen.

TABLE II. *Create* OPERATOR

```
Function C = CreateCluster(A,B)
1  C.size = 2;
2  C.cen  = (A+B)/2
3  C.seq  = [A; B]
4  C.shift= [0; 0]
```

TABLE III. *Add* OPERATOR

```
Function C = AddToCluster(C,A)
1  C.cen   = (C.cen*C.size+A*1)/(C.size+1)
2  C.size  =  C.size+1
3  C.seq   = [C.seq; A]
4  C.shift = [C.shift; 0]
```

TABLE IV. *Merge* OPERATOR

```
Function C' = MergeClusters(C1,C2)
1  C'.seq = [C1.seq; C2.seq]
2  C'.size = C1.size+C2.size
3  n1=C1.size,     m1=length(C1.cen)
4  n2=C2.size,     m2=length(C2.cen)
5  i=0
6  for off = 0 to m2
7    cen1 = [C2.cen(1,off), C1.cen]
8    cen2 = [C2.cen, C1.cen(1,m1+off-m2)]
9    C'.cen = (cen1*n1+cen2*n2)/(n1+n2)
10   C'.shift = [C1.shift+off; C2.shift]
11   bs(++i) = ComputeBitsave(C',C1,C2)
12   Ctmp(i) = C'
13 end for
14 for off = 1 to m1
15   cen1 = [C1.cen, C2.cen(1,m2+off-m1)]
16   cen2 = [C1.cen(1,off), C2.cen]
17   C'.cen = (cen1*n1+cen2*n2)/(n1+n2)
18   C'.shift = [C1.shift; C2.shift+off]
19   bs(++i) = ComputeBitsave(C',C1,C2)
20   Ctmp(i) = C'
21 end for
22 [bitsave ind] = max(bs);
23 C' = Ctmp(ind);
```

To summarize, our algorithm contains three operators (*create*, *add*, and *merge*), which all use MDL to decide the best choice at each step of the clustering. As there are no known indexing/motif discovery algorithms for MDL, we avail ourselves of two fast external modules that use Euclidean distance for motif discovery [20] and nearest neighbor search [17]. Using Euclidean distance as a fast proxy for MDL is possible because they are highly correlated when both are small (details omitted for brevity).

V. EXPERIMENTAL RESULTS

We begin by stating our experimental philosophy. To ensure our experiments are reproducible, all codes/data are available at [27]. In addition, the site contains many more experiments omitted due to space limitations. Furthermore, the website contains video animations of the clustering process for each dataset.

A. Comparison to Ground Truth

We begin by considering a time series for which we have access to the ground truth (albeit indirectly). Consider the time series shown in Figure 7.*top*. A visual inspection gives a hint of some structure, but even on this tiny example, it is not clear exactly what the clustering should be -- or even what is the natural length for potential clusters. This dataset was obtained by taking an audio snippet of a recording of Edgar Allen Poe's poem "The Bells" and transforming it in to the Mel-Frequency Cepstral Coefficients (MFCC) retaining only the first coefficient.

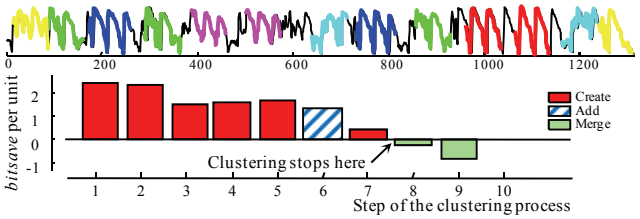

Figure 7. *top*) 29.8 seconds of an audio snippet, represented by the first coefficient in MCFF space, and then annotated with colors to reflect the clusters. *bottom*) A trace of the steps use to produce the clustering.

The clustering we obtained looks subjectively intuitive; however, because of the original source material we are in a unique position to do a more objective test. TABLE V shows the original source text brushed with the colors reflecting the clustering obtained.

TABLE V. THE TEXT CORRESPONDING TO THE TIME SERIES SHOWN IN FIGURE 7, ANNOTATED BY COLOR/FONT

Original Order	Grouped by clusters
In a sort of Runic rhyme, To the throbbing of the bells-- Of the bells, bells, bells, To the sobbing of the bells; Keeping time, time, time, As he knells, knells, knells, In a happy Runic rhyme, To the rolling of the bells,-- Of the bells, bells, bells-- To the tolling of the bells, Of the bells, bells, bells, bells, Bells, bells, bells,-- To the moaning and the groaning of the bells.	bells,bells, bells Bells, bells, bells Of the bells, bells, bells Of the bells, bells, bells the throbbing of the bells the sobbing of the bells the tolling of the bells To the rolling of the bells To the moaning and the time, time, time knells, knells, knells sort of Runic rhyme groaning of the bells.

The results are not perfect with reference to the text version. Recall that we are only considering *one* of the MFCC coefficients, instead of the ten plus typically used in speech processing. This allows some collisions, such as "time" and "knells". However, the structure recovered by our algorithm is significant. Note that our clusters are of different sizes (three items, and two items) and of different lengths (from 55 points to 70 points). Also, note that we could have had a single cluster of eighteen occurrences of the word "bells." However, that would have obfuscated the information that this word tends to be repeated in this work, as in "bells, bells, bells." These longer clusters are arguably more parsimonious.

B. Clustering a Noisy Dataset

In Figure 8 we show the results of clustering a noisy industrial dataset. The data comes from an industrial wire winding process. The original data consists of seven dimensions; here we show only the results of clustering the noisiest channel, labeled U1 (the results on the other channels are at [27]). Note that the data has significant non-uniform noise, including spikes and dropouts. While we do not have access to the ground truth here, the clusters, which have different sizes and length, clearly have the property of being similar *within* a cluster and dissimilar *between* clusters. Note that approximately 26% of the data remains unclustered.

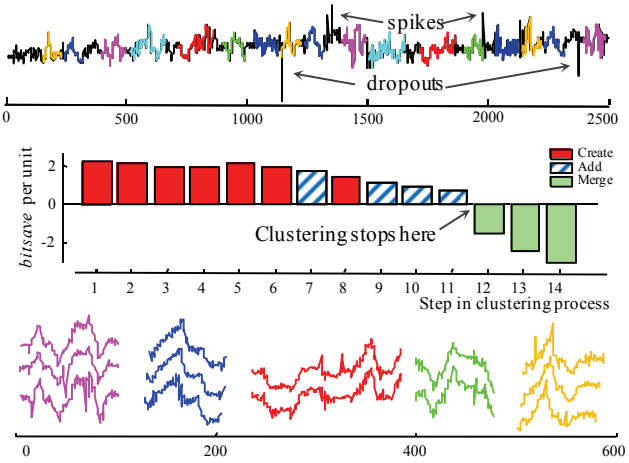

Figure 8. *top*) Dimension U1 of the *Winding* dataset. *middle*) A trace of the clustering steps produced by our algorithm. *bottom*) Representative clusters obtained.

C. Comparison to other Methods

As we noted above there are few candidate strawmen to compare our work to. Here we compare our work to the most referenced work in the literature. In a sequence of papers, Chen proposes a series of fixes for the stream clustering problem [4][5][6]. He demonstrates his ideas mostly on synthetic data; however, as shown in Figure 9.*right*, he also tests on short section of the Koski heartbeat dataset.

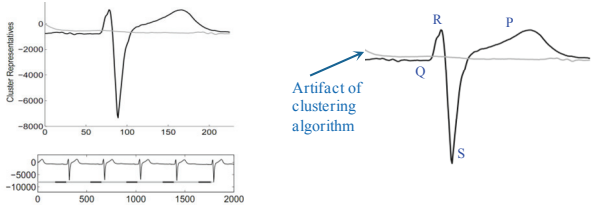

Figure 9. *left*) A screen dump of fig.11 from [5]. The original caption read "TF Clustering: Koski-ECG result". *right*) An annotation of the clusters by a USC cardiologist.

While the results are perhaps reasonable, it is not clear why we should have two clusters here since there is clearly just one heartbeat. In addition, there is a subtle artifact noticed by cardiologist, Dr. Helga Van Herle, whom we asked to examine this. The slight slope on the light-gray cluster show in Figure 9.*left* is not in the data; it comes from the fact that the input data is not an integer multiple of beats, instead being roughly 5.2 beats. Since the algorithm is trying to explain *all* the data, it must explain the *extra* P-wave by averaging it into a place where it does not belong. Furthermore, as acknowledged in the original paper, the algorithm requires the setting of several parameters and "magic numbers" (i.e., "*we chose p as the number of points in the time series divided by 15.*."). Finally, we note in passing that the algorithm requires multiple calls to a *quadratic* space and time (in the length of the time series) algorithm, which would make it impractical for many real data mining problems. Our algorithm requires linear space.

In Figure 10, we show the clustering we achieved on *exactly* the same dataset. We believe the results here are intuitively correct, discovering a complete *single* heartbeat as the cluster. Note that our algorithm explains 87.5% of the data; it does not try to explain the extra P-wave "bump" caused by the fact that we do not have an integer number of heartbeats.

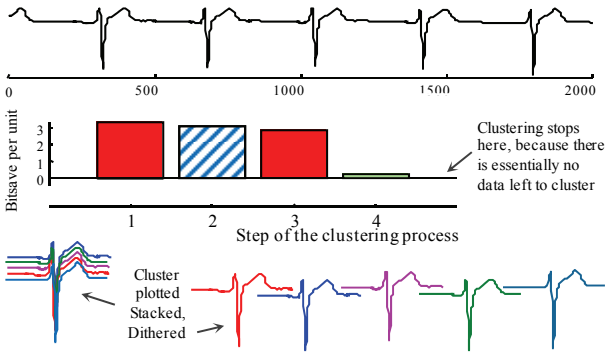

Figure 10. *top*) The same 2,000 datapoints from Koski-ECG as used in Figure 9. *middle*) A trace of the clustering steps produced by our algorithm. *bottom*) the single cluster discovered has five members.

D. Scalability

From our algorithm in Section IV.*B*, assume that `MotifDiscovery` takes time $O(T)$. In each *create* step, `MotifDiscovery` is called multiple times to find motifs of different length; we run it at most $O(s)$ times. Because each subsequence is of length at least s, there are at most $O(m/s)$ new clusters to be created. This is why the running time for creating new clusters is $O(T*s*m/s) = O(mT)$.

Assume that `NearestNeighbor` can be finished in time $O(ms)$. The maximum number of clusters we can have is $O(m/s)$, and the original time series can be updated only when a new motif is discovered, so the number of clustering steps (cf. line 2 in TABLE I) is at most $O(m/s)$. Thus, for add steps we have $O(ms * m/s * m/s) = O(m^3/s)$.

For *merge* steps, if a cluster is created by merging k clusters so far, the number of subsequences in that cluster is at most $O(k)$. The length of its center is at most $O(ks)$; therefore, the number of possible offsets is $O(ks)$, and *bitsave* calculation is finished in time $O(k^2s^2)$. The maximum number of clusters we can have is at most O(m/s), so we can have cluster of size k at most $O(m/sk)$ clusters, and there are at most $O(m/s)$ steps in our algorithm. This means that the running of *merge* steps is at most $O(m/s*(m/sk)^2*k^2s^2) = O(m^3/s)$. Hence, the total running time of our algorithm is at

most $O(mT+m^3/s)$ where T is a running time for a motif discovery. The empirical behavior is shown in Figure 11.

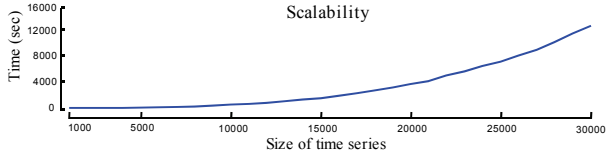

Figure 11. Running time of our algorithm on Koshi data when s = 350.

To put these results in perspective, the ornithology lab we are working with has spent *months* collecting data in the field (cf. Figure 5), so they are willing to wait the hour we require to cluster several minutes of audio. Nevertheless, we believe that a 100X speedup will soon be possible simply by caching some near redundant motifs calculations.

VI. DISSCUSION OF THE MDL CHOICE

Now that the reader has gleaned some intuition for our algorithm and its utility for clustering data, we will briefly revisit a discussion of why MDL on a discretized time series is our choice of measure to steer the clustering search.

We cannot use Euclidean distance (or the related *correlation* or *Dynamic Time Warping* etc [11][17]) directly because it does not allow us to compare the relative merits of clusters of different lengths or different sizes. In contrast, MDL does allow such meaningful comparisons. Moreover, in the limited case when MDL and Euclidean distance can be compared (when time series lengths are the same), we find that the two measures are *highly* correlated so long as they are small (if both are destined to be large, it does not really matter how correlated they are).

We work in the discrete space rather than the original continuous space because MDL requires it, and because working with the discretized time series makes *no perceptible difference* in classification (shown in Figure 2) or in similarity search, indexing, motif discovery or outlier discovery (omitted for brevity).

VII. CONCLUSIONS

In this work, we have shown that any attempt to cluster a single time series stream that insists on explaining *all* the data is almost certainly doomed to failure. We introduced a clustering representation that has the expressive power to ignore some of the data, and can have clusters with different length subsequences. We further showed an efficient and parameter-lite MDL based algorithm to perform the clustering. We have shown on our algorithm is effective on a wide variety of datasets.

ACKNOWLEDGEMENT

We would like to acknowledge the financial support for our research provided by the Royal Thai Government and NSF grants 0803410 and 0808770.

REFERENCES

[1] V. Athitsos, H. Wang, and A. Stefan, "A database-based framework for gesture recognition," Personal and Ubiquitous Computing, vol. 14, no. 6, 2010, pp. 511-526.

[2] T. Bastogne, H. Noura, A. Richard, and J. M. Hittinger, "Application of subspace methods to the identification of a winding process," Proc. of the 4th European Control Conference, Brussels, Belgium, 1997.

[3] D. Bouchard and N. I. Badler, "Semantic Segmentation of Motion Capture Using Laban Movement Analysis," IVA, 2007, pp 37-44 .

[4] J. R. Chen. "Making Subsequence Time Series Clustering Meaningful," ICDM, 2005, pp. 114-121.

[5] J. R. Chen., "Useful Clustering Outcomes from Meaningful Time Series Clustering," The Australasian Data Mining Conference. 2007.

[6] J. R. Chen., Making clustering in delay-vector space meaningful. Knowl. *Inf. Syst.* 11, 3 (2007), 369-385.

[7] Z. J. Chuang, C. H. Wu, and W. S. Chen, "Movement Epenthesis Generation Using NURBS-Based Spatial Interpolation," IEEE Trans. Circuit and Systems for Video Technology, vol. 16, no. 11, Nov. 2006, pp. 1313-1323.

[8] D. J. Cook and L. B. Holder, "Substructure Discovery Using Minimum Description Length and Background Knowledge," J. Artificial Intelligence Research, vol. 1 , 1994, pp. 231-255.

[9] G. Das, K. Lin, H. Mannila, G. Renganathan, and P. Smyth, "Rule Discovery from Time Series," Proc. of the 3rd KDD, 1998, pp. 16-22.

[10] A. M. Denton, C. A. Basemann, and D. H. Dorr, "Pattern-based time-series subsequence clustering using radial distribution functions," Knowledge and Information Systems journal, vol. 18, No. 1, Jan. 2009, pp. 1-27.

[11] H. Ding, G. Trajcevski, P. Scheuermann, X. Wang, and E. J. Keogh, "Querying and mining of time series data: experimental comparison of representations and distance measures," PVLDB, vol. 1, no. 2, 2008, pp. 1542-1552.

[12] S. C. Evans et. al. "MicroRNA target detection and analysis for genes related to breast cancer using MDLcompress," EURASIP J. Bioinform. Syst. Biol., 2007, pp. 1-16.

[13] S. C. Evans, E. Eiland, T. S. Markham, J. Impson, and A. Laczo, "MDLcompress for Intrusion Detection: Signature Inference and Masquerade Attack," MILCOM, Orlando, Florida, 2007.

[14] I. Jonyer, L. B. Holder, and D. J. Cook, "MDL-based context-free graph grammar induction and applications," Journal on Artificial Intelligence Tools, vol. 13, no. 1, 2004, pp. 65-79.

[15] S. D. Kamvar, D. Klein, and C. D. Manning, "Interpreting and Extending Classical Agglomerative Clustering Algorithms using a Model-Based approach," ICML, 2002, pp. 283-290.

[16] E. J. Keogh and J. Lin, "Clustering of time-series subsequences is meaningless: implications for previous and future research," Knowl. Inf. Syst., vol. 8, no. 2, 2005, pp. 154-177.

[17] E. J. 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, 2003, pp. 349-371.

[18] H. Li and N. Abe, "Clustering Words with the MDL Principle," Proc. of the 16th Int' Conf' on Computational Linguistics, 1996, pp. 5-9.

[19] M. Li and P. Vitanyi, An Introduction to Kolmogorov Complexity and Its Applications, 2nd ed., Springer Verlag, 1997.

[20] A. Mueen, E. J. Keogh, and N. B. Shamlo, "Finding Time Series Motifs in Disk-Resident Data," ICDM, 2009, pp. 367-376.

[21] S. Papadimitriou, J. Sun, C. Faloutsos, and P. S. Yu, "Hierarchical, Parameter-Free Community Discovery," PKDD 2008, pp. 170-187.

[22] E. Pednault, "Some Experiments in Applying Inductive Inference Principles to Surface Reconstruction," IJCAI, 1998, pp. 1603-09.

[23] R. A. Stine, "Model Selection Using Information Theory and the MDL Principle," Sociological Methods and Research, vol. 33, no. 2, Nov. 2004, pp. 230-260.

[24] Y. Tanaka, K. Iwamoto, and K. Uehara, K. "Discovery of time-series motif from multi-dimensional data based on MDL principle," Machine Learning, vol. 58, no. 2, 2005.

[25] C. S. Wallace and D. M. Boulton, 1968. An information measure for classification. Computer Journal 11, 2 (August 1968), 185-194.

[26] R. Yang, S. Sarkar, and B. L. Loeding, "Handling Movement Epenthesis and Hand Segmentation Ambiguities in Continuous Sign Language Recognition Using Nested Dynamic Programming," IEEE PAMI, vol. 32, no. 3, 2010, pp. 462-477.

[27] Supporting webpage. *http://www.cs.ucr.edu/~rakthant/TSEpenthesis*