Available online at www.sciencedirect.com
SCIENCE DIRECT®
Information Sciences 176 (2006) 2016–2041
INFORMATION SCIENCES
AN INTERNATIONAL JOURNAL
www.elsevier.com/locate/ins

Streaming data reduction using low-memory factored representations

David Littau *, Daniel Boley

Department of Computer Science and Engineering, University of Minnesota Twin Cities, 200 Union Street, SE, Minneapolis, MN 55455, United States

Abstract

Many special purpose algorithms exist for extracting information from streaming data. Constraints are imposed on the total memory and on the average processing time per data item. These constraints are usually satisfied by deciding in advance the kind of information one wishes to extract, and then extracting only the data relevant for that goal. Here, we propose a general data representation that can be computed using modest memory requirements with limited processing power per data item, and yet permits the application of an arbitrary data mining algorithm chosen and/or adjusted after the data collection process has begun. The new representation allows for the at-once analysis of a significantly larger number of data items than would be possible using the original representation of the data. The method depends on a rapid computation of a factored form of the original data set. The method is illustrated with two real datasets, one with dense and one with sparse attribute values.

Keywords: Streaming data; Data reduction; Clustering; PDDP; Matrix approximation

* Corresponding author.
E-mail addresses: littau@cs.umn.edu (D. Littau), boley@cs.umn.edu (D. Boley).

0020-0255/$ - see front matter
doi:10.1016/j.ins.2005.11.002

1. Introduction

As data sets have become larger and often unbounded, the concept of a data stream has become a useful model in data mining applications. The data stream model is applicable to data sets in which new data arrive constantly, such as network connection data, credit card transaction data, etc. Such data are usually examined once and either archived or deleted. The data stream model is also applicable to data sets which are so large that they cannot fit into memory at once. If the data cannot fit into memory it becomes expensive to apply data mining algorithms based on the static data model, since such algorithms usually scan the data several times.

In the data stream model [13], data points can only be accessed in the order in which they appear in the stream. Random access of data is not allowed. The amount of memory available is severely limited, much less than what would be required to store all the data at once. Working under these constraints, the goal of our streaming data mining pre-processing method is to extract as much useful information as possible from the data in a reasonable amount of time, while not fixing the task to be performed on the data in advance.

The usual approach in streaming data mining applications [1,2,8,10,12,16,15,21] is to first decide what data mining task is to be performed, and then tailor the processing to gather the information necessary for the specific task. This is an efficient approach to mining stream data. It allows the data to be collapsed into some very tiny, efficient representation whose memory footprint is limited to some constant size. The drawback to this technique is that if another task needs to be completed, it is often necessary to process the data again to gather the information for the different task. This can become very expensive very quickly.

One way to maintain the flexibility with respect to the task is to apply some kind of approximation technique. Representations of the data are constructed which are designed to reflect the most important qualities of the data while taking up less room than the original data, such as in the summaries used in [6,24]. Most representations of the data assign many data points to one representative, which means that individual data points are no longer distinguishable.

We present a pre-processing method for streaming data which does not require the data mining task to be selected beforehand. The data are represented in the vector space model. We create a low-memory factored representation (LMFR) of the data, such that each data point has a unique representation in the factored form. We accumulate data from the stream in the form of chunks we call *sections*. Once a given section of data has been processed it does not need to be accessed again. The low-memory representation allows for the at-once mining of a much larger piece of data than would be possible when using the original representation of the data. Any tasks which require the data points to be distinct, such as searching for outliers or

identifying individual data samples satisfying some new query (not known in advance), will function with the low-memory representation. If the desire is to perform a series of tasks such that the subsequent tasks depend on the results of previous tasks, the LMFR provides a general representation such that it would not be necessary to start over in processing the data stream, as may be required using other streaming data mining methods. Also, since the data in the low-memory representation remain ordered in the same manner as they arrived from the stream, it is simple to keep track of any time dependencies which are present. Note that while the LMFR allows for a representation of a significantly larger amount of data to be present in memory at once, an LMFR of an infinite data stream cannot be contained in memory.

This paper is organized as follows. First, we give a brief survey of previous techniques used to construct approximations to the data which consume less memory than the original representation of the data. Next, we give a description of the method used to construct the LMFR, along with the appropriate cost analysis. Then, we show some experimental results using the LMFR in the context of a clustering with the principal direction divisive partitioning (PDDP) [5] clustering method. The results indicate that the LMFR represents the data sufficiently to obtain a PDDP clustering comparable to the PDDP clustering of the original representation of the data.

2. Background

There are many streaming data mining tasks which can be performed using the LMFR. The works in [1,8,16] describe different tasks performed over a moving window. Using the LMFR in place of the original representation of the data, the window could be much larger. Babu and Widom [2] discussed processing continuous queries over the data stream, and noted that if the input data stream contains updates, a large portion of the input stream may need to be stored. To compress the data stream when storage is required, they suggest the use of *summaries*. The summaries are compact, but they tend to lose information about the data ordering, and make it impossible to distinguish the original data items. The LMFR provides a compact representation of the data in which individual data items have a unique representation. The problem of clustering data streams is addressed in [21]. Using the LMFR would allow for more stream data to be exposed to the algorithm at a given time.

The LMFR proposed in this paper is intended to be used in general streaming data mining applications. Since the LMFR is a low-memory approximation to the original data, we now discuss several previous efforts to compute alternate representations of the dataset that occupy less space.

The creation of summaries is a common way to compress data. The summaries used in [6,24] record the number of data items, the vector sum of the data items,

and the vector sum of squares of the data items represented by the summary. Using this information, it is possible to obtain the centroid and the diagonal of the covariance matrix of the data items represented by the summary. The summary is also easy to update. When a new vector is introduced, it is folded in with the closest summary as long as it is within a threshold distance of an existing summary. Otherwise, the new vector can be used to create a new summary, or if it appears to be an outlier, it can be eliminated from consideration.

Other summary construction methods are available, such as those using wavelets [7,11], histograms [17], or a combination of the two [20]. The point we wish to emphasis about summaries is that they assign more than one data item to a given representative of the data.

While the various summaries are a compact way to represent many data items, they have a few drawbacks. Once a data item is folded into a summary, it is no longer possible to distinguish that data item from any other data items. If the desire is to use the summaries for clustering, the summaries represent the smallest partitioning of the data that can be created. Also, the summaries lose a lot of information about the ordering of the data, which means a data mining application which depends on ordering may not be able to use the summaries in place of the original representation of the data. The various types of summaries were proposed for specific applications, so the number of post-processing algorithms which could be applied to a given summary is limited.

When it is necessary to have a unique representation of each data item, factoring techniques are often used. Two factoring methods which have been designed to save memory are the semi-discrete decomposition (SDD) [18] and the sparse SVD method developed in [25]. Both of these methods require multiple passes over the data, but we mention them because our factored representation can be used for subsequent data mining tasks in the same ways as these methods can. Each method was designed to find an alternative factorization to the singular value decomposition (SVD) that occupies less memory than the SVD and can take the place of the SVD when performing tasks such as latent semantic indexing (LSI) [3]. Both methods reduce the residual approximation error for each new pair of columns and the corresponding diagonal entry added to the factorization which takes the form

$$\mathbf{A}_k = \mathbf{X}_k \mathbf{D}_k \mathbf{Y}_k^{\mathrm{T}}, \tag{1}$$

though the exact terminology used in each work is different.

The SDD saves memory by limiting the row entries in $\mathbf{X}_k$ and $\mathbf{Y}_k$ to being either 1, 0, or -1. The method in [25] saves memory by restricting the number of non-zero entries in $\mathbf{X}_k$ and $\mathbf{Y}_k$. Both methods can be as expensive to compute as the standard SVD, but depending on the approach taken, the SDD can be less expensive than the SVD.

A third factored representation is the *concept decomposition* [9]. This decomposition requires a clustering of the entire dataset. Each original data point is

represented with a least-squares approximation using the centroids from the clustering. Rather than using an approach which reduces residual error, approximation accuracy is increased by computing more clusters. The method was developed as a faster alternative to the SVD for performing LSI. Our LMFR method was motivated in large part by the concept decomposition.

The LMFR method we present is less expensive to compute than the SVD, and it is simple to update the LMFR to incorporate new data as it arrives. Each data item has a unique representation in the LMFR, unlike the case when using summaries. In addition, the LMFR is constructed in a manner such that the order in which the data arrive in the stream is preserved, while the summaries lose most ordering information. Most data mining algorithms should function when replacing the original representation of the data with the LMFR. In its simplest form, the LMFR method can be thought of as a piecemeal concept decomposition with a sparse matrix of data loadings, glued together to form a complete decomposition. To this basic algorithm, we add a modest amount of additional processing to the individual matrix factors to keep the memory requirements as low as possible.

3. Constructing the low-memory factored representation

In this section, we describe how we construct the low-memory factored representation (LMFR) of the data. The LMFR is constructed in an incremental fashion. We accumulate the data as it arrives from the stream to form a section. We construct an LMFR of that section of data, after which we can delete the data comprising the section from memory. The LMFR of the section is then added to the global LMFR of all data seen previously. Since the global LMFR contains a unique representation for every data point, it is possible to apply any data mining algorithm to the LMFR that could have been applied to the original representation of the data.

We use the vector space model when constructing the LMFR. In the vector space model, each data item is defined as a column vector comprised of its attributes. One intuitive example of the vector space model is the case in which a document is converted to vector form. Each document i is represented by a column vector $\mathbf{x}_i$ consisting of word counts or frequencies. We collect all these columns vectors into a single $n \times \bar{m}$ matrix $\mathbf{X}$, where n is the number of attributes and $\bar{m}$ is the number of documents. If word j appears in the ith document l times, then $\mathbf{X}_{ij} = l$ (before it is scaled). For other kinds of data a given attribute is different. For example, an attribute of astronomical data sample could be position, brightness, the red-shift value, etc. The important point is that each attribute, be it a word or measurement, has a unique row assigned to it in the column vector representing the data item, and that each attribute has a numerical value. If the data are sparse, only non-zero entries need to be stored.

This section proceeds as follows. First, since we use the PDDP clustering method during the construction of the LMFR, we briefly describe PDDP. Then, we describe how to compute the LMFR for one section of data, followed by a description on how to construct the global LMFR reflecting all the data seen so far from the data stream. Finally, we describe various techniques which can be used to reduce the size of the global LMFR after it has been constructed.

3.1. PDDP

Principal direction divisive partitioning (PDDP) [5] is a top-down hierarchical clustering method for static data sets. PDDP builds a binary tree by recursively splitting clusters. The process starts with the root node, which contains all the data being clustered. To start, the root node is split into two child clusters. Then, the child cluster with the largest scatter (25) is split next. The process continues until the desired number of clusters is produced, or an automatic stopping criteria [4] is met.

A cluster is split using a hyperplane to define the two halves of the cluster, with all data on one side of the hyperplane being in one cluster, and all data on the other side of the hyperplane being in the other cluster. Ties are arbitrarily assigned to one side. The hyperplane is defined as being normal to the leading principal component of the data in the cluster, which is computed using a rank 1 singular value decomposition of the data in the cluster. The hyperplane is anchored at the projection of the mean value of the data onto the leading principal vector.

The PDDP clustering method is fast and scalable. While an in-depth complexity analysis of PDDP is beyond the scope of this paper, it has been determined that the complexity of PDDP is $O(mn \log(k_f))$, where m is the number of data items, n is the number of attributes per data item, and k_f is the number of clusters produced.

3.2. Factoring one section of data

We now describe how to construct a LMFR for one section of data. This construction only depends on data in the section. We assume that the section of data is small enough to fit into memory while allowing sufficient space for overhead computations.

Assume a given data item has n attributes. We accumulate m data items from the stream to form a section. The parameter m is chosen arbitrarily depending on the memory available. We use this section of data items to construct the $n \times m$ matrix $\mathbf{A}$. Our goal to construct an LMFR of $\mathbf{A}$

$$\mathbf{A} \approx \mathbf{CZ}, \tag{2}$$

where $\mathbf{C}$ is a $n \times k_c$ matrix of representative vectors and $\mathbf{Z}$ is a $k_c \times m$ matrix of data loadings. Each column $\mathbf{z}_i$ of $\mathbf{Z}$ approximates the corresponding column $\mathbf{a}_i$

of **A** using a linear combination of the vectors in **C**. A sparsity condition is enforced on **Z** to obtain the memory savings.

The first step in computing this factored form of **A** is to obtain the matrix of representative vectors **C**. We accomplish this task by clustering the data in **A**. We partition **A** into k_c clusters and compute the k_c centroids of the clusters. These centroids are collected into a $n \times k_c$ matrix **C**

$$\mathbf{C} = [\mathbf{c}_1 \mathbf{c}_2 \dots \mathbf{c}_{k_c}], \tag{3}$$

where the $\mathbf{c}_i$ are the centroids of the clusters. We use the PDDP [5] method to compute the clustering of **A** and hence **C** because PDDP is fast, but in principle, any clustering method could be used to obtain the components of **C**. For example, k-means could be used instead, but we preferred PDDP because, unlike k-means, PDDP does not depend on how the iteration is started [22]. Furthermore, if we use a clustering method that automatically adapts the number of clusters to the data, such as the PDDP method with its automated stopping test [4], we can gain an advantage for certain types of data streams. If many of the data in a section are either similar or the same, an adaptable technique is likely to produce fewer representative vectors, saving more memory than might occur when manually fixing the value of k_c. Once we have obtained the clusters and computed the centroids, the PDDP tree is discarded.

The matrix of data loadings **Z** is computed one column at a time. In approximating each column $\mathbf{a}_i$, we use only a small number (k_z) of the representatives in **C** in order to save memory. Therefore, each column $\mathbf{z}_i$ in **Z** has only k_z non-zero entries. For example, to approximate $\mathbf{a}_i$, we choose the k_z columns in **C** which are closest in Euclidean distance to $\mathbf{a}_i$ and implicitly collect them into an $n \times k_z$ matrix $\mathbf{C}_i$. Then the non-zero entries in the column $\mathbf{z}_i$ are obtained by solving for the k_z-vector $\hat{\mathbf{z}}_i$

$$\hat{\mathbf{z}}_i = \arg\min_{\mathbf{z}} ||\mathbf{a}_i - \mathbf{C}_i \mathbf{z}||_2^2. \tag{4}$$

If the k_z vectors in $\mathbf{C}_i$ are linearly independent, we use the normal equations with the Cholesky decomposition to solve the least-squares problem. If the normal equations fail, we use the more expensive SVD to get the least-squares approximation of the data item. Even though there has been no attempt to create orthogonal representative vectors, in the vast majority of cases the normal equations were sufficient to solve the least-squares problem. The high independence among the centroids **C** is consistent with the behavior observed in [9]. The parameters used to compute the LMFR are summarized in Table 1.

When $k_z = k_c$, this factorization of **A** is essentially identical to the concept decomposition [9]. We typically select a value for k_z such that $k_z \ll k_c$, which can result in significant memory savings. Since the memory savings are dependent on **Z** being sparse, we also require the condition that $k_z \ll n$. Thus,

Table 1
Definition of the parameters used in the low-memory approximation

Parameter	Description
m	Total number of data items per section
n	Number of attributes per data item
γ	Fill fraction for the attributes
k_c	Number of centroids per section
k_z	Number of centroids approximating each data item

The fill fraction γ is only appropriate for sparse data sets.

low-dimension data are not a good candidate for this factorization technique from a memory-savings standpoint.

Also, note that the memory savings are dependent on the LMFR remaining in factored form. If the product **CZ** were explicitly formed for all the columns in **Z** at once, the amount of memory occupied would be at least as much as that occupied by the original representation of the data. When factoring sparse data sets, the usual result is that **C** is denser than **A**, which means that the explicit formation of the product **CZ** would take up more memory than the original representation of the data.

The memory savings over the original representation of the data are significant, but there is a limit to the number of data items which can be represented in the LMFR. Each data item has a column associated with it in **Z**. That column in **Z** requires a finite amount of memory to contain it. Therefore, the LMFR cannot approximate the entire contents of an infinite data stream. Instead, it allows for the unique in-memory representation of many more data items than would be possible using the original representation of the data.

3.2.1. Cost of computing *CZ*

The cost of computing **CZ** for one section of data **A** can be broken down into the cost of computing each matrix separately. First, we will discuss the cost of computing **C**. If we use PDDP to obtain a clustering of **A**, then the cost of obtaining k_c clusters is

$$\text{Cost}_{\mathbf{C}} = m\gamma n \log_2(k_c), \tag{5}$$

where the parameters are as defined in Table 1. We assume we can extract the cluster centroids used to form **C** (3) at negligible cost, since the centroids are computed during the PDDP clustering and are readily available. Therefore, the cost of obtaining **C** is the cost of obtaining the clustering of **A**.

We construct **Z** one column at a time. To obtain one column $\mathbf{z}_i$ of **Z**, we must compute the Euclidean distance from $\mathbf{a}_i$ to every representative in **C**, find the k_z representatives in **C** closest to $\mathbf{a}_i$ and collect them to form $\mathbf{C}_i$, and finally compute the least-squares approximation to $\mathbf{a}_i$ using the representatives in $\mathbf{C}_i$.

It requires n multiplications and subtractions to compute the Euclidean distance from $\mathbf{a}_i$ to a single representative in $\mathbf{C}$. Since there are k_c representatives in $\mathbf{C}$, the total cost of computing the distances for one data item $\mathbf{a}_i$ is $k_c n$. We assume that the number of representatives k_z used to approximate $\mathbf{a}_i$ is very small compared to k_c, so that it will be less expensive to directly select the k_z closest representatives, rather than sorting the distances first. Therefore, it takes k_z k_c searches through the distances to find the k_z closest representatives. Once found, the representatives are used to form $\mathbf{C}_i$.

The final step is to compute the least-squares approximation for the data item using the k_z representatives obtained in the previous step. Assume the normal equations are used to obtain the least-squares approximation. The cost of computing a least-squares approximation using the normal equations is

$$\mathrm{Cost}_{\mathrm{LS}} = k_z^2 n + \frac{1}{3}k_z^3 \tag{6}$$

if we ignore the lower-order $\mathrm{O}(k_z^2)$ term. The total combined cost for obtaining $\mathbf{z}_i$ for one data item $\mathbf{a}_i$ is

$$\mathrm{Cost}_{\mathrm{Z}} = k_c n + k_c k_z + k_z^2 n + \frac{1}{3}k_z^3. \tag{7}$$

The combined cost of producing the factored representation $\mathbf{CZ}$ of $\mathbf{A}$ is

$$\mathrm{Cost}_{\mathrm{CZ}} = m\left(n(\gamma \log_2(k_c) + k_c + k_z^2) + k_c k_z + \frac{1}{3}k_z^3\right). \tag{8}$$

To obtain the cost per data item, we divide (8) by m, with the result being

$$\text{Amortized } \mathrm{Cost}_{\mathrm{CZ}} = n(\gamma \log_2(k_c) + k_c + k_z^2) + k_c k_z + \frac{1}{3}k_z^3. \tag{9}$$

The memory occupied by each section of the LMFR we have just computed is

$$\mathrm{memory}_{\mathrm{CZ}} = k_c n + \beta k_z m, \tag{10}$$

where the first term is the space occupied by $\mathbf{C}$ assuming it is completely dense, and the second term is the space occupied by $\mathbf{Z}$. The constant β is associated with the overhead required for each non-zero element present in sparse matrix storage. This last term is a natural consequence of the fact that we are keeping a representation of each individual data sample, but each data sample is represented by only k_z parameters, and in our experiments k_z is typically a very small number like 3. Each new section will also have its own $\mathbf{C}$ factor, but in the sequel we will discuss three possible ways to consolidate and/or reduce the space occupied by all the collected $\mathbf{C}$ factors.

3.3. Constructing the global LMFR

Now that we have a technique to compute the LMFR of one section of data, we need to apply the technique to constructing the LMFR of the entire data stream seen so far. Suppose that we have not seen any data from the stream, and that we accumulate the first section of data, comprised of m data items, from the stream. Let this first section of data be defined by the $n \times m$ matrix $\mathbf{A}^{(1)}$. We can construct the LMFR of $\mathbf{A}^{(1)}$

$$\mathbf{A}^{(1)} \approx \mathbf{C}^{(1)}\mathbf{Z}^{(1)}, \tag{11}$$

as shown in Section 3.2. At this point, we have a factored representation of the first section of data. We can now delete that data from memory and accumulate a new section of data. Let us call this new section of data $\mathbf{A}^{(2)}$. We construct a low-memory representation of $\mathbf{A}^{(2)}$,

$$\mathbf{A}^{(2)} \approx \mathbf{C}^{(2)}\mathbf{Z}^{(2)}, \tag{12}$$

in the exact same manner as was done for $\mathbf{A}^{(1)}$. We now have two separate LMFRs for the two sections of data. We can construct a LMFR that comprises both sections of data

$$\mathbf{C}_{\mathbb{G}}^{(2)} = [\,\mathbf{C}^{(1)} \quad \mathbf{C}^{(2)}\,] \quad \text{and} \quad \mathbf{Z}_{\mathbb{G}}^{(2)} = \begin{bmatrix} \mathbf{Z}^{(1)} & \mathbf{0} \\ \mathbf{0} & \mathbf{Z}^{(2)} \end{bmatrix}, \tag{13}$$

where $\mathbf{C}_{\mathbb{G}}^{(2)}$ and $\mathbf{Z}_{\mathbb{G}}^{(2)}$ represent the global factorization of the two sections of data seen so far. Because we use a sparse internal representation for $\mathbf{Z}_{\mathbb{G}}$, only the non-zero entries within the non-zero blocks must be stored.

This process continues as long as we have new data arriving. Each new section of data is approximated and added to the LMFR which reflects all sections of data previously seen. Therefore, if $i-1$ sections of data have been added to the LMFR, and the ith new section of data $\mathbf{a}^{(i)}$ has just been approximated as $\mathbf{C}^{(i)}\mathbf{z}^{(i)}$, we can create the updated global LMFR of the data, $\mathbf{C}_{\mathbb{G}}^{(i)}\mathbf{Z}_{\mathbb{G}}^{(i)}$

$$\mathbf{C}_{\mathbb{G}}^{(i)} = [\,\mathbf{C}_{\mathbb{G}}^{(i-1)} \quad \mathbf{C}^{(i)}\,] \quad \text{and} \quad \mathbf{Z}_{\mathbb{G}}^{(i)} = \begin{bmatrix} \mathbf{Z}_{\mathbb{G}}^{(i-1)} & \mathbf{0} \\ \mathbf{0} & \mathbf{z}^{(i)} \end{bmatrix}. \tag{14}$$

Note that each section of data would usually have the same size and use the same parameter values (cf. Table 1) when constructing the LMFR, but this is not a requirement. The algorithm is shown in Fig. 1.

We have already remarked (cf. Eq. (10)) that the LMFR represents each input data sample with k_z parameters, where k_z is a small number on the order of 3. Hence, it is clear that the memory requirement of the LMFR grows without bound as the input data are processed. We can reduce the memory growth up to a point. First, we note that since each data item is represented by a very small set of parameters, we can manipulate at once a much larger number of

Algorithm LMFR.

0. **Start** with a set of n-dimensional streaming data. Set the values for k_c and k_z (see Table 1), and p, the number of sections of streaming data that will be approximated.
1. **For** $i = 1, 2, \ldots, p$ **do**
2. **Accumulate** the latest m points from the data stream and use them to form the $n \times m$ matrix $\mathbf{A}^{(i)}$.
3. **Compute** the PDDP tree for the matrix $\mathbf{A}^{(i)}$ with k_c clusters.
4. **Assemble** the k_c centroids from the leaf clusters into an $n \times k_c$ matrix $\mathbf{C}^{(i)}$.
5. **Compute** the $k_c \times m$ matrix $\mathbf{Z}^{(i)}$ minimizing the quantity $\|\mathbf{M}_j - \mathbf{C}_j\mathbf{Z}_j\|_F$, subject to the constraint on the number of non-zero elements k_z in each column of $\mathbf{Z}_j$.
6. **If** $i = 1$, set $\mathbf{C}_{\mathbb{G}}^{(i)} = \mathbf{C}^{(i)}$ and $\mathbf{Z}_{\mathbb{G}}^{(i)} = \mathbf{Z}^{(i)}$.
7. **Else**, form $\mathbf{C}_{\mathbb{G}}^{(i)}$ and $\mathbf{Z}_{\mathbb{G}}^{(i)}$ as in (14) in the text.
8. **Result:** A low-memory representation of the first p sections of streaming data.

Fig. 1. LMFR algorithm.

such data items using the factored representation than with the original data. Second, we note that given that the data has been collected and grouped into sections, it is easy to remove stale data from the LMFR by removing the corresponding sections, thus adopting a sliding window approach. Third, we propose in Section 3.4 three methods for reducing the memory requirements by consolidating the columns of $\mathbf{C}_{\mathbb{G}}$. These methods can be used to put a hard bound on the space occupied by the $\mathbf{C}_{\mathbb{G}}$ factor. The $\mathbf{Z}$ factor, however, as mentioned previously, will always grow as new data are added to the factorization. The only control over the size of $\mathbf{Z}$ is through the parameter k_z (cf. Table 1), which must be greater than 1 if a unique representation of each data item is to be maintained.

3.4. Reducing the size of the global LMFR

Like any approximating technique, the LMFR has limits. Each data item approximated by the global LMFR has a unique column in $\mathbf{Z}_{\mathbb{G}}$, and that column requires a fixed amount of space in memory. Given a large enough data stream, it is possible that the global representation $\mathbf{C}_{\mathbb{G}}\mathbf{Z}_{\mathbb{G}}$ would exceed the available memory space. Before reaching that point, some method to reduce the size of $\mathbf{C}_{\mathbb{G}}\mathbf{Z}_{\mathbb{G}}$ would be required.

One way to reduce the memory footprint would be to eliminate some of the data points from the LMFR. This can be accomplished by deleting the corresponding columns in $\mathbf{Z}_{\mathbb{G}}$. When an entire section of data points has been

deleted, the corresponding columns of $\mathbf{C}_{\mathbb{G}}$ can also be deleted. If all individual data points must be preserved at least approximately, the only way to reduce the memory footprint of $\mathbf{C}_{\mathbb{G}}\mathbf{Z}_{\mathbb{G}}$ is to re-compute $\mathbf{C}_{\mathbb{G}}$, re-compute $\mathbf{Z}_{\mathbb{G}}$, or re-compute both matrices. Note, however, that once $\mathbf{C}_{\mathbb{G}}$ has been re-computed, all columns in $\mathbf{C}_{\mathbb{G}}$ will be shared across all approximations in $\mathbf{Z}_{\mathbb{G}}$, and stale data can only be removed by deleting the corresponding columns in $\mathbf{Z}_{\mathbb{G}}$.

The following methods reduce the memory requirements of the LMFR. The form of the LMFR remains unaltered. The same memory savings could be achieved by starting over with the original data stream while using more conservative LMFR parameter choices (cf. Table 1). These methods provide an alternative.

3.4.1. Directly re-computing $\boldsymbol{C}_G$ and $\boldsymbol{Z}_G$

One solution to reducing the size of $\mathbf{C}_{\mathbb{G}}$ and $\mathbf{Z}_{\mathbb{G}}$ is to compute a new global LMFR using the reconstructed data from the existing global LMFR. Assume we have already computed the $n \times k_c$ matrix $\mathbf{C}_{\mathbb{G}}$ and the $k_c \times m_{\mathbb{G}}$ matrix $\mathbf{Z}_{\mathbb{G}}$ which has k_z non-zero entries per column. We wish to compute

$$\widehat{\mathbf{C}}_{\mathbb{G}}\widehat{\mathbf{Z}}_{\mathbb{G}} \approx \mathbf{C}_{\mathbb{G}}\mathbf{Z}_{\mathbb{G}}, \tag{15}$$

where $\widehat{\mathbf{C}}_{\mathbb{G}}$ has dimension $n \times \hat{k}_c$ and $\widehat{\mathbf{Z}}_{\mathbb{G}}$ has dimension $\hat{k}_c \times m_{\mathbb{G}}$ with $\hat{k}_c \ll k_c$ and $\hat{k}_z$ non-zero entries per column. Typically $\hat{k}_z = k_z$, but this could be varied. We obtain $\widehat{\mathbf{C}}_{\mathbb{G}}$ from a clustering of $\mathbf{C}_{\mathbb{G}}$, as shown in Section 3.2. We compute each column in $\widehat{\mathbf{Z}}_{\mathbb{G}}$ in the same manner as in (4), except that in place of the original data item $\mathbf{a}_i$ we use the re-constructed data item

$$\grave{\mathbf{a}}_i = \mathbf{C}_{\mathbb{G}} - \mathrm{Z}_{\mathbb{G}i}, \tag{16}$$

where $\mathbf{Z}_{\mathbb{G}i}$ is the ith column of $\mathbf{Z}_{\mathbb{G}}$. Since we use the re-constructed data items when computing $\widehat{\mathbf{Z}}_{\mathbb{G}}$, we do not have to access the original data again. However, if we have access to the data and can afford the cost, it might be advantageous to use the original data to compute $\widehat{\mathbf{Z}}_{\mathbb{G}}$. We also have the option of setting $\widehat{k}_z$ to a value less than k_z, which can save a significant amount of memory. Once we have completed the computation, the new $\widehat{\mathbf{C}}_{\mathbb{G}}\widehat{\mathbf{Z}}_{\mathbb{G}}$ replaces the old $\mathbf{C}_{\mathbb{G}}\mathbf{Z}_{\mathbb{G}}$, which is discarded when we make the assignments

$$\mathbf{C}_{\mathbb{G}} \Leftarrow \widehat{\mathbf{C}} \quad \text{and} \quad \mathbf{Z}_{\mathbb{G}} \Leftarrow \widehat{\mathbf{Z}}_{\mathbb{G}}. \tag{17}$$

We then continue to augment the revised LMFR as new data arrive as in (14).

Since we choose the parameters used to construct $\widehat{\mathbf{C}}_{\mathbb{G}}\widehat{\mathbf{Z}}_{\mathbb{G}}$, we also know how much memory $\widehat{\mathbf{C}}_{\mathbb{G}}\widehat{\mathbf{Z}}_{\mathbb{G}}$ will occupy, and therefore how much memory savings we will achieve by completing the computations. The drawbacks are that the entire data set has to be reconstructed during the process, and that we are building an entire new $\mathbf{Z}_{\mathbb{G}}$ from scratch. This can be very expensive if there are many data items approximated by the global LMFR $\mathbf{C}_{\mathbb{G}}\mathbf{Z}_{\mathbb{G}}$, as we expect would be the usual case.

The additional costs of this method over those shown in (8) are the costs of reconstructing the data. Since $\mathbf{Z}_{\mathbb{G}}$ is sparse and we only need to consider non-zero entries in a column, reconstructing one column of data (16) requires nk_z multiplications and additions. Assuming a total of $m_{\mathbb{G}}$ data items, the total cost of computing $\widehat{\mathbf{C}}\widehat{\mathbf{Z}}_{\mathbb{G}}$ is

$$m_{\mathbb{G}}\left(n\left(\gamma\log_2(\widehat{k}_c)+\widehat{k}_c+k_z+\widehat{k}_z^2\right)+\widehat{k}_c\widehat{k}_z+\frac{1}{3}\widehat{k}_z^3\right). \tag{18}$$

3.4.2. Approximating $\boldsymbol{C}_{\mathbb{G}}$

If we are only interested in reducing the size of $\mathbf{C}_{\mathbb{G}}$, we can save some computation time over the method in Section 3.4.1. Assume as before that we have already computed the $n\times k_c$ matrix $\mathbf{C}_{\mathbb{G}}$ and the $k_c\times m_{\mathbb{G}}$ matrix $\mathbf{Z}_{\mathbb{G}}$ which has k_z non-zero entries per column. We wish to compute a LMFR of $\mathbf{C}_{\mathbb{G}}$ alone

$$\widetilde{\mathbf{C}}\widetilde{\mathbf{Z}}\approx\mathbf{C}_{\mathbb{G}}, \tag{19}$$

where $\widetilde{\mathbf{C}}$ has dimension $n\times\widetilde{k}_c$, $\widetilde{\mathbf{Z}}$ has dimension $\widetilde{k}_c\times m_{\mathbb{G}}$ with $\widetilde{k}_z$ non-zeroes per column, and $\widetilde{k}_c\ll k_c$. This LMFR of $\mathbf{C}_{\mathbb{G}}$ is computed in the usual manner as in Section 3.2.

Since we are only computing a new approximation of $\mathbf{C}_{\mathbb{G}}$, we do not need to reconstruct the data, which can save significant computation time. The resulting global LMFR becomes a three-matrix factorization

$$\widetilde{\mathbf{C}}\widetilde{\mathbf{Z}}\mathbf{Z}_{\mathbb{G}}. \tag{20}$$

Of course, this means that any data mining algorithm using the LMFR as an input would have to allow for factorizations consisting of more than two matrices.

The method could be generalized to handle multiple re-factorizations. For example, it would be possible to factor $\widetilde{\mathbf{C}}$ as

$$\widetilde{\mathbf{C}}=\widetilde{\widetilde{\mathbf{C}}}\,\widetilde{\widetilde{\mathbf{Z}}} \tag{21}$$

with the result being the global LMFR having the factorization

$$\widetilde{\widetilde{\mathbf{C}}}\,\widetilde{\widetilde{\mathbf{Z}}}\widetilde{\mathbf{Z}}\mathbf{Z}_{\mathbb{G}}. \tag{22}$$

This process of re-factorization could continue indefinitely. Carrying around these cascading re-factorizations would add a bit of complication and expense. It would still be possible to update $\mathbf{C}_{\mathbb{G}}$ and $\mathbf{Z}_{\mathbb{G}}$ in a manner analogous to (14) when new sections of data arrive, but with the additional complication of more than two factors. In principle, one could explicitly coalesce the factors

$$\grave{\mathbf{Z}}_{\mathbb{G}}=\widetilde{\mathbf{Z}}\mathbf{Z}_{\mathbb{G}} \tag{23}$$

leading to the following update for the global LMFR:

$$\mathbf{C}_{\mathbb{G}} \Leftarrow \widetilde{\mathbf{C}} \quad \text{and} \quad \mathbf{Z}_{\mathbb{G}} \Leftarrow \grave{\mathbf{Z}}_{\mathbb{G}}. \tag{24}$$

However, it is possible (indeed likely) that $\grave{\mathbf{Z}}_{\mathbb{G}}$ will be denser than $\mathbf{Z}_{\mathbb{G}}$, depending on the choice of $\tilde{k}_z$. This results in losing some of the memory savings inherent in factoring $\mathbf{C}_{\mathbb{G}}$. The density of the resulting $\mathbf{Z}_{\mathbb{G}}$ will increase each time (19), (20), (23) and (24) are computed, more so for the columns in $\mathbf{Z}_{\mathbb{G}}$ which existed the first time this technique was applied.

3.4.3. Hybrid approximation of $\boldsymbol{C}_G\boldsymbol{Z}_G$

We have presented two different methods for decreasing the memory footprint of $\mathbf{C}_{\mathbb{G}}\mathbf{Z}_{\mathbb{G}}$ as constructed in Section 3.3. The first method, recomputing both $\mathbf{C}_{\mathbb{G}}$ and $\mathbf{Z}_{\mathbb{G}}$, will be expensive due to both the cost of reconstructing the data and the costs associated with computing a new least-squares approximation to each data item. The second method, computing a LMFR of $\mathbf{C}_{\mathbb{G}}$, will have some issues with memory growth.

To balance the two, we suggest a third hybrid method in which the relatively inexpensive second method (computing an LMFR of $\mathbf{C}_{\mathbb{G}}$) is applied until the memory footprint of the LMFR grows too large. At that point, the more expensive first method of recomputing both $\mathbf{C}_{\mathbb{G}}$ and $\mathbf{Z}_{\mathbb{G}}$ is used to reduce the sparsity of $\mathbf{Z}_{\mathbb{G}}$ to the desired level.

This approach will be especially effective in cases where a large number of representatives are used when computing the LMFR of the latest section of data, and when there are already a large number of data points approximated in the LMFR. The relatively inexpensive computation of the LMFR of $\mathbf{C}_{\mathbb{G}}$ would help to offset the likely inevitable need to compute a new $\mathbf{Z}_{\mathbb{G}}$ from scratch.

4. Experimental results

In this section, we illustrate the method on two different data mining applications, one involving dense data and one involving sparse data. For the experiments, we chose to use clustering to validate the effectiveness of the LMFR for data mining. Specifically, we demonstrate that the performance of the clustering method PDDP [5] is not significantly hindered by replacing the original representation of the data by the LMFR of the data with respect to the quality of the clusters.

Clustering is just one possible application of the LMFR to streaming data mining, and is not intended as the sole application. If clustering is the sole goal, a streaming method which is specifically designed for clustering (e.g. [21]) would be faster since it avoids the pre-processing involved in constructing the LMFR. However, the LMFR can be used in the general case, not just for clustering, while the results of a pre-processing for a clustering-specific method will most likely not be usable for other data mining tasks.

4.0.1. Performance measures

We used scatter and entropy to measure clustering performance. The scatter is a measure of the cohesiveness of the data items in a cluster with respect to the centroid of the cluster. The scatter $s_{\mathbb{C}}$ of a cluster $\mathbf{M}_{\mathbb{C}}$ is defined as

$$s_{\mathbb{C}} \stackrel{\text{def}}{=} \sum_{j\in\mathbb{C}} (\mathbf{x}_j - \mathbf{w}_{\mathbb{C}})^2 = ||\mathbf{M}_{\mathbb{C}} - \mathbf{w}_{\mathbb{C}}\mathbf{e}^{\mathrm{T}}||_{\mathrm{F}}^2, \tag{25}$$

where $\mathbf{w}_{\mathbb{C}}$ is the centroid, $\mathbf{e}$ is the m-dimensional vector $[1\ 1\ \cdots\ 1]^{\mathrm{T}}$ and $\|\cdot\|_{\mathrm{F}}$ is the Frobenius norm. We used the normalized sum of the scatters across all clusters. Scatter is a relative performance measure, so it is usually appropriate only when comparing clusterings with the same number of clusters. When comparing two clusterings with the same number of clusters, the one with the lower scatter value is assumed to have higher cluster quality.

The entropy measures the coherence of a cluster with respect to how a cluster is labeled. An entropy calculation assumes that the labeling is perfect. The entropy of cluster j is defined by

$$e_j \stackrel{\text{def}}{=} -\sum_i \left(\frac{c(i,j)}{\sum_i c(i,j)}\right) \cdot \log\left(\frac{c(i,j)}{\sum_i c(i,j)}\right), \tag{26}$$

where $c(i,j)$ is the number of times label i occurs in cluster j. If all of the labels of the items in a given cluster are the same, then the entropy of that cluster is zero. Otherwise, the entropy is positive. The total entropy for a given clustering is the weighted average of the cluster entropies

$$e_{\text{total}} \stackrel{\text{def}}{=} \frac{1}{m}\sum_i e_i \cdot k_i. \tag{27}$$

As with the scatter, entropy is a relative performance measure, and a lower scatter value indicates better cluster quality.

4.0.2. Implementation details

The algorithms were implemented in MATLAB, and the experiments were performed on a custom-built PC. The PC had an AMD 2200+ CPU, 1 GB of memory, and 1.5 GB of swap space. The operating system used was Linux, Redhat 9.0.

4.1. Data sets

We used two data sets, one dense and one sparse, to evaluate the method to obtain a LMFR of streaming data. The dense data set was the intrusion detection data used in the KDD Cup 1999 competition. This data set was obtained from the UCI:KDD machine learning repository [14]. The point of the competition was to build a classifier that could determine the difference between a normal network connection and a network connection associated with an

Table 2
Summary of the KDD intrusion detection data

Dataset	KDD
Number of samples m	489,8431
Number of attributes n	122
Number of connection types	23
Connection type	*Number of appearances*
Normal	972,781
Smurf	2,807,886
Neptune	1,072,017
Satan	15,892
Ipsweep	12,481
Portsweep	10,413
Remaining 17	6961

All connection types other than "normal" are attacks of some kind. The data set was obtained from the UCI:KDD machine learning repository [14].

attack. We combined the test and training data into one large data set. A datum consists of attributes associated with computer network connections, such as the protocol used to make the connection (tcp, udp, icmp) the service type requested (ftp, http, etc.), the number of password failures, the number of root accesses, etc. The data set consists of both continuous and categorical attributes. The categorical attributes were converted to binary attributes in order to conform to the vector space model. Each attribute was scaled to have mean 0 and variance $\frac{1}{m}$. Finally, the data were randomly divided into 25 equal-sized sets. The KDD data set is summarized in Table 2.

The sparse data set was the document data set provided by the Reuters News Service [23]. It consists of roughly 810,000 news items dated from 1996-08-20 to 1997-08-19. After removing the stop words and applying Porter's stemming algorithm, removing documents with less than 10 words in them, and removing words that appeared in only one news document, the data set was reduced to 802,935 items composed from a possible 185,953 words in the dictionary. The dictionary contained a large number of misspelled words and other non-words, but was used "as is" anyway because there was no automated way to remove them. Since Reuters usually assigned many different topic labels to each document, there were 14,897 distinct combinations of topics. We treated each combination as a distinct topic. Finally, the data were scaled so each document vector had unit length. The data were then randomly divided into eight equal-sized sets. The Reuters data set is summarized in Table 3.

4.2. Effectiveness of the LMFR

First, we evaluate the effectiveness of the standard LMFR algorithm (cf. Fig. 1) in the context of a clustering operation. Each set of original data was

Table 3
Summary of the Reuters News Service data set

Dataset	Reuters
Number of samples m	802,935
Number of attributes n	185,953
Number of categories	14,897

The number of items m and the number of words in the dictionary n are the post-processing values. The data set was obtained directly from the Reuters News Service [23].

further subdivided into five sections of the same size. This meant that only a relatively small number of data points were exposed to the LMFR algorithm at any given time, which should highlight any side effects of the LMFR method. We compared the results of a PDDP clustering of the original representation of the data $\mathbf{M}$ with a PDDP clustering of the low-memory representation of the data $\mathbf{C}_G\mathbf{Z}_G$. We call the combination of a PDDP clustering of the LMFR $\mathbf{C}_G\mathbf{Z}_G$ Piecemeal PDDP (PMPDDP) [19]. Both clusterings were taken to the same number of final clusters k_f.

The parameters and results for the KDD data set are shown in Table 4, and the time taken to produce the LMFR $\mathbf{C}_G\mathbf{Z}_G$ is further illustrated in Fig. 2(a). There are no results for PDDP for $m > 979{,}685$ because the data occupied too much memory to allow for computational overhead. The clustering quality from PMPDDP is comparable to PDDP in both the scatter and entropy performance measures, which indicates the LMFR reflects the data sufficiently for a PDDP clustering of the KDD data set. The costs are higher for PMPDDP, with the majority of the time is spent computing the LMFR of the data. Once the LMFR is available, clusterings with a differing number of final clusters can be computed relatively inexpensively.

For example, in Table 4 for $m = 979{,}685$, quality as measured by entropy was actually better for PMPDDP (.114) than for PDDP (.120). The time to compute the LMFR was relatively high (1108.56 s), but the LMFR required substantially less memory than the original representation of the data $\mathbf{M}$ (48.75 MB vs. 956 MB). Once the LMFR was computed, the final clustering for PMPDDP could be carried out very fast (only 89.62 s vs. 282.44 for PDDP). The PDDP clustering method used in this final step works directly on the factored representation, whereas any other method needing the distances between data items would require the explicit reconstruction of the individual data items. The timing results for the different sizes indicate that the cost of PDDP is linear in the number of data items, as expected [5]. As a consequence, the average time per data item is a constant, as shown in Fig. 2(a).

The memory savings from the LMFR are significant. The entire KDD intrusion detection data set in its original representation would occupy over 4 GB of memory, beyond the limits of most desktop workstations. The LMFR of the entire KDD data set only requires about 200 MB of memory for the

Table 4
KDD data parameters and results

Dataset	KDD					
m	195,937	391,874	587,811	783,748	979,685	489,8431
# of sections	5	10	15	20	25	125
k_c per section	392	392	392	392	392	392
k_z	3	3	3	3	3	3
k_f	36	36	36	36	36	36
Scatter values, lower is better						
PDDP	3.179e–04	3.279e–04	3.276e–04	3.290e–04	3.288e–04	Won't fit
PMPDDP	3.257e–04	3.236e–04	3.271e–04	3.250e–04	3.275e–04	NA
Entropy values, lower is better						
PDDP	.127	.130	.129	.124	**.120**	Won't fit
PMPDDP	.0590	.0585	.0546	.120	**.114**	.113
Time taken by experiments, in seconds, on XP 2200+						
PDDP	39.72	89.68	140.45	204.87	**282.44**	Won't fit
Compute $\mathbf{C}_{\mathbb{G}}\mathbf{Z}_{\mathbb{G}}$	216.66	450.41	674.49	872.15	**1108.56**	5652.59
Cluster $\mathbf{C}_{\mathbb{G}}\mathbf{Z}_{\mathbb{G}}$	15.63	32.71	52.67	69.07	**89.62**	492.81
PMPDDP totals	232.29	483.12	727.16	941.22	1198.18	6145.40
Memory occupied by representation, in MB						
M	191.20	382.40	573.60	764.90	**956.20**	4780
$\mathbf{C}_{\mathbb{G}}\mathbf{Z}_{\mathbb{G}}$	8.24	16.48	24.72	39.00	**48.75**	206

The clustering performance of PDDP is not significantly hindered by replacing the original representation of the data with the LMFR. The time taken to produce the LMFR appears to be linear in the number of data items. The times for clustering the LMFR $\mathbf{C}_{\mathbb{G}}\mathbf{Z}_{\mathbb{G}}$ are similar to those for clustering the original representation of the data $\mathbf{M}$. The memory used by $\mathbf{C}_{\mathbb{G}}\mathbf{Z}_{\mathbb{G}}$ vs. $\mathbf{M}$ was less by a factor of 23. This allowed the clustering of almost 5 million data points, which would normally occupy 4.78 GB of memory. The scatter values for PMPDDP on the entire data set could not be computed since the data could not fit into memory to make the computation. See Table 1 for an explanation of the parameter names.

parameters selected, leaving plenty of memory space for computational overhead on a standard workstation.

The parameters and results for the Reuters data set are shown in Table 5, and the time to compute $\mathbf{C}_{\mathbb{G}}\mathbf{Z}_{\mathbb{G}}$ is further illustrated in Fig. 2(b). There was just enough room to contain the entire data set in memory on a 1 GB machine, although some paging did occur during the computations. The results indicate that the clusterings generated by PMPDDP are comparable to PDDP clusterings, which means that using the LMFR in place of the original representation of the data does not significantly hinder the clustering performance of PDDP. Again, we see that the costs of constructing the LMFR and clustering the LMFR are linear in the number of data items, though the costs themselves are much higher than when clustering with PDDP. However, a similar sparse data set which could not fit into memory would not be able to be mined with

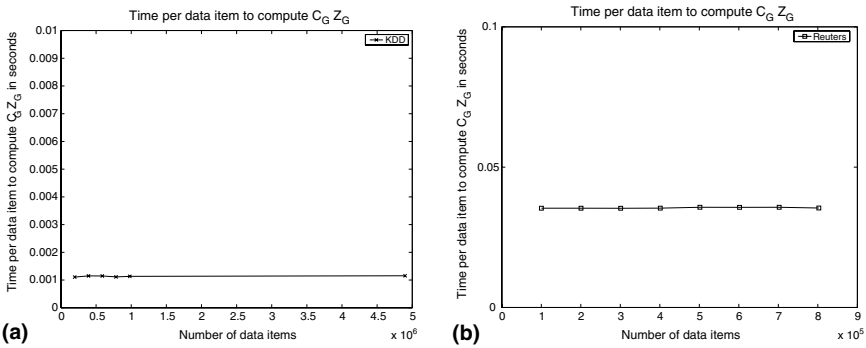

Fig. 2. Time taken to produce the LMFR $\mathbf{C}_G\mathbf{Z}_G$ for the KDD (a) and Reuters (b) data sets. The results indicate that the cost of producing $\mathbf{C}_G\mathbf{Z}_G$ is linear in the number of data items for both dense and sparse data.

any static data technique such as PDDP. The LMFR method allows for a much larger data set to be mined than would otherwise be possible.

Overall, we wish to point out that the LMFR saves a significant amount of memory over the original representation of the data. As a result, the LMFR creates an approximation to the data which allows for general data mining tasks, such as clustering, to be performed on a much larger piece of the data than would otherwise be possible.

4.3. Evaluation of the memory reduction methods

Next, we show some experimental results for the methods we can use to reduce the memory footprint of $\mathbf{C}_G\mathbf{Z}_G$ as described in Sections 3.4.1, 3.4.2, 3.4.3. To demonstrate the relative effectiveness of the methods, we changed the approach to the way the LMFR was constructed.

We modeled an extreme situation in which $\mathbf{C}_G$ was re-computed each time we added a new section to the global LMFR. We held the number of columns k_c in $\mathbf{C}_G$ to a constant number, equal to the total number of columns present after the global LMFR was computed for the entire data sets as in Section 4.2. This extreme situation was designed to highlight the various issues involved in re-computing $\mathbf{C}_G$, and is not representative of a typical LMFR construction which was shown in Section 3.3. In practice, we would expect to re-compute $\mathbf{C}_G$ infrequently.

We proceeded in the following manner. We approximated the first section of data directly, such that all k_c columns in $\mathbf{C}_G$ were associated with the first section. Then, to make room to incorporate next section of data, we computed a new $\mathbf{C}_G$ with half the original number of columns, $\frac{k_c}{2}$, using either the method from Section 3.4.1 or the method from Section 3.4.2. We updated $\mathbf{Z}_G$ as appropriate for the given method. Then the new section of data was loaded and a

Table 5
Reuters data parameters and results

Dataset	Reuters							
m	100,299	200,598	300,897	401,196	501,495	601,794	702,093	802,395
# of sections	5	10	15	20	25	30	35	40
k_c per section	201	201	201	201	201	201	201	201
k_z	3	3	3	3	3	3	3	3
k_f	320	320	320	320	320	320	320	320
Scatter values, lower is better								
PDDP	.7681	.7708	.7722	.7743	.7766	.7771	.7791	.7791
PMPDDP	.7707	.7759	.7776	.7804	.7813	.7821	.7833	.7832
Entropy values, lower is better								
PDDP	2.406	2.544	2.615	2.676	2.729	2.779	2.822	2.843
PMPDDP	2.446	2.580	2.654	2.705	2.763	2.813	2.863	2.877
Time taken by experiments, in seconds, on an XP 2200+								
PDD	630	1121	1555	1975.9	2501	2968	3435	3886
Compute $\mathbf{C}_{\mathbb{G}}\mathbf{Z}_{\mathbb{G}}$	3547	7095	10,637	14,204	17,907	21,478	25,062	28,446
Cluster $\mathbf{C}_{\mathbb{G}}\mathbf{Z}_{\mathbb{G}}$	7044	14,238	22,096	29,948	37,835	46,168	54,109	62,636
PMPDDP totals	10,592	21,333	32,733	44,151	55,742	67,645	79,172	91,081
Memory occupied by representation, in MB								
$\mathbf{M}$	92.75	185.9	279.5	373.7	468.1	562.9	658.2	753.1
$\mathbf{C}_{\mathbb{G}}\mathbf{Z}_{\mathbb{G}}$	33.63	67.40	102.9	135.5	169.7	204.1	238.8	273.1

As with the KDD data set, PDDP clustering quality using the LMFR $\mathbf{C}_{\mathbb{G}}\mathbf{Z}_{\mathbb{G}}$ is similar to the quality when using the original representation of the data $\mathbf{M}$. The cost of producing the LMFR again appears to be linear in the number of data items. One difference is that the cost of clustering the LMFR is an order of magnitude greater when using the LMFR $\mathbf{C}_{\mathbb{G}}\mathbf{Z}_{\mathbb{G}}$ in place of $\mathbf{M}$. This is due to the fact that $\mathbf{C}_{\mathbb{G}}$ is much denser than $\mathbf{M}$, so that more multiplications take place in the PDDP method using the LMFR in place of $\mathbf{M}$.

LMFR of that section was computed as in Section 3.2, again using a value of $\frac{k_c}{2}$ for the number of columns in $\mathbf{C}$. When the global LMFR was updated to reflect the latest section of data, as in (14), the result was that $\mathbf{C}_{\mathbb{G}}$ had k_c columns. This process of reducing the number of columns in $\mathbf{C}_{\mathbb{G}}$ by one-half and then adding the LMFR for the new section of data continued until the stream was exhausted. At that point, a PDDP clustering of the global LMFR was computed to determine whether the resulting LMFR was acceptable for general data mining computations.

To evaluate the hybrid method outlined in Section 3.4.3, we applied the following technique. Every fourth update of the global LMFR triggered the re-computation of both $\mathbf{C}_{\mathbb{G}}$ and $\mathbf{Z}_{\mathbb{G}}$, as shown in Section 3.4.1. Otherwise, we computed the LMFR of $\mathbf{C}_{\mathbb{G}}$ and updated $\mathbf{Z}_{\mathbb{G}}$ as in Section 3.4.2.

These methods can be contrasted with the method outlined in Section 3.3 and experimentally evaluated in Section 4.2. In Section 3.3, the global LMFR was computed without the intermediate step of re-computing $\mathbf{C}_{\mathbb{G}}$ and $\mathbf{Z}_{\mathbb{G}}$, and

$\mathbf{C}_{\mathbb{G}}$ was allowed to gradually increase in size rather than being held to a comparatively large constant number of columns after each section of data is added. However, barring the intermediate step of the re-computation of $\mathbf{C}_{\mathbb{G}}$ and $\mathbf{Z}_{\mathbb{G}}$ before adding a new section to the global LMFR, the methods outlined directly above are identical the algorithm presented in Section 3.3.

4.3.1. KDD results

The results for the time taken to compute $\mathbf{C}_{\mathbb{G}}\mathbf{Z}_{\mathbb{G}}$ and the memory occupied by $\mathbf{C}_{\mathbb{G}}\mathbf{Z}_{\mathbb{G}}$ for the KDD data are shown in Fig. 3. The value of k_c was set to 25,000, and k_z was set to 3. Note that this value of k_c is much greater than that used in previous experiments. MATLAB was not able to continue computations using the method from Section 3.4.2 past the addition of the eleventh section due to memory constraints. As expected, computing the LMFR of $\mathbf{C}_{\mathbb{G}}$ before a new section was added to the global LMFR was considerably less expensive than re-computing $\mathbf{Z}_{\mathbb{G}}$ each time, but it required much more memory to contain, especially as the number of columns in $\mathbf{Z}_{\mathbb{G}}$ increased. Fig. 3b shows a slow linear growth in the memory required by the "rebuild" method, almost entirely due to the added columns in the $\mathbf{Z}_{\mathbb{G}}$ matrix corresponding to the new data. The "sawtooth" effect in the memory growth for the hybrid method reflects the drop in memory requirement each time the factorization is regenerated using the method of (Section 3.4.1). It is seen that the amount of memory used returned to the expected minimum level each time that regeneration process was applied.

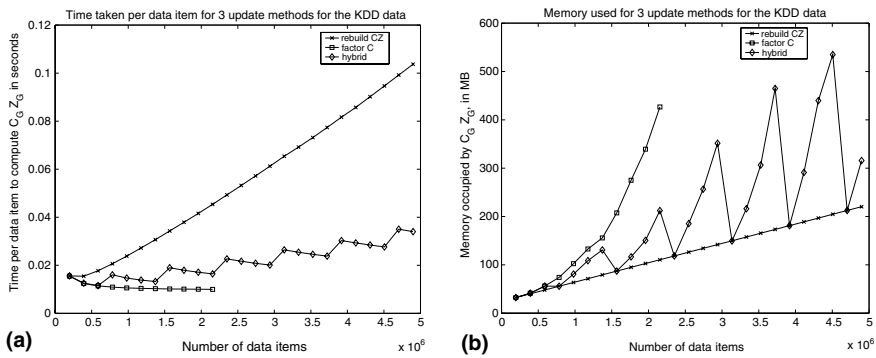

Fig. 3. Time taken to produce $\mathbf{C}_{\mathbb{G}}\mathbf{Z}_{\mathbb{G}}$ (a) and memory occupied by $\mathbf{C}_{\mathbb{G}}\mathbf{Z}_{\mathbb{G}}$ (b) for the KDD data set for the three memory reduction methods, rebuild **CZ** (Section 3.4.1), factor **C** (Section 3.4.2), and hybrid (Section 3.4.3). Computing the LMFR of $\mathbf{C}_{\mathbb{G}}$ was the least expensive computationally, and most expensive from a memory standpoint. Re-computing a new $\mathbf{C}_{\mathbb{G}}\mathbf{Z}_{\mathbb{G}}$ whenever a new section of data was added saved memory but was very expensive computationally. The hybrid method was effective at recovering the extra memory when computing an LMFR of $\mathbf{C}_{\mathbb{G}}$ while being much less expensive than re-computing a new $\mathbf{Z}_{\mathbb{G}}$ each time a new section of data was to be incorporated in the global LMFR.

Table 6
Memory reduction methods results for the KDD data set

Dataset	KDD				
	Scatter	Entropy	$\mathbf{C}_G\mathbf{Z}_G$ time (s)	Clust time (s)	Memory (MB)
Method					
Standard (Section 3.3)	3.322e–04	.114	5988	381.82	220.3
Rebuild $\mathbf{C}_G\mathbf{Z}_G$ (Section 3.4.1)	3.347e–04	.123	507,959	357.62	220.3
Hybrid (Section 3.4.3)	3.338e–04	.123	166,661	442	315.8

The performance for two of the three different methods are shown for the entire data set. The values from a PDDP clustering of the global LMFR as produced by the standard algorithm from Fig. 1 are shown for comparison purposes. All scatter values shown are computed with respect to $\mathbf{C}_G\mathbf{Z}_G$, not with respect to the original representation of the data. The clustering quality for the various memory reduction methods is comparable. The hybrid method appears to use more memory because the method halted before another full rebuild (Section 3.4.1) of $\mathbf{C}_G\mathbf{Z}_G$ was computed.

The clustering results for the KDD data are shown in Table 6. The clustering quality for the various techniques is comparable, which seems to indicate that the LMFRs produced by the various methods are all effective replacements for the original representation of the data.

The memory reduction methods appear to function correctly for the KDD data set. However, from the results it is apparent that the memory reduction methods should be used sparingly, since they can be expensive.

4.3.2. Reuters results

The results for the time taken to compute $\mathbf{C}_G\mathbf{Z}_G$ and the memory occupied by $\mathbf{C}_G\mathbf{Z}_G$ for the Reuters data are shown in Fig. 4. The value of k_c was set to 8080, and k_z was set to 3. Computing an LMFR of $\mathbf{C}_G$ was again the most efficient computationally but used the most memory of the three methods. Computing an entirely new $\mathbf{C}_G\mathbf{Z}_G$ was very expensive but resulted in a sub-linear increase in the amount of memory used. This is caused by the fact that $\mathbf{C}_G$ is getting denser as more data is added, so that the effect on the memory use when adding the LMFR for the latest section of data (14) is less severe. The "sawtooth" memory profile seen in Fig. 3b is much more hidden in Fig. 4b because the $\mathbf{C}_G$ factor occupies a significant amount of space, and tends to lose sparsity each time it is regenerated by the method of Section 3.4.2.

The clustering results for the Reuters data are shown in Table 7. The memory reduction methods have better performance in both scatter and entropy than the standard method. This is probably due to better representative vectors which comprise $\mathbf{C}_G$ being computed. With the standard method, a given section of data had only 201 representatives available to construct the LMFR of the section. With the updating methods, a given section of data used 4020 repre-

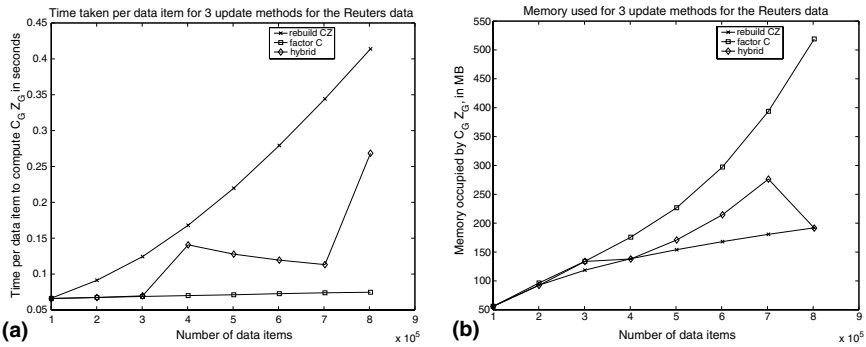

Fig. 4. Total time taken to produce $\mathbf{C}_G\mathbf{Z}_G$ (a) and memory occupied by $\mathbf{C}_G\mathbf{Z}_G$ (b) for the Reuters data set for the three memory reduction methods, rebuild **CZ** (Section 3.4.1), factor **C** (Section 3.4.2), and hybrid (Section 3.4.3). As with the KDD data, the time taken by recomputing a new $\mathbf{C}_G\mathbf{Z}_G$ each time a section of data is to be added is significantly greater than computing an LMFR of $\mathbf{C}_G$, but used less memory. The hybrid method was between the other two from both a memory and cost standpoint.

sentatives to construct the LMFR. Since so many representatives were available when computing the least-squares approximation, it makes sense that the overall approximation would improve. The improvement in the quality of the LMFR resulted in superior clustering performance. The additional representatives present are also the cause of the increased cost, since not only does it take longer to produce more representatives, it also takes longer to compute all the distances and complete the searches and least-squares computations required to construct $\mathbf{Z}_G$.

As with the KDD data, the results indicate that the memory reduction methods should be used infrequently. We suggest that they be applied sparingly and in

Table 7
Memory reduction methods results for the Reuters data set

Dataset	Reuters				
	Scatter	Entropy	$\mathbf{C}_G\mathbf{Z}_G$ time (s)	Clust time (s)	Memory (MB)
Method					
Standard (Section 3.3)	.7832	2.877	28,446	62,636	273.1
Rebuild $\mathbf{C}_G\mathbf{Z}_G$ (Section 3.4.1)	.7651	2.744	332,124	32,536	191.9
Factor $\mathbf{C}_G$ (Section 3.4.2)	.7668	2.738	59,944	45,702	519.1
Hybrid (Section 3.4.3)	.7651	2.736	215436	33134	191.5

The performance for the three different updating schemes are shown for the entire data set. The values from a PDDP clustering of the LMFR as produced by the standard algorithm from Fig. 1 are shown for comparison purposes.

a manner which significantly reduces the memory occupied by the global LMFR $\mathbf{C}_G\mathbf{Z}_G$, such that the memory recovered will offset the cost of the computations.

5. Summary

We have presented a method for constructing a low-memory representation of streaming data. The low-memory factored representation (LMFR) approximates each original data item uniquely. The ordering of the data is preserved. This low-memory representation can take the place of the original representation of the data in most data mining techniques. Replacing the original representation of the data with the low-memory representation allows for more data to be exposed at once to stream mining methods. It can also be used to mine data sets which would otherwise be too large to fit into memory. The representation is very well suited to windowing techniques, since expiring old data items from the representation is trivial.

The LMFR should also be useful in the case in which subsequent data mining tasks are performed based on the results of previous tasks. For this type of problem to be solved by a standard streaming data method, it may be necessary to re-scan the entire original data set in order to get the necessary information to solve subsequent tasks. The LMFR, however, maintains a compact, general representation of the original data, so the information can be extracted from the LMFR instead of re-scanning the original data set.

We demonstrated that the low-memory factored representation functions well in the context of clustering using PDDP. The quality of the clusterings of both the original representation and low-memory representation of the data were similar, which indicated that the LMFR captured the information necessary for a good clustering for the data sets examined. We expect that the LMFR can be used for many data mining tasks other than clustering, and is applicable when the data mining tasks would otherwise require multiple scans of the data to perform each desired task.

Acknowledgment

This work was partially supported by NSF Grant IIS-0208621.

References

[1] B. Babcock, M. Datar, R. Motwani, Sampling from a moving window over streaming data, in: Proceedings of Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, 2002, pp. 633–634.

[2] S. Babu, J. Widom, Continuous queries over data streams, ACM Sigmoid Record 30 (3) (2001) 109–120.

[3] M.W. Berry, S.T. Dumais, Gavin W. O'Brien, Using linear algebra for intelligent information retrieval, SIAM Review 37 (1995) 573–595.
[4] D. Boley, A scalable hierarchical algorithm for unsupervised clustering, in: R. Grossman, C. Kamath, P. Kegelmeyer, V. Kumar, R. Namburu (Eds.), Data Mining for Scientific and Engineering Applications, Kluwer, Dordrecht, 2001, pp. 383–400.
[5] D.L. Boley, Principal direction divisive partitioning, Data Mining and Knowledge Discovery 2 (1998) 325–344.
[6] P.S. Bradley, Usama M. Fayyad, C. Reina, Scaling clustering algorithms to large databases, in: Knowledge Discovery and Data Mining, 1998, pp. 9–15.
[7] K. Chakrabarit, M.N. Garofalakis, R. Rastogi, K. Shim, Approximate query processing using wavelets, in: Proceedings of the 2000 International Conference on Very Large Data Bases, 2000, pp. 111–122.
[8] M. Datar, A. Gionis, P. Indyk, R. Motwani, Maintaining stream statistics over sliding windows, in: Proceedings of Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, 2002, pp. 635–644.
[9] I.S. Dhillon, D.S. Modha, Concept decompositions for large sparse text data using clustering, Machine Learning 42 (1) (2001) 143–175.
[10] P. Domingos, G. Hulten, Mining high-speed data streams, in: Knowledge Discovery and Data Mining, 2000, pp. 71–80.
[11] A.C. Gilbert, Y. Kotidis, S. Muthukrishnan, M. Strauss, Surfing wavelets on streams: one-pass summaries for approximate aggregate queries, in: Proceedings of the 2001 International Conference on Very Large Data Bases, 2001.
[12] S. Guha, N. Mishra, R. Motwani, L. O'Callaghan, Clustering data streams, in: IEEE Symposium on Foundations of Computer Science, 2000, pp. 359–366.
[13] M. Henzinger, P. Raghavan, S. Rajagopalan, Computing on data streams, Technical Report 1998-011, Digital Systems Research Center, Palo Alto, CA, 1998.
[14] S. Hettich, S.D. Bay, The UCI KDD archive, 1999. Available from: <kdd.ics.uci.edu/>.
[15] G. Hulten, P. Domingos, Mining complex models from arbitrarily large databases in constant time, in: Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2002, pp. 525–531.
[16] G. Hulten, L. Spencer, P. Domingos, Mining time-changing data streams, in: Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM Press, San Francisco, CA, 2001, pp. 97–106.
[17] Y.E. Ioannidis, V. Poosala, Histogram-based approximation of set-valued query-answers, in: Proceedings of the 1999 International Conference on Very Large Data Bases, 1999, pp. 174–185.
[18] T.G. Kolda, D.P. O'Leary, A semidiscrete matrix decomposition for latent semantic indexing in information retrieval, ACM Transaction on Information Systems 16 (1998) 322–346.
[19] D. Littau, D. Boley, Using low-memory representations to cluster very large data sets, in: D. Barbará, C. Kamath (Eds.), Proceedings of the Third SIAM International Conference on Data Mining, 2003, pp. 341–345.
[20] Y. Matias, J.S. Vitter, M. Wang, Dynamic maintenance of wavelet-based histograms, in: Proceedings of the 2000 International Conference on Very Large Data Bases, 2000, pp. 101–110.
[21] L. O'Callaghan, N. Mishra, A. Meyerson, S. Guha, R. Motwani, Streaming-data algorithms for high-quality clustering, in: Proceedings of the 18th International Conference on Data Engineering, 26 February–1 March 2002, San Jose, CA, IEEE Computer Society, 2002, ISBN 0-7695-1531-2, pp. 685–696.
[22] S.M. Savaresi, D.L. Boley, Bisecting k-means and PDDP: a comparative analysis, Technical Report 00-048, University of Minnesota, Department of Computer Science, Minneapolis, MN, 2000.

[23] Reuters News Service. Reuters corpus, vol. 1, English Language, 1996-08-20 to 1997-08-19, 2000. Available from: <http://about.reuters.com/researchandstandards/corpus/available.asp>.
[24] T. Zhang, R. Ramakrishnan, M. Livny, BIRCH: a new data clustering algorithm and its applications, Data Mining and Knowledge Discovery 1 (2) (1997) 141–182.
[25] Z. Zhang, H. Zha, H. Simon, Low-rank approximations with sparse factors I: basic algorithms and error analysis, SIAM Journal on Matrix Analysis and Applications 23 (2002) 706–727.