On Clustering Techniques for Change Diagnosis in Data Streams

Charu C. Aggarwal and Philip S. Yu

IBM T. J. Watson Research Center
19 Skyline Drive, Hawthorne, NY 10532
{charu, psyu}@us.ibm.com

Abstract. In recent years, data streams have become ubiquitous in a variety of applications because of advances in hardware technology. Since data streams may be generated by applications which are time-changing in nature, it is often desirable to explore the underlying changing trends in the data. In this paper, we will explore and survey some of our recent methods for change detection. In particular, we will study methods for change detection which use clustering in order to provide a concise understanding of the underlying trends. We discuss our recent techniques which use micro-clustering in order to diagnose the changes in the underlying data. We also discuss the extension of this method to text and categorical data sets as well community detection in graph data streams.

1 Introduction

The results of many data stream mining applications such as clustering and classification [3,6,23] are affected by the changes in the underlying data. In many applications, change detection is a critical task in order to understand the nature of the underlying data. In many applications, it is desirable to have a concise description of the changes in the underlying data [1,2,9,14,15,23]. This can then be leveraged in order to make changes in the underlying task. A natural choice for creating concise descriptions of the change is the method of clustering. In this paper, we will examine a number of our recent techniques for change diagnosis of streams which utilize clustering as an approach. In particular, we will examine the method of micro-clustering, which can be used to examine changes in quantitative, categorical, or text data. We will also examine the technique of community detection in graph data streams.

The fast nature of data streams results in several constraints in their applicability to data mining tasks. For example, it means that they cannot be re-examined in the course of their computation. Therefore, all algorithms need to be executed in only one pass of the data. Clustering is a natural choice for summarizing the evolution of most kinds of data streams [3,23], since it provides a conduit for summarization of the data. These summaries can be used for a variety of tasks which depend upon the change diagnosis. We note that a different method for change diagnosis is discussed in [1], which provides visual summaries of the changing trends in the data. This can be very helpful for a

O. Nasraoui et al. (Eds.): WebKDD 2005, LNAI 4198, pp. 139–157, 2006.

variety of data mining tasks. In other applications in which the change needs to be reported in the context of a particular kind of segmentation, it is also helpful to use clustering for summarization methods. This paper will concentrate on such techniques.

One important fact about change detection applications is that most users apply these techniques only in the context of user-specific horizons. For example, a business analyst may wish to determine the changes in the data over a period of days, months, or years. This creates a natural challenge since a data stream does not allow the natural flexibility of re-examining the previous portions of the data. In this context, the use of clustering in conjunction with a pyramidal time frame can be very useful. The concept of the pyramid time frame has been discussed in [3], and turns out to be very useful for a number of applications.

The methods discussed in this paper are applicable to a wide variety of data types such as text, categorical data, and quantitative data. We will discuss the algorithms and methods in detail for each case. We will also discuss the extension of the method to graph data streams and the use of the technique for community detection and evolution.

This paper is organized as follows. In the next section, we will discuss the overall stream summarization framework, and its use for quantitative change detection. We will also provide a specific example of an intrusion detection application. In section 3, we will discuss how the technique can be used for text and categorical data. In section 4, we will discuss the use of the method for community detection and evolution. Section 5 discusses the conclusions and summary.

2 Micro-clustering and Quantitative Change Detection

The method of micro-clustering uses an extension of the cluster feature method discusses in [26]. Since data streams have a temporal component, we also need to keep track of the time stamps of arriving data points. In addition, we need to store the data periodically in a systematic way in order to access it later for the purposes of change diagnosis. Therefore, the data needs to be saved periodically on disk. A key issue is the choice of time periods over which the data should be saved. n this section, we will discuss both issue.

In order to store the current *state* of the clusters, we use a summary statistical representation which are referred to as *microclusters* [3]. The summary information in the microclusters 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 microclusters and pyramidal time frame more precisely.

Definition 1. *A microcluster 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* $(2 \cdot d + 3)$ *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 microcluster maintains similar summary information as the cluster feature vector of [26], 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 $\overline{CFT}(\mathcal{C})$. We note that the micro-clusters have a number of properties such as the *additivity property*, and the *linear update* property which makes them particularly suited to data streams:

- The additivity property ensures that it is possible to compute the micro-cluster statistics over a specific time horizon in order to determine the nature of the changes in the data.
- The linear update property is a direct consequence of the additivity property. Since each additional point can be incorporated into the cluster statistics by using additive operations pver the different dimensions, it ensures that the time required to maintain micro-cluster statistics increases linearly with the number of data points and dimensionality. This is an important property, since it ensures that the micro-cluster statistics are efficiently updateable.

Since recent data is more relevant than historical data, it is desirable to use a layered approach to saving the data at specific snapshots. For this purpose, we divide the snapshots into different *orders*, which follow the below rules:

- Snapshots of order i are stored at time intervals which are divisible by α^i.
- The last $\alpha + 1$ snapshots of order i are always stored.

We make the following observations about this pattern of storage:

- 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 facts are easy to confirm, the last needs to be validated explicitly. A proof of the last point is provided in [3]. We also note that the bound $(\alpha + 1) \cdot \log_\alpha(T)$ is an upper bound, since there is some overlapping between snapshots of different orders. In practice, only one copy of the snapshots need be maintained when there is overlap between snapshots of different orders.

The maintenance of snapshots is a fairly straightforward task. We only need to maintain the micro-cluster statistics dynamically, and update the cluster centers when necessary. In each iteration, we determine the closest micro-cluster center and assign the incoming data point to it. The cluster statistics are updated using this incoming data point. Because of the additivity property, this is a straightforward operation. The detailed clustering maintenance algorithm is discussed in [3]. Once the cluster statistics have been updated, we can periodically store them using the pyramidal time frame. The stored snapshots can then be used in order to determine the summary of the data evolution.

In the work of [1] and [12], the problem of change detection has been studied from the point of view of understanding how the data characteristics have changed over time. However, these papers do not deal with the problem of studying the changes in clusters over time. In the context of the clustering problem, such evolution analysis also has significant importance. For example, an analyst may wish to know how the clusters have changed over the last quarter, the last year, the last decade and so on. For this purpose, the user needs to input a few parameters to the algorithm:

- The two clock times t_1 and t_2 over which the clusters need to be compared. It is assumed that $t_2 > t_1$. In many practical scenarios, t_2 is the current clock time.
- The time horizon h over which the clusters are computed. This means that the clusters created by the data arriving between $(t_2 - h, t_2)$ are compared to those created by the data arriving between $(t_1 - h, t_1)$.

Another important issue is that of deciding how to present the changes in the clusters to a user, so as to make the results appealing from an intuitive point of view. We present the changes occurring in the clusters in terms of the following broad objectives:

- Are there new clusters in the data at time t_2 which were not present at time t_1?
- Have some of the original clusters been lost because of changes in the behavior of the stream?
- Have some of the original clusters at time t_1 shifted in position and nature because of changes in the data?
- Find the nature of the changes in the clusters at time t_1 till the time t_2.
- Find all the transient clusters from t_1 to time t_2. The transient clusters are defined as those which were born after time t_1, but expired before time t_2.

We note that the micro-cluster maintenance algorithm maintains the idlists which are useful for tracking cluster information. The first step is to compute $\mathcal{N}(t_1, h)$ and $\mathcal{N}(t_2, h)$ as discussed in the previous section. Therefore, we divide the micro-clusters in $\mathcal{N}(t_1, h) \cup \mathcal{N}(t_2, h)$ into three categories:

- Micro-clusters in $\mathcal{N}(t_2, h)$ for which none of the ids on the corresponding *idlist* are present in $\mathcal{N}(t_1, h)$. These are new micro-clusters which were created at some time in the interval (t_1, t_2). We will denote this set of micro-clusters by $\mathcal{M}^{added}(t_1, t_2)$.

- Micro-clusters in $\mathcal{N}(t_1, h)$ for which none of the corresponding ids are present in $\mathcal{N}(t_2, h)$. Thus, these micro-clusters were deleted in the interval (t_1, t_2). We will denote this set of micro-clusters by $\mathcal{M}^{deleted}(t_1, t_2)$.
- Micro-clusters in $\mathcal{N}(t_2, h)$ for which some or all of the ids on the corresponding *idlist* are present in the idlists corresponding to the micro-clusters in $\mathcal{N}(t_1, h)$. Such micro-clusters were at least partially created before time t_1, but have been modified since then. We will denote this set of micro-clusters by $\mathcal{M}^{retained}(t_1, t_2)$.

The macro-cluster creation algorithm is then separately applied to each of this set of micro-clusters to create a new set of higher level clusters. The macro-clusters created from $\mathcal{M}^{added}(t_1, t_2)$ and $\mathcal{M}^{deleted}(t_1, t_2)$ have clear significance in terms of clusters added to or removed from the data stream. The micro-clusters in $\mathcal{M}^{retained}(t_1, t_2)$ correspond to those portions of the stream which have not changed very significantly in this period. When a very large fraction of the data belongs to $\mathcal{M}^{retained}(t_1, t_2)$, this is a sign that the stream is quite stable over that time period. In many cases, the clusters in $\mathcal{M}^{retained}$ could change significantly over time. In such cases, we can utilize the statistics of the underlying data in order to calculate the shift in the corresponding centroid. This shift can be determined by computing the centroid of $\mathcal{M}^{retained}(t_1, t_2)$, and comparing it to $\mathcal{S}(t_1)$.

The process of finding transient clusters is slightly more complicated. In this case, we find the micro-clusters at each snapshot between t_1 and t_2. Let us denote this set by $\mathcal{U}(t_1, t_2)$. This is the *universal* set of micro-clusters between t_1 and t_2. We also find the micro-clusters which are present at either t_1 or t_2. This set is essentially equivalent to $\mathcal{N}(t_1, t_1) \cup \mathcal{N}(t_2, t_2)$. Any micro-cluster which is present in $\mathcal{U}(t_1, t_2)$, but which is not present in $\mathcal{N}(t_1, t_1) \cup \mathcal{N}(t_2, t_2)$. In other words, we report the set $\mathcal{U}(t_1, t_2) - \mathcal{N}(t_1, t_1) - \mathcal{N}(t_2, t_2)$ as the final result.

We also tested the micro-clustering method for evolution analysis on the network intrusion data set in [3]. This data set was obtained from the UCI machine learning repository [27]. First, by comparing the data distribution for $t_1 = 29, t_2 = 30, h = 1$ CluStream found 3 micro-clusters (8 points) in $\mathcal{M}^{added}(t_1, t_2)$, 1 micro-cluster (1 point) in $\mathcal{M}^{deleted}(t_1, t_2)$, and 22 micro-clusters (192 points) in $\mathcal{M}^{retained}(t_1, t_2)$. This shows that only 0.5% of all the connections in $(28, 29)$ disappeared and only 4% were added in $(29, 30)$. By checking the original data set, we find that all points in $\mathcal{M}^{added}(t_1, t_2)$ and $\mathcal{M}^{deleted}(t_1, t_2)$ are normal connections, but are outliers because of some particular feature such as the number of bytes of data transmitted. The fact that almost all the points in this case belong to $\mathcal{M}^{retained}(t_1, t_2)$ indicates that the data distributions in these two windows are very similar. This happens because there are no attacks in this time period.

More interestingly, the data points falling into $\mathcal{M}^{added}(t_1, t_2)$ or $\mathcal{M}^{deleted}(t_1, t_2)$ are those which have evolved significantly. These usually correspond to newly arrived or faded attacks respectively. Here are two examples: (1) During the period $(34, 35)$, all data points correspond to normal connections, whereas during $(39, 40)$ all data points belong to smurf attacks. By applying our change analysis procedure for $t_1 = 35, t_2 = 40, h = 1$, it shows that 99% of the smurf connections (i.e., 198

connections) fall into two $\mathcal{M}^{added}(t_1, t_2)$ micro-clusters, and 99% of the normal connections fall into 21 $M^{deleted}(t_1, t_2)$ micro-clusters. This means these normal connections are non-existent during $(39, 40)$; (2) By applying the change analysis procedure for $t_1 = 640, t_2 = 1280, h = 16$, we found that all the data points during $(1264, 1280)$ belong to one $\mathcal{M}^{added}(t_1, t_2)$ micro-cluster, and all the data points in $(624, 640)$ belong to one $\mathcal{M}^{deleted}(t_1, t_2)$ micro-cluster. By checking the original labeled data set, we found that all the connections during $(1264, 1280)$ are smurf attacks and all the connections during $(624, 640)$ are neptune attacks. In general, the micro-clustering method may be used to perform concise summarization and evolution analysis of data streams. This methodology can also be extended to categorical data. We discuss this method in the next section.

3 Change Detection for Text and Categorical Data

The problem of change detection for text data has been studied in [22], whereas some techniques for summarization of categorical data have been proposed in [16]. In this paper, we will use summarization as a tool for comprehensive change detection of text and categorical data. As in the case of quantitative data, we need to maintain a statistical representation of the summary structure of the data. We will refer to such groups as *cluster droplets*. This is analogous to the summary cluster feature statistics or micro-cluster statistics stored in [3,26]. We will discuss and define the cluster droplet differently for the case of text and categorical data streams respectively. For generality, we assume that a weight is associated with each data point. In some applications, this may be desirable, when different data points have different levels of importance, as in th ecase of a temporal fade function. First, we will define the cluster droplet for the categorical data domain:

Definition 2. *A cluster droplet $\mathcal{D}(t, \mathcal{C})$ for a set of categorical data points $\mathcal{C}$ at time t is referred to as a tuple $(\overline{DF2}, \overline{DF1}, n, w(t), l)$, in which each tuple component is defined as follows:*

- *The vector $\overline{DF2}$ contains $\sum_{i \in \{1 \ldots d\}, j \in \{1, \ldots d\}, i \neq j} v_i \cdot v_j$ entries. For each pair of dimensions, we maintain $v_i \cdot v_j$ values. We note that v_i is number of possible categorical values of dimension i and v_j is the number of possible values of dimension j. Thus, for each of the $v_i \cdot v_j$ categorical value pairs i and j, we maintain the (weighted) counts of the number of points for each value pair which are included in cluster $\mathcal{C}$. In other words, for every possible pair of categorical values of dimensions i and j, we maintain the weighted number of points in the cluster in which these values co-occur.*
- *The vector $\overline{DF1}$ contains $\sum_{i=1}^{d} v_i$ entries. For each i, we maintain a weighted count of each of the v_i possible values of categorical attribute i occurring in the cluster.*
- *The entry n contains the number of data points in the cluster.*
- *The entry $w(t)$ contains the sum of the weights of the data points at time t. We note that the value $w(t)$ is a function of the time t and decays with time unless new data points are added to the droplet $\mathcal{D}(t)$.*

– *The entry l contains the time stamp of the last time that a data point was added to the cluster.*

We note that the above definition of a droplet assumes a data set in which each categorical attribute assumes a small number of possible values. (Thus, the value of v_i for each dimension i is relatively small.) However, in many cases, the data might actually be somewhat sparse. In such cases, the values of v_i could be relatively large. In those instances, we use a sparse representation. Specifically, for each pair of dimensions i and j, we maintain a list of the categorical value pairs which have *non-zero* counts. In a second list, we store the actual counts of these pairs. In many cases, this results in considerable savings of storage space. For example, consider the dimension pairs i and j, which contain v_i and v_j possible categorical values. Also, let us consider the case when $b_i \leq v_i$ and $b_j \leq v_j$ of them have non-zero presence in the droplet. Thus, at most $b_i \cdot b_j$ categorical attribute pairs will co-occur in the points in the cluster. We maintain a list of these (at most) $b_{ij} < b_i \cdot b_j$ value pairs along with the corresponding counts. This requires a storage of $3 \cdot b_{ij}$ values. (Two entries are required for the identities of the value pairs and one is required for the count.) We note that if the number of distinct non-zero values b_i and b_j are substantially lower than the number of possible non-zero values v_i and v_j respectively, then it may be more economical to store $3 \cdot b_{ij}$ values instead of $v_i \cdot v_j$ entries. These correspond to the list of categorical values which have non-zero presence together with the corresponding weighted counts. Similarly, for the case of $\overline{DF1}$, we only need to maintain $2 \cdot b_i$ entries for each dimension i.

Next, we consider the case of the text data set which is an example of a *sparse numeric* data set. This is because most documents contain only a small fraction of the vocabulary with non-zero frequency. The only difference with the categorical data domain is the way in which the underlying cluster droplets are maintained.

Definition 3. *A cluster droplet $\mathcal{D}(t, \mathcal{C})$ for a set of text data points $\mathcal{C}$ at time t is defined to as a tuple $(\overline{DF2}, \overline{DF1}, n, w(t), l)$. Each tuple component is defined as follows:*

– *The vector $\overline{DF2}$ contains $3 \cdot wb \cdot (wb-1)/2$ entries. Here wb is the number of distinct words in the cluster $\mathcal{C}$. For each pair of dimensions, we maintain a list of the pairs of word ids with non-zero counts. We also maintained the sum of the weighted counts for such word pairs.*
– *The vector $\overline{DF1}$ contains $2 \cdot wb$ entries. We maintain the identities of the words with non-zero counts. In addition, we maintain the sum of the weighted counts for each word occurring in the cluster.*
– *The entry n contains the number of data points in the cluster.*
– *The entry $w(t)$ contains the sum of the weights of the data points at time t. We note that the value $w(t)$ is a function of the time t and decays with time unless new data points are added to the droplet $\mathcal{D}(t)$.*
– *The entry l contains the time stamp of the last time that a data point was added to the cluster.*

The concept of cluster droplet has some interesting properties that will be useful during the maintenance process. These properties relate to the additivity and decay behavior of the cluster droplet. As in the case of quantitative micro-clustering, it is possible to perform the cluster droplet maintenance using an incremental update algorithm. The methodology is examining the evolving clusters can also be extended to this case. Details are discussed in [5] with an example of its application to text and market basket data sets.

4 Online Community Evolution in Data Streams

In this section, we discuss the problem of detecting patterns of interaction among a set of entities in a stream environment. Examples of such entities could be a set of businesses which interact with one another, sets of co-authors in a dynamic bibliography data base, or it could be the hyperlinks from web pages. In each of these cases, the interaction among different entities can rapidly evolve over time. A convenient way to model the entity interaction relationships is to view them as graphs in which the nodes correspond to entities and the edges correspond to the interactions among the nodes. The weights on these edges represent the level of the interaction between the different participants. For example, in the case when the nodes represent interacting entities in a business environment, the weights on the edges among these entities could represent the volume of business transactions. A *community of interaction* is defined to be a set of entities with a high degree of interaction among the participants.

The problem of finding communities in dynamic and evolving graphs been discussed in [10,11,17,19,20,21,24,25]. Since most of the current techniques are designed for applications such as the web, they usually assume a *gradually evolving* model for the interaction. Such techniques are not very useful for a fast stream environment in which the entities and their underlying relationships may quickly evolve over time. In addition, it is important to provide a user the *exploratory capability* to query for communities over different time horizons. Since individual points in the data streams cannot be processed more than once, we propose a framework which separates out the *offline exploratory* algorithms from the online stream processing part. The online stream processing framework creates summaries of the data which can then be further processed for exploratory querying. Therefore, as in the case of micro-clustering, we focus on the use of an Online Analytical Processing (OLAP) approach for providing offline exploratory capabilities to users in performing *change detection* across communities of interest over different time horizons.

Some examples of exploratory queries in which a user may be interested are as follows:

(1) Find the communities with substantial increase in interaction level in the interval $(t - h, t)$. We refer to such communities as *expanding communities.*
(2) Find the communities with substantial decrease in interaction level in the interval $(t - h, t)$ We refer to such communities as *contracting communities.*

(3) Find the communities with the most stable interaction level in the interval $(t-h, t)$.

We note that the process of finding an emerging or contracting community needs to be carefully designed in order to normalize for the behavior of the community evolution over different time horizons. For example, consider a data stream in which two entities n_1 and n_2 share a high level of interaction in the period $(t-h, t)$. This alone does not mean that the interaction level between n_1 and n_2 is stable especially if these entities had a even higher level of interaction in the previous period $(t-2 \cdot h, t-h)$. Thus, a careful model needs to be constructed which tracks the behavior of the interaction graph over different time horizons in order to understand the nature of the change. In he next subsection we will discuss the process of online summarization of data streams, and leveraging this process for the purpose of data stream mining.

4.1 Online Summarization of Graphical Data Streams

In this subsection, we will discuss the overall interaction model among the different entities. We will also discuss the process of online summarization of the data stream. This interaction model is stored as a graph $G = (N, A)$, in which N denotes the set of nodes, and A denotes the set of edges. Each node $i \in N$ corresponds to an entity. The edge set A consists of edges (i, j), such that i and j are nodes drawn from N. Each edge (i, j) represents an interaction between the entities i and j. Each edge (i, j) also has a weight $w_{ij}(t)$ associated with it. This weight corresponds to the number of interactions between the entities i and j. For example, when the interaction model represents a bibliography database, the nodes could represent the authors and the weights on the edges could represent the number of publications on which the corresponding authors occur together as co-authors. As new publications are added to the database the corresponding weights on the individual edges are modified. It is also possible for new nodes to be added to the data as new authors are added to the original mix. In this particular example, the weight on each edge increases by one, each time a new co-authorship relation is added to the database. However, in many applications such as those involving business interaction, this weight added in each iteration can be arbitrary, and in some cases even negative.

In order to model the corresponding stream for this interaction model, we assume that a current graph $G(t) = (N(t), A(t))$ exists which represents the history of interactions at time t. At time $(t+1)$ new additions may occur to the graph $G(t)$. Subsequently, each new arrival to the stream contains two elements:

- An edge (i, j) corresponding to the two entities between whom the interaction has taken place.
- An *incremental* weight $\delta w_{ij}(t)$ illustrating the additional interaction which has taken place between entities i and j at time t.

We refer to the above pair of elements as representative of an *interaction event*. We note that the nodes i, j, or the edge (i, j) may not be present in $N(t)$ and

$A(t)$ respectively. In such a case, the node set $N(t)$ and edge set $A(t)$ need to be modified to construct $N(t+1)$ and $A(t+1)$ respectively. In the event that a given edge does not exist to begin with, the original weight of (i, j) in $G(t)$ is assumed to be zero. Also, in such a case, the value of the edge set $A(t+1)$ is augmented as follows:

$$A(t+1) = A(t) \cup \{(i, j)\} \tag{1}$$

In the event that either the nodes i or j are not present in $N(t)$, the corresponding node set needs to be augmented with the new node(s). Furthermore, the weight of the edge (i, j) needs to be modified. If the edge (i, j) is new, then the weight of edge (i, j) in $G(t+1)$ is set to δw_{ij}. Otherwise, we add the incremental weight δw_{ij} to the current weight of edge (i, j) in $G(t)$. Therefore, we ave:

$$w_{ij}(t+1) = w_{ij}(t) + \delta w_{ij}(t) \tag{2}$$

We assume that the set of interaction events received at time t are denoted by $\mathcal{E}(t)$. In each iteration, the stream maintenance algorithm adds the interaction events in $\mathcal{E}(t)$ to $G(t)$ in order to create $G(t+1)$. We refer to the process of adding the events in $\mathcal{E}(t)$ to $G(t)$ by the $\oplus$ operation. Therefore, we have:

$$G(t+1) = G(t) \oplus \mathcal{E}(t) \tag{3}$$

At each given moment in time, we maintain the current graph of interactions $G(t)$ in main memory. In addition, we periodically store the graph of interactions on disk. We note that the amount of disk storage available may often be limited. Therefore, it is desirable to store the graph of interactions in an efficient way during the course of the stream arrival. We will refer to each storage of the graph of interactions at a particular moment as a *frame*. Let us assume that the storage limitation for the number of frames is denoted by S. In this case, one possibility is to store the last S frames at uniform intervals of t'. The value of S is determined by the storage space available. However, this is not a very effective solution, since it means that a history of larger than $S \cdot t'$ cannot be recalled.

One solution to this problem is to recognize that frames which are more stale need not be stored at the same frequency as more recent frames. Let t_c be the current time, and t_{min} be the minimum granularity at which 0th tier snapshots are stored. We divide the set of S frames into $\theta = \log_2(t_c/t_{min})$ tiers. The ith tier contains snapshots which are separated by a distance of $t_{min} \cdot 2^{i-1}$. For each tier, we store the last S/θ frames. This ensures that the total storage requirement continues to be S. Whenever it is desirable to access the state of the interaction graph for the time t, we simply have to find the frame which is temporally closest to t. The graph from this temporally closest frame is utilized in order to approximate the interaction graph at time t. The tiered nature of the storage process ensures that it is possible to approximate recent frames to the same degree of (percentage) accuracy than less recent frames. While this means that the (absolute) approximation of stale frames is greater, this is quite satisfactory fir a number of real scenarios. We make the following observations:

Lemma 1. *Let h be a user-specified time window, and t_c be the current time. Then a snapshot exists at time t_s, such that $h/(1+\theta/S) \leq t_c - t_s \leq (1+\theta/S) \cdot h$.*

Proof. This is an extension of the result in [6]. The proof is similar.

In order to understand the effectiveness of this simple methodology, let us consider a simple example in which we store (a modest number of) $S = 100,000$ frames for a stream over 10 years, in which the minimum granularity of storage t_{min} is 1 second. We have intentionally chosen an extreme example (in terms of the time period of the stream) together with a modest storage capability in order to show the effectiveness of the approximation. In this case, the number of tiers is given by $\theta = \log_2(10 * 365 * 24 * 3600) \approx= 29$. By substituting in Lemma 1, we see that it is possible to find a snapshot which is between 99.97% and 100.03% of the user specified value.

In order to improve the efficiency of edge storage further, we need to recognize the fact that large portions of the graph continue to be identical over time. Therefore, it is inefficient for the stream generation process to store the entire graph on disk in each iteration. Rather, we store only incremental portions of the graph on the disk. Specifically, let us consider the storage of the graph $G(t)$ for the ith tier at time t. Let the last time at which an $(i+1)$th tier snapshot was stored be denoted by t'. (If no snapshot of tier $(i+1)$ exists, then the value of t' is 0. We assume that $G(0)$ is the null graph.) Then, we store the graph $F(t) = G(t) - G(t')$ at time t. We note that the graph $F(t)$ contains far fewer edges than the original graph $G(t)$. Therefore, it is more efficient to store $F(t)$ rather than $G(t)$. Another observation is that a snapshot for the ith tier can be reconstructed by summing the snapshots for all tiers larger than i.

Lemma 2. *We assume that the highest tier is defined by m. Let t_i be the time at which the snapshot for tier i is stored. Let t_{i+1}, t_{i+2} ... t_m be the last time stamps of tiers $(i+1) \ldots m$ (before t_i) at which the snapshots are stored. Then the current graph $G(t_i)$ at the time stamp t_i is defined as follows:*

$$G(t_i) = \sum_{j=i}^{m} F(t_j) \tag{4}$$

Proof. This result can be proved easily by induction. We note that the definition of $F(\cdot)$ implies that:

$$\begin{aligned} G(t_i) - G(t_{i+1}) &= F(t_i) \\ G(t_{i+1}) - G(t_{i+2}) &= F(t_{i+1}) \\ &\ldots \\ G(t_{m-1}) - G(t_m) &= F(t_{m-1}) \\ G(t_m) - 0 &= F(t_m) \end{aligned}$$

By summing the above equations, we obtain the desired result.

The above result implies that the graph at a snapshot for a particular tier can be reconstructed by summing it with the snapshots at higher tiers. Since there are at most $\theta = \log_2(S/t_{min})$ tiers, it implies that the graph at a given time can be reconstructed quite efficiently in a practical setting.

4.2 Offline Construction and Processing of Differential Graphs

In this subsection, we will discuss the offline process of generating differential graphs and their application to the evolution detection process. The differential graph is generated over a specific time horizon (t_1, t_2) over which the user would like to test the behavior of the data stream. The differential graph is defined over the interval (t_1, t_2) and is defined as a fraction of the interactions over that interval by which the level of interaction has changed during the interval (t_1, t_2). In order to generate the differential graph, we first construct the *normalized graph* at the times t_1 and t_2. The normalized graph $G(t) = (N(t), A(t))$ at time t is denoted by $\overline{G(t)}$, and contains exactly the same node and edge set, but with different weights. Let $W(t) = \sum_{(i,j) \in A} w_{ij}(t)$ be the sum of the weights over all edges in the graph $G(t)$. Then, the normalized weight $\overline{w_{ij}(t)}$ is defined as $w_{ij}(t)/W(t)$. We note that the normalized graph basically comprises the fraction of interactions over each edge.

Let t_1' be the last snapshot stored just before time t_1 and t_2' be the snapshot stored just before time t_2. The first step is to construct the graphs $G(t_1')$ and $G(t_2')$ at time periods t_1' and t_2' by adding the snapshots at the corresponding tiers as defined by Lemma 2. Then we construct the normalized graph from the graphs at times t_1' and t_2'. The differential graph is constructed from the normalized graph by subtracting out the corresponding edge weights in the original normalized graphs. Therefore, the differential graph $\Delta G(t_1', t_2')$ basically contains the same nodes and edges as $\overline{G(t_2')}$, except that the differential weight $\Delta w_{ij}(t_1', t_2')$ on the edge (i, j) is defined as follows:

$$\Delta w_{ij}(t_1', t_2') = \overline{w_{ij}(t_2')} - \overline{w_{ij}(t_1')} \tag{5}$$

In the event that an edge (i, j) does not exist in the graph $\overline{G(t_1')}$, the value of $w_{ij}(t_1')$ is assumed to be zero. We note that because of the normalization process, the differential weights on many of the edges may be negative. These correspond to edges over which the interaction has reduced significantly during the evolution process. For instance, in our example corresponding to a publication database, when the number of jointly authored publications reduces over time, the corresponding weights in the differential graph are also negative.

Once the differential graph has been constructed, we would like to find clusters of nodes which show a high level of evolution. It is a tricky issue to determine the subgraphs which have a high level of evolution. A natural solution would be find the clustered subgraphs with high weight edges. However, in a given subgraph, some of the edges may have high positive weight while others may have high negative weight. Therefore, such subgraphs correspond to the entity relationships with high evolution, but they do not necessarily correspond to entity relationships with the greatest increase or decrease in interaction level. In order to find the community of interaction with the greatest increase in interaction level, we need to find subgraphs such that most interactions within that subgraph have either a high positive or high negative weight. This is a much more difficult problem than the pure vanilla problem of finding clusters within the subgraph $\Delta G(t_1', t_2')$.

4.3 Finding Evolving Communities

In this section, we will define the algorithm for finding evolution clusters in the interaction graph based on the user defined horizon. The process of finding the most effective clusters is greatly complicated by the fact that some of the edges correspond to an increase in the evolution level, whereas other edges correspond to a decrease in the evolution level. The edges corresponding to an increase in interaction level are referred to as the positive edges, whereas those corresponding to a reduction in the interaction level are referred to as the negative edges.

We design an algorithm which can effectively find subgraphs of positive or negative edges by using a partitioning approach which tracks the positive and negative subgraphs in a dynamic way. In this approach, we use a set of seed nodes $\{n_1 \ldots n_k\}$ in order to create the clusters. Associated with each seed n_i is a partition of the data which is denoted by N_i. We also have a bias vector B which contains $|N|$ entries of $+1$, 0, or -1. The bias vector is an indicator of the nature of the edges in the corresponding cluster. The algorithm utilizes an iterative approach in which the clusters are constructed around these seed nodes. The overall algorithm is illustrated in Figure 1. As illustrated in Figure 1, we first pick the k seeds randomly. Next, we enter an iterative loop in which we refine the initial seeds by performing the following steps:

- We assign each nodes to one of the seeds. The process of assignment of an entity node to a given seed node is quite tricky because of the fact that we would like a given subgraph to represent either increasing or decreasing communities. Therefore, we need to choose effective algorithms which can compute the distances for each community in a different way. We will discuss this process in detail slightly later. We note that the process of assignment is sensitive to the nature of the *bias* in the node. The bias in the node could

Algorithm *FindEvolvingCommunities*(Graph: (N, A),
EdgeWeights: $\Delta w_{ij}(t_1, t_2)$, NumberOfClusters: k);
begin
Randomly sample nodes $n_1 \ldots n_k$ as seeds;
Let B be the bias vector of length $|N|$;
Set each position in B to 0;
while not(*termination_criterion*) **do**
begin
$(N_1 \ldots N_k)$ =*AssignNodes*(N, B, $\{n_1 \ldots n_k\}$);
B =*FindBias*$(N_1 \ldots N_k)$;
$(N_1' \ldots N_k')$ = *RemoveNodes*$(N_1 \ldots N_k)$;
{ Assume that the removed nodes are null partitions }
$(n_1 \ldots n_k)$ =*RecenterNodes*$(N_1 \ldots N_k)$;
end
end

Fig. 1. Finding Evolving Communities

represent the fact that the cluster seeded by that node is likely to become one of the following: (1) An Expanding Community (2) A Contracting Community (3) Neutral Bias (Initial State). Therefore, we associate a *bias bit* with each seed node. This bias bit takes on the values of $+1$, or -1, depending upon whether the node has the tendency to belong to an expanding or contracting community. In the event that the bias is neutral, the value of that bit is set to 0. We note that an expanding community corresponds to positive edges, whereas a contracting community corresponds to negative edges. The process of assignment results in k node sets which are denoted by $N_1 \ldots N_k$. In the assignment process,we compute the distance of each seed node to the different entities. Each entity is assigned to its closest seed node. The algorithm for assignment of entities to seed nodes is denoted by *AssignNodes* in Figure 1.

- Once the assignment step has been performed, we re-assess the bias of that seed node. This is denoted by *FindBias* in Figure 1. The overall approach in finding the bias is to determine whether the interactions in the community attached to that seed node represent expansion or contraction. We will discuss the details of this algorithm slightly later. The bias bit vector B is returned by this procedure.
- Some of the seed nodes may not represent a coherent community of interaction. These seed nodes may be removed by the community detection algorithm. This process is achieved by the algorithm *RemoveNodes*. The new set of nodes is denoted by $N_1' \ldots N_k'$. We note that each N_i' is either N_i or null depending upon whether or not that node was removed by the algorithm.
- The final step is to re-center the seeds within their particular subgraph. The re-centering process essentially reassigns the seed node in a given subgraph N_i' to a more central point in it. In the event that N_i' is a null partition, the recentering process simply picks a random node in the graph as the corresponding seed. Once the recentering process is performed, we can perform the next iteration of the algorithm. The recentering procedure is denoted by *RecenterNodes* in Figure 1. The new set of seeds $n_1 \ldots n_k$ are returned by this algorithm.

4.4 Subroutines for Determination of the Communities

In the afore-mentioned discussion, we described the overall procedure for finding the communities of interaction. In this section, we will discuss the details of the subroutines which are required for determination of these communities. We will first discuss the procedure *AssignNodes* of Figure 1. The process of assigning the nodes to each of the centroids requires the use of the bias information stored in the bias nodes. Note that if a seed node has positive bias, then it needs to be ensured that the other nodes which are assigned to this seed node are related to it by positive interactions. The opposite is true if the bias on the seed node is negative. Finally, in the case of nodes with neutral bias, we simply need to find a path with high absolute interaction level. In this case, the positivity or negativity of the sign matters less than the corresponding absolute value. In order to achieve

this goal, we define a *bias sensitive* distance function $f(n_1, n_2, b)$ between two nodes n_1 and n_2 for the bias bit b. For a given path P in the graph $\Delta G(t_1, t_2)$, we define the average interaction level $w(P)$ as the sum of the interaction levels on P divided by the number of edges on P. Therefore, we have:

$$w(P) = \sum_{(i,j) \in P} \Delta w_{ij}(t_1, t_2)/|P| \tag{6}$$

We note that a path with high average *positive* weight corresponds to a set of edges with increasing level of interaction. This can also be considered an expanding community. The opposite is true of a path with high average *negative* weight, which corresponds to a contracting community. Therefore, the value of $w(P)$ is equal to the weight of the path divided by the number of edges on that path. We also define the average *absolute* average interaction level $w^+(P)$ as follows:

$$w^+(P) = \sum_{(i,j) \in P} |\Delta w_{ij}(t_1, t_2)|/|P| \tag{7}$$

Note that the absolute interaction level does not have any bias towards a positive or negative level of interaction. Once we have set up the definitions of the path weights, we can also define the value of the interaction function $f(n_1, n_2, b)$. This interaction function is defined over all possible pairs of nodes (n_1, n_2).

$$\begin{aligned} f(n_1, n_2, b) = & \text{ Most positive value of } w(P) \ \forall \text{ paths} \\ & P \text{ between } n_1 \text{ and } n_2 \qquad \text{if } b = 1 \\ & \text{Modulus of most negative value of } w(P) \\ & \forall \ P \text{ between } n_1 \text{ and } n_2 \qquad \text{if } b = -1 \\ & \text{Largest value of } w^+(P) \ \forall \text{ paths} \\ & P \text{ between } n_1 \text{ and } n_2 \qquad \text{if } b = 0 \end{aligned}$$

We note that the above interaction function is defined on the basis of the sum of the interaction values over a given path. In some cases, this interaction function can provide skewed results when the path lengths are long. This could result in less effective partitioning of the communities. A different interaction function is defined as the minimum interaction on the path between two entities. However, the bias of the corresponding centroid on that path is used in order to define the interaction function. This minimum interaction $w(P)$ is defined as follows:

$$\begin{aligned} w(P) = & \min_{(i,j) \in P} \max\{\Delta w_{ij}(t_1, t_2), 0\} \\ & \qquad \text{if } b = 1 \\ & \max_{(i,j) \in P} \min\{\Delta w_{ij}(t_1, t_2), 0\} \\ & \qquad \text{if } b = -1 \\ & \min_{(i,j) \in P} |\Delta w_{ij}(t_1, t_2)| \\ & \qquad \text{if } b = 0 \end{aligned}$$

We note that the above mentioned function simply finds the minimum (absolute) weight edge of the corresponding sign (depending on the bias) between the two nodes. The corresponding interaction function $f(n_1, n_2, b)$ is defined in the same way as earlier. Henceforth, we will refer to the two above-defined functions as the average-interaction function and minimum interaction function respectively. In the latter case, the interaction distance corresponds to the interaction on the weakest link between the two nodes. As our experimental results will show, we found the minimum function to be slightly more robust than the average interaction function.

During the assignment phase, we calculate the value of the function $f(n_i, n, b)$ from each node n_i to the seed node n using the bias bit b. Each node n_i is assigned to the seed node n with the largest *absolute* value of the above-mentioned function. This process ensures that nodes are assigned to seeds according to their corresponding bias. The process of computation of the interaction function will be discussed in some detail slightly later.

Next, we determine the bias of each seed node in the procedure *FindBias*. In order to do so, we calculate the bias-index of the community defined by that seed node. The bias index of the community N_i is denoted by $\mathcal{I}(N_i)$, and is defined as the edge-weight fraction of the expanding portion of N_i. In order to do, so we divide the positive edge weights in the community by the total absolute edge weight in the same community. Therefore, we have:

$$\mathcal{I}(N_i) = \frac{\sum_{(p,q)\in N_i} \max\{0, \Delta w_{pq}(t_1, t_2)\}}{\sum_{(p,q)\in N_i} |\Delta w_{pq}(t_1, t_2)|} \tag{8}$$

We note that the bias index is 1 when all the edges in the community corresponding to increasing interaction, and is 0 when all the edges correspond to reducing interaction. Therefore, we define a threshold $t \in (0, 0.5)$. If the value of $\mathcal{I}(N_i)$ is less than t then the bias bit is set to -1. Similarly, if the value of $\mathcal{I}(N_i)$ is larger than $1 - t$, the bias bit is set to 1. Otherwise, the bias bit is set to zero.

Once the bias bits for the nodes have been set, we remove those seeds which have very few nodes associated with them. Such nodes usually do not correspond to a coherent community of interaction. This procedure is referred to as *RemoveNodes*. Thus, each set of nodes N_i is replaced by either itself or a null set of nodes. In order to implement this step, we use a minimum threshold on the number of nodes in a given partition. This threshold is denoted by mn_t. All partitions with less than mn_t entities are removed from consideration, and replaced by the null set.

The last step is to recenter the nodes within their corresponding partition. This denoted by the procedure *RecenterNodes* in Figure 1. The process of recentering the nodes requires us to use a process in which the central points of subgraphs are determined. In order to recenter the nodes, we determine the node which minimizes the maximum distance of any node in the cluster. This is achieved by computing the distance of all points in the cluster starting at each node, and finding the minimax distance over these different values. The process

of recentering helps to adjust the centers of the nodes in each iteration such that the process of partitioning the community sets becomes more effective over time.

4.5 Approximating Interaction Distances Among Nodes

The only remaining issue is to discuss the methodology for determining the interaction distances among nodes. We would like our algorithm to be general enough to find the maximum interaction distance for general functions. It is important to understand that the problem of finding the maximum interaction distance between two nodes is NP-hard.

Observation 4.1. *The problem of determining the maximum interaction distance between two nodes is NP-hard for arbitrary interaction functions.*

This observation is easily verified by observing that the problem of finding the longest path in a graph is NP-hard [7]. Since the particular case of picking the interaction function as the edge weight is NP-hard, the general problem is NP-hard as well.

However, it is possible to approximate the interaction distance of a maximum length using dynamic programming. Let wn_{ij}^t be the maximum interaction distance between two nodes using at most t nodes on that path. Let P_{ik}^t be the maximum length path between i and k using t edges. Let $PS_{ikj}^{\oplus}$ denote the path obtained by concatenating P_{ik}^t with the edge (k, j). Then, we define wn_{ij}^t recursively as follows:

$$wn_{ij}^0 = 0;$$
$$wn_{ij}^{t+1} = \max_k \{wn_{ij}^t, w(PS_{ikj}^{\oplus})\}$$

We note that this dynamic programming algorithm does not always lead to an optimal solution in the presence of cycles in the graph [7]. However, for small values of t, it approximates the optimal solution well. This is because cycles are less likely to be present in paths of smaller length. It is also important to understand that if two entities are joined only by paths containing a large number of edges, then such pairs of entities should not be regarded as belonging to the same community. Therefore, we imposed a threshold *maxthresh* on the maximum length of the (shortest interaction distance) path between two nodes for them to belong to the same community. For a pair of nodes in which the corresponding path length was exceeded, the value of the interaction distance is set to 0. In [4], we have tested the method extensively over a number of real and synthetic data sets. These results show that the technique is scalable and provides insights about significantly evolving communities in the data stream.

5 Conclusions and Summary

In this paper, we examined the application of clustering for change diagnosis and detection in data streams. We showed how clustering can be used in order to determine summaries of the underlying change in the data. The overall

framework is motivated by the micro-clustering technique [3], which provides a summary of the data for future change diagnosis. This summary can be used for text and categorical data, as well as community evolution in data streams. Such information has considerable value in a number of applications which require a clear understanding of the underlying data.

References

1. Aggarwal C. C.: A Framework for Diagnosing Changes in Evolving Data Streams, ACM SIGMOD Conference, (2003) 575–586.
2. Aggarwal C. C.: An Intuitive Framework for Understanding Changes in Evolving Data Streams, ICDE Conference, (2002).
3. Aggarwal C. C., Han J., Wang J., Yu P.: A Framework for Clustering Evolving Data Streams, VLDB Conference, (2003) 81–92.
4. Aggarwal C. C., Yu P. S.: Online Analysis of Community Evolution in Data Streams. *ACM SIAM Data Mining Conference*, (2006).
5. Aggarwal C. C., Yu P. S.: A Framework for Clustering Massive Text and Categorical Data Streams. *ACM SIAM Data Mining Conference*, (2006).
6. Aggarwal C, Han J., Wang J., Yu P.: On-Demand Classification of Data Streams. *ACM KDD Conference*, (2004).
7. Ahuja R., Magnanti T., and Orlin J.: *Network Flows: Theory, Algorithms and Applications*, Prentice Hall, Englewood Cliffs, New Jersey, (1992).
8. Babcock B., Babu S., Datar M., Motwani R., Widom J.: Models and Issues in Data Stream Systems, *ACM PODS Conference*, (2002) 1–16.
9. Chawathe S., and Garcia-Molina H.: Meaningful Change Detection in Structured Data. *ACM SIGMOD Conference Proceedings* (1997).
10. Cortes C., Pregibon D., and Volinsky C.: Communities of Interest, *Proceedings of Intelligent Data Analysis*, (2001).
11. Cortes C., Pregibon D., and Volinsky C.: Computational Methods for Dynamic Graphs, *Journal of Computational and Graphical Statistics*, 12, (2003), pp. 950-970.
12. Dasu T., Krishnan S., Venkatasubramaniam S., Yi K.: An Information-Theoretic Approach to Detecting Changes in Multi-dimensional data Streams. *Duke University Technical Report CS-2005-06* (2005).
13. Domingos P., and Hulten G.: Mining High-Speed Data Streams, *ACM SIGKDD Conference*, (2000).
14. Ganti V., Gehrke J., Ramakrishnan R.: A Framework for Measuring Changes in Data Characteristics. *ACM PODS Conference* (1999) pp. 126-137.
15. Ganti V., Gehrke J., Ramakrishnan R.: Mining and Monitoring Evolving Data. *IEEE ICDE Conference* (2000) pp. 439–448.
16. Ganti V., Gehrke J., Ramakrishnan R.: CACTUS- Clustering Categorical Data Using Summaries. *ACM KDD Conference* (1999) pp. 73–83.
17. Gibson D., Kleinberg J., and Raghavan P.: Inferring Web Communities from Link Topology, *Proceedings of the 9th ACM Conference on Hypertext and Hypermedia*, (1998).
18. Hulten G., Spencer L., and Domingos P.: Mining Time Changing Data Streams, *ACM KDD Conference*, (2001).
19. Imafuji N., and Kitsuregawa M.: Finding a Web Community by Maximum Flow Algorithm with HITS Score Based Capacity, *DASFAA*, (2003), pp. 101–106.

20. Kempe D., Kleinberg J., and Tardos E.: Maximizing the Spread of Influence Through a Social Network, *ACM KDD Conference*, (2003).
21. Kumar R., Novak J., Raghavan P., and Tomkins A.: On the Bursty Evolution of Blogspace, *Proceedings of the WWW Conference*, (2003).
22. Mei Q., Zhai C.: Discovering evolutionary theme patterns from text: an exploration of temporal text mining. *ACM KDD Conference*, (2005) pp. 198-207.
23. Nasraoui O., Cardona C., Rojas C., Gonzlez F.: TECNO-STREAMS: Tracking Evolving Clusters in Noisy Data Streams with a Scalable Immune System Learning Model, *ICDM Conference*, (2003) pp. 235-242.
24. Rajagopalan S., Kumar R., Raghavan P., and Tomkins A.: Trawling the Web for emerging cyber-communities, *Proceedings of the 8th WWW conference*, (1999).
25. Toyoda M., and Kitsuregawa M.: Extracting evolution of web communities from a series of web archives, *Hypertext*, (2003) pp. 28–37.
26. Zhang T., Ramakrishnan R., Livny M.: BIRCH: An Efficient Data Clustering Method for Very Large Databases. *ACM SIGMOD Conference*, (1996), pp. 103–114.
27. http://www.ics.uci.edu/∼mlearn.