M-Kernel Merging: Towards Density Estimation over Data Streams*

Aoying Zhou, Zhiyuan Cai, Li Wei, Weining Qian †
Department of Computer Science and Engineering, Fudan University
Laboratory of Intelligent Information Processing, Fudan University
{ayzhou,zycai,lwei,wnqian}@fudan.edu.cn

Abstract

Density estimation is a costly operation for computing distribution information of data sets underlying many important data mining applications, such as clustering and biased sampling. However, traditional density estimation methods are inapplicable for streaming data, which are continuously arriving large volume of data, because of their request for linear storage *and* square size calculation*. The shortcoming limits the application of many existing effective algorithms on data streams, for which the mining problem is an emergency for applications and a challenge for research. In this paper, the problem of computing density functions over data streams is examined. A novel method attacking this shortcoming of existing methods is developed to enable density estimation for large volume of data in* linear time*,* fixed size memory*, and* without lose of accuracy*. The method is based on* M-Kernel *merging, so that limited kernel functions to be maintained are determined intelligently. The application of the new method on different streaming data models is discussed, and the result of intensive experiments is presented. The analytical and empirical result show that this new density estimation algorithm for data streams can calculate density functions* on demand at any time *with* high accuracy for different streaming data models*.*

1 Introduction

Recently, it has been found that the technique of processing data streams is very important in a wide range of scientific and commercial applications. A **data stream** is such a model that a large volume of data is arriving continuously and it is either unnecessary or impractical to store the data in some forms. For example, transactions of banks, call records of telecommunications company, hit logs of web server are all these kinds of data. In these applications, decisions should be made as soon as various events (data) being received. It is not likely that processing accumulated data periodically by batches is allowed. Moreover, data streams are also regarded as models to access large data sets stored in secondary memory where performance requirements necessitate access with linear scans. In most cases, compared with the size of total data in auxiliary storage, the size of main memory is so limited that every time only a small portion of data can be loaded into memory, and with the time restrict only one scan is allowed to process the data set. How to apply an efficient way to organize and extract useful information from the data stream is a problem met by researchers from almost all fields. Though there are many algorithms for data mining, they are not designed for data stream.

To process high-volume, open-ended data streams, a method should meet some stringent criteria. In [6], Domingos presents a series of designed criteria, which are summarized as follows:

1. The time needed by the algorithm to process each data record in the stream must be small and constant; otherwise, it is impossible for the algorithm to catch up the pace of the data.

2. Regardless of the number of records the algorithm has seen, the amount of main memory used must be fixed.

3. It must be a *one-pass* algorithm, since in most applications, either the data is still not available, or there is no time to revisit old data.

4. It must have the ability to make a usable model available at any time, since we may never meet the end of the stream.

5. The model must be up-to-date at any point in time, that is to say, it must keep up with the changes of the data.

*This work is partially supported by "973" National Fundamental Research Programme of China (Grant No. G1998030414), National "863" Hi-Tech Programme of China (Grant No. 2002AA413310), and Fok Ying Tung Education Foundation (Grant No. 81062).

†The author is partially supported by Microsoft Research Fellowship.

Proceedings of the Eighth International Conference on Database Systems for Advanced Applications (DASFAA'03)
0-7695-1895/03 $17.00 © 2003 IEEE

The first two criteria are the most important and hard to achieve. Although much work have been done on scalable data mining algorithms, most algorithms still require an increasing main memory in proportion to data size, and their computation complexity is much higher than linear with data size. So they are not equipped to cope with data stream, for they will exhaust all available main memory or fall behind the data, some time or later.

Recent proposed techniques include clustering algorithms when objects arrive as a stream [8, 14], computing decision tree classifiers when the classification examples arrive as a stream [5, 9, 7], as well as approximate computing medians and quantiles in one pass [11, 12, 2].

Density estimation is a common and useful technique to do analysis on data stream. Given a sequence of independent random variables identically drawn from a specific distribution, the density estimation problem is to construct a density function of the distribution based on the data drawn from it. Density estimation is a very important problem in numerical analysis, data mining and many scientific research fields [15]. Knowing the density distribution of a data set, we can have an idea of the distribution in the data set. Moreover, based on the knowledge of density distribution, we can find the dense or sparse area in the data set quickly; and medians and other quantiles can be easily calculated. So in many data mining applications, such as density-biased sampling and density-based clustering, density estimation is an inevitable step [3, 10]. Kernel density estimation is a widely studied nonparametric density estimation method [4], and is a good practical choice for low or medium sized data set. But it will become computationally expensive when involving large data sets. Zhang et.al. provide a method to obtain a fast kernel estimation of the density in very large data sets by constructing a CF-tree on the data [17], but they didn't consider the data stream environment. However, calculating density function over data stream has many practical applications. For example, density function is useful in clustering and outlier detection operations. It can also be used to verify whether two or more data streams are drawn from the same distribution, which is a useful and significative method in stock price analysis and scientific monitoring.

The contributions of the paper can be summarized as:

1. To the best of our knowledge, this is the earliest work of density estimation over data streams, targeting at calculating accurate density functions *online* with *limited memory* in *linear time*.

2. We bring forward a new concept of *M-Kernel*, which is the kernel function for a group of data. The introducing of *M-Kernel* enables the building of our efficient density estimation algorithm using limited main memory.

3. The application of our density estimation algorithm on two different streaming data models, i.e. complete model and windowing model, are discussed.

4. Analytical and experimental results prove the effectiveness and efficiency of our algorithm. It is a *one-pass* algorithm and needs only a *fixed-size* main memory. The running time is in *linear* with the size of the data stream. Meanwhile, the algorithm has an acceptable error rate when compared with traditional kernel density estimation. Another advantage is that it can maintain a useable model at any point in time.

The organization of the rest of the paper is as follows: in the next section (section 2) we formally define the problem that no algorithm can do density estimation over data stream efficiently. In section 3, we introduce a naive algorithm first and then evolve into a novel density estimation method which can handle data stream efficiently. We discuss the applications of the method in section 4. Section 5 includes the experimental results. And we conclude in section 6.

2 Problem Description

In this section, we briefly review the kernel density estimation method and show the problems when using it to deal with streaming data.

Given a sequence of independent random variables $x_1, x_2, \ldots$ identically distributed with density $f(x)$, the density estimation problem is to construct a sequence of estimators $\hat{f}_n(\mathbf{x}^n; x)$ of $f(x)$ based on the sample $(x_1, \ldots, x_n) \equiv \mathbf{x}^n$.

The kernel method is a widely studied nonparametric density estimation method. The equation of it on n data points is defined as:

$$\hat{f}_n(x) = \frac{1}{nh} \cdot \sum_{i=1}^{n} K(\frac{x - X_i}{h}) = \frac{1}{n} \cdot \sum_{i=1}^{n} K_h(x - X_i) \qquad (1)$$

Note that $K_h(t) = h^{-1}K(h^{-1}t)$ for $h > 0$. Clearly, $\hat{f}(x)$ is nonnegative, integrates to one, and is a density function. In Figure 1, the construction of such estimation is demonstrated. The dashed lines denote the individual standard normal shaped kernels, and the solid line the resulted estimation. Kernel density estimation can be thought of as being obtained by placing a "bump" at each point and then summing the height of each bump at each point on the X-axis. The shape of the bump is defined by a kernel function, $K(x)$, which is a unimodal, symmetric, nonnegative function that centers at zero and integrates to one. The spread of the bump is determined by a window or bandwidth, h, which controls the degree of smoothing of the estimation. Kernel density estimation has many desirable properties [17]. It is simple, no curse of dimension, no need

Proceedings of the Eighth International Conference on Database Systems for Advanced Applications (DASFAA'03)
0-7695-1895/03 $17.00 © 2003 IEEE

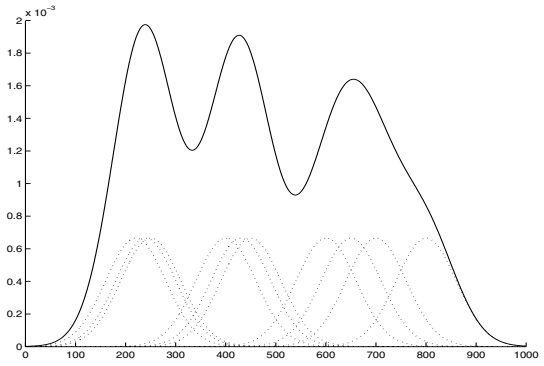

Figure 1. Construction of a fixed weight kernel density estimate(solid curve). The normal kernels are shown as the dotted lines.

to know the data range in advance, and the estimation is asymptotically unbiased, consistent in a mean-square sense, and uniformly consistent in probability.

For low to medium data sizes, kernel estimation is a good practical choice. However, if the kernel density estimation is applied to very large data sets, it becomes computationally expensive and space intensive. Suppose there are n data points in the data set, generally speaking, we need n distinct "bumps", or kernel functions in $\hat{f}(x)$ to represent them, whose space complexity is $O(n)$. Obviously, the most great drawback of the method is that it must get the n data and keep them in the memory before doing estimation. When data comes in the form of continuous data stream, the large volume and the endlessness of the data stream make it impossible to get all data in advance and keep them in the memory. That is to say, *the kernel density estimation can only deal with static data sets, it cannot handle data stream.*

3 Kernel Merging Algorithms

In this section, we first give a method which can get precise density estimation over stream data using less memory than traditional kernel density estimation methods under certain conditions. Then an approximate algorithm is proposed which can process infinite stream data in fixed-size memory. It has been shown in [16] that the choice of $K(x)$ is not critical to the result. In the following discussion, we choose normal function ($K(x) = \frac{1}{\sqrt{2\pi}} e^{\frac{-x^2}{2}}$) as the kernel function since it has good mathematical properties.

3.1 M-Kernel

Traditionally, each data is represented by a kernel function. If the sample size from the distribution is n, there will be n kernels to be calculated and stored in $\hat{f}_n(x)$, as shown in Equation 1. Therefor, if we want to handle data stream efficiently, we must find a way to reduce the amount of kernels when calculating $\hat{f}_n(x)$. A straightforward idea is to use one kernel function to represent more than one data points. Based on observation, we get the following equations:

$$K_h(x - C) + K_h(x - C) = 2K_h(x - C) \quad (2)$$

$$\rho_1 K_h(x - C) + \rho_2 K_h(x - C) = (\rho_1 + \rho_2) K_h(x - C) \quad (3)$$

The equations convey that if two or more data points have same data value, we can use one large kernel to represent them. A weight value is given to each kernel function to indicate how many data points it will represent. By doing this, the amount of kernels will be smaller than that of data points. We call a kernel representing only one data point a *simple kernel* and a kernel representing more than one data point an *M-Kernel*. An *M-Kernel* has three parameters: weight ρ, mean X_i and bandwidth h. It stands for a kernel function like $\frac{\rho}{h} K(\frac{x - X_i}{h})$.

3.2 A Naive Algorithm for Density Estimation

Algorithm 1 NaiveDensityEstimate(X)

Input: data stream $X = \{x_1, x_2, \ldots, x_n, \ldots\}$
Output: a density function $\hat{f}(x)$ over data stream X

```
1:  Initiate (Kernel_List);
2:  Count=0;
3:  while stream not end do
4:     Read x_i from the stream;
5:     Count++;
6:     if Kernel_List.FindKernelByMean(x_i) == TRUE then
7:        CurKernel=Kernel_List.GetKernelByMean(x_i);
8:        CurKernel.Weight++;
9:     else
10:       MKernel= new Kernel(x_i,1);
11:       Kernel_List.InsertKernel(MKernel);
12:    end if
13: end while
14: for i = 0 to Count do
15:    f̂(x) += Kernel_List.GetKernelByOrder(i);
16: end for
```

Based on the Equation 2 and 3 given above, we can construct a naive algorithm which is shown in Algorithm 1. In the algorithm, simple kernels having same value are merged into an *M-Kernel* (line 6∼8). Since the *M-Kernel* here is equal to the sum of the *simple kernel*s, the result of this algorithm will be exactly the same as that of traditional kernel density estimation. If there are lots of data points with same value in the data set, the algorithm will take much smaller memory than traditional method. However the memory used by the algorithm is very sensitive to the distribution of the data set. And in the extreme situation (no two points in the data set have same value), it will be equal to the traditional method.

Proceedings of the Eighth International Conference on Database Systems for Advanced Applications (DASFAA'03)
0-7695-1895/03 $17.00 © 2003 IEEE

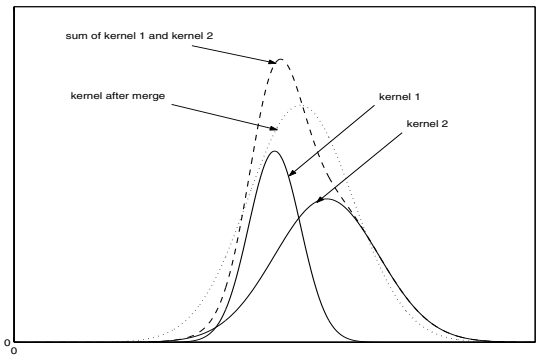

Figure 2. Kernel merging operation

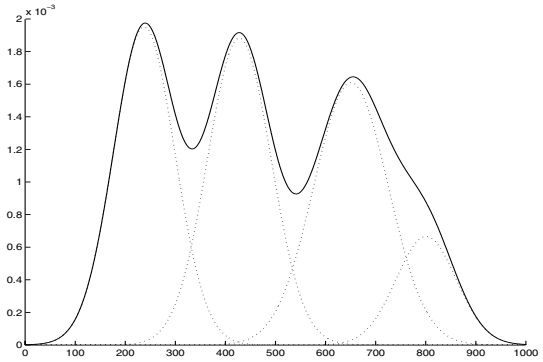

Figure 3. Construction of a distinct weight kernel density estimate (solid curve). The *M-Kernel* are shown as the dotted lines

3.3 Approximate Kernel Merging

To process data stream efficiently, only fixed amount of memory can be used by the algorithm. If it could not be achieved by only merging points with same data value, we must use some approximate method to reduce the size of memory used by the algorithm.

As we can see in Figure 2, two kernels which are similar but do not have same value can also be merged into an *M-Kernel*. We formulize the kernel merging as below.

$$\rho_i K_{h_i}(x - X_i) + \rho_j K_{h_j}(x - X_j) \approx (\rho_i + \rho_j) K_{h_m^*}(x - X_m^*) \quad (4)$$

when $|X_i - X_j| < \varepsilon_1$ and $|h_i - h_j| < \varepsilon_2$

The left of the equation stands for the sum of two kernels placed at point X_i and X_j with bandwidth h_i and h_j respectively, while the right of the equation stands for the *M-Kernel* placed at point X_m^* with bandwidth h_m^*.

Since most of time we can not find a normal kernel absolutely tally with the sum of two kernels, there will be some accuracy lost in the kernel merging. We define the accuracy lost in the merge as m_cost, which is the L^1 distance between dotted line and dashed line in Figure 2.

$$\begin{aligned} m\_cost &= g(X_m^*, h_m^*) \\ &= \int \big|\rho_i K_{h_i}(x - X_i) + \rho_j K_{h_j}(x - X_j) \\ &\quad -(\rho_i + \rho_j) K_{h_m^*}(x - X_m^*)\big| dx \end{aligned} \quad (5)$$

And we try to minimize m_cost in every merge. Because $g(X, h)$ is a bounded function, we can find the global minimum point of the function. Suppose the value of x and y are X_m^* and h_m^* respectively when we get the minimum point of $g(x, y)$, then we use X_m^* and h_m^* as the parameters of the merged kernel. That is to say, when $x = X_m^*$ and $y = h_m^*$, the m_cost is minimized and we can use $(\rho_1 + \rho_2) \cdot K_{h_m^*}(x - X_m^*)$ to replace the sum of the two individual kernel functions.

For we do not know the derivatives of the $g(X, h)$ in Equation 5, we use *downhill simplex method* to find the minimum of the function. The *downhill simplex method* is due to Nelder and Mead [13]. It is a fast multidimensional minimization method whose computer burden is small.

With Equation 4, after $(n - m)$ times of merging, we get m *M-Kernel* with parameter $X_i^*, h_i^*, \rho_i^* (i = 1 \ldots m)$ from $X_1, \ldots, X_n$. Then the density estimator can be written as

$$\hat{f}_n^*(x) = \frac{1}{n} \cdot \sum_{j=1}^{m} \frac{\rho_j^*}{h_j^*} \cdot K(\frac{x - X_j^*}{h_j^*}) \quad where \quad \sum_{j=1}^{m} \rho_j^* = n \quad (6)$$

Compared to the curves in Figure 1, in Figure 3, we merge some kernel functions into *M-Kernel* (the dotted line). And there are only four *M-Kernel*s left after merging, so the merging operation can significantly reduce the memory used by the algorithm. However, the final density curve (the solid line) is very close to the one synthesized by ten kernel functions. The difference between the two curves is the accuracy lost when we do the merge, which is the *absolute error* of our algorithm. Our experiments will show that the difference between the two density curves is very small, and can be controlled by some parameters.

3.4 An Approximate Algorithm for Density Estimation

Based on Equation 4 and the naive algorithm we mentioned in section 3.2, we propose an approximate density estimation algorithm which is shown in Algorithm 2. In the course of the processing, an *m*-entry buffer is maintained in main memory. It contains an array of elements with the format $\langle X^*, h^*, \rho^*, m\_cost \rangle$ and sorted by X^*. The element

Proceedings of the Eighth International Conference on Database Systems for Advanced Applications (DASFAA'03)
0-7695-1895/03 $17.00 © 2003 IEEE

denotes that kernel $K(x)$ have parameters X^*, bandwidth h^*, and weight ρ^*, which means the kernel function is like $\frac{\rho^*}{h^*} \cdot K(\frac{x-X^*}{h^*})$, and m_cost is the merging cost with next kernel in the buffer. An observation is, kernels with small distance in value X^* will have smaller m_cost than those with larger distance in value X^*. So we only need to calculate the merging cost of neighboring kernels in the buffer since it will be smaller than that of the kernels which are not neighboring in the buffer. When a new data comes, an element will be generated and inserted into the buffer. If the buffer is full, an element with the smallest m_cost will be picked out. Suppose that it is the ith element in the buffer, then the ith element and $(i+1)$th element will be merged into a new element. Then a buffer entry will be freed. And the procedure will go on to deal with next data in the stream.

Line 2 to Line 16 is the main loop of the algorithm. Every time a new data comes, we generate an element for it and insert it into the buffer (Line5∼6) ordered by μ. If the buffer is full, we need to find a pair of kernels with the lowest merging cost and merge them (Line9∼13). Line 7 and line 15 are very important in the algorithm, for they take charge of maintaining the list of merging cost. At most time, only one element in the buffer changed, so at most two merging costs (in 1-dimensional) need to be updated.

Algorithm 2 ApproximateDensityEstimate(X)

Input: data stream $X = \{x_1, x_2, \ldots, x_n, \ldots\}$
Output: a density function $\hat{f}(x)$ over data stream X

```
 1: Initiate (Kernel_List);
 2: Kernel_List.SetBufferSize(m);
 3: while stream not end do
 4:    Read x_i from the stream;
 5:    CurKernel= new Kernel(x_i,1);
 6:    Kernel_List.InsertKernel(CurKernel);
 7:    Kernel_List.RecalculateMergeCost();
 8:    if Kernel_List.IsFull() then
 9:       TmpKernel=Kernel_List.FindMinimalMergePair();
10:       MKenrel=MergeKernel(TmpKernel, TmpKernel.NextKernel);
11:       Kernel_List.DeleteKernel(TmpKernel.NextKernel);
12:       Kernel_List.DeleteKernel(TmpKernel);
13:       Kernel_List.InsertKernel(MKernel);
14:    end if
15:    Kernel_List.RecalculateMergeCost();
16: end while
17: for i = 0 to Kernel_List.Count() do
18:    f̂(x) += Kernel_List.GetKernelByOrder(i);
19: end for
```

3.5 Algorithm Analysis

It is easy to see that during the procedure, the buffer size is kept constant. So the space complexity of the algorithm is $O(m)$, where m is irrespective with the size of the data stream. And for every record processed, we only have to calculate the merging cost with its neighboring kernel functions, and do the merge if the list is full. So, we can see that the computational cost of each record is within a constant value. That is to say, the total calculating cost is in linear with the size of the data stream. We have detailed analysis about it in the experimental part.

4 Discussion

By default, the algorithm is running under the *complete model* in which all data in the stream are of equal importance. The output of the algorithm is the density function over the whole data stream. So newly-coming data will have same weight as old data which have already been summarize into kernels.

Another stream model is *window model*, in which we take more emphasis on recent data than the old data. Most of the time, we only care about recent N data in the data stream. With some adaption, our algorithm can handle data stream in *window model* too. A "fadeout" function is used to decrease the contribution of the old data in the data stream, so that recent data will receive more attention. The function can be implemented differently according to users' requirement. For example, one fadeout function, which decreases the weight of old data linearly, may work in the following way. Suppose at one time, there are m kernels in the buffer. When a new data (x_{n+1}) comes, we will decrease the weight of the original m kernels by function, for example $(1-a\%)$, and assign corresponding weight to the new coming data to keep the sum of the weight of all kernels equal to the amount of the data. Equation 8 illustrates how to use the fadeout function.

$$\hat{f}_n^*(x) = \frac{1}{n} \cdot \sum_{j=1}^{m} \rho_j^* K_{h_j^*}(x - X_j^*) \tag{7}$$

$$\hat{f}_{n+1}^*(x) = (1-a\%) \cdot \hat{f}_n^*(x) + (1+a\%) \cdot K_h(x - X_{n+1}) \tag{8}$$

In this way, new data will be drawn more emphasis than the old data. Since we are discussing how to handle data stream efficiently in this paper, we focus on complete model and omit the details of fadeout functions here.

5 Experiments

The algorithm is implemented in C++. The program runs on a PC workstation with 1.4GHz Pentium IV processor and 256Mb RAM using Windows 2000 Server.

We construct four large synthetic data sets and carry out a series of experiments on them to test the performance of our algorithm. The goal of the experiments is to evaluate the accuracy of the proposed algorithm, and to prove that the time cost by per record is within a small constant value, which means the algorithm can output the results in linear

Proceedings of the Eighth International Conference on Database Systems for Advanced Applications (DASFAA'03)
0-7695-1895/03 $17.00 © 2003 IEEE

Table 1. Test Data sets

Distr.	Cumulative Distribution Function
Distr. I	$F_1(x) = N(1200, 200)$
Distr. II	$F_2(x) = 0.4N(200, 100) + 0.3N(800, 200) + 0.2N(1900, 50) + 0.1N(1100, 50)$
Distr. III	$F_3(x) = \sum_{i=1}^{10} 0.1N(i * 300 - 100, 50)$

Table 2. Test Data Streams

Data Streams	Size	Distribution	Type
Data Stream 1	100,000	Distribution I	Random
Data Stream 2	100,000	Distribution II	Random
Data Stream 3	100,000	Distribution III	Random
Data Stream 4	100,000	Distribution II	Ordered

time. We also do experiments to show the relationship between the buffer size and the precision of the result. Additionally, experiments show the ability of the algorithm to output a usable model at any point of the processing.

5.1 Experimental Setup

Some synthesized data streams are generated to test our algorithm. First, we choose three distribution functions, shown in Table 1. Then, data streams are drawn randomly and independently from these distributions. Table 2 shows that *data stream* $1 \sim 3$ are drawn randomly from distributions $I \sim III$, accordingly. The *data stream* 4 is also drawn from distribution II, but it is sorted by the value.

The classical kernel density estimation can be regarded as the best nonparametric density estimation algorithm. So we take the result produced by kernel density estimation as the base line to measure the accuracy of our algorithm. The difference between the outputs of the two algorithms can be regarded as the error of our algorithm.

The results of kernel density estimation (KDE) that will be used as the criteria are calculated by Beardah's program [1] implemented in MATLAB.

The accuracy of our algorithm can be evaluated by *absolute error*$= \int |\hat{f}(x) - \tilde{f}(x)|dx$, where $\hat{f}(x)$ is the kernel density estimation result on static data set and $\tilde{f}(x)$ is the result of our algorithm over the corresponding data stream.

5.2 Experimental Results

The experiments are divided into three parts:

- Firstly, the test of the running time on data streams of different sizes reveals that the running time of our algorithm is in linear with the data size.

- Secondly, we study the effect of the buffer size on the precision of the result, which shows that the error rates decrease with the increasing of the buffer size. And we can make the conclusion that, given a large enough buffer size, the accuracy of our algorithm will be rather high.

- Thirdly, we demonstrate the ability of our algorithm to output the result at any point of the processing.

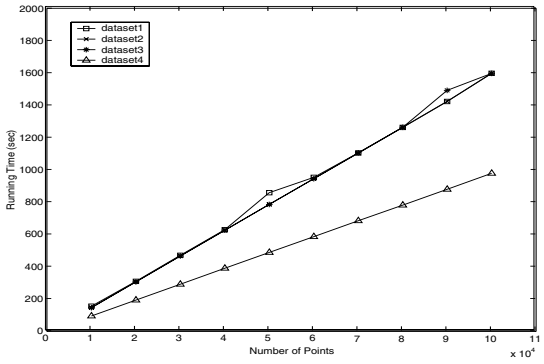

Figure 4. Running Time vs Stream Size

Running Time: The crucial characteristic of stream algorithm is that it must keep up with the continuously coming data stream, that is to say, the running time must be in linear with the data size. We test data streams of different sizes and record corresponding running time of the algorithm. Figure 4 shows how the running time scales up on the four data sets. We can see that as the size of the data sets increases, the running time increases linearly. This figure also exhibits that one curve has a distinctly low slope compared with the other three curves. We notice that this curve represents the data stream in which data increases strictly. Apparently, a sorted data set will consume less time while being processed, because the sorting procedure in the algorithm will run faster on a sorted data.

Buffer Size: The buffer size is an important parameter in this algorithm; it has large infection on the accuracy of estimation result. In this experiment, on data stream 2, we range the buffer size from 500 to 2,000 to check the fluctuation of the accuracy rate. Table 3 shows the result. It can be known from the table that when the buffer size increases, the error rate decreases, while the running time keeps almost consistent. And obviously, if the buffer size equals to the size of the data set, our algorithm degenerates to traditional kernel density estimation, and the error rate drops to zero.

Incremental: One of the main characteristics of stream data is that it may be infinitive. So the ability to output result at any point of the processing is of great importance to data stream algorithms. To test this capability, we run our program on data stream 2 and 4. The two data streams

Proceedings of the Eighth International Conference on Database Systems for Advanced Applications (DASFAA'03)
0-7695-1895/03 $17.00 © 2003 IEEE

Table 3. Absolute error & Running Time at different Buffer Size

Buffer Size	Absolute error	Running Time(sec)
500	2.50e-002	1603.51
600	1.70e-002	1603.32
700	1.44e-002	1603.98
800	1.09e-002	1603.74
900	8.52e-003	1603.83
1,000	7.16e-003	1608.60
1,100	4.58e-003	1606.01
1,200	4.82e-003	1610.23
1,300	3.98e-003	1602.69
1,400	3.09e-003	1603.19
1,500	2.63e-003	1607.64
1,600	2.45e-003	1602.91
1,700	2.05e-003	1604.00
1,800	1.87e-003	1606.46
1,900	1.50e-003	1629.98
2,000	1.33e-003	1637.92

are drawn from the same distribution, stream 4 has been sorted while stream 2 has not. The two data streams both have 100k points of data and we output the results after every 10k points of data are processed. For each data steam, we get 10 small figures, shown in Figures 5 and 6 respectively. The dotted curves represent the density estimation made over the whole data set and the solid curves represent the density estimation got in the midst of processing. In both figures, we can see that the medial results accord with the final result, but they approach it in different ways. Since data stream 2 is not sorted, the data points are distributed randomly all through the data set. We can get a rough outline of the final density curve at any output point. The more data we get, the closer the medial result is to the final result, but it is not the case to the sorted data stream 4. In the middle of the processing, only part of the data is available. And because it has been sorted, it concentrates on one area of the data set. So the density curve we get at each interval is only part of the final curve. But the part is very precise compared with the final result, since nearly all data needed to generate the part have been obtained.

6 Conclusions and Future Work

Along with the appearance of more and more continuous data in real-life applications, efficient mining on data streams become a challenge for existing data mining algorithms, which is partially because of the high cost on both storage and time of distribution computation. This paper introduces an effective and efficient algorithm calculating density functions over data streams based on *M-Kernel* merging. Since only fixed size memory is needed, regardless of the amount of data, it is suitable for streaming data. The discussion and analysis show that our method is flexible for different streaming data models. Furthermore, it is easy to build high-level data mining algorithms on our density estimation algorithm. Analysis and experiments prove that our algorithm is at the speed data arrival and only need one scan on the data stream. The accuracy of the result is comparable to that of the kernel density estimation. And this method is superior than the existent methods for it can output a useable model at any time of the processing.

The future work includes the integration of density estimation with more data mining applications. Another improvement we are going to make is to incorporate the bandwidth-decision technique into our algorithm so that the bandwidth can be self-adapted to data streams.

7 Acknowledgments

The authors would like to thank Dr. Rajeev Rastogi and Chaofeng Sha for their valuable advice.

References

[1] C. C. Beardah and M. J. Baxter. Matlab routines for kernel density estimation and the graphical representation of archaeological data. Technical report, Department of Mathematics, Statistics and Operational Research, The Nottingham Trent University, http://science.ntu.ac.uk/msor/ccb/densest.html, 1995.

[2] F. Chen, D. Lambert, and J. C. Pinheiro. Incremental quantile estimation for massive tracking. In *Proc. of Int'l Conf. on Knowledge Discovery and Data Mining (KDD'2000)*, pages 579–522, 2000.

[3] Y. Cheng. Mean shift, mide seeking, and clustering. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 17(8):790–799, August 1995.

[4] D. Comaniciu and P. Meer. Distribution free decomposition of multivariate data. In *Int'l Workshop on Statistical Techniques in Pattern Recognition*, 1998.

[5] P. Domingos and G. Hulten. Mining high-speed data streams. In *Proc. of Int'l Conf. on Knowledge Discovery and Data Mining (KDD'2000)*, pages 71–80, 2000.

[6] P. Domingos and G. Hulten. Catching up with the data: Research issues in mining data streams. In *Workshop on Research Issues in Data Mining and Knowledge Discovery*, 2001.

[7] P. Domingos and G. Hulten. Learning from infinite data in finite time. In *Advances in Neural Information Processing Systems*, 2002.

[8] S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams. In *Proc. of Symp. on Foundations of Computer Science (FOCS'2000)*, pages 359–366, 2000.

[9] G. Hulten, L. Spencer, and P. Domingos. Mining time-changing data streams. In *Proc. of Int'l Conf. on Knowledge Discovery and Data Mining (KDD'2001)*, pages 97–106, 2001.

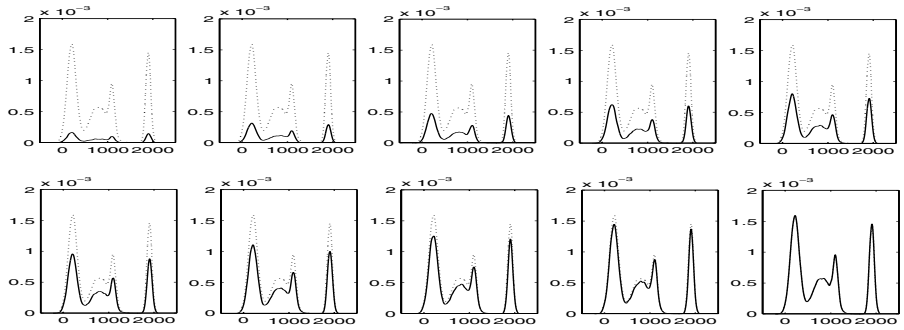

Figure 5. The medial outputs of data stream 2

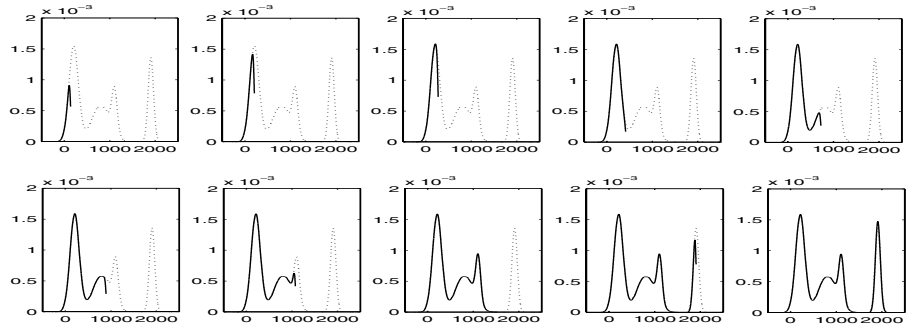

Figure 6. The medial outputs of data stream 4

[10] G. Kollios, D. Gunopoulos, N. Koudas, and S. Berchtold. An efficient approximation scheme for data mining tasks. In *Proc. of Int'l Conf. on Data Engineering (ICDE'2001)*, pages 453–462, 2001.

[11] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Approximate medians and other quantiles in one pass with limited memory. In *Proc. of ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD'98)*, pages 426–435, 1998.

[12] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Random sampling techniques for space efficient online computation of order statistics of large datasets. In *Proc. of ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD'99)*, pages 251–262, 1999.

[13] J. A. Nelder and R. Mead. A simplex method for function minimization. *The Computer Journal*, 7(4):308–313, January 1965.

[14] L. O'Callaghan, N. Mishra, A. Meyerson, and S. Guha. Streaming-data algorithms for high-quality clustering. In *Proc. of Int'l Conf. on Data Engineering (ICDE'2002)*, 2002.

[15] S. R. Sain. *Adaptive Kernel Density Estimation*. PhD thesis, Rice University, August 1994.

[16] M. P. Wand and M. C. Jones. *Kernel Smoothing*. Chapman & Hall, London, UK., 1995.

[17] T. Zhang, R. Ramakrishnan, and M. Livny. Fast density estimation using cf-kernel for very large databases. In *Proc. of Int'l Conf. on Knowledge Discovery and Data Mining (KDD'99)*, pages 312–316, 1999.

Proceedings of the Eighth International Conference on Database Systems for Advanced Applications (DASFAA'03)
0-7695-1895/03 $17.00 © 2003 IEEE