MITSUBISHI ELECTRIC RESEARCH LABORATORIES
http://www.merl.com

Exemplar Learning for Extremely Efficient Anomaly Detection in Real-Valued Time Series

Jones, M.J.; Nikovski, D.N.; Imamura, M.; Hirata, T.

TR2016-027 March 2016

Abstract

We investigate algorithms for efficiently detecting anomalies in real-valued one-dimensional time series. Past work has shown that a simple brute force algorithm that uses as an anomaly score the Euclidean distance between nearest neighbors of subsequences from a testing time series and a training time series is an effective anomaly detector. We investigate a very efficient implementation of this method and show that it is still too slow for most real world applications. Next, we present a new method based on summarizing the training time series with a small set of exemplars. The exemplars we use are feature vectors that capture both the high frequency and low frequency information in sets of similar subsequences of the time series. We show that this exemplar-based method is both much faster than the efficient brute force method as well as a prediction-based method and also handles a wider range of anomalies. Our exemplar-based algorithm is able to process time series in minutes that would take other methods days to process.

Journal of Data Mining and Knowledge Discovery

Exemplar Learning for Extremely Efficient Anomaly Detection in Real-Valued Time Series

Michael Jones · Daniel Nikovski · Makoto Imamura · Takahisa Hirata

Abstract We investigate algorithms for efficiently detecting anomalies in real-valued one-dimensional time series. Past work has shown that a simple brute force algorithm that uses as an anomaly score the Euclidean distance between nearest neighbors of subsequences from a testing time series and a training time series is an effective anomaly detector. We investigate a very efficient implementation of this method and show that it is still too slow for most real world applications. Next, we present a new method based on summarizing the training time series with a small set of exemplars. The exemplars we use are feature vectors that capture both the high frequency and low frequency information in sets of similar subsequences of the time series. We show that this exemplar-based method is both much faster than the efficient brute force method as well as a prediction-based method and also handles a wider range of anomalies. Our exemplar-based algorithm is able to process time series in minutes that would take other methods days to process.

Keywords anomaly detection · time series · exemplar learning

1 Introduction

The problem of anomaly detection in real-valued time series has a number of useful applications. It is important for detecting faults in industrial equipment (equipment condition monitoring), detecting abnormalities in electrocardiograms (patient health monitoring) and detecting interesting phenomena in scientific data (such as detecting

Michael Jones and Daniel Nikovski
MERL
201 Broadway
E-mail: {mjones,nikovski}@merl.com

Makoto Imamura and Takahisa Hirata
Mitsubishi Electric
Information Technology Center
E-mail: {Imamura.Makoto,Hirata.Takahisa}@bx.MitsubishiElectric.co.jp

stars in astronomical light data) to name a few. With the rise of big data, it is increasingly important for anomaly detection algorithms to be very efficient and to scale to large time series. We present a robust anomaly detection algorithm that is efficient enough to process very large time series.

We formulate the problem of anomaly detection as follows. Given a training time series which defines normal behavior of a signal and a testing time series which may contain anomalies, find all parts of the testing time series that do not have a close match to any part of the training time series. There have been a number of different algorithms proposed for solving this basic problem (see [5] for a survey). The variety of different approaches include predictive techniques [16, 13] that predict the current time series value from past values, an immunology inspired approach [8], Self Organizing Maps (SOM) [21], trajectory modeling [17], subspace trajectories [15], and autoregressive models [3].

Most of the past work has focused on a single domain or only presented results on a small set of different time series. It is very difficult to know how robust and general an algorithm is unless it is tested on many different time series. One very simple algorithm has proven to be very effective over a wide range of different types of time series. It uses a sliding window over the testing time series to find the closest matching subsequence of the training time series using the Euclidean distance to measure the distance between subsequences [11]. The Euclidean distance to the nearest neighbor subsequence is the anomaly score for each testing subsequence. We will call this simple algorithm, the Brute Force Euclidean Distance (BFED) algorithm. Pseudo-code for a naive implementation of this algorithm is given in Figure 1. This algorithm is the basis of the discord algorithm of Keogh et al. [11]. Their paper showed how to greatly speed up this simple algorithm if only the top few discords of a long time series are needed. The top discord is the subsequence with the largest nearest neighbor distance to the training time series - i.e. it is the most anomalous subsequence in the testing time series. In our case, we cannot apply Keogh et al.'s efficient discord finding algorithm because we require an anomaly score for every testing subsequence. Nevertheless, their work showed the effectiveness of using the Euclidean distance on subsequences for finding anomalies over a wide range of different types of time series. This finding was also confirmed by Chandola et al. [6] who compared many different anomaly detection methods for one dimensional real-valued time series including the BFED algorithm (called WINC in their paper), kernel based algorithms, predictive methods, and segmentation based techniques. Their results showed that the BFED algorithm was the most accurate over the 19 time series they tested.

In the next section we will evaluate the speed of an optimized BFED algorithm. Then in Sections 3, 4 and 5 we will introduce an exemplar-based method, and show in Section 6 that it is orders of magnitude faster and detects a wider range of anomalies.

2 Speeding up the BFED algorithm

Since the BFED algorithm is one of the best algorithms for detecting anomalies in real-valued time series, a natural question is whether it is fast enough to be useful in practice. Certainly, a naive implementation of BFED is not very useful for real appli-

```
Input: training time series T[1...n], testing time series Q[1...m] and subsequence length w
Output: vector of anomaly scores S[1..m-w]
for i=1 to m-w do
    bsf = INF;
    for j=1 to n-w do
        d = 0;
        for k=1 to w do
            d = d + (T[j+k] − Q[i+k])^2
        d = sqrt(d);
        if d < bsf then
            bsf = d;
    S[i] = bsf;
```

Algorithm 1: Pseudo-code for the naive implementation of the Brute Force Euclidean Distance anomaly detection algorithm.

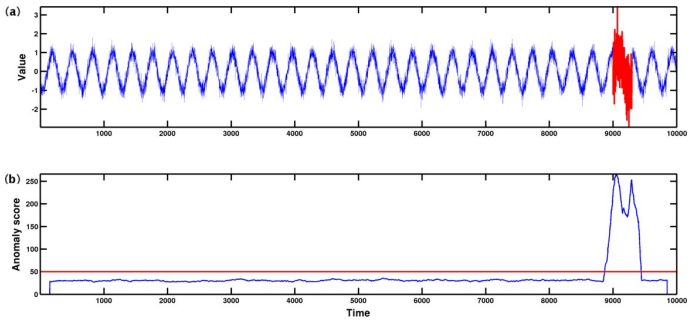

Fig. 1 a) Noisy sine testing time series. An anomaly consisting of larger magnitude noise exists in locations 9001 to 9300. The training time series is not pictured but is essentially the same as the normal portion of the testing time series. b) Anomaly scores computed from BFED algorithm. The red line is a threshold above which an anomaly is indicated.

cations. Its running time is $O(nmw)$ where n is the length of the training time series, m is the length of the testing time series, and w is the length of each subsequence. As we will show, the details of the implementation can greatly effect the actual running time, but even the most efficient implementation cannot handle very large time series.

To evaluate the running time of various implementations of the BFED algorithm, we will use a noisy sine time series for training and testing. The testing time series is shown in Figure 1a. It consists of a sine wave with Gaussian noise added (mean 0, standard deviation .25). An anomaly exists from time steps 9001 to 9300 which consists of abnormally large noise (standard deviation .75). The training time series was generated with the same parameters but without any anomalies. Each has 10,000 time steps. Since the period of the sine wave is roughly 300 time steps, this is the window size used for anomaly detection. The naive implementation requires $9700 * 9700 > 94$ million distance calculations between subsequences of size 300.

A recent paper by Rakthanmanon et al. [18] showed how to greatly speed up subsequence search using a set of optimizations called the UCR Suite. Their paper

redefined the state-of-the-art for searching within a time series. The focus of that paper was on using dynamic time warping as the distance function, but it also discussed optimizations using Euclidean distance which we focus on in this paper. In their paper, each subsequence is first z-normalized (by subtracting the mean and dividing by the standard deviation) before comparing it to other subsequences. For anomaly detection this is not always the best thing to do since a difference in either the mean or standard deviation of a subsequence compared to non-anomalous time series may indicate an anomaly. For this reason, we do not z-normalize subsequences to compare them.[1]

Despite this difference, all of the optimizations for speeding up subsequence search using Euclidean distance in the UCR Suite can still be used to greatly speed up the BFED algorithm. There are three optimizations used in the UCR Suite. The first is simply to use the squared Euclidean distance which eliminates the need to compute square roots. The second is to use early abandoning which means that the calculation of the sum of squared differences between two subsequences is abandoned if the partial sum is already greater than the best-so-far distance. The best-so-far distance is the minimum distance found between the test subsequence and all training subsequences tried so far. The third optimization is to reorder the elements of the subsequences being compared so that elements that are more likely to have large differences are evaluated first. In [18], this is done by sorting the testing subsequence by the absolute values of the z-normalized subsequence values. Since we are not using z-normalization, we can instead sort by the absolute values of the raw subsequence values. This is different, but it also leads to a significant speed-up.

Finally, we will add one more optimization that comes from the fact that, in our case, we are finding nearest neighbors for multiple overlapping subsequences in the testing time series instead of having only one query subsequence. We can take advantage of this by initializing the best-so-far distance of testing subsequence $i+1$ to the Euclidean distance between it and the nearest neighbor of testing subsequence i. This gives us a low starting value for the best-so-far distance which results in more distance calculations abandoning early.

We tested the effect of each of these optimizations on the noisy sine example of Figure 1a. The running times for implementations that successively add each of these optimizations is shown in Table 1. We should note that we modified the code provided by UCR [18] to create our implementations of these variations of the BFED algorithm. All experiments in this paper were done on a 3.16 GHz Intel processor. The table shows that the early abandoning optimization has the largest effect by far. The sqrt optimization is unimportant. Reordering and improved initialization of the best-so-far distance have modest effects. Together these optimizations improve the speed of the BFED algorithm by a factor of about 4.2 on this example. Incidentally, all of the implementations shown in Table 1 yield exactly the same anomaly scores which are shown in Figure 1b.

Although the speed of the optimized BFED algorithm is reasonable on this example, it becomes impractical on modestly large time series. For example, for a noisy

[1] We did test the z-normalized BFED algorithm and found it to be less accurate and slower for anomaly detection.

Method	Run Time
BFED (no optimizations)	199.02 sec
BFED (no sqrt)	198.80 sec
BFED (no sqrt, early abandon)	53.98 sec
BFED (no sqrt, early abandon, reorder)	49.04 sec
BFED (no sqrt, early abandon, reorder, initialization of best-so-far distance)	46.96 sec

Table 1 Comparison of the running times of various implementations of brute force Euclidean distance BFED anomaly detection.

sine problem with 1 million time steps for training and 10 million time steps for testing, the number of Euclidean distance calculations between subsequences is about 10 trillion and the running time is about 51 days for the fully optimized BFED implementation. In the next sections we will present an anomaly detection algorithm that can process this data in less then 4 minutes while improving on the accuracy of BFED as well as reducing the memory requirements.

3 Exemplars for fast anomaly detection

For the problem of anomaly detection the *location* of the best matching training subsequence for each testing subsequence is not needed. Only the *distance* is necessary. This fact allows the possibility of replacing the training time series with a more compact summary of it. With this insight, we propose to replace the training time series with a set of exemplars that summarize all the subsequences in it. The first question is what an exemplar should be. One possibility is for each exemplar to be simply a different raw subsequence of the training time series. The problem with this is that very many such exemplars would be needed to retain all of the variations present in the training time series. Another possibility is for each exemplar to be an average of similar subsequences. Averaging subsequences results in smoothing and the loss of most of the stochastic components (such as noise) of the subsequences. Each exemplar would mainly retain the different trajectories present in the training data. The set of exemplars should ideally represent both the trajectories and the stochastic variations present in the training subsequences. With this motivation we propose a representation of exemplars we call Statistical and Smoothed Trajectory (SST) features. This representation was also used in an earlier paper of ours on anomaly detection in multidimensional time series [10].

We represent a subsequence as a trajectory component that captures the shape of the time series within the window, and a statistical component that captures the stochastic component. These components can also be thought of roughly as the low frequency (trajectory) and high frequency (stochastic) components. The trajectory component is computed using a simple fixed-window running average to yield a smoothed time series after subtracting the mean of the window. Because of smoothing, half of the values in the smoothed time series can be discarded without losing important information. Thus, the trajectory component has w/2 elements. See Figure 2b. The statistical component is a small set of statistics computed over time series

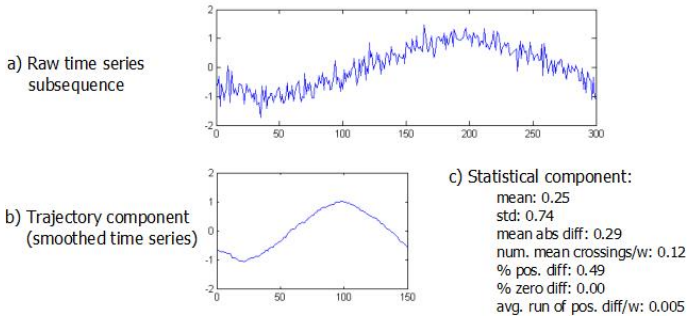

Fig. 2 Example time series subsequence (a) along with its trajectory (b) and statistical (c) components.

values in the subsequence which are designed to characterize the stochastic information in the raw time series subsequence. The statistics used for experiments in this paper are mean, standard deviation, mean of the absolute difference ($|T[i] - T[i+1]|$ where $T[i]$ is the value of time series T at time step i), number of mean crossings divided by window length, percentage of positive differences, percentage of zero differences, and the average length of a run of positive differences divided by window length. Figure 2c shows the vector of statistics for an example window. This choice of statistics has worked well in practice, but other statistics would likely also work well. Combining the trajectory and statistical components into one feature vector yields a vector of w/2+7 real values.

We would like to note that the concept of motifs [14] is similar to exemplars but has important distinctions. A motif is typically defined as an approximately repeated subsequence of a time series [7]. Exemplars (in the context of this paper) are not necessarily repeating. The set of exemplars learned from a training time series should represent every subsequence of the time series including subsequences that have few or no close matches in the rest of the time series.

The next question is how to learn a set of exemplars given a training time series. There is a rich literature on this topic [1]. In the next section we present an efficient exemplar learning algorithm that is tailored for overlapping subsequences of a time series.

4 Efficient exemplar learning

The goal of exemplar learning is to find a set of feature vectors that accurately represent all of the variations in the set of subsequences of the training time series. Our approach is to initialize the set of exemplars to the set of SST features for all subsequences in the training time series. In other words, each SST feature vector for all subsequences is a separate exemplar to begin with. Our exemplar learning algorithm then successively merges exemplars with small distance between them until there are no nearby exemplars left to merge.

A naive implementation of this basic idea is very inefficient. It would require $O(n^2)$ distance computations (to find the nearest neighbor for each subsequence) each

of which is $O(w)$ for a total of $O(n^2w)$. This is the same time complexity as the naive BFED algorithm when the length of the training and testing time series are both n. We will present a hierarchical merging algorithm that runs in $O(nw)$ for typical time series.

First, we need to define a distance between SST feature vectors v_1 and v_2. Given the success of the Euclidean distance for anomaly detection already discussed, we define the SST distance as the Euclidean distance between the trajectory components plus the Euclidean distance between the statistical components weighted by $\frac{w/2}{7}$ in order to give equal weight to each component.

$$dist(v_1, v_2) = \sum_{i=1}^{l} (v_1.t(i) - v_2.t(i))^2 + \frac{l}{7} \sum_{i=1}^{7} (v_1.s(i) - v_2.s(i))^2 \quad (1)$$

where v_1 and v_2 are two feature vectors, $v_j.t$ is the length $l = w/2$ trajectory component of v_j, and $v_j.s$ is the length 7 statistical component of v_j.

4.1 Initial merging

Based on the observation that overlapping subsequences often have small distances between them, we reduce the size of the initial exemplar set by merging the SST feature vectors of similar overlapping subsequences. To merge two SST feature vectors we simply take a weighted average of the two vectors. The weight for each feature vector is the number of features vectors that have already been averaged into it. This weight is 1 for all of the initial exemplars and is the sum of the two weights when feature vectors are merged.

To explain the initial merging procedure in more detail, let us define v_i as the feature vector corresponding to subsequence T[i...i+w-1]. A threshold on the distance is needed to determine whether two SST feature vectors are close enough. We describe a method for automatically selecting this threshold in subsection 4.4. The initial merging algorithm starts with feature vector v_1 and computes the distance to successive feature vectors until the distance is greater than the threshold. Let v_a be the last feature vector whose distance to v_1 is below threshold. Then we continue searching forward for the furthest overlapping feature vector to v_a whose distance is below threshold. Call this feature vector v_b. Feature vectors v_1 through v_b are then merged. This process is then repeated starting at the next feature vector v_{b+1}. This initial merging of overlapping subsequences is fast ($O(nw)$ since it only makes one pass over the set of SST feature vectors) and typically results in about a 90% reduction in the number of exemplars. Figure 3 illustates the initial merging procedure.

4.2 Hierarchical exemplar learning

After the initial merging phase, the resulting set of exemplars are passed to an hierarchical exemplar learning algorithm. The goal of this algorithm is to efficiently find and merge all similar exemplars (still represented by SST feature vectors) - not just ones that represent overlapping subsequences. Let the number of exemplars in the

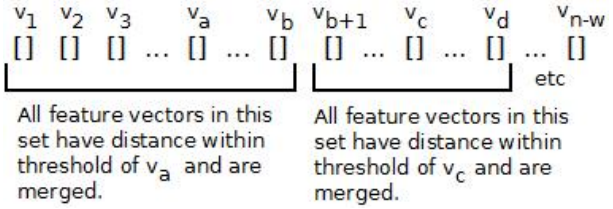

Fig. 3 Illustration of initial exemplar merging procedure. See text for explanation.

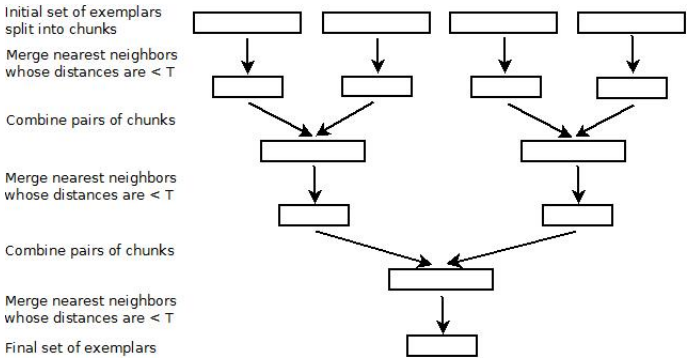

Fig. 4 Illustration of hierarchical exemplar learning. Each rectangle represents a chunk of exemplars. On each iteration, a brute force exemplar learning algorithm is used to merge exemplars in each chunk and thereby reduce its size. Then pairs of chunks are combined and the process repeats until there is only one chunk of exemplars remaining.

current set be $\hat{n}$. The hierarchical exemplar learning algorithm splits the current set of exemplars into $\lfloor \hat{n}/C \rfloor$ chunks of size C. (The size of the last chunk will be smaller than C unless $\hat{n}$ is an exact multiple of C.) We call C the chunk size. The first C exemplars are assigned to the first chunk, the second C to the second chunk, and so on. In the experiments we report on later, C is set to 150. This value is not terribly important although it does effect the speed of the algorithm somewhat. For each chunk, a brute force exemplar learning algorithm is run. The brute force exemplar learning algorithm first finds the nearest neighbor exemplar in the chunk for each exemplar in the chunk. To do this it uses the distance function of equation 1 and a brute force search with early abandoning. The two exemplars with the smallest distance between them are merged. The nearest neighbor of the merged exemplar is found and any other exemplars that had one of the two exemplars as a nearest neighbor have their nearest neighbors recalculated. This process of merging the two closest exemplars continues until the smallest nearest neighbor distance is above a threshold. At that point the brute force exemplar learning algorithm stops and returns the final set of exemplars for that chunk.

After the brute force exemplar learning algorithm has been run on each chunk, pairs of chunks are combined into one new chunk. This simply means that all exemplars in the two chunks are put into a single new chunk. The brute force exemplar

learning algorithm is then run on each of the new chunks and then these chunks are paired and combined and so on until there is only one chunk remaining. This is the final set of exemplars returned by the hierarchical exemplar learning algorithm. Figure 4 illustrates the basic idea of the algorithm. It is worth noting that this algorithm can also be used to efficiently update a set of exemplars given new training data without having to run exemplar learning from scratch.

4.3 Time complexity analysis

To derive the time complexity of the hierarchical exemplar learning algorithm, first let $\hat{n}$ be the number of exemplars after the initial merging procedure. Since $\hat{n}$ is a fraction of n (the length of the training series), it is $O(n)$ in length. The algorithm starts with $\hat{n}/C$ chunks with each chunk containing C exemplars. The brute force algorithm run on a single chunk of size C has time complextiy $O(C^2w) = O(w)$ since C is a constant. Thus, the total time complexity for the first iteration of hierarchical exemplar learning is $\frac{\hat{n}}{C} \cdot O(C^2w) \Rightarrow O(nw)$. On the second iteration, there are half as many chunks, each with some constant, r, times C exemplars in each chunk. The coefficient, $r, (0 < r \leq 2)$ depends on the training time series. Each chunk may have a different coefficient, so let f_1 be the maximum (worst-case) r over all chunks for the second iteration. This yields a bound on the total time for the second iteration of $\frac{\hat{n}}{2C}$ chunks times $\lambda(f_1C)^2w$ time to process each chunk since each chunk has f_1C or fewer exemplars, where λ is a constant which allows us to remove the big O notation for now. Simplifying, we get $\lambda\frac{\hat{n}}{2}f_1^2Cw$ total time for the 2nd iteration.

Continuing on in this fashion, a bound on the time complexity for the i^{th} iteration can be written

$$t_i = \lambda\frac{\hat{n}}{2^i}(f^i)^2Cw \tag{2}$$

where t_i is the total time for iteration i and $f = \max_i(f_i), 0 < f \leq 2$ is the worst case factor that the chunk size changes by on each iteration. Summing the times for each iteration (there are $\log_2\hat{n}$ iterations total) yields the total time complexity of the algorithm:

$$T = \sum_{i=0}^{\log_2\hat{n}}(\lambda\frac{\hat{n}}{2^i}(f^i)^2Cw) = \lambda(\hat{n}Cw)\sum_{i=0}^{\log_2\hat{n}}(\frac{f^2}{2})^i \tag{3}$$

The sum $\sum_{i=0}^{\log_2\hat{n}}(\frac{f^2}{2})^i$ depends on the value of f. If $f = \sqrt{2}$ then $\frac{f^2}{2} = 1$ and $\sum_{i=0}^{\log_2\hat{n}}1 = \log_2\hat{n}$ which yields a total time complexity of $O(nw\log_2 n)$. If $f < \sqrt{2}$ then $\frac{f^2}{2} < 1$ and the sum converges so that $\sum_{i=0}^{\log_2\hat{n}}(\frac{f^2}{2})^i < 1/(1-\frac{f^2}{2}) \in \mathscr{R}$ which yields a total time complexity of $O(nw)$. Finally, if $f > \sqrt{2}$ then $\frac{f^2}{2} > 1$ and the sum diverges so that $\sum_{i=0}^{\log_2\hat{n}}(\frac{f^2}{2})^i = \sum_{i=0}^{\log_2\hat{n}}2^{ai}$ where a is a real number chosen so that $2^a = \frac{f^2}{2}$. Since $0 < f \leq 2$ then $0 < \frac{f^2}{2} \leq 2$ Thus, $0 < 2^a \leq 2$. This implies $-\infty < a \leq 1$.

Using the formula for the sum of a geometric series,

$$\sum_{i=0}^{\log_2\hat{n}}2^{ai} = \frac{1-(2^a)^{\log_2(\hat{n})}}{1-2^a} = \frac{1-(\hat{n})^a}{1-2^a} \tag{4}$$

Time series length	10^3	10^4	10^5	10^6	10^7
Running time (sec)	.05	.44	4.15	41.59	441.61

Table 2 Running times for hierarchical exemplar learning for various lengths of the training time series.

which is at most $O(\hat{n})$ since $a \leq 1$. Thus, if $f > \sqrt{2}$ the total time complexity is at worst $O(n^2 w)$.

For the best case time complexity to occur the requirement is that $f < \sqrt{2}$ which means that when pairs of chunks are combined the combined chunk size is less than $\sqrt{2}$ times the chunk size on the previous iteration. This means that brute force exemplar learning should reduce the number of exemplars in each chunk to at most .707 times the starting number of exemplars in a chunk. A roughly 30% reduction is not difficult to obtain. In practice, on every time series we have found to test on (including all of the time series in the experimental section), the worst case time complexity has not occurred. For example, we tested the hierarchical exemplar learning algorithm on the noisy sine time series with different numbers of time steps (from 1000 to 10 million). The running times are shown in Table 2. The table clearly shows that the running time is linear in the size of the training time series.

4.4 Setting the SST distance threshold

As described earlier, the inital merging procedure and hierarchical exemplar learning both use a threshold on the SST distance (given by equation 1) to determine when two feature vectors are similar enough to merge. This threshold is chosen automatically by computing the mean (μ) and standard deviation (σ) of the distance between an SST feature vector for a subsequence starting at location i and one starting at location $i+s$ where s is a constant whole number chosen based on the subsequence length. In the experiments described later, we use $s = 1 + (w/100)$. The mean, μ, and standard deviation, σ, are computed by sampling a set of training subsequences at random locations and computing the distance between v_i and v_{i+s}. The number of subsequences sampled is a constant (we used 1000), and does not depend on the length of the training time series. The threshold is then set to $\mu + 3\sigma$. The intuition behind this threshold is that it allows most subsequences whose locations are within s time steps of each other to be merged.

4.5 Exemplar learning using k-means clustering

As a simple alternative to hierarchical exemplar learning, we have also experimented with a k-means clustering algorithm to choose exemplars for a fixed choice of k. The k-means algorithm first randomly selects k exemplars from the initial set of SST feature vectors computed from the training time series to serve as the initial means. Every SST feature vector is then assigned to the nearest mean. All feature vectors assigned to a mean are averaged together to yield a new mean. This process of assigning feature vectors to a mean and then averaging to update the means is iterated

a fixed number of times. We tested this simple algorithm on the 26 test sets described in Section 6 with k set to 50 (which is the average number of exemplars chosen by our hierarchical exemplar learning algorithm on the test sets) and the number of iterations set to 3. With these parameter settings, the accuracy of SST exemplars is slightly lower compared to hierarchical exemplar learning (detection rate of 42/45 versus 44/45 on the 26 test sets from Section 6) with slightly slower overall speed (13.83 seconds versus 11.84 seconds for all of the test sets in Section 6). The accuracy of k-means exemplar learning could be made to match that of hierarchical exemplar learning if a method of automatically optimizing k were used. However, this would make the k-means clustering algorithm much slower since multiple choices for k would need to be tested. Thus, k-means clustering is a reasonable alternative for exemplar learning, but appears to be unable to match the speed and accuracy of the hierarchical exemplar learning algorithm.

5 Anomaly detection with SST exemplars

After a set of exemplars have been learned to summarize the training time series, there is one final step to finalize our model. Since each exemplar represents a set of similar SST features, we have not just a mean feature vector for each exemplar, but also a standard deviation for each component of the feature vector. The standard deviation is computed during exemplar learning by keeping track of the sum of squares of each component of each feature vector that is merged to form an exemplar. After exemplar learning completes, we use the following formula to compute the standard deviation for each component of the feature vector:

$$\sigma_j = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(v_{ij}^2) - \mu_j^2} \tag{5}$$

where σ_j is the standard deviation of the jth component of the exemplar, $\{v_i\}$ is the set of SST feature vectors that were merged together to form this exemplar, N is the number of such feature vectors, and μ_j is the mean of the jth component of the exemplar.

Thus, each exemplar is represented by a mean SST feature vector (with $w/2+7$ components) and a standard deviation for each component of the SST feature vector (also with $w/2+7$ components) for a total of $w+14$ real numbers to represent each exemplar.

Computing the standard deviation for each component of the SST feature tells us how much a test SST feature can deviate from an exemplar's mean before the deviation becomes unusual (i.e. not common in the training time series). Given this motivation, we define a distance between a single SST feature vector, v (computed from a single time series subsequence) and an exemplar, e, that includes both mean and standard deviation components.

$$d(v,e) = \sum_{i=1}^{l} max(0, \frac{|v.t(i) - e.t(i)|}{e.\sigma(i)} - 3)$$

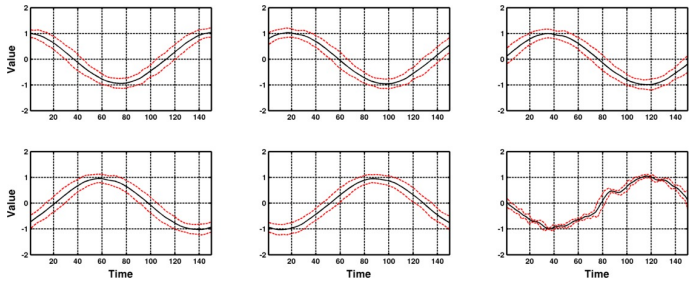

Fig. 5 Illustration of eight exemplars learned for the noisy sine time series. Only the trajectory component of the SST feature is shown. The red dashed curves indicate 3 standard deviations from the mean trajectory (solid black curve).

$$+\frac{l}{7}\sum_{i=1}^{7} max(0, \frac{|v.s(i) - e.s(i)|}{e.\varepsilon(i)} - 3) \tag{6}$$

where v is the SST feature vector for the current window consisting of a trajectory vector, $v.t$ and a statistical vector $v.s$, e is an exemplar consisting of trajectory ($e.t$) and statistical ($e.s$) vectors as well as the corresponding standard deviation vectors, $e.\sigma$ for the trajectory vector and $e.\varepsilon$ for the statistical component. The length of a trajectory component is $l = w/2$.

This distance corresponds to assigning 0 distance for each element of the trajectory or statistical component that is less than 3 standard deviations from the mean and otherwise assigning the absolute value of the difference divided by the standard deviation for each element that is more than 3 standard deviations from the mean. In equation 6 and in our experiments, the statistical component is given equal weighting to the trajectory component, although this weighting can be changed based on the application (for example, in some domain, the trajectory component may not be that important in which case the statistical component could be given more weight).

This distance function is used to assign an anomaly score to each subsequence in the testing time series. For a given testing subsequence, the SST feature vector is computed and a brute force nearest neighbor search is done for the nearest exemplar using the distance function in equation 6. This nearest neighbor search uses early abandoning as described in section 2 to make it very efficient. Thus, the anomaly score, $S[i]$, for subsequence i is:

$$S[i] = min_j\, d(v_i, e_j) \tag{7}$$

where $\{e\}_{j=1}^{N}$ is the set of exemplars learned from the training time series by hierarchical exemplar learning. The time required to assign anomaly scores to every testing subsequence is thus $O(Nmw)$ where $N \ll n$ is the number of exemplars, m is the length of the testing time series and w is the subsequence length.

To illustrate the kinds of exemplars that are learned by exemplar learning, Figure 5 shows the mean and standard deviation of the trajectory components for some of the exemplars learned for the noisy sine time series introduced earlier. (The statistical components of the exemplars are not illustrated.) Notice that the exemplars are basically shifted sine waves.

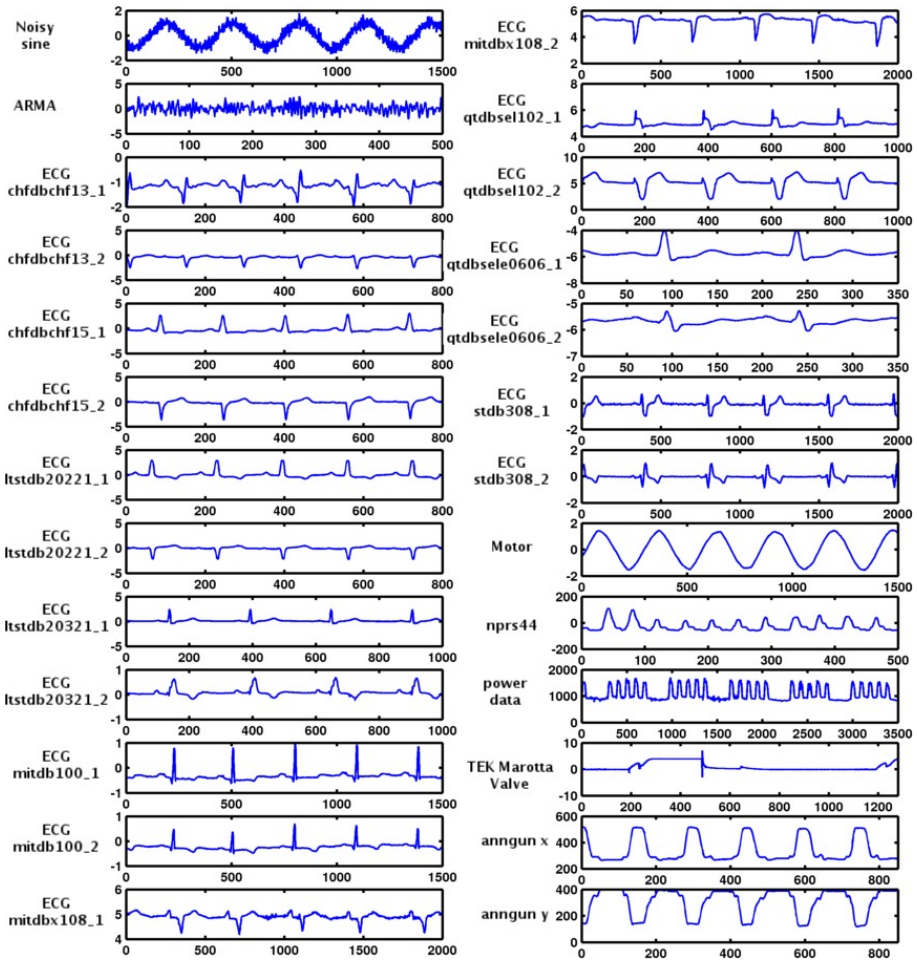

Fig. 6 Part of each training time series used in our experiments

6 Experiments

We tested our anomaly detection algorithm on a wide variety of different time series including all of the time series used in the paper [11] (available from [12]) as well as some others. A portion of the training time series for the 26 data sets we used is shown in Figure 6. Many of the time series from [12] were not originally split into training and testing series. In these cases, we have split the original time series into a training time series that only contains normal data with no anomalies and a testing time series that contains some anomalous sections.

We tested our SST exemplar-based anomaly detection algorithm as well as the BFED algorithm and our implementation of the Support Vector Regression (SVR) based algorithm of Ma and Perkins [16]. For our algorithm a set of SST exemplars is

first learned on the training set as described in section 4 and then anomaly scores are computed for every subsequence of the testing set using the distance from equation 6 as described in section 5. The same parameters (except for the subsequence length w) were used for all time series. A step size of 1 is used to advance from one subsequence to the next so that consecutive subsequences overlap for all but their first and last elements.

The SVR-based algorithm learns a linear combination of Gaussian functions using Support Vector Regression to predict a value of the time series, $T[i+w]$ given a subsequence, $T[i,...,i+w-1]$. This nonlinear function is learned from the training time series and then used to predict each value of the testing time series. The anomaly score is the squared difference between the prediction and the observed value.

The subsequence length, w, is chosen manually and is the main parameter of all of the algorithms. None of the algorithms is very sensitive to the choice of w although its choice can effect the type of anomalies that can be detected. For periodic time series, choosing w to be roughly the length of the period works well.

We compute detection rates for various false positive rates across all testing time series. The false positive rate is the fraction of non-anomalous subsequences that are above threshold. The detection rate is the fraction of anomalous regions (not subsequences) of a time series that are detected as anomalous. An anomalous region is considered to be detected if at least one subsequence in the region has an anomaly score above threshold. This convention reflects the fact that in practice it is not important that every subsequence within an imprecisely labeled anomalous region be detected as anomalous. What is important is that at least one high anomaly score occurs in an anomalous region. A receiver operating characteristic (ROC) curve across all testing time series is computed by computing total detections over total anomalies for a fixed false positive rate for each testing time series.

The ROC curves across all testing time series for our SST exemplar-based algorithm, the BFED algorithm and the SVR-based algorithm are shown in Figure 7. The SST exemplar-based algorithm is significantly more accurate than the other two.

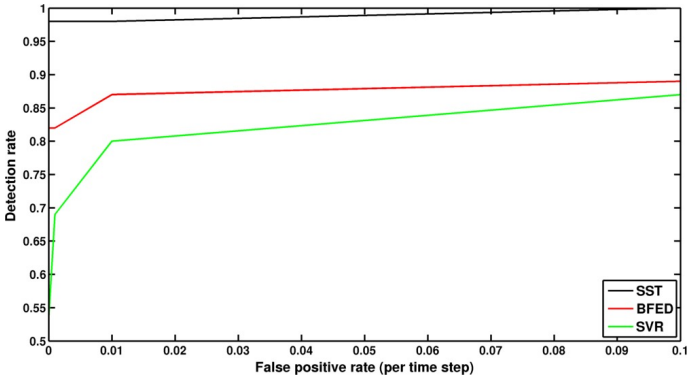

Fig. 7 ROC curves for all three methods tested.

Data set			SST exemplars		BFED	SVR
Name	Train, test length	w	N	Det. rate	Det. rate	Det. rate
noisy sine	10000, 10000	300	49	4/4	1/4	4/4
ARMA	10000, 100000	100	12	2/2	0/2	0/2
chfdbchf13_1	1875, 1875	160	17	1/1	1/1	1/1
chfdbchf13_2	1875, 1875	160	17	1/1	1/1	1/1
chfdbchf15_1	7500, 7500	160	72	1/1	1/1	0/1
chfdbchf15_2	7500, 7500	160	23	1/1	1/1	0/1
ltstdb20221_1	1875, 1875	160	24	1/1	0/1	1/1
ltstdb20221_2	1875, 1875	160	24	1/1	1/1	0/1
ltstdb20321_1	1875, 1875	200	23	1/1	1/1	0/1
ltstdb20321_2	1875, 1875	200	31	1/1	1/1	1/1
mitdb100_1	2700, 2700	300	36	1/1	1/1	0/1
mitdb100_2	2700, 2700	300	102	1/1	1/1	0/1
mitdbx108_1	5000, 5000	400	71	1/1	0/1	0/1
mitdbx108_2	5000, 5000	400	114	1/1	0/1	0/1
qtdbsel102_1	22500, 22500	200	156	1/1	1/1	1/1
qtdbsel102_2	22500, 22500	200	38	1/1	1/1	1/1
qtdbsel0606_1	700, 2300	70	15	1/1	1/1	1/1
qtdbsel0606_2	700, 2300	70	19	1/1	1/1	1/1
stdb308_1	2400, 3000	400	58	1/1	1/1	0/1
stdb308_2	2400, 3000	400	53	1/1	1/1	1/1
motor	7500, 30000	300	63	10/10	10/10	10/10
nprs44	2000, 4500	100	61	1/1	1/1	1/1
power data 1	11000, 15000	700	80	4/4	4/4	0/4
power data 2	11000, 9040	700	80	0/1	1/1	0/1
TEK	5901, 9099	256	79	3/3	3/3	0/3
anngun x	5625,5625	170	33	1/1	1/1	0/1
anngun y	5625,5625	170	39	1/1	1/1	0/1
Totals	**150501, 286139**	**N.A.**	**1309**	**44/45**	**37/45**	**24/45**

Table 3 Detection rates for our method (SST exemplars), the very efficient implementation of BFED, and the SVR-based method. Detection rates are for a threshold that yields 0 false positives for that data set and method. For our method we also list N, the number of exemplars learned. w is the subsequence length chosen for each time series.

More detailed results on each of the testing time series are shown in Tables 3 and 4. Table 3 gives the lengths of the training and testing time series and the subsequence length (w) used followed by the detection rate for 0 false positives for each of the three methods. Our method correctly detects 44 out of 45 of the anomalies while BFED detects 37 out of 45 anomalies and the SVM-based algorithm detects 24 out of 45. Table 4 gives the training and testing running times for each algorithm. (There is no training required for the BFED algorithm.) The improvement in speed for our method is striking. Our algorithm takes a total of 11.84 seconds for both training and testing for all of the time series compared to 582.38 seconds for BFED and 1826.6 seconds for the SVR-based algorithm. The speed-up is even greater for longer time series. We generated a noisy sine training time series with 1 million time steps and a testing time series with 10 million time steps. Our algorithm took 41.62 seconds for training (exemplar learning) and 177.15 seconds for testing for a total of a little over

3.6 minutes. The fully optimized BFED algorithm took over 51 days to process the same training and testing time series. This is a speed-up of over 20,000 which makes it practical to handle such large time series.

Data set	SST exemplars		BFED	SVR	
Name	Train time	Test time	Test time	Train time	Test time
noisy sine	0.41	0.16	46.35	128.99	61.74
ARMA	0.51	0.46	309.87	54.76	348.10
chfdbchf13_1	0.05	0.01	0.93	0.10	0.07
chfdbchf13_2	0.05	0.01	0.26	0.20	0.13
chfdbchf15_1	0.22	0.06	2.14	14.79	8.51
chfdbchf15_2	0.20	0.05	1.98	4.33	2.12
ltstdb20221_1	0.06	0.01	0.14	0.39	0.22
ltstdb20221_2	0.05	0.01	0.13	0.40	0.12
ltstdb20321_1	0.06	0.02	0.26	0.29	0.15
ltstdb20321_2	0.06	0.04	2.04	0.14	0.08
mitdb100_1	0.13	0.04	2.83	0.49	0.30
mitdb100_2	0.14	0.07	2.99	0.36	0.24
mitdbx108_1	0.28	0.22	17.8	0.77	0.57
mitdbx108_2	0.30	0.23	20.31	4.44	3.17
qtdbsel102_1	0.99	0.25	23.62	7.61	3.74
qtdbsel102_2	0.71	0.20	34.98	107.29	31.11
qtdbsel0606_1	0.01	0.01	.10	0.03	0.05
qtdbsel0606_2	0.01	0.01	.10	0.01	0.04
stdb308_1	0.14	0.08	2.08	0.68	0.58
stdb308_2	0.14	0.08	1.99	0.52	0.43
motor	0.28	0.43	27.48	7.33	9.08
nprs44	0.05	0.06	2.43	1.12	2.23
power data 1	0.96	1.13	40.58	284.74	397.87
power data 2	0.96	0.71	19.56	284.74	232.50
TEK	0.16	0.14	6.56	13.95	18.80
anngun x	0.14	0.07	5.71	17.93	17.34
anngun y	0.14	0.07	9.16	18.10	17.47
Totals	**7.21**	**4.63**	**582.4**	**669.8**	**1156.8**

Table 4 Running times (seconds) for our method (SST exemplars), the very efficient implementation of BFED, and the SVR-based algorithm. Both training and testing times are given for our method as well as the SVR-based method. The BFED algorithm does not use training.

Due to space constraints, we cannot discuss each of the testing time series in detail. We instead focus on some of the more interesting ones that differentiate our anomaly detection algorithm from the other algorithms. In the result figures, we show the testing time series with the anomalous regions indicated in red. Below that we show the anomaly scores for our method as well as the BFED algorithm and SVR-based algorithm. The thresholds choosen for each method (giving zero false positives) are also drawn on the anomaly score plots in red.

Earlier, we showed a noisy sine time series as an example. We created a new noisy sine testing time series containing 4 anomalies to illustrate one type of anomaly that SST exemplars and the SVR-based algorithm can detect but BFED cannot. The first

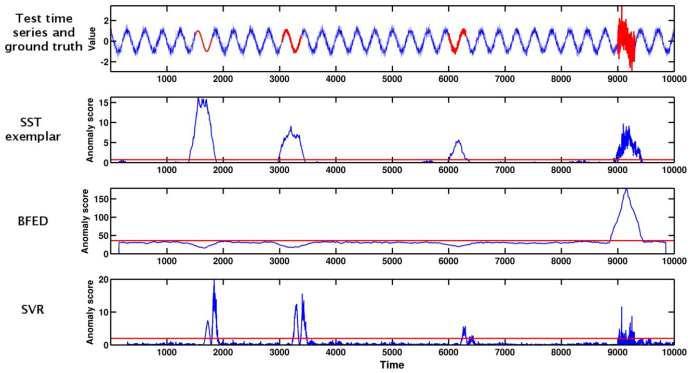

Fig. 8 Noisy sine example. Four anomalies were inserted into this synthetic time series. Our algorithm and the SVR-based algorithm detect all 4. The BFED algorithm only detects the last anomaly.

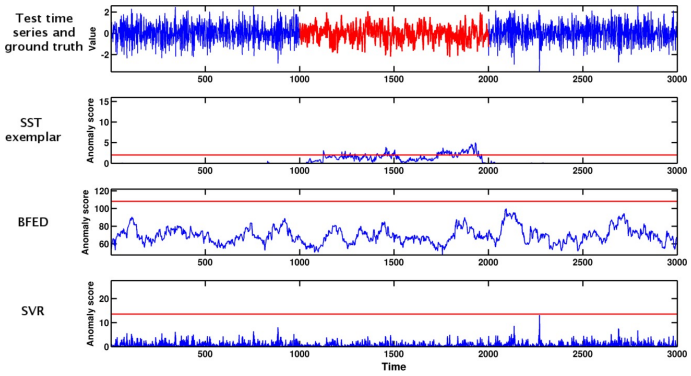

Fig. 9 ARMA example, first anomaly. Our algorithm detects this anomaly which was generated from a different ARMA model than the normal data. The BFED and SVR-based algorithms fail to detect it.

3 anomalies in the noisy sine testing time series contain smaller amplitude noise than the training time series. The Gaussian noise in the training time series has standard deviation .25. The first anomaly in the testing time series has no noise in the sine wave. This stands out as a clear anomaly in Figure 8 starting at time step 1500. The second anomaly has Gaussian noise with standard deviation 0.1 and is also visible starting at time step 3000. The third anomaly, starting at time step 6000 has Gaussian noise 0.15 and is barely perceptible. The fourth anomaly has larger amplitude noise than normal (0.75 standard deviation) and is clearly visible starting at time step 9000. Our algorithm and the SVR-based algorithm detect all 4 of these anomalies as shown in the second and fourth plots of Figure 8. For the BFED algorithm, the first 3 anomalies have lower anomaly scores than other regions (the exact opposite of what we want). Only the fourth anomaly with greater noise is detected.

As an example of a type of anomaly that our algorithm can handle but both the BFED and SVR-based algorithms fail on, we use an autoregressive moving average (ARMA) model to create a training time series of length 10,000. The same ARMA

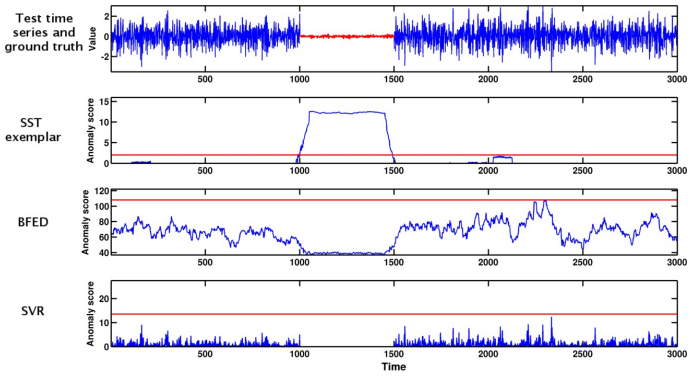

Fig. 10 ARMA example, second anomaly. This obvious anomaly is easily detected by our algorithm but is missed by both the BFED and SVR-based algorithm.

model is used to create a 100,000 length testing time series with two anomalies inserted. The first anomaly, shown in Figure 9 consists of a different ARMA model inserted into the testing time series. The difference is difficult to see visually. In this case our algorithm clearly detects the anomaly while the BFED algorithm and the SVR-based algorithm fail to detect it (see Figure 9). The second anomaly inserted into this time series was created by multiplying a section of the testing time series by 0.1. This anomaly is clearly visible in Figure 10. Our algorithm easily detects it, while the BFED and SVR-based algorithms indicate that the anomalous region is actually the least anomalous.

There are a number of electrocardiogram (ECG) time series in the testing set. Each of these contains two signals, which are processed independently. They contain a variety of different anomalies from abnormal spacing of spikes to abnormal shapes of the signals, some of which are very subtle. Furthermore, a normal ECG time series contains many normal variations which should not be confused for anomalies. One example is the ECG time series labeled ltstdb20221 that is shown in Figure 11. The anomalous region shows a premature contraction. Our algorithm detects this anomaly. The BFED algorithm also show elevated anomaly scores in the anomalous region, but it has even higher anomaly scores for normal subsequences at the beginning of the testing series. This results in the threshold being set high and the anomalies being missed. The SVR-based algorithm weakly detects this anomaly.

Next, we look at the power demand data set. This time series has power consumption for a Dutch research facility for the year 1997 (one power measurement every 15 minutes for 365 days). It shows a characteristic weekly pattern that consists of 5 power usage peaks corresponding to the 5 weekdays followed by 2 days of low power usage on the weekends. Anomalous weeks occur when one or more of the normal usage peaks during a week do not occur due to holidays. We used a section of 11,000 time steps in the middle of the year (not containing holidays) for training. This splits the remaining data into two separate testing time series which we evaluated separately. A window size of 700 was selected since this is approximately the number of time steps in one week. The first testing time series is shown in Fig-

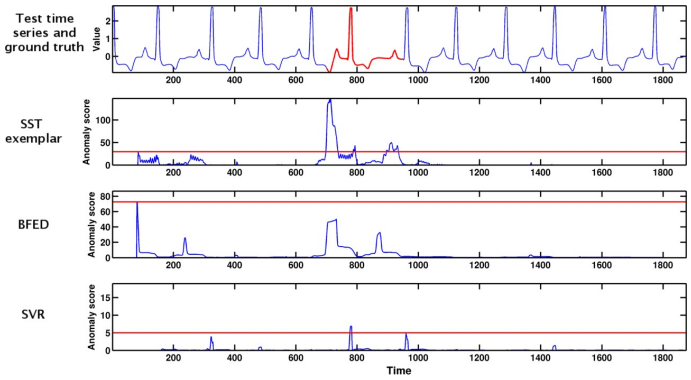

Fig. 11 An ECG example. Our method and the SVR-based algorithm successfully detect the anomaly while the BFED algorithm has a larger response in a normal region at the beginning of the time series.

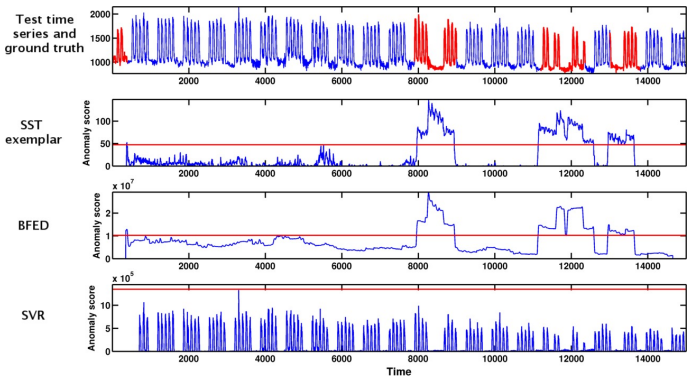

Fig. 12 Power data, 1st test series. Our method and the BFED method detect the anomalous regions (caused by holidays) in this time series, while the SVR-based method does not.

ure 12. Our SST exemplar algorithm and the BFED algorithm detect the 4 anomalous regions in this case while the SVR-based algorithm does not detect any of the anomalous regions. The second testing time series is shown in Figure 13. This is the only case in which our method fails to detect the anomaly. Actually, our algorithm does have high anomaly scores in the anomalous region. However, another region that is not labeled as anomalous has even higher anomaly scores which causes the threshold to be set too high to detect the anomaly. If one looks carefully at this other region with high anomaly scores, one can see an unusual dip and then increase in the power usage during a weekend period. Depending on the goal of the system, one might indeed classify this as anomalous. The BFED algorithm correctly detects the labeled anomalous region in this case while the SVR-based algorithm does not.

Another interesting testing time series is the Space Shuttle Marotta Valve time series [9]. This data set consists of 3 different time series labeled TEK14, TEK16 and TEK17. We use approximately the last half of TEK14 and the first half of TEK16

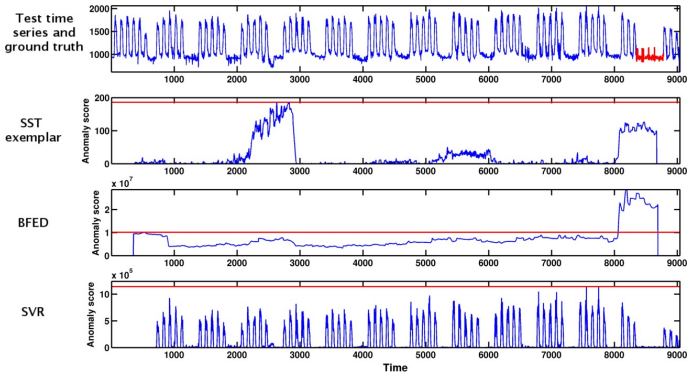

Fig. 13 Power data, 2nd test series. The BFED algorithm detects the anomaly while the SVR-based algorithm does not. Our method also has high anomaly scores for the anomalous region, but has even higher scores for a "normal" region. Close examination of the false detection reveals a unique pattern not seen in the training data.

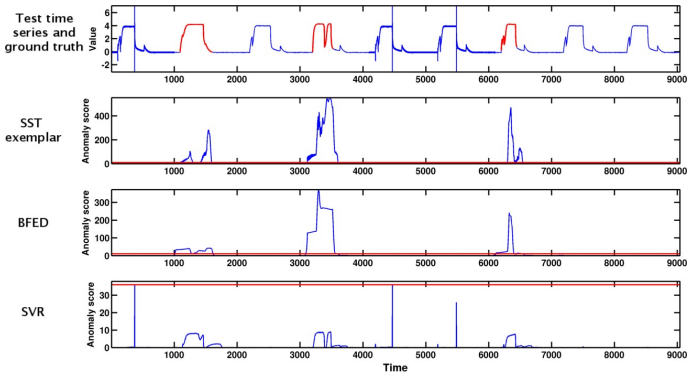

Fig. 14 Marotta Valve time series. Our method as well as the BFED algorithm detect all three anomalies. The SVR-based algorithm has higher anomaly scores in non-anomalous regions which causes it to fail to detect the actual anomalies.

(which do not contain anomalies) as the training time series. The training time series has 5901 time steps. The remainder of TEK14 and TEK16 as well as all of TEK17 are used as a testing time series and contains 9099 time steps. Results are shown in Figure 14. Our method as well as the BFED algorithm detect all three of the anomalies. The SVR-based algorithm seems to be effected by some spikes in the normal regions of the time series which causes high anomaly scores in these normal regions. This causes the threshold to be set high to avoid false positives which results in three missed detections for this method.

7 Conclusions

We have presented an algorithm for detecting anomalies in real-valued time series that improves over previous algorithms both in terms of accuracy and speed. We compared our algorithm to the simple yet effective brute force Euclidean distance algorithm which has proven to be the most accurate over a variety of different testing time series in previous work. We first investigated optimizing the running time of the simple brute force Euclidean distance (BFED) anomaly detection algorithm using the UCR Suite of optimizations. We found that while BFED can be greatly sped up, it is still not efficient enough in practice for very large times series. Next, we presented our new algorithm based on selecting a small set of exemplars to represent the variety of subsequences present in a training time series. The exemplars use Statistical and Smoothed Trajectory features which capture both the characteristic trajectory (low frequency information) as well as the stochastic characteristics (high frequency information) present in similar subsequences. We presented a novel algorithm for efficiently learning exemplars. Once exemplars are learned, assigning anomaly scores to the testing subsequences is very fast. We show improved results over the BFED algorithm and an SVR-based prediction algorithm on a large variety of different time series. We also show that our algorithm can process very large time series in minutes that would take the optimized BFED algorithm many days to process. In addition to the speed and accuracy improvements, our algorithm also requires less memory than BFED since only the exemplars need to be stored and not the entire training time series.

There are other possibilities for speeding up subsequence similarity search. For example, k-d trees [4] or related data structures could be used. However, because subsequences typically have a large number of elements (dimensions), k-d trees are no better than exhaustive search in this case. Indexing methods are another possibility [2]. The main drawback of indexing approaches is that they require storing all of the training subsequences (as opposed to only a small set of exemplars), as well as a substantial memory overhead for the indexing structure. This makes indexing approaches impractical for use with very long training time series. However, combining indexing with exemplars to quickly find the nearest exemplar appears to be a promising direction for future research.

8 Compliance with Ethical Standards

Conflict of Interest: The authors declare that they have no conflict of interest.

References

1. D. Aha, D. Kibler, and M. Albert, *Instance-Based Learning Algorithms* Machine Learning, Vol. 6, pp. 37-66. (1991).
2. I. Assent, R. Krieger, F. Afschari and T. Seidl, *The TS-Tree: Efficient Time Series Search and Retrieval*, EDBT (2008).
3. S. Bay, K. Saito, N. Ueda, and P. Langley, *A Framework for Discovering Anomalous Regimes in Multivariate Time-Series Data with Local Models*, Stanford Technical Report, (2004)

4. J. Bentley, *Multidimensional binary search trees used for associative searching*, Comm. of the ACM, 18(9) (1975)
5. V. Chandola, A. Banerjee, and V. Kumar, *Anomaly Detection: A Survey*, ACM Computing Surveys, Vol. 41, No. 3, (2009).
6. V. Chandola, D. Cheboli, and V. Kumar, *Detecting Anomalies in a Time Series Database*, Dept. of Computer Science and Engineering, Univ. of Minnesota Technical Report, TR 09-004 (2009).
7. B. Chiu, and E. Keogh, and S. Lonardi, *Probabilistic Discovery of Time Series Motifs*, SIGKDD (2003).
8. D. Dasgupta and S. Forrest, *Novelty Detection in Time Series Data using Ideas from Immunology*, 5th Int. Conf. on Intelligent Systems, (1996).
9. B. Farrell and S. Santuro, *NASA Shuttle Valve Data*, http://www.cs.fit.edu/∼pkc/nasa/data/ (2005).
10. M. Jones and D. Nikovski and M. Imamura and T. Hirata, *Anomaly Detection in Real-Valued Multi-dimensional Time Series*, Proceedings of the 2nd International ASE Conference on Big Data Science and Computing (2014).
11. E. Keogh, and J. Lin, and A. Fu, *HOT SAX: Finding the Most Unusual Time Series Subsequence: Algorithms and Applications*, ICDM (2005).
12. E. Keogh (2005). www.cs.ucr.edu/∼eamonn/discords/
13. E. Koskivaara, *Artificial Neural Network Models for Predicting Patterns in Auditing Monthly Balances*, J. of the Operational Research Soc., Vol. 51 (1996).
14. J. Lin, and E. Keogh, and S. Lonardi, and P. Patel, *Finding Motifs in Time Series*, SIGKDD (2002).
15. B. Liu, H. Chen, A. Sharma, G. Jiang, and H. Xiong, *Modeling Heterogeneous Time Series Dynamics to Profile Big Sensor Data in Complex Physical Systems*, IEEE Int. Conf. on Big Data (2013).
16. J. Ma and S. Perkins, *Online Novelty Detection on Temporal Sequences*, SIGKDD (2003).
17. M. Mahoney and P. Chan, *Trajectory Boundary Modeling of Time Series for Anomaly Detection*, Workshop on Data Mining Methods for Anomaly Detection at KDD (2005).
18. T. Rakthanmanon, B. Campana, A. Mueen, G. Batista, B. Westover, Q. Zhu, J. Zakaria, and E. Keogh, *Searching and Mining Trillions of Time Series Subsequences under Dynamic Time Warping*, KDD (2012).
19. A. Rusiecki, *Robust neural network for novelty detection on data streams*, Int. Conf. on Artificial Intelligence and Soft Computing (ICAISC) (2012).
20. C. Shahabi, X. Tian, and W. Zhao, *TSA-tree: A Wavelet-Based Approach to Improve the Efficiency of Multi-Level Surprise and Trend Queries on Time-Series Data*, 12th International Conf. on Scientific and Statistical Database Management (SSDBM), (2000).
21. A. Ypma and R. Duin, *Novelty detection using Self-Organizing Maps*, In Proc. of ICONIP (1997).