

Real-Time Classification of Streaming Sensor Data

Shashwati Kasetty, Candice Stafford^a, Gregory P. Walker^a, Xiaoyue Wang, Eamonn Keogh

Department of Computer Science and Engineering,

Department of Entomology^a

University of California, Riverside

{kasettys, xwang, eamonn}@cs.ucr.edu, {staffc01, gregory.walker}@ucr.edu

Abstract

The last decade has seen a huge interest in classification of time series. Most of this work assumes that the data resides in main memory and is processed offline. However, recent advances in sensor technologies require resource-efficient algorithms that can be implemented directly on the sensors as real-time algorithms. We show how a recently introduced framework for time series classification, *time series bitmaps*, can be implemented as efficient classifiers which can be updated in constant time and space in the face of very high data arrival rates. We describe results from a case study of an important entomological problem, and further demonstrate the generality of our ideas with an example from robotics.

1. Introduction

The last decade has seen a huge interest in classification of time series [12][8][14]. Most of this work assumes that the data resides in main memory and is processed offline. However recent advances in sensor technologies require resource-efficient algorithms that can be implemented directly on the sensors as real-time algorithms. In this work we show how a recently introduced framework for time series classification, *time series bitmaps* [14], can be implemented as ultra efficient classifiers which can be updated in constant time in the face of very high data arrival rates. Moreover, motivated by the need to be robust to concept drift, and to spot new behaviors with minimal lag, we show that our algorithm can be amnesic and is therefore able to discard outdated data as it ceases to be relevant.

In order to motivate our work and ground our algorithms we begin by presenting a concrete application in entomology which we will use as a running example in this work. However in our experiments we will consider a broader set of domains and show results from applications across various fields.

1.1 Monitoring Insects in Real-Time

In the arid to semi-arid regions of North America, the beet leafhopper (*Circulifer tenellus*), shown in Figure 1, is the only known vector (carrier) of curly top virus, which causes major economic losses in a number of crops including sugarbeet, tomato, and beans [7]. In order to mitigate these financial losses, entomologists at the University of California, Riverside are attempting to model and understand the behavior of this insect [19].

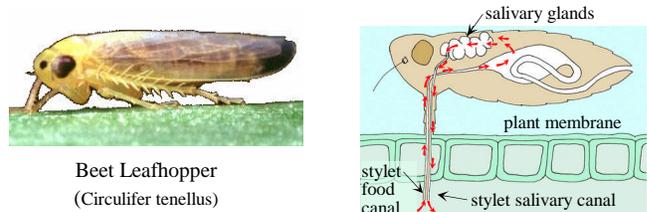


Figure 1: *left*) The insect of interest. *right*) Because of the circulatory nature of the insects feeding behavior, it can carry disease from plant to plant

It is known that the insects feed by sucking sap from living plants, much like the mosquito sucks blood from mammals and birds. In order to understand the insect's behaviors, entomologists can glue a thin wire to the insect's back, complete the circuit through a host plant and then measure fluctuations in voltage level to create an Electrical Penetration Graph (EPG) as shown in Figure 2.

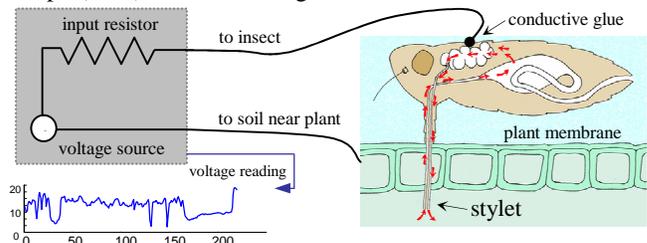


Figure 2: A schematic diagram showing the apparatus used to record insect behavior

This method of recording the insect's behavior appears to be capturing useful information. That is to say, skilled entomologists have been able to correlate various behaviors they have observed by directly watching the insects, with simultaneously recorded time series. However the abundance of such data opens a plethora of questions, including:

Can we automatically detect when the beet leafhopper is in a certain phase such as pathway or phloem ingestion?

Being able to detect these phases automatically would save many hours of time spent by entomologists to analyze the EPGs manually. This could open avenues for detecting new behaviors that entomologists have not been able to model thus far. To be truly real-time, the scheme must be algorithmically time and space efficient in order to deal with the high data rates sensed by the sensors. It should be able to detect patterns in data as the data is sensed.

We propose to tackle this problem using Time Series Bitmaps (TSB) [14]. In essence TSBs are a compact summary or signature of a signal. While TSB's have been shown to be

useful for time series classification in the past [14][13], the fundamental contribution of this work is to show that we can maintain TSBs in constant time, allowing us to deal with very high rate data.

We defer a discussion of TSBs until Section 3; however, Figure 3 gives a visual intuition of them, and their utility.

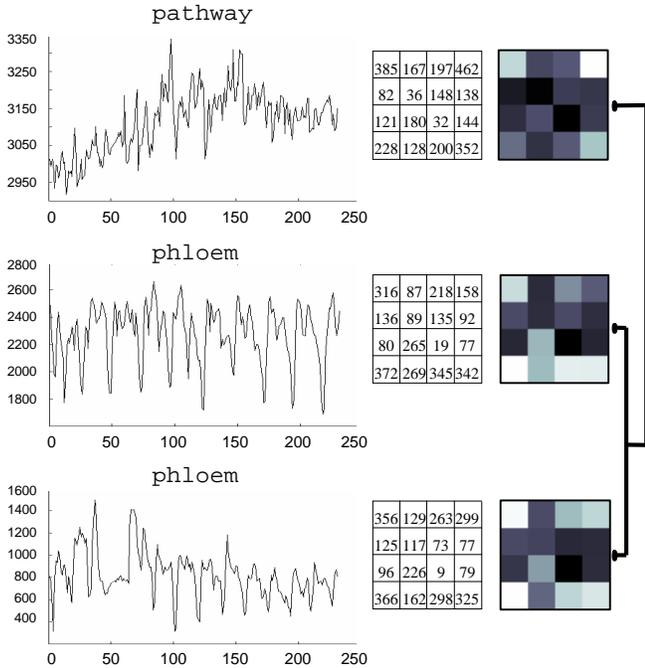


Figure 3: Three traces of insect behavior clustered using time series bitmaps. The square matrices are raw counts. The bitmaps correspond to these raw counts scaled between 0 and 256 then mapped to a colormap. Note that the y-axis values are relative, not absolute, voltage values.

The raw signals have information extracted from them regarding the frequencies of short “sub-patterns”. These raw counts of sub-patterns are recorded in a square matrix, and the Euclidean distance between these matrices can effectively capture the similarity of the signals, which can then be used as an input into classification, clustering and anomaly detection algorithms [13][14]. While it is not necessary for classification algorithms, we can optionally map the values in the matrices to a colormap to allow human subjective interpretation of similarity, and higher level interpretation of the data.

Time Series Bitmaps require only a small amount of memory to store the values of the square matrix. Since these square matrices are updated in real-time, the amount of memory needed is a small constant. Furthermore, as we shall show, the operations on these matrices are also done in constant time.

2. Background and Related Work

To the best of our knowledge, the proposed method of maintaining Time Series Bitmaps (TSBs) in constant time and space per update is novel. Work has been done towards deploying algorithms on sensors that use the Symbolic Aggregate Approximation (SAX) representation [22][17], and

as we shall see, SAX is a subroutine in TSBs, however, neither of the two works uses TSBs.

TSBs are aggregated representations of time series. Another aggregation scheme is presented in [15], where data maps are created that represent the sensory data as well as temporal and spatial details associated with a given segment of data. However, these data maps are not analyzed in real-time, but deposited at sink nodes that are more powerful for pattern analysis.

In [21], the authors introduce an anomaly detection program using TSBs and SAX. However, the authors do not update the TSBs in constant time, but recalculate them from scratch for every TSB. In our work, we introduce a way to maintain these TSBs in constant time without having to recalculate them from scratch, saving time that makes our algorithm truly real-time. Moreover, we tackle the problem of classification, while [21] provides an algorithm for anomaly detection.

Finally, there are dozens of papers on maintaining various statistics on streaming, see [4] and the references therein. However none of these works address the task maintaining a class prediction in constant time per update.

3. Review of SAX/Bitmaps

For concreteness we begin with a review of the time series bitmap representation. For ease of exposition, we begin with an apparent digression: How can we summarize long DNA strings in constant space?

Consider a DNA string, which is a sequence of symbols drawn from the alphabet $\{A, C, G, T\}$. DNA strings can be very long. For example the human mitochondrial DNA has 16,571 such symbols, beginning with $GATCACGGTCTATCACCC\dots$ and ending with $\dots ACATCACGATG$. Given the great length of DNA strings a natural question is how can we summarize them in a compact representation? One approach would be to map a DNA sequence to a matrix of four cells based on the frequencies of each of the four possible base pairs. This produces a numeric summary; we can then further map the observed frequencies to a linear colormap to produce a visual summary as shown in Figure 4.

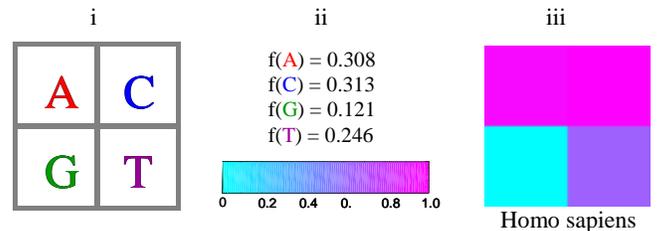


Figure 4: i) The four DNA base pairs arranged in a 2 by 2 grid. ii) The observed frequencies of each letter can be indexed to a colormap as shown in iii.

Note that in this case the arrangement of the four letters is arbitrary, and that the choice of colormap is also arbitrary.

We begin by assigning each letter a unique key value, k :

A	→	0	C	→	1	G	→	2	T	→	3
---	---	---	---	---	---	---	---	---	---	---	---

We can control the desired number of features by choosing l , the length of the DNA words. Each word has an index for the location of each symbol, for clarity we can show them explicitly as subscripts. For example, the first word with $l = 4$ extracted from the human mitochondrial DNA is $G_0A_1T_2C_3$. So in this example we would say k_0 is **G**, $k_1 = \mathbf{A}$, $k_2 = \mathbf{T}$ and $k_l = \mathbf{C}$.

To map a word to a bitmap we can use the following equation to find its row and column values:

$$col = \sum_{n=0}^{l-1} (k_n * 2^{l-n-1}) \bmod 2^{l-1}, \quad row = \sum_{n=0}^{l-1} (k_n \text{div} 2) * 2^{l-n-1}$$

Figure 5 shows the mapping for $l = 1, 2$ and (part of) 3.

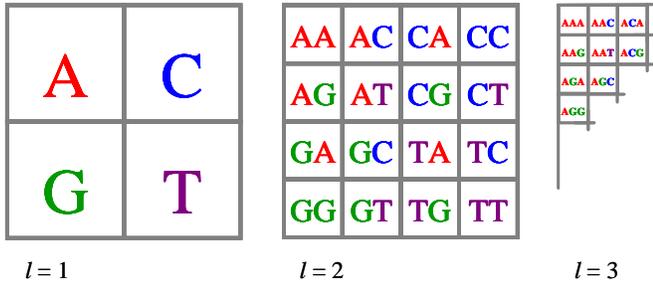


Figure 5: The mapping of DNA words of $l = 1, 2$ and 3. (The colors of the text are just to allow visualization of the mapping).

If one examines the mapping in Figure 5, one can get a hint as to why a bitmap for a given species might be self-similar across different scales. For example note that for any value of l , the top column consists only of permutations of A and C, and that the two diagonals consist of permutations of A and T, or G and C. Similar remarks apply for other rows and columns.

In the rest of this paper, we use the alphabet $\{a,b,c,d\}$ and we choose to use bitmaps of size 4×4 or $l = 2$. Figure 6 below was created using this alphabet, and a different colormap than the DNA example. The icons shown here were generated from a subsequence of a non-probing behavior waveform of the beet leafhopper. Refer to section 4 for details on each beet leafhopper behavior.

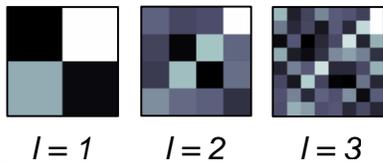


Figure 6: The icons created for a subsequence of a non-probing behavior waveform for the beet leafhopper at every level from $l = 1$ to 3.

Having shown how we can convert DNA into a bitmap, in the next section we show how we can convert real-valued time series into pseudo DNA, to allow us to avail of bitmaps when dealing with sensors.

3.1 Converting Time Series to Symbols

While there are at least 200 techniques in the literature for converting real valued time series into discrete symbols [1], the SAX technique of Lin et. al. is unique and ideally suited

for our purposes [16]. The SAX representation is created by taking a real valued signal and dividing it into equal sized sections. The mean value of each section is then calculated. This produces a reduced dimensionality piecewise constant approximation of the data. This representation is then discretized in such a manner as to produce a word with approximately equi-probable symbols. Figure 7 shows the first 64 data points of the phloem phase waveform in the bottom of Figure 3 after converting it to a discrete string.

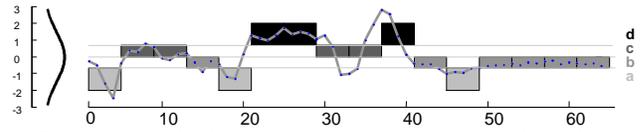


Figure 7: A real valued time series being discretized into the SAX word `accbaddccdbabbbb`.

Note that because we can use SAX to convert time series into a symbolic string with four characters, we can then trivially avail of any algorithms defined for DNA, including the bitmaps introduced in the last section.

SAX was first introduced in [16], and since then it has been used to represent time series in many different domains including automated detection and identification of various species of birds through audio signals [9], and analysis of human motion [2], telemedicine and motion capture analyses.

4. Our Algorithm in Context

We demonstrate our results on a case study for an important entomological problem, and then further demonstrate the generality of our ideas with an example from robotics.

Our algorithm uses SAX (Symbolic Aggregate Approximation), a symbolic representation for time series data [10], to summarize sensor data to a representation that takes up much less space, yet captures a signature of the local (in time) behavior of the time series. We further compact the data by aggregating the SAX representations of segments of data to create a square matrix of fixed length or Time Series Bitmaps (TSBs) [14][13].

We introduce novel ways to maintain these TSBs in constant time. These optimizations make our algorithm run significantly faster, use very little space, and produce more accurate results, while being amnesic and using the most recent and relevant data to detect patterns and anomalies in real-time. With these improvements in time and space requirements, this algorithm can be easily ported to low-power devices and deployed in sensor networks in a variety of fields.

4.1 Entomology Case Study

As noted in Section 1.1, entomologists are studying the behavior of beet leafhopper (*Circulifer tenellus*) by gluing a thin wire to the insect's back, completing the circuit through a host plant and then measuring fluctuations in voltage level to create an Electrical Penetration Graph (EPG). This method of recording the insect's behavior appears to be capturing useful information. Skilled entomologists have been able to correlate various behaviors they have observed by directly watching the

insects, with simultaneously recorded time series. However, the entomologists have been victims of their own success. They now have many gigabytes of archival data, and more interestingly from our perspective, they have a need for *real-time* analyses.

For example, suppose an entomologist has a theory that the presence of carbon dioxide can suppress a particular rarely seen but important insect behavior. In order to prove this theory, the entomologist must wait until the behavior begins, then increase the concentration of carbon dioxide and observe the results. If we can successfully classify the behavior in question automatically, we can conduct experiments with hundreds of insects in parallel, if we cannot automate the classification of the behavior, we are condemned to assigning one entomologist to watch each insect – an untenable bottleneck. Before giving details of our algorithm in Section 4.2, we will provide some examples and illustrations of the types of behaviors of interest.

4.2 Characteristic Behaviors to Classify

The beet leafhopper’s behavior can be grouped into 3 phases of feeding behavior and 1 phase of non-probing or resting behavior, making this a classification problem of 4 classes. There are several other behaviors of the beet leafhopper which have not yet been identified, and we will exclude these from our experiments.

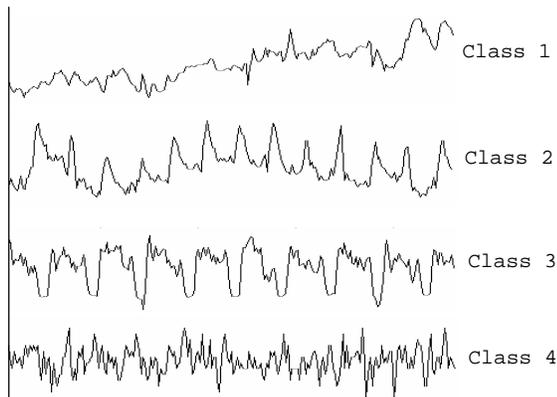


Figure 8: Examples of the Different Waveforms

The original measurements were made in terms of voltage. However, only the relative values matter, so no meaning should be attached to the absolute values of the plot. Note that we have made an effort to find particularly clean and representative data for the figure. In general the data is very complex and noisy.

4.2.1 Class 1 – Pathway. There is no ingestion in this phase but it is believed that it occurs prior to other ingestion behaviors. During the initial stages of feeding, pathway waveforms are produced. There are several variations of pathway phase waveforms, each of which have varied characteristics. One variation of pathway is quite similar to phloem ingestion and non-probing behavior in that it is characterized by low amplitude fluctuations, which makes this variation difficult to classify. In our work, we will consider all the variations together as 1 general pathway phase behavior. An example pathway phase waveform is shown in Figure 8.

4.2.1 Class 2 – Phloem Ingestion. In this phase, the beet leafhopper is seen to be ingesting phloem sap. The waveforms in this phase are known to have low amplitude fluctuation and occur at low voltage levels. There are varied behaviors among phloem ingestion phase waveforms; however, in this work we only classify the general phloem ingestion phase which encompasses all sub-behaviors. An example phloem ingestion phase waveform is shown in Figure 8. Note that this particular waveform has characteristic reoccurring “spikes”, but the mean value can wander up and down in a very similar manner to the *wandering baseline* effect in cardiology [3]. This drift of the mean value has no biological meaning, neither in cardiology nor here. However, as we shall see, this wandering baseline effect can seriously degrade the performance of many classic time series classification techniques.

4.2.3 Class 3 – Xylem/Mesophyll Ingestion. In this phase, the beet leafhopper is seen to be ingesting xylem sap. Occasionally, it is seen to be ingesting mesophyll sap. However, the waveforms of the two are indistinguishable. For entomologists, this phase is easiest to recognize since visually the waveform has very characteristic and typical features. The voltage fluctuation has high amplitude and a very regular repetition rate. An example xylem / mesophyll ingestion waveform is shown in Figure 8.

Class 4 – Non-Probing / Resting. In this phase, the beet leafhopper is resting on the surface of the leaf. Sometimes the beet leafhopper may move around, and the insect’s walking and grooming behaviors can cause large fluctuations in voltage level. Usually, when the insect is resting, the fluctuation levels are low and somewhat flat. An example non-probing phase waveform is shown in Figure 8.

4.3 Classification in Real-Time

With the abundance of data available, it becomes impractical and time consuming for a human to analyze and classify each of these behaviors visually. If we can automatically classify this behavior using an efficient classification algorithm, it could save many hours of the entomologists’ time. The benefits multiply if the behaviors can be captured in real-time as they occur, without having to record many hours of data for offline processing later.

Our algorithm is able to handle this data despite it being erratic and unpredictable, which is the case with most sensor data. We outline and describe our algorithm below, and in Section 5, we show results from experiments in which we consider streaming data, and classify the behavior that is occurring in real-time.

4.4 Maintaining TSBs in Constant Time

While SAX [10] forms the basis of our algorithm and we use TSBs [14] to aggregate the raw data, the novel contribution of our work is the way we maintain the TSBs in constant time, enabling tremendous improvement in the input rate we can handle, as well as opening up the possibility of creating an efficient classifier that can be deployed in low-power devices. These improvements allow us to process data at a high rate, while still classifying and producing results in real-time.

Because long streams of data kept in memory will become outdated and meaningless after some time, and must be discarded periodically, our algorithm is amnesic, maintaining a certain constant amount of history of data at all times. This is especially useful when there is a continuous stream of data in which there are transitions from one class to another. Our algorithm can capture these changes since the classifier is not washed out by hours of outdated data that is not relevant to the current state. Furthermore, by choosing an appropriate window length it can capture these changes with minimal lag. The pseudocode for maintaining TSBs in constant time and classifying them is outlined in Table 1.

Table 1: Maintaining TSBs in Constant Time

	Function
1	classifyTSBs(N,n,a,historySize)
2	historyBuffer[historySize][n]
3	curTimeSeries[N]
4	curTSB[a times a] = 0
5	input = getInput()
6	while curTimeSeries.size() < N and
7	input != EOF:
8	curTimeSeries.append(input)
9	input = getInput()
10	curSAXWord = sax(curTimeSeries,N,n,a)
11	incrementTSB(curTSB,curSAXWord)
12	historyBuffer.append(curSAXWord)
13	while historyBuffer.size() < historySize and
14	input != EOF:
15	curTimeSeries.pop()
16	curTimeSeries.append(input)
17	curSAXWord = sax(curTimeSeries,N,n,a)
18	incrementTSB(curTSB,curSAXWord)
19	historyBuffer.append(curSAXWord)
20	input = getInput()
21	classify(curTSB)
22	while input != EOF:
23	curTimeSeries.pop()
24	curTimeSeries.append(input)
25	curSAXWord = sax(curTimeSeries,N,n,a)
26	removedWord = historyBuffer.pop()
27	decrementTSB(curTSB,removeWord)
28	historyBuffer.append(curSAXWord)
29	incrementTSB(curTSB,curSAXWord)
30	classify(curTSB)
31	input = getInput()

The input parameters to this algorithm are the 3 SAX parameters of N, n and a, along with the historySize parameter. The algorithm begins by creating two circular arrays that will hold the current data being processed and stored (lines 1-2). The curTSB array of size a times a will hold the Time Series Bitmap counts.

The curTimeSeries array (line 2) holds the current sliding window of data that will be converted to a SAX word. The SAX parameters that we use in the beef leafhopper problem are N=32 and n=16. This corresponds to a sliding window size of 32 data points that will be converted to a SAX word of 16 characters. These parameters are fixed constants in our algorithm but can be changed for other applications if necessary, although fine-tuning parameters too much could lead to a problem of overfitting the data [12]. The alphabet size use is a=4 (lines 8, 14, 22) in order to produce a square time series bitmap of size 4x4 stored here as an array of size 16 (line 3).

The historyBuffer (line 1) is a two dimensional array that will hold the most recent SAX words in it. The number of SAX words it holds is specified by historySize. We have

fixed this to be 200 in our implementation for the beet leafhopper. We estimated this by visually inspecting graphs and noticing that it is a large enough timeframe of data to indicate the type of behavior being exhibited. If more points are needed to classify, the historySize can be increased. Conversely, if less points are needed or if memory is severely scarce, the historySize can be decreased. We have refrained from fine tuning this parameter to prevent overfitting but our experiments suggest that we can obtain good results over a relatively large range of values [12].

Initially, the two buffers need to be filled as long as there is input. This is done in the first two while loops. Every time the curTimeSeries buffer is filled, it needs to be converted to a SAX word, and curTSB needs to be updated by incrementing the appropriate positions. Once the historyBuffer is filled, we can proceed with real-time classification. The first TSB representing the initial historyBuffer is classified (line 18). Then for each new input, the TSB is computed and classified (line 27). The classifier we use is one nearest neighbor with Euclidean distance between the TSBs as the distance measure [14].

4.5 Optimizations in Time and Space

The optimizations we propose to save time and space arise from the observation that a new TSB need not be created for each new input. We can update the TSB by removing the oldest SAX word in historyBuffer, decrementing the appropriate fields in the TSB for the removed word, appending the newest word to the historyBuffer, and updating the TSB by incrementing its fields for the new word. Similarly, curTimeSeries need not be refilled each time there is a new input. The oldest value in curTimeSeries can be removed and the new input can be added. Note that in the implementation, curTimeSeries and historyBuffer need to be circular arrays in order to perform updates in constant time. Figure 9 illustrates lines 19-28.

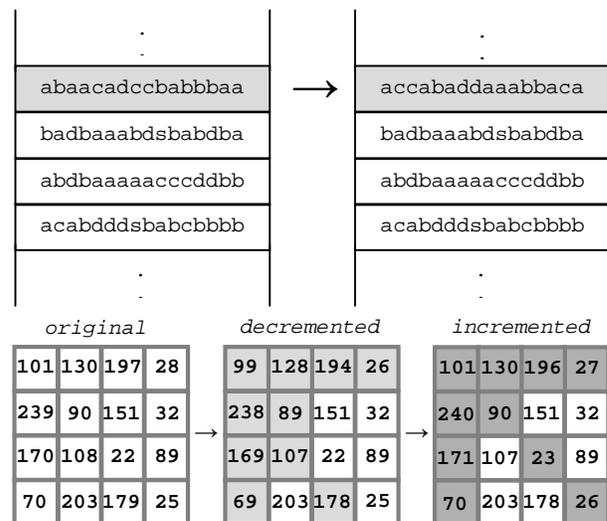


Figure 9: Maintaining TSBs in Constant Time

Figure 9 shows the status of the historyBuffer before and after a new input is processed. The new input mapped to

the SAX word `accabaddaaabbaca`, takes the place of the oldest SAX word in the circular array `historyBuffer`, which in this case is the word `abaacadccbabbbaa`. This change needs to be reflected in the array `curTSB`. Figure 9 shows the status of the `curTSB` after the change. The appropriate values of the substring counts for the substrings in the SAX word being removed need to be decremented. Then, the values need to be incremented for the substrings of the new SAX word added.

4.6 Training the Classifier

To classify a given segment of data into a particular class, we need to create reference TSBs. After preprocessing the data and performing the necessary conversions of format to plain ASCII text, we proceed to convert the streams of data to bitmaps using the same algorithm as in Table 1, and store these bitmaps in a file.

We could use all the annotated data as training data, however this has two problems. First, the sheer volume of data would make both the time and space complexity of nearest neighbor classification untenable. Second, the training data is noisy and complex and it may have mislabeled sections. Our solution is to do data editing, also known as prototype selection, condensation, etc [18]. In essence, our idea is to cluster the data and choose a small number of cluster centers as prototypes for the dominant class in that cluster.

We begin by randomly sampling the data to reduce the training set size while still maintaining the original distribution of the data. Once we have randomly sampled, we can cluster the data to find good representative models for each class to use in the classification. Since the Time Series Bitmap is an aggregation of time series subsequences, it is necessary that the bitmaps be randomly selected to avoid the problems that arise with clustering time series subsequences [11]. We proceed to cluster the bitmaps by using KMeans. The best centroids (i.e the ones that have the purest class labels) are computed for each class and these centroids make up the final training classifiers that are provided to the real-time algorithm in Table 1. The pseudocode for finding the training TSBs for each class X is presented in Table 2.

Table 2: Finding Training TSBs for Each Class

1	Function findOptimalClusters(TSBsClassX)
2	TSBsClassX = minMaxNormalize(TSBsClassX)
3	distances[99]
4	centroidLists[99]
5	for k = 2 : 100
6	[clusters,sumd] = kmeans(TSBsClassX,k)
7	minDist = sum(sumd)
8	minClusters = clusters
9	for i = 2 : 5
10	[clusters,sumd] = kmeans(TSBsClassX,k)
11	if sum(sumd) < minDist
12	minDist = sum(sumd)
13	minClusters = clusters
14	distances.append(minDist)
15	centroidLists.append(clusters)
16	for i = 1 : 98
17	curDist = distances[i]
18	if (curDist - distances[i+1])
19	/ curDist < 0.01
20	return i,centroidLists[i]
	return null,null

Before beginning the KMeans clustering, we first normalize the data using min-max normalization, in which every TSB is scaled to be between 0 and 255 (line 1). For this part of the algorithm, it would suffice to scale between 0 and 1, but since the TSBs could potentially be mapped to a colormap for visualization, it is more suitable to scale it between 0 and 255. The results would be the same regardless of which scaling is used as long as the data is min-max normalized.

Cluster sizes from 2 to 100 are tested (lines 4-14), and each test is run 5 times (lines 8-12). For each cluster size, the resulting centroids from the best test run are recorded (lines 11-14). The best test run is the one in which the sum of the distances from each instance to the nearest centroid is the lowest. The next step is to find the best number of clusters. Although the sum of the distance to the nearest centroid will decrease monotonically as the number of clusters increases, the distance change becomes negligible after some time. It is more efficient to choose fewer number of clusters since it reduces the size of the final training set created from the centroids of these clusters. To find the best cluster size, we compute the difference in distance sum between two consecutive cluster sizes starting from cluster size 2, and terminate the search when this difference is less than 1% (lines 17-18). The best cluster size and the corresponding centroids for that cluster size are returned.

There may be rare cases when the algorithm does not find such a cluster size, and in that case the return values would be null. In such a case, the difference threshold of 1% can be increased, and the algorithm can be run again.

5. Experimental Results

In this section we describe detailed results from the beet leafhopper problem, and in order to hint at the generality of our work, we briefly present results from experiments on a robot dataset. To aid easy replication of our work, we have placed all data at: <http://acmserver.cs.ucr.edu/~skasetty/classifier>.

5.1 Beet Leafhopper Behavior Dataset

The classification results of the beet leafhopper behavior problem largely agree with expectations. Our algorithm classifies classes known to be easy with high accuracy, and does reasonably well on more difficult classes. The results are presented in Table 3.

Table 3: Classification Accuracies
Beet Leafhopper Problem

Class	Accuracy	# of TSBs
Pathway	42.56%	610,538
Phloem	64.93%	1,241,560
Xylem/Mesophyll	71.94%	1,432,586
Non-Probing	95.03%	412,130
Overall	67.31%	3,696,814
<i>Default (Overall)</i>	<i>38.75%</i>	<i>3,696,814</i>

Note that we classified all 3,696,814 examples available, without throwing out the difficult or noisy cases.

As described in Section 4, the pathway phase behavior has several variations, and in our classification, we grouped all of these sub-behaviors together as a single pathway phase behavior. The waveforms of these sub-behaviors vary, so it is expected that the classification accuracy may not be as high as the other classes. Similarly, the phloem ingestion phase behavior has several varieties, and we grouped these together as well. However, the phloem phase behavior is classified correctly 64.93% of the time, which is much higher than the phloem phase accuracy of 42.56%. The xylem/mesophyll ingestion phase is easiest for entomologists to detect, and as expected, our classifier mirrors this, classifying accurately 71.94% of the data. The non-probing behavior is clearly different from the other three behaviors, because the insect is simply resting on the leaf, moving around or grooming during this phase. As expected, it was easiest to detect this behavior, with a classification accuracy of 95.03%.

We compared our algorithm with several competitors, including the following using the min-max normalized time series subsequences: Euclidean distance [11], the distance between the energy of the two Fourier coefficients with the maximum magnitude [6], the distance between the energy of the first 10 Fourier coefficients [6], and the difference in variance. We made every effort to find the best possible settings for competitors that have parameters.

Due to the slow running times of the other algorithms, we reduced the size of the test set by randomly selecting 1% of the testing data from each class. Since the training data was smaller (with 15,015 instances), we selected 10% of this data randomly to create the new training set. The results of all the classes together are presented in Table 4.

Table 4: Accuracy Comparison Beet Leafhopper Problem

Classifier	Accuracy
Default Rate	38.75%
TSBs with KMeans	68.94%
Euclidean Distance	41.84%
distFFT with maxMag	41.59%
distFFT with firstCoeff	42.41%
Variance Distance	40.65%

It is clear that overall, across all classes, our algorithm performs much better than the other algorithms we tested it against. It beats the second best algorithm by more than 26%. The distribution of the 4 classes in the test set is not equal. After sampling 1% from each class, we have 6,105 instances for pathway, 12,416 instances for phloem ingestion, 14,326 instances for xylem or mesophyll ingestion and 4,121 instances for non-probing. Therefore, the default classification rate is 38.75%. Clearly, the other algorithms are only marginally better than default. Our algorithm beats the default rate by more than 30%.

We used this downsampled data on our algorithm to create the confusion matrix in Figure 10. The diagonal shows the accuracy of our classifier on each class. The pathway phase is the only behavior our algorithm does not accurately classify

above the default rate. The rest of the classes classify well above the default rate.

		Test Class			
Predicted Class		Pathway	Phloem	Xylem / Mesophyll	Non-Probing
	Pathway	34.37%	12.22%	18.74%	2.84%
	Phloem	23.54%	72.72%	6.76%	2.81%
	Xylem / Mesophyll	6.96%	4.55%	73.12%	0.07%
	Non-Probing	35.14%	10.51%	1.39%	94.27%

Figure 10: Confusion matrix of test class versus predicted class

This confusion matrix illuminates several complexities and characteristics of this dataset that make classifying it particularly challenging. It is interesting to note that very rarely is a class misclassified as the xylem / mesophyll ingestion behavior. This is because the xylem / mesophyll ingestion phase has a very distinct waveform characterized by constant repetition and high voltage fluctuations as we described in Section 4.

The pathway waveforms are particularly difficult to model, and therefore, difficult to classify correctly due to the high variations in the sub-behaviors of this class. They are misclassified at a very high rate as non-probing or phloem ingestion. This is because there is a particular variation of pathway that has the low amplitude voltage fluctuation characteristic of non-probing (during resting) and phloem ingestion waveforms, and this variation of pathway is the most frequent in our dataset.

It is natural that the non-probing or resting behavior waveforms classify correctly at a very high percentage (94.27% in this case) since the other behaviors are all related to feeding. Although pathway waveforms are some times misclassified as non-probing behavior, the converse is not true. We attribute this to the low number of variations within the non-probing behavior class. The algorithm only needs to model two types of waveforms for this class. The waveforms are somewhat flat with low voltage fluctuations when the insect is resting, and high fluctuations are typical when the insect is grooming. On the other hand, the pathway class has four distinct sub-behaviors making it much more difficult to model. Moreover, as mentioned above, the most confusing variation of pathway waveforms is the most frequent variation in our dataset.

The data used to generate the results in Figure 10 follow a similar overall trend as the results in Table 3 generated from running our algorithm on a test set 100 times as large and a training set 10 times as large. The larger the dataset, the more difficult it is to classify due to the unpredictable and erratic behavior in sensor data. Here, we can see that our algorithm scales well and maintains accuracy rates overall as the dataset grows in size.

5.2 Robot Dataset

To illustrate the generality of our algorithm, we have run additional experiments on different datasets using a similar

setup and procedure as for the beet leafhopper behavior classification problem. The same parameters were used as well.

The Sony AIBO is a small quadruped robot that comes equipped with a tri-axial accelerometer. We obtained accelerometer data for the AIBO moving on various surfaces: on a field, on a lab carpet and on cement [20]. We applied our algorithm to this dataset to see if it could detect which surface the robot was moving on for a given waveform. Like the beet leafhopper dataset, we passed the streams of data for each surface to generate the TSBs, ran the training algorithm in Table 2 on randomly sampled TSBs from the training data streams, and classified the TSBs using the one nearest neighbor algorithm with Euclidean distance between the TSBs as the distance measure. Table 5 shows the results.

Table 5: Classification Results from Robot Dataset

	Accuracy
Default Rate across 3 classes	38.42%
X-Axis Data across 3 classes	73.36%
Y-Axis Data across 3 classes	60.84%
Z-Axis Data across 3 classes	62.97%

Like the beet leafhopper dataset, the distribution of the number of data points in each class of the robot accelerometer dataset is also unequal. The default accuracy rate was calculated to be 38.42%. For all three, the x-axis, y-axis and z-axis acceleration data, our algorithm clearly beats the default rate, with the x-axis data being most easy to classify.

7. Conclusion

In this work, we have introduced a novel way to update Time Series Bitmaps in constant time. We have demonstrated that an amnesic algorithm like the one we propose can accurately detect complicated patterns in the erratic sensor data from an important entomological problem. Our algorithm is fast and accurate, and optimized for space. We have also described the wide range of applications for our algorithm by demonstrating its effectiveness in classifying robotic data. Future work includes deployment on low-power sensors and further applications in other domains.

8. References

- [1] J. Almeida, J. Carrico, A. Maretzek, P. Noble, and M. Fletcher. Analysis of genomic sequences by Chaos Game Representation. In *Bioinformatics*, 17(5):429-37, 2001.
- [2] V. Bhatkar, R. Pilkar, J. Skufca, C. Storey, and C. Robinson. Time-series-bitmap based approach to analyze human postural control and movement detection strategies during small anterior perturbations. *ASEE, St. Lawrence Section Conference*, 2006.
- [3] L. Burattini, W. Zareba, and R. Burattini. Automatic detection of microvolt T-wave alternans in Holter recordings: effect of baseline wandering. *Biomedical Signal Processing and Control* 1(2), pp. 162-168, 2006.
- [4] M. Datar, A. Gionis, P. Indyk, R. Motwani. Maintaining Stream Statistics over Sliding Windows. In *Proc. SIAM-ACM Symp. on Discrete Algorithms*, pp. 635-644, 2002.
- [5] C. Daw, C. Finney, and E. Tracy. A review of symbolic analysis of experimental data. In *Review of Scientific Instruments*, volume 74, no. 2, pages 915-930, 2003.
- [6] G. Janacek, A. Bagnall, and M. Powell. A likelihood ratio distance measure for the similarity between the Fourier transform of time series. *PAKDD*, 2005.
- [7] S. Kaffka, B. Wintermantel, M. Burk, and G. Peterson. Protecting high-yielding sugarbeet varieties from loss to curly top, 2000. <http://sugarbeet.ucdavis.edu/Notes/Nov00a.htm>
- [8] K. Kalpakis, D. Gada, and V. Puttagunta. Distance measures for effective clustering of ARIMA time-series. In *Proc. of the 1st IEEE International Conference on Data Mining*, 2001.
- [9] E. Kasten, P. McKinley, and S. Gage. Automated Ensemble Extraction and Analysis of Acoustic Data Streams. *ICDCS Systems Workshops*, 2007.
- [10] E. Keogh, J. Lin, and A. Fu. HOT SAX: Efficiently finding the most unusual time series subsequence. In *Proc. of the 5th IEEE Intl Conference on Data Mining*, pp. 226-233, 2005.
- [11] E. Keogh, J. Lin, and W. Truppel. Clustering of Time Series Subsequences is Meaningless: Implications for Past and Future Research. In *Proc. of the 3rd IEEE International Conference on Data Mining*, pp. 115-122, 2003.
- [12] E. Keogh, S. Lonardi, and C. Ratanamahatana. Towards Parameter-Free Data Mining. In *Proc. of the 10th ACM SIGKDD*, 2004.
- [13] E. Keogh, L. Wei, X. Xi, S. Lonardi, J. Shieh, and S. Sirowy. Intelligent icons: Integrating lite-weight data mining and visualization into GUI operating systems. *ICDM*, 2006.
- [14] N. Kumar, N. Lolla, E. Keogh, S. Lonardi, C.A. Ratanamahatana, and L. Wei. Time-series bitmaps: A practical visualization tool for working with large time series databases. In *Proc. of SIAM SDM '05*, 2005.
- [15] M. Li, Y. Liu, and L. Chen. Non-threshold based event detection for 3D environment monitoring in sensor networks. *ICDCS*, 2007.
- [16] J. Lin, E. Keogh, S. Lonardi, and B. Chiu. A symbolic representation of time series, with implications for streaming algorithms. *Proc. of the 8th SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, 2003.
- [17] D. Minnen, T. Starner, I. Essa, and C. Isbell. Discovering characteristic actions from on-body sensor data. *10th International Symposium on Wearable Computer*, 2006.
- [18] E. Pekalska, R. Duin, and P. Paclik. Prototype selection for dissimilarity-based classifiers. *Pattern Recognition*, Vol. 39, No.2, pages 189-208, 2006.
- [19] C. Stafford and G. Walker. Characterization and correlation of DC electrical penetration graph waveforms with feeding behavior of beet leafhopper. *Under submission*, 2008.
- [20] D. Vail and M. Veloso. Learning from accelerometer data on a legged robot. In *Proc. of the 5th IFAC/EURON Symposium on Intelligent Autonomous Vehicles*, 2004.
- [21] L. Wei, N. Kumar, V. Lolla, E. Keogh, S. Lonardi, and C. Ratanamahatana. Assumption-Free Anomaly Detection in Time Series. *Proc. of the 17th Intl Scientific and Statistical Database Management Conference*, pp. 237-240, 2005.
- [22] M. Zouboulakis and G. Roussos. Escalation: Complex event detection in wireless sensor networks. In *Proc. of 2nd European Conference on Smart Sensing and Context*, 2007.