Chapter 2

ON CLUSTERING MASSIVE DATA STREAMS: A SUMMARIZATION PARADIGM

Charu C. Aggarwal
IBM T. J. Watson Research Center
Hawthorne, NY 10532
charu@us.ibm.com

Jiawei Han
University of Illinois at Urbana-Champaign
Urbana, IL
hanj@cs.uiuc.edu

Jianyong Wang
University of Illinois at Urbana-Champaign
Urbana, IL
jianyong@tsinghua.edu.cn

Philip S. Yu
IBM T. J. Watson Research Center
Hawthorne, NY 10532
psyu@us.ibm.com

Abstract

In recent years, data streams have become ubiquitous because of the large number of applications which generate huge volumes of data in an automated way. Many existing data mining methods cannot be applied directly on data streams because of the fact that the data needs to be mined in one pass. Furthermore, data streams show a considerable amount of temporal locality because of which a direct application of the existing methods may lead to misleading results. In this paper, we develop an efficient and effective approach for mining fast evolving data streams, which integrates the *micro-clustering* technique

with the high-level data mining process, and discovers data evolution regularities as well. Our analysis and experiments demonstrate two important data mining problems, namely *stream clustering* and *stream classification*, can be performed effectively using this approach, with high quality mining results. We discuss the use of micro-clustering as a general summarization technology to solve data mining problems on streams. Our discussion illustrates the importance of our approach for a variety of mining problems in the data stream domain.

1. Introduction

In recent years, advances in hardware technology have allowed us to automatically record transactions and other pieces of information of everyday life at a rapid rate. Such processes generate huge amounts of online data which grow at an unlimited rate. These kinds of online data are referred to as *data streams*. The issues on management and analysis of data streams have been researched extensively in recent years because of its emerging, imminent, and broad applications [11, 14, 17, 23].

Many important problems such as clustering and classification have been widely studied in the data mining community. However, a majority of such methods may not be working effectively on data streams. Data streams pose special challenges to a number of data mining algorithms, not only because of the huge volume of the online data streams, but also because of the fact that the data in the streams may show temporal correlations. Such temporal correlations may help disclose important data evolution characteristics, and they can also be used to develop efficient and effective mining algorithms. Moreover, data streams require *online mining*, in which we wish to mine the data in a continuous fashion. Furthermore, the system needs to have the capability to perform an *offline analysis* as well based on the user interests. This is similar to an online analytical processing (OLAP) framework which uses the paradigm of pre-processing once, querying many times.

Based on the above considerations, we propose a new stream mining framework, which adopts a tilted time window framework, takes micro-clustering as a preprocessing process, and integrates the preprocessing with the incremental, dynamic mining process. Micro-clustering preprocessing effectively compresses the data, preserves the general temporal locality of data, and facilitates both online and offline analysis, as well as the analysis of current data and data evolution regularities.

In this study, we primarily concentrate on the application of this technique to two problems: (1) stream clustering, and (2) stream classification. The heart of the approach is to use an online summarization approach which is efficient and also allows for effective processing of the data streams. We also discuss

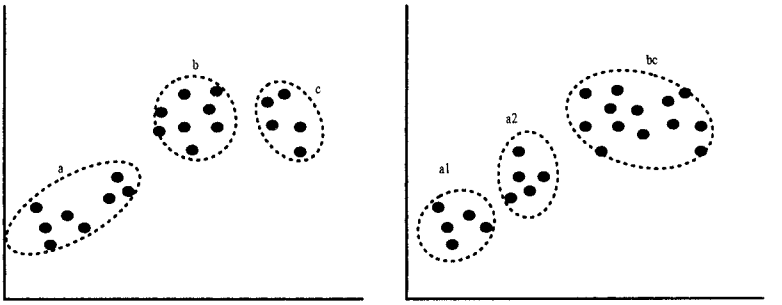

Figure 2.1. Micro-clustering Examples

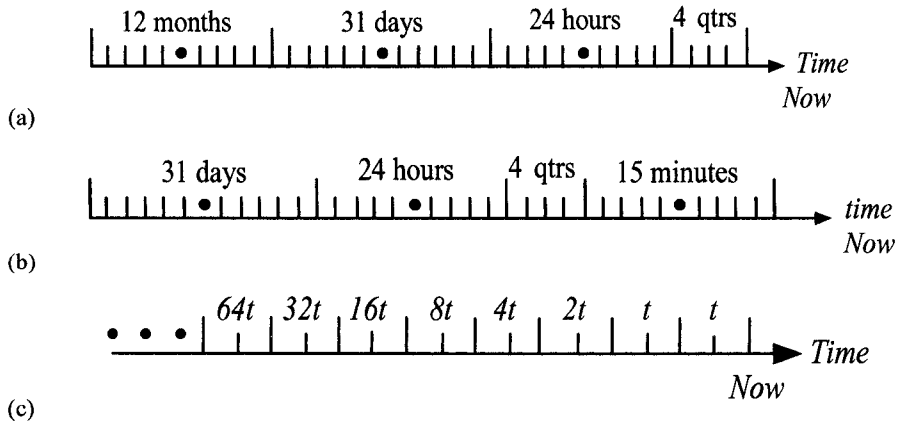

Figure 2.2. Some Simple Time Windows

a number of research directions, in which we show how the approach can be adapted to a variety of other problems.

This paper is organized as follows. In the next section, we will present our micro-clustering based stream mining framework. In section 3, we discuss the stream clustering problem. The classification methods are developed in Section 4. In section 5, we discuss a number of other problems which can be solved with the micro-clustering approach, and other possible research directions. In section 6, we will discuss some empirical results for the clustering and classification problems. In Section 7 we discuss the issues related to our proposed stream mining methodology and compare it with other related work. Section 8 concludes our study.

2. The Micro-clustering Based Stream Mining Framework

In order to apply our technique to a variety of data mining algorithms, we utilize a micro-clustering based stream mining framework. This framework is designed by capturing summary information about the nature of the data stream. This summary information is defined by the following structures:

• **Micro-clusters:** We maintain statistical information about the data locality in terms of micro-clusters. These micro-clusters are defined as a temporal extension of the *cluster feature vector* [24]. The additivity property of the micro-clusters makes them a natural choice for the data stream problem.

• **Pyramidal Time Frame:** The micro-clusters are stored at snapshots in time which follow a pyramidal pattern. This pattern provides an effective trade-off between the storage requirements and the ability to recall summary statistics from different time horizons.

The summary information in the micro-clusters is used by an offline component which is dependent upon a wide variety of user inputs such as the time horizon or the granularity of clustering. In order to define the micro-clusters, we will introduce a few concepts. It is assumed that the data stream consists of a set of multi-dimensional records $\overline{X}_1 \ldots \overline{X}_k \ldots$ arriving at time stamps $T_1 \ldots T_k \ldots$. Each $\overline{X}_i$ is a multi-dimensional record containing d dimensions which are denoted by $\overline{X}_i = (x_i^1 \ldots x_i^d)$.

We will first begin by defining the concept of micro-clusters and pyramidal time frame more precisely.

DEFINITION 2.1 *A micro-cluster for a set of d-dimensional points $X_{i_1} \ldots X_{i_n}$ with time stamps $T_{i_1} \ldots T_{i_n}$ is the (2·d+3) tuple $(\overline{CF2^x}, \overline{CF1^x}, CF2^t, CF1^t, n)$, wherein $\overline{CF2^x}$ and $\overline{CF1^x}$ each correspond to a vector of d entries. The definition of each of these entries is as follows:*

• *For each dimension, the sum of the squares of the data values is maintained in $\overline{CF2^x}$. Thus, $\overline{CF2^x}$ contains d values. The p-th entry of $\overline{CF2^x}$ is equal to $\sum_{j=1}^{n} (x_{i_j}^p)^2$.*

• *For each dimension, the sum of the data values is maintained in $\overline{CF1^x}$. Thus, $\overline{CF1^x}$ contains d values. The p-th entry of $\overline{CF1^x}$ is equal to $\sum_{j=1}^{n} x_{i_j}^p$.*

• *The sum of the squares of the time stamps $T_{i_1} \ldots T_{i_n}$ is maintained in $CF2^t$.*

• *The sum of the time stamps $T_{i_1} \ldots T_{i_n}$ is maintained in $CF1^t$.*

• *The number of data points is maintained in n.*

We note that the above definition of micro-cluster maintains similar summary information as the cluster feature vector of [24], except for the additional information about time stamps. We will refer to this temporal extension of the cluster feature vector for a set of points $\mathcal{C}$ by $\overline{CFT}(\mathcal{C})$. As in [24], this summary

information can be expressed in an additive way over the different data points. This makes it a natural choice for use in data stream algorithms.

We note that the maintenance of a large number of micro-clusters is essential in the ability to maintain more detailed information about the micro-clustering process. For example, Figure 2.1 forms 3 clusters, which are denoted by a, b, c. At a later stage, evolution forms 3 different figures a1, a2, bc, with a split into a1 and a2, whereas b and c merged into bc. If we keep micro-clusters (each point represents a micro-cluster), such evolution can be easily captured. However, if we keep only 3 cluster centers a, b, c, it is impossible to derive later a1, a2, bc clusters since the information of more detailed points are already lost.

The data stream clustering algorithm discussed in this paper can generate approximate clusters in any user-specified length of history from the current instant. This is achieved by storing the micro-clusters at particular moments in the stream which are referred to as *snapshots*. At the same time, the current snapshot of micro-clusters is always maintained by the algorithm. The macro-clustering algorithm discussed at a later stage in this paper will use these finer level micro-clusters in order to create higher level clusters which can be more easily understood by the user. Consider for example, the case when the current clock time is t_c and the user wishes to find clusters in the stream based on a history of length h. Then, the macro-clustering algorithm discussed in this paper will use some of the additive properties of the micro-clusters stored at snapshots t_c and $(t_c - h)$ in order to find the higher level clusters in a history or *time horizon* of length h. Of course, since it is not possible to store the snapshots at each and every moment in time, it is important to choose particular instants of time at which it is possible to store the state of the micro-clusters so that clusters in any user specified time horizon $(t_c - h, t_c)$ can be approximated.

We note that some examples of time frames used for the clustering process are the natural time frame (Figure 2.2(a) and (b)), and the logarithmic time frame (Figure 2.2(c)). In the natural time frame the snapshots are stored at regular intervals. We note that the scale of the natural time frame could be based on the application requirements. For example, we could choose days, months or years depending upon the level of granularity required in the analysis. A more flexible approach is to use the logarithmic time frame in which different variations of the time interval can be stored. As illustrated in Figure 2.2(c), we store snapshots at times of $t, 2 \cdot t, 4 \cdot t \ldots$. The danger of this is that we may jump too far between successive levels of granularity. We need an intermediate solution which provides a good balance between storage requirements and the level of approximation which a user specified horizon can be approximated.

In order to achieve this, we will introduce the concept of a pyramidal time frame. In this technique, the snapshots are stored at differing levels of granularity depending upon the recency. Snapshots are classified into different *orders* which can vary from 1 to $\log(T)$, where T is the clock time elapsed since the

beginning of the stream. The order of a particular class of snapshots define the level of granularity in time at which the snapshots are maintained. The snapshots of different order are maintained as follows:

• Snapshots of the i-th order occur at time intervals of α^i, where α is an integer and $\alpha \geq 1$. Specifically, each snapshot of the i-th order is taken at a moment in time when the clock value[1] from the beginning of the stream is exactly divisible by α^i.

• At any given moment in time, only the last $\alpha + 1$ snapshots of order i are stored.

We note that the above definition allows for considerable redundancy in storage of snapshots. For example, the clock time of 8 is divisible by 2^0, 2^1, 2^2, and 2^3 (where $\alpha = 2$). Therefore, the state of the micro-clusters at a clock time of 8 simultaneously corresponds to order 0, order 1, order 2 and order 3 snapshots. From an implementation point of view, a snapshot needs to be maintained only once. We make the following observations:

• For a data stream, the maximum order of any snapshot stored at T time units since the beginning of the stream mining process is $\log_\alpha(T)$.

• For a data stream the maximum number of snapshots maintained at T time units since the beginning of the stream mining process is $(\alpha + 1) \cdot \log_\alpha(T)$.

• For any user specified time window of h, at least one stored snapshot can be found within $2 \cdot h$ units of the current time.

While the first two results are quite easy to see, the last one needs to be proven formally.

LEMMA 2.2 *Let h be a user-specified time window, t_c be the current time, and t_s be the time of the last stored snapshot of any order just before the time $t_c - h$. Then $t_c - t_s \leq 2 \cdot h$.*

Proof: Let r be the smallest integer such that $\alpha^r \geq h$. Therefore, we know that $\alpha^{r-1} < h$. Since we know that there are $\alpha+1$ snapshots of order $(r-1)$, at least one snapshot of order $r-1$ must *always* exist before t_c-h. Let t_s be the snapshot of order $r-1$ which occurs just before $t_c - h$. Then $(t_c - h) - t_s \leq \alpha^{r-1}$. Therefore, we have $t_c - t_s \leq h + \alpha^{r-1} < 2 \cdot h$.

Thus, in this case, it is possible to find a snapshot within a factor of 2 of any user-specified time window. Furthermore, the total number of snapshots which need to be maintained are relatively modest. For example, for a data stream running for 100 years with a clock time granularity of 1 second, the total number of snapshots which need to be maintained are given by $(2 + 1) \cdot \log_2(100 * 365 * 24 * 60 * 60) \approx 95$. This is quite a modest requirement given the fact that a snapshot within a factor of 2 can always be found within any user specified time window.

It is possible to improve the accuracy of time horizon approximation at a modest additional cost. In order to achieve this, we save the $\alpha^l + 1$ snapshots

Order of Snapshots	Clock Times (Last 5 Snapshots)
0	55 54 53 52 51
1	54 52 50 48 46
2	52 48 44 40 36
3	48 40 32 24 16
4	48 32 16
5	32

Table 2.1. An example of snapshots stored for $\alpha = 2$ and $l = 2$

of order r for $l > 1$. In this case, the storage requirement of the technique corresponds to $(\alpha^l + 1) \cdot \log_\alpha(T)$ snapshots. On the other hand, the accuracy of time horizon approximation also increases substantially. In this case, any time horizon can be approximated to a factor of $(1 + 1/\alpha^{l-1})$. We summarize this result as follows:

LEMMA 2.3 *Let h be a user specified time horizon, t_c be the current time, and t_s be the time of the last stored snapshot of any order just before the time $t_c - h$. Then $t_c - t_s \leq (1 + 1/\alpha^{l-1}) \cdot h$.*

Proof: Similar to previous case.

For larger values of l, the time horizon can be approximated as closely as desired. For example, by choosing $l = 10$, it is possible to approximate any time horizon within 0.2%, while a total of only $(2^{10} + 1) \cdot \log_2(100 * 365 * 24 * 60 * 60) \approx 32343$ snapshots are required for 100 years. Since historical snapshots can be stored on disk and only the current snapshot needs to be maintained in main memory, this requirement is quite feasible from a practical point of view. It is also possible to specify the pyramidal time window in accordance with user preferences corresponding to particular moments in time such as beginning of calendar years, months, and days. While the storage requirements and horizon estimation possibilities of such a scheme are different, all the algorithmic descriptions of this paper are directly applicable.

In order to clarify the way in which snapshots are stored, let us consider the case when the stream has been running starting at a clock-time of 1, and a use of $\alpha = 2$ and $l = 2$. Therefore $2^2 + 1 = 5$ snapshots of each order are stored. Then, at a clock time of 55, snapshots at the clock times illustrated in Table 2.1 are stored.

We note that a large number of snapshots are common among different orders. From an implementation point of view, the states of the micro-clusters at times of 16, 24, 32, 36, 40, 44, 46, 48, 50, 51, 52, 53, 54, and 55 are stored. It is easy to see that for more recent clock times, there is less distance between successive snapshots (better granularity). We also note that the storage requirements

estimated in this section do not take this redundancy into account. Therefore, the requirements which have been presented so far are actually worst-case requirements.

These redundancies can be eliminated by using a systematic rule described in [6], or by using a more sophisticated geometric time frame. In this technique, snapshots are classified into different *frame numbers* which can vary from 0 to a value no larger than $\log_2(T)$, where T is the maximum length of the stream. The frame number of a particular class of snapshots defines the level of granularity in time at which the snapshots are maintained. Specifically, snapshots of frame number i are stored at clock times which are divisible by 2^i, but not by 2^{i+1}. Therefore, snapshots of frame number 0 are stored only at odd clock times. It is assumed that for each frame number, at most $max\_capacity$ snapshots are stored.

We note that for a data stream, the maximum frame number of any snapshot stored at T time units since the beginning of the stream mining process is $\log_2(T)$. Since at most $max\_capacity$ snapshots of any order are stored, this also means that the maximum number of snapshots maintained at T time units since the beginning of the stream mining process is $(max\_capacity) \cdot \log_2(T)$. One interesting characteristic of the geometric time window is that for any user-specified time window of h, at least one stored snapshot can be found within a factor of 2 of the specified horizon. This ensures that sufficient granularity is available for analyzing the behavior of the data stream over different time horizons. We will formalize this result in the lemma below.

LEMMA 2.4 *Let h be a user-specified time window, and t_c be the current time. Let us also assume that $max\_capacity \geq 2$. Then a snapshot exists at time t_s, such that $h/2 \leq t_c - t_s \leq 2 \cdot h$.*

Proof: Let r be the smallest integer such that $h < 2^{r+1}$. Since r is the smallest such integer, it also means that $h \geq 2^r$. This means that for any interval $(t_c - h, t_c)$ of length h, at least one integer $t' \in (t_c - h, t_c)$ must exist which satisfies the property that $t' \bmod 2^{r-1} = 0$ and $t' \bmod 2^r \neq 0$. Let t' be the time stamp of the last (most current) such snapshot. This also means the following:

$$h/2 \leq t_c - t' < h \tag{2.1}$$

Then, if $max\_capacity$ is at least 2, the second last snapshot of order $(r-1)$ is also stored and has a time-stamp value of $t' - 2^r$. Let us pick the time $t_s = t' - 2^r$. By substituting the value of t_s, we get:

$$t_c - t_s = (t_c - t' + 2^r) \tag{2.2}$$

Since $(t_c - t') \geq 0$ and $2^r > h/2$, it easily follows from Equation 2.2 that $t_c - t_s > h/2$.

Frame no.	Snapshots (by clock time)
0	69 67 65
1	70 66 62
2	68 60 52
3	56 40 24
4	48 16
5	64 32

Table 2.2. A geometric time window

Since t' is the position of the latest snapshot of frame $(r-1)$ occurring before the current time t_c, it follows that $(t_c - t') \leq 2^r$. Subsituting this inequality in Equation 2.2, we get $t_c - t_s \leq 2^r + 2^r \leq h + h = 2 \cdot h$. Thus, we have:

$$h/2 \leq t_c - t_s \leq 2 \cdot h \tag{2.3}$$

The above result ensures that every possible horizon can be closely approximated within a modest level of accuracy. While the geometric time frame shares a number of conceptual similarities with the pyramidal time frame [6], it is actually quite different and also much more efficient. This is because it eliminates the double counting of the snapshots over different frame numbers, as is the case with the pyramidal time frame [6]. In Table 2.2, we present an example of a frame table illustrating snapshots of different frame numbers. The rules for insertion of a snapshot t (at time t) into the snapshot frame table are defined as follows: (1) if $(t \bmod 2^i) = 0$ but $(t \bmod 2^{i+1}) \neq 0$, t is inserted into $frame\_number\ i$ (2) each slot has a $max\_capacity$ (which is 3 in our example). At the insertion of t into $frame\_number\ i$, if the slot already reaches its $max\_capacity$, the oldest snapshot in this frame is removed and the new snapshot inserted. For example, at time 70, since $(70 \bmod 2^1) = 0$ but $(70 \bmod 2^2) \neq 0$, 70 is inserted into frame_number 1 which knocks out the oldest snapshot 58 if the slot capacity is 3. Following this rule, when slot capacity is 3, the following snapshots are stored in the geometric time window table: 16, 24, 32, 40, 48, 52, 56, 60, 62, 64, 65, 66, 67, 68, 69, 70, as shown in Table 2.2. From the table, one can see that the closer to the current time, the denser are the snapshots stored.

3. Clustering Evolving Data Streams: A Micro-clustering Approach

The clustering problem is defined as follows: for a given set of data points, we wish to partition them into one or more groups of similar objects. The similarity of the objects with one another is typically defined with the use of some distance measure or objective function. The clustering problem has been

widely researched in the database, data mining and statistics communities [12, 18, 22, 20, 21, 24] because of its use in a wide range of applications. Recently, the clustering problem has also been studied in the context of the data stream environment [17, 23].

A previous algorithm called STREAM [23] assumes that the clusters are to be computed over the entire data stream. While such a task may be useful in many applications, a clustering problem may often be defined only over a portion of a data stream. This is because a data stream should be viewed as an infinite process consisting of data which continuously evolves with time. As a result, the underlying clusters may also change considerably with time. The nature of the clusters may vary with both the moment at which they are computed as well as the time horizon over which they are measured. For example, a data analyst may wish to examine clusters occurring in the last month, last year, or last decade. Such clusters may be considerably different. Therefore, we assume that one of the inputs to the clustering algorithm is a time horizon over which the clusters are found. Next, we will discuss CluStream, the online algorithm used for clustering data streams.

3.1 Micro-clustering Challenges

We note that since stream data naturally imposes a one-pass constraint on the design of the algorithms, it becomes more difficult to provide such a flexibility in computing clusters over different kinds of time horizons using conventional algorithms. For example, a direct extension of the stream based k-means algorithm in [23] to such a case would require a simultaneous maintenance of the intermediate results of clustering algorithms over all possible time horizons. Such a computational burden increases with progression of the data stream and can rapidly become a bottleneck for online implementation. Furthermore, in many cases, an analyst may wish to determine the clusters at a previous moment in time, and compare them to the current clusters. This requires even greater book-keeping and can rapidly become unwieldy for fast data streams.

Since a data stream cannot be revisited over the course of the computation, the clustering algorithm needs to maintain a substantial amount of information so that important details are not lost. For example, the algorithm in [23] is implemented as a continuous version of k-means algorithm which continues to maintain a number of cluster centers which change or merge as necessary throughout the execution of the algorithm. Such an approach is especially risky when the characteristics of the stream change over time. This is because the amount of information maintained by a k-means type approach is too approximate in granularity, and once two cluster centers are joined, there is no way to informatively split the clusters when required by the changes in the stream at a later stage.

Therefore a natural design to stream clustering would be separate out the process into an online micro-clustering component and an offline macro-clustering component. The online micro-clustering component requires a very efficient process for storage of appropriate summary statistics in a fast data stream. The offline component uses these summary statistics in conjunction with other user input in order to provide the user with a quick understanding of the clusters whenever required. Since the offline component requires only the summary statistics as input, it turns out to be very efficient in practice. This leads to several challenges:

- What is the nature of the summary information which can be stored efficiently in a continuous data stream? The summary statistics should provide sufficient temporal and spatial information for a horizon specific offline clustering process, while being prone to an efficient (online) update process.
- At what moments in time should the summary information be stored away on disk? How can an effective trade-off be achieved between the storage requirements of such a periodic process and the ability to cluster for a specific time horizon to within a desired level of approximation?
- How can the periodic summary statistics be used to provide clustering and evolution insights over user-specified time horizons?

3.2 Online Micro-cluster Maintenance: The CluStream Algorithm

The micro-clustering phase is the online statistical data collection portion of the algorithm. This process is not dependent on any user input such as the time horizon or the required granularity of the clustering process. The aim is to maintain statistics at a sufficiently high level of (temporal and spatial) granularity so that it can be effectively used by the offline components such as horizon-specific macro-clustering as well as evolution analysis. The basic concept of the micro-cluster maintenance algorithm derives ideas from the k-means and nearest neighbor algorithms. The algorithm works in an iterative fashion, by always maintaining a current set of micro-clusters. It is assumed that a total of q micro-clusters are stored at any moment by the algorithm. We will denote these micro-clusters by $\mathcal{M}_1 \ldots \mathcal{M}_q$. Associated with each micro-cluster i, we create a unique id whenever it is first created. If two micro-clusters are merged (as will become evident from the details of our maintenance algorithm), a *list* of ids is created in order to identify the constituent micro-clusters. The value of q is determined by the amount of main memory available in order to store the micro-clusters. Therefore, typical values of q are significantly larger than the natural number of clusters in the data but are also significantly smaller than the number of data points arriving in a long period of time for a massive data stream. These micro-clusters represent the current snapshot of clusters

which change over the course of the stream as new points arrive. Their status is stored away on disk whenever the clock time is divisible by α^i for any integer i. At the same time any micro-clusters of order r which were stored at a time in the past more remote than α^{l+r} units are deleted by the algorithm.

We first need to create the initial q micro-clusters. This is done using an offline process at the very beginning of the data stream computation process. At the very beginning of the data stream, we store the first $InitNumber$ points on disk and use a standard k-means clustering algorithm in order to create the q initial micro-clusters. The value of $InitNumber$ is chosen to be as large as permitted by the computational complexity of a k-means algorithm creating q clusters.

Once these initial micro-clusters have been established, the online process of updating the micro-clusters is initiated. Whenever a new data point $\overline{X_{i_k}}$ arrives, the micro-clusters are updated in order to reflect the changes. Each data point either needs to be absorbed by a micro-cluster, or it needs to be put in a cluster of its own. The first preference is to absorb the data point into a currently existing micro-cluster. We first find the distance of each data point to the micro-cluster centroids $\mathcal{M}_1 \ldots \mathcal{M}_q$. Let us denote this distance value of the data point $\overline{X_{i_k}}$ to the centroid of the micro-cluster $\mathcal{M}_j$ by $dist(\mathcal{M}_j, \overline{X_{i_k}})$. Since the centroid of the micro-cluster is available in the cluster feature vector, this value can be computed relatively easily.

We find the closest cluster $\mathcal{M}_p$ to the data point $\overline{X_{i_k}}$. We note that in many cases, the point $\overline{X_{i_k}}$ does not naturally belong to the cluster $\mathcal{M}_p$. These cases are as follows:

- The data point $\overline{X_{i_k}}$ corresponds to an outlier.
- The data point $\overline{X_{i_k}}$ corresponds to the beginning of a new cluster because of evolution of the data stream.

While the two cases above cannot be distinguished until more data points arrive, the data point $\overline{X_{i_k}}$ needs to be assigned a (new) micro-cluster of its own with a unique id. How do we decide whether a completely new cluster should be created? In order to make this decision, we use the cluster feature vector of $\mathcal{M}_p$ to decide if this data point falls within the *maximum boundary* of the micro-cluster $\mathcal{M}_p$. If so, then the data point $\overline{X_{i_k}}$ is added to the micro-cluster $\mathcal{M}_p$ using the CF additivity property. The maximum boundary of the micro-cluster $\mathcal{M}_p$ is defined as a factor of t of the RMS deviation of the data points in $\mathcal{M}_p$ from the centroid. We define this as the *maximal boundary factor*. We note that the RMS deviation can only be defined for a cluster with more than 1 point. For a cluster with only 1 previous point, the maximum boundary is defined in a heuristic way. Specifically, we choose it to be r times that of the next closest cluster.

If the data point does not lie within the maximum boundary of the nearest micro-cluster, then a new micro-cluster must be created containing the data

point $\mathcal{X}_{i_k}$. This newly created micro-cluster is assigned a new id which can identify it uniquely at any future stage of the data steam process. However, in order to create this new micro-cluster, the number of other clusters must be reduced by one in order to create memory space. This can be achieved by either deleting an old cluster or joining two of the old clusters. Our maintenance algorithm first determines if it is safe to delete any of the current micro-clusters as outliers. If not, then a merge of two micro-clusters is initiated.

The first step is to identify if any of the old micro-clusters are possibly outliers which can be safely deleted by the algorithm. While it might be tempting to simply pick the micro-cluster with the fewest number of points as the micro-cluster to be deleted, this may often lead to misleading results. In many cases, a given micro-cluster might correspond to a point of considerable cluster presence in the past history of the stream, but may no longer be an active cluster in the recent stream activity. Such a micro-cluster can be considered an outlier from the current point of view. An ideal goal would be to estimate the average timestamp of the last m arrivals in each micro-cluster [2], and delete the micro-cluster with the least recent timestamp. While the above estimation can be achieved by simply storing the last m points in each micro-cluster, this increases the memory requirements of a micro-cluster by a factor of m. Such a requirement reduces the number of micro-clusters that can be stored by the available memory and therefore reduces the effectiveness of the algorithm.

We will find a way to approximate the average timestamp of the last m data points of the cluster $\mathcal{M}$. This will be achieved by using the data about the timestamps stored in the micro-cluster $\mathcal{M}$. We note that the timestamp data allows us to calculate the mean and standard deviation[3] of the arrival times of points in a given micro-cluster $\mathcal{M}$. Let these values be denoted by $\mu\mathcal{M}$ and $\sigma\mathcal{M}$ respectively. Then, we find the time of arrival of the $m/(2 \cdot n)$-th percentile of the points in $\mathcal{M}$ assuming that the timestamps are normally distributed. This timestamp is used as the approximate value of the recency. We shall call this value as the *relevance stamp* of cluster $\mathcal{M}$. When the least relevance stamp of any micro-cluster is below a user-defined threshold δ, it can be eliminated and a new micro-cluster can be created with a unique id corresponding to the newly arrived data point $\overline{X_{i_k}}$.

In some cases, none of the micro-clusters can be readily eliminated. This happens when all relevance stamps are sufficiently recent and lie above the user-defined threshold δ. In such a case, two of the micro-clusters need to be merged. We merge the two micro-clusters which are closest to one another. The new micro-cluster no longer corresponds to one id. Instead, an $idlist$ is created which is a union of the the ids in the individual micro-clusters. Thus, any micro-cluster which is result of one or more merging operations can be identified in terms of the individual micro-clusters merged into it.

While the above process of updating is executed at the arrival of each data point, an additional process is executed at each clock time which is divisible by α^i for any integer i. At each such time, we store away the current set of micro-clusters (possibly on disk) together with their id list, and indexed by their time of storage. We also delete the least recent snapshot of order i, if $\alpha^l + 1$ snapshots of such order had already been stored on disk, and if the clock time for this snapshot is not divisible by α^{i+1}. (In the latter case, the snapshot continues to be a viable snapshot of order $(i+1)$.) These micro-clusters can then be used to form higher level clusters or an evolution analysis of the data stream.

3.3 High Dimensional Projected Stream Clustering

The method can also be extended to the case of high dimensional projected stream clustering . The algorithms is referred to as HPSTREAM. The high-dimensional case presents a special challenge to clustering algorithms even in the traditional domain of static data sets. This is because of the sparsity of the data in the high-dimensional case. In high-dimensional space, all pairs of points tend to be almost equidistant from one another. As a result, it is often unrealistic to define distance-based clusters in a meaningful way. Some recent work on high-dimensional data uses techniques for *projected clustering* which can determine clusters for a specific subset of dimensions [1, 4]. In these methods, the definitions of the clusters are such that each cluster is specific to a particular group of dimensions. This alleviates the sparsity problem in high-dimensional space to some extent. Even though a cluster may not be meaningfully defined on all the dimensions because of the sparsity of the data, some subset of the dimensions can always be found on which particular subsets of points form high quality and meaningful clusters. Of course, these subsets of dimensions may vary over the different clusters. Such clusters are referred to as *projected clusters* [1].

In [8], we have discussed methods for high dimensional projected clustering of data streams. The basic idea is to use an (incremental) algorithm in which we associate a set of dimensions with each cluster. The set of dimensions is represented as a d-dimensional bit vector $\mathcal{B}(\mathcal{C}_i)$ for each cluster structure in $\mathcal{FCS}$. This bit vector contains a 1 bit for each dimension which is included in cluster $\mathcal{C}_i$. In addition, the maximum number of clusters k and the average cluster dimensionality l is used as an input parameter. The average cluster dimensionality l represents the average number of dimensions used in the cluster projection. An iterative approach is used in which the dimensions are used to update the clusters and vice-versa. The structure in $\mathcal{FCS}$ uses a decay-based mechanism in order to adjust for evolution in the underlying data stream. Details are discussed in [8].

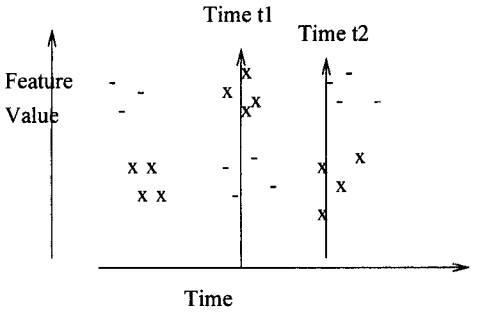

Figure 2.3. Varying Horizons for the classification process

4. Classification of Data Streams: A Micro-clustering Approach

One important data mining problem which has been studied in the context of data streams is that of stream classification [15]. The main thrust on data stream mining in the context of classification has been that of one-pass mining [14, 19]. In general, the use of one-pass mining does not recognize the changes which have occurred in the model since the beginning of the stream construction process [5]. While the work in [19] works on time changing data streams, the focus is on providing effective methods for incremental updating of the classification model. We note that the accuracy of such a model cannot be greater than the best sliding window model on a data stream. For example, in the case illustrated in Figure 2.3, we have illustrated two classes (labeled by 'x' and '-') whose distribution changes over time. Correspondingly, the best horizon at times t_1 and t_2 will also be different. As our empirical results will show, the true behavior of the data stream is captured in a temporal model which is sensitive to the level of evolution of the data stream.

The classification process may require simultaneous model construction and testing in an environment which constantly evolves over time. We assume that the testing process is performed concurrently with the training process. This is often the case in many practical applications, in which only a portion of the data is labeled, whereas the remaining is not. Therefore, such data can be separated out into the (labeled) training stream, and the (unlabeled) testing stream. The main difference in the construction of the micro-clusters is that the micro-clusters are associated with a class label; therefore an incoming data point in the training stream can only be added to a micro-cluster belonging to the same class. Therefore, we construct micro-clusters in almost the same way as the unsupervised algorithm, with an additional class-label restriction.

From the testing perspective, the important point to be noted is that the most effective classification model does not stay constant over time, but varies with

progression of the data stream. If a static classification model were used for an evolving test stream, the accuracy of the underlying classification process is likely to drop suddenly when there is a sudden burst of records belonging to a particular class. In such a case, a classification model which is constructed using a smaller history of data is likely to provide better accuracy. In other cases, a longer history of training provides greater robustness.

In the classification process of an evolving data stream, either the short term or long term behavior of the stream may be more important, and it often cannot be known a-priori as to which one is more important. How do we decide the window or horizon of the training data to use so as to obtain the best classification accuracy? While techniques such as decision trees are useful for one-pass mining of data streams [14, 19], these cannot be easily used in the context of an *on-demand classifier* in an evolving environment. This is because such a classifier requires rapid variation in the horizon selection process due to data stream evolution. Furthermore, it is too expensive to keep track of the entire history of the data in its original fine granularity. Therefore, the on-demand classification process still requires the appropriate machinery for efficient statistical data collection in order to perform the classification process.

4.1 On-Demand Stream Classification

We use the micro-clusters to perform an *On Demand Stream Classification Process*. In order to perform effective classification of the stream, it is important to find the correct time-horizon which should be used for classification. How do we find the most effective horizon for classification at a given moment in time? In order to do so, a small portion of the training stream is not used for the creation of the micro-clusters. This portion of the training stream is referred to as the horizon fitting stream segment. The number of points in the stream used for horizon fitting is denoted by k_{fit}. The remaining portion of the training stream is used for the creation and maintenance of the class-specific micro-clusters as discussed in the previous section.

Since the micro-clusters are based on the entire history of the stream, they cannot directly be used to test the effectiveness of the classification process over different time horizons. This is essential, since we would like to find the time horizon which provides the greatest accuracy during the classification process. We will denote the set of micro-clusters at time t_c and horizon h by $\mathcal{N}(t_c, h)$. This set of micro-clusters is determined by subtracting out the micro-clusters at time $t_c - h$ from the micro-clusters at time t_c. The subtraction operation is naturally defined for the micro-clustering approach. The essential idea is to match the micro-clusters at time t_c to the micro-clusters at time $t_c - h$, and subtract out the corresponding statistics. The additive property of micro-

clusters ensures that the resulting clusters correspond to the horizon $(t_c - h, t_c)$. More details can be found in [6].

Once the micro-clusters for a particular time horizon have been determined, they are utilized to determine the classification accuracy of that particular horizon. This process is executed periodically in order to adjust for the changes which have occurred in the stream in recent time periods. For this purpose, we use the horizon fitting stream segment. The last k_{fit} points which have arrived in the horizon fitting stream segment are utilized in order to test the classification accuracy of that particular horizon. The value of k_{fit} is chosen while taking into consideration the computational complexity of the horizon accuracy estimation. In addition, the value of k_{fit} should be small enough so that the points in it reflect the immediate locality of t_c. Typically, the value of k_{fit} should be chosen in such a way that the least recent point should be no larger than a pre-specified number of time units from the current time t_c. Let us denote this set of points by $\mathcal{Q}_{fit}$. Note that since $\mathcal{Q}_{fit}$ is a part of the training stream, the class labels are known a-priori.

In order to test the classification accuracy of the process, each point $\overline{X} \in \mathcal{Q}_{fit}$ is used in the following nearest neighbor classification procedure:

- We find the closest micro-cluster in $\mathcal{N}(t_c, h)$ to $\overline{X}$.
- We determine the class label of this micro-cluster and compare it to the true class label of $\overline{X}$. The accuracy over all the points in $\mathcal{Q}_{fit}$ is then determined. This provides the accuracy over that particular time horizon.

The accuracy of all the time horizons which are tracked by the geometric time frame are determined. The p time horizons which provide the greatest dynamic classification accuracy (using the last k_{fit} points) are selected for the classification of the stream. Let us denote the corresponding horizon values by $\mathcal{H} = \{h_1 \ldots h_p\}$. We note that since k_{fit} represents only a small locality of the points within the current time period t_c, it would seem at first sight that the system would always pick the smallest possible horizons in order to maximize the accuracy of classification. However, this is often not the case for evolving data streams. Consider for example, a data stream in which the records for a given class arrive for a period, and then subsequently start arriving again after a time interval in which the records for another class have arrived. In such a case, the horizon which includes previous occurrences of the same class is likely to provide higher accuracy than shorter horizons. Thus, such a system dynamically adapts to the most effective horizon for classification of data streams. In addition, for a stable stream the system is also likely to pick larger horizons because of the greater accuracy resulting from use of larger data sizes.

The classification of the test stream is a separate process which is executed continuously throughout the algorithm. For each given test instance $\overline{X_t}$, the above described nearest neighbor classification process is applied using each $h_i \in \mathcal{H}$. It is often possible that in the case of a rapidly evolving data stream, different horizons may report result in the determination of different class labels. The majority class among these p class labels is reported as the relevant class. More details on the technique may be found in [7].

5. Other Applications of Micro-clustering and Research Directions

While this paper discusses two applications of micro-clustering, we note that a number of other problems can be handled with the micro-clustering approach. This is because the process of micro-clustering creates a summary of the data which can be leveraged in a variety of ways for other problems in data mining. Some examples of such problems are as follows:

- **Privacy Preserving Data Mining:** In the problem of privacy preserving data mining, we create condensed representations [3] of the data which show k-anonymity. These condensed representations are like micro-clusters, except that each cluster has a minimum cardinality threshold on the number of data points in it. Thus, each cluster contains at least k data-points, and we ensure that the each record in the data cannot be distinguished from at least k other records. For this purpose, we only maintain the summary statistics for the data points in the clusters as opposed to the individual data points themselves. In addition to the first and second order moments we also maintain the covariance matrix for the data in each cluster. We note that the covariance matrix provides a complete overview of the distribution of in the data. This covariance matrix can be used in order to generate the pseudo-points which match the distribution behavior of the data in each micro-cluster. For relatively small micro-clusters, it is possible to match the probabilistic distribution in the data fairly closely. The pseudo-points can be used as a surrogate for the actual data points in the clusters in order to generate the relevant data mining results. Since the pseudo-points match the original distribution quite closely, they can be used for the purpose of a variety of data mining algorithms. In [3], we have illustrated the use of the privacy-preserving technique in the context of the classification problem. Our results show that the classification accuracy is not significantly reduced because of the use of pseudo-points instead of the individual data points.

- **Query Estimation:** Since micro-clusters encode summary information about the data, they can also be used for query estimation . A typical example of such a technique is that of estimating the selectivity of queries.

In such cases, the summary statistics of micro-clusters can be used in order to estimate the number of data points which lie within a certain interval such as a range query. Such an approach can be very efficient in a variety of applications since voluminous data streams are difficult to use if they need to be utilized for query estimation. However, the micro-clustering approach can condense the data into summary statistics, so that it is possible to efficiently use it for various kinds of queries. We note that the technique is quite flexible as long as it can be used for different kinds of queries. An example of such a technique is illustrated in [9], in which we use the micro-clustering technique (with some modifications on the tracked statistics) for futuristic query processing in data streams.

- **Statistical Forecasting:** Since micro-clusters contain temporal and condensed information, they can be used for methods such as statistical forecasting of streams . While it can be computationally intensive to use standard forecasting methods with large volumes of data points, the micro-clustering approach provides a methodology in which the condensed data can be used as a surrogate for the original data points. For example, for a standard regression problem, it is possible to use the centroids of different micro-clusters over the various temporal time frames in order to estimate the values of the data points. These values can then be used for making aggregate statistical observations about the future. We note that this is a useful approach in many applications since it is often not possible to effectively make forecasts about the future using the large volume of the data in the stream. In [9], it has been shown how to use the technique for querying and analysis of future behavior of data streams.

In addition, we believe that the micro-clustering approach is powerful enough to accomodate a wide variety of problems which require information about the summary distribution of the data. In general, since many new data mining problems require summary information about the data, it is conceivable that the micro-clustering approach can be used as a methodology to store condensed statistics for general data mining and exploration applications.

6. Performance Study and Experimental Results

All of our experiments are conducted on a PC with Intel Pentium III processor and 512 MB memory, which runs Windows XP professional operating system. For testing the accuracy and efficiency of the CluStream algorithm, we compare CluStream with the STREAM algorithm [17, 23], the best algorithm reported so far for clustering data streams. CluStream is implemented according to the description in this paper, and the STREAM K-means is done strictly according to [23], which shows better accuracy than BIRCH [24]. To make the comparison fair, both CluStream and STREAM K-means use the same amount of memory.

Specifically, they use the same stream incoming speed, the same amount of memory to store intermediate clusters (called Micro-clusters in CluStream), and the same amount of memory to store the final clusters (called Macro-clusters in CluStream).

Because the synthetic datasets can be generated by controlling the number of data points, the dimensionality, and the number of clusters, with different distribution or evolution characteristics, they are used to evaluate the scalability in our experiments. However, since synthetic datasets are usually rather different from real ones, we will mainly use real datasets to test accuracy, cluster evolution, and outlier detection.

Real datasets. First, we need to find some real datasets that evolve significantly over time in order to test the effectiveness of CluStream. A good candidate for such testing is the KDD-CUP'99 Network Intrusion Detection stream data set which has been used earlier [23] to evaluate STREAM accuracy with respect to BIRCH. This data set corresponds to the important problem of automatic and real-time detection of cyber attacks. This is also a challenging problem for dynamic stream clustering in its own right. The offline clustering algorithms cannot detect such intrusions in real time. Even the recently proposed stream clustering algorithms such as BIRCH and STREAM cannot be very effective because the clusters reported by these algorithms are all generated from the entire history of data stream, whereas the current cases may have evolved significantly.

The Network Intrusion Detection dataset consists of a series of TCP connection records from two weeks of LAN network traffic managed by MIT Lincoln Labs. Each n record can either correspond to a normal connection, or an intrusion or attack. The attacks fall into four main categories: DOS (i.e., denial-of-service), R2L (i.e., unauthorized access from a remote machine), U2R (i.e., unauthorized access to local superuser privileges), and PROBING (i.e., surveillance and other probing). As a result, the data contains a total of five clusters including the class for "*normal connections*". The attack-types are further classified into one of 24 types, such as buffer-overflow, guess-passwd, neptune, portsweep, rootkit, smurf, warezclient, spy, and so on. It is evident that each specific attack type can be treated as a sub-cluster. Most of the connections in this dataset are *normal*, but occasionally there could be a burst of attacks at certain times. Also, each connection record in this dataset contains 42 attributes, such as duration of the connection, the number of data bytes transmitted from source to destination (and vice versa), percentile of connections that have "SYN" errors, the number of "root" accesses, etc. As in [23], all 34 continuous attributes will be used for clustering and one outlier point has been removed.

Second, besides testing on the rapidly evolving network intrusion data stream, we also test our method over relatively stable streams. Since previously re-

ported stream clustering algorithms work on the entire history of stream data, we believe that they should perform effectively for some data sets with stable distribution over time. An example of such a data set is the KDD-CUP'98 Charitable Donation data set. We will show that even for such datasets, the CluStream can consistently beat the STREAM algorithm.

The KDD-CUP'98 Charitable Donation data set has also been used in evaluating several one-scan clustering algorithms, such as [16]. This data set contains 95412 records of information about people who have made charitable donations in response to direct mailing requests, and clustering can be used to group donors showing similar donation behavior. As in [16], we will only use 56 fields which can be extracted from the total 481 fields of each record. This data set is converted into a data stream by taking the data input order as the order of streaming and assuming that they flow-in with a uniform speed.

Synthetic datasets. To test the scalability of CluStream, we generate some synthetic datasets by varying base size from 100K to 1000K points, the number of clusters from 4 to 64, and the dimensionality in the range of 10 to 100. Because we know the true cluster distribution a priori, we can compare the clusters found with the true clusters. The data points of each synthetic dataset will follow a series of Gaussian distributions, and to reflect the evolution of the stream data over time, we change the mean and variance of the current Gaussian distribution every 10K points in the synthetic data generation.

The quality of clustering on the real data sets was measured using the sum of square distance (SSQ), defined as follows. Assume that there are a total of N points in the past horizon at current time T_c. For each point p_i, we find the centroid C_{p_i} of its closest macro-cluster, and compute $d(p_i, C_{p_i})$, the distance between p_i and C_{p_i}. Then the SSQ at time T_c with horizon H (denoted as $SSQ(T_c, H)$) is equal to the sum of $d^2(p_i, C_{p_i})$ for all the N points within the previous horizon H. Unless otherwise mentioned, the algorithm parameters were set at $\alpha = 2$, $l = 10$, $InitNumber = 2000$, and $t = 2$.

We compare the clustering quality of CluStream with that of STREAM for different horizons at different times using the Network Intrusion dataset and the Charitable donation data set. The results are illustrated in Figures 2.4 and 2.5. We run each algorithm 5 times and compute their average SSQs. The results show that CluStream is almost always better than STREAM. All experiments for these datasets have shown that CluStream has substantially higher quality than STREAM. However the Network Intrusion data set showed significantly better results than the charitable donation data set because of the fact the network intrusion data set was a highly evolving data set. For such cases, the evolution sensitive CluStream algorithm was much more effective than the STREAM algorithm.

We also tested the accuracy of the *On-Demand Stream Classifier*. The first test was performed on the Network Intrusion Data Set. The first experiment

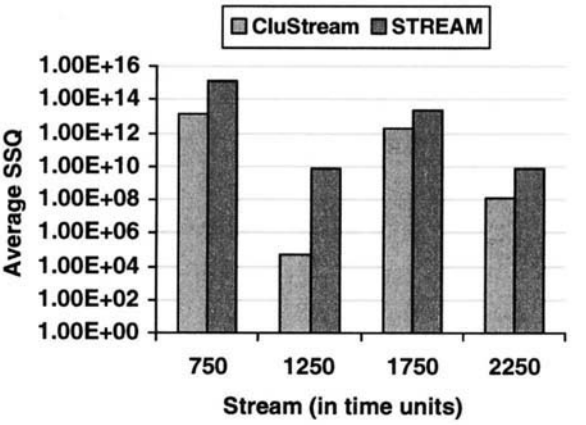

Figure 2.4. Quality comparison (Network Intrusion dataset, horizon=256, stream_speed=200)

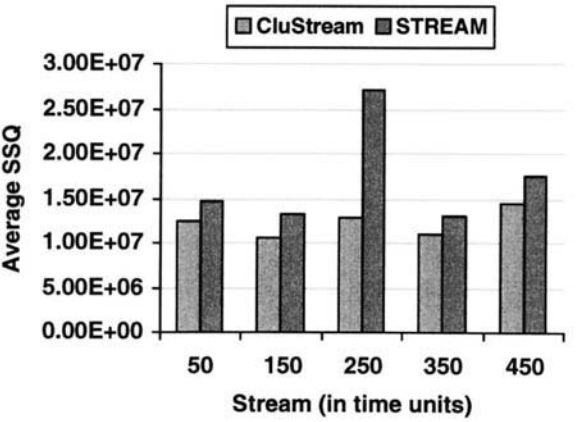

Figure 2.5. Quality comparison (Charitable Donation dataset, horizon=4, stream_speed=200)

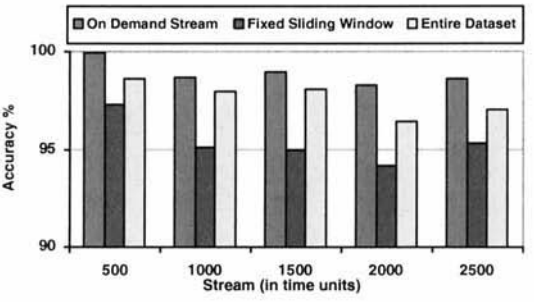

Figure 2.6. Accuracy comparison (Network Intrusion dataset, stream_speed=80, buffer_size=1600, k_{fit}=80, *init_number*=400)

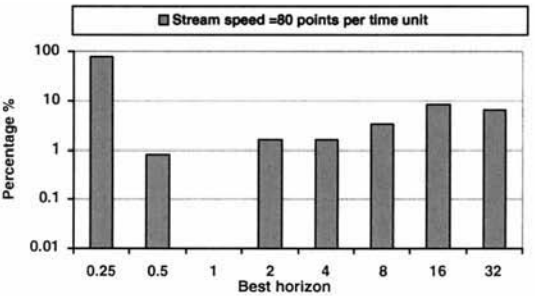

Figure 2.7. Distribution of the (smallest) best horizon (Network Intrusion dataset, Time units=2500, buffer_size=1600, k_{fit}=80, *init_number*=400)

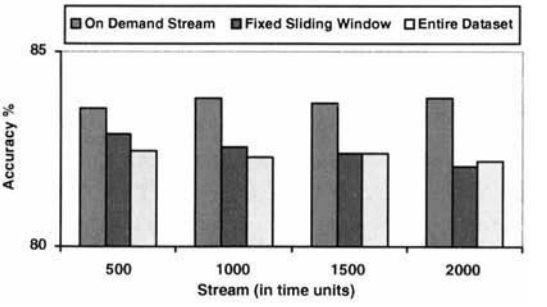

Figure 2.8. Accuracy comparison (Synthetic dataset B300kC5D20, stream_speed=100, buffer_size=500, k_{fit}=25, *init_number*=400)

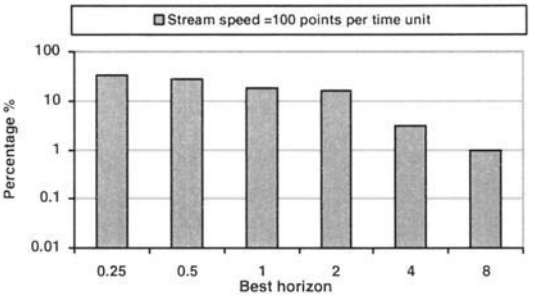

Figure 2.9. Distribution of the (smallest) best horizon (Synthetic dataset B300kC5D20, Time units=2000, buffer_size=500, k_{fit}=25, $init\_number$=400)

was conducted with a stream speed at 80 connections per time unit (i.e., there are 40 training stream points and 40 test stream points per time unit). We set the buffer_size at 1600 points, which means upon receiving 1600 points (including both training and test stream points) we'll use a small set of the training data points (In this case k_{fit} =80) to choose the best horizon. We compared the accuracy of the *On-Demand-Stream classifier* with two simple one-pass stream classifiers over the entire data stream and the selected sliding window (i.e., sliding window $H = 8$). Figure 2.6 shows the accuracy comparison among the three algorithms. We can see the *On-Demand-Stream classifier* consistently beats the two simple one-pass classifiers. For example, at time unit 2000, the *On-Demand-Stream classifier*'s accuracy is about 4% higher than the classifier with fixed sliding window, and is about 2% higher than the classifier with the entire dataset. Because the class distribution of this dataset evolves significantly over time, either the entire dataset or a fixed sliding window may not always capture the underlying stream evolution nature. As a result, they always have a worse accuracy than the *On-Demand-Stream classifier* which always dynamically chooses the best horizon for classifying.

Figure 2.7 shows the distribution of the best horizons (They are the smallest ones if there exist several best horizons at the same time). Although about 78.4% of the (smallest) best horizons have a value 1/4, there do exist about 21.6% best horizons ranging from 1/2 to 32 (e.g., about 6.4% of the best horizons have a value 32). This also illustrates that there is no fixed sliding window that can achieve the best accuracy and the reason why the *On-Demand-Stream classifier* can outperform the simple one-pass classifiers over either the entire dataset or a fixed sliding window.

We have also generated one synthetic dataset B300kC5D20 to test the classification accuracy of these algorithms. This dataset contains 5 class labels and 300K data points with 20 dimensions. We first set the stream speed at 100 points

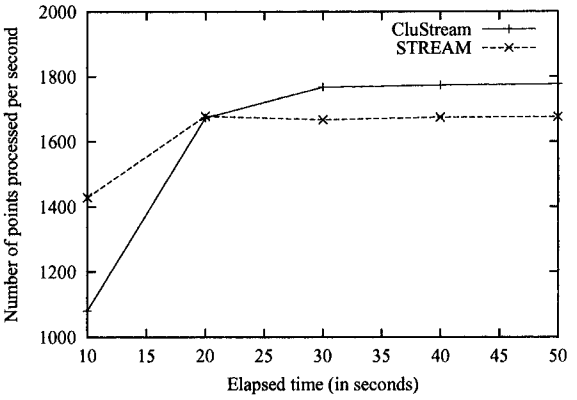

Figure 2.10. Stream Proc. Rate (Charit. Donation data, stream_speed=2000)

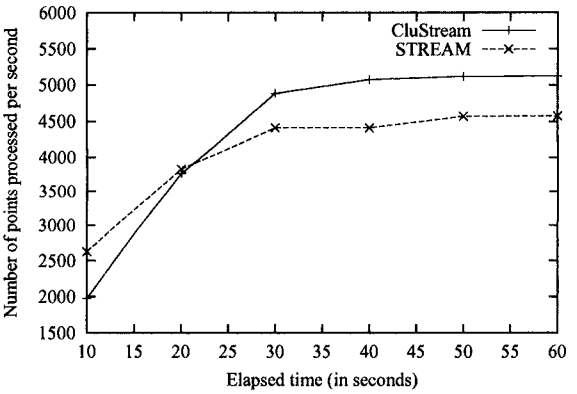

Figure 2.11. Stream Proc. Rate (Ntwk. Intrusion data, stream_speed=2000)

per time unit. Figure 2.8 shows the accuracy comparison among the three algortihms: The *On-Demand-Stream classifier* always has much better accuracy than the other two classifiers. Figure 2.9 shows the distribution of the (smallest) best horizons which can explain very well why the *On-Demand-Stream classifier* has better accuracy.

We also tested the efficiency of the micro-cluster maintenance algorithm with respect to STREAM on the real data sets. We note that this maintenance process needs to be performed both for the clustering and classificiation algorithms with minor differences. Therefore, we present the results for the case of clustering. By setting the number of micro-clusters to 10 times the number of natural clusters, Figures 2.10 and 2.11 show the stream processing rate (i.e.,

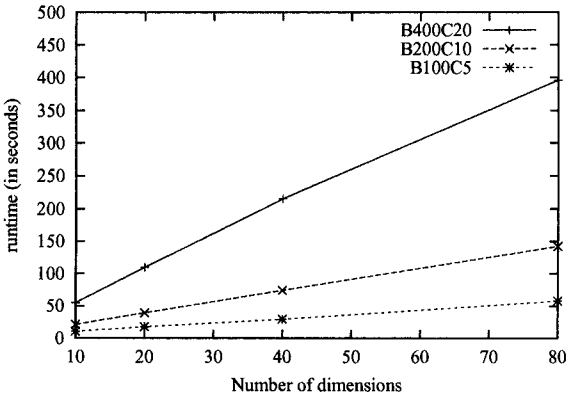

Figure 2.12. Scalability with Data Dimensionality (stream_speed=2000)

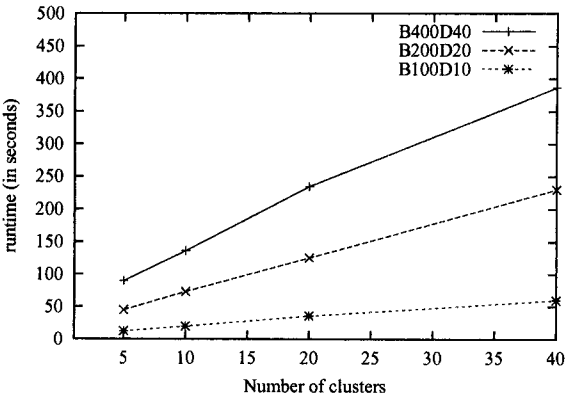

Figure 2.13. Scalability with Number of Clusters (stream_speed=2000)

the number of points processed per second) as opposed to the running time for two real data sets. Since CluStream requires some time to compute the initial set of micro-clusters, its precessing rate is lower than STREAM at the very beginning. However, once steady state is reached, CluStream becomes faster than STREAM in spite of the fact that it needs to store the snapshots to disk periodically. This is because STREAM takes a few iterations to make k-means clustering converge, whereas CluStream just needs to judge whether a set of points will be absorbed by the existing micro-clusters and insert into them appropriately.

The key to the success of micro-cluster maintenance is high scalability. This is because this process is exposed to a potentially large volume of incoming data and needs to be implemented in an efficient and online fashion. The most time-consuming and frequent operation during micro-cluster maintenance is that of finding the closest micro-cluster for each newly arrived data point. It is clear that the complexity of this operation increases linearly with the number of micro-clusters. It is also evident that the number of micro-clusters maintained should be sufficiently larger than the number of input clusters in the data in order to obtain a high quality clustering. While the number of input clusters cannot be known a priori, it is instructive to examine the scalability behavior when the number of micro-clusters was fixed at a constant large factor of the number of input clusters. Therefore, for all the experiments in this section, we will fix the number of micro-clusters to 10 times the number of input clusters. We will present the scalability behavior of the CluStream algorithm with data dimensionality, and the number of natural clusters.

The first series of data sets were generated by varying the dimensionality from 10 to 80, while fixing the number of points and input clusters. The first data set series B100C5 indicates that it contains 100K points and 5 clusters. The same notational convention is used for the second data set series B200C10 and the third one B400C20. Figure 2.12 shows the experimental results, from which one can see that CluStream has linear scalability with data dimensionality. For example, for dataset series B400C20, when the dimensionality increases from 10 to 80, the running time increases less than 8 times from 55 seconds to 396 seconds.

Another three series of datasets were generated to test the scalability against the number of clusters by varying the number of input clusters from 5 to 40, while fixing the stream size and dimensionality. For example, the first data set series B100D10 indicates it contains 100K points and 10 dimensions. The same convention is used for the other data sets. Figure 2.13 demonstrates that CluStream has linear scalability with the number of input clusters.

7. Discussion

In this paper, we have discussed effective and efficient methods for clustering and classification of data streams. The techniques discussed in this paper utilize a micro-clustering approach in conjunction with a pyramidal time window. The technique can be used to cluster different kinds of data streams, as well as create a classifier for the data. The methods have clear advantages over recent techniques which try to cluster the whole stream at one time rather than viewing the stream as a changing process over time. The CluStream model provides a wide variety of functionality in characterizing data stream clusters over different time horizons in an evolving environment.

This is achieved through a careful division of labor between the online statistical data collection component and an offline analytical component. Thus, the process provides considerable flexibility to an analyst in a real-time and changing environment. In order to achieve these goals, we needed to the design the statistical storage process of the online component very carefully. The use of a *pyramidal time window* assures that the essential statistics of *evolving* data streams can be captured without sacrificing the underlying *space- and time-efficiency* of the stream clustering process.

The essential idea behind the CluStream model is to perform effective data summarization so that the underlying summary data can be used for a host of tasks such as clustering and classification. Therefore, the technique provides a framework upon which many other data mining tasks can be built.

Notes

1. Without loss of generality, we can assume that one unit of clock time is the smallest level of granularity. Thus, the 0-th order snapshots measure the time intervals at the smallest level of granularity.

2. If the micro-cluster contains fewer than $2 \cdot m$ points, then we simply find the average timestamp of all points in the cluster.

3. The mean is equal to $CF1^t/n$. The standard deviation is equal to $\sqrt{CF2^t/n - (CF1^t/n)^2}$.

References

[1] Aggarwal C., Procopiuc C., Wolf J., Yu P., Park J.-S. (1999). Fast algorithms for projected clustering. *ACM SIGMOD Conference.*

[2] Aggarwal C., Yu P. (2000). Finding Generalized Projected Clusters in High Dimensional Spaces, *ACM SIGMOD Conference.*

[3] Aggarwal C., Yu P.. (2004). A Condensation Approach to Privacy Preserving Data Mining. *EDBT Conference.*

[4] Agrawal R., Gehrke J., Gunopulos D., Raghavan P (1998). Automatic Subspace Clustering of High Dimensional Data for Data Mining Applications. *ACM SIGMOD Conference.*

[5] Aggarwal C (2003). A Framework for Diagnosing Changes in Evolving Data Streams. ACM SIGMOD Conference.

[6] Aggarwal C., Han J., Wang J., Yu P (2003). A Framework for Clustering Evolving Data Streams. *VLDB Conference.*

[7] Aggarwal C, Han J., Wang J., Yu P. (2004). On-Demand Classification of Evolving Data Streams. *ACM KDD Conference.*

[8] Aggarwal C., Han J., Wang J., Yu P. (2004). A Framework for Projected Clustering of High Dimensional Data Streams. *VLDB Conference.*

[9] Aggarwal C. (2006) on Futuristic Query Processing in Data Streams. *EDBT Conference.*

[10] Ankerst M., Breunig M., Kriegel H.-P., Sander J. (1999). OPTICS: Ordering Points To Identify the Clustering Structure. *ACM SIGMOD Conference.*

[11] Babcock B., Babu S., Datar M., Motwani R., Widom J. (2002). Models and Issues in Data Stream Systems, *ACM PODS Conference.*

[12] Bradley P., Fayyad U., Reina C. (1998) Scaling Clustering Algorithms to Large Databases. *SIGKDD Conference.*

[13] Cortes C., Fisher K., Pregibon D., Rogers A., Smith F. (2000). Hancock: A Language for Extracting Signatures from Data Streams. ACM SIGKDD Conference.

[14] Domingos P., Hulten G. (2000). Mining High-Speed Data Streams. *ACM SIGKDD Conference.*

[15] Duda R., Hart P (1973). *Pattern Classification and Scene Analysis*, Wiley, New York.

[16] Farnstrom F,, Lewis J., Elkan C. (2000). Scalability for Clustering Algorithms Revisited. *SIGKDD Explorations*, 2(1):pp. 51–57.

[17] Guha S., Mishra N., Motwani R., O'Callaghan L. (2000). Clustering Data Streams. *IEEE FOCS Conference.*

[18] Guha S., Rastogi R., Shim K. (1998). CURE: An Efficient Clustering Algorithm for Large Databases. *ACM SIGMOD Conference.*

[19] Hulten G., Spencer L., Domingos P. (2001). Mining Time Changing Data Streams. *ACM KDD Conference.*

[20] Jain A., Dubes R. (1998). Algorithms for Clustering Data, *Prentice Hall,* New Jersey.

[21] Kaufman L., Rousseuw P. (1990). Finding Groups in Data- An Introduction to Cluster Analysis. *Wiley Series in Probability and Math. Sciences.*

[22] Ng R., Han J (1994). Efficient and Effective Clustering Methods for Spatial Data Mining. *Very Large Data Bases Conference.*

[23] O'Callaghan L., Mishra N., Meyerson A., Guha S., Motwani R (2002). Streaming-Data Algorithms For High-Quality Clustering. *ICDE Conference.*

[24] Zhang T., Ramakrishnan R., and Livny M (1996). BIRCH: An Efficient Data Clustering Method for Very Large Databases. *ACM SIGMOD Conference.*