Rare Time Series Motif Discovery from Unbounded Streams

Nurjahan Begum
UC Riverside
nbegu001@cs.ucr.edu

Eamonn Keogh
UC Riverside
eamonn@cs.ucr.edu

ABSTRACT

The detection of *time series motifs*, which are approximately repeated subsequences in time series streams, has been shown to have great utility as a subroutine in many higher-level data mining algorithms. However, this detection becomes much harder in cases where the motifs of interest are vanishingly rare or when faced with a never-ending stream of data. In this work we investigate algorithms to find such rare motifs. We demonstrate that under reasonable assumptions we must abandon any hope of an exact solution to the motif problem as it is normally defined; however, we introduce algorithms that allow us to solve the underlying problem with high probability.

Keywords

Motif Discovery, Time Series, Streaming Data

1. INTRODUCTION

Time series motifs are approximately repeated patterns found within a longer time series. Since their definition a decade ago [18], dozens of researchers have used them to support higher-level data mining tasks such as clustering, classification, dictionary learning [5], visualization, rule discovery and anomaly detection [25]. While the original time series motif discovery algorithm was an approximate algorithm, the recently introduced MK algorithm provides a scalable exact solution [25]. For a user-defined length of interest, the MK algorithm finds the pair of non-overlapping subsequences that have the minimal Euclidean distance to each other. The algorithm has been further generalized to a limited streaming setting; it is now possible to find and maintain the motif pair in a sliding window, say the last k minutes, of a continuous stream [24].

In spite of recent progress in motif discovery, there are two related situations for which there are currently no tractable solutions. If the motifs are extremely rare, then it is extremely unlikely that we will see two of them within k minutes of each other, as assumed by [24]. Depending on data arrival rates and the hardware platform used, k might be as large as 20 minutes. However, as shown in Figure 1, we may be interested in finding patterns that occur only once a month or less. If we ignore the streaming case and assume that the m-datapoints are stored in memory, we are still limited by the scalability of the current fastest known algorithm [23][25], which is O($m$log(m)) with high constants. For some of the datasets we wish to consider (cf. Section 6.3), this would require decades of CPU time.

In this work we introduce an efficient framework that allows us to solve such problems with *very* high probability. Our key observation is based on an understanding of how motif discovery is actually used in domains as diverse as electroencephalography [25], entomology and nematology [5].

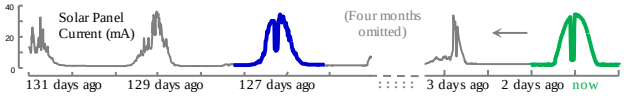

Figure 1: A never-ending time series stream from a weather station's solar panel [2], only a fraction of which we can buffer. A pattern we are observing now seems to have also occurred about four months ago.

It is important to recognize that the narrowly defined time series motif *pair* discovery problem as defined in [25] is really a proxy problem for the underlying task of finding more generally repeating patterns. For example, suppose that there are ten examples of an unknown repeated behavior in a dataset with ten million items, and we wish to discover them. We could employ the MK algorithm, which would find the pair from the ten patterns that are the *minimum* distance apart, and then use a quick linear scan to find the other eight patterns. Indeed, this is suggested in [25]. Note, however, that it would suffice for our purposes to find *any two* of the ten repeated patterns; we do not really need the smallest distance pair, or any specified pair. This is a simple but important observation, because it allows us to succeed if we find any one of 45 pairs, clearly a much easier task. As we shall show, this slightly relaxed assumption allows us to solve problems that are otherwise intractable, and to discover repeated patterns in unbounded streams. The patterns discovered by our algorithm can serve as an input to higher-order algorithms that do semi-supervised learning [11], or look for changes in the frequencies of the discovered patterns that may signal *anomalous* behavior [18] etc.

2. DEFINITIONS AND NOTATION

In order to concretely state the problem at hand, we will define the key terms used. We begin with a definition of our data type of interest, *time series*:

Definition 1 *Time Series*: A time series $TS = ts_1, ts_2, ...,ts_m$ is an ordered set of m real-valued variables, where ts_m is the most recent value.

We are only interested in the local properties of a time series; thus, we confine our interest to subsections of the time series, which are called *subsequences:*

Definition 2 *Subsequences*: Given a time series TS of length m, a subsequence of TS is a sampling of length $l \leq m$ of contiguous

Proceedings of the VLDB Endowment, Vol. 8, No. 2

positions from *TS* starting from position *i*. Formally, $TS_{i,l} = ts_i, ts_{i+1}...ts_{i+l-1}$ for $1 \le i \le m-l+1$.

The subsequences from a time series are extracted by the use of a *sliding window:*

Definition 3 *Sliding Window:* For a time series *TS* of length *m*, and a user-defined subsequence of length *n*, all possible subsequences of *TS* can be found by sliding a window of size *n* across *TS*.

It is well understood in the literature that (with very rare and well-defined exceptions) it is meaningless to compare time series unless they are z-normalized [15][18] .

Definition 4 *Z-normalized Subsequence:* If the values of a time series subsequence TS_P have an approximately zero mean with standard deviation (and variance) in a range close to one, then TS_P is called a z-normalized subsequence.

A common task associated with subsequences is to determine if a given subsequence is similar to other subsequences under some distance measure. This notion is formalized in the definition of a *match:*

Definition 5 *Match:* Given a positive real number *T* (called *threshold*) and a time series *TS* containing subsequences TS_P and TS_Q beginning at position *P* and *Q*, respectively, if $D(TS_P, TS_Q) \le T$, then TS_Q is called a matching subsequence of TS_P.

These notations are summarized in Figure 2. The obvious definition of a match is necessary to formally define a *trivial match.* Clearly, the best matches to a subsequence tend to be located one or two points to the left or the right of the subsequence in question. Almost all algorithms need to exclude such trivial solutions. The concrete definition is given below:

Definition 6 *Trivial Match:* Given a time series *TS* containing subsequences TS_P and TS_Q beginning at position *P* and *Q*, respectively, TS_Q is a trivial match to TS_P if either $P = Q$ or a subsequence TS_Q' beginning at *Q'* such that $D(TS_P, TS_Q') > R$ and either $Q < Q' < P$ or $P < Q' < Q$ does not exist.

We are now in a position to define *objects*, which are non-overlapping subsequences from a time series stream:

Definition 7 *Object:* Given a time series $TS = (ts_1, ts_2, ...,ts_m)$ two objects of length *l* are subsequences $(TS_{i,l}, TS_{j,l})$ of *TS* such that $1 \le i \le i + l-1 < j \le m-l$.

The reason we consider only non-overlapping subsequences is to avoid trivial matches, which must be excluded to define the success condition of our problem.

To measure the distance between objects, we use the ubiquitous Euclidean distance measure [1][8]:

Definition 8 *Euclidean Distance:* Given two time series (or time series subsequences) TS_P and TS_Q, both of length *m*, the Euclidean distance between them is the square root of the sum of the squared differences between each pair of the corresponding data points: $D(TS_P, TS_Q) \equiv \sqrt{\sum_{i=1}^{m}(TS_{P_i} - TS_{Q_i})^2}$

The reader may imagine that using the Dynamic Time Warping (DTW) distance measure could produce better results for the task at hand. However, this is *not* the case. As shown in [25], for time series objects which are *very* similar (suggestive of these being motifs), the values of Euclidean distance and DTW must be *very* tightly related. Given this, we use the ubiquitous Euclidean distance measure for calculating the similarity between the time series patterns [9][14][22][23]. We are now in a position to give a concrete problem statement.

3. PROBLEM DEFINITION

Assume we are given a never-ending time series stream **S** that mostly produces instances of *patternless* data in **R**, and with some low probability *p*, instances of an *unknown* pattern in **G** (we will define *patternless* later). As shown in our running example in Figure 2, items in **G** appear visually similar to each other, but are *not* perfectly similar. Our goal is simply to detect an instance in **G** as early as possible.

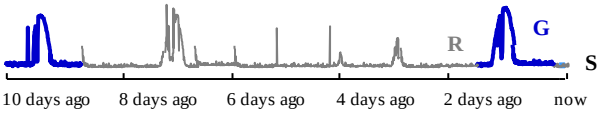

Figure 2: A never-ending time series stream S produces mostly patternless data in R, and with *very* low probability, instances in G.

Note that our problem is to *detect* instances in **G** as early as possible; but we are not interested in *testing the significance* of these instances. Our algorithm is designed to be a subroutine for higher level algorithms, and the ranking of the patterns in **G** in terms of significance can be done by these algorithms [6][7][13]. Further note that we are assuming that we are given the lengths of the patterns-of-interest by domain experts, an assumption which is typical in the data mining community [5][13][22][30]. However, recent work by [23] shows that this assumption can be relaxed with little overhead.

Note that we assume there is no explicit test to check if a pattern is in **G**. Concretely, patterns in **G** have statistical properties that are also typical of data in **R**. Thus, the defining assumption of this work is that the only way we can tell if an item is in **G** is to note that it is sufficiently similar to another item also suspected to be from **G**, which we call a match (cf. Definition 5).

More formally, we assume a distance threshold *T* such that:

- Any two objects, where at least one is *not* in **G**, are unlikely to have a distance < *T*.
- Any two instances in **G** are likely to have a distance < *T*.

The former point is essentially an informal definition of *patternless*, but must come with a caveat. As **R** approaches infinity (recall we have an unbounded stream) and given that we are dealing with z-normalized objects, the space of possible shapes of a time series will eventually be exhausted and **R** must eventually produce two instances that are less than the threshold apart. Thus, the assumption of the existence of a distance threshold *T* discussed above is relative to a sample size of **S** that is approximately *w*, the amount of memory we can devote to this task (measured in the number of instances that can be stored). The threshold *T* may be given to us in the form of domain knowledge, or we may have to learn it from the data (cf. [33]). This idea is illustrated in Figure 3.

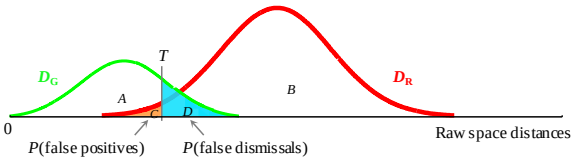

Figure 3: D_G and D_R represent the all-pair distance distributions of the patterns in G and R, respectively. The distance threshold *T* represents the boundary that

separates the decision whether the two patterns in question belong to G or not.

Note that the two distributions here are Gaussian. This is *often* the case for real data (cf. Figure 10 and Figure 12), and we adopt this assumption for ease of exposition. We note, however, that generalizations to other distributions are trivial. We further note that for illustration purposes, the $D_{\mathbf{R}}$ distribution is only slightly larger than the distribution $D_{\mathbf{G}}$; however, in the problems we wish to deal with, we expect the prior probability of the patterns in **R** (i.e., 1-*p*) to be many orders of magnitude larger.

Finally, a critical observation from Figure 3 is that the two distributions overlap. Thus, no matter what value of *T* we have, we must deal with some probability of error. In particular, two types of error can occur:

- The region marked *C* is proportional to the probability that we will falsely believe that two patterns from **R** that happen to be similar are exemplars from **G**.
- The region denoted *D* is proportional to the probability that two exemplars that are from **G** will not be identified as such because they happen to be farther apart than average.

For real-world problems we expect the area of *C* + *D* to be much smaller than that illustrated in Figure 3. Obviously, we can adjust the threshold based on our relative tolerance for each type of error. This relative tolerance itself may depend on the application [13].

We are finally in a position to formally define the task at hand:

Problem Statement: Given an unbounded time series stream **S** that mostly produces instances in **R**, but with some *very low* probability *p*, produces instances of an *unknown* pattern in **G**, and a user-defined distance threshold *T*, detect and return an instance in **G** such that the expected number of instances in **S** seen is minimized.

For example, recall our running example shown in Figure 1 and Figure 2. The distinctive pattern (shown *colored*/**bold**) consisting of a smooth "dump" with a "dropout" near the center, appears to have been caused by the shadow of a pole falling on the solar panel during a rare cloudless day. Such patterns occur five or six times a year (at that location), so if we wait *long enough* we will probably see two close together, perhaps even on consecutive days. However our task is to discover such patterns as soon as possible, independent of when/how often, they occur.

3.1 A Brute Force Algorithm

Given the assumptions above, a trivial algorithm suggests itself. We can simply keep the first *k* items of **S** in memory, and then, when the k^{th} + 1 item arrives, we compare it to all *k* items currently stored and report "`current item is a member of` **G**" if we find that $D(k+1, j) < T$ and $j < k+1$.

Note that this algorithm has some probability of making a type I or type II error, but this is intrinsic to our assumptions, and no algorithm can do better.

This brute force algorithm is clearly optimal, but also clearly untenable. We are assuming that **S** is an infinite stream, and *p* is a very small number. Thus, we will eventually run out of space to store items, or the time needed to make all the comparisons.

We can at least *mitigate* the time complexity of the naïve algorithm using off-the-shelf dynamic indexing techniques (recall that because we are dealing with *real-valued* high dimensional objects, we cannot avail of the *O*(1) equality tests available to the *discrete* analogue problems) [10]. Therefore, we concern ourselves here with the more difficult resource limitation: space constraints.

3.2 Brute Force with Limited Memory

In our problem setting, we assume that we must work with *C*, a cache of a fixed size *w*. Here, *w* is the number of instances that can be stored in main memory. Note that while taking the distances of the cached patterns, we only consider patterns which are *not* trivial matches (cf. Definition 6) to each other (cf. Section 2).

It is clear that the performance of any cache-based algorithm depends critically on the size of the cache. Consider the two following special cases. If $w = \infty$, then the cache-based algorithm is as good as the trivial algorithm discussed above. If $w = 2$, then the probability of an instance in **G** being in the cache as another instance in **G** arrives is just *p*, and we will typically have to wait a *very* long time before reporting success.

Given this observation, our metric of success can be seen as the expected number of objects seen from **S** before detecting an object in **G**. We are now in a position to give a derivation of this metric, which we discuss in the next section.

3.2.1 Derivation of the Success Metric

In order to derive the theoretical model for the success metric, we denote the number of objects that must be seen from **S** before reporting success, Ψ.

By definition, at each time step, **S** produces an instance in **G** with probability *p*. By the time we detect an instance in **G**, the elements in *C* may or may not experience cache replacement(s).

Case 1: No cache replacement

Consider the case when there has *not* been any cache replacement by the time we report success. In order to find the probability of success, we denote the number of objects observed from **S** until the *first* and *second* instances in **G** are in *C* by Ψ_1 and Ψ_2, respectively.

Each Ψ_i is a geometric random variable with success probability *p*. Because there has not been any cache replacement when we report success, the probability of having the first two instances in **G** be in *C* is: $p_{\mathbf{G}_C} = p + p = 2p$.

Case 2: Cache replacement

Now consider the case when there has been cache replacement at least once. After discarding an element from the cache, the remaining *w*-1 elements in the cache can be in either of the following two categories:

Category 1: All of the *w*-1 patterns in *C* are instances in **R**. The probability of this event is: $P(\boldsymbol{R}_{w-1}) = (1-p)^{w-1}$.

Category 2: All but one of the *w*-1 patterns in *C* are instances in **R**. The probability of this event is: $P(\boldsymbol{R}_{w-2}\boldsymbol{G}) = (1-p)^{w-2} * p * (w-1)$.

We can report success if and only if the w^{th} pattern to be inserted into *C* is in **G**, and the previous *w*-1 patterns in *C* belong to Category 2. Therefore, the probability of the success event is: $P(SuccessEvent) = p * P(\boldsymbol{R}_{w-2}\boldsymbol{G})$.

If the patterns in *C* are such that no two of them are instances in **G**, then we call it a *failure event*. We describe the probabilities of this event below:

If the w^{th} pattern to be inserted into *C* is in **R**, and the previous *w*-1 patterns in *C* belong to Category 2, then the probability of the failure event is: $P(FailureEvent_1) = (1-p) * P(\boldsymbol{R}_{w-2}\boldsymbol{G})$.

If the w^{th} pattern to be inserted into *C* is in **R**, and the previous *w*-1 patterns in *C* belong to Category 1, then the probability of the failure event is: $P(FailureEvent_2) = (1-p) * P(\boldsymbol{R}_{w-1})$.

If the w^{th} pattern to be inserted into *C* is in **G**, and the previous *w*-1 patterns in *C* belong to Category 1, then the probability of the failure event is: $P(FailureEvent_3) = p * P(\boldsymbol{R}_{w-1})$.

Given this analysis, the probability of having the first two instances in **G** be in *C* is:

$$p_{\mathbf{G}_C} = \frac{P(SuccessEvent)}{P(SuccessEvent)+\sum_{i=1}^{3} P(FailureEvent_i)}.$$

Based on the analyses above, $p_{\mathbf{G}_C}$ becomes:

$$p_{\mathbf{G}_C} = \begin{cases} 2p & (Case\ 1) \\ \frac{P(SuccessEvent)}{P(SuccessEvent)+\sum_{i=1}^{3} P(FailureEvent_i)} & (Case\ 2) \end{cases} \quad (1)$$

Therefore, after seeing *n* objects in **S**, the probability of failing to detect an instance in **G** is: $p_{failure} = (1-p_{\mathbf{G}_C})^n$. Given this, the probability of success after observing *n* instances in **S** is: $p_{success} = 1 - p_{failure}$.

More generally, Figure 4 shows the relationship between cache size and the number of items seen before detecting an instance in **G**, under the following concrete assumptions. One in a hundred objects in **S** belongs to **G**; everything else belongs to **R**. We further assume (just for this toy example) that the moment we have two instances in **G** in the cache, we can unambiguously detect that fact.

From Figure 4 we observe that if *w* = 20, then we have to see about 2,857 objects to have a 0.99 probability of correctly identifying an instance in **G**. As *w* gets larger, the number of objects needed to reach this 0.99 probability threshold decreases, but we can clearly see diminishing returns. Using a cache size five times larger only reduces the number of objects we need to see to about 918 objects, and moving to an arbitrarily large cache size ($w = \infty$) further improves this down to about 227. For comparison, with *w* = 2, the 0.99 probability is not reached until we have seen 46,049 objects (this value is truncated from Figure 4 for clarity).

Note that our toy example considers the case when *p* = 0.01; however, we expect to deal with real-world problems in which *p* may be several orders of magnitude smaller. Such values of *p* will "stretch" the x-axis, but our core observations about the diminishing returns properties remain.

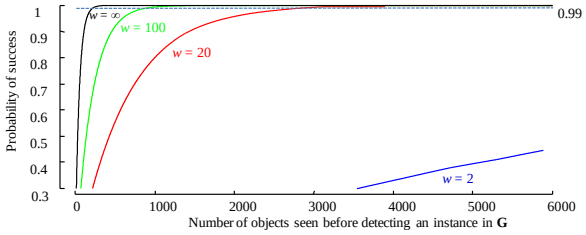

Figure 4: The number of objects that must be seen to find a pattern in G, for different cache sizes *w*, with a desired probability of success. The 0.99 probability is highlighted with a horizontal dashed line.

Also note that we have not stated which cache replacement policy we used in Figure 4. As we shall show below, the two most obvious candidates, First-In-First-Out (FIFO) and Random Replacement (RR), *both* correspond to this analysis.

3.3 Analysis of Cache Replacement Policies

Two obvious cache replacement policies are First-In-First-Out (FIFO) and Random Replacement (RR) [28]. As the names suggest, in FIFO, the oldest object in the cache is discarded at each time step, whereas in RR, the object to be discarded is selected randomly. Note that the FIFO replacement policy is vulnerable to an adversarial case [3]. Imagine a version of our problem in which **S** produces instances in **G** at uniform time instances. If the number of objects produced by **S** after an instance in **G** is greater than or equal to *w*-1, then we can *never* detect an instance in **G** using the FIFO cache replacement policy. The obvious solution to mitigate such a problem is to use *randomization*, which is often used in algorithms to avoid such pathological cases [21]. It is important to note that this adversarial worst case is *not* pathologically unlikely. For example, some manufacturing machines may produce a special pattern as the machine is recalibrated during a shift change, every eight hours. A FIFO policy with a cache size of seven hours would never discover this pattern.

Using the analysis in Section 3.2, we can more formally define the metric of success for our problem. For the case without any cache replacement, the number of objects observed from **S** until two instances in **G** are in *C*, i.e., Ψ, is a negative binomial random variable. We define Ψ as $\psi = \sum_{i=1}^{2} \psi_i$.

Because the expectation of a sum of random variables is the sum of their expectations, we have: $E(\psi) = E\left(\sum_{i=1}^{2} \psi_i\right) = \sum_{i=1}^{2} E(\psi_i) = \frac{1}{p} + \frac{1}{p} = \frac{2}{p}$. (Recall from Section 3.2, each Ψ_i is a geometric random variable with success probability *p*). Therefore, if *w* >= 2/*p*, we expect no cache replacement.

For the case with at least one cache replacement, $E(\Psi)$ depends on the number of objects seen from **S** *immediately before* the cache replacement, $\Psi_{replacement'}$, and *after* the cache replacement(s), $\Psi_{replacement}$, until we report finding a pattern in **G**. $\Psi_{replacement'}$ is simply *w*, and $\Psi_{replacement}$ is a geometric random variable with success probability $p_{\mathbf{G}_C}$ as in Equation 1. Therefore, for this case, $E(\psi) = \frac{1}{p_{\mathbf{G}_C}} + w$. Similar to our observation about the case without cache replacement, if *w* < 2/*p*, we expect cache replacement(s) to occur.

In Figure 5 we show that the expected number of elements seen from **S** before detecting an instance in **G** (our success metric stated in Section 3.2) decreases with increasing cache size. In addition to this, Figure 5 shows how closely the success metric agrees with the empirical results for both of our candidate cache replacement policies.

As Figure 5 suggests, the larger the cache is, the fewer objects in **S** we can expect to see before detecting a pattern in **G**. This observation does not seem directly exploitable, as the cache size is a domain constraint. However, as we will show in Section 4, we can "virtually" increase the cache size by changing the representation of the data.

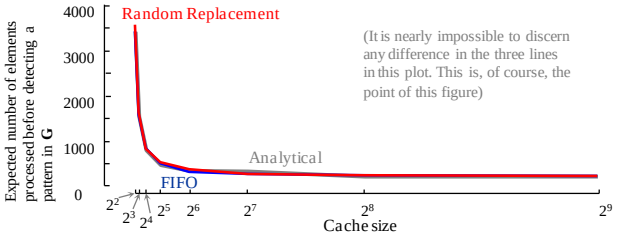

Figure 5: The experimentally-determined success metric for FIFO and RR policies closely agrees with the theoretically-derived metric for a wide range of values of *w*.

3.4 A Magic Sticky Cache

The analysis in Section 3.3 shows that for a fixed *w*, the more objects we see, the higher the probability is of detecting an instance in **G**. Given *w* and *p*, we can predict the performance using the analysis in Section 3.3. The results show that a larger cache size helps improve the performance. Beyond making the cache size larger, is there any other way we could improve the performance?

To answer this question, we can perform a *gedankenexperiment*. Imagine for a moment that we could "magically" control the discard probabilities of the patterns in **G** and **R** from the cache, such that patterns from **G** tend to "stick" in the cache for longer. In particular, imagine we somehow can discard items in **R** with, say, a 50 or 100 times greater probability than items in **G**. This would improve the chances of an item from **G** *remaining* in the cache as a new **G** exemplar arrives, and thus improve Ψ. This intuition is illustrated with an experiment shown in Figure 6.

As we can see, making the cache "stickier" for items that *might* be in **G** significantly improves our chance of detecting an instance in **G**. However, it is not clear yet how we could imbue the cache with this ability.

To summarize: The ideas in this section suggest two possible approaches (which are not necessarily exhaustive or mutually exclusive) that we can consider for improving the performance in our problem:

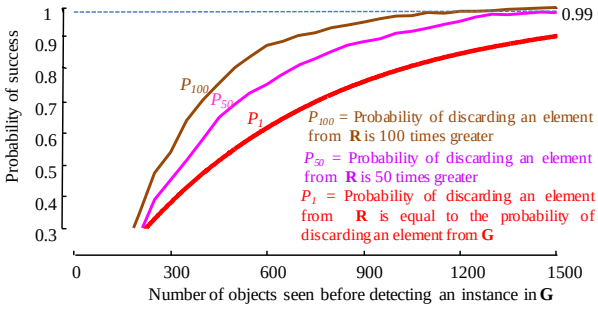

Figure 6: The red/bold line here is identical to the *w* = 20 line in Figure 4. If we could somehow increase the probability of discarding an element from R we would obtain a significant improvement

- Find a cache replacement policy that *minimizes* the probability of discarding instances in **G** as opposed to instances in **R**= **S - G** (cf. Figure 6);
- Change the representation of the data, such that we can fit more objects into *C*. This implicitly improves the probability of keeping an instance of **G** in the cache (cf. Figure 4 and Figure 5).

In the next section, we explore the latter idea, and in Section 5 we discuss the former idea.

4. DATA REPRESENTATION POLICY

4.1 Initial Observation

Recall from Figure 5 that the larger the cache is, the higher the probability is of detecting an instance in **G**. We can emulate the effect of a larger cache by changing the representation of the data. Compressing or downsampling the data allows more objects to fit in the cache; we can thus expect to detect an instance in **G** *sooner* than in the raw space.

This idea requires some careful consideration. While time series data is typically amiable to lossless compression techniques such as delta encoding or Lempel Ziv, such methods require decompression before the Euclidean distance calculations, and are thus more suited to archiving data. They would clearly introduce an intractable overhead for a fast-moving stream, where we would have to (re)decompress each instance at every cache insertion.

Downsampling (or equivalently, *lossy compression*) avoids this problem, but introduces a new issue we must deal with. From Figure 7, we can see that changing the representation of the data inevitably changes the distances between objects.

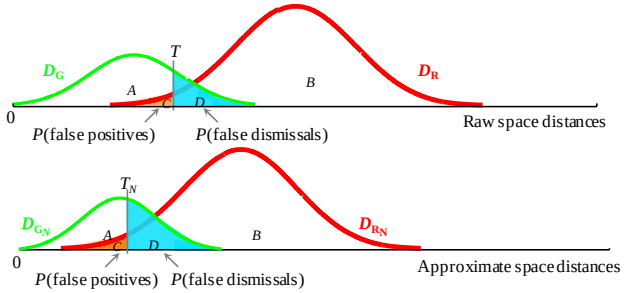

Figure 7: *top*) D_G and D_R are identical to the all-pair distance distributions of the patterns in G and R, respectively, in Figure 3. *bottom*) After downsampling the data, the new distance distributions (D_{GN} and D_{RN}) and the new threshold T_N shift left.

If we consider only orthonormal transformations (DFT, DWT, SVD, etc.), the distances between any pairs of objects can *only* be reduced [1]. Because of this *lower-bounding* property, the distributions in Figure 7.*top*) can only shift to the left (Figure 7.*bottom*). As a consequence, the distance threshold *T* can also only shift left in the approximate space (T_N).

Note, however, that we cannot say how the overlapping area of the distance distributions will change – because *either* of the distribution's standard deviations could increase, decrease, or remain the same (Figure 7.*bottom*), therefore, the area *C* + *D* could also change arbitrarily. In practice, however, this area always increases, and in the next section we show how to incorporate this into our cost model.

4.2 Cost Model

It is important to note that in our problem definition, of the two errors we can make while detecting an instance in **G**, one "hurts" more than the other. This is because if we falsely miss two exemplars in **G**, then we still have the *hope* of detecting an

instance in **G** in the future. However, if we mistakenly say two exemplars are in **G**, when in fact at least one is in **R**, then we can never detect two instances in **G**, as we immediately stop our search. Of course, we could adapt our definition such that we do not actually stop, perhaps buffering the tentative motif and continuing the search. However, the tentative motif must be examined by a human or algorithm at some point [13]. Thus, these false positives do have an inescapable cost [13].

Given this observation, we can design a cost model in which we fix the probability that we mistakenly claim two patterns to be in **G** (area *C* in Figure 7). This fixing results in an *increasing tendency* of the probability of falsely dismissing two true patterns in **G** (area *D* in Figure 7) as we downsample the data. As a consequence, the probability that two patterns in the cache are *believed* to be from **G** *tends to* decrease. We defer the calculations of these probabilities until Section 4.2.1. However, Figure 8 shows an empirical illustration of this observation.

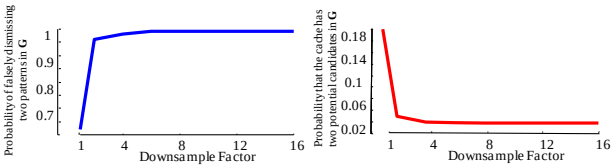

Figure 8: *left*) When downsampling, the probability of falsely dismissing two instances in G tends to increase if we fix the probability that two cache patterns are claimed to be in G by mistake. *right*) Consequently, the probability that two patterns in the cache are believed to be from G tends to decrease.

From Figure 8.*left*) we can see that if we fix the probability that two patterns in *C* are falsely identified to be in **G**, then as we downsample more, the probability of the false dismissals of two cached patterns in **G** tends to increase. As a consequence, the probability that two cached patterns are potential candidates in **G** tends to decrease (Figure 8.*right*).

We are now in a position to describe the derivation of these probabilities and to formally define the 'costs' involved.

4.2.1 Cost Calculation

Assume that we know the distance threshold *T* in the raw space, which can come either from user input or from a threshold learning model (cf. [33]). Further assume that we know the means (μ_{D_G},μ_{D_R}) and standard deviations (σ_{D_G},σ_{D_R}) of both of the distance distributions in Figure 7.*top*. Given these, we can calculate the standard score of *T* corresponding to each of these distributions as below:

$$z_{False\ Positive} = (T - \mu_{D_R})/\sigma_{D_R}$$
$$z_{False\ Dismissal} = (T - \mu_{D_G})/\sigma_{D_G}$$

Using the standard normal probability table, we can calculate the area to the left of $z_{False\ Positive}$, which is the probability of mistakenly claiming two patterns to be in **G**. Similarly, we can calculate the area to the right of $z_{False\ Dismissal}$, which is the probability of falsely dismissing two true patterns in **G**. In Figure 7.*top*, these errors are represented as shaded areas labeled *C* and *D*, respectively. From now on, we will refer to these errors as α and β, respectively.

Given these two error probabilities, Figure 3 illustrates that the probability that two cached patterns are believed to be exemplars in **G** is:

$$p_G = (A + C)/(A + B + C + D). \qquad (2)$$

As noted above, we fix α to be a constant independent of the downsampling rate. Therefore, the probability of falsely dismissing two true patterns in **G** in this space becomes greater. We illustrate this error as the shaded area *D* in Figure 7.*bottom*. We call this error β_N, and calculate it as follows.

From the fixed α, we can calculate $z_{False\ Positive}$. Using this $z_{False\ Positive}$, we can calculate T_N (cf. Figure 7.*bottom*) as below:

$$T_N = \mu_{D_{R_N}} + (z_{False\ Positive} * \sigma_{D_{R_N}})$$

where $\mu_{D_{R_N}}$ and $\sigma_{D_{R_N}}$ are the mean and standard deviation of the distance distribution D_{R_N} in Figure 7.*bottom*.

We can calculate the standard score of T_N corresponding to the distribution D_{G_N} in Figure 7.*bottom* as:

$$z_N = (T_N - \mu_{D_{G_N}})/\sigma_{D_{G_N}}$$

Using the standard normal probability table, we can calculate the area to the right of z_N, which is β_N. β and β_N correspond to the probabilities shown in Figure 8.*left*) for the original and downsampled spaces, respectively.

From the areas *A*, *B*, *C* and *D* in Figure 7.*bottom*), we can calculate the probability that two cached patterns are believed to be exemplars in **G**, p_{G_N} using equation 2. p_G and p_{G_N} correspond to the probabilities shown in Figure 8.*right*) for the original and downsampled spaces, respectively.

More generally, Figure 9 shows the relationship between a 'virtually' large cache made by downsampling the data and the number of objects we expect to see before reporting success, under the following conditions. We make a synthetic dataset of two classes, **G** and **R**, by distorting an instance of Gun point and FaceAll datasets [16], respectively, with some Gaussian noise. We assume the rareness of the patterns in **G** is 1/100 and our cache is allowed to store just two patterns in raw format. Under the assumption that we can unambiguously detect instances in **G**, we show the expected number of objects we need to see before reporting success in Figure 9.*left*).

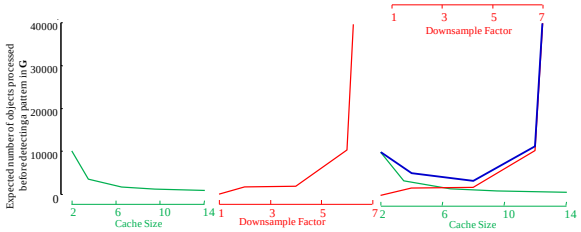

Figure 9: *left*) The *help* factor: A larger cache allows faster detection of instances in G. *center*) The *hurt* factor: The greater the downsampling, the slower the detection. *right*) The performance of the overall system is based on the influence of these two factors, with downsample factors of 2 to 4 performing best.

As noted above, due to downsampling the data, we can no longer identify instances in **G** unambiguously; therefore, we detect two instances in **G** with probability p_G. Because we fix the false positive error probability, the probability of falsely dismissing true instances in **G** tends to *increase* as we downsample more (Figure 8.*left*). This increased probability of false dismissals of the patterns in **G** results in a decreasing tendency of p_G (Figure 8.*right*). As a consequence, we expect to

see an increase in the expected number of objects in **S** we examine before detecting an instance in **G**. This effect is illustrated in Figure 9.*center*).

If we review these two observations: *A larger cache allows faster detection of an instance in* **G** and *A "larger" cache* (*emulated by downsampling*) *causes slower detection of an instance in* **G** *given the higher probability of not recognizing a pair from* **G**, they suggest that there must be a cache size that maximizes this tradeoff, i.e., $cost_{final} = cost_{help} + cost_{hurt}$. In Figure 9.*right*), this cache size is achieved with downsample factors from two to four.

There are two important observations we must make before moving on. First, note that the cache size that maximizes this tradeoff has a fairly wide "valley," suggesting that this parameter is not too sensitive. Second, while the best downsampling factor depends on the data and its sampling rate, it *can* be robustly learned on a small amount of training data. That is to say, if we find that the best downsampling factor for one person's electrooculography data at 1024Hz is about eight to ten, we can expect this to generalize well to other individuals.

4.3 Dimensionality Reduction

In our simple analysis in the previous section, we assumed that we placed more objects into the cache by *downsampling*. However, the reader will appreciate that there are more sophisticated dimensionality and cardinality reduction techniques to reduce the size of a time series. Indeed, the literature is replete with dimensionality reduction techniques, such as Singular Value Decomposition (SVD), Discrete Fourier Transform (DFT), Discrete Wavelet Transform (DWT), and Piecewise Aggregate Approximation (PAA) [31]. Here we consider PAA as the dimensionality reduction technique, because it is simple, incrementally computable and has linear time complexity [14]. We note that PAA and DWT are logically identical if both the original and reduced dimensionality are integer powers of two, and nearly identical otherwise [14].

Recall our observation in Figure 7; in Figure 10 we demonstrate that we get a similar shifting of the distance distributions under PAA as downsampling.

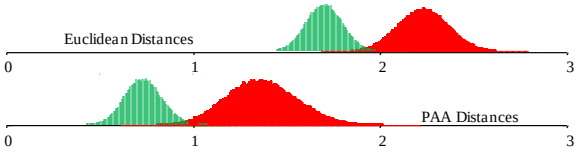

Figure 10: *top*) The all-pair distance distributions of the patterns in G and R in the raw space. *bottom*) After reducing the dimensionality by an integer factor of 5, the new distance distributions shift left.

As noted in Section 3.3, a larger cache size allows faster detection of a pattern in **G**, and dimensionality reduction emulates the effect of a larger cache. But as described in Section 4.1, working in the dimensionality-reduced space also makes it harder to determine if two objects in the cache belong to **G**. Therefore, we should expect to see a similar "*help and hurt*" effect as illustrated in Figure 9. To see this, in Figure 11 we conducted a similar experiment on the same synthetic dataset in Section 4.2.1 using PAA instead of downsampling.

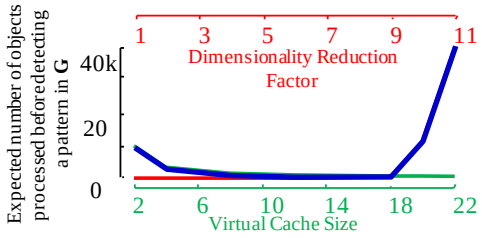

Figure 11: *help* factor vs. *hurt* factor using PAA. The tradeoff between both factors suggests that a dimensionality reduction factor of about 10 is best here.

4.4 Cardinality Reduction

We use the SAX (Symbolic Aggregate Approximation) [18] approach for cardinality reduction. This is because SAX is unique in allowing a distance calculation in the symbolic space that is commensurate with the Euclidean distance.

In Figure 12 we show that if we reduce the volume of time series with *cardinality* reduction rather than *dimensionality* reduction (cf. Figure 10), the distance distributions in the approximate space do not shift to the left as much. This is a promising sign that *cardinality* reduction might be a better technique for the task at hand.

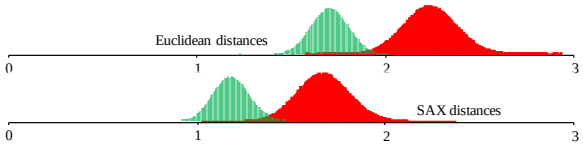

Figure 12: *top*) The all-pair distance distributions of the patterns in G and R in the raw space. *bottom*) After reducing only the cardinality of the data by a factor of 5 (6 bits), the new distance distributions shift left (assuming the original data points are 32 bits).

As hinted at in the observation in Section 4.1, we get the similar "*help and hurt*" behaviors after doing cardinality reduction on the same synthetic dataset in Section 4.2.1. We illustrate this in Figure 13.

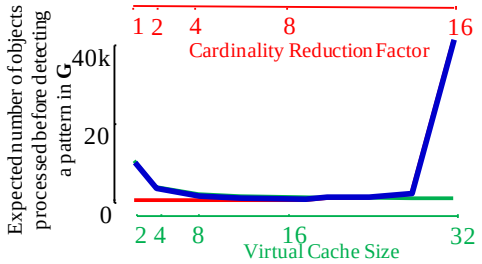

Figure 13: The tradeoff between the *help* and *hurt* factors using SAX suggests that a cardinality reduction factor of about 8 is best.

We can now answer the following question: Of the three techniques introduced to emulate a large cache, which is best? To see this, we plot the results of the experiments in this section on a single commensurate axis in Figure 14. We can see that cardinality reduction gives the minimum value of the expected number of objects we need to see before we report success. Moreover, cardinality reduction has a very wide flat "valley,"

meaning a large range of parameter choices produces excellent results.

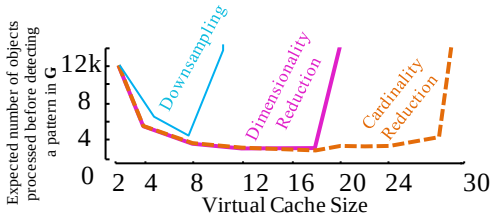

Figure 14: Of the three data reduction techniques, cardinality reduction dominates the other two over the entire range of virtual cache sizes.

Note that these results are for a single dataset with a single setting. However, many additional experiments (archived at [33]) confirm this general behavior.

5. STICKY CACHE ALGORITHM

As hinted at in the thought experiment in Section 3.4, if we had a 'magic' cache in which the *potential* instances in **G** tend to remain for longer, the probability of early detection of an instance in **G** increases significantly. In order to realize this idea, we need to create a biased cache replacement policy that reduces the probability of discarding potential **G** items from the cache as opposed to instances in **R**. This seems to open a chicken-and-egg paradox, as finding a pair of objects from **G** is our *goal*. In this section we will show how we can use a Bloom filter [4] to resolve this paradox.

A Bloom filter is a space-efficient randomized data structure (bit array) to support membership queries with no false negative and a small false positive probability [4]. It uses *k* independent hash functions, each of which maps some set element to one of the *m* bit positions of the array and sets the hashed bit positions. A membership query takes an input element and feeds it to the *k* hash functions to get *k* array positions. If any of the bit positions is found to be 0, then the element is *definitely* not a set member; otherwise, the element is *probably* a set member. Bloom filters have been widely used in frequent pattern mining [17][27]. To the best of our knowledge, this paper is the first work where Bloom filters have been used for *real-valued* time series.

The high-level intuition behind our idea is as follows. For every subsequence we see, we will use a Bloom filter to "remember" seeing (a SAX representation of) it. Before inserting the SAX word corresponding to the subsequence into the Bloom filter, we check to see if we have *already* seen this SAX word. If we have, this is suggestive that the subsequence *may* be from **G**. Given that evidence, we should make sure that it "sticks" in the cache longer than the subsequences we have only seen once.

There are some obvious caveats to this idea. Two SAX words being identical does not guarantee that both original real-valued sequences come from **G**, and two real-valued sequences that come from **G** can map to different SAX words. However, as we shall show, for reasonable SAX parameters false positives and false negatives are rare, and a collision in the SAX space is *really* highly predictive of membership in **G**.

Note that for this idea to work we actually need to see *three* items from **G**. The first is inserted into the Bloom filter as with all items. The second item collides with the first, which tells us to store the *real-valued* version of the second item in the cache, marked with low priority for deletion. Finally, when the third item arrives it will be recognized to be within threshold *T* of the second item, and we can report success.

The success of our method depends on finding reasonable values for the two SAX parameters - word and alphabet size. This is straightforward, so we relegate the discussion to [33].

5.1 Setting Appropriate Bloom Filter Size

As noted in Section 5, we require a biased cache replacement policy to detect instances in **G** earlier. The Bloom filter tags each instance in **S** as potential members in **G** or **R**. Based on these tags, we make the potential instances in **R** $\sum$ times more likely to be discarded from the cache than that of a potential instance in **G**. It is important to note that $\sum$ is not a critical parameter. Consider the two extreme situations. If $\sum = 1$, then the sticky cache policy degenerates to RR. In contrast, if $\sum = \infty$, the cache gets filled with potential instances in **G**, which results in frequent flushes of the cache, and decreases the probability of detecting instances in **G**. Note that the Bloom filter is not "free"; therefore, we must use up a portion of *C* for it. As the reader will appreciate, the two extreme choices of using almost *all* of *C*, or almost *none* of *C* (thus essentially degenerating to RR), are unlikely to perform well. The following analysis allows us to derive the appropriate size allocation for the Bloom filter in *C*.

Assuming *C* can hold at most ρ instances of length λ each, then the size of *C* in bytes is: $w_{cache} = 8\lambda\rho$[1]. If we restrict *C* to hold at most ρ_r instances ($\rho_r \leq \rho$), and allocate the remaining space to the Bloom filter, then the size of the Bloom filter in bytes is: $w_{Bloom} = \lfloor w_{cache} - 8\lambda\rho_r \rfloor$. Therefore, the number of unique elements hashed into the Bloom filter is: $b_{Bloom} = 8w_{Bloom}/bitsPerElement$[2].

Based on the analysis above, we perform an experiment in which we vary the cache /Bloom filter allocation in *C* to identify the region in which we obtain significant performance improvement over the naive RR policy. We use a fixed buffer which can hold at most 32 patterns each of length 150; thus, $w_{cache} = 38{,}400$ bytes. We vary the number of cache elements/ Bloom filter allocation. From Figure 15 we can see that the cache holding 8 to 12 patterns performs best and beyond this allocation, the performance starts degrading.

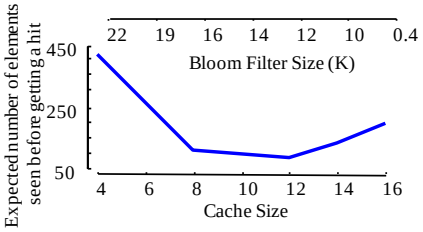

Figure 15: For a fixed w = 38,400 bytes (32 patterns of length 150), $\sum$ = 64, and p = 1/100, a cache allocation of 9,600 to 14,400 bytes (i.e., 8 to 12 patterns) performs best.

The reader will have anticipated the following question: if we exploit the performance improvement of the sticky cache in addition to the cardinality reduction technique (the best

[1] Each value of the time series takes 8 bytes.

[2] 9.58 bits per element for 1% FP probability of the Bloom filter.

technique from Section 4.4), can we do even better? To see this, we perform cardinality reduction of the patterns by a factor of 8 (the best reduction factor as of Figure 13), and redo the sticky cache experiment. We plot the results of this experiment with the other data reduction techniques in Section 4 with the same setup and plot the results on a common axis in Figure 16.

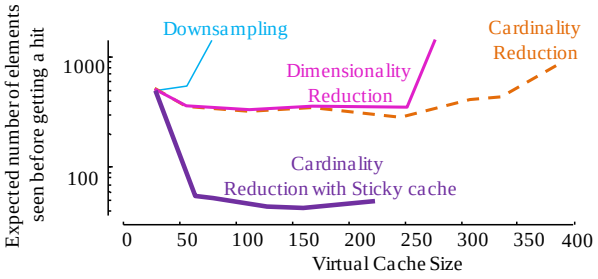

Figure 16: Out of all optimization techniques we discuss, the Sticky cache approach with a cardinality reduction factor of eight performs best. See also Figure 14, which shows a subset of this data.

In the next section we more formally describe the sticky cache algorithm.

5.2 Algorithms

In order to elucidate our sticky cache framework, we first explain how we use the Bloom filter in order to identify potential instances in **G**. Later we describe the cache maintenance policy.

5.2.1 Detection of Potential Target Instances

For a given alphabet size a and word size ω, we call the probability that two instances in **G** and **R** will map to the same SAX string $p_{\mathbf{G}sameSAXString(a,\omega)}$ and $p_{\mathbf{R}sameSAXString(a,\omega)}$, respectively. As explained in Section 5.1, in order to exploit the Bloom filter for detecting potential instances in **G**, we have to determine the approximate SAX parameters so that $p_{\mathbf{G}sameString}$ is maximized and $p_{\mathbf{R}sameString}$ is minimized. In Table 1, we describe the algorithm.

Recall our assumption of the existence of a distance threshold as a form of domain knowledge in Section 3. Based on this domain knowledge, we form a buffer of instances in **G** in line 1. We randomly sample patterns of the length of our interest from our input time series in order to form a buffer of instances in **R** in line 2. From lines 3-9, we determine the appropriate SAX parameters using the scoring function outlined in [33].

Table 1: SAX Parameter Selection Algorithm

Input	**TS**, the input time series **l**, length of the subsequences **A**, predefined set of SAX alphabet sizes **Ω**, predefined set of SAX word sizes
Output	a, the appropriate SAX alphabet size ω, the appropriate SAX word size
1	GBuffer = formGbuffer(TS,l) //using domain knowledge
2	RBuffer = formRbuffer(TS,l) //by random sampling
3	**for** i = 1 to size(A)
4	**for** j = 1 to size(Ω)
5	$p_{normalizedGsameSAXString}$ =findNormalizedProbability(**G**Buffer,i,j)
6	$p_{normalizedRsameSAXString}$ =findNormalizedProbability(**R**Buffer,i,j)
7	**end**
8	**end**
9	(a,ω)=findMaxNormalizedScore($p_{normalizedGsameSAXString}$, $p_{normalizedRsameSAXString}$)

We are now in a position to use the Bloom filter in order to detect potential instances in **G**. In Table 2, we outline the algorithm.

Table 2: Potential Target Instance Selection Algorithm

Input	**TS**, the input time series **subSeq**, subsequence in question a, the appropriate SAX alphabet size ω, the appropriate SAX word size $\sum_s$, likeliness factor of potential instances in **R** to be discarded from the cache
Output	$\sum$, weight of the potential instances in **R**
1	SAX_word = quantizeBySAX(subseq, a, ω);
2	bool exists = existsInBloomFilter(SAX_word)
3	**if** exists == true
4	$\sum$ = 1
5	**return**
6	**else**
7	$\sum$ = $\sum_s$;
8	**if** saturatedBloomFilter()== false
9	insertInBloomFilter(SAX_word)
10	**return**;
11	**else**
12	flushBloomFilter()
13	insertInBloomFilter(SAX_word)
14	**return**
15	**endif**
16	**endif**

In line 1, we quantize the input subsequence with the appropriate SAX parameters we discovered in Table 1. We check whether the discretized SAX word exists in the Bloom filter in line 2. If the SAX word exists, then we mark it as a potential instance in **G** and assign weight 1 (line 4). Otherwise, the potential instances in **R** are assigned the weight that determines how likely they are to be discarded from the cache (line 7) and insert the word into the Bloom filter (lines 9 and 13). We check to see whether the Bloom filter is saturated or not (lines 8 and 11). In case the filter *is* saturated, we flush it (line 12).

5.2.2 Cache Maintenance

Using the algorithm outlined above, we 'tag' the instances by a user-defined factor, which determines the relative discard probabilities for *potential* instances **R** and **G**. We describe the algorithm in Table 3.

Table 3: Cache Maintenance Algorithm

Input	**subSeq**, subsequence in question w, size of the cache C, the cache $\sum$, likeliness factor of subSeq to be discarded from the cache **LikelinessBuffer**, the buffer storing $\sum$
Output	**success**, flag indicating successful cache insertion
1	**if** currentCacheElementCount ≤ w //underflow
2	currentCacheElementCount++
3	InsertInCache(subSeq, currentCacheElementCount)
4	//insert $\sum$ in LikelinessBuffer
5	InsertInLikelinessBuffer($\sum$,currentCacheElementCount)
6	success = true
7	**else** //overflow
8	**//Pathological Situation 1**
9	//if all potential instances in **G** are in C
10	**if** sum(LikelinessBuffer) == w
11	flushBloomFilter()
12	flushCache()
13	flushLikelinessBuffer()
14	currentCacheElementCount = 0
15	InsertInCache(subSeq,1)//insert in first location
16	InsertInLikelinessBuffer($\sum$,1)
17	SAX_word= quantizeBySAX(subseq, a, ω)
18	insertInBloomFilter(SAX_word)
19	currentCacheElementCount++
20	**//Pathological Situation 2**
21	**else if** saturatedBloomFilter()==true
22	flushBloomFilter()

```
23      currentCacheElementCount++
24      InsertInCache(subSeq, currentCacheElementCount)
25      InsertInLikelinessBuffer(∑,currentCacheElementCount)
26      SAX_word= quantizeBySAX(subseq, a, ω)
27      insertInBloomFilter(SAX_word)
28    else
29      loc = calculateCacheDiscardLocation(LikelinessBuffer)
30      InsertInCache(subSeq, loc)
31      InsertInLikelinessBuffer(∑,loc)
32    endif
33    success = true
```

If there is no cache overflow, we insert the subsequence into the next available cache slot (lines 1-6). Otherwise, we check for two pathological situations. If the cache is full of potential instances in **G** (which is *extremely* unlikely to occur provided that we have set *a*, ω properly), then we flush the cache, the Bloom filter and the buffer storing the likeliness factor of each cache element, and start from scratch (lines 10-19). We also check for saturation of the Bloom filter, and if we find it, we flush the Bloom filter (lines 21-22). Otherwise, we determine the discard location based on the likeliness factors of the elements and insert instances into the cache accordingly (lines 28-32). We describe the cache discard location algorithm in Table 4.

Table 4: Cache Discard Location Calculation Algorithm

```
Input   LikelinessBuffer, the buffer storing ∑
Output  loc, cache discard location
1   ind = generateRandomIndex([1,sum(LikelinessBuffer)])
2   cumSumArray  = cumulativeSum(LikelinessBuffer)
3   //find the first smallest index greater than or equal
    //to ind in cumSumArray
4   loc = firstSmallestIndex(cumSumArray,ind)
5   return
```

We generate a random index between 1 and the total weight in the LikelinessBuffer (line 1). We calculate the desired discard location so that potential instances in **R** are ∑ times more likely to be discarded (lines 2-4), and return.

6. EXPERIMENTAL EVALUATION

We begin by noting that *all* experiments (including all the figures above) are completely reproducible. All experimental code and data (and additional experiments omitted for brevity) are archived in perpetuity at [33].

The goal of our experiments is to show that our algorithm is more efficient than any obvious strawman technique, and that it is not particularly sensitive to the parameter choices. In addition, we show the utility of our approach with case studies on two real-world datasets.

6.1 Rate of Detection

We begin by comparing our algorithm with the naive RR cache replacement policy. RR is a simple, but highly effective algorithm for many problems.

From the UCR archive we take examples of class 1 from the MALLAT dataset [16] and consider them as target motifs. We randomly embed these into a much longer random walk time series with different occurrence probabilities, and our task is to recover at least one matching pair as soon as possible.

We consider the cache size to be a function of the target patterns' occurrence probability. In particular, our cache can buffer at most only 10% of the rareness of the target motifs. In other words, if our target motifs have 1/100 occurrence probability, then our cache can hold at most only 10 patterns. In addition, we use 3.2, 12 and 16 as the values of *T*, *a*, and *ω*, respectively. Furthermore, we will show that the setting of these parameters is not a black art; there exists a wide range of possible values of the parameters that have no significant impact on the performance of our algorithm.

In order to show how quickly we detect the target patterns, we do the following. We run our sticky cache algorithm and the RR algorithm under the same experimental conditions and record the average number of objects we must process until we get the first true positive (i.e., a pair of patterns *are* in the cache, and our algorithm recognizes this fact).

As we are interested in our algorithm's performance relative to RR, as shown in Figure 17, we plot the results relative to the *mean* performance of RR, with values less than one indicating that our algorithm offers an improvement. In addition, to show the *quality* of our algorithm, we record the number of false positives we see before we attain success.

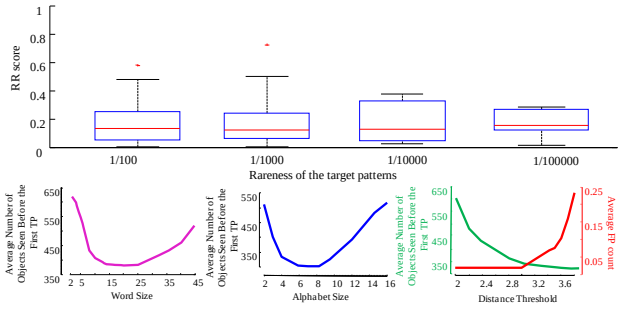

Figure 17: *top*) For the sticky cache algorithm, the average number of objects seen before the first true positive is expressed with respect to the results for RR policy at different rareness factors. *bottom*) The parameter robustness of our algorithm offers a wide range of possible values that have no significant impact on performance (from left to right *ω, a and T*).

From Figure 17.*top*), we can see that in the worst case, the sticky cache algorithm is faster than the RR policy by a factor of 2. In addition to this, because the median score of the sticky cache algorithm remains almost constant (~0.18) for different rareness factors, we can say that on average, our algorithm is ~6 times faster than the RR policy. In particular, on average, the probability that an object is a false positive before we attain success is at most only ~2% [33]. The results above suggest the utility of an *adaptive T* as a function of the rareness factor. We modified our model to achieve this; however, because this is straightforward, we relegate the discussion to [33].

Recall our claim that our algorithm is not sensitive to the parameter choices. In order to show this, we do the following experiment. Assume that our data generator generates the target patterns with a probability of 1/100. Keeping the same experimental setup we discussed above, we vary *ω*, *a* and *T*, and show the result in Figure 17.*bottom*). The results show too conservative or too liberal values of *ω* and *a* result in an increase in the average number of objects we see before the first true positive. In addition, if we make *T* too liberal, then the average false positive count increases and vice versa. However, the results clearly show the existence of a wide range of choice of these parameters which confirms our algorithm is not parameter-sensitive.

6.2 Worst-Case Time Complexity

Consider the worst-case scenario. For each pattern TS_i of length m in question, we make its SAX representation first. This step needs O(m) time. In order to detect whether TS_i is a potential instance in **G** or **R**, we hash it into the Bloom filter. Assume the number of independent hash functions we use in the Bloom filter is h. Given this, querying if the SAX representation of TS_i is in the Bloom filter, and if not, hashing it into the Bloom filter, requires O(h) time. In the worst possible case, the Bloom filter will be saturated, and we flush it. This is a constant-time operation. After this, we insert TS_i into a cache of size w. For an overflowed cache, discarding a cache element requires O(1) time. After we insert TS_i into the cache, we calculate its distance from all cache elements (recall this is the worst-case scenario) until the participating patterns pass the threshold test. This needs O(wm) time. Therefore, the overall worst-case time complexity of our algorithm is O(m) + O(h) + O(wm). In practice, this means our somewhat naive implementation can handle 250Hz under typical parameter settings, and a carefully optimized implementation could easily handle 1,000Hz.

6.3 Case Studies

6.3.1 Wildlife Monitoring

Wildlife monitoring by examining sensor traces has been shown to be a useful tool for measuring the health of the environment [29]. In some cases, we may have a *known* bird call we would like to monitor, but here we consider the more difficult task of detecting previously *unknown* calls. Our only assumptions are that the call will be repeated at least once.

Assume we monitor the audio trace of a ten-hour-long night of a forest [26]. Given a data rate of 62 Hz in the Mel-Frequency Cepstral Coefficients (MFCC) space, we will see about 2.2 million data points. Assume we have a fixed-size memory which can buffer at most only 1/4000 of the subsequences that appear on this night. Our final assumption is that we have a predefined distance threshold for detecting a pair of target patterns (which we learned offline on a handful of known bird calls). Given these assumptions, if a bird calls randomly ten times during this night, we can ask the following question: *What fraction of nights can we expect to detect at least one pair (any pair) of bird calls*? Recall that detecting a *single* pair is sufficient for the wildlife monitoring task. In order to answer this question, we perform the following experiment. In a ten-hour-long audio trace of environmental sounds, we randomly insert ten approximately three-second-long calls of a White Crowned Sparrow (*Zonotrichia leucophrys*) [32]. We run our sticky cache algorithm ten times on this dataset and in each run we continue monitoring until we detect the first true positive pairs in the cache.

Our experimental results tell us we can expect to detect the target bird 98 out of 100 nights. Impressive as these results are, they are somewhat pessimistic. After a careful analysis of the results we discovered that the "false positives" are actually true bird sounds. In Figure 18 we show examples of both the injected bird calls we recovered, and other bird calls (unknown species) we recovered.

We do not report the timing experiments, except to note that we can easily search a dataset with an arrival rate much *faster than real-time* on a laptop, suggesting we could handle real-time even on a resource-limited recording device.

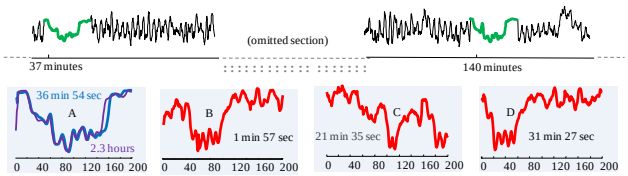

Figure 18: *top*) A snippet of the ten-hour-long audio trace in the MFCC space. The injected bird calls are shown in green/bold. *bottom*) A) The detected motif pairs occurring at 37 minutes and 2.3 hours, respectively. B – D) Three examples of *unknown* bird calls discovered.

6.3.2 Energy Disaggregation

The problem of reducing energy consumption has attracted increasing interest in recent years. To illustrate how our approach may help in solving energy disaggregation problems, we consider one year of energy usage data, containing 0.5 million points [20]. For simplicity, we consider a meter that monitors the electricity usage of just two appliances– a refrigerator and a dishwasher. From personal experience, we assume the dishwasher cycles are approximately 1.5 hours long. As before, we use a cache which buffers at most 5% of the data, and had 20, 12, and 7.5 as values of ω, a, and T, respectively. In order to show how *effective* our algorithm is, we annotate the ground truth by careful human inspection. As soon as our algorithm detects a target motif pair, we flush the cache and continue scanning the dataset. We show the result in Figure 19.

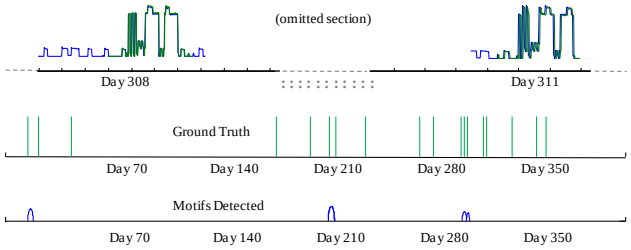

Figure 19: *top*) An excerpt of the electricity usage of a refrigerator and dishwasher. The dishwasher patterns have been marked in green/bold. *bottom*) The ground truth locations (dishwasher usage occurrences) are shown as green arcs. The locations identified by our algorithm are shown as blue arcs.

From Figure 19 we can see that our algorithm detected eight dishwasher motif pairs, most of which are many days apart.

7. RELATED WORK

In recent years researchers have devoted significant attention to efficiently discovering motifs in *static* offline databases [9][19][30]. Until [25], all scalable motif discovery algorithms were *approximate*. In [25], the authors proposed an *exact* motif discovery algorithm which was tenable for a database of millions of objects. Although the worst case time complexity of [25] was quadratic, in practice this is a very pessimistic bound and it was shown to be fast enough to serve as a subroutine for summarization, near-duplicate detection, etc.

Because most data sources are not static, and we may need to deal with unbounded streams, the necessity of *online* discovery of time series motifs has been noted [24]. However the only work devoted to this problem limits its consideration to the last k minutes, for some small k [24]. This means that [24] maintains motifs based on the most recent history of the stream. However, as we noted in our real-world case studies, we may need to find

patterns that repeat hours, days or even weeks apart. For such cases, it is very unlikely that motifs will occur in the same window. In addition to this, if we consider the *huge* volume of data that we wish to process, we are bounded by the scalability of the fastest offline algorithm for this problem [23][25]. Our work is different in a sense that we detect *very* sparse motifs using a *very* limited buffer compared to the size of the data with *very* high probability. In the context of the data *volume*, the interested reader might think of [22], which detects motifs from gigabyte-scale databases. However, [22] is a *multi-pass* algorithm, whereas we explicitly address situations where we can scan the data only *once* to detect motifs.

Various discrete analogues of our problem *have* seen significant research; see [10] and the references therein. In the discrete space the ability to directly test for equality and to directly hash that data, makes the "rare pattern" problem significantly easier. However our use of Bloom filters was inspired by this community's literature [10]. Bloom filters have been an area of active research in the database community for the last decade, with research effort in both applying them to various problems and introducing variants such as "*forgetting*" Bloom filters [10].

8. CONCLUSIONS

We have argued that for most applications, finding the closest pair of subsequences is intractable and *unnecessary*. It suffices to find any pair of repeated patterns. *Any* pair can be used to alert an ornithologist to listen to a snippet of bird calls (cf. Section 6.3.1), or allow a technician to build a "dictionary" of electrical demand patterns (cf. 6.3.2), etc. Based on this observation, we have introduced the first algorithm that can detect repeated patterns in unbounded *real-valued* time series. We have demonstrated efficiency and effectiveness of this algorithm on both synthetic and real-world datasets.

9. REFERENCES

[1] Agrawal, R., Faloutsos, C., and Swami, A. *Efficient Similarity Search in Sequence Databases.* Springer, 1993.

[2] Barrenetxea, G., et al. *Sensorscope: Out-of-the-Box Environmental Monitoring*. IPSN, 2008.

[3] Bhattacharjee, R., Goel, A., and Lotker, Z. *Instability of FIFO at Arbitrarily Low Rates in the Adversarial Queuing Model*. SIAM Journal on Computing 34, no. 2, pp. 318-332, 2005.

[4] Bloom, B. H. *Space/Time Trade-offs in Hash Coding with Allowable Errors*. Communications of the ACM, vol. 13, issue 7, pp. 422 – 426, 1970.

[5] Brown, A. E. X., et. al. *A Dictionary of Behavioral Motifs Reveals Clusters of Genes Affecting Caenorhabditis elegans Locomotion.* Proceedings of the National Academy of Sciences 110, no. 2 pp. 791-796, 2013.

[6] Carton, C. et. al. *Fast Repetition Detection in TV streams Using Duration Patterns*. CBMI, 2013.

[7] Castro, N. C., et al. *Significant Motifs in Time Series*. Statistical Analysis and Data Mining 5, 2012.

[8] Chan, K. P., and Fu, A. W. C. *Efficient Time Series Matching by Wavelets*. Proc' of the 15th IEEE ICDE, 1999.

[9] Chiu, B., Keogh, E., and Lonardi, S. *Probabilistic Discovery of Time Series Motifs.* ACM SIGKDD, 2003.

[10] Cormode, G., and Hadjieleftheriou, M. *Methods for Finding Frequent Items in Data Streams*, VLDB Journal, 19(1), 3-20, 2010.

[11] Dasgupta, D., et. al. *Novelty Detection in Time Series Data Using Ideas from Immunology*. Proc' of the International Conference on Intelligent Systems, 1996.

[12] Hamming, R. W. *Error Detecting and Error Correcting Codes*. Bell System Technical Journal 29, no. 2, 1950.

[13] Hao, Y., Chen, Y., Zakaria, J., Hu, B., Rakthanmanon, T., and Keogh, E. *Towards Never-Ending Learning from Time Series Streams.* Proc' of the 19th ACM SIGKDD, 2013.

[14] Keogh, E., Chakrabarti, K., Pazzani, M., and Mehrotra, S. *Dimensionality Reduction for Fast Similarity Search in Large Time Series Databases*. KAIS 3, pp. 263-86, 2001.

[15] Keogh, E., and Kasetty, S. *On the Need for Time Series Data Mining Benchmarks: A Survey and Empirical Demonstration*. Proc' of the 8th ACM SIGKDD, 2002.

[16] Keogh, E. The UCR Time Series Classification/ Clustering Page: *www.cs.ucr.edu/~eamonn/time_series_data*

[17] Lan, B., Ooi, B. C., and Tan, K. L. *Efficient Indexing Structures for Mining Frequent Patterns*. Proceedings of the IEEE 18th ICDE, pp. 453-462, 2002.

[18] Lin, J., Keogh, E., Wei, L., and Lonardi, S. *Experiencing SAX: A Novel Symbolic Representation of Time Series.* DMKD vol. 15, no. 2, pp. 107 – 144, 2007.

[19] Lin, J., Keogh, E., Lonardi, S., Patel, P. *Finding Motifs in Time Series.* Proc. of the 2nd Workshop on Temporal Data Mining. 2002.

[20] Makonin, S., et. al. *AMPds: A Public Dataset for Load Disaggregation and Eco-Feedback Research*. EPEC, pp. 1-6. 2013.

[21] Mitzenmacher, M., and Upfal, E. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[22] Mueen, A., et al. "A Disk-aware Algorithm for Time Series Motif Discovery." *Data Mining and Knowledge Discovery* 22.1-2, 2011.

[23] Mueen, A. *Enumeration of Time Series Motifs of All Lengths.* Proc'of the IEEE ICDM, pp. 547-556, 2013.

[24] Mueen, A., and Keogh, E. *Online Discovery and Maintenance of Time Series Motifs.* Proc' of the 16th ACM SIGKDD, 2010.

[25] Mueen, A., Keogh, E., Zhu, Q., Cash, S., and Westover, M. B. *Exact Discovery of Time Series Motifs*. Proc'of the 9th SIAM SDM, pp. 473-484, 2009.

[26] URL *http://www.youtube.com/watch?v=ndL6m5vHVhw*

[27] Pietracaprina, A., Riondato, M., Upfal, E., and Vandin, F. *Mining Top-K Frequent Itemsets Through Progressive Sampling*. DMKD, vol 21, no. 2, 2010.

[28] Smith, J. E., and Goodman, J. R. *Instruction Cache Replacement Policies and Organizations*. IEEE Transactions on Computers, 1985.

[29] Trifa, V., et. al. *Automated Wildlife Monitoring Using Self-Configuring Sensor Networks Deployed in Natural Habitats*, 2007.

[30] Vahdatpour, A., et. al. *Toward Unsupervised Activity Discovery Using Multi-Dimensional Motif Detection in Time Series.* IJCAI. Vol. 9. 2009.

[31] Wang, X., et al. *Experimental Comparison of Representation Methods and Distance Measures for Time Series Data*. DMKD, vol. 26, issue 2, pp. 275 – 309, 2013.

[32] Xeno-canto, Sharing Bird Sounds from Around the World, *www.xeno-canto.org/*, accessed on Feb 11, 2014.

[33] Project website: www.cs.ucr.edu/~nbegu001/RareMotif.htm