MERLIN: Parameter-Free Discovery of Arbitrary Length Anomalies in Massive Time Series Archives

Takaaki Nakamura[1] Makoto Imamura[2] Ryan Mercer[3] Eamonn Keogh[3]
[1]Mitsubishi Electric Corporation, Japan [2]Tokai University, Japan [3]University of California Riverside, CA USA
nakamura.takaaki@dy.mitsubishielectric.co.jp imamura@tsc.u-tokai.ac.jp {rmerc002,eamonn}@cs.ucr.edu

***Abstract*—Time series anomaly detection remains a perennially important research topic. If anything, it is a task that has become increasingly important in the burgeoning age of IoT. While there are hundreds of anomaly detection methods in the literature, one definition, *time series discords*, has emerged as a competitive and popular choice for practitioners. Time series discords are subsequences of a time series that are maximally far away from their nearest neighbors. Perhaps the most attractive feature of discords is their simplicity. Unlike many parameter laden methods, discords require only a single parameter to be set by the user: the subsequence length. In this work we argue that the utility of discords is reduced by sensitivity to this single user choice. The obvious solution to this problem, computing discords of all lengths then selecting the best anomalies (under some measure), seems to be computationally untenable. However, in this work we introduce MERLIN, an algorithm that can efficiently and exactly find discords of all lengths in massive time series archives. We demonstrate the utility of our ideas on a large and diverse set of experiments and show that MERLIN can discover subtle anomalies that defy existing algorithms or even careful human inspection. Moreover, we show how to exploit computational redundancies to make MERLIN two orders of magnitude faster than comparable algorithms.**

Keywords—Time Series; Anomaly detection; Multi-Scale

I. INTRODUCTION

Time series data is ubiquitous in industrial, medical and scientific settings. One of the most basic time series analytical tasks is to simply spot anomalous regions. This may be the end goal of the analytics, or simply a preprocessing step for a downstream task. There are at least hundreds of algorithms for finding anomalies, but which should we use?

Since their introduction, *Time Series Discords* have emerged as a competitive approach for discovering anomalies [5]. For example, a team lead by Vipin Kumar conducted an extensive empirical comparison concluding that "*on 19 different publicly available data sets, comparing 9 different techniques* (time series discords) *is the best overall.*" [3]. We attribute much of this success to the simplicity of the definition. Time series discords are intuitively defined as the subsequences of a time series that are maximally far away from their nearest neighbors. This definition only requires a single user specified parameter, the subsequence length. With only a single parameter to set[1], it is harder to overfit the anomaly definition, and overfitting seems to be a major source of false positives for this task [3][16].

To help the reader appreciate the importance of the subsequence length in anomaly discovery, let us consider an excerpt of the Gasoil Plant Heating Loop Data Set [4]. This data set had a simulated cyber-attack introduced at the time indicated by the red dashed line shown in Figure 1.

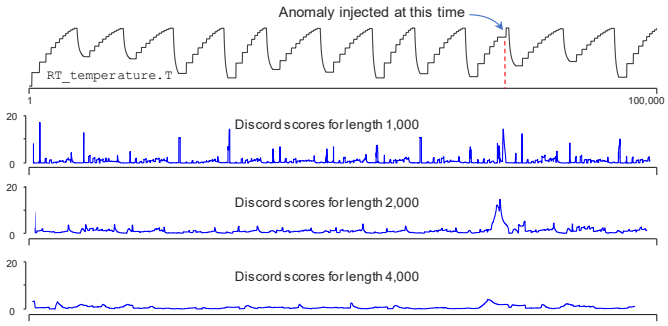


Figure 1: *top*) An excerpt from Filonov's Gasoil dataset, a reading from `RT_temp.T` [4]. *bottom*) The discord scores for three lengths, 1,000, 2,000 and 4,000. The higher the score, the more anomalous the corresponding subsequence is.

We computed the anomaly scores for every subsequence for three different lengths. For the shortest length of 1,000, it is unsurprising that we get many spurious anomalies. This system transitions between discrete temperature states, giving it a "staircase" effect. If the subsequence length is less than the length of a step, the z-normalization "blows up" the subsequence and produces unstable results. At the longer length of 4,000 the curse of dimensionality is beginning to dominate. As noted by Beyer et. al. "*as dimensionality increases, the distance to the nearest data point approaches the distance to the farthest data point*" in [2].

However, consider the plot for subsequences of length 2,000 shown in Figure 1. There is a clear peak at the correct location. Moreover, it is significantly larger than the mean value of the scores, giving a clear visual signal that this is a true anomaly. This example shows that there is a "sweet spot" (or rather, sweet *range*) for subsequence length when performing anomaly discovery. In some cases, the analyst may have a first-principles-model or experience to suggest a good value, but anomaly/novelty discovery is often exploratory by nature.

Before continuing, we will take the time to reiterate the utility of discord discovery in the vast space of anomaly detection techniques [3][4][8][9][11][12][15][16][17][18][20]. In essence, we want to answer the following question: "*why*

[1] Note that some *algorithms* that discover discords may have other parameters, the discord representation itself requires just a single parameter.

make an effort to address the noted weakness of discords, rather than invent or use a different method?" Figure 2 shows the discord scores computed for a benchmark dataset that has been considered in dozens of research efforts [17].

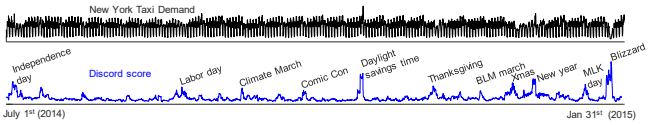


Figure 2: *top*) Six months of taxi demand in New York City. *bottom*) The discord scores for subsequence length of one day. Most of the discords discovered have an intuitive meaning.

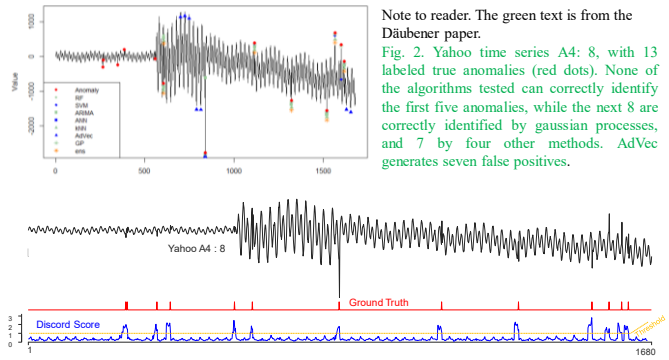
Note that the discords discovered have different causes. Some are predicable holidays, some are caused by ad-hoc events, like the hastily organized BLM march, and some are weather events. One anomaly is simply a bookkeeping error; setting the clock back by one hour for daylight saving time made it appear as if the taxi demand doubled just after midnight.

Paper [9] also considers this dataset. While they find some true positives, they also find many false positives. However, they tell us that "*the parameters for this experiment are w = 30, k = 6, q = 5, h_1 = −3.57, and h_2 = −4.28*." Similarly, there are many research efforts on deep learning anomaly detection. One recent paper using an LSTM model also considers this taxi dataset [10]. It *does* find Xmas, New Year, and the blizzard, but fails to find Thanksgiving, the BLM march, or the (apparent even to the human eye) daylight-savings-time anomaly.

These two comparisons highlight the attractiveness of discords for practitioners. It is hard to imagine that most practitioners would be able and willing to carefully set the five parameters of Markov Chain approach [9], or the dozen or so parameters/choices for a LSTM model [16]. Moreover, even if they did so, with so many parameters to fit on a small dataset, avoiding overfitting would be very challenging.

Because the effectiveness of discords is central to our work, we will take the time to consider just one more motivating example. A recent paper conducted a "bake-off" with eight *diverse* representatives of the state-of-the-art anomaly detection algorithms (as opposed to simply minor variants of a single approach) [11]. Figure 3 contrasts the results on one benchmark (Yahoo) data set with time series discords.

The authors of this study noted, "*None of the algorithms tested can correctly identify the first five anomalies,... AdVec generates seven false positives...*" In contrast to these eight approaches, the discord approach performs perfectly on this task, assuming only that its one parameter is a reasonable value. The goal of this research effort is to remove the need to set *even* that sole parameter. We call our proposed algorithm MERLIN[2]. MERLIN can efficiently and exactly discover discords of every possible length, then either report all of them, or just the top-K-discords under an arbitrary user defined scoring metric.



Note to reader. The green text is from the Däubener paper.
Fig. 2. Yahoo time series A4: 8, with 13 labeled true anomalies (red dots). None of the algorithms tested can correctly identify the first five anomalies, while the next 8 are correctly identified by gaussian processes, and 7 by four other methods. AdVec generates seven false positives.

Figure 3: *top*) A screen capture from [11] showing the performance of eight state-of-the-art anomaly detectors on one of the Yahoo benchmarks [8]. *bottom*) Time series discords (here, of length 8) have a perfect score on this problem, with only the mildest of assumptions.

The rest of this paper is organized as follows. In Section II we introduce background material and related work. Section III introduces our proposed algorithm, before we offer an extensive empirical evaluation in Section IV. We conclude with a discussion of our findings in Section V.

II. Background and Related Work

In this section, we introduce all the necessary definitions and notations, including a review of an existing algorithm for discord discovery that we will use as a starting point for our research. We will also consider related work to put our ideas in context [7].

A. Definitions

We begin by defining the data type of interest *Time Series*:

Definition 1: A *time series* $\mathbf{T} = t_1, t_2, \ldots, t_n$ is a sequence of n real values.

Our distance measures quantify the distance between two time series based on local subsections called *subsequences*:

Definition 2: A *subsequence* $\mathbf{T}_{i,L}$ is a contiguous subset of values with length L starting from position i in time series $\mathbf{T}$; the subsequence $\mathbf{T}_{i,L}$ is in form $\mathbf{T}_{i,L} = t_i, t_{i+1}, \ldots, t_{i+L-1}$, where $(1 \le i \le n - L + 1)$ and L is a user-defined subsequence length with value in range of $3 \le L \le |\mathbf{T}|$.

Here we allow L to be as short as three, although that value is pathologically short for almost any domains.

Many time series analytical algorithms need to compare subsequences using some distance measure *Dist;* here we use the z-normalized Euclidean distance. As pointed out by the original authors of the discord definition, we must be careful to exclude certain trivial matches from any meaningful definitions of subsequence similarity by defining *non-self matches*.

Definition 3: *Non-Self Match*: Given a time series $\mathbf{T}$ containing a subsequence C of length L beginning at position p and a matching subsequence M beginning at q, we say that

[2] This name is a play on the fact that the first paper on time series discords was titled "Approximations to Magic" [5]. Merlin was the magician of the Arthurian legend. In addition, Mitsubishi Electric Corporation's subsidiary in the USA is called MERL (Mitsubishi Electric Research Laboratory, Boston).

M is a non-self match to C at distance of Dist(M,C) if $|p - q| \geq L$.

We can now use this definition of non-self matches to define *time series discords*:

Definition 4: *Time Series Discord*: Given a time series **T**, the subsequence D of length L beginning at position i is said to be the discord of **T** if D has the largest distance to its nearest non-self match. That is, ∀ subsequences C of **T**, non-self match M_D of D, and non-self match M_C of C, min(*Dist*(D, M_D)) > min(*Dist*(C, M_C).

The starting location of the discord is recorded in `index` and its distance to its nearest neighbor is record in `distance`. All previous efforts to find discords considered only a single length, however we plan to consider all lengths in a given range, thus producing an array of discords indexed by the length i, $\texttt{discord}_i = [\texttt{index}_i, \texttt{distance}_i]$.

For simplicity, we define only the top-1 discord, the generalization to top-K is trivial [7]. Having defined discords, we will next review an algorithm to discover them.

B. A Review of the SOTA Discord Discovery Algorithm

Our proposed algorithm makes repeated use of the discord discovery algorithm introduced in [7]. The algorithm was unnamed in that work, so for clarity we will call it DRAG, which is both a truncated version of the inventor's name, and a backronym that stands for Discord Range Aware Gathering.

For any user-given length, the algorithm requires a single input parameter r. This value should ideally be set such that it a little less than the *discord distance*, that is, the distance between the discord and its nearest neighbor. Of course, that distance is unknown at this point, so the user must provide an estimate. If this estimate is accurate, just a little less than the eventually discovered true discord value, then DRAG has a time and space complexity of just O(nL). If the estimate is much too small, the algorithm will give the correct result, but have a time and space complexity of O(n^2). In either case, we call any invocation of DRAG that used an r value less than the eventually returned discord distance a *success*.

In contrast, if the estimate for r is too large, the algorithm will return null, a situation we denote as a *failure*. Of course, the situation can be remedied, but requires the user to reduce the r value and try again. This sensitivity to r parameter was largely glossed over in the original paper [7], but as we will show in Section III it is a significant limitation of DRAG. However, as we will later explain, we have solved this issue for MERLIN.

We refer the reader to [7] for a detailed explanation of the DRAG algorithm, but for completeness we will give a brief overview. The DRAG algorithm is a two-phase algorithm, with each phase being a pass across the time series.

- **Phase I**: As shown in Table 1 the algorithm initializes a set C, of candidate discords by placing the first subsequence in C. The algorithm then "slides" along the time series examining each subsequence. If the subsequence currently under consideration is greater than r from any item in the set, then it may be the discord, so it is added to the set. However, if any items in the set C are less than r from the subsequence under consideration, we know that they could not be discords, thus they are admissibly pruned from the set. At the end of Phase I, the set C is guaranteed to contain the true discord, possibly with some additional false positives.

Table 1: Phase I, Candidate Selection Algorithm

```
Input:     T:     Time series
           L:     Subsequence length
           r:     Range of discords
Output:    C:     Candidate set of discords
1   C = {}                          // Start with empty set
2   For I = 1 to |T| - L + 1        // Scan all subsequences
3       iscandidate = true
4       For j in C
5           If I and j are NOT trivial matches
6               If dist(Ti,L, Tj,L) < r
7                   C = C \ j
8                   iscandidate = false // We can prune this
9       If iscandidate
10          C = C ∪ {i}             // Add to candidate set
11  If NOT isemptyset()
12          return C                // Implicitly return success
13  Else
14          return failure          // Explicitly return failure
```

Note that the algorithm can end in *failure* (line 14). Or, we can regard this situation as successfully finding no discord greater that the threshold of r. If the user wants the find the discord regardless of its eventual distance, she must run the algorithm again with a smaller value for r. We will have more to say about this issue in Section III.*a*.

After Phase I has built a set of candidate discords, we are now ready to run Phase II to refine them.

- **Phase II**: As shown in Table 2, we again slide along the time series, this time refining the candidates to remove the false positives. We simply consider each subsequence's distance to every member of our set, doing a *best-so-far* search for each candidate's nearest neighbor. The algorithm returns a sorted list of all discords with a distance greater than r (there is guaranteed to be at least one). The largest such score is our top-1 discord.

Table 2: Phase II, Discords Refinement Algorithm

```
Input:     C:     Candidate set of discord
           T:     Time series
           L:     Subsequence length
           r:     Range of discords
Output:    D:     Set of discords (index, distance)
1   D = {}                              // Start with empty set
2   For i = 1 to |T| - L + 1            // Scan all subsequences
3       isdiscord = true
4       For j in C                      // Scan all candidates
5           If i and j are NOT trivial matches
6               d = dist(Ti,L, Tj,L)
7               If d < r
8                   C = C \ j
9                   isdiscord = false // Eliminate candidate
10              else
11                  dj = min(dj, d)
12      If isdiscord
13          D = D ∪ {(j, L, dj)}    // Add to the set of true..
14  return D                            // ..discord and return
```

Given this (brief) review of the algorithm, it is easy to see why its performance depends so critically on the user's choice of r. A pessimistically small value for r will mean that in Phase I

most subsequences will be added to the candidate set, exploding the time and space complexity to the $O(n^2)$ case. However, if r is chosen well, the size of this set remains very small relative to n. For example, in [7] they show that even with a million subsequences, for a *good* value of r, the size of C does not exceed 50 candidates, making the algorithm effectively $O(nL)$.

C. Related Work

In the previous section we claimed DRAG is the state-of-the-art in *discord* discovery (we are not (yet) claiming state-of-the-art in *anomaly* detection). The reader may be surprised to find that we did not list the more recent Matrix Profile (MP) algorithms as state-of-the-art [6]. The MP algorithms (STOMP/SCRIMP etc.) surely are state-of-the-art for *motif discovery*, and as a side-effect of motif discovery, they happen to also compute discords. However, the MP algorithms are all $O(n^2)$. It is impressive that their time complexity is independent of L, as almost all algorithms in this space scale poorly with L. Nevertheless, for our purposes these algorithms compute much more information than is needed, and are thus much slower than we can achieve for the limited task-at-hand.

There are also algorithms that discover discords by discretizing the time series, typically using SAX, and hashing the symbolic words that correspond to subsequences [5][18]. The basic idea being that a lack of collision for a word is evidence that the word might be unique, hence correspond to a discord. After the candidates have been identified this way, an algorithm similar to Phase II in Table 2 can be used to refine them. These algorithms *can* be competitive with DRAG, but only if three parameters for SAX are very carefully set [18].

The more general area of anomaly detection is increasingly difficult to review. In particular there has been a recent explosion of papers on deep learning for anomaly detection [4][9][10][11][16][17][20]. This is a diverse group of research efforts; the one thing that they have in common from our point of view is that they all require many critical parameters to be set[3]. For example, Paper [9] explicitly lists five parameters (and perhaps has a few more in the background), the LSTM network in [16] requires eight parameters. Clearly deep learning has had an enormous impact in image processing, NLP etc. However, as we hinted at in Figure 2 and Figure 3, and as we will later empirically show, it is not obvious that deep learning out performs simpler and more direct shape based methods.

III. THE MERLIN ALGORITHM

We begin by illustrating some novel observations about the sensitivity of DRAG to the r parameter.

A. Exploitible Obervations about DRAG

Consider the small synthetic dataset shown in Figure 4: it is simply a slightly noisy sine wave with an obvious "anomaly" embedded in it starting at location 1,000.

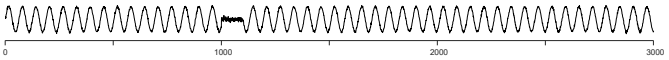


Figure 4: A slightly noisy sine wave with an anomaly embedded at location 1,000.

What would be an appropriate value of r here given that we wish to discover discords of length 512? Even with significant experience with the DRAG algorithm, it is not immediately obvious to us. To gain some intuition, in Figure 5, we considered every possible value of r from 1 to 40, in increments of 0.25, measuring both how long DRAG takes, and whether it ended in *success* or *failure*.

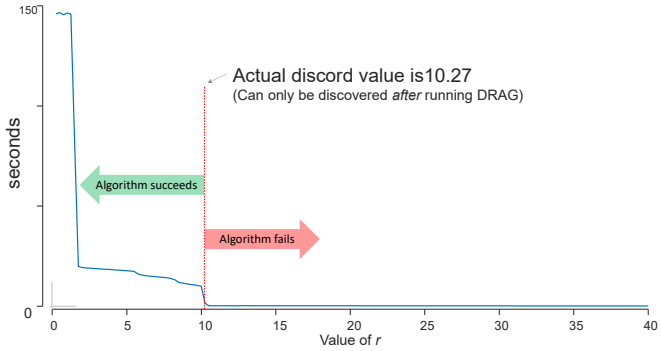


Figure 5: The time taken for DRAG given values for r that range from 1.0 to 40.0. For any value greater than 10.27 the algorithm reports *failure* and must be restarted with a lower value.

After the fact, we know that the true discord value is 10.27. The reader will appreciate that this value, or rather, this value minus a tiny epsilon, is the optimal setting of r [7].

Suppose that we had guessed $r = 10.25$, then DRAG would have taken 1.48 seconds to find the discord. However, had we guessed a value that was just 2.5% less, DRAG would have taken 9.7 times longer. Has we guessed $r = 1.0$ (a perfectly reasonable value on visually similar data), DRAG would have taken 98.9 times longer.

In the other direction, had we guessed any greater than 1% more, DRAG would have *failed*. The time it takes to complete a failed run is about 1/6 the time of our successful run when $r =$ was set to the 10.25 guess. So, while failure is cheaper, it is not free. This eliminates certain obvious algorithms to find a good value for r. For example, we could have tried every integer from 40 downwards until *success*, but that would have cost 29 *time-for-failures* plus one *time-for-success* with $r = 10$, which is about 39.2 seconds, or about 26 times worse than our "lucky" guess of $r = 10.25$.

Note that a failure lets us know that our guess for r was too high, but otherwise does not appear to contain exploitable information as to a better value for r.

One might imagine that there is some simple heuristic for setting r. If there is, it has eluded us (and, to the best of our knowledge, the rest of the community that uses this algorithm [3]). Even on datasets that are superficially similar to each

[3] Note, we distinguish between parameters that merely effect the speed or memory footprint, vs. parameters that can change the anomalies discovered. It is the latter we call "critical".

other, say two examples of ten minutes of healthy teenage female electrocardiograms, the best value for *r* can differ by at least two orders of magnitude.

In summary, choosing a good value for *r* is critical for DRAG to be efficient, but it is a very difficult parameter to set. However, for our task-at-hand, there is a ray of hope. The best value for *r*, for discords of length *L*, is likely to be very similar to the best value for *r*, for discords of length *L*-1. To see this, we measured the correlation between the optimal *r* for discords with lengths differing by one, for all *L* from 16 to 512 for the example shown in Figure 4. The correlation was 0.998.

It is important to ward off a possible misunderstanding, suggested by this very high correlation; these differences are typically very small, but they are not necessarily all positive. Because we are working with z-normalized Euclidean distance, when we make the discord length longer, the discord score can increase, decrease or stay the same. The blue line shown in Figure 6 illustrates this fact.

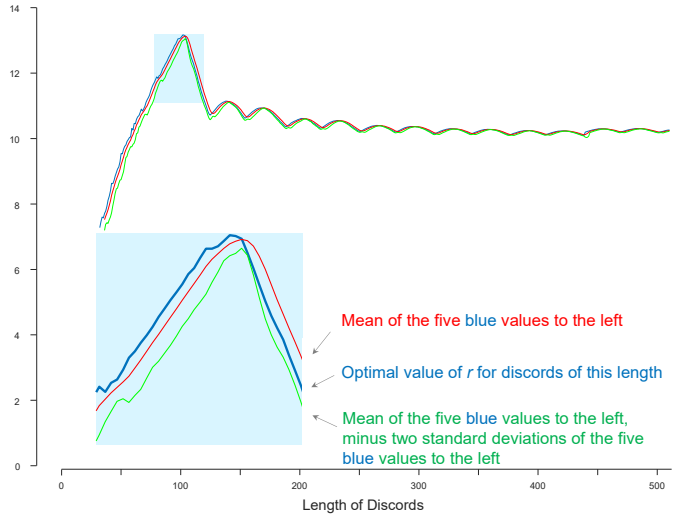


Figure 6: (*blue line*) The discord score, which is also the optimal setting for *r*, for the dataset shown in Figure 4. The inset shows a zoom-in of the region from 64 to 100. Here we can more clearly see the blue line is accompanied by a red line, which attempts to predict it, using only the five previous values.

As Figure 6 makes clear, the obvious idea of using the last discord$_i$ distance to set the value for *r* when attempting to discover discord$_{i+1}$ is a bad idea. In this example, it would result in 45.4% of the runs ending in *failure*. Thus, we want the value of *r* to be a "little less" that discord distance. The meaning of "little less" here depends on the data and on the lengths currently considered, so we propose to learn it by looking at the variance of the last few (say five) discord values.

Thus, we have an informal algorithm to set the value of *r*.

Compute the discords working from the minimum to the maximum length. At each stage, compute the mean μ, and standard deviation σ, of the last five discord distances, and for the next invocation of DRAG, use $r = \mu - 2\sigma$. If DRAG reports *failure*, repeatedly subtract another σ from the current value of *r* until it reports *success*.

Using this simple prediction algorithm on the dataset shown in Figure 4, we would have zero failures. Moreover, on average, the value predicted would be 99.03% of the optimal value for *r*.

This idea leaves just one thing unspecified. How do we set *r* for the first five discord lengths? We do have an *upper* bound as to the largest possible discord distance for time series of length *L*, it is simply the largest possible distance between any pair of subsequences of length *L*, which is $2\sqrt{L}$. So, for the first length of discord we attempt to discover, we can set $r = 2\sqrt{L}$, and keep halving it until we get a *success*. In general, $2\sqrt{L}$ is a very weak bound, and likely to produce many failures. So, we do not want to do this for the next four items. Here instead, we can use the previous discord distance, minus an epsilon, say 1%. In the very unlikely event that this was too conservative and resulted in a failure, we can keep subtracting an additional 1% until we get a *success*.

Table 3 formalizes this algorithm.

Table 3: The MERLIN Algorithm

```
Input:    T:     Time series
          MinL:  Subsequence length lower bound
          MaxL:  Subsequence length upper bound
Output:   D:     Set of discords (index, length, distance)

r = 2 × sqrt(MinL)     // Set r to its largest possible value
distance_minL = -inf   // Allow entry into loop
While distance_minL < 0 // Find first discord
  [index_minL , distance_minL]= DRAG(T, MinL,r)
  r = r × ½            // If loop repeats, make r smaller
End
For i = MinL + 1 to MinL + 4 // Find next 4 discords
   distance_i = -inf           // Allow entry into loop
    While distance_i < 0       // Decrease r till success
     r = 0.99 × distance_i-1
     [index_i, distance_i]= DRAG(T,i,r)
     r = r × 0.99 // If loop repeats, make r a little smaller
    End
End
For i = MinL + 5 to MaxL   // Find all remaining discords
  M = mean(distance_i-1 to i-5) // Use local info about..
  S =  STD(distance_i-1 to i-5) // ..the mean and STD..
  r = M - (2×S)            // ..to predict good value for r
  [index_i, distance_i]= DRAG(T,i,r)
  While distance_i < 0     // looks like our r was too high..
     [index_i, distance_i]= DRAG(T,i,r) //..so lets reduce..
     r = r - S                        // ..it until success
   End
End
```

The algorithm has an apparently arbitrary choice. Why work from the minimum to the maximum length, rather than the other way around? Recall that is it only for the first invocation of DRAG that we are completely uncertain about a good value for *r*, and we may have multiple failure runs and/or invoke DRAG with too small of a value for *r*, making it run slow. It is much faster to do this single unoptimized run on the shorter subsequence lengths.

B. *Defeating MERLIN*

There are two circumstances where MERLIN can dramatically fail. Fortunately, there are trivial fixes.

If there is a constant region longer than MinL, then our attempt to z-normalize before computing the Euclidean distance will divide by zero. However, it is trivial to monitor for and report or ignore such regions. Depending on user choice, such regions may warrant flagging as an anomaly or

not. For example, in hospital settings the data is replete with constant regions, due to disconnection artifacts during bed transfers etc. In contrast, a constant region in an insertable cardiac monitor (pacemaker) is almost certainly battery failure or heart-failure, in either case warranting an alarm.

Another way MERLIN could fail is if the anomaly happens twice, and essentially looks the same both times. This has been called the "twin freak" problem. This can be solved by changing the first nearest neighbor (Definition 4) to the k^{th} nearest neighbor. However, in practice this rarely seems to be an issue. For example a paper that wanted to show an anomaly detection method that was invariant to "twin freaks" had to resort to copying and pasting data to contrive the situation [20]. In this work we use only the simple first nearest neighbor.

IV. Experimental Evaluation

We begin by stating our experimental philosophy. We have designed all experiments such that they are easily reproducible. To this end, we have built a webpage that contains all datasets, code and random number seeds used in this work, together with spreadsheets which contain the raw numbers [23]. This philosophy extends to all the examples in the previous section.

A. *Metrics of Success and the Unsuitability of Benchmarks*

There are now a handful of benchmark datasets in the literature. We have already considered (a subset) of them in Figure 1, Figure 2, Figure 3 and Figure 10, and we will consider more below. However, we believe that the reader should be somewhat skeptical of research efforts that report *only* summary statistics on these datasets. There are at least two reasons for such skepticism.

- Consider the NYC Taxi example which is part of the NAB benchmark [17]. This dataset is labeled as having five anomalies, but as Figure 2 shows, this dataset has at least twice that number of anomalies. For example, the benchmark does not list the daylight-saving time anomaly, which is arguably the most visually jarring anomaly in the dataset. Any algorithm that does find this anomaly will be penalized as having produced a false positive. In [23] we show more examples of mislabeled benchmark data.

- A large fraction of the benchmark datasets contain anomalies that are so obvious that they are trivial to detect. For example, consider Figure 7, which show examples from the Mars Science Laboratory [16], NAB [17] and Yahoo benchmarks. It is hard to imagine any reasonable algorithm failing to find such anomalies. Even if the benchmark data *also* includes some challenging anomalies, counting success on these trivial problems can artificially inflate metrics of success such as ROI curves, giving the illusion of progress. See Appendix A for more information and examples.

For the reasons above, we think that a direct *visual* summary of the output of a proposed anomaly detection algorithm on diverse datasets can offer the reader the most forceful summary of the algorithm's strengths and weaknesses (although we must be careful to avoid attempting "*proof-by-anecdote*"). For that reason, we have chosen to show fifteen diverse examples below.

It is important to note that our discussion some issues with the benchmark datasets should *in no way* be interpreted as criticism. These groups have spent tremendous time and effort to make a resource available to the entire community and should rightly be commended. It is simply that we must be aware of the limitations of metrics reported on them without visual context.

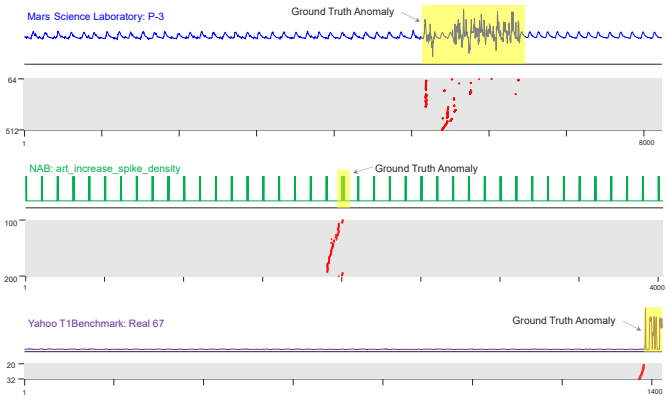


Figure 7: Examples from the three main anomaly benchmark datasets that we regard as too simple to be informative for algorithm comparison. *top*) From the NASA benchmark [16]. *center*) From the NAB benchmark [17]. *bottom*) From the Yahoo Benchmark.

As such, we have endeavored to have many such examples in this work. In particular, before performing conventional experiments to compare MERLIN to the state-of-the-art, we begin with some case studies that give the reader an appreciation of the kind of subtle anomalies that MERLIN can discover.

B. *Discovery of Ultra Subtle Anomalies*

Virtually all anomaly detection benchmarks in the literature contain anomalies that also yield to casual visual inspection. Of course, this does not mean that algorithms that can detect such anomalies are of no utility. Human inspection, especially at scale, is expensive. Nevertheless, it is interesting to ask if we can detect *very* subtle anomalies, that would defy human inspection. However, this seems to beg the question, how can we know if a time series contains such ultra-subtle anomalies?

We propose the follow experiment to allow us to obtain ground-truth subtle anomalies. Consider Figure 8, which shows the electrocardiogram (ECG) of a 51-year old male, with an obvious anomaly at about the half-way point. The anomaly is so obvious that surely *any* algorithm could discover it.

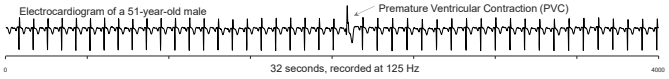


Figure 8: An ECG signal with an obvious anomaly (a PVC).

However, suppose we consider *only* the Central Venous Pressure (CVP) data, which was recorded in parallel. The ECG is an electrical signal, whereas the CVP is a mechanical signal, the blood pressure in the *venae cavae*. Moreover, because the CVP reflects the amount of blood returning to the heart, the elasticity of the blood vessels tends to dampen out any irregularities in the heartbeat. As Figure 9 shows, the PVC anomaly is not visually apparent in the CVP, yet MERLIN clearly indicates at the correct location.

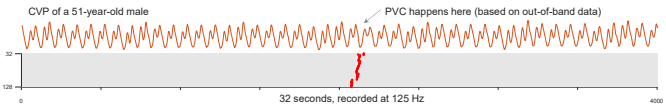


Figure 9: A CVP signal recorded in parallel with the ECG shown in Figure 8 does not show visual evidence of an obvious anomaly caused by the PVC, yet MERLIN clearly indicates its presence.

Note that our inability to see the anomaly in Figure 9 shows should not be attributed to the small size of the figure (the reader is invited to see a larger reproduction here [23]) or our lack of medical experience. Dr. Greg Mason with almost forty years of experience viewing such data, could not detect this anomaly.

To see that this was not pure luck, let us consider a dataset from a totally different domain with a similarly subtle anomaly. In Figure 10 we show a snippet of data from the Mars Science Laboratory (MSL) rover, Curiosity [16].

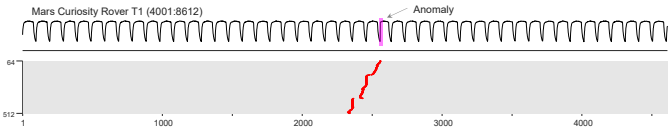


Figure 10: A signal from the Mars Curiosity rover was annotated as having an anomaly from 2550 to 2585 (pink bar) [16]. While the cause of the anomaly is unclear, MERLIN has no difficulty finding it.

In the paper that introduced this dataset, the authors introduced a LSTM network that could also find this anomaly [16]. However, to do so, they required training data, and the careful setting of *eight* parameters. In contrast, MERLIN finds this subtle anomaly with no training data, and only the weakest of hints as to the anomaly lengths (*MinL* and *MaxL*) to consider.

C. *Anomalies at Different Scales.*

In this section we anecdotally demonstrate the utility of being able to discover multiscale anomalies. We simply wish to show that anomalies that differ by at least an order of magnitude can exist even in quotidian datasets.

We begin by revisiting the NYC Taxi demand dataset shown in Figure 2. In Figure 11 we show a subset of the data, with just the top-1 motif of every length from 5 hours to four days.

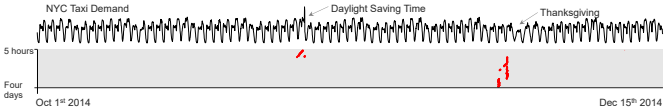


Figure 11: As subset of the Taxi demand dataset shown in Figure 2, shown with all discords the range of 5 to 96 hours.

While the daylight-saving anomaly directly effects only two hours, the shape of these two hours is only usual in the context of the few hours that surround them. Similarly, while Thanksgiving is somewhat unusual in its lower passenger volume and lack of a rush hour peak of people leaving the city after work, a somewhat similar pattern to this also happens on the weekends. However, in the context of being surrounded by normal days, Thanksgiving *is* unusual. The discords of up to four days long discovered by MERLIN in Figure 11 reflect this.

We also considered a similar but much longer dataset of passenger volume at the Taipei Xinjian District Office metro station. We searched from ten hours to ten days. Over this enormous range of scales, only seven distinct anomalies are discovered, Figure 12.*bottom* shows four of them. Note that some of the anomalies have natural causes (weather events), and some are cultural artifacts such as Chinese New Year.

Before leaving this section, we considered some final examples in this vein. As hinted at in Figure 13.*top,* the city of Melbourne has released almost a decade's worth of pedestrian traffic volume from various sites in the city [13].

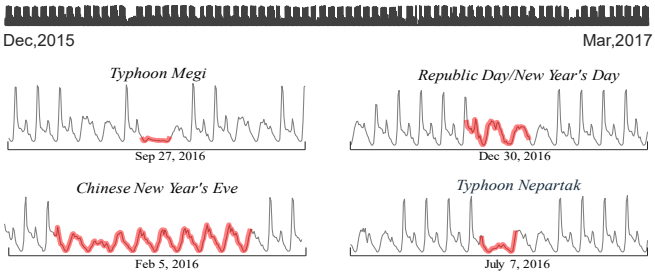


Figure 12: *top*) Passenger volume at a Taipei metro station. Four of the anomalies discovered are shown in context.

While there is good spatial coverage, the temporal resolution is very low at just one datapoint per hour. Because of this, like most of the many research groups that explored this data resource, we originally only searched for anomalies of length days or weeks [15]. However, as Figure 13, hints at, using MERLIN to free ourselves from assumptions about possible anomaly duration allowed us to find unexpectedly short anomalies.

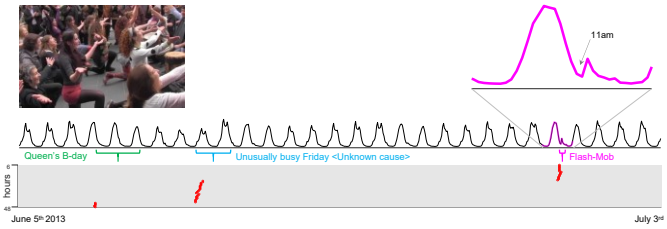


Figure 13: *bottom*) A month of pedestrian traffic volume on Bourke Street Mall in Melbourne. *top*) the shortest anomaly discovered is semantically meaningful, it corresponds to a flash-mob dance performance (video at [14]) that restricted traffic for about ten minutes.

Given this ability to find motifs at all scales, we begin to find unexpected anomalies everywhere. Three years after the flash-mob happened, we discovered another short and subtle anomaly on the same street. With a little investigation we realized it corresponded to a car attack in which an individual deliberately drove at pedestrians, killing six and injuring twenty seven [21].

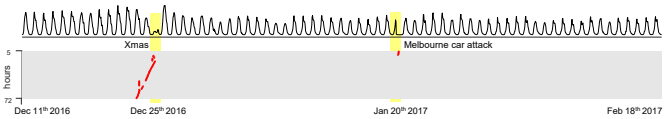


Figure 14: Two months of pedestrian traffic volume on Bourke Street Mall in Melbourne. The anomaly for Xmas is to be expected, but what caused the short anomaly on Jan-20-2017?

D. *Scalability*

To demonstrate the scalability of our algorithm, we compare it to the Matrix Profile algorithm SCRIMP [6]. In a sense, this is unfair to SCRIMP, which discovers not only the discords, but

also motifs. Nevertheless, it is a very scalable algorithm because it is implemented in a way that makes it constant in the length of the subsequences. We used the latest version of the code available from the author's website [6], disabling the GUI interface, which required significant time overhead.

We also wish to test the effectiveness of our method to set the value of r for MERLIN by sharing information across different values of L. To do this, we implemented the method for setting r suggested in [7], which we rerun for every value of L. This algorithm is denoted as DRAG-multilength, or $DRAG_{ML}$. Note that $DRAG_{ML}$ differs from MERLIN *only* in how r is set.

The time needed for SCRIMP is independent of the data. However, the time needed for the two other algorithms depends on the data. The best case would be a dataset like the one shown in Figure 7.*top*, a mostly repetitive time series with a dramatic discord that is very far from its nearest neighbor. To avoid such bias, we will use the worst-case dataset from MERLIN, *random walk*. For such data, the top-1 discord is only slightly further away from its nearest neighbor than any randomly chosen subsequence, meaning that the candidate set built in Table 1 grows relatively large even if given a good value for r.

Note that STOMPs performance is independent of the structure of the data, but the other algorithm's performance does (weakly) depend on the it, we averaged over ten runs. Figure 15 shows the results of datasets ranging for 2^{12} to 2^{16}.

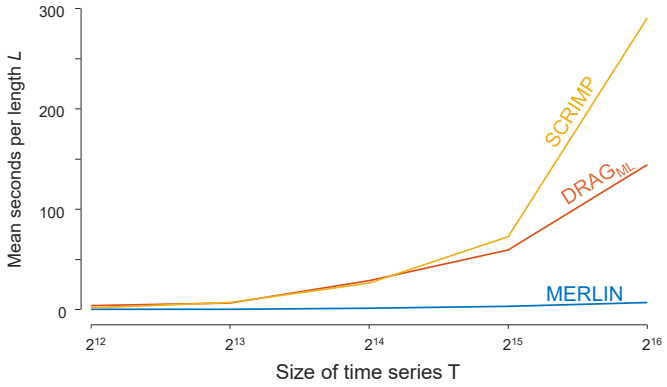


Figure 15: The scalability of MERLIN, DRAG and STOMP in the face of increasingly large datasets.

For short time series, all algorithms perform similarly, but as the time series grow longer, SCRIMPs quadratically complexity begins to show. While MERLINs *first* run (for L = MinL) is no faster that DRAG, its subsequent runs are greatly accelerated by the predicted value of r, and the amortized cost is about 21 times faster by the time we consider time series of length 2^{16}. To put these numbers in context, 2^{16}, datapoints is about 18 minutes of data recorded at 60 Hz. Suppose we suspected that there were anomalies of length 1 second in our data, but we wanted to bracket our search with every value for 30 to 90 datapoints. This would take MERLIN just 7.1 minutes, faster than "real-time".

E. *First look at the Yahoo! Webscope Benchmarks*

In recent years, the Yahoo Webscope anomaly datasets have emerged as the de-facto benchmark for anomaly and changepoint detection. This diverse archive consists of 367 time series, of various lengths in four different classes A1/A2/A3/A4 with class counts 67/100/100/100. While class A1 has real data from computational services, classes A2, A3, and A4 contain synthetic anomaly data with increasing complexity. We previously showed examples from this benchmark in Figure 3 and Figure 7.*bottom*.

Before presenting summary statistics on the entire archive, we will take the time to consider one example in detail. Because most of these datasets have multiple anomalies, this is an ideal opportunity to show the output of the top-K discords. In Figure 16.*top* we show an example with seven anomalies.

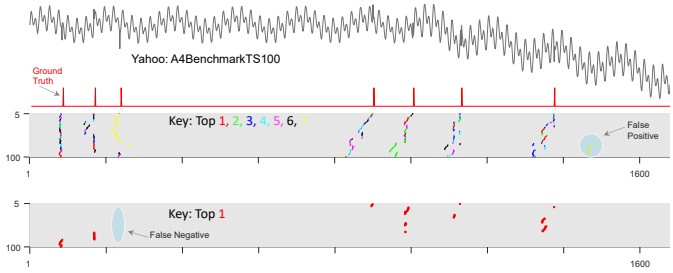


Figure 16: *top*) An example of one the synthetic datasets from the Yahoo archive with seven anomalies, whose location is marked by the red binary vector. *center*) The result of running MERLIN to discover the Top-7 anomalies. *bottom*) The result of running MERLIN to discover just the top-1 anomalies.

We know that examples in this subset have *point* anomalies, so a smaller value of MaxL would be appropriate. However, we "stress test" our algorithm by considering unreasonably long discords up to length 100. In Figure 16.*center* shows that had we consider only 5 to 64, we would have obtained perfect results. It is only when we consider MaxL for an unrealistic value of greater than 65, that we obtain a single false positive, and then only for the 7th discord. Another way to consider how effective MERLIN is here is to see how many of the seven anomalies we can detect if we only consider the *single* top-1 discord. As Figure 16.*bottom* shows we would still detect six out of seven true positives, and have no false negatives.

F. *Large Scale Resutls on the Yahoo! Webscope Benchmarks*

To evaluate on all Yahoo 367 datasets [8], we need to define some criteria for correct anomaly detection. Below we explain our reasoning behind our choice for metric of success.

Note that a complete anomaly detection system must have two parts, (I) A *prediction* of the most likely location(s) to contain anomalie(s), and (II) an *evaluation mechanism* (often simply thresholding) to determine if those locations warrant been flagged as anomalies. In this work, we have mostly avoided a discussion of the second part, as it is moot unless we can robustly point to candidate anomalies. Also note that in many real-world applications, the second part is not needed. For example, an analyst might query: "*Show me the top-five most unusual events in the oil plant in 2018*". Likewise, thresholds can often be learned with simple human-in-the-loop algorithms. In brief, the user can simply examine a sorted list of all candidate anomalies. The discord distance of the first one she rejects as "not an anomaly", can be used as the threshold for future datasets from the same domain. Thus, we argue that the first task is the most critical, and most worthy of evaluation.

Some of the Yahoo datasets have an issue that confounds evaluation. In the example shown in Figure 16, the anomalies are all well-spaced apart, however in the example shown in Figure 3 the anomalies are just two datapoints apart. It is hard to imagine critiquing an algorithm that called these two events a single anomaly. More generally, we must also consider the precision of the algorithm's prediction of location. If an anomaly is located at say location 600, we should surely reward an algorithm that predicts 599 or 602. Thus, for simplicity, we reward any prediction that is no further off than ± 1% of **T** from the stated location. This does not significantly increase the default rate while allowing us to bypass the issues above.

Given these considerations, we proposed the following metric of successes, which we believe to be fair, transparent and reproducible. Each algorithm is tasked with locating the *one* location it thinks most likely to be anomalous (We removed the handful of examples that have no claimed anomaly). If that location is within ± 1% of **T** from a ground truth anomaly, we count that prediction as a success.

We compare to the LSTM method introduced in [16], which is one of the most highly cited deep learning for anomaly detection papers in recent years. We used the authors own implementation, carefully tuning it as advised in [16]. We allow the LSTM to "cheat" by training on a subset of the test data.

For MERLIN, we set MinL = 3 (this is the minimum possible value) and the MaxL = 20, and recorded the median location of the 18 predictions as the algorithm's single prediction. This is a sub-optimal policy for us if there are two or more anomalies of around that length, but makes the evaluation simple.

Under this metric MERLIN had a recall of 80.0% and the LSTM had a recall of 58.3%. While this result is strongly in our favor, because of the data quality issues discussed above, we do not weight it as heavily as the visual evidence presented in the many visual examples shown in this work.

G. Results on the NASA Benchmarks

The NASA dataset [16] has garnered significant attention in recent years, but as Figure 7.*top* hints at, some of the tasks are trivial. In fact, that understates the case. Many of the anomalies consist of changes of variance/range by up to three orders of magnitude (examples A1, B1, D12, E7, P4, T3 etc.), and can trivially be detected by algorithms dating back to the 1950s [19] (see Appendix A for a concrete example of this).

In addition, for some the examples, the labeled anomaly region comprises up to half the data (examples A7, D2, M1, M2 etc.), meaning that a random choice would have a better than even chance of being a true positive. To bypass this issue, we scanned all the datasets for examples that were not obviously solvable by the human eye in under five seconds. Excluding near redundant examples, only three datasets passed that test, the results of running MERLIN on them is shown in Figure 17. Apart from a small region of a presumptive false positive in Figure 17.*center*, we achieve perfect results. (We say "*presumptive*" because this dataset also has a handful of labeling omission errors, we point them out at [23]). Note that the bottom examples both had *two* anomalies, which we found with just the *single* top-1 discord of various lengths.

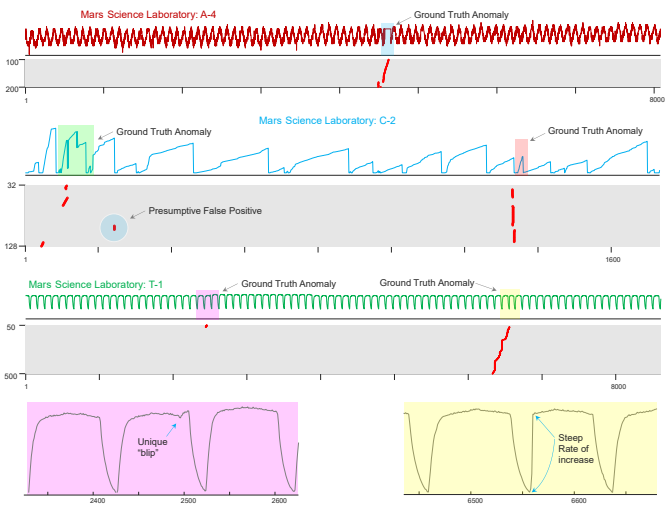


Figure 17: The results of running MERLIN on three diverse and most difficult examples from the NASA benchmark [16]. *top*) The single anomaly in A-4 is easily discovered. *center*) The two anomalies in C-2 are discovered, but there may be a short region where we report a false positive. *bottom*) The two correctly detected anomalies are so subtle that we show annotated zoom-ins to explain them.

H. Results on the Gasoil Benchmarks

Like the NASA dataset, we regard the Gasoil benchmark [4] as being too easy to be interesting. Note that we are only making this claim with regard to *anomaly detection*, it may be useful for causality detection etc. In Figure 18 we show the results of running MERLIN on two of the more challenging examples.

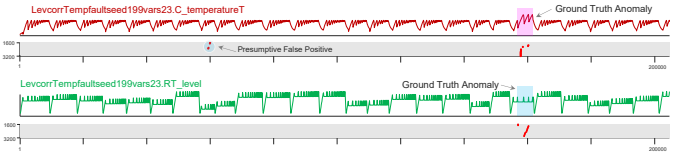


Figure 18: The results of running MERLIN on two diverse difficult examples from the Gasoil benchmark [16]. *top*) The single anomaly in TempT is easily discovered, but there may be a small region where we report a false positive. *bottom*) The single anomaly in RT_level is easily discovered.

V. Conclusions and Discussion

Ahmed and Mahmood created an influential taxonomy of anomalies into *point* anomalies, *contextual* anomalies and *collective* anomalies [22]. While we refer the reader to the original paper for the exact definitions, a review of this work shows that MERLIN was able to discovery examples of each type. For example, Figure 16 shows *point* anomalies that Yahoo embedded into a dataset. The Queen's Birthday example in Figure 13 is a classic example of a *contextual* anomaly. The shape of the day is smooth, missing the shaper features caused by typical weekday rush-hour commuting. Such days are not intrinsically rare, they happen on most weekends, but one only sees three such days in a row in the context of a three-day weekend. Finally, the anomalies shown in the Gasoil dataset in Figure 18, are classic collective anomalies. This observation is suggestive of the generality of MERLIN.

More generally, we have shown that *time series discords*, a simple, decades-old anomaly detection definition is surprisingly viable in many domains. In particular, it is *at least* competitive with the more complex deep learning methods, which require both training data and a plethora of parameters to be tuned.

Some researchers in the community had noted the utility of discords, but waived off from using them, noting "*discords are limited* (*because*) *a fixed length must be specified in advance, making it a clearly suboptimal approach for applications dealing with climate data events of varying length*" [12]. Our introduction of MERLIN removes this last barrier to adoption.

Finally, we would like to end with a note for the anomaly detection research community. In recent years there has been an explosion of deep learning work on anomaly detection, including works that introduced or evaluated the four benchmarks we consider in this work [11][16][17]. However, we feel that there is currently little evidence presented that the complexity of these approaches is warranted. Recall that for the most part we can reproduce or improve upon these results, without even looking at the training data, and using a method that is, by any reasonable standard, an order of magnitude simpler[4]. Please note that we do not doubt the utility of deep learning in general, or the ingenuity of these papers. However, we believe that the community needs to:

- Expand the list of strawmen it compares to. Perhaps half the benchmark problems can be solved by algorithms created in the 1950s [19] (See Appendix A). Simple ideas should be compared to, if the community is to justify complexity.
- Consider more challenging benchmarks.
- Directly visualize algorithm predictions on many examples, to give the reader a better appreciation of strengths and weaknesses of the proposed approach. Internally, we did this for over a dozen methods (not shown due to space limitations), and found it incredibly useful to understand when methods work, and when they fail.

There are several directions for future work, the most pressing of which are generalizing MERLIN to handle multi-dimensional data, and to handle streaming data. In addition, note that all the results shown in this work complexity ignored the training data. We plan to exploit such data, if only to learn values for MinL and MaxL.

REFERENCES

[1] A. Bagnall et al., "The great time series classification bake off: a review and experimental evaluation of recent algorithmic advances," Data Min. Knowl. Discov., 31(3), 2017, pp. 606-660.

[2] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft, "When Is Nearest Neighbor Meaningful?" ICDT, 1999, pp. 217-235.

[3] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," ACM Comput. Surv. 41(3), 2009.

[4] P. Filonov, A. Lavrentyev, and A. Vorontsov, "Multivariate industrial time series with cyber-attack simulation: Fault detection using an lstm-based predictive data model," arXiv preprint arXiv:1612.06676, 2016.

[5] J. Lin, E. Keogh, A. Fu, H. Van Herle, "Approximations to Magic: Finding Unusual Medical Time Series," CBMS, 2005, pp. 329-334.

[6] C.C.M. Yeh et al., "Matrix Profile I: All Pairs Similarity Joins for Time Series: A Unifying View that Includes Motifs, Discords and Shapelets," Proc' of 16th IEEE ICDM, 2016, pp. 1317-22.

[7] D. Yankov, E. Keogh, and U. Rebbapragada, "Disk Aware Discord Discovery: Finding Unusual Time Series in Terabyte Sized Datasets," ICDM, 2007, pp. 381-390.

[8] N. Laptev and S. Amizadeh, "S5 - A Labeled Anomaly Detection Dataset, version 1.0(16M)." 2015. Distributed by Yahoo Research. https://webscope.sandbox.yahoo.com/catalog.php?datatype=s&did=70

[9] I. Vasheghani-Farahani, et. al. "Time Series Anomaly Detection from a Markov Chain Perspective," ICMLA, 2019, pp. 1000-1007.

[10] V. Zhang, "A Tour of AI Technologies in Time Series Prediction," 2019.

[11] S. Däubener, et. al, "Large Anomaly Detection in Univariate Time Series: An Empirical Comparison of Machine Learning Algorithms," *19th Industrial Conference on Data Mining ICDM 2019*, 2019.

[12] B. Barz, Y. G. Garcia, E. Rodner and J. Denzler, "Maximally divergent intervals for extreme weather event detection," OCEANS 2017 - Aberdeen, Aberdeen, 2017, pp. 1-9.

[13] "City of Melbourne - Pedestrian Foot Traffic." www.pedestrian.melbourne.vic.gov.au/ (accessed May 22, 2020).

[14] Mitzi McRae, Melbourne Djembe - Bourke St Mall Flash Mob - 29th June 2013. (Jul. 20, 2013). Accessed: May 22, 2020. [Online Video]. Available: www.youtube.com/watch?v=gLzDFjiRQE8

[15] M. Doan, et.al (2015), Profiling pedestrian distribution and anomaly detection in a dynamic environment. CIKM 2015, pp 1827-30

[16] K. Hundman, et al. Detecting Spacecraft Anomalies Using LSTMs and Nonparametric Dynamic Thresholding. KDD 2018: 387-395

[17] Ahmad, S., Lavin, A., Purdy, S., & Agha, Z. (2017). Unsupervised real-time anomaly detection for streaming data. Neurocomputing. 2017.

[18] E. Keogh, J. Lin, A. W. Fu: HOT SAX: Efficiently Finding the Most Unusual Time Series Subsequence. ICDM 2005: 226-233

[19] Page, E. S. (1957). "On problems in which a change in a parameter occurs at an unknown point". Biometrika. 44 (1/2): 248–252.

[20] Y. Bu, L. Chen, A. W. Fu, Dawei Liu: Efficient anomaly monitoring over moving object trajectory streams. KDD 2009: 159-168

[21] https://en.wikipedia.org/wiki/January_2017_Melbourne_car_attack

[22] M. Ahmed, A. Mahmood: Network Traffic Pattern Analysis Using Improved Information Theoretic Co-clustering Based Collective Anomaly Detection. SecureComm (2) 2014: 204-219

[23] MERLIN Webpage: https://sites.google.com/view/merlin-find-anomalies

APPENDIX A: Some Benchmark Datasets are Trivial

In the main text we noted that some fraction of the benchmark data yield to simple algorithms from the 1950s [19]. Here we demonstrate that claim. This is important because it confounds any comparison of algorithms. For example, suppose we find that Olympic powerlifter Long Qingquan can lift 1, 2, 3 and 300 kg, and that the current author can lift 1, 2 and 3 kg. It would be foolish to conclude that because they agree on ¾ of the lifting tasks, that they are almost equally strong.

A further simplified version of the sixty-three year old algorithm in [19] is:

```
flag = zeros(size(T));          %% Code can be run in Matlab
for i = 4 : length(T)-4
 if std(T(i+1:i+4)) - std(T(i-3:i)) > 1,  flag(i) = 1;, end;
end;
```

In Figure 19 we show the results of running this code on two benchmark datasets that yield to such simple algorithms.

[4] An order of magnitude simpler in terms of number of parameters to set, of the number of lines of code written etc.

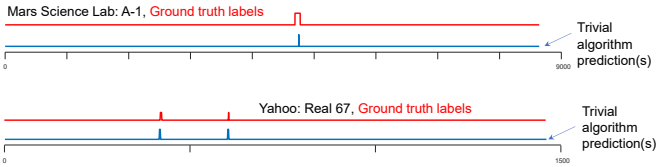


Figure 19: Two (of many) examples of benchmark datasets that yield to the trivial hard-coded algorithm shown above. *top*) From NASA [16]. *bottom*) From Yahoo [8].