Real-time Outlier Detection Over Streaming Data

Conference Paper · July 2019

DOI: 10.1109/SmartWorld-UIC-ATC-SCALCOM-IOP-SCI.2019.00063

Real-time Outlier Detection over Streaming Data

1st Kangqing Yu
School of Computer Science
Carleton University
Ottawa, Canada
kangqingyu@cmail.carleton.ca

2nd Wei Shi
School of Information Technology
Carleton University
Ottawa, Canada
wei.shi@carleton.ca

3rd Nicola Santoro
School of Computer Science
Carleton University
Ottawa, Canada
santoro@scs.carleton.ca

4th Xiangyu Ma
School of Information Technology
Carleton University
Ottawa, Canada
johnnyma@cmail.carleton.ca

***Abstract*—Designing outlier detection algorithms over streaming data involves several issues such as concept drift, temporal context, transience, uncertainty, etc. Moreover, to produce results in real-time with limited memory resources, the processing of such data must occur in an online fashion. Therefore, real time detection of outliers on streaming data faces more challenges than performing the same task on batches of data. Several methods have been proposed to detect outliers over streaming data, among which a sliding window technique is frequently used. In this technique, only a chunk of data is kept in memory at each point in time and used to build predictive models. The size of the data in memory simultaneously is referred to as the size of a sliding window. The correctness of the outlier detection results depends largely on the choice of window size. Other similar techniques exist but most of them fail to address the properties of streaming data, and thus produce results exhibiting poor accuracy.**

In this paper, we present an online outlier detection algorithm, over streaming data, that addresses the aforementioned challenges. The proposed algorithm adopts the sliding window technique, however efficiently mines in memory a statistical summary of previous observed data, which contributes to the prediction of incoming data. It further addresses the concept drift problem that exists in streaming data. We evaluated the accuracy of our algorithm on both synthetic and real-world datasets. Results show that the proposed method detects outliers over streaming data with higher accuracy than SOD_GPU algorithm proposed in [9], even when concept drifts occur. The algorithm does not require a secondary memory for processing and is further accelerated using CUDA GPU.

***Index Terms*—outlier detections, streaming data, parallel processing, sliding-window, CUDA**

I. Introduction

A. Background

An *outlier* in a dataset is a data point that is considerably different from the rest of the data as if it is generated by a different mechanism [9]. The minority groups form by outliers in a dataset create patterns that can be recognized from their distributions in the dataset. Applications of outlier detections occur in numerous fields, including fraud detection, network intrusion detection, environment monitoring, etc.

Streaming data is a continuous, unbounded sequence of data records accompanied and ordered by implicit or explicit timestamps [17]. Therefore, when a data stream is *transient*, namely the data points are only available partially at any given point in time, it is impossible to achieve random access on the entire dataset, which is commonly required on outlier detection on static data. Moreover, comparing to static data, streaming data carries *uncertainty* and *concept drift*. *Uncertainty* means that data points are vulnerable to noises and thus unreliable [21]. *Concept drift* means that the distribution of data points is not fixed, and it may change over time [10]. Apart from these considerations, when working on applications that process streaming data, the temporal context and its impact on the results should be considered accordingly. In addition, since the processing on streaming data is online, when designing solutions, computational and memory resource limitations must be taken into consideration. These make data mining over streaming data a challenging task.

B. Contributions

In this paper, we propose a new solution that is based on a non-parametric algorithm *SOD_GPU* (Stream Outlier Detector-GPU) presented by Xia et al. [9]. *SOD_GPU* is based on a density estimation over *non-overlapping window* and a *statistical binned summary* to detect outliers in streaming data. Our proposed algorithm *Cumulative Kernel Density Estimator with Retrospect*, or *C_KDE_WR* in short, further extends *SOD_GPU* algorithm by introducing a novel retrospective step, which takes retroactively into consideration the historical outliers detected, when detecting outliers in the current window. We also introduce a weight assigning scheme to further evaluate data points based on its temporal context. The proposed algorithm is implemented and accelerated using a parallel GPU programming framework, *NVIDIA CUDA* [1]. We compare and report the accuracy of our method with *SOD_GPU* [9] based on different criteria using both synthetic and real datasets. A t-test result further confirms that the accuracy of our method is significantly improved due to the novelties introduced into the outlier detection process.

II. Related Works

A. Distance-based Model

The *distance-based* model introduced by Knorr and Ng [11] was among the very first outlier detection methods that detect outliers on static data. It calculates the pair-wise Euclidian distance between all data and, if one data point has less than k neighbours within distance R, it is considered an outlier. There are variants of this static distance-based approach. For instance, Ramaswamy et al. [14] purposed a method where an outlier is defined by considering the total number of objects whose distance to its k^{th} nearest neighbour is smaller than itself. Angiulli and Pizzuti [1] introduced a method where an outlier is defined by taking into account the sum of the distances from 1^{st} up to the k^{th} nearest neighbours. Later on, several methods have been proposed to extend outlier detection onto streaming data [3], [8], [12], [17]. One of the most popular methods uses a *sliding-window* to help with detecting outliers. Based on the benchmark among all DODDS algorithms given by Luan Tran et al. [22], the MCOD algorithm introduced by M.Kontaki et al. [12] appear to have the best performance. In [12], the solution uses a *event-based framework* to avoid unnecessary computations. In addition, to minimize the cost of range query due to the arrival of new object, it employs evolving micro-clusters to minimize the complexity. The time complexity of this algorithm is guaranteed to be $O(n \log k)$ while maintaining the space complexity to be $O(nk)$, where n is the number of data points and k refers to the parameter of KNN.

B. Density-based Model

The *density-based* model is another way to detect outlier on static data. The idea is to assign a degree of being outlier (a score) based on the density of local neighbourhood, given some predefined restrictions. One of the popular density-based methods is LOCI (Local Correlation Integral). In [13], D. Pokrajac et al. presented an incremental version of LOCI over streaming data. The authors gave theoretical evidence to show that the insertion of new data points as well as deletion of an old data point affects only a limited number of neighbours. The outlier detection does not depend on the total number of records in current dataset and is bounded by $O(n \cdot logn)$ time complexity after insertion of n data points.

[1]Compute Unified Device Architecture: https://developer.nvidia.com/cuda-zone

C. Statistical-based (Parametric) Model

The *statistical-based* model is also known as *parametric* model. The detection model is formulated based on the distribution of data [20] (e.g. Gaussian distribution). One of the most popular distribution used is *Gaussian mixture*, where each data point is given a formulated score. A data point that has a higher score than a given threshold is declared as an outlier. Another popular model is *auto regression*. It works by building a predictive model and defining a cutoff limit. An outlier is detected if it is beyond the cutoff limit by comparing the metrics against the predictive model. Statistical-based models are usually computational inexpensive but assume a fixed distribution in a static dataset. An autoregressive or AR model, also known as an infinite impulse response filter or all-pole model, describes the evolution of a variable measured over the same sample period as a linear function of only its past evolution [4]. It is popular for time series outlier detection and its definition is as follows:

$$x(t) = a_1(t) \times x(t-1) + ... + a_n(t) \times x(t-n) + \xi(t)$$

where $x(t)$ is the series under investigation, a_i $(0 < i \leq n)$ are the autoregression coefficients, n is the order of the autoregression and $\xi(t)$ is usually assumed to be a Gaussian white noise. The coefficient parameters $a_i(t)$ are estimated based on the given time series $x(t), ...x(t-n)$. The model can then be used to predict future time series by defining a threshold.

D. Sliding-window-based Model

One of the most popular outlier detection technique that considers the temporal notation of streaming data uses a *sliding-window* [3], [8], [12]. In each sliding-window, a portion of the data that shares the same temporal context is kept in memory. Outliers are decided solely based on those data in memory. Data is deleted from memory when it expires.

Window size W decides when a data point expires and the slide size S defines the frequency of an algorithm execution involving this data point. Base on W and S, a sliding window can be categorized as *time-based* or *count-based*. In a time-based window, W and S are both defined as time intervals [12]. Each time-based window of size W has a starting time T_{start}, an ending time $T_{end} = T_{start} + W$. The number of data points in each window may vary. S is a time interval between the starting time of any two contiguous windows. In case of count-based window, W is measured as a fixed amount of data points, and S is a predefined number of data points after which the count for a new window starts.

For both time-based window and count-based window, depending on the relationship between window size W and slide size S, a sliding-window can be further categorized into *non-overlapping window* and *overlapping window*. Base on the definition of W and S, each point in a sliding window will experience $x = |\frac{W}{Slide}|$ slides before it expires [12]. If $x = 1$, which means window size W is equal to slide size S, a window is referred to as non-overlapping and each data point in it will be used for calculation only once. Otherwise, a window is called an overlapping window and each data point will be calculated multiple times.

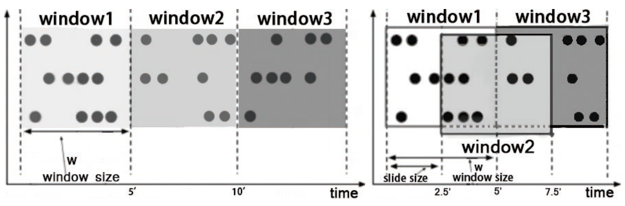


Fig. 1: Non-overlapping (left) vs Overlapping time-based window (right)

E. Kernel Density (Non-parametric) Model

The *Kernel Density Estimator* (KDE) is a non-parametric method to estimate probability density function of random variables [19]. It has become increasing popular as an efficient way to detect outliers over streaming data. The probability density function $f(x)$ is defined as:

$$f(x) = \frac{1}{n}\sum_{n=1}^{n} k_{h_i}(x_i - x)$$

where $k_{h_i}(x)$ is the kernel functions with bandwidth h_i. The bandwidth can be calculated online using Scott's rule [19]. In [9], Xia *et al.* use GPU to accelerate kernel density estimator. The proposed solution uses *non-overlapping sliding window* and a *statistical binned summary* to detect outliers in high volume and high dimensional streaming data. In this method historical data are mined efficiently into bins.

F. Clustering-based Model

K-Mean clustering is used in outlier detection in solution proposed in [6]. Data in each sliding window is clustered. But unlike the distance based approach, the detected outliers are not reported immediately but rather considered as *candidate outliers*. A metric which measure the mean value of each cluster is maintained and carried over to the next window in the stream in order to be further compared with data in future windows. On the other hand, *K-Median clustering* algorithm presented in [5] clusters each chunk of data into k to $k\log(n)$ clusters. The weighted medians found in current window is passed to the next one in order to detect outliers. Both approaches require user input on value k a priori.

III. Algorithm *C_KDE_WR*

A. SOD_GPU and C_KDE_WR General Description

SOD_GPU is based on a cumulative approximation of the probability density function $f(x)$ on current data points contained in a non-overlapping sliding window as well as statistical binned summary that is mined from historical data. In SOD_GPU, Gaussian kernel density estimator is used to approximate the density function as it gives smooth estimation over the entire dataset [9]. A popular technique presented in [7] is used to turn all historical data into a binned summary. When the density on $f(x)$ is larger than a threshold θ, a data point is considered as an outlier. θ is estimated dynamically based on the average of all data points in the current window. We adopt the basic SOD_GPU framework and propose C_KDE_WR to further address concept drifting problem on streaming data in order to reduce false positive on the results. A complete list of steps of C_KDE_WR is described in Algorithm 1. We describe a detailed new KDE calculation on a sliding window in subsections III-B and III-C. We then present how to calculate a bin index, how to maintain bin statistics as well as how to perform a density estimation on a binned summary in subsection III-D. We further present how to choose a density estimation threshold and set up an outlier factor in subsection III-E. Finally, we introduce how to detect and address concept drift in subsection III-F.

Algorithm 1 C_KDE_WR

```
Input: bins, candidate_outliers, window_data
Output: inliers, outliers
 1: bandwidths = calculate_bandwidth (window_data);
    // using E.q 3
 2: query = window_data + candidate_outliers;
 3: window_density = calculate_window_density (query,
    window_data, bandwidths); // using E.q 4
 4: bin_density = calculate_bin_density (query,
    bins.values(), bandwidths, bins.weights()); // using E.q 8
 5: threshold = 1 / AVG (window_density + bin_density) * $\xi$;
    // using E.q 9
 6: next_candidates_outliers = {};
 7: for $p$ in query do
 8:     if (p.density $<$ threshold) then
 9:         p.rank += 1;
10:     else
11:         p.rank -= 1;
12:     end if
13:     if (p.rank == RANK) then
14:         outliers $\cup$ p;
15:     else if (p.rank $\leq$ 0) then
16:         inliers $\cup$ p;
17:     else
18:         next_candidates_outliers $\cup$ p;
19:     end if
20: end for
21: candidate_outliers = next_candidates_outliers;
22: points_with_indices=calculate_bin_index(window_data);
    // using E.q 5
23: bins=maintain_bin_statistics (bins, points_with_indices);
    // using E.q 6, 7
24: return bins, inliers, outliers
```

B. Kernel Density Estimator

In order to address the concept drift problem associated with streaming data, we estimate it dynamically as new data points arrive, so that the probability density function $f(x)$ always reflects the most recent data distribution. To address the uncertain property of streaming data, we use *kernel density* *estimator* to estimate the probability density function. Different from the histogram-based estimation, where the occurrence of each data point x_i increases by 1 the density of $f(x)$ at its corresponding bin, the kernel estimator increase the density at x_i only by a probability of $p(x_i)$, then distribute the rest $1 - p(x_i)$ to x_i's neighbours. The closer the data values are to x_i, the higher their probabilities will be distributed. Such probability estimation models the uncertainty carried from streaming data points.

If $(x_1, x_2, ..., x_n)$ are n data points that have been observed so far, the probability density function $f(x)$ is defined by Eq. 1, where $k(x)$ is called the *kernel function*.

$$f(x) = \frac{1}{n}\sum_{i=1}^{n} k(x_i - x) \tag{1}$$

The kernel function is responsible for distributing the probability of occurrence around data point x_i. In this application, we choose Gaussian kernel as it gives a smooth estimation. The Gaussian kernel is given by:

$$k(x, x') = \frac{1}{(2\pi)^{D/2}H} exp\Big\{ - \frac{1}{2}\Big(\frac{x - x'}{H}\Big)^2 \Big\} \tag{2}$$

where D is the dimension of data points and $H = (h_1, h_2, ...h_n)$ is the bandwidth of the kernel function. The bandwidth is used to control the extend to which, the rest of data value, other than x' should be distributed. The larger the bandwidth H, the more it will be distributed to data points other than x'. As we use Gaussian kernel, probability of occurrence will be distributed to all data points from $-\infty$ to $+\infty$ [19]. However, the majority of density will still be distributed to the neighbourhood of x' only. We use Scott's rule [19] to calculate the bandwidth at each dimension based on the following formula:

$$h_i = \sigma_i n^{1/D+4} \tag{3}$$

where σ_i is the standard deviation of data points at dimension i. In our application, to estimate the overall distribution of the probability density function $f(x)$, we defined the *cumulative kernel density estimator* function $f_{cumulative}(x)$ by adding the kernel estimator in the sliding window $f_{window}(x)$ and the kernel estimator in a binned summary $f_{bin}(x)$ accordingly. Therefore,

$$f_{cumulative}(x) = f_{window}(x) + f_{bin}(x)$$

In particular, for a binned summary, we modified the kernel estimators slightly using the bin implementation so that it would not require to store the entire historical data points, and the number of evaluations is therefore reduced. To address the temporal notation of streaming data, we also introduce a factor on kernel estimator function of binned summary to weight each bin accordingly. The details are presented in the following subsections.

C. *Sliding Window*

Due to the unbounded nature of streaming data, it is unpractical to store all observed data in a limited amount of memory in order to approximate the kernel estimators. In C_KDE_WR, we only store the most recent data points in memory at each time in the *sliding-window* as introduced in Section II-D. We divide streaming data into chunks of windows at a regular time interval. If we let W denote the window size and T_0 denote the starting time, the window boundaries will therefore be $T_0 + W$, $T_0 + 2W$, ..., $T_0 + jW$ $(j > 0)$.

For the data points in a sliding window, they contribute directly to the density function $f(x)$ as defined in Eq. 1. Outliers are found over these windowed data. Therefore, each data point x that is in the current sliding window, its density over the sliding window is defined by Eq. 4 after applying Gaussian kernel, where n is the number of data points in a sliding window.

$$f_{window}(x) = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{(2\pi)^{D/2}H} exp\Big\{ - \frac{1}{2}\Big(\frac{x - x_i}{H}\Big)^2 \Big\} \tag{4}$$

However, as decisions are only made based on the current window data and no historical data are considered, the result might not be accurate. In addition, the effectiveness of this naive windowed approach will highly depend on the hard-to-define window size W. Therefore, only keeping the sliding window is not enough both in terms of efficiency and accuracy. We also need to maintain some synopsis of historical data.

D. *Binned Summary*

When data points expire from the current window, they are not discarded. Rather, they are mined into something called *binned summary* that has been calculated statistically. As the kernel density estimator requires large amount of computations and it is unpractical to store all expired data points in memory, the binned summary is a popular and efficient implementation for kernel estimator that would not require storing the entire history of observed data and it can hugely reduce the number of evaluations [7]. There are many implementations of binned summary in literature and we apply the one introduced in [7]. In our proposed C_KDE_WR algorithm, the binned summary needs to perform these computational steps, explained in the following: These steps are: 1) calculate bin index; 2) maintain bin statistics; and 3) density estimation over bins.

1) Calculate Bin Index: The idea in this binned implementation approach is to divide the entire range of data points into some equally spaced intervals and bin each expired point into these intervals accordingly. To find the bin index, assume there are N data points and each consists of D dimensions. For each dimension j, we find the upper bound $max(x_j)$ and lower bound $min(x_j)$ in order to derive the length of that dimension, and then divide it by a pre-defined value k to get its width, Δ. Therefore:

$$\Delta = [max(x_j) - min(x_j)]/k$$

To find the corresponding bin index (where this data point belongs to) for each data point x_i, firstly, we map the input values in each dimension of x_{ij} into interval [0, 1] using the following function:

$$x_{ij} = \frac{x_{ij} - min(x_j)}{max(x_j) - min(x_j)}$$

Then, we encode the data point x_i as:

$$< I_{i1}, I_{i2}, I_{i3}, ..., I_{iD} >$$

where $I_{ij} = x_{ij}/\Delta$. After that, we use the following formula to find the corresponding bin index for data point x_i:

$$B_{x_i} = (I_{iD}-1)k^{D-1}+(I_{i(D-1)}-1)k^{D-2}+...+(I_{i2}-1)k+I_{i1} \quad (5)$$

The result of these operations is to assign each expired data point into its corresponding bin B_i, where $0 \leq i \leq k^D$. It is worth noticing that the number of total possible bins, which is k^D, grows exponentially with the number of dimension; however we are only interested in the non-empty bins. As data in the real-world turns to be clustered, the number of actual non-empty bins m turns out to be much smaller than the total number of possible bins $m << k^D$ [9]. Therefore, we are not worried about the curse of dimension problem here.

2) Maintain Bin Statistics: For each bin, we maintain the number of data points (noted as bin count C_i) and their aggregate mean value vector (noted as $M_i =< \mu_{i1}, \mu_{i2}, ..., \mu_{iD} >$) for the points that fall into this bin. The μ_{ij} here is the average mean value of all data points in B_i at dimension j. Additionally, we also maintain the mean value vector μ and the standard deviation Σ over the entire dataset. These bin statistics are maintained and updated at the end of each batch when current window expires as follows.

Assume we are currently processing the n^{th} window and for each non-empty bin B_i, we have processed and aggregated $n-1^{th}$ windows of expired data. C_i^{n-1} denotes the total number of data points that fall into bin B_i up to window $n-1^{th}$; M_i^{n-1} denotes the mean value vector of data points in bin B_i up to window $n-1$; c_i^n denotes the number of data points at bin B_i in current n^{th} window; And μ_i^n is the mean value vector of data points at bin B_i in current n^{th} window. To update the mean value vector (M_i^n) and the bin count (C_i^n) at bin index i before expiring n^{th} window, we apply the following formulas respectively:

$$M_i^n = \frac{c_i^n * \mu_i^n + C_i^{n-1} * M_i^{n-1}}{c_i^n + C_i^{n-1}} \quad (6)$$

$$C_i^n = c_i^n + C_i^{n-1} \quad (7)$$

3) Binned Summary Density Estimation: For a binned summary, its density function is calculated slightly differently than those in sliding widow defined by Eq. 4. The bin B_i contributes to the density function $f(x)$ by taking into considerations both its mean value vector M_i and the number of data points C_i in B_i bin. We modify the kernel estimators slightly as the one defined in Eq. 1 for a binned summary. Therefore, for a data point x in the current window, its density over a binned summary is defined by Eq. 8 after applying Gaussian kernel, where m is the number of bins in the binned summary:

$$f_{bin}(x) = \frac{1}{C}\sum_{i=1}^{m} \frac{C_i}{(2\pi)^{D/2}H} exp\Big\{-\frac{1}{2}\Big(\frac{x - M_i}{H}\Big)^2\Big\} \quad (8)$$

The density of a data point is calculated cumulatively by applying Eq. 4 and Eq. 8 together, in order to define the outlier factor of data points. However, density of a bin B_i is purely dependent on the number of data points C_i in that bin. Past bins that have not been updated for a long period that may not match the current trend of data distribution could still have an equal impact as those recent bins. This will not help in addressing the temporal property of streaming data as we want the model to always be consistent with the most up-to-date trend of data distribution.

E. Threshold and Outlier Factor

To decide the outlierness of a data point x, we define its *outlier factor* by calculating the inverse of the density of the point x on the cumulative kernel density function $f(x)$. Thus, the outlier factor f_o is defined by Eq. 9

$$f_o = \frac{1}{f(x)} \quad (9)$$

We also define threshold $\theta_{threshold}$ on outlier factor f_o to cut-off the limit on the precise definition of outlier. The threshold $\theta_{threshold}$ is defined by the average density of all points in $f_{cumulative}(x)$, noted as p_{avg} and the parameter ξ, given by:

$$\theta_{threshold} = \frac{1}{p_{avg} * \xi}$$

where $0 < \xi < 1$. Notice that the threshold θ will be adjusted and re-calculated as data points are continuously being observed since the p_{avg} is updated dynamically.

F. Concept Drift Detection

1) Candidate Outliers Retrospection: In order to address the concept drift problem on streaming data, we use the following strategy: when an outlier is detected in the current window, rather than reporting and confirming the decision immediately, it will be treated as *candidate outlier* and re-evaluated again in the future windows. This is because when a concept drift occurs, the underlying kernel estimator model needs time to self-adjust to reflect the latest change of data distribution. Reporting outliers immediately may generate a large number of false positive when concept drift starts to emerge. Therefore, we introduce a *retrospect step* to re-consider the decision made on the candidate outliers in the current window. Such re-evaluate occurs in a number of future windows. More specifically, we assign a rank r to track the number of times a candidate outlier is has been continuously considered as a candidate outlier. Each r is incremented by 1 or -1 accordingly depending on whether it is consider a candidate outliers or not in this window. When a r reaches a pre-defined value R, it is considered as a true outlier and reported consequently. In the case that r count reaches zero, it is removed from the candidate outlier list.

2) Forgetting Factor: When calculating the density of a data point over a binned summary, the freshness of the summary must be considered. Namely, bins that are older should gradually fade away, consequently have less impact on deciding the densities of current data than those recent ones. Therefore, we introduce a *forgetting factor* over binned summary when calculating the kernel estimator. The introduction of the forgetting factor will help us address the temporal property of streaming data. Recent bins are clearly more interesting to us than the old bins; therefore, recent bins should receive more weights than the less recent ones.

Exponential forgetting is a weight assigning scheme which gives more weight to the recent data points and less weight to the older data points when weight decreases exponentially from present to past [2]. We apply exponential forgetting in a binned summary by storing the timestamp for each bin when it was last updated, then sort the bins accordingly. The relative weight between two consecutive bins is a constant called forgetting factor λ, where $0 < \lambda < 1$. The most recent bin receives weight 1 and the older ones will receive a weight λ on top of the weight from its previous bin. Therefore each bin receives a weight according to it relative position to the most recent bin in this sorted list. The forgetting factor λ is selected using the bootstrapping method presented in [2].

Let $(B_1, B_2, B_3, ..., B_m)$ be bins that are sorted according to their last updated timestamps, $(M_1, M_2, M_3, ..., M_m)$ be their corresponding mean value vectors, and $(C_1, C_2, C_3, ..., C_m)$ be their bin counts. The corresponding bin weights are denoted as $(\lambda^{n-1}, \lambda^{n-2}, \lambda^{n-3}, ..., 1)$. If we apply these weights to the KDE function defined over a binned summary in Eq. 8, the probability density function becomes Eq. 10 after applying the exponential forgetting factor.

$$f_{bin}(x) = \frac{1}{\sum_{i=1}^{m} \lambda^{m-i} C_i} \sum_{i=1}^{m} \frac{\lambda^{m-i} C_i}{(2\pi)^{D/2} H} exp\left\{ -\frac{1}{2}\left(\frac{x - M_i}{H}\right)^2 \right\} \tag{10}$$

IV. Implementation

A. Density Estimation on GPU

The density estimation is performed on GPU in the same fashion as SOD_GPU method [9], taking advantages of CUDA streams and shared memory technique introduced in CUDA platform. The only difference in here is that in *Kernel 2* and *Kernel 3* functions, where we not only need to transfer data in the current window, but also those data tuple that have been detected as candidate outliers in previous windows to calculate their outlier factors. Other implementation details remain unchanged.

For further details of GPU implementation, please refer to the original paper of the SOD_GPU method in [9].

B. Binned Summary Maintenance on CPU

Unlike GPU, programming on a multi-core CPU follows MIMD (Multiple Instructions Multiple Data) architecture. For updating on a binned summary, most of the computations are statistically based, which can simply be implemented by using a *reduce* operation. The reduce operation can easily be parallelized on a multi-core system as we have seen in many frameworks. In our case, each time we receive a new sliding window full of data points, we need to mine their statistics and update these information into our existing binned summary. Hence, we design the maintenance step on a binned summary in following two phases:

1) In-window Mining: During a *In-window mining*, we calculate statistics on all data points within the current window. As mentioned in Section III-D2, we calculate the mean value vector μ_i^n and bin count c_i^n for each of its corresponding bin B_i in order to calculate its cumulative mean value vector M_i^n and bin count C_i^n at bin index i in this window. Before calculating μ_i^n and c_i^n, we first need to determine the bin index i for all data points in the current window using Eq. 5 and group them by using the *keyedBy* operation. We then apply the *reducers* for each of these indexed groups to derive the μ_i^n and c_i^n for their corresponding bins appeared in this window. In addition, we also need to find the most recent data point (point with the largest timestamp) for each bin B_i in this window and mark it as the last updated timestamp for this bin.

2) Out-of-window Mining: Once we get the μ_i^n, c_i^n from the current window, we can easily derive the next bin statistic for B_{i+1}, including mean value vector M_i^n, bin count C_i^n. Once we obtained the bin statistic for each updated bin B_i, we also need to update its *last-updated timestamp* by setting it to the timestamp of the most recent data point in that bin from the last window.

V. Experiments

A. Datasets

1) Synthetic Datasets: We measured the accuracy of C_KDE_WR on synthetic datasets generated from Gaussian mixture distribution with outlier points generated from uniform distribution within a given range. We chose the Gaussian mixture model because its distribution can change over time. A change of distribution simulates the concept drift of streaming data. In order to simulate this change, we generate $10,000$ data samples from eight multi-dimensional data using Gaussian distributions with different means but same variances. These points were considered as inliers and are ordered by the distributions that they belong to. 100 outlier points were generated uniformly and inserted into inlier points in a random order.

2) Real-world Datasets: We also measured the accuracy of C_KDE_WR on two real-world datasets obtained from UCI machine learning library ([2]): KDDCup99 network dataset for the intrusion detector learning task, and Covertype forest cover dataset for cover type prediction task in forest. Both of these

[2]http://archive.ics.uci.edu/ml/datasets.html

datasets are originally designed for classification tasks. In our case, we chose classes with minority instances as outlier points (i.e. less than 10% occurrence). For KDDCup99 dataset, we chose points belong to *normal*, *smurf* and *nepturn* classes as inliers. Points belongs to other classes were considered as outliers. For Covertype dataset, points belong to class *Spruce-Fir* and *Lodgepole Pine* were chosen as inliers. Other points were considered to be outliers. We did feature selections on those datasets and take log on the selected attributes whose values deviate largely from the rest. For each of these two datasets, we randomly chose $10,000$ samples based on the proportion of each class, in which outlier points are uniformly distributed. Our proposed C_KDE_WR work with both time-based sliding window and count-based sliding window. The performance evaluation reported in this paper is based on count-based implementation. Therefore, window size is chosen as containing $1,000$ data points so there would be 10 windows in each experiment.

B. Test Environment

We used NVIDIA CUDA framework to accelerate the algorithm in order to achieve low latency for real-time computation. Furthermore, we leverage *shared memory* in thread block to reduce the bandwidths of memory transfer between host and device, and used *CUDA streams* to overlap memory transfer between different kernels invocations. All experiments were performed on a server with Ubuntu 16.04 operating system, equipped with an Intel 3.3GHz quad-core CPU and 64GB host memory, along with an NVIDIA GTX 1080 Ti GPU (6.1 compute capability) with 11GB device memory and 3584 CUDA cores. The CUDA runtime version used is 9.2, which was the latest one at writing time. Implementation is written in Python running with Numba compiler [3], which is a just-in-time compiler for CUDA.

C. Evaluation Criteria

Outlier detection can be thought of as a special type of binary classification task since each data point needs to be classified as either inlier or outlier. The only difference is that the dataset used for outlier detection is hugely unbalanced. In order to measure the model accurately, we use *Precision*, *Recall* and *F-Score* metrics, which are widely used for accuracy evaluation in machine learning. Furthermore, we performed t-test between our C_KDE_WR algorithm and the SOD_GPU algorithm proposed in [9]. We recorded the p-value, confidence interval and the variance on different datasets to further demonstrate that our proposed novelties have improved over its counterpart in terms of accuracy.

Precision is defined as the number of correctly detected outliers (true positives) divided by the total number of detected outliers (true positives + false positives). *Recall* is defined as the number of correctly detected outliers divided by the total number of outliers in the dataset (true positives + false negatives), and *F-Score* is defined as:

$$F_{score} = \frac{2 \times precision \times recall}{precision + recall}$$

D. Accuracy Evaluation

We set ξ to 0.1 and k to 100 in C_KDE_WR. Three retrospects are required to finalizing a true outlier detection. 0.5 is selected as forgetting factor λ and the window size is set at 1000 for all synthetic and real-world datasets. We performed the experiments 30 times in each case by shuffling the outlier points uniformly within the inlier points. Fig. 2 illustrates the comparison of results between the two algorithms. More specifically, Fig. 2 (a) shows the average accuracies of C_KDE_WR and SOD_GPU, in terms of Precision, Recall and F-Score, after 30 independent runs on KddCup99 dataset. Our proposed C_KDE_WR algorithm performs better in terms of Precision but slightly lower than SOD_GPU in terms of Recall score. The results on CoverType and those synthetic datasets are very similar as we can see from Fig. 2 (b) (c) and (d).

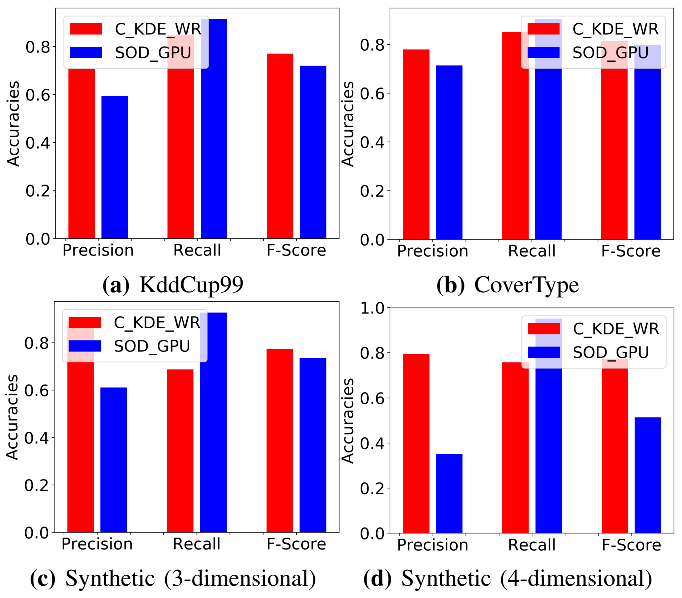


Fig. 2: Average accuracy comparisons on KddCup99 dataset (a), CoverType dataset (b), and Synthetic datasets (c) (d)

Furthermore, our results show that C_KDE_WR improves over SOD_GPU in the overall F-Score on all datasets. This claim is further supported by Fig 3 and the t-test results we have obtained in Table I. In Fig 3, the F-Scores of our C_KDE_WR algorithm after a number of repeated experiments are significantly higher than those of the SOD_GPU algorithm on both synthetic and real-world datasets as illustrated. The same conclusion can be made on table I that further supports the hypothesis that our C_KDE_WR algorithm have improved over the SOD_GPU algorithm significantly in terms of Precision and F-Score.

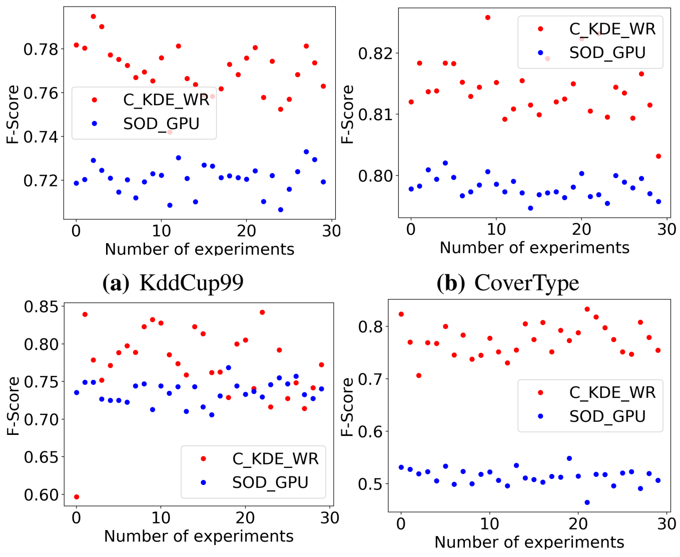
VI. Conclusion and Future Work

In this paper, we present a novel algorithm C_KDE_WR to effectively and accurately detect outliers from streaming

[3]http://numba.pydata.org/numba-doc/latest/index.html

TABLE I: T-test: C_KDE_WR vs SOD_GPU

	KddCup99 Dataset		CoverType Dataset		Synthetic Dataset	
	Precision	F-Score	Precision	F-Score	Precision	F-Score
p-value	<2.2e-16	<2.2e-16	<2.2e-16	<2.2e-16	<2.2e-16	<2.2e-16
confidence interval	(0.106, 0.119)	(0.045, 0.055)	(0.064, 0.680)	(0.014, 0.018)	(0.264, 0.291)	(0.018, 0.056)
variance	9.964e-6	5.669e-5	1.059e-6	2.853e-4	6.982e-5	9.941e-5
coefficient of variation	2.813%	4.792%	1.561%	5.611%	3.015%	26.691%



(a) KddCup99 **(b)** CoverType

(c) Synthetic (3-dimensional) **(d)** Synthetic (4-dimensional)

Fig. 3: F-Score differences on KddCup99 dataset (a), CoverType dataset (b), and Synthetic datasets (c) (d)

data. To achieve real-time results with low latency, we leverage NVIDIA CUDA framework to accelerate the execution of our algorithm on GPU. As supported by the experiment results performed on both synthetic and real-world datasets, our proposed C_KDE_WR outperforms SOD_GPU in terms of detection accuracy, which is the state of the art at writing time. More specifically, C_KDE_WR achieves a lower number of false positives and thus, increases the precision and F-score metrics. This further proves the effectiveness of our novelties introduced. We conclude that C_KDE_WR algorithm can be applied to applications with continuous streaming data on detecting anomalies in real-time without requiring any prior knowledge and secondary memory.

Although in most cases the false positive count of C_KDE_WR is considerably reduced, comparing to SOD_GPU, we believe that the number in some specific cases can still be further reduced. Furthermore, a good outlier detection result relies on the efficiency of raw dataset pre-processing as well as the selection of features. We hope to further reduce the false negative count when more precise definition of outliers is provided in more specific applications. A more efficient data pre-processing step for datasets in a more general context is yet to be found.

References

[1] F. Angiulli and C. Pizzuti. Outlier mining in large high-dimensional data sets. *IEEE Transactions on Knowledge and Data Engineering*, 17(2):203–215, 2005.

[2] T.J Brailsford, J.HW Penm, and R.D. Terrell. Selecting the forgetting factor in subset autoregressive modelling. *Journal of Time Series Analysis*, 23(6):629–649, 2002.

[3] L. Cao, D. Yang, Q. Wang, Y. Yu, J. Wang, and E.A Rundensteiner. Scalable distance-based outlier detection over high-volume data streams. In *Proceedings in* 30th *IEEE International Conference on Data Engineering (ICDE)*, pages 76–87, March 2014.

[4] D. Curiac, O. Banias, F. Dragan, C. Volosencu, and O. Dranga. Malicious node detection in wireless sensor networks using an autoregression technique. In *Proceedings of the* 3rd *International Conference on Networking and Services (ICNS)*, pages 83–83, 2007.

[5] P. Dhaliwal, MPS Bhatia, and P. Bansal. A cluster-based approach for outlier detection in dynamic data streams (korm: k-median outlier miner). *arXiv preprint arXiv:1002.4003*, 2010.

[6] M. Elahi, K. Li, W. Nisar, C. Lv, and H. Wang. Efficient clustering-based outlier detection algorithm for dynamic data stream. In *Proceedings of* 5th *International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*, volume 5, pages 298–304, 2008.

[7] J. Fan and J.S Marron. Fast implementations of nonparametric curve estimators. *Journal of Computational and Graphical Statistics*, 3(1):35–56, 1994.

[8] D. Georgiadis, M. Kontaki, A. Gounaris, A.N Papadopoulos, K. Tsichlas, and Y. Manolopoulos. Continuous outlier detection in data streams: An extensible framework and state-of-the-art algorithms. In *Proceedings of the* 38th *International Conference on Management of Data (SIGMOD)*, pages 1061–1064, 2013.

[9] C. HewaNadungodage, Y. Xia, and JJ. Lee. Gpu-accelerated outlier detection for continuous data streams. In *Proceedings of the* 30th *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1133–1142, 2016.

[10] N. Jiang and L. Gruenwald. Research issues in data stream association rule mining. *SIGMOD Rec.*, 35(1):14–19, March 2006.

[11] E.M Knox and R.T Ng. Algorithms for mining distance-based outliers in large datasets. In *Proceedings of the* 24th *International Conference on Very Large Databases (VLDB)*, pages 392–403, 1998.

[12] M. Kontaki, A. Gounaris, A.N Papadopoulos, K. Tsichlas, and Y. Manolopoulos. Continuous monitoring of distance-based outliers over data streams. In *Proceedings of the* 27th *IEEE International Conference on Data Engineering (ICDE)*, pages 135–146, 2011.

[13] D. Pokrajac, A. Lazarevic, and L.J. Latecki. Incremental local outlier detection for data streams. In *Proceedings of the* 1st *IEEE Symposium on Computational Intelligence and Data Mining (CIDM)*, pages 504–515, 2007.

[14] S. Ramaswamy, R. Rastogi, and K. Shim. Efficient algorithms for mining outliers from large data sets. In *Proceedings of the* 19th *International Conference on Management of Data (SIGMOD)*, pages 427–438, 2000.

[15] S. Sadik and L. Gruenwald. Dbod-ds: Distance based outlier detection for data streams. In *Proceedings of the* 21st *International Conference on Database and Expert Systems Applications (DEXA)*, pages 122–136, 2010.

[16] S. Sadik and L. Gruenwald. Online outlier detection for data streams. In *Proceedings of the* 15th *Symposium on International Database Engineering & (IDEAS)*, pages 88–96, 2011.

[17] S. Sadik and L. Gruenwald. Research issues in outlier detection for data streams. *SIGKDD Explor. Newsl.*, 15(1):33–40, 2014.

[18] S. Sadik, L. Gruenwald, and E. Leal. In pursuit of outliers in multi-dimensional data streams. In *Proceedings of the* 4th *IEEE International Conference on Big Data (Big Data)*, pages 512–521, 2016.

[19] D.W Scott. *Multivariate Density Estimation: Theory, Practice, and Visualization*. John Wiley & Sons, 2015.

[20] S. Sreevidya et al. A survey on outlier detection methods. *International Journal of Computer Science and Information Technologies (IJCSIT)*, 5(6), 2014.

[21] N. Tatbul. Streaming data integration: Challenges and opportunities. In *Proceedings of the* 26th *IEEE International Conference on Data Engineering Workshops (ICDEW)*, pages 155–158, 2010.

[22] L. Tran, L. Fan, and C.s Shahabi. Distance-based outlier detection in data streams. *Proc. VLDB Endow.*, 9(12):1089–1100, 2016.