Evolution in Data Streams

Weiyun Huang, Edward Omiecinski, Leo Mark
College of Computing
Georgia Institute of Technology
Atlanta, GA 30318
{wyhuang,edardo,leomark}@cc.gatech.edu
Feb 28, 2003

Abstract

Conventional data mining deals with static data stored on disk, for example, using the current state of a data warehouse. In addition, the data may be read multiple times to accomplish the mining task. Recently, the data stream paradigm has become the focus of study, where data is continuously arriving as a sequence of elements and the data mining task has to be done in a single pass. An example is to construct a model(s) of the data as in clustering or classification in a single pass and with limited memory. Data arrives as one or multiple potentially infinite streams under the data stream model. Data streams can flow at variable rates and the underlying models often change with time. The current work in data stream mining does not focus on change (“evolution”) and that is precisely our main focus. Monitoring the changes in the models becomes as important as obtaining the models. Therefore, stream data mining not only needs to mine data incrementally and decrementally (in order to keep track of recent data), but also has to provide methods to monitor/detect the changes of underlying models. We consider this problem as “data evolution”. Of equal importance, the mining algorithms themselves need to be adaptive/dynamic when the flow rate of data streams change dramatically. That is, the algorithms should be able to downgrade accuracy in order to handle a data burst, or to do a more thorough analysis when data flow is slow. We consider this problem as “algorithm evolution”. We will study both data evolution and algorithm evolution. We will provide efficient algorithms to incrementally/decrementally mine stream data, good techniques to store data models and detect/monitor the changes, and a set of algorithms that can switch from “high resolution” to “low resolution” in order to adapt to the flow rate.

1 Introduction

A data stream is a sequence of data points which usually can only be read once and does not support random access. Generally the data points are time ordered, with some exceptions, for instance, data sent from distributed sources may arrive slightly out of order. In many applications there are multiple distributed data streams that need to be processed together to produce models/knowledge.

Because of the dynamic nature of the data stream(s), it is very important to monitor/detect the changes underlying the data, which we call “data evolution”. Monitoring evolution of clusters and outliers is part
of the research we propose here. In order to study evolution in the context of data streams we need to also consider clustering algorithms that work in this context.

Furthermore, because of the potential dramatic change of the flow rate, the algorithms that process the data need to be adaptive. That is, when the data flow rate is high, in order to process the data, the algorithms may need to temporarily sacrifice accuracy; and when flow rate is low, the algorithms can do elaborate analysis to more accurate models. This transition and its effect have not been studied.

1.1 Applications

Good stream mining algorithms are desirable for many real-life applications. Transaction data from all kinds of retailers form a data stream, or multiple data streams. The analysis of this kind of data can generalize user behaviors, extract user groups with different profit levels, produce sale patterns, and provide other information that retailers want to know. Mining a credit card usage data stream can help detect fraud. Network intrusion detection, satellite image processing, server load monitoring and scientific experiment monitoring, are all practical applications that can greatly benefit from stream data mining.

In our research we will apply our algorithms to real-life data. We are collaborating with the CNS(Computing and Networking Services) staff of the College of Computing, Georgia Institute of Technology. CNS is maintaining the computing and networking infrastructure in the College of Computing. Hardware failures, outside attacks and improper machine usage can bring major impact on the education and research activities in college. Therefore, it is very important to detect the problems in time and find the possible reasons.

CNS uses the SNORT [51] system to monitor the college-wide network traffic. SNORT is an open source network intrusion detection system. It can analyze network traffic and log packets in real time. It can be used as a sniffer, a packet logger, or a network intrusion detection system. Given an appropriate configuration it can produce ascii or binary information on various network activities and/or attacks, and even generates real-time alerts. This data can be very helpful for the diagnosis of network/computer problems and intrusion detection. However, in order to monitor all the interesting activities in our college network, SNORT needs to log a lot of information which is very difficult for humans to read and analyze. We intend to apply our stream mining and evolution detection algorithms to these real-life data sets to generate useful information for CNS. We expect that the domain knowledge CNS has about this data will help us validate our approach and algorithms.

1.2 Background – Data Stream Processing Model

A data stream is a sequence of data points \( \{X_1, X_2, \cdots, X_n, \cdots\} \), where \( n \) is the number of points seen so far. It is often the case that the total number of data points in a data stream is infinite. Each data point \( X_i \) is a \((m+1)\)-tuple \((v_1, v_2, \cdots, v_m, t)\), where \( m \) is the number of attributes and the \( t \) can be a sequence number that can uniquely identify a point, or a timestamp (may be valid time, transaction time [55] or both), or an “age” indicating the point’s location in a window [21]. Timestamps can be used to identify points, group data points, or explain the domain-specific results generated by mining algorithms. Usually, \( t \) is not used as a data attribute in the mining process, but it can be used to preprocess the data or query the mining results.

In a stream environment, we do not have the luxury to scan the data repeatedly in order to mine the underlying models. Given limited memory and limited time, we need to focus mainly on recent data, while keeping an overview of stale data.

In order to monitor the changes occurring in the data stream, we may want to focus on “recent data”, or rather, the data from some point of time in the past until the current time, as shown in figure 1. The model
has a “current window”, sliding on the data stream. The block of data falling in that window is the “recent data” we are interested in. Note that even this block of “recent data” may be too big to fit in the buffer in memory. As time goes by, the “current window” slides and each step it moves is called a “differential unit”. For example, in practice, the current window could be one week long and each differential unit could be a day. Now suppose every day we want to obtain the clustering result for the last 7 days. With a traditional clustering algorithm what we can do is to retrieve the last 7 days’ data every time and rerun the algorithm, which is obviously expensive. And since the sliding windows are overlapping, we have to process part of the data repeatedly. This model is also referred to as the “block evolution” [23] or “moving window” model [4].

Differential mining is a very difficult problem because the deletion of stale data may cause the loss of some important information. For example, if we are monitoring a data stream where each record contains an \( n \)-dimensional point, and we try to output the maximum pairwise distance in fixed-length time range (we can consider this as a “current window”, say 5 hours long) upon user request. That is, if the user queries at 1pm, then the maximum distance from 1pm to 6pm is returned; if the user queries at 1:01pm, then the maximum distance from 1:01pm to 6:01pm is returned, and so forth. If the mining is incremental only – we return the maximum distance that appeared so far – then we can simply store the maximum pairwise distance seen so far. However, when differential mining is required, things are more difficult. One or both endpoints which produced the stored pairwise distance may become stale and get deleted at time \( t \) so the pairwise distance we have recorded is meaningless after that. However, because before \( t \) this distance is the maximum, the real solution (which before time \( t \) is not the solution) may be discarded because it is not the maximum at that time. Therefore, in the differential case, only saving the maximum pairwise distance is not sufficient. More information has to be stored in the buffer. However, buffer space is often limited, therefore we may only be able to store partial information and estimate the answer based on that. How much information is needed to generate sufficiently accurate estimation, and what partial information shall we save, and how to generate estimation based on partial information, are all interesting and difficult problems to explore.

It is often the case that the data in a data stream comes from distributed data sources. These data sources may be far away from each other and their local clocks may not be synchronized. The transmission delay can make the data points appear out of order, which introduces additional complexity to the problem.

1.3 Goal of Our Research

The goal of our research is to explore the approaches to solve data stream mining problems, especially evolution in clustering and outliers. In brief, the work will cover four aspects.

1. The possibility of applying traditional data mining techniques in a stream environment. We want to explore the assumptions to be made in order to apply non-stream version data mining techniques in a
stream environment, and the factors that may affect the degree of accuracy. Also, in the case that a technique is not suitable for a stream environment, we want to find the fundamental barriers.

2. Clustering on stream data and the evolution of clusters. Data points can be clustered as they flow through the system and as time goes by. A sequence of sets of clusters can be formed. How to describe the evolution of the clusters and give users feedback is a big challenge and an important research issue. We want to develop methods to describe the evolution in real time (as the clusters are forming) so that users can monitor the data stream easily.

3. Outlier detection and the evolution of outliers. Outlier detection problems can be solved from different angles. Some algorithms are clustering based, some are classification based, and other methods are based on data distribution, and so forth. We will explore and develop the algorithms that do outlier detection in one pass of the data.

4. Flow rate adaptation. Current stream mining algorithms all use a large data set to simulate a stream. The algorithms use fixed parameters, or the parameters are only related to accuracy requirements. However, in practice, the rate of a data stream may dramatically change and we need adaptive algorithms to handle fast bursts of data. That problem can partially be solved by adding caching servers between data sources and mining algorithms, but it is also critical to consider algorithms to adjust the parameters and sacrifice some accuracy in order to process all the data in time.

The rest of this proposal is organized as follows. In section 2 we review the related work. In section 3 we present some initial results. Section 3 discusses the foundation of evolution study - a framework that can generate “states” from data stream. Section 4 and 5 talk about evolution of data and evolution of algorithms. Section 6 concludes this proposal.

2 Related Work

2.1 Data Streams

More and more attention has been paid to stream data processing [12, 14, 15, 40, 34, 28, 26, 10, 17]. The data stream environment brings many challenges from both the algorithm and system point of view. Babcock et al. [3, 42] addressed fundamental models and issues in processing and managing stream data and presented a data stream management system. Ganti et al. [23] discussed incremental model building and maintenance when data points were added into system by blocks. Carney et al. [17] proposed a new DBMS for storing and processing stream data. The work by these researchers identified some common issues shared by different data mining problems. Below we look at some fundamental problems in stream data mining. The related research on clustering and outlier detection is reviewed in separate subsections because our research will mainly focus on these two aspects.

Decision trees are useful tools to solve the classification problem. Hulten, Domingos et al [34, 15] present a stream-version decision tree learning system VFDT, which is based on Hoeffding trees. They also research the problem of classification of time-changing data: the modified VFDT algorithm can generate new subtrees when the new incoming data makes the current tree sufficiently inaccurate. This work is a good example of stream processing and change detection/react.

Besides conventional data mining models produced by classification and clustering, it is beneficial to maintain some other properties of stream data as well [39, 13, 21, 20, 19]. Querying over data streams has also been studied [24, 14, 22, 26]. Babcock et al. [5] proposed an algorithm to maintain the $k$ largest data values in distributed streams.
2.2 Clustering on Stream Data

Work has been done to cluster data incrementally [11, 16, 9], but since stream data usually can only be seen once in a fixed order, algorithms need to be not only incremental, but also able to finish processing in one pass. Bradley et al. [9] proposed a set of algorithms to cluster large amounts of data in one disk pass. These algorithms can significantly reduce the disk I/O. However in terms of time complexity, some of the compression schemes incur so much computational cost that the overhead cancels the savings made by less disk I/O. Farnstrom et al. [18] presented a simple algorithm which can cluster data more efficiently. These algorithms require only one pass over the data, which is ideal in the data stream environment.

Numerical data clustering has been extensively studied. For example, Guha et al. [30, 29, 43] presented constant-factor approximation algorithms for the k-Median problem within the data stream model.

Although in the real world most of the data sets contain categorical features, there is less related work on clustering categorical data than on numerical data. One reason is that there are a lot of classical data analysis algorithms that deal with numerical data very well. Also, it is easier to work in Euclidean space. Gilson et al. [27] proposed an approach based on dynamical systems to cluster categorical data. Guha et al. defined a new distance metric to categorical data and proposed the ROCK algorithm [31] based on that. However this distance metric requires heavy computation when dealing with high dimensional categorical data. Recently Barbara et al. [7, 6] presented an entropy-based algorithm that provided an alternative method for categorical clustering. Huang [33, 32] presented the k-modes algorithm, an extension to the well-known k-means algorithm in the categorical data domain. This algorithm can analyze high dimensional categorical data efficiently. Our data stream clustering algorithm proposed in section 3.5 is based on the k-modes algorithm.

Random sampling is an important method to analyze large amounts of data. In the data stream scenario we can also consider random sampling as a way to keep an approximation of the original data. However, different from the conventional case, we do not have a complete data set at any time since the data is streaming. Therefore we must resort to a reservoir type algorithm [25, 50], which provides an effective way to keep a sample of the stream data.

2.3 Outlier Detection

The definition of outlier varies in different domains. Usually, outliers are the ones that deviate from certain statistical models or expectations from previous experience [54]. There are many ways to describe a point’s deviation from the model. One approach [52] addresses this problem by looking at the $k^{th}$ nearest neighbor of the points. Other researchers also proposed their own outlier definitions [38, 48].

However, the difficulty in processing the data prompts us to explore different definitions of outlier. If in some domain, a definition of outlier could significantly reduce the computation overhead and can still capture important characteristics of the data, there is no point to stick to the “common” outlier definition. An example is the “sequential exception problem” defined by Arning et al [2]. The authors use the implicit redundancy in the data to detect deviations. Their main idea is that “after seeing a series of similar data, an element disturbing the series is considered an exception”.

Closely related to the research proposed in this proposal, we first presented the idea of logging a stream of events on a set of objects while at the same time incrementally maintaining the current state of the objects in the context of temporal databases [36]. The idea of querying and analyzing a logged stream of events was subsequently presented in [35]. An initial prototype of a high-performance Logger was described in [41].
Based on the definition of outlier, multiple outlier detection approaches have been studied.

Outlier detection can be done using all the dimensions of the data, or using projection to low-dimensional space [1]. Some solutions are from a machine learning point of view [53], and others use statistics based approaches such as depth [37]. However, statistics based approaches often suffer from the curse of dimensionality.

Another major category of outlier detection methods is clustering based. For example, the FindOut algorithm [54] removes the clusters from the original data and the points left are the outliers. It uses some signal-processing techniques (namely discrete wavelet transforms) instead of classic data mining algorithms.

As reviewed above, multiple aspects of research have been done on stream data, such as the stream project at Stanford University [42, 4, 5, 3, 13, 39], massive data streams research at Yale University [21, 20, 19], and stream query processing work at Cornell University [24, 14]. Our research is different from above in that we mainly focus on the evolution of clustering and outliers, as well as the evolution of algorithms. Therefore, the stream clustering and outlier detection algorithms we study are differential (both incrementally and decrementally, which will be elaborated later) and serve as the foundation of evolution study.

In related work, we developed an efficient clustering algorithm for handling large market basket data [47] and an efficient as well robust algorithm for clustering large data sets [46]. The robustness refers to the ability of our improved EM algorithm to handle high dimensional data, noisy data and the zero variance problem. We have also done work with regard to adaptive algorithms, which we refer to as algorithm evolution in this proposal, although in regard to the relational join operation being executed on a multiprocessor system [44]. With regard to incremental approaches, we developed an approach for mining association rules when data is continuously being added to the database without having to restart the mining procedure over but by building on prior results [45].

3 Stream Data Mining Framework

The foundation of data evolution detection and analysis is a framework to mine the models for stream data. Here the “models” include but are not limited to the clusters and the various statistics of distributions of data. Our research will mainly cover the evolution of data clusters and outliers, so in the sections below we address these two aspects.

A “differential” mining algorithm is needed to process live data streams. It should satisfy the following requirements:

1. It should process the data in one pass. If the data stream has to be preprocessed before running the clustering algorithm, the preprocessing/clustering tasks have to be done online (or within reasonably short delay). Making multiple passes over the data is not acceptable.

2. Mining should be done incrementally. In this way we can continuously add new data points into the existing clusters and adjust the clusters accordingly.

3. Mining should be done decrementally. We should be able to remove the effect of stale data from the existing models.

We use the term “differential” to refer to the incremental/decremental characteristics of the algorithm.
3.1 Framework for Stream Data Clustering and Outlier Detection

In order to detect the changes, states need to be obtained first. Therefore a differential mining framework is needed. Generally it will follow several steps:

1. Mine first batch of data.
2. Store the models (in memory, and/or in secondary storage).
3. Incrementally add new batch of data.
4. When necessary, delete stale data and reorganize stored models.

The details of these steps depend on the kind of application that is dealt with.

3.2 The Differential Clustering Algorithm

In order to cluster a data stream differentially, the stream clustering algorithm follows the steps below. Although a brand-new algorithm can be designed specifically, it is more convenient to make use of an off-the-shelf algorithm. Let us assume that the differential clustering algorithm makes use of a “regular” clustering algorithm C. As an example, in the experiment section we take the k-modes [33, 32] algorithm to cluster categorical data, and maintain a data buffer. The reason of using categorical data is that this type of data is more common in practice and relatively less research has been done about it. Obviously, differential clustering and outlier detection are not limited to categorical data.

1. Fill the buffer, or put all the available points in the buffer. The set of points in the buffer is D.
2. Choose k starting points from D.
3. Apply the clustering algorithm C to D and generate a set of clusters cls. Since D stays in buffer this process should be efficient.
4. Compress D based on cls, get a new representation of D, denote as D_{compression}. Depending on the compression scheme, part of or the entire buffer is freed.
5. If needed, remove the representation of stale data from D_{compression}.
6. Put new data points into the buffer for a new D (it may contain some points from a prior D). The modes of the clusters in cls are used as the starting points.
7. Go to step 3, until no new points are available.

This algorithm can produce clusters efficiently. Also, by periodically storing D_{compression} to secondary storage, we can easily go back to a previous time point and do further analysis. It also makes error recovery easier. If the clustering process is interrupted at some point, with the stored D_{compression} we could resume the process with a lower cost, rather than restarting from the beginning.

The generated model can be stored in a database, with timestamps. It enables later query of models based on time.
3.3 Differential Algorithm Used in Experiments

In section 3.5 we present some preliminary results using categorical data. Our work will not be restricted to this specific algorithm, and we only show this as an example to illustrate a possible approach.

One important aspect of categorical data clustering is to choose the dissimilarity measure, or the distance metric. Assume we are looking at a data set $\mathcal{D}$, with $n$ data points, $m$ dimensions. One metric proposed by Huang [32] is that the distance between two data points $\vec{x} = (x_1, x_2, \cdots, x_m)$ and $\vec{y} = (y_1, y_2, \cdots, y_m) \in \mathcal{D}$ is defined as the number of dimensions along which two points have different values:

$$\text{dist}(\vec{x}, \vec{y}) = \sum_{j=1}^{m} \delta(x_j, y_j)$$

(1)

where

$$\delta(x_j, y_j) = \begin{cases} 0 & (x_j = y_j) \\ 1 & (x_j \neq y_j) \end{cases}$$

This metric is simple and effective, meaningful in the unsupervised learning scenario since we do not have much knowledge about individual dimensions.

To an arbitrary point $\vec{x} = (x_1, x_2, \cdots, x_m) \in \mathcal{D}$, the cluster it belongs to at time $t$ is $cl(\vec{x}, t)$. If $t_f$ is the time we query for a clustering solution, we call $cl(\vec{x}, t_f)$ the “latest cluster” $\vec{x}$ belongs to. The mode of a cluster $cl$ is denoted as $\text{mode}(cl)$. The error of a clustering solution $cls$ is then defined as:

$$\text{error}(cls) = \frac{1}{n} * \sum_{\vec{x} \in \mathcal{D}} \text{dist}(\vec{x}, \text{mode}(cl(\vec{x}, t_f)))$$

(2)

For each dimension $d$ of the data, with respect to a data set $D$, a most frequent value $\text{mov}(d, D)$ is defined (from statistics’ point of view it is essentially the mode of dimension $d$. However here we use $\text{mov}(d, D)$ to distinguish from the multi-dimensional mode of a cluster). The data set $D$ can be the whole data set $\mathcal{D}$ or any subset of it.

$$\text{mov}(d, D) = \arg\max_v \text{freq}_d(v, D)$$

$$\text{freq}_d(v, D) = \frac{\text{count}_{\vec{x} \in D}(x_d = v)}{|D|}$$

Here $\text{count}_{\vec{x} \in D}(x_d = v)$ is the number of occurrences of value $v$ along dimension $d$ for the points in $D$.

A cluster’s “tightness” is decided by the frequency of most frequent value for each dimension and the distances from its members to its mode.

Given a collection of data points $\mathcal{D}$, the radius of a cluster $cl$ is the maximum distance between any point in the cluster and its mode.

$$\text{radius}(cl) = \max_{\vec{x} \in cl} \text{dist}(\vec{x}, \text{mode}(cl))$$

A cluster $cl$ is tight given threshold $(f, r)$ iff for each dimension $d$, $\text{freq}_d(\text{mov}(d, D)) \geq f$ and $\text{radius}(cl) \leq r$.

3.4 Data and Model Compression in Clustering

Numerical data can be compressed by keeping the means, standard deviations, and other “sufficient statistics” of data [9]. However, categorical data cannot be treated similarly since the result of comparing two values for any dimension can only be “the same” or “different”.
3.4.1 Compression for Categorical Data

Below we discuss the three compression schemes we used for compressing categorical data.

1. Naive compression Assume that all the points in the current clusters will not change membership later. Therefore, after the data in the buffer are grouped into clusters, we use the mode of each cluster to represent all the points in it, and use the number of points as the weight. In other words, we approximate each cluster by a set of identical data points taking the value of the mode.

2. Random sampling compression Assume that all the points in the current clusters will not change membership later. For each cluster \( c_i \), take a random sample of all the points in it and use \( \frac{|c_i|}{|\text{sample}|} \) as weight, where \( |c_i| \) is the number of points in cluster \( c_i \) and \( |\text{sample}| \) is the sample size. We use a modified version of the algorithm X proposed by Vitter [50] to draw the samples from stream data.

3. Dynamic tightness compression Before compression, the algorithm evaluates the “tightness” of each cluster, and this compression scheme only compresses the data in tight clusters. However, since the tightness of clusters depends on the characteristics of the data set, it is very possible that in most of the cases the cluster is not tight, therefore only a small portion of the data points will be compressed. In that case, the buffer would be quickly filled with stale data points, and no new points could be added. Therefore we dynamically change the “tightness” threshold with some heuristic rules: (1) when the buffer is about 90% full, we compress all the points just like naive compression does, and loosen the tightness threshold; (2) if the buffer is less than 30% full, we tighten the threshold.

We also tried another compression scheme, which involves doing secondary clustering of the points in the buffer, and compressing the tight secondary clusters. For the data set we used, the number of secondary clusters is about 4 or 5 times the number of clusters the queries desired. In terms of accuracy this scheme also works, but the computation overhead is significant and makes it even slower than regular k-modes. Therefore we did not use it in the experiments.

3.4.2 High-Biased Histograms

Depending on the size of the domain of a categorical attribute, the size of a histogram could be big. Then the compressed cluster image would occupy a lot of buffer space. Hence we have to trim the histograms to a certain length so as to limit the size of the auxiliary data structures. If a cluster is “tight”, then a small number of most frequently occurring values should be sufficient to represent the histogram fairly well (the other values, since they appear only a few times, would not make much difference later). During our experiment we found that trimming the histogram to length 5 (that is, only recording the 5 most frequently occurring values and their frequencies) provides reasonable accuracy.

3.4.3 Categorical Data and Numerical Data

Sometimes, a data set contains both categorical and numerical data and this makes it very hard to compute the distance between two data points. It is known that in some cases it is inappropriate to simply use the integer representation of categorical data as numerical values [33]. However, most of the data sets in reality contain both numerical and categorical data values. How to deal with the numerical values in the data sets is a domain-related problem.

Generally speaking there are two options: One approach is to design a hybrid distance between the two points. The distance between two points is a linear combination of the distance computed only with
categorical values and the distance computed only using numerical values [32]. The other option is to convert the numerical values by discretizing them according to some rules and to consider the converted values as categorical ones. The difficulty in implementing the first option is that it is usually hard to decide the weights assigned to the two parts of the distance calculation generated by numerical and categorical values. It will be even harder if we consider the numerical data normalization used in the preprocessing phase. In practice, the weight can be pre-assigned according to domain knowledge or according to test-runs on a sample of similar data.

3.5 Preliminary Clustering Experiments

Below some preliminary clustering experiments are reported. We use data sets residing on local disk to simulate streams.

3.5.1 Differential Clustering

To do differential clustering, we divide the data into many differential units, and set the size of one “current window” as five differential units. Every time the window slides, one oldest differential unit is removed and a new differential unit is added from the stream. Once we have the approximated clustering result for the last “current window” of data. We use the regular k-modes algorithm to cluster the corresponding portion of the data set and compare the accuracy.

In order to test the ability of the differential clustering algorithm to detect the evolution of the clusters, we create a “hybrid” synthetic data set. This data set contains 1 million points, and the number of attributes is 20. In this data set, the first half million points are generated from a set of Gaussian distributions, with means in the range of [0.0, 20.0]. And the second half million points are generated from another set of Gaussian distributions, with means in the range of [30.0, 50.0]. Then the numerical values are converted to the nearest integer values. Each half of the data set contains 10 clusters. We use 2000 points as the differential unit size and 10000 points as current window size. We run the differential algorithm (with three compression schemes) to this new synthetic data set and compare the clusters of the last current window with the ones obtained by the regular k-modes algorithm. To reduce the impact of initial clusters, we use 10 points randomly chosen from the first window as the starting points and run the algorithms 5 times then choose their best results for comparison. Figure 2 shows the results. The differential algorithm achieves good accuracy, which means that it can find the clusters from the last current window in spite of the old data which is totally different from the new data.

3.5.2 Incremental Clustering Results

We also compare our approach with the COOLCAT algorithm [7]. Since we do not have the actual implementation of the COOLCAT, we compare our result with the result presented in [7], using the congressional voting data (this data is small, containing 435 points and 16 attributes; the larger synthetic data sets used in their experiments were not readily accessible). The preliminary results are shown in table 1. The experimental setup, and the performance value of the COOLCAT algorithm is are copied from [7]. We used the naive compression scheme and 10kbytes buffer. The platform is a PentiumIII 550 MHz PC with 256 MB memory. The platform used by the COOLCAT is a Pentium III 800 MHz Dell server with 1GB memory.

In order to compare with the COOLCAT, we adopted the error measurements used in [7]. In brief,
Figure 2: Error comparison for differential clustering, hybrid data set. Clustering methods: 1:naive compression; 2:random sampling; 3:dynamic threshold; 4:regular k-modes

Table 1: Results on congressional voting data

<table>
<thead>
<tr>
<th>Alg</th>
<th>m</th>
<th>CU</th>
<th>External Entropy</th>
<th>running time (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>COOLCAT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0%</td>
<td></td>
<td>2.9350</td>
<td>0.4975</td>
<td>0.16</td>
</tr>
<tr>
<td>10%</td>
<td></td>
<td>2.9350</td>
<td>0.4975</td>
<td>0.26</td>
</tr>
<tr>
<td>20%</td>
<td></td>
<td>2.9350</td>
<td>0.4975</td>
<td>0.28</td>
</tr>
<tr>
<td>Differential</td>
<td></td>
<td>2.9152</td>
<td>0.2636</td>
<td>0.013</td>
</tr>
</tbody>
</table>

external entropy of a cluster $C_k$ with respect to attribute $A$ is defined as:

$$E(C_k) = \sum_j P(A = V_j) \log P(A = V_j)$$

and the expected entropy of the clusters is defined as

$$\bar{E}(ds) = \sum_k \left( \frac{|C_k|}{|D|} E(C_k) \right)$$

The smaller $\bar{E}(ds)$ is, the better the clustering results.

The category utility (CU) function is defined as:

$$CU = \sum_k \frac{|C_k|}{|D|} \sum_i \sum_j [P(A_i = V_{ij}|C_k)^2 - P(A_i = V_{ij})^2]$$

, where $A_i$ is the $i^{th}$ attribute and $V_{ij}$ is the $j^{th}$ value of $i^{th}$ attribute. The bigger $CU$ is, the better the clusters are.

Table 1 shows the results on congressional voting data [49]. It’s the United States Congressional Voting Records for the year 1984. Each attribute of the data set is the vote on one issue. $m$ is the percentage of re-processed points. It is a parameter of the COOLCAT algorithm. For this data set, $m$ does not affect the performance very much. The external entropy can only be a loose measurement here because the voting results may not always be consistent with what party the congressmen are in. For this dataset, our differential clustering algorithm achieves comparable clustering performances with the COOLCAT, but with much less time. In practice many data streams can provide a large amount of data in a very short time, which requires the algorithm to process data very fast. If the data flow is very slow, we can simply store the data to disk.
and run an off-the-shelf clustering algorithm. The reason of designing a new stream mining algorithm is to handle a much faster data flow.

3.6 Outlier Detection

Outlier detection is an important problem with many applications. This technique can be used to detect fraud, network intrusion, etc. Compared with other data mining problems, outlier detection might generate results that are easier for human users to understand.

As mentioned in section 2, there are many ways to attack the outlier detection problem. One method is clustering based. Our differential clustering algorithm can be used to generate outliers. Once a set of clusters $cls$ is generated, the outliers can be decided based on the cluster radius threshold $r$ and cluster size threshold $s$: First a centroid set $CS$ is constructed, where

$$CS = \{ \text{mode}(cl) | cl \in cls \land |cl| > s \}$$

Then a point $p$ is an outlier if

$$\forall ct \in CS, dist(p, ct) > r$$

Many existing outlier detection algorithms assume the non-existence of domain knowledge. In these approaches, outliers are selected based on the difference of a small percentage of data with a large portion of data. In other words, we assume that the majority of data would never be outliers. This is generally true when a huge amount of data (or the whole set of data) are provided. However, in stream environment, since we often only look at a “current window” of data, it’s very possible that the majority of data within a “window” is actually the outliers in the whole data set (or the data within a a much larger window). One way to solve this problem is to add some domain knowledge into the mining process. Bejar et al. talked about using domain knowledge to guide the clustering process [8], but generally how to couple the domain knowledge and models is a difficult problem.

Non-clustering based techniques can also be used for outlier detection. For example, if we can classify data points as “normal” and “abnormal” based on domain knowledge or some training data, then a stream version classification algorithm can output “abnormal” points as outliers based on the classification.

4 Evolution of Data

The evolution of data will be determined by the evolution of the models learned from the data. Once we have the stream mining algorithm, we can easily generate models for data seen so far, or for recent data. These models represent the “states” of the data stream. Immediately the following questions arise: how do we store the states, how do we detect and display the change of these states, and what kind of knowledge can we get from the changes. Ganti et al. [23] discussed change detection in terms of decision tree models and class of frequent itemsets. A statistical fit test can also be used to measure how a data set fits a model, therefore it can also be used to detect evolution of models.

4.1 Evolution of Clusters

Clustering is an effective technique to reduce the problem space. The clusters, or the centroids of the clusters can to a large extent represent the original data but with very small space, therefore some expensive analysis
There are many characteristics to describe clusters.

1. The centroid. The centroid, or the mean, or the “mode” of a cluster represented the majority or average value of attribute values.

2. The size (i.e. number of points in this cluster).

3. The radius. It’s the biggest distance from a point in the cluster to the cluster’s centroid. Depending on the distance metric “radius” can have different definitions.

4. The shape. A cluster’s shape can also be a feature of the cluster.

5. The distribution of each dimension.

6. The number of clusters.

7. The density of the clusters.

These characteristics and some other statistics of clusters can be used to detect and represent the evolution of time-related clusters. For example, suppose we are monitoring the network traffic related to a large set of computers in a public lab. The data attributes we are using can be ip address, port, number of packets, session time, etc. The network traffic log records can form clusters. If all the users are mainly visiting some web pages, then we can expect that the number of data packets from each ip address will be small and the source ip addresses of incoming packets will be scattered, and the boundary of clusters may be fuzzy. If suddenly a very dense cluster appears indicating that there are many data packets coming from a few ip addresses, then a system administrator needs to be notified. It may imply an attack, or simply that some new software release becomes available from several websites and users are downloading it. Either case is of interest to the system administrator and he/she needs to know it in a timely manner. We would like to explore how these features can be adopted to analyze the trend in the data for real applications.

An online evolution monitoring algorithm can have multiple user-defined triggers. For example, if the clustering algorithm is the differential clustering algorithm mentioned in section 3.2, then a trigger based on average(or minimum, maximum) cluster radius can easily be installed. During the process when data points in the buffer are added into existing clusters, the radius of each cluster can easily be maintained. Once the algorithm converges for the current block of data, the radius of each cluster can be compared against the threshold described in the trigger. If the radius exceeds the threshold, then the algorithm will generate an alert requiring human interaction.

Visualization is another important aspect in presenting the evolution of clusters to a user. Clusters will slightly change their shapes, diameters, centroids and so forth even if the underlying distribution of data does not change. How to distinguish these slight adjustments from “dramatic” changes caused by abnormal data can be difficult. In many cases, it is very complicated to describe the changes numerically to a user. However, if the clustering results are somehow visualized, then a human monitoring the process might easily notice the possible abnormalities. Therefore, a good visualization tool is helpful. The ideal tool should be able to present the clusters on-line, i.e., should be efficient enough to present the models as they are forming or shortly after they are formed. The disadvantage of visualization is that categorical data usually are very hard to present.

---

1Some of the clustering algorithms assume that the clusters form spheres in a high-dimensional space, but there are many algorithms that can capture clusters with different shapes.
Not only the changes themselves can be of interest, the rate of changing can also provide insight to the evolution. For example, the algorithm can monitor not only the drift of centroids, but also how fast the centroids are moving in the data space.

4.2 Evolution of Outliers

Similar to the other data mining models, outliers can also evolve. The changes in the outlier set, and the rate of the changes, are important as well as the outliers themselves.

The underlying model of a data stream can be evolving. This makes outliers relative to time. An outlier based on first block of data may not be an outlier after seeing the whole data set. In a stream environment we often can not see the whole data set and have to make decisions online, based on the observations we have so far.

The fact that we cannot see the whole data may make things complicated. If the data block in the “current window” is too small, the outlier we get from it may be meaningless. In the extreme case, we might get a burst of data in which almost all the data points are similar to each other. In that case, we might want to keep on sliding the window and wait until we see “sufficient” data in order to make a decision. How much data shall we see before making a decision depends on the outlier definition/parameters and the characteristics of data.

In practice, underlying models of stream data can slightly drift in “normal” cases. Therefore, we need a measurement of outlier changes. This measurement should be easy to compute, since we do not want to introduce too much overhead. It should also be easy to understand, because the users will need to set some threshold based on this measurement, so that some trigger can be installed in the online monitoring algorithm.

No matter what outlier detection approach is used, the process of detecting outliers could be considered as a process of training/computing the following function:

\[ f(\bar{x}) : D \rightarrow \{0, 1\} \]

where \( D \) is the block of data currently being studied, and

\[ f(\bar{x}) = \begin{cases} 
1 & (\bar{x} \text{ is an outlier}) \\
0 & (otherwise) 
\end{cases} \]

With the “current window” sliding, different \( f(\bar{x}) \)’s are generated. Suppose that when the current window is at the \( i^{th} \) position, the function is \( f_i \), and the set of outliers generated is \( OS_i \). We define the measurement of outlier changes as \( ocp(\text{outlier change percentage}) \). The \( ocp \) at \( (i + 1)^{st} \) window position is:

\[ ocp_i := 1 - \frac{\sum_{\bar{x} \in OS_i} f_{i+1}(\bar{x})}{|OS_i|} \] (3)

The user can set the \( ocp \) threshold to trigger a system alert, and \( ocp \) values for different windows can be plotted, and the rate of \( ocp \)’s changing can also be studied.

5 Evolution of Algorithms

Existing algorithms seldom take into account the change of flow rate of the data streams. As we mentioned earlier, stream mining algorithms need to be adaptive. When a batch of data arrives, generally there are the following options.
1. No processing. When the data flow rate is high, it is possible that the only thing that can be done is to buffer the data and hope the algorithm can get a chance to process it later. In the extreme case, maybe the buffering of all the data points may be impossible. In that case, a back-up sample of the data can be buffered. There are some previous results about keeping a sample of stream data [4].

2. Partial data processing. Get a sample from the data and only process this sample.

3. Low resolution processing. A set of algorithms is needed for this option. Among these algorithms, there should be “fast” algorithms that can generate a coarsely estimated results with little space and time, and elaborate algorithms that can generate accurate results with large space and relatively long time. When data flow rate is high, the “fast” algorithms are used.

4. High resolution processing. This is a companion to last option, which means that when time and space allows, elaborated algorithms can be used to get a more accurate model.

An analogue of low resolution and high resolution processing is the video broadcast: when network bandwidth is limited, frames may be dropped so the video’s resolution will go down. And when bandwidth is adequate, video data will be transmitted without any loss.

The transition among the four options needs guidelines. Multiple factors dynamically decide which option to choose. The selection of the options itself is a learning problem.

5.1 Cost Function

The maximum flow rate that can be handled is restricted by space and computational power. The space (memory needed for processing, temporary storage space for buffering data points) and time for each option, and the time/space for switching among the options all have to be considered. Although the stream mining framework is general, these time/space factors are all specific to the algorithms. And it may even depend on data (for example, some iterative algorithms converge faster for a certain distribution of data than for other distributions). It may not always be possible to get the exact value of these factors. Instead, a sufficiently accurate estimation of these factors and a reasonable switching strategy can be used to get the optimal (or sub-optimal) selection of options.

The “dynamic threshold compression schemes” mentioned in section 3.4.1 is a simple example of the selection based on cost. In that case, since we are not really dealing with changing flow rate, memory space is the main restriction for our processing. When the buffer is full, no more points can be taken in and the processing cannot continue. So the threshold is dynamically changed in order to compress data in the buffer (which certainly may sacrifice accuracy). In order to deal with all kinds of time/space restrictions and real-time requirements (see below), more strategies have to be designed.

5.2 Real-time Requirements

If we think of the periodically generated models as a live broadcast, then the real-time requirements have to be taken into consideration. The cost function is not the only measurement needed to decide the selection of options. Sometimes even if the temporary storage can hold more data points and later generate better results, in order to continue the processing and change detection in time, we may need to generate coarse models. In this case, different options may be executed in parallel: coarse estimations are generated for the immediate use, while later the models can be adjusted for further analysis.
6 Conclusion

The data stream paradigm has recently become the focus of much interest. Data streams can flow at variable rates and the underlying models often change with time. The current work in data stream mining does not focus on change (adaptation) and that is precisely our main focus. Monitoring the changes in the models becomes as important as obtaining the models. Therefore, stream data mining not only needs to mine data incrementally and decrementally (in order to keep track of recent data), but also has to provide methods to monitor/detect the changes of underlying models. We consider this problem as data evolution. Of equal importance, the mining algorithms themselves need to be adaptive/dynamic when the flow rate of data streams change dramatically. The stream algorithms should be able to downgrade accuracy in order to handle a data burst, or to do a more thorough analysis when data flow is slow. We consider this problem as algorithm adaptation (evolution).

We will study both data evolution and algorithm evolution. We will provide efficient algorithms to incrementally/decrementally mine stream data, good techniques to store data models and detect/monitor the changes, and a set of algorithms that can switch from “high resolution” to “low resolution” in order to adapt to the flow rate.

This work has importance to many real life applications that fit within the data stream model (e.g., network sensor data, bank ATM transaction data and web server log data) and our work will provide for more efficient data mining methods as well as for the discovery of potentially valuable information with regard to data evolution.

References


