A Self-Learning and Online Algorithm for Time Series Anomaly Detection, with Application in CPU Manufacturing

Xing Wang[1] Jessica Lin[1] Nital Patel[2] Martin Braun[2]
[1]George Mason University [2]Intel Corporation
{xwang24, jessica}@gmu.edu {nital.s.patel, martin.w.braun}@intel.com

ABSTRACT

The problem of anomaly detection in time series has received a lot of attention in the past two decades. However, existing techniques cannot locate where the anomalies are within anomalous time series, or they require users to provide the length of potential anomalies. To address these limitations, we propose a self-learning online anomaly detection algorithm that automatically identifies anomalous time series, as well as the exact locations where the anomalies occur in the detected time series. We evaluate our approach on several real datasets, including two CPU manufacturing data from Intel.[1] We demonstrate that our approach can successfully detect the correct anomalies without requiring any prior knowledge about the data.

CCS Concepts

•**Computing methodologies** → **Anomaly detection;** •**Mathematics of computing** → *Time series analysis;* •**Information systems** → *Data stream mining;*

Keywords

Time Series; Anomaly Detection; Self-learning

1. INTRODUCTION

A significant volume of manufacturing data is time series in nature, e.g. sequence of events in an equipment log (1GB/lot), trace data generated by equipment sensors, factory events; test results logged by a tester, etc. This volume of data is overwhelming to the end user. In a recent example, the process generates 630MB of trace data per lot, and engineers struggled for a year to understand and characterize relationships in the data. Nevertheless, fault detection, or excursion prevention (EP), methodologies—typically developed to characterize and monitor such trace data—have remained static for over two decades. New approaches for large scale data mining and knowledge discovery are not being leveraged. As an example, current fault or excursion detection techniques rely on the analysts to manually segment the time series into several characteristic windows, and in each window, the analysts then set the thresholds to detect deviations. Our work aims to automate the entire process and introduces a new self-learning and online anomaly detection algorithm for time series data to mitigate the aforementioned problems.

Current methods that detect anomalies in time series can be categorized into detection of point anomalies, structural anomalies and anomalous time series. Point anomalies, commonly known as outliers, are data points that are significantly different from others [6, 21]. Structural anomalies are subsequences whose shape do not conform to the rest of the observed, or expected patterns [13, 29, 21]. The detection of anomalous time series, which we refer to as whole time series anomaly, aims to detect time series whose average deviation from other time series is significant [8, 15].

For fault detection in manufacturing data, it is critical to detect not only the anomalous time series, but also the exact locations of the structural anomalies. Existing methods for time series anomaly detection focus on either detecting local (subsequence) anomalies in a long time series, or detecting anomalous time series in a group of time series data.

For example, window-based anomaly detection techniques [13, 14, 21] can find local structural anomalies; however, the user must specify the number and length of the expected anomalies. In real-world applications, there may be new anomalies with unpredictable behaviours and unknown length. For window-based techniques to detect all anomalies, they have to try every possible length as input. However, iterating through all possible lengths is inefficient, and requires specific criteria to determine the true anomalies. Such requirement on window length thus limits the application of current methods to only those where users have prior knowledge of the data or the anomalies.

A well-known window-based anomaly detection technique, time series discord discovery [13], defines an anomaly as the subsequence in a long time series that has the largest distance to its (k^{th}) nearest neighbor (the original algorithm sets $k = 1$). However, the effectiveness of the approach greatly relies on the user-defined window length and parameter k. The technique would fail if there exist clusters of anomalies containing more than k instances, or if there exist multiple clusters of anomalies of varying lengths or sizes. It can be generalized to finding anomalous *whole time series* in a time series dataset [27]. However, the result is based on

[1]This work was supported in part by Intel Corporation and Semiconductor Research Corporation (SRC).

CIKM'16 , October 24-28, 2016, Indianapolis, IN, USA

DOI: http://dx.doi.org/10.1145/2983323.2983344

overall deviation, according to the distances computed using whole time series, hence it cannot pinpoint the exact locations of the local anomalies without further investigation.

As a concrete example, consider the time series data in Figure 1, which contains five time series, two different groups of anomalies and three anomalous subsequences (shown as red/bold). These two groups of anomalies occur in window w_2 (length 153) and w_4 (length 89), respectively. Since the anomalies have different lengths, and there exists more than one anomalous instance of the same type, it is hard for current techniques to correctly identify all of them.

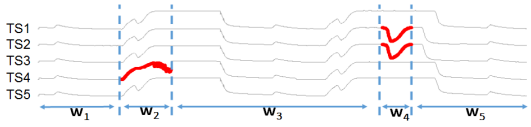

Figure 1: Example of anomalous subsequences.

In Figure 1, the time series are pre-segmented into five windows $(w_1, w_2, \ldots, w_5)$. Once they are segmented, we can apply a clustering-based anomaly detection algorithm to detect anomalous subsequences within each window. The challenge, then, is how to segment the time series properly into windows. We propose a symbolic clustering algorithm that discretizes time series data, and learns common patterns in the data via grammar induction to find the initial segmenting information, as well as initial clusters in each window. Note that we do not assume any information about the potential anomalies in advance. However, in our manufacturing application, data are expected to behave similarly in the same time window. While we allow some warping in the matching of patterns, in application such as ours, a typical sliding-window based approach that compares subsequences from different parts of time series would not be appropriate.

Furthermore, to alleviate users' burden of having to set the parameters, we introduce a self-learning dynamic clustering algorithm which learns cluster boundaries and the number of clusters automatically from the input data.

To summarize, we focus on the problems of discovering global anomalies as well as local anomalies within individual time series. Our algorithm can detect variable-length anomalies without prior knowledge or assumptions on anomalies. Our work has the following significant contributions:

- We propose SLADE-TS (Self-Learning Anomaly DEtection for Time Series), an algorithm that automatically identifies anomalous time series as well as their exact variable-length anomalous subsequences.
- Our algorithm is unsupervised and adaptively learns from data incrementally. It is able to handle streaming time series and identify anomalies in real time.
- By using a clustering-based anomaly detection algorithm and introducing an anomaly scoring strategy, we are able to detect anomalies even if there exists more than one type of normal data or anomaly data.

The rest of the paper is structured as follows. In Sec 2 we discuss background and related work on time series anomaly detection. We introduce our algorithm in Sec 3. In Sec 4, we evaluate our approach on multiple datasets, including real manufacturing data from Intel. Our paper closes with conclusion and discussion for future work in Sec 5.

2. BACKGROUND AND RELATED WORK

To precisely state the problem at hand, and to relate our work to previous research, we will define the key terms used throughout this paper. We begin by defining our data type, time series:

Time series $TS = t_1, \ldots, t_n$ is a set of scalar observations ordered by time. In this paper, time series arrives in an order of $TS_1, TS_2, ...$

Since one of our goals is to find local anomalies, we consider time series subsequences:

Subsequence ss of time series TS is a contiguous sampling $t_i, \ldots, t_{i+m-1}$ of points of length $m < n$, where i is an arbitrary position, such that $1 \leq i \leq n - m + 1$.

Segmented Windows The time series are segmented into a set of non-overlapping windows. The segmentation is data-adaptive rather than based on pre-defined cut-points. Subsequences from the same Segmented Window have the same length. **Anomaly Windows** are Segmented Windows that contain anomalies.

Figure 1 shows examples of *Segmented Windows* and *Anomaly Windows*. Three anomalous subsequences are detected. Two *Anomaly Windows* (w_2, w_4) are identified from five *Segmented Windows* ($w_1 w_2, \ldots, w_5$).

Similarity Thresholds $T = \{T_1, T_2, ..., T_N\}$ is a collection of N scalar values for N Segmented Windows. The algorithm learns the value of N, as well as each scalar value T_w, which is used to determine the assignment of subsequences to clusters within a window w. Details about clustering using T will be discussed in Section 3.2. **Normalized Similarity Thresholds** $T_{norm} = \{T_{norm\_1}, ..., T_{norm\_N}\}$ is used for normalized subsequences.

2.1 Related Work

Many time series anomaly detection algorithms have been proposed in recently years [5, 9, 15, 28, 25], focusing on different types of anomalies, including point outliers, change points, anomalous time series, or structural (subsequence) anomalies. To identify anomalous time series, some works extract features from the original time series and perform anomaly detection in the transformed feature space. The authors in [8] extract commonly known representative features of time series, then perform anomaly detection on the first two principal components. In previous work [26], we extract features that capture the characteristics of medical alarms to detect anomalies in medical time series.

Another type of anomaly detection algorithms use clustering techniques to detect abnormal behaviors [3, 19, 24]. Clustering-based algorithms often have difficulty detecting anomalous time series that contains subtle anomalies, since the algorithms use global measures to determine cluster membership, and deviation from subtle, localized anomalies can be dwarfed by global noises. Even in the case when the algorithms are able to identify an anomalous time series, they cannot locate the exact sections in the time series that contribute most for the abnormality. In order to pinpoint the exact anomalous locations in a time series, analysts may need to visually inspect the detected time series to find the anomalous parts. As will be shown later, our algorithm not only detects the anomalous time series, but also highlights the sections that *explain* the abnormality. Our algorithm outputs the top K (K is determined by the algorithm automatically) subsequences that exhibit abnormal behaviors.

Contrary to anomaly detection based on global measures on whole time series, window-based techniques can achieve better localization of anomalies. Time series discord discovery algorithms [13, 14] are a type of window-based algo-

rithms that capture the most unusual subsequence(s) within a long time series. While discord discovery has been shown to achieve better performance than other existing techniques [4], to identify a discord, the user must specify its length. There are two limitations with imposing this requirement in real-world applications. First, the user may not know the exact length, or even the best range of lengths for the potential discords in advance. Second, there might exist multiple discords of different lengths in a time series [21]. It would be extremely cost prohibitive to consider all window lengths. We proposed a solution to mitigate this problem in [21], but the algorithm still requires an initial seed length as input.

Another limitation with discord-based approaches is that the definition of discord is tied to the number of nearest neighbors, i.e. the discord is the object with the largest distance to its k^{th} nearest neighbor. Imagine the simplest case where $k = 1$. If an anomaly occurs twice or more, then each anomalous instance would have a match that is very similar, in which case the algorithm would fail to detect them as anomalies. Having a fixed value k is not ideal because we simply do not know how many times an anomaly would occur. In this paper, we address the aforementioned problems. To the best of our knowledge, our work is the first to solve the problem without any prior knowledge.

3. OUR APPROACH

We start by first discussing the overview of our proposed algorithm.

3.1 Overview of SLADE-TS algorithm

Our algorithm consists of four parts: (a) Adaptive segmentation by symbolic clustering; (b) Dynamic clustering; (c) Parameter self-tuning; and (d) Anomaly Scoring. The work flow is shown in Figure 2.

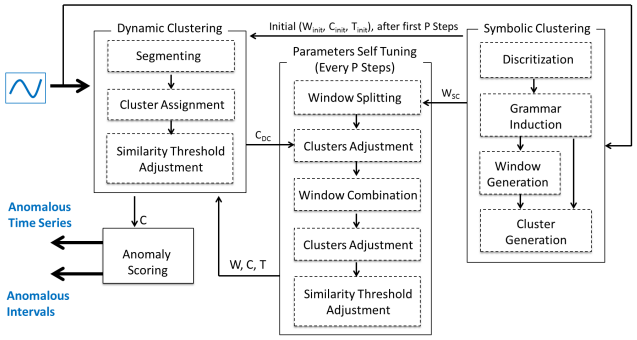

Figure 2: The work flow of SLADE-TS algorithm.

We assume that the data come in a streaming fashion. We use a small amount of time series to first segment the data and obtain the positions of the *Segmented Windows* (i.e. via Symbolic Clustering, Sec 3.3). When a new time series arrives[2], we segment the time series into subsequences according to the learned *Segmented Windows* positions, and perform dynamic clustering to assign every subsequence to one of the clusters in the corresponding window (Sec 3.2). The Anomaly Scoring algorithm assigns anomaly scores to subsequences based on the clustering results (Sec 3.5). By aggregating the anomaly scores of the subsequences, we can identify both global and local anomalies.

SLADE-TS is a self-learning algorithm. When time series TS_i is processed, it updates important information to be used in dynamic clustering subsequently, including (i) *Segmented Windows* W, for cutting time series into subsequences; (ii) Clusters C; and (iii) *Similarity Thresholds* T. We name them WCT. We use the updated WCT to segment and cluster time series TS_{i+1}. Initial values of WCT are determined by the symbolic clustering algorithm using the first p time series. In our experiment, we set p as 5.

3.2 Dynamic Clustering

Dynamic clustering was first introduced in [23]. The authors in [1] then used this idea to detect outliers in data streams. We use dynamic clustering to cluster subsequences extracted from different time series. Note that this is different from the *time series subsequence clustering* problem studied in [12], in which case subsequences are extracted from the same time series. Also different from the problem in [1], we do not have the instances to be clustered. Specifically, before applying dynamic clustering, we need to cut time series into non-overlapping windows. We can then cluster subsequences that belong in the same window. The segmentation of time series into different windows is an important step for our algorithm. If a window is too large, we might miss some subtle anomalies. If a window is too small, true anomalies could be dominated by noise, or the detected anomalies may be only a subsection of longer anomalies.

In the next two sections, we will formalize an algorithm to determine the initial parameters WCT for our dynamic clustering algorithm, and an algorithm that self-adjusts windows dynamically. We defer the detailed discussion on segmentation until then. To demonstrate the idea of dynamic clustering, we assume for now that the time series is already segmented.

The algorithm in [1] starts with an empty set of clusters. When a new instance arrives, unit cluster that contains single instance is created. Once a maximum number cl of such clusters have been created, the algorithm begins the process of online cluster maintenance. In this work, we do not start from an empty set, as initial clusters C_{init} and *Similarity Thresholds* T_{init} will have been generated by the symbolic clustering algorithm along with the segmentation. Therefore, we start the process of online cluster maintenance with the initial information when a new time series arrives. For now, we assume that we already have C_{init} and T_{init}.

The dynamic clustering algorithm is outlined in Algorithm 1. There are four input parameters: time series TS, *Segmented Windows* W, current clusters C and *Similarity Threshold* T. We have defined TS, W and T in Section 2. Clusters C contain existing clusters learned so far for each window. These parameters are initialized by the symbolic clustering algorithm and updated in a self-learning process. The dynamic clustering algorithm starts by extracting subsequences from the new time series instance based on the *Segmented Windows* W generated by the symbolic clustering algorithm, Line 2. In the nested for-loop in Lines 3-15, we assign the subsequence ss_w from window w of TS to a cluster, by first trying to find an existing cluster close to it (Lines 5-10). If the closest existing cluster is close enough (i.e. within similarity threshold T_w for window w, we assign ss_w to it and update T_w (Lines 14-15). Otherwise, we create

[2]We use the whole time series to demonstrate the idea, but our approach also works in the scenario when points of time series come in a streaming fashion.

a new cluster for ss_w (Line 12). We describe the details in the following subsections.

Algorithm 1 Dynamic clustering algorithm

Input: Time series TS, window information W, current clusters C, similarity threshold for every window T.
Output: New clusters C_{DC}, new threshold T_{DC}

```
1: function DynamicCluster(TS, W, C, T)
2:     S ← CutTS(TS, W)  ▷ S is a collection of subsequences
3:     for each w in W do
4:         d_min ← INF
5:         for each C_i in C_w do          ▷ C_w is all clusters in w
6:             d_i ← D(ss_w, c_i)                          ▷ ss_w ∈ S
7:             nd ← D_norm(ss_w, c_i)
8:             if (d_i ≤ T_w)&(nd ≤ T_norm_w)&(d_i < d_min) then
9:                 d_min ← d_i
10:                C_o ← C_i       ▷ C_o is the closest cluster to ss_w
11:        if d_min == INF then          ▷ Create a new cluster
12:            C_w.add(NewCluster(ss_w))
13:        else                                        ▷ Assign a cluster
14:            AssignCluster(ss_w, C_o)
15:            UpdateSimilarityThreshold(d_min)
16:    return (C_DC, T_DC)
```

3.2.1 Finding Closest Cluster

We define ss_w as the subsequence from the new time series in window w. The distance value $D(ss_w, c_i)$ is computed over all clusters in window w, where c_i is the centroid of cluster C_i. We also compute the distance between normalized ss_w and c_i, in order to capture the shape difference. If $D_{norm}(ss_w, c_o) \leq T_{norm\_w}$, where T_{norm_w} is the similarity threshold for the normalized subsequences in window w, and $D(ss_w, c_o) < D(ss_w, c_j), \forall j$, cluster C_o is chosen as the closest cluster for subsequence ss_w.

The normalized distance gives more weight to shape difference while un-normalized distance focuses on the value difference. While in most time series applications, normalization is important as we want to capture shape (dis)similarity rather than actual values, in our application we want to detect *both* shape and value differences, so we use both the un-normalized distance and normalized distance. We note that this can be easily changed depending on the application. To find the closest cluster for ss_w, we need to consider three scenarios:

(i) $\forall C_j$ in w, $D(ss_w, c_j) > T_w$. In this case, there is no cluster close to ss_w.

(ii) $\exists C_i$ in w, $D(ss_w, c_i) \leq T_w$, but $D_{norm}(ss_w, c_i) > T_{norm\_w}$. In this case, although cluster C_i is close to ss_w using un-normalized data, the distance for normalized data > threshold. We still do not assign ss_w to it.

(iii) $\exists C_o$ in w, $D(ss_w, c_o) \leq T_w$, $D_{norm}(ss_w, c_o) \leq T_{norm\_w}$ and $D(ss_w, c_o) < D(ss_w, c_j), \forall C_j$ in w. In this case, the closest cluster C_o is found.

In this paper, we define C_o found in case (iii) as the closest cluster. If no such qualified cluster found (situations (i) and (ii)), we conclude there is no suitable cluster for ss_w.

3.2.2 Assigning Cluster

If there is no suitable cluster found for ss_w, our algorithm creates a new cluster that contains the solitary instance ss_w. The new cluster is a potential true anomaly or the beginning of a new normal behavior. Further understanding of it may be obtained as more data are collected.

Otherwise, if the closest cluster C_o is found, we assign ss_w to C_o. Meanwhile, the cluster centroid of C_o should be updated by Equation (1), where CN is the number of instances in cluster C_o before the new instance ss_w is added.

As the centroid of C_o is updated, the distance between the old instances in C_o and the new cluster centroid should be updated as well. However, if we call the distance updating method every time when a new instance is added into a cluster, it will be inefficient and unnecessary. Therefore, we perform batch update when p new time series have arrived. We will describe the details in section 3.4.2.

$$c_{o\_new} = (c_o * CN + ss_w) \,/\, (CN + 1) \tag{1}$$

3.2.3 Updating Similarity Threshold

Similarity threshold T_w is used to determine if there is a suitable cluster for ss_w. Each window has a different threshold. The intuition behind this is that different sections of the time series may have different variance tolerance. *Similarity Thresholds* are dynamically determined based on the distance values of past assignments. The mean μ_t and standard deviation σ_t of the distance values to the assigned cluster centroid in the history of assignments are computed at the time of arrival of the t^{th} instance. The threshold is set as in Equation (2), which corresponds to the three sigma rule [20] with a normal distribution assumption.

$$T = \mu + 3 * \sigma, (\mu : mean, \sigma : standard\ deviation) \tag{2}$$

Normalized Similarity Thresholds for normalized subsequences are computed separately using the same equation.

3.3 Symbolic Clustering Algorithm

Dynamic clustering needs WCT as input for further processing. If we do not have this information in advance, or if the parameters are chosen poorly, the clustering results may be undesirable. Therefore, we introduce a symbolic clustering algorithm that provides a heuristic to determine the initial WCT without prior knowledge of the data. Our symbolic clustering algorithm first discretizes continuous time series values into symbolic representation. It then identifies repeated patterns by learning a grammar from the data, in order to approximate the best segmentation, as well as the best clusters and threshold value per window.

In our previous work [16, 22], we proposed a grammar-based algorithm for efficient discovery of variable-length frequent patterns. In this work, we introduce a grammar-based segmentation and clustering algorithm. Our approach is built on a two phase process: (i) context-free grammar induction; (ii) clustering using grammar rules. The algorithm is outlined in Algorithm 2.

The algorithm takes a time series TS and the grammar learned from the past time series as input. Lines 2-5 performs initial segmentation of time series. Details are discussed in the following subsections. In the nested for-loop in Lines 6-13, we cluster subsequences in each segmented window w using the grammar information (details are discussed in 3.3.4). We compute the distances between subsequences and their respective centroids in Line 12. Similarity Threshold T_w for window w is computed in Line 13.

3.3.1 Discretization

Before applying our grammar induction algorithm, we prepare the data by converting each time series into a sequence

Algorithm 2 Symbolic clustering algorithm

Input: Time series TS, grammar G (empty if it is the first TS).
Output: Window information W, current clusters C, similarity threshold for every window T, updated grammar G'

function SymbolicCluster(TS, G)
  $dTS \leftarrow$ Discretize(TS) ▷ Discretize time series TS
  $G' \leftarrow$ GrammarInduction(dTS, G)
  $FM \leftarrow$ ComputeFreq(G') ▷ Frequency Matrix FM
  $W \leftarrow$ CutWindow(FM) ▷ Cut FM at changing points
  for each w in W **do**
    for each g in G' **do**
      if g covers the same location of w **then**
        $C_i \leftarrow$ time series covered by g
        $C_w.add(C_i)$ ▷ C_i becomes one cluster in w
    $C.add(C_w)$
    $Des_w \leftarrow$ distances between subsequences and centroid
    $T_w = \mu_w + 3 * \sigma_w$
  return (W, C, T, G')

of tokens, where each token represents a discretized version of a segment from the time series. In this work, we modify the discretization algorithm slightly: instead of using Symbolic Aggregate approXimation (SAX) [17], we use fixed-height discretization because we want to detect both shape and value differences, and SAX only applies to normalized time series.

For each segment, we first reduce the dimensionality with Piecewise Aggregate Approximation (PAA) [11], as shown in Figure 3. More specifically, PAA divides time series into s equal-sized partitions and computes a mean value for each partition. It then maps these values to symbols according to a pre-defined set of breakpoints that divide the range of values into α equal-height regions, where α is the alphabet size. The parameters of discretization, such as the segment length or PAA size, are set with default values in experiment. For example, the subsequence length could be set as 5% of the segment length.

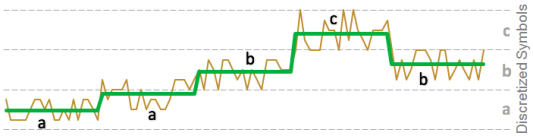

Figure 3: Time series TS (gold), its PAA representation (green/bold), and the converted discretized word. The discretized word is *aabcb*

3.3.2 Constrained Grammar Induction

To describe grammar induction in detail, consider the following sequence of *words*. Each word is considered an atomic unit, or a *terminal* in the sequence: $S_1 =$ *aba bac cab acc bac cab*. We feed the string S_1 into a context-free grammar induction algorithm such as Sequitur, a string compression algorithm that infers a context-free grammar in linear time and space [18]. When applied to a sequence of words, Sequitur treats each word as a token and compresses the sequence by learning a context-free grammar from the input. The algorithm recursively reduces all *digrams* (consecutive pairs of tokens) occurring more than once in the input string to a single new non-terminal symbol.

The following table shows the grammar induced by Sequitur from S_1. The algorithm reduces the length of the input string by creating a grammar whose rules are encoded by *non-terminal* $R1$, which reveals repeated patterns in the input. $R1$ describes a simplified version of the repeated pattern [*bac cab*].

Grammar Rule	Expanded Grammar Rule
R0 → aba R1 acc R1	*aba bac cab acc bac cab*
R1 → bac cab	*bac cab*

Different from the original grammar induction method as shown above, in this work we modify the algorithm so that it is position-aware. That is, we consider the subsequence position information when we generate grammar rules. Doing so, S_1 becomes $S_1 = aba_1\ bac_{151}\ cab_{301}\ acc_{451}\ bac_{601}\ cab_{751}$. The subscript of each word denotes the starting position of the subsequence in the original time series. In this example, the original time series is divided into six non-overlapping segments, each with length 150. Each segment is then discretized, with parameters $s = 3$ (i.e. 3 PAA partitions since each word has three letters), $\alpha = 3$ (cardinality of alphabet is 3: 'a', 'b', 'c'). To determine a match between two digrams, in addition to having identical words, they also must occur at the same position of their respective time series. In other words, we only compare subsequences from different time series if they start at the same positions (with some modification, we could also allow some "warping" between the tokens). In this example, the two occurrences of "bac" are no longer a match because they have different location subscripts, denoting that these patterns are associated with different time intervals. Now suppose we have another sequence from a different time series, $S_2 = aba_1\ bac_{151}\ cab_{301}\ aca_{451}\ cac_{601}\ aaa_{751}$. Between S_1 and S_2, the tokens that can be merged are aba_1 bac_{151} cab_{301} with grammar rules $R1 \rightarrow R2\ cab_{301}$ and $R2 \rightarrow \text{aba}_1\ \text{bac}_{151}$, since they match on both the words and locations. The expectation is that time series data should have similar value and shape at the same time point. This is important for some real-world applications. For example, in the process of reflow soldering, the temperature is critical to the quality of solder joint[3]. The temperature profile, which shows the transient behavior of the solder reflow oven, should be the same for each item. Otherwise, it could yield a manufacturing defect. Therefore, a subsequence should only be a candidate anomaly if it is different from other subsequences at the same position (e.g. same point in the reflow soldering process).

3.3.3 Generating Segmented Windows

In most cases, anomalies do not last for the entire time series. If we try to detect anomalies using the whole time series, we may miss subtle anomalies, and we would not be able to identify the exact local anomalies.

Generating *Segmented Windows* is difficult because it requires information about potential anomalies. This is a "deadlock" problem here: we need proper segmentation to detect anomalies, while finding anomalies is the best way to segmenting the windows. We will show how we adapt grammar rules to solve this paradox.

We use frequency coverage of time series points to generate *Segmented Windows*. We define frequency coverage as the number of time series covered by the same grammar rule. With each grammar rule we also store the locations

[3]Reflow soldering: https://en.wikipedia.org/wiki/Reflow_soldering

of subsequences that are covered by it, as well as the indices of their respective time series. We can then compute the frequency coverage of each point in a time series. Suppose S_1 and S_2 from the grammar induction example are the discretized string of TS_1 and TS_2 respectively. In this example, the frequency coverage for data points in the range 1-450 of TS_1 and TS_2 is 2 because R_1 covers two time series. It means that these two time series share the same pattern at that time interval. The frequency coverage of other data points will be set to 1 since no grammar rule covers them.

Consider another example shown in Figure 4. The dataset contains 11 time series $(TS_1, TS_2, \ldots, TS_{11})$, each of which has 35 time points. The first 5 data points in every time series are in the rule that covers all the eleven time series. So the frequency of the first 5 time points for all time series should be 11. After computing the frequency for every time point, we can get a frequency matrix that contains frequency for all time points in all time series, as shown in Figure 4. We set the cut points at the locations where the frequency changes. The intuition behind this is that the changing of frequency suggests the changing of pattern groups. If frequency changes at point A, the group information on the left and right side of point A are different. If a time series starts to form a new group at point A, it possibly starts to show anomalous behavior. In contrast, if it starts to show the same behavior as other time series from point B, then the anomalous behavior only occurs between point A and point B. Thus, the changing points should be the start or end location of anomalous behavior. The collection of all cut points define the *Segmented Windows* W.

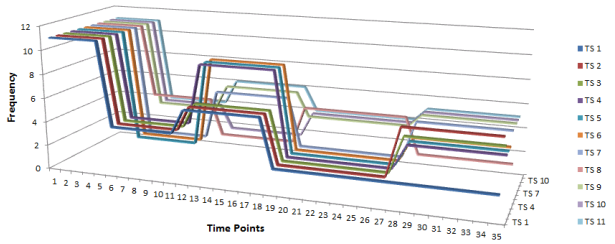

Figure 4: Frequency coverage of each time point for $TS_1 - TS_{11}$. We generate Segmented Windows by cutting at frequency-change points. In this example, they will be {1-5,6-11,12-18,19-27,28-35}

3.3.4 Symbolic Clustering in Each Window

After segmenting the time series, we generate initial clusters using grammar rules. In each window, we search for rules that overlap the same window. We assign subsequences into the same cluster if they are covered by same rule. It is possible that a subsequence is (partially or wholly) covered by more than one grammar rule. We merge clusters if they share a large fraction of the same instances. Otherwise, we break tie by assigning a subsequence to the closer cluster.

Because the grammar rules may not cover all subsequences, the clusters generated by symbolic clustering algorithm may not contain all subsequences from each *Segmented Window*. In addition, since the symbolic cluster algorithm works in symbolic space, there may be some information loss due to the discretization. To determine cluster membership for the missed subsequences in window w, we select subsequences that do not get assigned to any cluster, and call the cluster adjustment function described in Section 3.4.2 for furthur processing. To refine clusters for a window w, we do the same by selecting subsequences whose distances to their respective centroids are greater than T_w, and calling the cluster adjustment function.

After we obtain clusters for all *Segmented Windows* by symbolic clustering algorithm, we generate cluster centroid for each cluster and compute the distances from the cluster centroid to every instance belonging to that cluster. Since we have the distances of all subsequences to their respective centroids, we can compute the initial *Similarity Thresholds* by calling similarity threshold updating function introduced in Section 3.2.3.

3.4 Parameters Self Tuning

Our algorithm is a self-learning algorithm, that is, the accuracy gets better when more data arrive. The information we get from past data, such as *Segmented Windows*, may only be good for past data but not current data. Thus, we need to update WCT as more data arrive. In our algorithm, when every p new time series arrive, we call symbolic clustering algorithm to get new *Segmented Windows*. However, instead of using the new values directly to replace the old ones, as shown in Figure 2, we adjust WCT with the following steps: (i) Window Splitting; (ii) Clusters Adjustment; (iii) Window Combination; (iv) Clusters Adjustment; (v) Similarity Threshold Adjustment.

The details of step (ii) and (iv) are introduced in section 3.4.2, while step (v) has been discussed in section 3.2.3. We discuss the detail of step(i) and (iii) here.

3.4.1 Window Optimization

The initial *Segmented Windows* are generated by the symbolic clustering algorithm. However, the window information may not be perfect for anomaly detection because: (a) we may lose information by discretization. (b) The grammar induction algorithm in symbolic space requires exact match, which may cause incorrect changes of frequency coverage. (c) The frequency coverage that is used to generate *Segmented Windows* reflects the grouping differences between different sections of time series, but it is not the exact cluster information. In order to get more accurate *Segmented Windows*, we propose two techniques here: window *combination* and window *splitting*.

Window Combination.

The constrained grammar induction method uses exact match to find frequent patterns. This strategy may cut a pair of long similar patterns into two pairs of shorter patterns if there are few mismatches in the middle of the long pattern. Therefore, the generated *Segmented Windows* may be smaller than they should be. In addition, the discretization step may produce different symbols for similar values if these values happen to reside on opposite sides of a breakpoint (See Figure 3. The second 'a' is very close to the breakpoint. A value slightly larger than it would land on the other side of the line, and be assigned to symbol 'b'). Such boundary problem may also cause the breakage of a long pattern. To obtain appropriate size of windows, we introduce a window combination technique here.

The idea of window combination strategy is straightforward. We combine two consecutive windows if the clustering results in those two windows are similar. The intuition is that, same clustering results in two consecutive windows suggest that the anomalous behaviors, if any, remain the

same from one window to the next. In this case, we should extend the length of the anomalies by merging windows.

In order to define similar clusters, we apply Jaccard Coefficient [10] to the comparison of clustering results. For two consecutive windows, we use the clustering result from the left one as the ground truth. Then we compute the Jaccard distance between the clusters from the two consecutive windows. If the Jaccard distance is large enough, say 80%, the clustering results from the two windows are similar, in which case we should merge the windows into a larger one.

Window Splitting.

Due to the online property of our algorithm, the current data at some time points may not contain information for future data. For example, the initial p time series that we use to set the parameters could all be normal and similar to each other, in which case we may get a large window with length as long as the whole time series. However, at some time in the future, we may encounter some anomalies in the new data. In this case, we should split the window into smaller ones. The discretization process may also cause window sizes to be larger than they should be since each symbol is an approximation of a range of values in the original time series. To mitigate these problems, we introduce a window splitting technique.

A brute-force way to accomplish this goal is to use a relatively small sliding window to cut the large window into smaller ones and apply clustering algorithm for each small window. We can then call the window merge method introduced previously to get the correct windows. However, this method is inefficient and requires parameters such as the sliding window size.

As mentioned before, our algorithm calls the symbolic clustering algorithm with every p new time series arrived, and finds *Segmented Windows* efficiently. So each time a new set of *Segmented Windows* is generated, we can compare them against the current set of windows. We split an old window into smaller ones if there are several new windows that overlap the old window.

3.4.2 Cluster Adjustment

Since clustering is done incrementally, the cluster centroid may move considerably after processing many instances. This may cause two potential problems. (i) Some instances in the cluster may no longer belong to the cluster if their distances to the new centroid are now greater than the threshold. (ii) Clusters may become very close to each other that we should merge them. To solve these two problems, our algorithm updates the clusters when instances are added to clusters.

To determine whether any clusters should be merged, our algorithm computes pairwise distances between centroids. If a distance is smaller than the similarity threshold, we merge the clusters. The centroid of the new cluster should also be updated accordingly.

To determine whether any cluster should be split, when a new instance ss_w is added into a cluster and the cluster centroid has changed, we compute the distance between the new cluster centroid to every instance in the cluster. If the distance between instance ss_{w_i} and its centroid is greater than the similarity threshold, our algorithm removes ss_{w_i} from the cluster, and calls the dynamic clustering algorithm that we introduced in Section 3.2 to re-assign it to a new cluster. At the same time, since ss_w is removed from the old cluster, we need to update the centroid of the old cluster.

In practice, we do not need to update distances and perform cluster adjustment every time an instance is added to a cluster, because the change of centroid is likely very insignificant when the number of new instances added is much smaller than the number of instances in the cluster. Thus, we perform batch updates after we have processed p new time series, where p is an integer value between one and the number of total time series.

3.5 Anomaly Scoring

After we have the clustering results for each *Segmented Window*, we compute anomaly scores for all windows, and all time series. One way to obtain an anomaly score for a time series is to take the average of anomaly scores from all windows. This method does not work well if most windows do not contain anomalies, since the scores from unrelated subsequences will have negative impact on the final aggregated score. Another approach is to select the top h windows with the largest anomaly scores, and take the average of these h scores. For this method, the parameter h needs to be set properly. The third approach is to set anomaly score as the number of times the running average of the anomaly scores exceeds a threshold. However, this method does not consider the magnitude of the anomaly scores since every score has the same weight if it exceeds the threshold. In this paper, we use a fourth approach, which sets the overall anomaly score as the average of the scores from the anomalous subsequences whose scores exceed a threshold T_{as}. T_{as} is also computed by equation (2), here μ and σ are the average and standard deviation of anomaly scores of all subsequences.

Within each window, the anomaly score for each instance is computed based on the clustering result. We can assign score to an instance based on the number of instances in the same cluster. The ones with larger numbers have smaller anomaly scores. We could also compute anomaly score based on density: lower density means higher anomaly score. In our experiments, we modified the scoring method from [7], which considers both the size of cluster and the distance to cluster centroid, as shown in equation (3):

$$AS = D(ss_w, c_i) * (1 - \frac{|C_i|}{|C_{big}|}). \quad (3)$$

where c_i is the centroid of cluster C_i that contains instance ss_w, if cluster C_i contains more than $T_{support}$ instances. Otherwise, c_i is the centroid of cluster i closest to ss_w that has enough number of instances. C_{big} is the cluster that has the largest number of instances. $|C_i|$ and $|C_{big}|$ are the numbers of instances in clusters C_i and C_{big}, respectively. We define $T_{support}$ as $|C_{big}|/2$.

4. EXPERIMENTAL EVALUATION

The goal of our experiments is to show that our algorithm not only detects the anomalous time series but also identifies where the anomalous subsequences are. We also show that our algorithm is able to find the correct anomalies without setting parameters like the number of anomalies or the length of anomalies. The parameters we need are only for discretization step, which do not require prior knowledge of dataset. In addition, we demonstrate the utility of our approach on real-world manufacturing data from Intel.

We compare our algorithm with window-based discord techniques HOTSAX [13] (other discord algorithms such as RRA [21] generates similar results). We also compare with clustering-based anomaly detection algorithm.

4.1 Dutch Power Consumption Data

The power data contains 52 weeks of power demand by a research facility. The typical weekly power consumption is shown in Figure 5. The original power demand data is a single time series with the length of one year. In this experiment, since we want to detect anomalies within a week, we pre-process the data by cutting the time series into 52 time series, each one contains one week of data from Monday to Sunday.

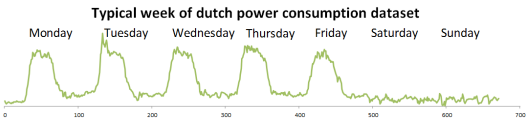

Figure 5: Typical power consumption in one week.

Our SLADE-TS algorithm detected eight anomalous time series with the discretization parameters: initial segment length = 20, $\alpha = 5$, PAA size = 3. The results are shown in the top of Figure 6. The lengths and locations of anomalous subsequences for each anomalous time series are also identified. Two examples of the detected anomalous time series and their corresponding anomalous subsequences, highlighted in grey, are shown in the bottom of Figure 6. In the left plot, SLADE-TS detected an anomalous pattern on Monday (Liberation Day) and Thursday (Ascension Day). The detected anomalous subsequence in the right plot can be interpreted as the national holiday of Whit Monday.

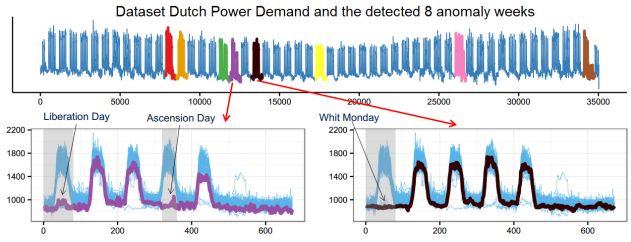

Figure 6: Top: 8 anomalous weeks detected by SLADE-TS. Bottom: zoomed-in plot of 2 anomalous weeks and the corresponding events. Purple and black time series are detected anomalous weeks while blue time series are other weeks.

Figure 7 shows the comparison of the anomalies detected by SLADE-TS and discord algorithm (HOTSAX) on the same time series. The discord algorithm requires two parameters as input: length of discords and numbers of discords. In order to compare with them, we set the discord length close to the length of one day, which is 70 in this data set. The number of discords is set to 8, which is the actual number of anomalous time series. The number of nearest neighbors are set to 1. For a fair comparison, we modified the discord algorithm to only compare subsequences that are at the same locations, e.g. Monday to Monday, Tuesday to Tuesday, etc. The discord algorithm detected 5 correct anomalous time series, two of them are shown here for comparison. For the same time series, SLADE-TS and discord algorithm found different anomalous locations. As shown in Figure 7, SLADE-TS found anomalies at locations matching *Good Friday*, *Christmas Day* and *Dat after Christmas*, while discord algorithm found anomalies at the night before *Good Friday*, and *Christmas Eve*. Note this is a relatively "simple" problem for discord discovery since the length of the anomalies can be pre-determined (i.e. one day).

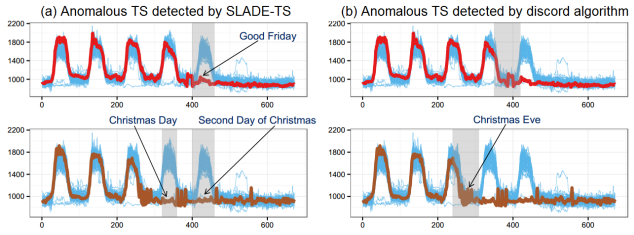

Figure 7: Detected anomalous subsequences from the same anomalous weeks (top: 12^{th} week, bottom: 51^{th} week) by SLADE-TS and discord algorithm.

We also compare our algorithm with clustering-based anomaly detection approach. We choose a recently proposed clustering algorithm [2] based on dynamic time warping (DTW), which outperforms existing clustering algorithms. However, it is not an anomaly detection algorithm itself, so we added an extra step to detect anomalies from the clustering result. This algorithm requires parameter setting of the number of clusters. From the experiments, this clustering algorithm achieves its best performance when the number of cluster is set to 7, so this is what we report. In total, the DTW-based clustering algorithm found eight anomalous weeks, seven of which are the same as anomalous weeks detected by SLADE-TS. The different detected anomalies are shown in Figure 8.

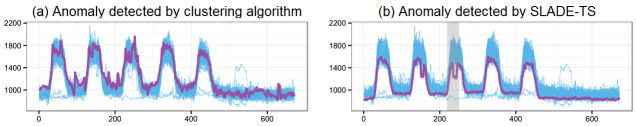

Figure 8: Different anomalous weeks detected by clustering algorithm and SLADE-TS.

The clustering-based anomaly detection algorithm uses the whole time series for clustering, so it could miss anomalies that only occur in short sub-sections. On the other hand, since the time series are noisy, the small deviations could add up over time, resulting in large distances. As a result, the time series could be misclassified as anomaly. For example, in Figure 8, the detected anomalous time series in (a) was detected as anomaly but there is no significant difference. The anomalous time series in (b) was missed by the DTW-based clustering algorithm even though there is an obviously unusual pattern on Wednesday.

Besides the problems of false anomalies and parameter setting, the whole time series clustering based algorithm cannot detect where the anomalies are within time series while SLADE-TS finds the information automatically. Knowing exactly where the anomalies occur is highly valuable in pinpointing problems and looking for solutions.

4.2 Intel Production Dataset 1

This dataset comprises of normalized temperature profiles observed by thermocouples in an epoxy cure oven. The data has been normalized to rescale the actual temperature/time for proprietary reasons - but it retains the overall transient behaviors. Typical excursion prevention (EP) applications would involve an engineer using his intuition and judgement

to setup time windows and various metrics (maximum value, mean, slope etc.) to capture the behavior of the time series within each window in order to ensure the process stays on target and the dynamic response of the oven is captured.

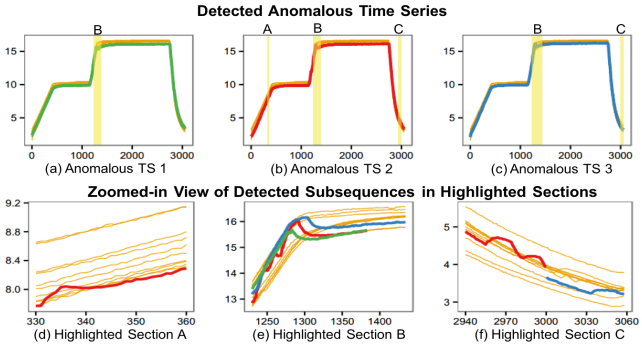

Figure 9: Top: anomalous time series detected by SLADE-TS algorithm. Highlighted sections indicate locations of anomalous subsequences. Bottom: zoomed-in view of anomalous subsequences.

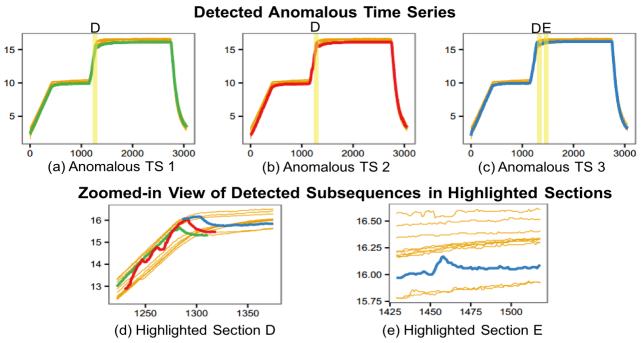

Figure 10: Anomalies detected by discord alg.

Anomalies detected by SLADE-TS and discord algorithm are shown in Figure 9 and 10 respectively. Our algorithm found 3 anomalous time series (shown as top 3 plots of Figure 9, detected anomalous time series are shown in bold, the highlighted sections are the corresponding anomalous subsequences) and 6 anomalous subsequences (bottom 3 plots are zoomed-in view of detected anomalous subsequences. Their colors match the colors of their time series). The discretization parameters we used are: initial segment length = 20, $\alpha = 6$, PAA size = 3. To have discord algorithm also detect 3 anaomalous time series and find as many anomalous subsequences as possible, we set the number of discords as 4. With larger number of discords, discord algorithm returns incorrect anomalous time series in addition to the 3 correct ones, while with smaller number of discord, it finds fewer anomalous subsequences. Also, from prior knowledge of data set, we set discord length as 90 to achieve its optimal performance.

Comparing the results, the discord algorithm detected anomalies with fixed length which is given by the user. SLADE-TS automatically detected anomalies with various lengths without prior knowledge. Besides the variable lengths of anomalous subsequences, the number of anomalies is also automatically detected, as shown in Figure 9.

Both algorithm detected the same anomalous time series, but some corresponding anomalous subsequences are different. The anomalous subsequences shown in the zoomed-in view of Figure 9 (e) and Figure 10 (d) are similar. The different anomalies detected are shown in Figure 10 (e) (by discord algorithm), and Figure 9 (d) and (f) (by SLADE-TS). It is hard to say which results are better without knowing the ground truth. However, it is clear that the discord algorithm needs proper input parameters to detect the correct anomalies, and some of these parameters are non-trivial to set, e.g. the number of nearest neighbors. We also tried other parameter settings for the discord algorithm, but the current setting provides the best results. SLADE-TS, on the other hand, learns these critical parameters automatically.

4.3 Intel Production Dataset 2

This data set comprises of normalized temperature data off the monitor thermocouples in a 20 zone solder reflow oven. In order to ensure a good solder joint it is imperative that the oven behave in a repeatable fashion. As trays move through each oven zone the temperature in that zone is perturbed and the oven PID controller attempts to compensate. By observing the transient behavior of this response one can detect impeding issues that will impact joint quality. Specifically, the temperature profile has to ensure that the flux is correctly activated without oxidizing the paste and the solder is allowed to melt just enough to attach itself to the pads without damaging components on the package.

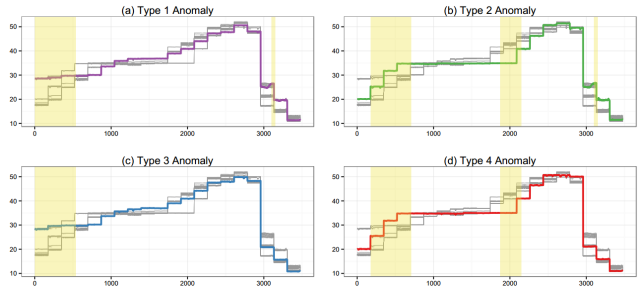

Figure 11: 4 types of anomalies detected by SLADE-TS. Highlighted sections are anomalous subsequences. Colored bold time series are examples of anomalous traces. Time series in gray are other traces (include normal and anomalous traces).

There are 286 time series in production dataset 2, and 41 of them are anomalies. Each time series is around 3500 in length. The discretization parameters are: initial segment length = 50, $\alpha = 6$, PAA size = 3. We correctly detected all 41 anomalies. We also detected four different types of anomalies based on their anomaly scores and anomalous locations. As shown in Figure 11, our algorithm detected 14 anomalous time series of type 1 (Figure 11 (a)), 9 of type 2 (Figure 11 (b)), 10 of type 3 (Figure 11 (c)) , and 8 of type 4 (Figure 11 (d)). By further separating the detected anomalies into different types, our algorithm can potentially help domain experts better understand the anomalies.

Besides the anomalous traces, our algorithm also found the anomalous sections within them, shown as the highlighted parts in Figure 11. For type 1 anomaly, the anomaly locations are 1-535, 3103-3153. As we can see, even for the normal time series, there are some variations. However, these variations are within the normal range. One advantage of our algorithm is that we do not have to set the normal range; the algorithm will find the range automatically. For

example, in Figure 11 (a), even in the un-highlighted parts there are still variations between the red anomalous time series and normal time series. Our algorithm is able to correctly differentiates them from the true anomalies.

If we use a clustering algorithm directly on the whole time series, we can also find these 41 anomalies. However, it requires proper setting of parameters, such as the number of clusters or the density threshold of each cluster. With our algorithm, this extra step can be completely omitted. In addition, clustering algorithm is not able to provide the location information of the anomalies, while our algorithm can accurately identify the exact time the anomalies occur and their duration (length). When in real production process, our algorithm can accurately detect when an abnormal behavior happens and how long it lasts. Such additional information will be highly valuable in identifying the production problems and looking for solutions.

5. CONCLUSION AND FUTURE WORK

In this paper, we introduce a self-learning online anomaly detection algorithm on time series data. Our SLADE-TS algorithm detect anomalous time series automatically without any prior knowledge of the data. In addition, SLADE-TS can pinpoint the locations of anomalous subsequences within a time series. SLADE-TS finds the lengths and the number of anomalous subsequences automatically.

One advantage of our algorithm is it requires minimal configuration, if not zero. However, for different datasets, the notion of anomaly may differ. In future work, we will extend our algorithm to use the feedback from users to customize the algorithm for different datasets.

The distance measure used in this paper is Euclidean Distance (ED). ED is fast, but it requires the time series to be well aligned. For future work, we will incorporate other distance measure such as dynamic time warping to make our algorithm more robust.

6. REFERENCES

[1] C. C. Aggarwal and S. Y. Philip. On clustering massive text and categorical data streams. *Knowledge and information systems*, 24(2):171–196, 2010.

[2] N. Begum, L. Ulanova, J. Wang, and E. Keogh. Accelerating dynamic time warping clustering with a novel admissible pruning strategy. In *KDD*, 2015.

[3] S. Budalakoti, A. N. Srivastava, R. Akella, and E. Turkov. Anomaly detection in large sets of high-dimensional symbol sequences. *Tech. Rep*, 2006.

[4] V. Chandola, D. Cheboli, and V. Kumar. Detecting anomalies in a time series database. *Tech. Rep.*, 2009.

[5] M. Gupta, J. Gao, C. Aggarwal, and J. Han. Outlier detection for temporal data. *Synthesis Lectures on Data Mining and Knowledge Discovery*, 2014.

[6] D. M. Hawkins. *Identification of outliers*, volume 11. Springer, 1980.

[7] Z. He, X. Xu, and S. Deng. Discovering cluster-based local outliers. *Pattern Recognition Letters*, 2003.

[8] R. J. Hyndman, E. Wang, and N. Laptev. Large-scale unusual time series detection. In *Proceedings of International Conference on Data Mining series*, 2015.

[9] H. Izakian and W. Pedrycz. Anomaly detection and characterization in spatial time series data: A cluster-centric approach. *IEEE.T.Fuzzy Syst.*, 2014.

[10] P. Jaccard. The distribution of the flora in the alpine zone. *New phytologist*, 11(2):37–50, 1912.

[11] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra. Dimensionality reduction for fast similarity search in large time series databases. *Knowledge and information Systems*, 2001.

[12] E. Keogh and J. Lin. Clustering of time-series subsequences is meaningless: implications for previous and future research. *Knowl. and Inf. Syst.*, 2005.

[13] E. Keogh, J. Lin, and A. Fu. Hot sax: Efficiently finding the most unusual time series subsequence. In *ICDM*, 2005.

[14] E. Keogh, J. Lin, S.-H. Lee, and H. Van Herle. Finding the most unusual time series subsequence: algorithms and applications. *Knowl. and Inf. Syst.*, 2007.

[15] N. Laptev, S. Amizadeh, and I. Flint. Generic and scalable framework for automated time-series anomaly detection. In *KDD*, 2015.

[16] Y. Li, J. Lin, and T. Oates. Visualizing variable-length time series motifs. In *SDM*, pages 895–906, 2012.

[17] J. Lin, E. Keogh, L. Wei, and S. Lonardi. Experiencing SAX: a novel symbolic representation of time series. *Data Mining and knowledge discovery*, 2007.

[18] C. G. Nevill-Manning and I. H. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *J. Artif. Intell. Res.(JAIR)*, 1997.

[19] A. Pires and C. Santos-Pereira. Using clustering and robust estimators to detect outliers in multivariate data. In *the Int'l. Conf. on Robust Stats.*, 2005.

[20] F. Pukelsheim. The three sigma rule. *The American Statistician*, 48(2):88–91, 1994.

[21] P. Senin, J. Lin, X. Wang, T. Oates, S. Gandhi, A. P. Boedihardjo, C. Chen, and S. Frankenstein. Time series anomaly discovery with grammar-based compression. In *EDBT*, pages 481–492, 2015.

[22] P. Senin, J. Lin, X. Wang, T. Oates, S. Gandhi, A. P. Boedihardjo, C. Chen, S. Frankenstein, and M. Lerner. Grammarviz 2.0: a tool for grammar-based pattern discovery in time series. In *ECML/PKDD*, pages 468–472. Springer, 2014.

[23] K. Sequeira and M. Zaki. Admit: anomaly-based data mining for intrusions. In *KDD*, 2002.

[24] H. Sun, Y. Bao, F. Zhao, G. Yu, and D. Wang. Cd-trees: An efficient index structure for outlier detection. In *WAIM*. Springer, 2004.

[25] H. Wang, M. Tang, Y.-S. Park, and C. E. Priebe. Locality statistics for anomaly detection in time series of graphs. *Sig. Pro., IEEE Trans. on*, 2014.

[26] X. Wang, Y. Gao, J. Lin, H. Rangwala, and R. Mittu. A machine learning approach to false alarm detection for critical arrhythmia alarms. In *ICMLA*, 2015.

[27] L. Wei, E. Keogh, and X. Xi. Saxually explicit images: finding unusual shapes. In *ICDM*, 2006.

[28] Y. Xie, J. Huang, and R. Willett. Change-point detection for high-dimensional time series with missing data. *J. Sel. Top. Signal Process.*, 2013.

[29] Y. Zhang, N. Meratnia, and P. Havinga. Outlier detection techniques for wireless sensor networks: A survey. *Com. Surveys & Tutorials, IEEE*, 2010.