DIMACS Technical Report 2003-43
December 2003

Mining Deviants in Time Series Data Streams

by

S. Muthukrishnan[1,2] Rahul Shah[3] Jeffrey Scott Vitter[4]

[1]Permanent member
[2]Dept. of Computer Science, Rutgers Universuty, New Brunswick NJ 08903 & AT&T Research, Florham Park NJ 07932. Email: `muthu@cs.rutgers.edu`. Supported by NSF EIA 0087022, NSF ITR 0220280 and NSF EIA 02–05116.
[3]Dept. of Computer Sciences, Purdue University, West Lafayette IN 47907. Email: `rahul@cs.purdue.edu`. Supported in part by the Army Research Office through grant DAAD19–01–1–0725 and by the National Science Foundation through grant CCR–9877133.
[4]School of Science, Purdue University, West Lafayette IN 47907. Email: `jsv@purdue.edu`. Supported in part by the Army Research Office through grant DAAD19–01–1–0725 and by the National Science Foundation through grant CCR–9877133.

ABSTRACT

One of the central tasks in managing, monitoring and mining data streams is that of identifying outliers. There is a long history of study of various outliers in statistics and databases, and a recent focus on mining outliers in data streams. Here, we adopt the notion of "deviants" from Jagadish et al [22] as outliers. Formally, deviants are defined based on a representation sparsity metric, i.e., deviants are values whose removal from the dataset leads to an improved compressed representation of the remaining items. Thus, deviants are not global maxima/minima, but rather these are appropriate local aberrations. Deviants are known to be of great mining value in time series databases.

We present first-known algorithms for identifying deviants on massive data streams. Our algorithms monitor streams using very small space (polylogarithmic in data size) and are able to quickly find deviants at any instant, as the data stream evolves over time. For all versions of this problem—uni- vs multivariate time series, optimal vs near-optimal vs heuristic solutions, offline vs streaming—our algorithms have the same framework of maintaining a hierarchical set of candidate deviants that are updated as the time series data gets progressively revealed.

We show experimentally using real network traffic data (SNMP aggregate time series) as well as synthetic data that our algorithm is not only remarkably accurate in determining the deviants, but also that evolution of deviants over time reveals interesting artifacts in data streams.

1 Introduction

A challenge in data mining is to develop methods for mining data streams. Data streams are ubiquitous: observations of atmospheric conditions (rainfall, temperature), network traffic (highway, telephone, internet, web click), financial transactions (point-of-sales, credit card transactions, stock tickets), etc. Traditional databases deal with data stored in finite, persistent data sets with moderate update rates. Streams are potentially infinite, continuous, arriving at a fast rate and in large amount. Mining data streams entails rather unique constraints: (1) individual data items have to be processed extremely fast to match the stream rate, (2) only limited storage is available for processing–or capturing and archiving–data at stream rate, (3) mining has to be adaptive to changing trends in the stream, and (4) mining often has to support decision systems in *near-real* time because a number of applications are derived from monitoring systems that feed alarms, operations and other regulatory activities that require real-time response. These challenges in data stream mining have now engaged the database community; see [33] for a survey and [34] for a topical selection of papers.

Clustering, classification and outlier analyses are some of the fundamental mining tasks. There has been a significant amount of work on clustering and classification in data streams. The focus of this paper is on *outlier analysis*. The notion of outliers is intuitive in general, but there are many formal approaches to defining and searching for them in Statistics, Databases and KDD communities (a selection of books is [31, 20, 3]). The simplest outliers are "heavy hitters", i.e., values that are large. Heavy hitters are relevant in network traffic applications and have been studied extensively in that context (for example [11, 29], though this is by no means exhaustive). In general, one seeks more sophisticated notions of outliers. A common approach is to define outliers as those that are *unexpected* based on some posited probability distribution satisfied by the data; a small selection of references of this genre include [28, 32, 33]. An interesting distance-based approach to outliers was originally proposed by Knorr and Ng [26]. There are other approaches as well, such as clustering-, classification-, depth-based, etc. Many of these methods describe a global criterion for outliers. An interesting tension, in the area of outlier analyses, is based on the local vs global aspect of outliers. There is a large body of work on finding outliers within local windows by using one of the outlier approaches above within windows (for example, see the density-based approach of [4]). Finally, a recent work studies local outliers, but at multiple granularities [27].[1]

In this paper, we adopt the notion of outliers introduced in Jagadish et al [22], namely, that of *deviants*. Deviants are outliers defined based on a representation sparsity metric. In particular, fix the histogram representation of the input data (the discussion may proceed equally in terms of any other representation mechanism for input data). A histogram comprises a few "buckets" that are used to summarize the data lossily. Say we have a budget of B words to store the histogram. Suppose we removed $k \leq B$ points out of the data and stored them separately, and used the rest of the budget of B words to lossily compress the remaining data distribution. Now, the problem is posed as an optimization one. What is the choice of k points, so that the combined representation of k points and the remaining histogram representation best approximates the input data distribution? (The formal definition of this problem will be presented in Section 2.) The k points that are removed are called *deviants*. The authors in [22] showed that there is a choice of k for which one obtains the

[1] This paper also has a nice overview of outlier detection literature.

"best" overall representation, and that the deviant points can be used as a highly effective mining device for timeseries data. They also presented efficient algorithms for finding the deviants, taking time $O(n^2k^2B)$, on stored data of size n. Their methods do not work in the data stream case.

In this paper, we study the problem of estimating deviants on data streams. Intellectually, deviants are intriguing because they implicitly combine the local outlier property (deviants are points that "stick out" in the context of a local bucket) with the global context (the bucket boundaries are based on global criteria of sparse data representation). Technically, finding deviants appears to be challenging. In fact, the result in [22] relied on double dynamic programming, and it captures deviants over a large class of solutions. However, we observe here that the deviants found in [22] are not optimal. Rather, one can obtain a truly optimal set of deviants using a structural property of deviants within buckets (Lemma 1). Thus the problem of optimal choice of deviants is fairly sophisticated. Finally, deviants seem to apply quite naturally to data streams which are time series observations. In many applications that generate data streams–eg., IP network data–a few buckets suffice to capture most of the energy of the data distributions [11]. Hence, for relatively small parameters b and k, one can obtain a few meaningful deviants for data streams. This is quite attractive for data mining.

We initiate the study of deviant mining on data streams. Our main contributions are as follows.

- We first consider the univariate time series case where at each time t, we have a function $f(t)$ that is observed.

  We present the first known optimal algorithm for finding provably optimal deviants. More importantly, we present the first known provably near-optimal streaming algorithms for mining deviants in data streams. Our algorithm uses $O((k^3/\epsilon)\log n)$ space to store a summary data structure of the stream taking $O((k^4/\epsilon)\log n)$ time to process each item. When queried, we can obtain deviants in time $O((k^3/\epsilon)\log n)$. Algorithms that work in less than linear space and time need to necessarily output an approximation rather than the exact answers for this problem. We prove that the approximation is $1+\epsilon$, for any user specified tolerance ϵ, on the error of approximating the datastream using the combined representation of deviants and the histogram on the remainder. We also present heuristics based on *pseudo deviants*, which do not have the provably approximate guarantee, but perform well nevertheless, and are significantly faster.

  Besides simple outliers such as heavy hitters [11, 19], we do not know of any outlier detection algorithm that works on data streams, i.e., in sublinear space and time.

- We next consider the multivariate time series case where at each time t, we have several functions $f_1(t), f_2(t), \ldots, f_\ell(t)$ that are observed. This is typically the case in practice, and ℓ is small. For example, in SNMP network traffic data, one typically measures the number of packets and the number of bytes on both incoming and outgoing directions in a communication link, per time interval t. Even in the offline case where all the data is stored, this problem is likely to be hard. We present heuristic solutions based on *relaxed deviants* for finding deviants in massive multivariate time series data streams. There are the first known algorithms for deviant analysis in multivariate time series.

- We perform systematic experiments with real data–SNMP data of aggregated traffic flow in backbone links of an ISP–and show the following. Our approximations are remarkably accurate in determining near-optimal deviants efficiently over large data streams. Our solutions can be used to continuously monitor the change in deviant patterns over time and can give a good measure of how deviant-like the data is. Also, the experiments show that our solutions runs much faster and storage used is less than what our worst case bounds predict.

All our solutions are derived using a uniform framework. We maintain a hierarchical set of potential bucket boundaries as we stream through the input. For each potential bucket, we maintain a set of *candidate* items (that may be deviants, pseudo deviants or relaxed deviants), that is suitable for the problem under consideration. The precise details differ: which items are suitable candidates, how to update them, and how to update the bucket boundaries, etc. This framework is not novel in algorithm design: many of the recent data stream algorithms use such a framework including those that find wavelets [12], quantiles [13], histograms [16], heavy hitters [7], etc. However it is novel to combine this framework with a set of candidate items per potential bucket at each level of the hierarchy. For the univariate case, the candidates we identify form a small superset of items in a bucket which are *guaranteed* to contain the deviants. (A similar characterization is inherent in [6] which may be thought of as a 1-bucket version of our Lemma 1.) A similar characterization does not exist for the multivariate case. For the sake of efficiency, some of our solutions use candidates that can be identified quickly which prove quite effective in experimental analysis, but do not a priori have guarantees.

As a side-effect of our study, we obtain a histogram based summary of stream data which, for a given space bound, is more accurate than the best known previous methods [16, 18]. This is because given a budget of B space, using some for deviants and rest for histogramming the remainder produces a compressed representation no worse than that of using the entire budget on histogramming the dataset without using any deviants. Therefore, our solutions are useful to get better data stream summaries for time series data streams. We do not explore this side-effect in depth in this paper, because it was explored in [22] in the offline case.

1.1 Map.

The rest of the paper is organized as follows. In Section 2 we formally present problems of our interest. In Section 3 we first outline some properties of the sum-square metric with deviants and associated data structures. Then, we provide a small space, fast per-item processing time approximation algorithm. We also prove its approximation guarantee and give space and time bounds. In Section 4 we describe a faster heuristic, based on what we call pseudo deviants. In Section 5 we extend this to the multivariate time series data streams. Section 6 provides some experimental observations over real and synthetic data sets and we conclude in section 7.

2 Models and Problem Formulations

Here, we give some definitions and provide an optimization framework for the problem of finding deviants in the data stream.

2.1 Histogramming.

First, we formally define the problem of finding histogram representation. While constructing the histogram, we break the data stream $v_1, v_2, ..., v_n$ into *buckets* of contiguous indices. Then, all the numbers in a given bucket B are replaced by a single value h. The histogramming error of this bucket is the *sum squared error* (SSE) given by $\sum_{v \in B}(v-h)^2$. Given a bucket, it is easy to see that the value h which minimizes this histogram error is just the average (mean) of all the values in B and the minimum possible error then in B is nothing but the variance of all the values in B. A simple expression for the variance $VAR[B]$ is $\sum_{v \in B} v^2 - (\sum_{v \in B} v)^2/|B|$. However, the non-trivial task here is to find out how to split the given sequence into buckets.

Problem 1 Optimal Histogram Construction.
Given a sequence X of length n and number of buckets k, find a partition of X into k contiguous buckets such that the total error of the histogram representation that is $\sum_{i=1}^{k} VAR[B_i]$ is minimum over all such partitions.

Jagadish et al [23] gives an optimal $O(kn^2)$ dynamic programming algorithm for the above problem. When this problem is seen as a streaming data problem, this algorithm takes $O(kn)$ space and $O(kn)$ per-item processing time.

In the data stream context, one provably can not find the optimal histogram using only sublinear space. So, one seeks an *approximate histogram* that has k buckets, but SSE is at most $1+\epsilon$ of the SSE of the optimal histogram.

Guha et al [16] show that when we only want to find a histogram representation which is approximate up to the factor $1+\epsilon$ of the optimal, this can be achieved by a streaming algorithm which takes $O(k^2 \log n/\epsilon)$ space and the same per-item processing time. [17] gives an even faster algorithm using wavelets. Typically, in these applications, n is very large and k is a small constant. We can only afford per-item-time and space sublinear in n, preferably logarithmic in n.

2.2 Deviant Histogramming.

We next describe the deviant histogramming problem. [22] defines this problem. Let $E(X, k)$ denote the minimum histogram error (SSE) for sequence X when we are allowed to use k buckets. Consider omitting a subset D from the sequence X. Now, if $E(X - D, k - |D|)$ is less than $E(X, k)$, then rather than storing a k bucket histogram representation of X we would prefer to store these values in set D separately and then store a $k - |D|$ bucket histogram of $X - D$. This representation would use exactly the same space but will give lesser error.[2] Any such set D, such that $E(X - D, k - |D|) < E(X, k)$ is then called a set of deviants for a particular value of k. D is said to be an *optimal deviant set* if it admits the minimum error over all such possible deviant sets. The problem then is to find such a set and create corresponding histogram:

Problem 2 Optimal Deviant Histogram (ODH). *Given a sequence X, and number of resources k, find the optimal deviant set D and $k - |D|$ bucket histogram on the remaining elements such that $E(X - D, k - |D|)$ is minimized.*

Jagadish et al [22] give an $O(n^2 k^3)$ algorithm for the above problem. Their solution uses double dynamic programming. However, it is not guaranteed to produce the optimal solution to this problem since the problem does not have the optimal subproblem structure [8]. Their algorithm work incrementally over a bucket, and assumes that if the new value in the bucket is not a deviant then the previously selected deviants do not change. However, this is false. Consider, for example that initially the values in a block are $50, 50, 50, 50, 10, 10, 10, 11$. Then the best 4 deviants are the last four numbers. Let us add 20 to this block. Now even though 20 is not a deviant, the best 4 deviants are $50, 50, 50, 50$. Thus, double dynamic programming does not work optimally. We correct this algorithm to find the optimal deviants using structural properties of deviants and the data structures to maintain them.

In the data stream case, again, one can not get the optimal deviant histogram, so the goal is to produce one with SSE at most $1 + \epsilon$ times the error of the optimal deviant histogram.

We note that there is also a related formulation of this problem where the number of buckets and the number of deviants are *both* fixed. We shall call this problem *kDH.*

Problem 3 k Deviant Histogram (kDH). *Given a sequence X, and number of buckets b and number of deviants k, find the optimal deviant set D of size k and a b-bucket histogram on the remaining elements such that $E(X - D, b)$ is minimized over all such possible deviants sets such that $|D| = k$.*

In kDH we specify the exact number of deviants we want to extract; in contrast, in ODH problem, it is chosen by the algorithm to achieve the minimum possible error. By using kDH we do not necessarily achieve better error than basic histogramming with $k + b$ buckets and we never achieve better error than ODH with $k + b$ resources. An $O(n^2 b k^2)$ algorithm is given by [22] for this problem.

2.3 Multivariate Deviants.

We now generalize the definitions above to the multivariate case. Now, the data stream $\vec{v}_1, \ldots, \vec{v}_n$ is such that each $\vec{v}_i$ is a d-dimensional vector. All the vectors in a bucket B are replaced by a vector $\vec{h}$. The histogramming error of this bucket is $\sum_{\vec{v} \in B} ||\vec{v} - \vec{h}||_2^2$ where $||..||_2$ is the L_2 norm of the vector. With this notion of error, the definitions above of optimal as well as approximation (deviant) histogram problems are generalized in a straightforward way for the multivariate case.

[2] More precisely, it takes $2\ell - 1$ words to store a ℓ bucket histogram and 2ℓ words to store ℓ deviants. In order to avoid the annoying factor of 2 throughout, we have defined problems without paying attention to the representation of deviants and histograms, and counting merely the number of deviants and buckets.

Histogramming or finding deviants has not been previously generalized in such a way to multivariate time series case, so our formulation is novel. Typically, d is small since most measurement and monitoring applications that generate time series data streams measure a small set of attributes over a long period of time.

3 Univariate Deviant Histogram Problem

We will first identify a structural property of univariate deviants. This will motivate the candidate data structures which we will describe next. Subsequent to that we describe the optimal deviant histogram algorithm. We then introduce the hierarchical bucketing framework incorporating the candidate deviants, and derive a provably approximate algorithm for the univariate deviant histogram problem for the data stream case.

3.1 Structural Property of Deviants.

Let $v_1, v_2, ..., v_n$ be the data stream. Let B_{ij} be a block (or a set) consisting of data points $\{v_i, v_{i+1}, ..., v_j\}$. Let $VAR[B_{ij}]$ denote the variance of set B_{ij}. Whenever convenient we shall also denote this by $VAR[i,j]$. Let $S_i = \sum_{j=1}^{i} v_j$ and let $S_i^{'} = \sum_{j=1}^{i} v_j^2$. $S_0 = S_0^{'} = 0$ Then,

$$VAR[i,j] = (S_j^{'} - S_{i-1}^{'}) - \frac{(S_j - S_{i-1})^2}{j-i+1}$$

Now let us assume we are allowed to omit k values from B_{ij} so that the remaining numbers in B_{ij} have least possible variance. We call it *k-variance* of the block and denote it by $VAR_k[i,j]$. Then,

$$VAR_k[i,j] = \min_{T \subseteq B_{ij}, |T|=k} VAR[B_{ij} - T]$$

Let $S_T = \sum_{v \in T} v$ and $S_T^{'} = \sum_{v \in T} v^2$. Then, $VAR_k[i,j] = \min_{T \subseteq B_{ij}, |T|=k} (S_j^{'} - S_{i-1}^{'} - S_T^{'}) - (S_j - S_{i-1} - S_T)^2/(j-i-k+1)$. And such a $T = T_{ij}^k$ which achieves the minimum k-variance is the optimal set of k deviants.

Let H_{ij}^k be the sequence of k highest values $h_{ij}^1 \geq h_{ij}^2 \geq .. \geq h_{ij}^k$ of B_{ij}. Similarly, let L_{ij}^k be the sequence of k lowest values $l_{ij}^1 \leq l_{ij}^2 \leq .. \leq l_{ij}^k$ of B_{ij}. Let T_{ij}^k be the optimal set of k-deviants in B_{ij}. Then,

Lemma 1 *(Candidacy Lemma) $T_{ij}^k \subseteq H_{ij}^k \cup L_{ij}^k$. More specifically,*

$$VAR_k[i,j] = \min_{0 \leq l \leq k} VAR[B_{ij} - (H_{ij}^l \cup L_{ij}^{k-l})]$$

That is, the optimal set of k deviants always consists of the l highest and remaining $k-l$ lowest values, for some $l \leq k$.

Proof : Let $\bar{x}$ be the average of numbers in $B_{ij} - T_{ij}^k$. Then, for any element $t \in T_{ij}^k$ and for any element $b \in B_{ij}$, $|t - \bar{x}| \geq |b - \bar{x}|$. If not so, we can replace t by b in T_{ij}^k to form a new set T'. Then,

$$VAR[B_{ij} - T'] \leq \sum_{x \in B_{ij} - T} (x - \bar{x})^2 < VAR[B_{ij} - T_{ij}^k]$$

This contradicts the optimality of T_{ij}^k. Now, $|t - \bar{x}| \geq |b - \bar{x}|$ implies that all the elements $t \in T_{ij}^k$ are the k values furthest from some point $\bar{x}$. This simply means that $T_{ij}^k = H_{ij}^l \cup L_{ij}^{k-l}$ for some value of $l \in \{0, 1, ..., k\}$. ■

A similar lemma is in [6] for outlier analysis within a single bucket. As we will see, we will use this lemma in an intricate way, maintaining such candidates in each of hierarchically structured buckets over data streams.

Using this notion of k-variance, problems ODH and kDH can be reformulated. For example, the formulation of kDH would be: Find the partition P of the index set $\{1, 2, ..., n\}$ into b buckets $p_1, p_2, .., p_b$ such that $\sum_{i=1}^{b} VAR_{k_i}[p_i]$ is minimized subject to the constraint that $\sum_{i=1}^{b} k_i = k$.

3.2 Candidate Data Structures for Deviants.

Here, we describe our data structure to find k-variance of a given block B_{ij} and to find the corresponding deviants. We call it the *k-variance data structure* and denote it by k-$VDS[i, j]$. This data structure takes $O(k)$ space and answers each variance query in $O(k)$ time. As the size of the block grows, the data structure can be updated to correspond to the larger block in $O(k) time$.[3]

The data structure k-$VDS[i, j]$ consists of the following:

1. H_{ij}^k sorted in decreasing order.

2. L_{ij}^k sorted in increasing order.

3. S_{ij} and $S_{ij}^{'}$.

H_{ij} and L_{ij}, according to candidacy lemma 1 form a set of candidates from which k (or less than k) deviants can be chosen. This forms an important part of the streaming solution because we have to visit the time series in only one pass and can maintain only limited (sublinear) storage, and cannot remember every point visited for its possible selection as a deviant.

For singleton block $B[i, i]$, it is easy to construct k-$VDS[i, i]$. H_{ii}^k and L_{ii}^k are both singleton lists consisting of element v_i. $S_{ii} = v_i$ and $S_{ii}^{'} = v_i^2$. We can quickly construct k-$VDS[i, j + 1]$ from k-$VDS[i, j]$ as follows:

1. Insert v_{j+1} into H_{ij}^k to form $H_{i,j+1}^k$. $H_{i,j+1}^k$ may or may not contain v_{j+1}. In case it does, the lowest element h_{ij}^k is removed and does not appear in $H_{i,j+1}$. This takes $O(k)$ time.

2. Likewise, insert v_{j+1} into L_{ij}^k.

[3] Actually the updates can be done in $O(\log k)$ time but that does not improve our overall bounds.

3. $S_{i,j+1} = S_{ij} + v_{j+1}$ and $S'_{i,j+1} = S'_{ij} + v^2_{j+1}$.

Once we maintain this data structure k-$VDS[i,j]$, we can easily find $VAR_p[i,j]$ along with the list of corresponding deviants for any $p \in \{0,..,k\}$ in $O(k)$ time. For this we try all possible sets of deviants D ($|D| = p$) such that $D = H^l_{ij} \cup L^{p-l}_{ij}$ and we choose the one which gives minimum variance for $B_{ij} - D$. The variance value corresponding to each set D of deviants can be computed in constant time from S_{ij}, S'_{ij}, S_D and S'_D. We can enumerate these possible sets of deviants as l goes from 0 to p. And we can also keep track of S_D and S'_D values in constant time (after $O(k)$ preprocessing) as we enumerate these sets.

3.3 Optimal Deviant Histogram.

We derive an $O(n^2k^2)$ dynamic programming algorithm for finding the optimal deviant histogram on n-sized time series. Here k is the total number of buckets plus deviants.

Let $DH[k,n]$ denote the minimum SSE for histogramming $v_1, ..., v_n$ with k being number of buckets plus deviants used for histogramming. Then,

$$\boxed{\begin{array}{l} DH[1,n] = VAR_0[1,n] \\ DH[p,1] = 0 \\ DH[p,n] = \min_{1\le x<n, 0\le i\le p-2}\{ \\ \qquad\qquad DH[p-i-1,x] + VAR_i[x+1,n]\} \end{array}}$$

To calculate $DH[k,n]$, we first calculate $VAR_p[i,j]$ for all values of $p(\le k), i, j(\le n)$. This makes it $O(kn^2)$ values. And each value can be calculated incrementally in $O(k)$ time. Hence this takes $O(k^2n^2)$ time. Also, the storage is $O(kn^2)$ because for each block B_{ij} we store H_{ij} and L_{ij} which amounts to $O(k)$ values per block. Given this, we calculate the table of $DH[p,m]$ which makes $O(kn)$ values. Computing each value takes $O(kn)$ time. Hence, totally, we spend $O(k^2n^2)$ time and $O(kn^2)$ space. Similar time and space bounds were claimed by [22], but as discussed earlier, their double dynamic programming solution does not give optimal deviants.

3.4 Approximate Deviant Histogram on Data Streams.

Consider what happens when we apply the abovementioned algorithm to the streaming setting. When item v_i is finished processing, we have computed and stored $DH[p,x]$ for all values of $p \in \{0,..,k\}$ and for all values of $x \in \{1,..,i\}$. We have also stored k-variance data structures for all blocks $B[j,i]$ for all values of $j \in \{0,..,i\}$. So, the total storage is $O(ik)$ thus far. On seeing the next value v_{i+1}, we update the k-variance data structures for blocks $B[j,i]$ to $B[j,i+1]$ for all $j \in \{0,..,i\}$ and we add k-variance data structure for the singleton block $B[i+1,i+1]$. This takes $O(ik)$ time in all. Then we compute $DH[p,i+1]$ for all values of $p \in \{0,..,k\}$. This takes $O(ik^qy2)$ time because we scan $O(ik)$ different combinations and generating VAR_i in each case takes $O(k)$ time. Thus we can conclude that this dynamic programming algorithm transforms directly into streaming algorithm with $O(nk)$ storage and $O(nk^2)$ processing time per item.

In the data stream case, we would like an algorithm that works in $O(poly(k)poly(\log n))$ space. The difficulty in the algorithm presented earlier was that x may take n values. x would take on fewer values if we settle for $1+\epsilon$ approximation. So the idea is to maintain $DH[k,x]$ only for logarithmically many interspersed values of x. This is achievable because when we only go for $1+\epsilon$ approximation, we need to store $DH[k,x]$ value only when it is more by certain factor than that for previous value of x. Note that $DH[k,x]$ is a monotonically increasing function of x and the maximum value it can achieve is nR^2 where R is the largest number in the series. So if we maintain a constant factor $1+\delta$ between each successive $DH[k,x]$ values stored, then we have to store at most $O((\log n + \log R)/\log(1+\delta))$ values of x.

3.4.1 The Algorithm.

Instead of $DH[p,x]$ as in dynamic programming algorithm, we maintain $ADH[p,x]$ (approximate deviant histogram) here. We will show that $ADH[p,x]$ closely approximates $DH[p,x]$.

Let $ADH[p,x]$ denote the cost of our solution where p is the number of buckets plus deviants in $v_1,\dots,v_x$. Our algorithm will inspect the values indexed in increasing order. Let the current index being considered be x. The parameter δ will be fixed later. For every $0 \le p \le k$, the algorithm will maintain $ADH[p,y]$ for values $1 = y_1^p < y_2^p < y_3^p < \dots < y_l^p = x-1$ of y such that $ADH[p,y_i^p] \le (1+\delta)ADH[p,y_{i-1}^p+1]$ for all $i \le l$ and $ADH[p,y_i^p] > (1+\delta)ADH[p,y_{i-1}^p]$ for all $i < l$. Associated with each $ADH[p,y]$ value we also store the first block (block containing y) along with the list of deviants in this first block in the solution of $ADH[p,y]$. This is maintained to reconstruct the histogram and get the deviants when we complete our pass on the entire data set. For each such value of y, we have also maintained k-$VDS[y+1,x-1]$. Thus the total storage is $O(k^2 l)$. Additionally, the algorithm also maintains a benchmark value $b = ADH[p,y_l^{p-1}+1]$ ($y_l^{p-1}+1$ can be the same as y_l^p).

On seeing the xth value v_x, the algorithm does the following:

1. It updates k-$VDS[y+1,x-1]$ to k-$VDS[y+1,x]$. There are at most kl values of y. So this can be done in $O(k^2 l)$ time.

2. Then it computes $ADH[p,x]$ for each value of p. For this it scans $ADH[q,y]$ for each value $q < p$ and each value of y in $y_1^q, y_2^q, .., y_l^q$. It checks the quantity $ADH[q,y] + VAR_{p-q-1}[y+1,x]$ and selects the combination which minimizes this sum. Along with the $ADH[p,x]$ value, it also stores the $VAR_{p-q-1}[y+1,x]$ term which minimized it and the list of corresponding deviants in the block $[y+1,x]$. Each VAR computation takes $O(k)$ time and we try $O(kl)$ different values. So this takes $O(k^2 l)$ time for each value of p and hence totally, $O(k^3 l)$.

3. Once $ADH[p,x]$ values are computed, the algorithm decides, for each of them, whether to discard the previous value $ADH[p,x-1]$ and replace it by $ADH[p,x]$ or to just add $ADH[p,x]$ to the collection increasing l by 1. It replaces the previous value if $ADH[p,x] \le (1+\delta)b$. Otherwise, (i.e. $ADH[p,x] > (1+\delta)b$) it increments l, adds $ADH[p,x]$ to the collection and sets new benchmark $b = ADH[p,x]$.

4. Finally, the algorithm computes $ADH[k, n]$. Then it outputs the boundaries of the first bucket $[n, y+1]$ and its list of q deviants, and then repeats this recursively on $ADH[k-q-1, y]$ to get the entire list of deviants and buckets.

3.4.2 Correctness.

Theorem 1 *The values generated by our algorithm approximate the values for the optimal dynamic programming algorithm. The following inequality gives the approximation bound:*

$$ADH[p, x] \leq (1+\delta)^{p-1} DH[p, x]$$

Proof : We first note that $DH[k, x] \leq DH[k, y]$ whenever $x \leq y$. Also $VAR_k[y, z] \leq VAR_k[x, z]$ whenever $x \leq y$. Now, we shall prove the theorem by induction in p. Base case, $p = 1$, is trivially true because $ADH[1, x] = DH[1, x] = VAR_0[1, x]$. Now let's assume $ADH[q, x] \leq (1+\delta)^{q-1} DH[q, x]$ for all $q < p$. Then, let's say $DH[p, x] = DH[q, z] + VAR_{p-q-1}[z+1, x]$ for some q, z. That is these q, z are the minimizing combination. Let t be the smallest index such that $y_t^q \geq z$. Then $z \geq y_{t-1}^q + 1$. So, $DH[q, z] \geq DH[q, y_{t-1}^q + 1]$. By induction hypothesis $ADH[q, y_{t-1}^q + 1] \leq (1+\delta)^{q-1} DH[q, y_{t-1}^q + 1]$. Also, by our construction, $ADH[q, y_t^q] \leq (1+\delta) ADH[q, y_{t-1}^q + 1]$. Combining all these, we get

$$ADH[q, y_t^q] \leq (1+\delta)^q DH[q, z]$$

Since our algorithm, while computing $ADH[p, x]$, checks $ADH[q, y_t^q]$, $ADH[p, x] \leq ADH[q, y_t^q] + VAR_{p-q-1}[y_t^q + 1, x]$. Also, $VAR_{p-q-1}[y_t^q + 1, x] \leq VAR_{p-q-1}[z+1, x]$. Hence, $ADH[p, x] \leq (1+\delta)^q DH[q, z] + VAR_{p-q-1}[z+1, x]$. Since $q < p$, this implies,

$$ADH[p, x] \leq (1+\delta)^{p-1} DH[p, x]$$

■

3.4.3 Analysis.

Since, we want $ADH[k, n] \leq (1+\epsilon) DH[k, n]$ we need to set δ such that $(1+\delta)^k \leq 1+\epsilon$. Thus setting $\log(1+\delta) = \epsilon/k$ is sufficient. The number l of values maintained is bounded above by $O(\log DH[k, n+1]/\log(1+\delta))$. $DH[k, n+1] \leq nR^2$ where R is the largest number in the series. Also, for our operations, we would assume the size of each number is $O(\log n)$ i.e., $\log R = O(\log n)$. Therefore, $l = O(k \log n/\epsilon)$. Combining this with running times in previous subsections we get the following theorem:

Theorem 2 *Our algorithm gives $(1+\epsilon)$ approximation to optimal deviant histogramming (ODH) problem. The space required for the algorithm is $O(k^3 \log n/\epsilon)$ and per item processing time is $O(k^4 \log n/\epsilon)$.*

At a high level, the algorithm we have described so far is an adaptation of the histogramming algorithm on the data stream developed by [16]. Alternatively, it may be thought of as a generalization of the outlier analysis in [6] to multiple buckets, and to the streaming context. Our algorithm relies on careful adaptation of [16], additionally using the data structures implied by the Candidacy Lemma in each partition that is maintained over time [6].

Our analysis above extends that in [16]; again, the extension requires the combinatorial properties of the deviants to show certain monotonicity holds as the time series unravels. In what follows, we will further extend this framework and derive a number of heuristic algorithms for other deviants.

3.5 Extension to kDH.

We can similarly apply the same technique to the dynamic programming algorithm used to solve kDH. Here, instead of $DH[k,n]$, we have to define k-$DH[k,b,n]$ which denotes the minimum square error in histogramming $v_1, v_2, ..v_n$ with b buckets and k deviants removed. A simple dynamic programming would store $O(nkb)$ values and each value can be computed in $O(nk^2)$ time as earlier. ($O(k)$ for variance calculation and nk possible values to check.) In our approximate streaming model, we replace n by $(b/\epsilon)\log n$ for storage. Recall that with each value we also store k-VDS. Hence, the total storage is $O((k^2b^2/\epsilon)\log n)$ and per-item running time is $O((k^3b^2/\epsilon)\log n)$.

Theorem 3 *We can compute $(1+\epsilon)$ approximation to kDH problem with b buckets and k deviants using $O((k^2b^2/\epsilon)\log n)$ space and $O((k^3b^2/\epsilon)\log n)$ per item processing time.*

4 Pseudo-Deviant Algorithm

In this section, we will show another algorithm for the univariate deviant histogram problem on data streams. This has the same hierarchical framework as the data stream algorithm above. Along the notion of deviants we shall define slightly different outliers called *pseudo-deviants* or *pdeviants* for short. As will be evident, they are faster to compute but have a slightly different notion of optimality than deviants. They follow the optimality of deviants in the sense of alignment of bucket boundaries but when it comes to choosing deviants within the bucket they perform differently. Hence, they may be suboptimal overall. However, the speed of computation is not the only main motivation for defining pdeviants. Pdeviants are also very attractive when we extend this idea to multivariate data streams. This will be discussed in the next section.

Pseudo Deviants. Let $AVE[i,j]$ denote the average (mean) of the block B_{ij}. Let $\hat{T}$ be the set of k points in B_{ij} which are the farthest from $AVE[i,j]$. i.e. set of k points with highest $|v_l - AVE[i,j]|$ value. Then *k-pseudovariance* (*k-pvariance* for short) of a block B_{ij} is defined as $PVAR_k[i,j] = VAR[B_{ij} - \hat{T}]$ and such a set $\hat{T}$ is called the set of *k-pseudodeviants* (*k-pdeviants* for short). Note that $PVAR_k[i,j] \geq VAR_k[i,j]$ by the minimality of $VAR_k[i,j]$. As pointed out by lemma 1 the difference between deviants T and pdeviants $\hat{T}$ is that the deviants T are the k farthest values from $AVE[B_{ij} - T]$ while pdeviants are k farthest values from $AVE[B_{ij}]$. As will be evident from analysis and experiments, pdeviants are not significantly different from deviants in most cases (aside from some pathological examples) and are much easier and faster to compute over streams.

Problem 4 Optimal Pdeviant Histogram (OPH). *Given a sequence $v_1, v_2, ..v_n$ and a number of resources k, find a partition P into b buckets $p_1, p_2, p_3, .., p_b$ along with pdeviants such that $\sum_{i=1}^{b} PVAR_{k_i}[p_i]$ is minimized subject to the constraint $b + \sum_{i=1}^{b} k_i = k$.*

k-pseudovariance vs k-variance. How much does k-pseudovariance differ from k-variance? In pathological examples pseudovariance can be arbitrarily higher than the variance, but for practical cases there is very little difference between the two. In fact, the experiments on real network data, in the next section, shows no difference at all. However, it is easy to come up with an example where this difference is significant. Consider the example where we have a block of 8 elements and $k = 4$. Let the elements be $\{20, 10, 21, 9, 21, 9, 20, 9\}$. Then pseudodeviants are $\{21, 9, 9, 9\}$ leaving 80.75 as the 4-pseudovariance while the deviants are $\{20, 21, 21, 20\}$ leaving 0.75 as the 4-variance. However, this is a pathological example, characterized by the fact that k is comparable to the bucket size and most values are clustered around the two extremes. The very essence of the streaming model is that k is much smaller then a typical bucket size. Typically $k = o(n^{1/2})$. In this case, the difference between mean of the bucket and the mean obtained by removing deviants is a very small quantity say Δ. Then if x is the minimum deviation distance of any deviant from the mean of buckets with deviants removed, all the pseudodeviants deviate at least $x - \Delta$ from the mean. This means the reduction in SSE by removing deviants is at most more by $2\Delta * 100/x$ percent than that caused by pseudodeviants.

Candidate Data Structure. For pdeviants, the k-pvariance data structure is almost the same as that for the deviants. We maintain H_{ij} and L_{ij} in sorted order. Each can be updated in $O(k)$ time. The main difference is that in $O(k)$ time totally, we can answer all of the k pvariance queries. Thus, each pvariance, on an average, can be answered in $O(1)$ time instead of $O(k)$. This is where pdeviants achieve the speed up by factor of a k.

Algorithm. The algorithms for pdeviants are almost exactly the same as the ones for deviants. The only change is in how the pseudovariance for a given block is computed. The k-pvariance data structure here maintains H_{ij}, L_{ij} as before and moreover it maintains a value μ which is the average of the values in the block. μ is merely $S_{ij}/(j - i + 1)$. Now to form the set of pseudodeviants P, it merely merges the sorted lists H_{ij} and L_{ij} and finds k pseudodeviants in the order of their distance from μ. This merge procedure can also give the pseudovariance values for each k in a single pass. Thus a k-PVDS data structure can be updated in $O(k)$ time and each pseudovariance query on the given structure can be answered in $O(1)$ time (given the preprocessing during merge).

This means the offline algorithm for k-pseudodeviants (similar to the one in section 3.3) runs in time $O(n^2k)$ because processing the ith point takes $O(ik)$ (instead of $O(ik^2)$ earlier). Similarly, for the streaming case, the updates can be done in $O(k^2l)$ time giving the following theorem.

Theorem 4 *There is an algorithm which gives $(1+\epsilon)$ approximation to the optimal pseudodeviant histogramming (OPH) problem. The space required for the algorithm is $O(k^3 \log n/\epsilon)$ and per item processing time is also $O(k^3 \log n/\epsilon)$.*

5 Multivariate Data Streams

In this section, we will consider histogramming and deviant-finding in multivariate data streams, which has not been studied before in the literature. First consider the offline version where all data is stored. Further, consider a simplified version of the problem where the number of buckets is 1 and we wish to remove k deviants. Given n vectors, which k of these should be removed so that the remaining vectors will have minimum variance? This problem is closely related to the problem in [10] of outlier removal in high dimensions, as well as to learning linear threshold functions in presence of noise [2]. Although we have not proved our problem to be NP-Hard, we suspect that it is very likely to be hard.

Offline pseudodeviants. In the case of pseudodeviants, the mean to be considered is the average of entire bucket without removal. Hence, pseudodeviants are easily computed over multivariate time series (MTS). We now outline the offline algorithm to compute the pseudodeviants on MTS. The algorithm is similar to the offline dynamic programming for univariate time series in section 3.3, with the exception that k-VDS contains all the points in the bucket, not just H_{ij} and L_{ij}. Hence, the update of the k-VDS takes $O(n)$ time in the worst case. Therefore, instead of being an $O(n^2k)$ algorithm, this becomes an $O(n^3)$ algorithm[4]. To compute the k-variance of a given block, we have to first obtain the mean value of the bucket and then amongst all the points in the bucket choose the k farthest ones to this mean value point. This is because even a small change in the mean value point of the bucket can result in completely new pseudodeviants and hence we cannot really limit the number of candidate points for pseudodeviants (or deviants) as in the case of univariate time series.

The absence of candidate based structure also means that we cannot actually calculate the pseudodeviants or approximate pseudodeviants in the streaming model.

Example. Consider an example of a time series in 2 dimensions for which we want to find k pseudodeviants in a bucket of $n+1$ points. Let the first n points visited consist of n/k groups of k points each at unit distance from the origin $(0,0)$. Each group consists of k copies of the same point. The ith group has points whose angle from vector $(0,1)$ is $2\pi ki/n$. Now, any of these n/k groups could form the group of k-pseudodeviants depending on the $n+1$st point.

Streams and Relaxed deviants. The mean value point of the bucket, in most cases is progressively less likely to change as the bucket size grows. Hence, a reasonable heuristic to guess the candidates based on current position of the mean value point is as follows. Choose a parameter c (this parameter will depend on dimensionality). Maintain the first ck elements in the bucket as candidates. Now at the entry of each new point in the bucket, find the new mean value point. If all the candidates maintained are farther from this mean value point than the new point then the list of candidates remains the same. Else, insert this new point in the list of candidates and remove the one which is the nearest to the mean value point. Now, out of these ck candidates, choose the k farthest from the mean value point

[4]We assume here that the dimensionality is a small constant. Otherwise we shall have a extra factor of d.

as pseudodeviants. We call these *relaxed deviants* or *rdeviants* for short. They may be far different from pseudodeviants in pathological cases, but they are nearly similar in practical data sets.

Given a bucket b of data points $\vec{v}_i, .., \vec{v}_j$ (also denoted B[i,j]), a point $\vec{v}_l$, $i \leq l \leq j$, is one of the k *relaxed deviants* if and only if

1. $\vec{v}_l$ is one of the ck farthest elements from $AVE(B[i,l])$ in amongst the elements in $B[i,l]$.

2. Among the all points in $B[i,j]$ which satisfy the previous criteria, $\vec{v}_l$ is one of the k farthest elements from $AVE(B[i,j])$

If R is the set of k relaxed deviants in $B[i,j]$, then the k-rvariance $RVAR_k[i,j] = VAR(B[i,j] - R)$. The problem Optimal Rdeviant Histogram (ORH) can be defined similar to previous deviant histogram problems above. Interestingly, note that if entire data stream were to be reversed, deviants and pdeviants stay the same while rdeviants can be different. This is because the definition of rdeviants depends on the directionality of the stream.

Candidate data structure. The candidate data structure for $B[i,j]$ consists of ck candidates sorted in decreasing order of their distance from $AVE(B[i,j])$. Call them $\vec{r}_1, \vec{r}_2, ..., \vec{r}_{ck}$. We also store the partial sums $\vec{S}_l = \sum_{m=1}^{l} \vec{r}_m$ and $S'_l = \sum_{m=1}^{l} ||\vec{r}_m||_2^2$. Note that S' are scalars while S are vectors. Then, variance calculation is the same as in previous sections.

Algorithm. The algorithm again only differs from the univariate case in how the k-rvariance data structure is updated. During each update we sort the candidates; assuming parameter c and dimensionality d are small constants, the updates can be done in $O(k \log k)$ time. And then with this preprocessing, each rvariance query can be answered in $O(1)$ time. Hence, the per-item processing time is $O(k^3 \log k \log n/\epsilon)$.

6 Experiments

We implemented the exact as well as the approximate versions of ODH and kDH problems. The experiments were done on a pentium PC (2.78GHz). We used these algorithms over synthetic as well as real (IP network) data. Based on a large number of experiments carried out, we present some representative and interesting observations.

Figure 1 is a synthetic data designed for an intuitive test of the scheme. Figure 2 shows some of the time series data we used for most of the experiments. The data set D1 is SNMP data (total number of megabytes sent per fixed time window) over a lightly loaded network link. D1_1000 is an expanded snapshot of first 1000 points of data set D1. While experiments have been done on many SNMP data sets including the one with heavy traffic link, we shall use D1 as our main representative. We report some interesting facts with heavy traffic links also in the subsections to follow. We shall use synthetically generated data for experiments with multivariate time series.

In general, for our experiments, we choose a parameter d which is equal to $1/\delta$. For the worst case approximation ratio as we discussed in the earlier section, $\log(1+\delta)$ has to be set

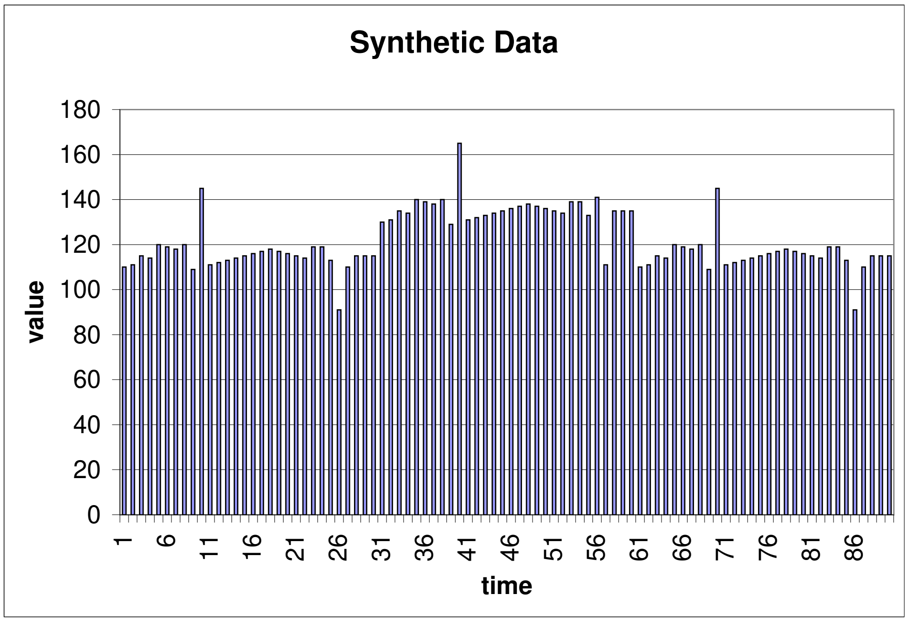


Figure 1: Demonstration data

to ϵ/k. i.e., $\delta = 2\epsilon/k$. This means that for approximating a solution within 1% error with 10 resources we need $d = 2000$. However, this is only in the worst case. We observe that, in most practical cases even $d = 10$ or $d = 50$ can achieve 1% error. As an added advantage, our algorithm stores fewer values and works quite fast. We discuss this further in one of the subsections which follow. Unless stated otherwise we use the streaming (approximate, ADH) algorithm for ODH. In places we shall also use optimal (dynamic programming, DH) algorithm or the streaming algorithm for kDH problem. But this will be explicitly mentioned.

6.1 Experimental example.

This data set (figure 1) consists of 90 points and is artificially designed for an intuitive test of the scheme. The sequence of the first 30 points is the same as sequence of the last 30 points. The middle 30 points are also the same, but shifted above by 20 units. Two deviants are purposefully placed in each subsequence of 30. Apart from these two deviants, the rest of the points are very close to each other. This is very simple input: even with d as small as 2, the algorithm achieves the optimal. Given 9 resources, the algorithm uses three buckets $[0, 28], [29, 58]$ and $[59, 89]$ and identifies the six deviants as $145, 91, 165, 111, 145, 91,$ with sum-square error (SSE) of 774, which is optimal. Given 3 or fewer resources, the algorithm uses only buckets for histogramming. Each additional resource more than 3 is used as

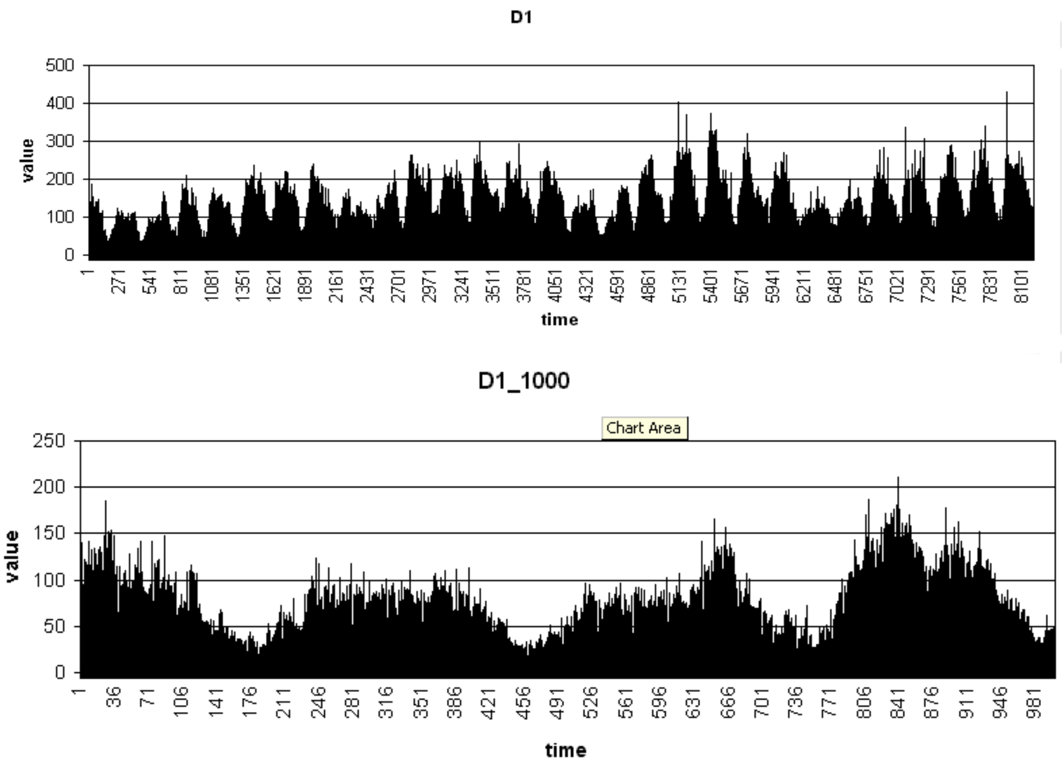


Figure 2: SNMP Data set

d	SSE	deviants with buckets
5	464617	{[999-945] : 88, 84}, {[944-792] : 210}, {[637-121] : 124, 118, 118, 113, 113}, {[0-120] : 185}
10	444679	{[999-939] : 88}, {[938-794] : 210, 187}, {[793-394] : 165, 156, 141, 138, 137}, {[393-234] : 83}, {[122-0] : 185}
20	435677	{[999-937] : 106}, {[936-788] : 210}, {[787-398] : 165, 156, 141, 138, 137, 136}, {[121-0] : 185}
30	431805	{[940-787] : 210}, {[786-401] : 165, 156, 141, 138, 137, 136, 135}, {[120-0] : 185}
OPT	428223	{[938-787] : 210}, {[786-399] : 165, 156, 141, 138, 137, 136, 135}, {[119-0] : 185}

Table 1: Deviants captured at different levels of approximations

deviants; for more than 9 resources, more buckets get added.

6.2 Convergence of approximation algorithm towards the optimal.

In most cases, even 1% error tolerance is enough for a suboptimal solution to not catch one or more of the significant deviants. For example, consider data set D1_1000 (Figure 2). With $k = 9$ and $b = 6$ the optimal error (SSE) is 428223, 1% of which is 4282. In the optimal solution, a point with value 185 in the first bucket is a deviant. However, even if it was not determined to be a deviant, the error in approximating the data would have been less than 4282. Thus the question in general is whether the approximation algorithm identifies the significant deviants given by the optimal algorithm. There is no definitive answer to this question, but all our experimental experience indicates that our approximation algorithm captures significant deviants identified by the optimal.

First we will see how the algorithm behaves as it converges towards the optimal. Table 1 shows the results for different values of precision parameter d. We list deviants along with the buckets in which they fall (we do not list the buckets where there are no deviants) and we give the SSE in each case. We identify the deviants by their values rather than their positions because it is easy to check them in the figure. The bucket gives the rough position of the deviants.

The trend is that as we approach the optimal solution, the bucket structure resembles the bucket structure of the optimal solution. However, this being an approximation, we store only the sparse values of breakpoints which can be used as bucket boundaries. Hence, the bucket boundaries are somewhat misaligned.

We observe that deviants also turn out to be the same. This is because once the bucket is defined, our algorithm picks the optimal deviant set for the particular bucket. The algorithm admits an approximation error only due to the boundary mis-alignment of bucket. Hence, as the bucket structure becomes closer to that of optimal, we get the almost the same deviants except for a few deviants which could be different if they were near some bucket boundaries.

We see this as an interesting artifact. When $d = 10$ the algorithm reports 88 as the deviant in bucket $[999 - 939]$, however when $d = 20$ due to slight change in the boundary 106 is caught as a deviant. Both of these are are not kept as deviants when we reach optimality. This is because there are more significant deviants caught elsewhere in other buckets as the bucket structure is refined.

Also, point 187 which was taken as a deviant in case $d = 10$, in general seems to be more significant then the other deviants caught by the optimal algorithm. However, according to our SSE metric it is not. The algorithm decides that it is a part of a narrower bucket $[938 - 787]$ and the overall average of this bucket is high.

Hence, it seems that if the error of our approximation is small, we get almost the same deviants. However, this is not completely true. When done over larger datasets there are cases sometimes where the bucket structure of the suboptimal is very different from the optimal (for that matter there could be two optimal solutions with radically different bucket structures, theoretically speaking). Then, although the clearly significant deviants are the same, we might have deviants which are different, because the deviants are bucket dependent. Although this does not guarantee the same solution, the deviants found by suboptimal algorithm are also interesting deviants in themselves.

Figure 3 shows convergence of SSE to optimal as d is increased for the data set consisting of first 500 points of D1.

6.3 Deviant trends over time.

In this subsection, we shall consider how the significant deviants emerge as the time series (data stream) progresses. We use data set D1 for this experiment. The deviants which were found to be significant earlier become less significant and often get replaced by altogether new sets of deviant as time progresses. We observe this trend in D1. It indicates that data behavior becomes more deviant-like as time progresses. We use $d = 100$ for all these experiments.

For $k = 6$ and $b = 8$, we get all the deviants in the final bucket $\{[8191 - 3758] : 432, 429, 402, 375, 369, 341\}$. This is quite intuitive looking at the figure. For the part of D1 restricted to its first 4096 values, with $b = 4$ and $k = 4$ it still catches all the deviants in the last bucket (top half). When we further restrict D1 to the first 2048 points, the deviants are spread-out $\{[1712 - 1398] : 53\}, \{[999 - 0] : 210, 187, 185\}$. This effect can be easily monitored by our streaming algorithm in the online fashion because as a part of dynamic programming scheme we compute near-optimal solutions for all values of k, b and n.

6.4 Emergence of deviants with varying number of resources.

When we experiment using the ODH model, the number of resources have to be split between buckets and deviants. These comparisons of the splitting of resources was done by [22]. We observe that it is almost always the case that initial resources are used as buckets for histogramming and then, as resources increase, the algorithm uses them as deviants. For D1 restricted to first 500 values, when the number of resources is 10 it catches only one deviant $\{185\}$. With 15 resources, it catches two $\{185; 147\}$. With 20 resources, it catches 7 deviants $\{185, 80; 147, 141, 141; 124, 44\}$ indicating that all the new resources are used as deviants. There is also a significant drop in SSE at this step from 67799 to 59338. This drop of more than 10% is all due to deviants. Also, the order in which deviants emerge with increasing number of resources indicates their significance or ranking. Our data structure can be simultaneously queried in real time with different k, b values. This can be used to establish the hierarchy of significant deviants at any point in time.

Also, we wish to note that we conducted other experiments, mainly on heavy traffic links where SNMP data is more bucket-like than deviant-like. In such cases, it is hard to capture deviants using ODH. All the resources are allocated for bucketing. However, instead of ODH if we use kDH forcing some deviants, the SSE goes up by 10-30%. This percentage rise in

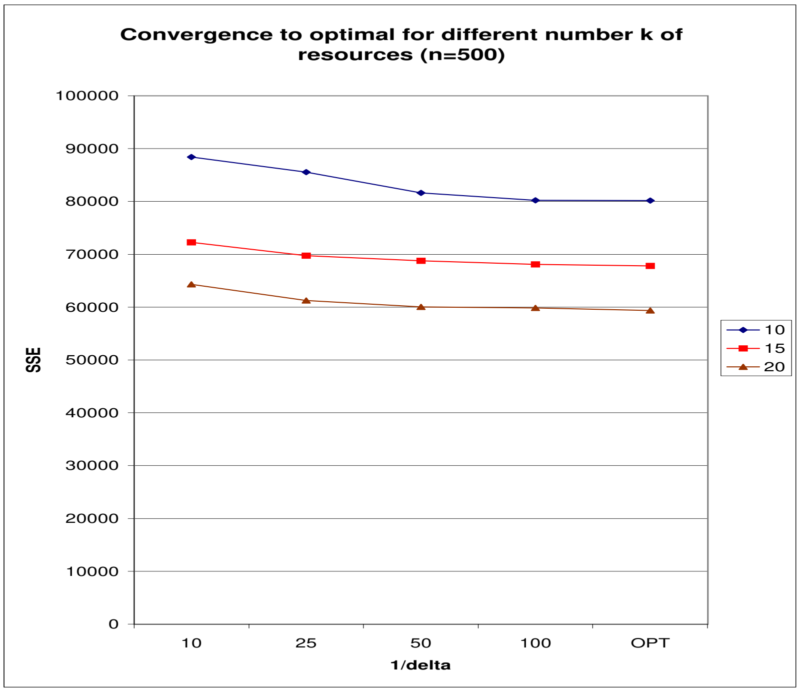


Figure 3: Convergence to optimal

SSE can indicate that data distribution in more prone to buckets than deviants and vice versa.

6.5 Timing analysis.

Here, we present some details on the running times of algorithms, for different parameter values. Figures 4 and 5 are two charts generated from the same table. These indicate much faster running times than the actual theoretical bounds. Theoretically, the running time of algorithm should scale linearly with d. However, we observe it is quite sublinear, and actually more like $\log d$. Hence, even though in the worst case our data storage is linear in d, we observe a much more sublinear trend indicating smaller storage and faster access. With k, our running times scale roughly around $k^{2.6}$, although our theoretical bound is $O(k^4)$. We get one k factor down by isolating d from k which for theoretical requirements is linear in k. The further improvement can be attributed to the sparsity in dynamic programming.

Figure 6 shows the same plot on both logarithmic and normal scales. Here for the purpose of comparision we also show the timings of the optimal dynamic programming (non-streaming) algorithm DH. Based on the theoretical analysis, we expect the figure's running times to scale as $n \log n$ for the approximation algorithm and to scale as n^2 for the

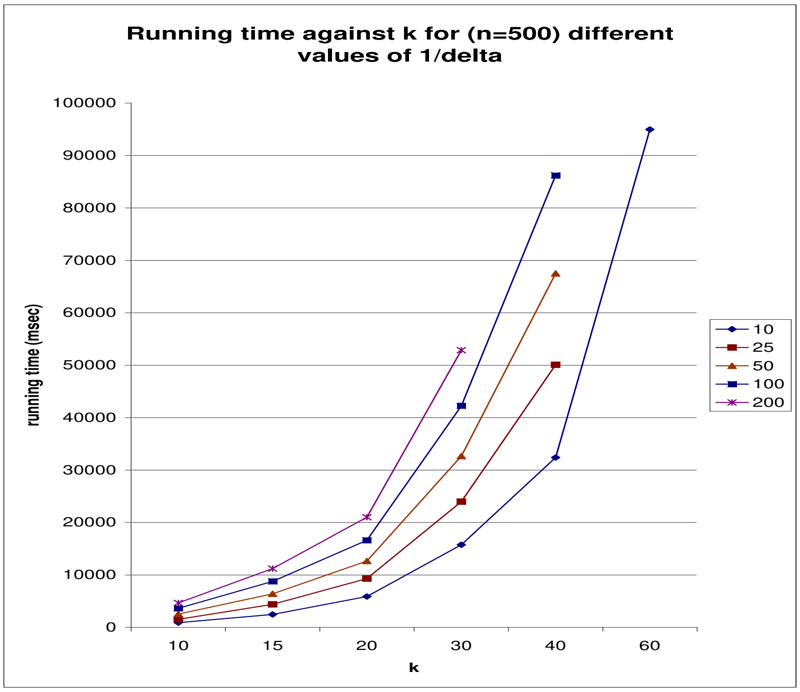


Figure 4: Time vs k

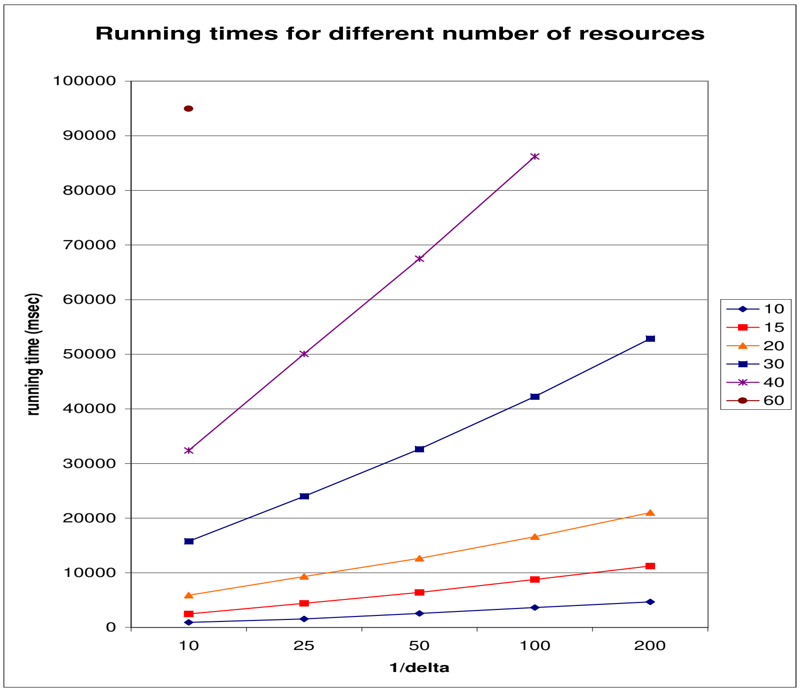
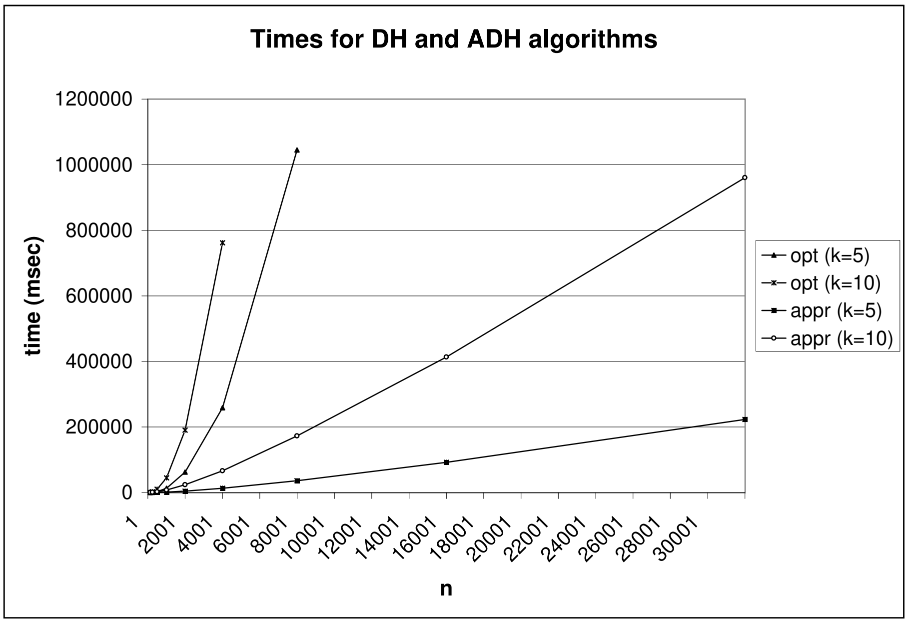


Figure 5: Time vs d



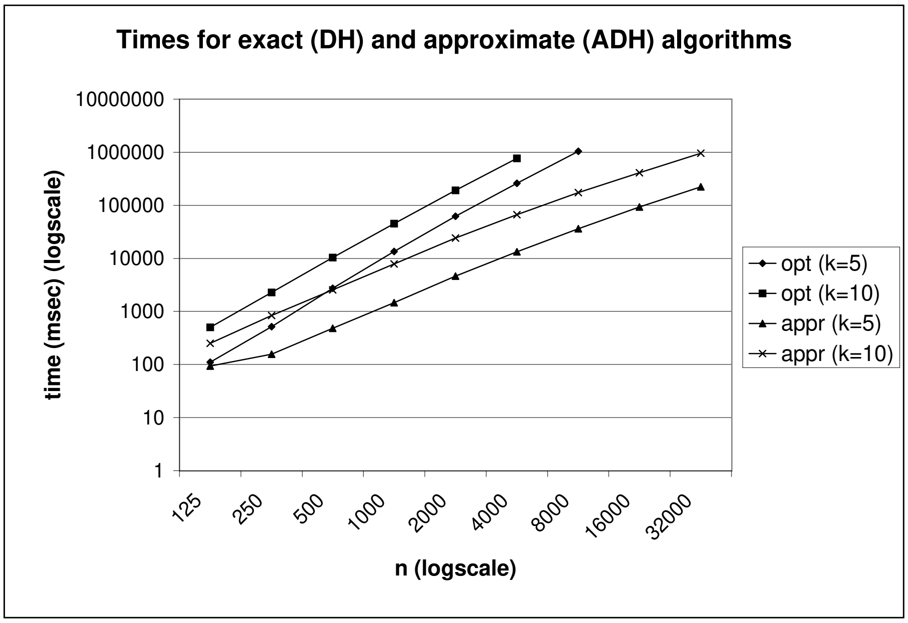


Figure 6: Time vs n

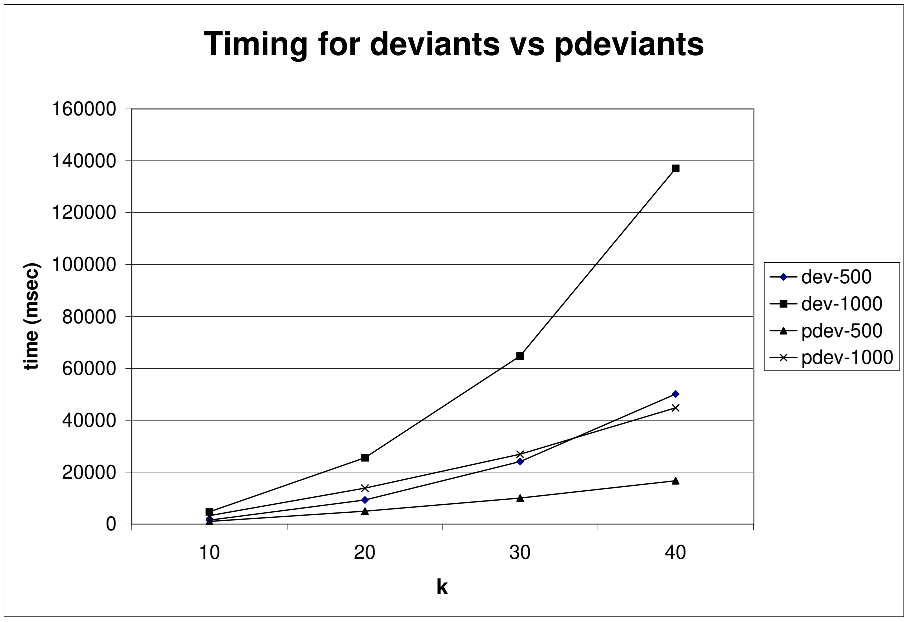


Figure 7: Comparative times for deviants and pdeviants

exact dynamic programming algorithm. The plot for approximate deviant algorithm looks almost linear with a very slight upward curvature for the $\log n$ factor. This plot was done for $d = 50$ and $k = 5, 10$. The plot on normal scale shows parabolic (quadratic) behavior for DH (exact) and almost linear behavior for ADH (approximate-streaming). The logscale plot shows the slope of ADH converges to 1 while the slope for DH remains 2.

6.6 Pseudodeviants vs Deviants.

We used data set D1 to do the comparative experiments for pseudodeviants and deviants. We found absolutely no difference in the deviants and pseudodeviants reported for various values of parameters. The comparative processing times for pdeviants are much lower than those for deviants. In figure 7 we take $n = 500, 1000$ and $d = 25$. In figure 7 we see that against k, the time complexity for pseudodeviants tends to grow as $k^{1.8}$ while the empirical growth proportionality for deviants seems to grow as $k^{2.6}$. Again, these should be k^3 and k^4 respectively, but we cut down a factor of k in both by fixing d. Thus, pseudodeviants are seen to serve as effective and fast substitutes for deviants.

6.7 Multivariate time series.

For the experiments with multivariate time series, we use synthetically generated 3 dimensional data, best modeled as blockwise gaussian with outliers. The mean value point and the deviation of data are chosen uniformly at random. Each coordinate of the mean value point is chosen from $[-50, 50]$ and deviation is chosen uniformly at random from $[0, 5]$. At each point there is 0.01 probability of changing these parameters to new ones. Also, at each point there is 0.01 probability that it is introduced as an outlier in which case this particular point can be chosen just like the mean value point above. For real data, we use SNMP data with incoming and outgoing traffic volume on communication links.

We did various experiments for offline as well as streaming cases. We found that rdeviants found by streaming algorithm are almost the same as pdeviants found by the offline algorithm. Furthermore, the significant deviants thus found are the same. When we increase the number of resources, sometimes we get one or two deviants to be different from the pdeviants but they are not the significant ones.

Here are the results from a representative experiment. We used a 500-point time series in 3 dimensions. We chose k as $4, 8, 12, 16$ respectively and fix $c = 1$. The SSEs obtained by rdeviants in each case were $142725, 122117, 104923, 100502$ and those by pdeviants were $142725, 122117, 104923, 99233$. The only difference appears when $k = 16$ and is due to the fact that rdeviants captures one less deviant than pdeviants. It fails to capture the point number 256 from the list of pdeviants which is the points at indices $\{256, 318, 370, 373, 463\}$. However, if SSE were to be taken as a measure of effectiveness of the deviants captured, rdeviants is close enough to pdeviants. Again, as the value of c is raised, the algorithm works slower but the SSE grows remarkably closer and even a small difference like previous is hard to find.

7 Concluding Remarks

Deviants are intriguing outliers, combining local and global features. We have presented first known, efficient algorithms for finding deviants—optimal, approximate, heuristic or univariate, multivariate, etc—on data streams. Pdeviants are motivated by their speed of computation in the univariate case and in multivariate case they are in fact the only known solution. Rdeviants are motivated by the fact that even pdeviants can not be computed effectively in the data stream model in the case of multivariate time series. We have seen by the experiments that pdeviants are remarkably close to deviants and in multivariate case, rdeviants are remarkably close to pdeviants. Hence, we think that rdeviants provide a good substitute for deviants for mining multivariate time series data streams. The problem of finding an near-optimal algorithm, i.e. one with provable approximation guarantees, for deviants in multivariate case still remains open. Outlier detection in data streams is an important problem, and we expect many different kinds of outlier analyses to be studied over time.

There is a related question of dynamic maintenance of deviants. Recently, it was shown that histograms themselves – quantiles, v-opt histograms– can be dynamically maintained [13, 14] in presence of arbitrary inserts and deletes. However, *such a result is impossible* for deviants: from our structural lemma 1, it follows that even for optimization with one bucket with one deviant, we need to maintain the maximum as well as minimum in presence of inserts and deletes. It is easy to prove that the maximum (or minimum) value can not be maintained, even to reasonable approximation, using only sublinear space in presence of deletes.

Our experimental results show that our algorithms are faster than our worst case bounds. Also, the approximate algorithms (with reasonable precision parameters) capture significant deviants of the optimal algorithms, as our experimental discussions show. Finally, if one monitors the SSE of the data stream representation using deviants, we believe it will be a good indicator for how deviant the data is at any time in the stream, and that may prove valuable as a mining tool.

References

[1] C. Aggarwal, J. Han, J. Wang, P. Yu. A Framework for clustering evolving data streams. VLDB 2003.

[2] A. Blum, A. Frieze, R. Kannan and S. Vempala. A polynomial time algorithm for learning noisy linear threshold functions. Algorithmica, 22(1) 1999, 35–52.

[3] V. Barnett and T. Lewis. Outliers in Statistical Data. John Wiley, 1994.

[4] M.M. Breunig, H.P. Kriegel, R.T. Ng, and J. Sander. Lof: Identifying density-based local outliers. In Proc. ACM SIGMOD Conf. 2000, pages 93-104, 2000.

[5] Y. Chen, G. Dong, J. Han, J. Pei, B. W. Wah, and J. Wang, Online Analytical Processing Stream Data: Is It Feasible? Proc. 2002 ACM-SIGMOD Int. Workshop on Research Issues on Data Mining and Knowledge Discovery (DMKD'02), Madison, WI, June 2002.

[6] S. Chaudhuri, M. Datar, R. Motwani, V, Narasayya. Overcoming Limitations of Sampling for Aggregation Queries. ICDE 2001.

[7] G. Cormode, F. Korn, S. Muthukrishnan and D. Srivastava. Hierarchical heavy hitters on data streams. VLDB 2003.

[8] T. Cormen, C. Leisersen and R. Rivest. Introduction to Algorithms.

[9] P. Domingos and G. Hulten. Mining high-speed data streams. In Proc. of the 2000 ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining, pages 71–80, August 2000.

[10] J. Dunagan and S. Vempala. Optimal outlier removal in high dimensional spaces. Proc. Symp. on Theory of Computing (STOC), 2001.

[11] C. Estan and G. Varghese. New Directions in Traffic Measurement and Accounting. SIGCOMM 2002.

[12] A. Gilbert, Y. Kotidis, S. Muthukrishnan and M. Strauss. Surfing wavelets on streams. VLDB 2001, 79–88.

[13] A. Gilbert, Y. Kotidis, S. Muthukrishnan and M. Strauss. How to Summarize the Universe: Dynamic Maintenance of Quantiles. VLDB 2002.

[14] A. Gilbert, S. Guha, P. Indyk, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Fast, small-space algorithms for approximate histogram maintenance. STOC 2002: 389-398.

[15] A. Gionis, H. Mannila. Finding recurrent sources in sequences. RECOMB 2003.

[16] S. Guha, N. Koudas and K. Shim. Data Streams and Histograms. STOC 2001.

[17] S. Guha, P. Indyk, S. Muthukrishnan and M. Strauss Histogramming Data Streams with Fast Per-Item Processing. ICALP 2002.

[18] S. Guha and N. Koudas Approximating a Data Stream for Querying and Estimation: Algorithms and Performance Evaluation. ICDE 2002.

[19] S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams. In Proc. of the 2000 Annual Symp. on Foundations of Computer Science, pages 359–366, November 2000.

[20] D.M. Hawkins. Identification of Outliers. Chapman and Hall, 1980.

[21] G. Hulten, L. Spencer, and P. Domingos. Mining time-changing data streams. In Proc. of the 2001 ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining, August 2001.

[22] H. V. Jagadish, N. Koudas and S. Muthukrishnan Mining Deviants in a Time Series Database. VLDB 1999.

[23] H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. C. Sevcik and T. Suel. Optimal Histograms with Quality Guarantees. VLDB 1998.

[24] E. Keogh, S. Lonardi, B. Chiu. Finding surprising patterns in a time series database in linear time and space. SIGKDD 2002.

[25] E. M. Knorr and R. T. Ng Algorithms for Mining Distance-Based Outliers in Large Datasets. VLDB 1998.

[26] E.M. Knorr, R.T. Ng, and V. Tucakov. Distance-based outliers: Algorithms and applications. VLDB Journal, 8:237–253, 2000.

[27] S. Papadimitriou, H. Kitawaga, P. Gibbons and C. Faloutsos. LOCI: Fast Outlier Detection Using the Local Correlation Integral.

[28] V. Puttagunta and K. Kalpakis Adaptive Methods for Activity Monitoring of Streaming Data.

[29] http://www.cs.columbia.edu/ids/concept/

[30] S. Ramaswamy, R. Rastogi, and K. Shim. Efficient algorithms for mining outliers from large data sets. In Proc. ACM SIGMOD 2000, pages 427–438, 2000.

[31] P.J. Rousseeuw and A.M. Leroy. Robust Regression and Outlier Detection. John Wiley and Sons, 1987.

[32] B. Yi, N. Sidiropoulos, T. Johnson, H. V. Jagadish, C. Faloutsos, and A. Biliris. Online data mining for coevolving time sequences. In Proc. of the 2000 Intl. Conf. on Data Engineering, pages 13–22, March 2000.

[33] Research problems in data mining. http://www-courses.cs.uiuc.edu/ cs497jh/ppt/topics01.ppt

[34] M. Zaki. Online, Interactive and Anytime Data Mining. SIGKDD Explorations, Vol2, Issue 2.