Density Estimation Over Data Stream*

Aoying Zhou
Dept. of Computer Science,
Fudan University
220 Handan Rd.
Shanghai, 200433, P.R. China
ayzhou@fudan.edu.cn

Zhiyuan Cai
Dept. of Computer Science,
Fudan University
220 Handan Rd.
Shanghai, 200433, P.R. China
zycai@fudan.edu.cn

Li Wei
Dept. of Computer Science,
Fudan University
220 Handan Rd.
Shanghai, 200433, P.R. China
lwei@fudan.edu.cn

ABSTRACT

Density estimation is an important but costly operation for applications that need to know the distribution of a data set. Moreover, when the data comes as a stream, traditional density estimation methods cannot cope with it efficiently. In this paper, we examined the problem of computing density function over data streams and developed a novel method to solve it. A new concept *M-Kernel* is used in our algorithm, and it is of the following characteristics: (1) the running time is in linear with the data size, (2) it can keep the whole computing in limited size of memory, (3) its accuracy is comparable to the traditional methods, (4) a useable density model could be available at any time during the processing, (5) it is flexible and can suit with different stream models. Analytical and experimental results showed the efficiency of the proposed algorithm.

Keywords

Data Stream, Kernel density estimation

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, transaction of banks, call records of telecommunications company, hit logs of web server are all this 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 a model to access large data sets stored in secondary memory where performance requirements necessitate access with linear scans. In most cases, comparing to the size of total data in auxiliary storage, the size of main memory is such small that every time only a small portion of data can be load in memory to process, and under the time restrict only one scan is allowed to the data. Therefore, despite all our efforts in scaling up data mining algorithms, they still cannot handle such data efficiently.

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 the 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 following:

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.

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

Recent proposed techniques include clustering algorithms when the objects arrive as a stream [8, 14], computing de-

*(Produces the permission block, copyright information and page numbering). For use with ACM_PROC_ARTICLE-SP.CLS V2.0. Supported by ACM.

cision tree classifiers when the classification examples arrive as a stream [5, 10, 7], as well as approximate computing medians and quantiles in one pass [12, 13, 2].

Another common but useful technique used on data stream is **Density estimation**. 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 [9, 15]. By 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, density-based clustering, density estimation is an inevitable step to them [3, 11]. Kernel density estimation is a widely studied nonparametric density estimation method [4]. But it becomes 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 construct a CF-tree on the data [16]. Calculating density function over data stream has many practical applications. A straightforward one is that it can be used to verify whether two or more data streams are drawn from the same distribution.

1.1 Our Contribution

The contributions of the paper can be summarized as:

1. To the best of our knowledge, it is the earliest work of density estimate over data stream.

2. We bring forward a new concept of *M-Kernel*, which solves the incompatibility of limited main memory and continuously coming data. And an algorithm is proposed based on it to solve density estimation problem over data stream.

3. 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.

4. What we provide is a basic notion to handle data stream, which can be extended and integrated to many data mining applications such as density-biased sampling.

1.2 Paper Organization

The organization of the rest of the paper is as follows: in the next section (Section 2) we formally define the problem. A new density estimation method that can handle data stream is presented in section 3. Section 4 includes the experimental results. And we conclude in section 5.

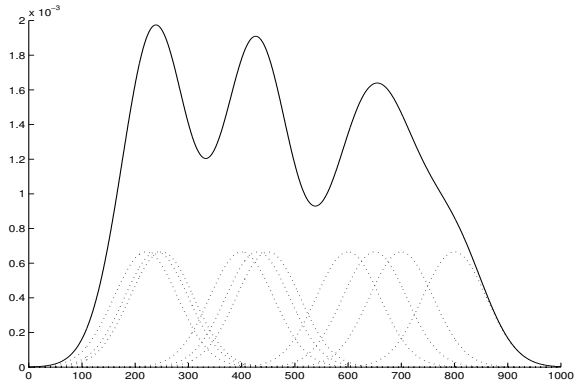

Figure 1: Construction of a fixed weight kernel density estimate(solid curve). The normal kernels are shown as the dotted lines.

2. PROBLEM DESCRIPTION

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

Given a sequence of independent random variables $x_1, x_2, \ldots$ identically distributed with density f, 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 based on n data points is defined as:

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

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 estimates is demonstrated. The dashed lines denote the individual standard normal shaped kernels, and the solid line the resulting estimate. 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 taken to be 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 [16]. It is simple, no curse of dimension, no need to know the data range in advance, and the estimate 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. Because there are n distinct "bumps", or kernel functions in $\hat{f}(x)$, generally speaking, it needs $O(n)$ space to store the n kernel functions. If we want to calculate the density value at a specific point α, it needs

to scan all n kernel functions to compute $\hat{f}(\alpha)$. And the most great drawback of the method is that it must get the n data before begin to estimate the density function. That is to say, *the kernel density estimation can only deal with static data sets, it cannot handle data stream.*

3. DEAL WITH DATA STREAM

In this section, we first give two observations on classical kernel density estimation method, and then present a new density estimation method, which can calculate density function over data stream efficiently. The algorithm seems like classical kernel density estimation, but there is one great difference between them, that is, the efficiency. It only uses a fix-sized main memory, which is irrespective of the total number of records in the stream, all through the processing. And the time complexity is in linear with the size of the data stream.

3.1 Incremental Density Estimation

Traditionally, $\hat{f}_n(x)(n = 1, 2, \ldots)$ is calculated independently from random variables $X_1, X_2, \ldots$. Here, we want to calculate $\hat{f}_{t+1}(x)$ based on $\hat{f}_t(x)$ and the random variable X_{t+1}.

Observation 1. The classical kernel density estimation can be calculated in an incremental manner.

If the main memory is unlimited, then the density function can be calculated in an incremental way. That is to say, the data stream can be scanned only once. Naively, we can estimate the density function in such an incremental manner.

$$\begin{aligned}
\hat{f}_{t+1}(x) &= \frac{1}{t+1}\sum_{i=1}^{t+1} K_h(\frac{x-X_i}{h}) \\
&= \frac{1}{t+1}\left(\sum_{i=1}^{t} K_h(\frac{x-X_i}{h}) + K_h(\frac{x-X_{t+1}}{h})\right) \\
&= \frac{1}{t+1}\left(t\hat{f}_t(x) + K_h(\frac{x-X_{t+1}}{h})\right) \\
&= \frac{t}{t+1}\hat{f}_t(x) + \frac{1}{t+1}K_h(\frac{x-X_{t+1}}{h}) \qquad (2)
\end{aligned}$$

Obviously, in equation (2), $\hat{f}_{t+1}(x)$ can be calculated from $\hat{f}_t(x)$ and X_{t+1}. So when a new data record comes, only some predetermined calculations are needed, while we need to keep t kernels in main memory when processing the $t+1$ data record.

3.2 Kernel Merging

When the amount of data increases, what we need to record expands accordingly and it will be difficult to keep them in the limited main memory when the data is tremendous. Furthermore, it is also computationally expensive since totally there are still n terms in the $\hat{f}_n(x)$, and for the density on a specific point α, it still needs to scan all n kernel functions to compute $\hat{f}(\alpha)$.

How to solve the conflict of increasing data and limited memory? We observe that with some acceptable information lost we can summary the kernel functions, and keep them all in the main memory.

Traditionally, each data is represented by a kernel function. If the sample size from the distribution is n then there are n kernels to be calculated and stored in $\hat{f}_n(x)$. Now we try to reduce it and fix the amount of kernels when calculating $\hat{f}_n(x)$. A weight value is given to the kernel function, so it can represent more than one data. Then the amount of kernels can be smaller than the amount of data points. We call kernel represent only one data *simple kernel* and kernel represent more than one data *M-Kernel*.

Observation 2. The sum of two kernel functions can be approximated by a large kernel function (*M-Kernel*) whose weight equates to the amount of data it will represented.

Based on the observation 2, two simple kernels (i and j) can be represented by one *M-Kernel* (m) with weight two. That is,

$$\frac{1}{h_i}K(\frac{x-X_i}{h_i}) + \frac{1}{h_j}K(\frac{x-X_j}{h_j}) \cong \frac{\mathbf{2}}{h_m^*}K(\frac{x-X_m^*}{h_m^*}) \qquad (3)$$

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. At most time, we cannot find a kernel with parameter X_m^* and h_m^* to make the two side of Equation 3 absolutely equivalent. But analysis and experiments will show the difference between them is very small.

More generally, two *M-Kernel*s (s and t) can also be represented by a single *M-Kernel* (u).

$$\frac{p_s^*}{h_s^*}K(\frac{x-X_s^*}{h_s^*}) + \frac{p_t^*}{h_t^*}K(\frac{x-X_t^*}{h_t^*}) \cong \frac{p_s^*+p_t^*}{h_u^*}K(\frac{x-X_u^*}{h_u^*}) \qquad (4)$$

With Equations 3 and 4, after $(n-m)$ times of merging, we get m M-kernel with parameter $X_i^*, h_i^*, p_i^*\ i = (1 \ldots m)$ from $X_1, \ldots, X_n$. Then the estimator can be written as

$$\hat{f}_n^*(x) = \frac{1}{n}\sum_{j=1}^{m}\frac{p_j^*}{h_j^*}K(\frac{x-X_j^*}{h_j^*}) \qquad (5)$$

where

$$\sum_{j=1}^{m} p_j^* = n \qquad (6)$$

Figure 2 illustrates the kernel merging operation. Kernel 1 and kernel 2 are two different kernels (simple kernel or M-kernel), and we want to find a kernel (M-kernel) to represent the two kernel functions. Intuitively, it must be very close to the dashed-line in the figure, which is the sum of the two kernel functions.

In Figure 3, we merge some kernel functions into M-kernel (the dotted line). Compared to Figure 1, there are only four M-Kernel left after merging, so the merging operation

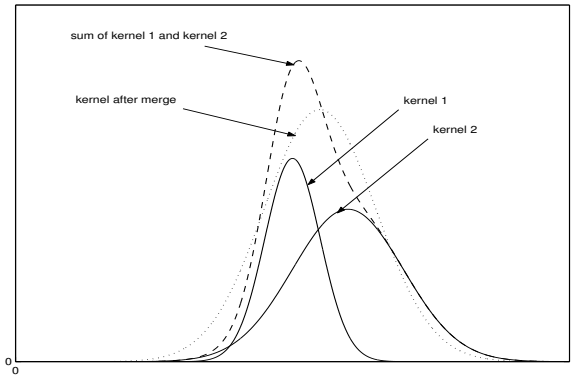

Figure 2: Kernel merging operation

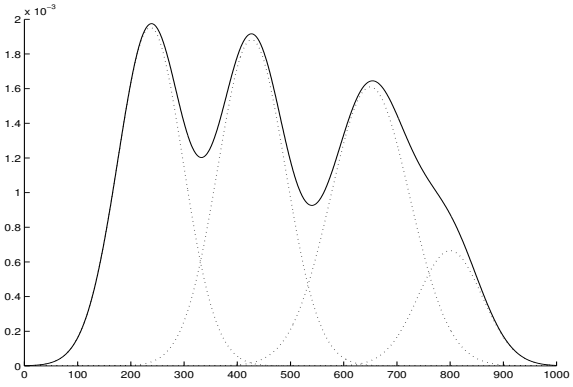

Figure 3: Construction of a distinct weight kernel density estimate (solid curve). The M-kernel are shown as the dotted lines

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 merging, which we call it *merging cost*. Our experiments will show that the difference between the two density curves is very small, and can be controlled by some parameters.

3.3 Applied to Data Stream

Based on the two observations we proposed above, by using the incremental and merging techniques, now we can extend the classical kernel density estimation to cope with data stream. We keep a fixed-size kernel function list in the main memory. Every time a new data record comes, the density function, which we put on the data point, is added to the kernel function list. When the list is full, we merge some kernel functions having the lowest merging cost in the list to empty some entries. We can see that the memory used by the algorithm is fixed, and it only requires short constant time to process each record in the stream. At any time of the processing, we can synthesize the kernel function by summing all the kernel functions in the list. Because the amount of kernel functions in the list is fixed, the synthesization can also be done in constant time.

3.4 Algorithm

The detailed algorithm is shown in Figure 4. In the course of the processing, an m-entry buffer is maintained in main memory. It contains an array of elements with the format $\langle \mu, \sigma, \rho, m\_cost \rangle$. The element denotes that kernel $K(x)$ have parameters μ, bandwidth σ, and weight ρ, which means the kernel function is like $\frac{\rho}{\sigma} K(\frac{x-\mu}{\sigma})$, and m_cost is the merging cost with next kernel in the buffer. A heuristic method is applied here, that is, kernels with small distance in value μ will have smaller merging cost than kernels with large distance in value μ. So the buffer is sorted by μ, and we just need to calculate the merging cost of neighboring kernels in the buffer since it will 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 17 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 (Line4∼5). Then we sort the buffer and count the merging value of the new data (Line6∼7). If the buffer is full, we need to find a pair of kernels with the lowest merging cost and merge them (Line9∼14). Line 7 and line 14 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.

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

```
ALGORITHM DensityOverStream ()
INPUT:  Data stream (x_1, x_2, ...)
OUTPUT:  Density function f̂(x)
PROCEDURE

 1. Initiate a buffer with m entries;
 2. While the stream is not end
 3.    Read a new data point x_i from the stream;
 4.    Fill the element ⟨μ, σ, ρ, m_cost⟩ with
       ⟨x_i, h, 1, max_cost⟩
 5.    Insert the element into the buffer
 6.    Sort the buffer by μ
 7.    Update the column m_cost.
 8.    If the buffer is full
 9.       Find the entry with the smallest m_cost
       value in the buffer;
10.       Merge the two entries into one
11.       Insert the merged entry into the buffer
12.       Free an entry
13.       Sort the buffer by μ
14.       Update the column m_cost.
15.    End If
16.    Output the buffer and synthesize the
       density function f̂(x) if needed
17. End While

ENDPROCEDURE
```

Figure 4: Stream algorithm

stream. And for every record processed, we only have to calculate the merging cost with its neighboring kernel functions, and do the merging 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 detail analysis about it in our experiments.

We use some merging methods to reduce the memory requirement of the algorithm, but the cost is the calculation results would definitely have variances from the previous results.

The cost of a single merge operation, which is shown in Equation 4 is

$$\int_{-\infty}^{\infty} (\frac{p_s^*}{h_s^*}K(\frac{x-X_s^*}{h_s^*})+\frac{p_t^*}{h_t^*}K(\frac{x-X_t^*}{h_t^*})-\frac{p_s^*+p_t^*}{h_u^*}K(\frac{x-X_u^*}{h_u^*}))^2dx \tag{7}$$

and the total cost of the estimator (Equation 5) is

$$\int_{-\infty}^{\infty} (\hat{f}_n(x) - \hat{f}_n^*(x))^2 dx \tag{8}$$

where $\hat{f}_n(x)$ is the traditional estimation result and $\hat{f}_n^*(x)$ is the result.

3.6 Discussion

The algorithm can run in two models: the *complete* model and the *window* model. In complete model, all data in the stream are of equal interest. While in window model, we take more emphasis on the recent data than the old data. With a "fadeout" function to decrease the contribution of the old data to the result, the complete model can be easily extended to window model. The function can be implemented differently according to users' requirement. For example, one fadeout function, which decreases the weight of old data gradually, may work in following way. Suppose at one time, there are m kernels in the buffer. Then a new data comes, and we decrease the weight of the original m kernels by a small proportion, say 0.1%, 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. 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.

The algorithm can cope with multi-dimensional data as well as 1-dimensional data. In traditional kernel density estimation, multivariate estimator has the same form with univariate estimator. Because our algorithm is based on the result of traditional kernel estimate, so it can be easily generalized to any dimension.

4. 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.

Table 1: Test Data sets

Distribution	Density function
Distribution I	$f_1(x) = N(1200, 200)$
Distribution II	$f_2(x) = 0.4N(200, 100) + 0.3N(800, 200)$ $+0.2N(1900, 50) + 0.1N(1100, 50)$
Distribution 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

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 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.

4.1 Experimental Setup

Some synthesized data streams are generated to test our algorithm. First, we choose three distribution functions, shown in Table 1. The distribution of these three density functions are illustrated in Figures 5, 6 and 7, respectively. Next, data streams are drawn randomly and independently from these distributions. Table 2 shows that. Data Stream 1-3 are drawn randomly from distribution I-III, accordingly. Data Stream 4 is also drawn from distribution II, but it is ordered 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.

4.2 Experimental Results

The experiments are divided into three parts:

- Firstly, 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.

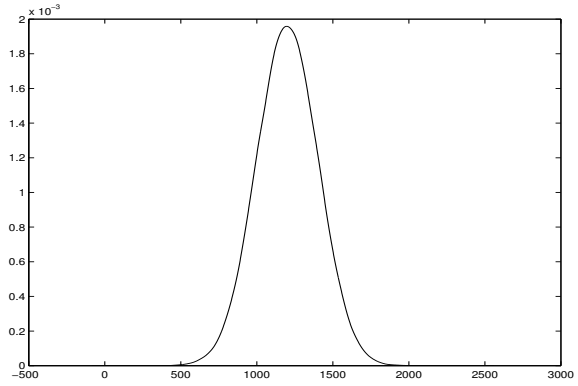

Figure 5: Distribution I

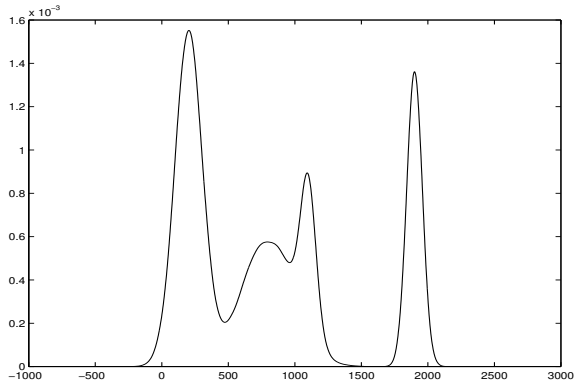

Figure 6: Distribution II

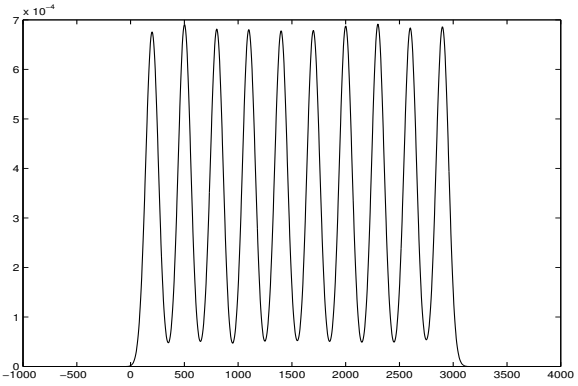

Figure 7: Distribution III

Table 3: Running Time at different Stream Size

Stream Size	Running Time (sec)			
	Stream 1	Stream 2	Stream 3	Stream 4
10,000	147.482	156.064	147.442	95.607
20,000	306.851	310.005	307.381	194.309
30,000	466.881	471.277	467.051	292.370
40,000	626.721	629.774	625.589	391.172
50,000	786.210	858.844	786.791	489.013
60,000	944.968	953.671	945.098	586.483
70,000	1105.369	1104.428	1103.426	684.964
80,000	1263.476	1262.885	1263.576	781.864
90,000	1491.815	1423.236	1423.777	879.684
100,000	1597.176	1597.677	1596.936	978.476

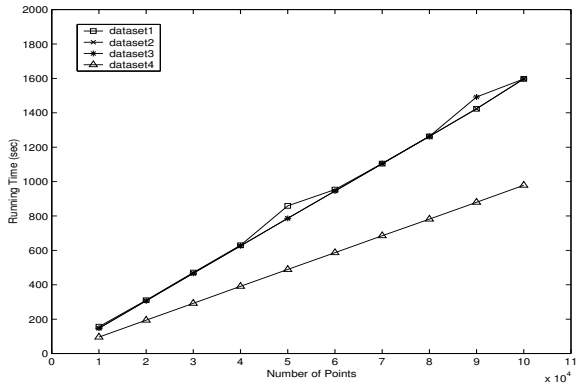

Figure 8: Running Time vs Stream Size

- Secondly, studying the effect of the buffer size on the precision of the result, this 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.

Running Time: The crucial character 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, which is shown in Table 3. Figure 8 shows how the running times scale 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, the buffer size used by the algorithm is ranged from 500 to 2,000

Table 4: Absolute error & Running Time at different Buffer Size

Buffer Size	Absolute error	Running Time(sec)
500	2.499939213600023e-002	1603.505
600	1.697454938451870e-002	1603.315
700	1.437965286060358e-002	1603.976
800	1.087668066985989e-002	1603.736
900	8.520659939131679e-003	1603.826
1,000	7.163892404911190e-003	1608.603
1,100	4.576511431885187e-003	1606.009
1,200	4.816045369671360e-003	1610.225
1,300	3.979656974547638e-003	1602.694
1,400	3.088250626503199e-003	1603.185
1,500	2.629348819357051e-003	1607.641
1,600	2.450198291522439e-003	1602.914
1,700	2.051291947776107e-003	1603.996
1,800	1.866623460766134e-003	1606.459
1,900	1.499587474588055e-003	1629.983
2,000	1.333507266908428e-003	1637.915

Figure 9: Buffer size vs accuracy and running time

to check the change in accuracy rate. Table 4 shows the result and in Figure 9 the solid line denotes the error rate while dotted line the running time. It can know from the figure 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 data size, our algorithm is generalized to kernel density estimation, the error rate drop 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 stream are drawn from the same distribution, while stream 4 has been sorted, 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 10 and 11 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, while 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 has been obtained.

5. CONCLUSIONS AND FUTURE WORK

Density estimation over data stream has become more and more interesting in database community, while classical kernel method cannot handle data stream effectively. In this paper, we bring forward a new concept of *M-Kernel* and propose a novel density estimation algorithm based on it. The former guarantees that the latter only needs a fixed-size memory, regardless the amount of data in the stream. Analysis and experiments prove that our algorithm is capable to process data stream and build up its density function at the speed data arrival. 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.

In the future, we will integrate this algorithm to more data mining applications to testify its efficiency. 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.

6. ADDITIONAL AUTHORS

Additional authors: Weining Qian (Dept. of Computer Science, Fudan University, email: `wnqian@fudan.edu.cn`) .

7. 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 *Proceedings of International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 579–522, 2000.

[3] Y. Cheng. Mean shift, mide seeking, and clustering. *IEEE Transactions 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 *Proceedings of International Conference on Knowledge Discovery and Data Mining (KDD)*, 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 *Proceedings of Symposium on Foundations of Computer Science (FOCS)*, pages 359–366, 2000.

[9] D. Gunopulos, G. Kollios, V. J. Tsotras, and C. Domeniconi. Approximating multi-dimensional aggregate range queries over real attributes. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 463–474, 2000.

[10] G. Hulten, L. Spencer, and P. Domingos. Mining time-changing data streams. In *Proceedings of International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 97–106, 2001.

[11] G. Kollios, D. Gunopoulos, N. Koudas, and S. Berchtold. An efficient approximation scheme for data mining tasks. In *Proceedings of International Conference on Data Engineering (ICDE)*, pages 453–462, 2001.

[12] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Approximate medians and other quantiles in one pass with limited memory. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 426–435, 1998.

[13] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Random sampling techniques for space efficient online computation of order statistics of large datasets. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 251–262, 1999.

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

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

[16] T. Zhang, R. Ramakrishnan, and M. Livny. Fast density estimation using cf-kernel for very large databases. In *Proceedings of International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 312–316, 1999.

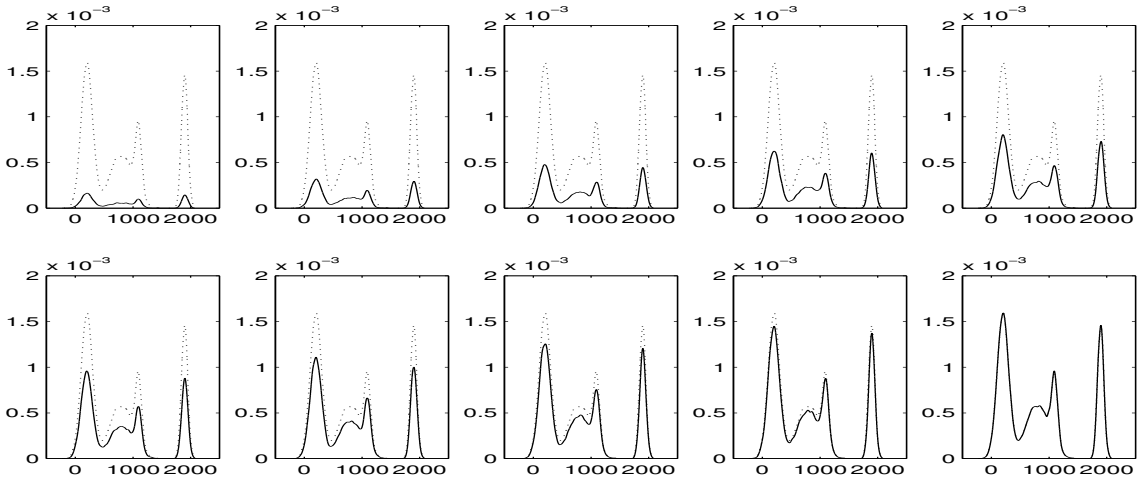

Figure 10: The medial outputs of data stream 2

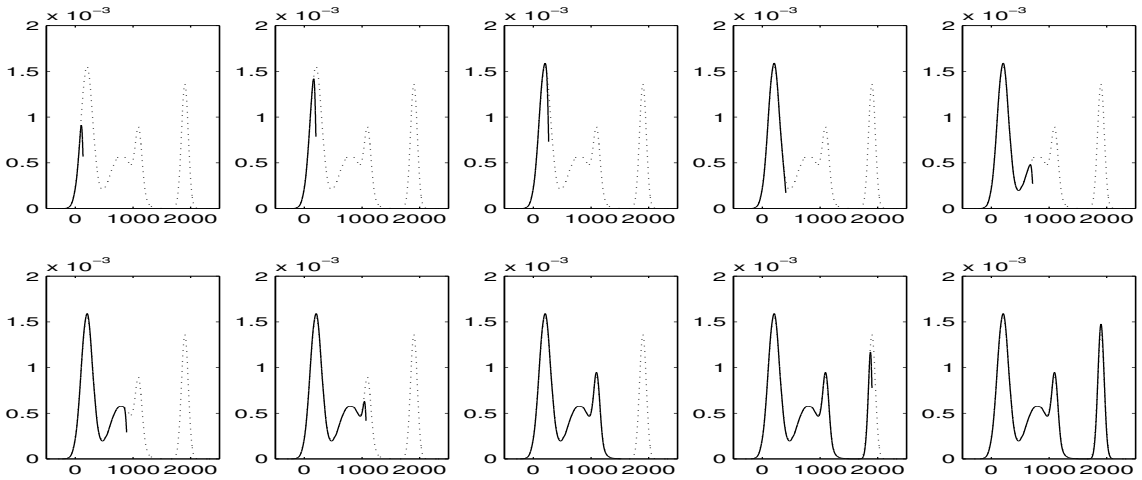

Figure 11: The medial outputs of data stream 4