Real-time Outlier (anomaly) Detection over Data Streams

by

Kangqing YU

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
In partial fulfillment of the requirements
For the Master degree in
Computer Science

School of Computer Science, Faculty of Science
Carleton University

© Kangqing YU, Ottawa, Canada, 2018
Abstract

Outlier detection has become an increasing challenging data-mining task in modern applications due to the fact that the data may come in the form of streams rather than batches. When designing an algorithm in a streaming context, several factors need to be taken into considerations, including concept drift, temporal context, transiency, uncertainty, etc [33]. In addition, the processing of data needs to be online in order to provide real-time results with limited memory resources. Several methods have been proposed in literatures to detect outliers over data streams. Among all of them, a very popular technique is called sliding window, which only keeps a chunk of data in memory and build predictive models based on those data. However, there are several problems with this approach. One of them is that the decision is purely based on model that has been built from the data in current window and the correctness of this approach depends hugely on the choice of window sizes. Other similar techniques exist but most of them fail to address those properties of data streams and therefore rendering poor accuracies.

In this thesis, we will present two online algorithms to detect outliers over data streams, based on the sliding window approach, to address all the aforementioned challenges when processing data over streams. Specifically, one is based on a classical parametric algorithm called Local Outlier Factor (LOF) and takes advantages of overlapping sliding window to support continuous monitoring of status of data points in window; the other is based on a non-parametric algorithm called kernel density estimation and uses non-overlapping window where detected outliers are treated as a candidate outliers for further analysis before coining them as true outliers. Both algorithms keep statistical summaries of previous windowed data in memory that help with the decisions of future data. Proved by different datasets, our proposed methods can detect outliers accurately over data streams, where data distribution may change over time. Both algorithms do not require secondary memory and only reserve limited amount of primary memory to store the windowed data and the statistical summary. Obsolete data will be discarded as soon as they expire from window.
Acknowledgements

First and foremost I would like to thank my MS.c supervisor, Prof Nicolas Santoro, for providing me with the strength and perseverance to complete this thesis and for blessing me with an amazing support, both in my academic and personal life.

Besides, I would like to thank all my examinors of my MS.c program. Your prestigious feedbacks on my thesis are of vital value to the completion of this thesis and my future academic career.

I would also like to take this opportunity to express my gratitude to Prof Frank Dehne for giving me inspiration on this thesis. The idea of this thesis comes directly from his graduate course at Carleton University and this thesis works as an extension over the project I did in his course.

I would like to thank all the course instructors, that I have been involved with during my time in Carleton University and University of Ottawa for giving me the knowledge and background to finish this thesis and MS.c degree. The knowledge I learnt from these courses are the foundations of completion of my MS.c degree.

In the end, I would like to thank my parents for their continuous supports on my graduate study after I resigned my jobs in UK. The supports they gave me are the backbone of completion of my MS.c degree. Thank you mom and dad, I love you!
Table of Contents

List of Tables viii

List of Figures ix

1 Introduction 1
  1.1 Background ...................................................... 1
  1.2 Motivation ........................................................ 2
  1.3 Our Contribution ............................................... 5
  1.4 Thesis Organization ............................................. 6

2 Problem Statements and Related Work 8
  2.1 Overview .......................................................... 8
  2.2 Definitions and Problem Statements ........................... 9
    2.2.1 Outlier Definition ......................................... 9
    2.2.2 Data Streams Definitions and Properties ................ 10
    2.2.3 Challenges of Outlier Detection over Data Streams ........ 11
  2.3 Related Works .................................................. 15
    2.3.1 Outlier detection models with labelled data ............ 15
2.3.2 Outlier detection models without labelled data ........................................ 17
2.4 Thesis Goal .................................................................................................. 25

3 Cumulative Local Outlier Factor .................................................................. 27
3.1 Overview ...................................................................................................... 27
3.2 A Basic Technique ...................................................................................... 28
  3.2.1 Local Outlier Factor .............................................................................. 28
  3.2.2 Incremental LOF .................................................................................. 29
3.3 Proposed Technique ..................................................................................... 37
  3.3.1 Overview .............................................................................................. 37
  3.3.2 Maintenance of active data points .......................................................... 38
  3.3.3 Maintenance of virtual data points ......................................................... 40
  3.3.4 Summary .............................................................................................. 44
3.4 Message-passing Implementation ............................................................... 46
  3.4.1 General Framework ............................................................................... 46
  3.4.2 Coordinator Actor ............................................................................... 49
  3.4.3 Window Actor ...................................................................................... 50
  3.4.4 Cluster Actor ....................................................................................... 50
  3.4.5 LOF Processor Actor ........................................................................... 51
  3.4.6 K-NN Actor ......................................................................................... 52
3.5 Complexity Analysis ..................................................................................... 52
  3.5.1 Overview .............................................................................................. 52
  3.5.2 Time Complexity .................................................................................. 53
  3.5.3 Space Complexity ............................................................................... 56
## 4 Cumulative Kernel Density Estimator with Retrospect

4.1 Overview ................................................................. 58

4.2 Methodology ............................................................ 59

4.2.1 SOD_GPU in General .............................................. 59

4.2.2 Kernel Density Estimator ......................................... 60

4.2.3 Sliding Window ....................................................... 61

4.2.4 Binned Summary ..................................................... 62

4.2.5 Threshold and Outlier Factor .................................... 66

4.2.6 Concept Drift Detection ........................................... 67

4.3 Parallel Implementation ............................................... 68

4.3.1 General Framework ................................................ 68

4.3.2 Density estimation on GPU ........................................ 69

4.3.3 Binned summary maintenance on CPU .......................... 74

4.4 Complexity Analysis .................................................. 75

4.4.1 Overview ............................................................. 75

4.4.2 Time Complexity .................................................... 76

4.4.3 Space Complexity ................................................... 77

## 5 Experiments and Results

5.1 Overview ................................................................. 79

5.2 Datasets ................................................................. 80

5.2.1 Synthetic Dataset .................................................... 80

5.2.2 Real-life Dataset ..................................................... 80
5.2.3 Streaming Environment ................................. 81

5.3 Evaluation Criteria ........................................ 82
  5.3.1 Accuracy Evaluation ................................. 82
  5.3.2 Performance Evaluation ............................... 83

5.4 Benchmark and Results .................................... 83
  5.4.1 C_LOF .................................................. 83
  5.4.2 C_KDE_WR ............................................. 86

6 Conclusion and Future Works 90
  6.1 Overview ............................................... 90
  6.2 Conclusions ............................................. 91
  6.3 Future Works ............................................ 92

APPENDICES 101
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Time takes to process one window in C_LOF with different threads size</td>
<td>85</td>
</tr>
<tr>
<td>5.2</td>
<td>Latency of Kernel 2 &amp; 3 with different block size</td>
<td>89</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Evolving 2D data stream[22] .................................................. 3

2.1 The LOF caption ................................................................. 18

2.2 The LOF caption ................................................................. 20

3.1 The Insertion operation in Incremental Local Outlier Factor (LOF) algorithm [30] .................................................. 30

3.2 The Deletion operation in Incremental Local Outlier Factor (LOF) algorithm [30] .................................................. 33

3.3 The actors used in C_LOF algorithm and their relationships ............ 47

3.4 The sequence diagram of C_LOF algorithm based on message-passing implementation .................................................. 49

4.1 Maintenance of bin summary [22] ............................................. 64

4.2 GPU Grid and thread block layout for Kernel 2 and Kernel 3 [22] ........ 73

5.1 Accuracy of C_LOF on different parameters of k on Synthetic dataset 1 . . 84

5.2 Accuracy of C_LOF on different parameters of k on Synthetic dataset 2 . . 84

5.3 Accuracy comparison between C_KDE_WR and SOD_GPU on Synthetic dataset 1 . . .................................................. 86

ix
5.4 Accuracy comparison between C_KDE_WR and SOD_GPU on Synthetic dataset 2 ................................................... 87
5.5 Accuracy comparison between C_KDE_WR and SOD_GPU on KddCup99 dataset ................................................... 87
5.6 Accuracy comparison between C_KDE_WR and SOD_GPU on CoverType dataset ................................................... 88
Chapter 1

Introduction

1.1 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 [22]. An interesting property of outliers is that they form minority groups in the dataset, and their patterns can be recognized from the distributions in datasets themselves rather than relying on a separate training set, which is labelled and expensive to generate in most cases. Data mining without labelled data is also called unsupervised learning from a machine learning perspective. A very popular task of unsupervised learning is clustering, where similar data points form into a cluster repeatedly until all data points are assigned into a group. Outlier (anomaly) detection works the other way round. Rather than finding the clusters, which consist of majority of data points, it finds spatial data points that do not seem to belong to any clusters. Applications of outlier detections occur in numerous fields, including fraud detection, network intrusion detection, environment monitoring, etc.

A data stream is a continuous, unbounded sequence of data records accompanied and ordered by implicit or explicit timestamps [33]. One important property of data streams is that they are transient, which means that data points are only available partially at any
given point in time, and random access on the entire dataset is not possible. Moreover, data streams are characterized by *uncertainty* and *concept drift*. *Uncertainty* means that data points are vulnerable to external events (noise) and thus are unreliable [39]. *Concept drift* means that the distribution of data points is not fixed, and they may change over time [23]. Apart from these considerations, when working on applications that process data streams, their temporal contexts need to be considered. In addition, the processing needs to be online or incremental, and data are processed one after the other to leverage the requirement on computational and memory resources. This makes data mining over data streams a challenging task, and therefore it forms a new research area over last decades.

1.2 Motivation

Online outlier detection over data streams can be very challenging because of the concept drift property of data streams. To be more specific, suppose you have a data stream that keeps coming indefinitely; at one time $t_i$, you identify object $o_k$ as being an outlier in the current window; And after some time $T$, at time $t_i + T$ when the entire recent history is considered, object $o_k$ may become an inlier. See Figure 1.1. Good algorithms for outlier detection over data streams should support concept drift detection over a time range since the same data point could be outlier in a specific time period but inlier in another [26].
In addition, all data points from a data stream have a temporal context since data streams are considered as infinite series of data points and each point has a timestamp that is associated with it to indicate its time of arrival. This means that when detecting outliers in a data stream, rather than treating each point in the entire history of data equally, the time notation of each observed point is different should be considered differently.

The definition of outliers is application-specific and in most case vague [33]. Different applications and users may have different definitions of outliers. It is generally considered difficult to produce definitions of outliers that can be applied in every case, and many results on outlier detection depend heavily on the effectiveness of data representation and distance metrics. A more stable approach is to use a supervised learning method where data are labelled to give guaranteed accuracy. However, as supervised learning requires expensive labelled data and it only works over a static dataset where data distribution is fixed; Hence, it is generally considered inappropriate to use supervised learning for data-mining over data streams. Unsupervised learning on the other hand does not require labelled data and the underlying data does not need to be fixed. In this thesis, we are only interested in the unsupervised learning approaches. The discussion of outlier detection is from a very general prospective (mathematical and algorithmic) and the specific application of the proposed algorithm to different fields is beyond the scope of this thesis.
As the rapid growing of modern applications, stream programming has become a must in many applications from different fields. Indeed, the increasing popularity of stream programming has led to a new research area compared to a long history of static data processing [9]. This is also true for outlier detection. Outlier detection over data streams can have many applications from different fields, including fraud detections, network intrusion detection, environmental monitoring etc. What is interesting in these applications is that we want to find abnormal behaviours over data streams in real-time, with very low latencies. These can be of vital importance in applications such as fraud detections for instance. Suppose you are a financial institution that issues credit cards to end customers and you are monitoring all transactions for each of your customers. You want to stop the fraud transactions as soon as possible to minimize loss for your end users, given that you have the availability of data streams (monitored transactions in this case). The same can be true for other applications where data may come in different formats, such as activity monitoring, where data come in form of video frames and you want to find suspicious behaviours (such as terrorism) in a crowd of people. Apparently, the quicker you detect such anomalies, less loss you are going to suffer.

This drives the motivation of this thesis to find a way to effectively pull out outliers from an unbounded sequences of data in real-time. In the literature, several methods have been introduced to solve this problem. Many of them are direct extensions to classical outlier detection algorithms that only work on batch data. Most of them fail to address the properties of data streams and therefore do not produce promising results. This thesis aims to overcome the challenges mentioned before with the goal of developing an efficient online algorithm to detect anomalies over data streams.
1.3 Our Contribution

The algorithms we proposed are based on two efficient outlier detection algorithms that have been used. The first one calculates the reachabilities between data points and is based on a classical algorithm called *Local Outlier Factor (LOF)* [27], which is a proximity (density)-based method for outlier detection. D. Pokrajac et al. have proposed an paper to make the LOF algorithm process incrementally [30]. In this thesis, we further extend this algorithm to make it work over data streams by employing *overlapping sliding window*, where the status of all points in the current window that do not expire are being continuously monitored for their LOF scores. All expired inlier points are grouped into clusters using a DBSCAN-like algorithm [16] and each cluster is represented by one *virtual data point*. These virtual points are combined with *active data points* in the current window and are fed into LOF algorithm in order to find outliers from active data points. We call this method *Cumulative Local Outlier Factor*, or *C_LOF* in short.

The second algorithm is based on a non-parametric algorithm from the statistical field, called *kernel density estimator*, which estimates the probability density function from all observed data and decides the outlier score of a data point by judging its probability density over the density function. In [22] C. HewaNadungodage et al. implemented a so-called SOD_GPU\(^1\) algorithm which is based on density estimation over *non-overlapping* window and *statistical binned summary*, accelerated by GPU to effectively detect outliers in continuous data streams. The results seem very promising as it is 20X faster compared to a multicore CPU implementation and it achieves a even higher accuracy rate compared to the other approaches in the literature. In this thesis, we further extend this SOD_GPU algorithm by introducing a novel concept drift detection module, which works by re-considering (retrospecting) the earlier detected outliers in future windows to reduce false positives and weighting each observed points based on its temporal context. We

\(^1\)Stream Outlier Detector-GPU
call this method *Cumulative Kernel Density Estimator with Retrospect*, or *C.KDE.WR* in short.

Both of these algorithms use *sliding-windows* to keep a number of active data points in the memory along with the statistical summaries (or clusters) of all expired data. Each algorithm has different advantages over the other: one requires pre-defined parameters but supports continuous data status monitoring; the other is fully parameterless and supports detection of outliers even in higher-dimensional data. The choice of which one to use is application-dependant.

We implement our first algorithm using *actor model* design, which leverages a technique in reactive programming called *message-passing*, making the application scalable and resilient to failures. In the implementation of our second algorithm, we use the CUDA\(^2\) framework, which is a parallel computing platform on NVIDIA GPUs to achieve high performance computing.

We do experiments of these two algorithms in both synthetic and real-life datasets and compare their accuracies with different parameter settings as well as with other similar methods in literature. We also do benchmark on implementations of these two algorithms in terms of their latencies. Results of these experiments will be analyzed and discussed in details.

### 1.4 Thesis Organization

We begin the thesis by formalizing the problems and conducting the literature review on outlier detection over dynamic data streams in Chapter 2. Several methods will be mentioned, including *Local Outlier Factor* and *Kernel Density Estimator*, which are what our proposed algorithms are based on.

\(^2\)Compute Unified Device Architecture: https://developer.nvidia.com/cuda-zone
In Chapter 3, we will introduce our first proposed algorithm, namely \textit{C\_LOF}, based on Local Outlier Factor, and explain the algorithm in details. We will mention how to implement the LOF algorithm in an incremental fashion based on the work done by D. Pokrajac et al. [30] and how we extend it to work over data streams with the help of overlapping sliding window and clustered virtual data points. We will explain how we implement the algorithm using actor model to achieve both scalability and fault tolerance.

In Chapter 4, our second algorithm will be introduced, namely \textit{C\_KDE\_WR}, based on Kernel Density Estimator. We will explain the SOD\_GPU algorithm proposed in [22] in details, including how it applies kernel estimator to work over data streams by mining non-overlapping window and statistical binned summary. We will explain how we deal with concept drift by treating each detected outliers as candidates and retrospect our decisions based on data to be received in future. Points in the binned summary are also weighed by forgetting factor to address the temporal notion. Additionally, we will also discuss how we implement the algorithm in GPU with help of CUDA framework and leverage the coalesced memory access and CUDA streams techniques to give low latency on memory bandwidths.

Chapter 5 shows the experimental results of our two algorithms against testing datasets in terms of accuracy and performance. And Chapter 6 concludes our work, and includes discussions on the potential future work of this thesis.
Chapter 2

Problem Statements and Related Work

2.1 Overview

In this chapter we will first focus on the definitions of outliers and data streams. We will then discuss the challenges that need to be addressed about outlier detection over data streams. Then we will shed light on several works related to anomaly detection, starting from the supervised methods from the machine learning field. We will also mention some unsupervised methods over static data, including some of the most widely used in real-world. For some of these methods, we will introduce works that have extended them so to work over data streams. One particular type of technique will be emphasized here, namely sliding-window, which is widely used to detect outliers over data streams. Lastly, we will formalize our goals and what we are trying to achieve in this thesis.
2.2 Definitions and Problem Statements

2.2.1 Outlier Definition

An outlier is a data point which is significantly different from other data points, or does not conform to the expected normal behaviour, or conforms well to a defined abnormal behaviour [11]. However, the exact definition of outlier is still very vague and application dependant. For instance, the fraud records in credit card transactions data are very different from those “malicious” connection records in TCP connection data. Based on the classification given by [33], there are three types of outliers among all applications:

- **Type I outliers**: An individual data point that deviates largely in a dataset is termed as a Type I outlier. By definition they are the simplest type and it is very easy to identify them. Intuitively they are isolated points in the dataset in terms of attribute values [33].

- **Type II Outliers**: A data point that is isolated with respect to other data points in the context is called a type II outlier. The context refers to the semantic relationship among the data points’ attributes and such context can be something like time, location etc. This type of outlier was first studied by Barnett & Lewis [9] and most of them are associated with time series or temporal context [33].

- **Type III Outliers**: A particular group of data points that appear as outliers with respect to the entire dataset is termed type III outliers. No data point in a small subset is an outlier with respect to the other points in the subset, but as a group, they are outliers. Barnett & Lewis [9] called them Innovation Outliers and they are very difficult to identify in time series data since they can influence other data in the same context [33].

In case of data streams, each data point is associated with temporal information and
therefore, they are type II outliers. For example, a temperature of 30°C is not an outlier with respect to the entire days in a year, but is most likely an outlier if we consider the temperature in winter only. In addition to type II, an outlier in data streams can also be type III where sequence of behaviours are abnormal within a specific time range compared to the entire history. In case of network intrusion detection, a group of hackers may attack the network simultaneously and all of them are outliers that need to be detected individually despite of similarity within their group.

2.2.2 Data Streams Definitions and Properties

A data stream is a continuous, unbounded sequence of data records accompanied and ordered by implicit or explicit timestamps [33]. Unlike static data which is available for random access, the data in a stream is only available partially at each given point of time and the observed data may not be available in future. The formal definition of data stream is as follows by:

Definition 1. Data Stream: A data stream is a possible infinite series of data points \(o_{n-2}, o_{n-1}, o_n, \ldots\), where data point \(o_n\) is received at time \(o_n.t\).

Based on the nature of data stream, when designing an application that process data streams, several properties of data streams need to be taken into consideration, in particular: temporal notation; transiency; uncertainty; concept drift. Temporal notation is a temporal context that is associated with each data point that arrived from a data stream, this can be the timestamp of when the data point arrives in our applications [33]. Transiency means that unlike static data which is possible for random access, data from data streams are only partially accessible at each given point of time [33]. It is possible to store all observed data points in a secondary memory for random access but data from the future are still inaccessible at the current time. Like other static dataset, data from data streams can also has noises and therefore contain uncertainties [33]. Unlike outliers,
noises are within the variances of underlying distribution and they should be distinguished from outliers, which are completely not governed by the underlying distribution. *Concept drift* means that the underlying distribution of data streams may change over time and therefore any predictive model that assumes a fixed distribution will fail [23]. Indeed, the handling of *concept drift* is a main challenge of most data-mining tasks over data streams and will be discussed in details in the following.

### 2.2.3 Challenges of Outlier Detection over Data Streams

The properties of data streams mentioned in previous section give rise to many challenges in outlier detection over data streams. Apart from these, there are other challenges that are based on the nature of the outlier detection task itself. In the following, I will give detailed explanations for each of such identified challenges. These challenges are usually left out or not addressed properly when designing an algorithm or an applications that detect outliers over data streams in most literatures I reviewed by time of writing of this thesis. The increasing demands of data mining tasks over data streams have lead to a new research area that needs to be studied properly in terms of the problems that arise from it.

#### 2.2.3.1 Temporal notation

Each data point in a stream is associated with a timestamp that indicates its arrival time in our applications (implicitly or explicitly). The timestamps give data from streams a temporal context, which should not be ignored when processing the data over streams because it contains information of vital importance about the data that can be of a particular interest in our applications, like outlier detection for instance [33].

Surprisingly, many applications in the real world do no consider this notion of time when processing the data streams, or the temporal information is not addressed properly.
Ideally, data points should be compared more with those who share the same or similar temporal context. If the temporal information of a data point is far from the current one, it should have less impact on the current data. Otherwise, it should impact more. This notation of time gives freshness to the recent data and obsolete data should be gradually faded away. As for outlier detection over data streams, the consideration of temporal context will make sure that we detect type II outliers with time as its context from the data streams.

**Challenge 1.** *The data should be processed with its temporal context associated with it.*

### 2.2.3.2 Transiency

*Transiency* is the main property of data streams and it means that data are not available for random access and only part of data is available at each point in time. To address this transiency property of data streams, some applications adopt a so-called *store-and-process* fashion to process data streams, which means the entire history of data that have been observed is stored in a secondary memory and when new data points arrive, they are processed along side with new data points [33]. While this naive approach seems to work combined with offline algorithms, this method has serious scalability issue as the number of data size increases, especially when the memory requirement is a constraint. In addition, the algorithm itself is not online at all as the entire dataset is kept in memory. Instead, obsolete data should be removed from memory when they expire as they probably contribute little to the prediction of new data.

**Challenge 2.** *The streaming applications should not store the entire history of data indefinitely and when they expire, they should be removed immediately.*

The deletion of obsolete data results in the data streams to be processed in a *one-after-the other* fashion. Unlike static data, which stay on the disk and are available for batch
processing in any time, the processing of data streams needs to be online and normally requires only one-pass, which means each data point will only be iterated once.

**Challenge 3.** *The algorithms should be online or incremental that mainly process new data arrive in streams.*

Despite the deletion of obsolete data, the processing of newly arrived data should still consider the historical data to some extent in order to achieve a better model. This would require a summary be built on top of expired data that would cost significantly less memory as compared to the keeping of entire historical data.

**Challenge 4.** *The newly arrived data in streams should be processed by considering not only on data that share similar temporal context, but also on a summary that is mined from all historical data.*

### 2.2.3.3 Concept drift

Concept drift is the most notorious property of data streams and it means that the underlying data in streams may not form a fixed distribution. Specifically, this means that the data distribution may change over the time (new behaviours and trends) as new data are coming in an unpredictable way [23]. Dealing with concept drift is a main challenge in most streaming applications and that is also why most statistic based models fail when processing data streams because they always assume a fixed distribution. In an ideal situation, the algorithm should be capable to detect concept drift when it occurs in data streams in a timely manner to avoid producing large amount of false positives.

**Challenge 5.** *The streaming applications should build models that are adaptive and are capable of detecting such changes in data patterns to avoid false positives.*
2.2.3.4 Uncertainty

Uncertainty means that the data in streams may contain noises and thus become unreliable as they may come from different sources. In a real-world environment, the applications that collect data streams are exposed to an open environment and are vulnerable to external events and therefore making the data in them uncertain [33]. As for outlier detection, one challenge is to distinguish such noises from anomalies that are both contained within data streams. Some similarity metrics used in most data mining tasks such as Euclidian distances, Manhattan distances or even Cosine distances fail to measure similarity and dissimilarity when data are uncertain. In contrast, probability metrics work better in these scenarios compared to deterministic metrics because probability has uncertainty incorporated in it by its nature.

Challenge 6. The similarity metrics used should be able to measure similarities even when the data contains noises.

2.2.3.5 Multi-dimensionality

Data that has a large number of attributes is called multi-dimensional data. This number could be hundreds or thousands. Data that come from data streams can sometimes be high dimensional. The problem arises when we measure similarities between two data points that are high dimensional using similarity metrics such as Euclidian or Manhattan distance. Such distance metrics do not work and become meaningless to represent similarities between data points on higher dimension spaces [1]. For the calculation of density functions, the density space will grow dramatically as the number of dimensions increases, and therefore the computation time becomes expensive [35]. This is known as “curse of dimension” problem in many areas including data mining, machine learning etc. Normally this issue is solved by projections of higher dimensional data into lower dimensional spaces.
Challenge 7. The similarity metrics used should be able to measure similarities in high dimensional data or the data need to be projected into lower dimensional spaces.

2.2.3.6 Masquerading

Masquerading refers to the fact that outliers may attempt to hide within existing distributions and therefore can hardly be identified [30]. This means at different point in time, outliers may exist at different scales and, in order to identify them properly, the algorithm should be able to process data at different level of granularity. This would require the adaptively updates of hyper-parameters that govern the decisions of outliers to reflect such changes.

Challenge 8. The parameters that are used to define threshold, which decide outliers should be dynamically adjusted in order to avoid masquerading in existing data distributions.

2.3 Related Works

2.3.1 Outlier detection models with labelled data

The difference between supervised and unsupervised models is the presence of labelled data. Labelled data can be helpful in building predictive model that imply prior knowledge of data. However, they also have significant drawbacks when mining over streaming data.

2.3.1.1 Supervised model

Supervised learning typically requires building a prediction model for rare events based on manually labelled data (the training set), and use it to classify new event based on this learnt model [12, 24]. In other words, the outlier detection problem in this case becomes a classification problem where we are only interested in the minority class whose data
deviate largely from the rest. A thing to take note when using supervised method for training is imbalanced data: The predictive models developed using conventional machine learning algorithms could be biased and inaccurate because of the number of observations in one class of the dataset is significantly lower than the other. To handle imbalanced data, several methods can be used, including resampling, boosting, bagging [20, 38, 42, 45].

The main problem of using supervised methods is that, in order to ensure accuracy, large number of labelled data need to be generated, which is unpractical in practice. However, supervised approaches may give guaranteed performance when the data distribution is static. Some machine learning algorithms can be used in the supervised context to detect outliers, such as Support Vector Machines, Neural Network, K-Mean and KNN etc. A recent technique introduced by Harkins et al. [21] takes advantage of replicator neutral network (RNN) to detect outliers.

2.3.1.2 Semi-supervised model

To overcome the scarcity of labelled data in supervised learning, semi-supervised learning [6, 41] only requires a small number of training data with some unlabeled data to obtain better predictions. It is known that applying semi-supervised learning to anomaly detection can improve the detection accuracy [46]. One approach introduced by Jing Gao et al. [18] uses K-mean clustering in unsupervised learning, adding penalties to the objective function for mislabelled data points and optimizes the overall objective function.

Although some of these techniques discussed above may generate very promising results, they work well only with static data and typically do not fit into the context of dynamic streaming context. In other words, both supervised and semi-supervised methods will assume that they have random access over the underlying data, while this is not possible for data streams because of its transiency property. Another problem with supervised approaches is that they fail to capture the changes of data pattern since they
assume a fixed data distribution and therefore violate the concept drift property of data streams. Therefore, it is considered inappropriate to use supervised method (including semi-supervised) over data stream for outlier detection.

2.3.2 Outlier detection models without labelled data

In contrast, unsupervised learning methods do not require labelled input and typically do not assume a fixed data distribution as the model can be dynamically built based on variations of data. Many best-known techniques of outlier detection fall into this category. They can mostly be classified into two categories: unsupervised outlier detection over static data and unsupervised outlier detection over data streams. In this thesis, we are only interested in outlier detection over data streams. But most of these algorithms are developed over static data. We will look into some of those classical algorithms that work over static data and discuss those work over data streams.

2.3.2.1 Distance-based model over static data

The Distance-based model introduced by Knorr and Ng [25] 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 some variants of the static distance-based approaches, and their ideas are quite similar. For instance, Ramaswamy et al. [31] 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 [4] introduced a method where an outlier is defined by taking into account the sum of the distances from 1st to the \( k^{th} \) nearest neighbours. These methods are sometimes referred as KNN in the domain of machine learning.
2.3.2.2 Density-based model over static data

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. A popular example of this approach is Local Outlier Factor (LOF) algorithm [27], on which one of our proposed algorithms is based. It uses the concept of reachability to coin the density of data points: the density of each data point is measured by considering the reachability of this data point, in regards to the reachabilities of its neighbours’, as shown in Figure 2.1. Another popular density-based method is called LOCI (Local Correlation Integral), which use Multi Granularity Deviation Factor (MDEF) to measure how the neighbourhood count of a particular data point compares with that of the values in its sampling neighbourhood [29].

![Local Outlier Factor (LOF) algorithm](http://scikit-learn.org/stable/auto_examples/neighbors/plot_lof.html)

Figure 2.1: The Local Outlier Factor (LOF) algorithm

2.3.2.3 Statistical-based (Parametric) model over static data

The Statistical-based model is also known as parametric model. This technique is formulated by using the distribution of the data points available for processing. The detection

---

1 Figure from [http://scikit-learn.org/stable/auto_examples/neighbors/plot_lof.html](http://scikit-learn.org/stable/auto_examples/neighbors/plot_lof.html)
model is formulated to fit the data with reference to the distribution of data [36] and normally models the underlying data using a mixture of distributions (e.g. Gaussian distribution). One of the most popular one used is the *Gaussian mixture* model [43], where each data point is given a formulated score, and data points which have a high score are declared as 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 some metrics against the predictive model. Statistical-based models are usually computational inexpensive but most of them require parameters as inputs and they also assume a fixed distribution in dataset, which in most case do not fit into streaming context.

### 2.3.2.4 Deviation-based model over static data

The *Deviation-based* model is an approach developed from the *statistical-based* model; In this model first introduced by Arning et al. [5], an outlier is detected if the feature space of one data point deviates largely from other data points (in local or global set) and the variance is minimized when removing such a point. Aggarwal and YU [2] proposed a technique where a point is an outlier if, in some lower dimensional projection, it is present in a local region of abnormally low density. This method is also efficient with high dimensional data.

### 2.3.2.5 Sliding-window-based model over data streams

To address the temporal notation of data streams in outlier detection, many techniques have been introduced. Of all those techniques, one of the most popular is called *sliding-window*, where a portion of data that share the same temporal context is kept in memory, and outliers are decided based on those data in memory. Windowed data are deleted from memory when they expire. Each window of size $|W|$ has a starting time $T_{\text{start}}$, an
ending time $T_{end} = T_{start} + |W|$. Based on the definition of $W$, there are two types of sliding-window: time-based window and count-based window:

**Definition 2. Count-based window:** Given data point $o_n$ and a fixed number of data points $W$, the count-based window $D(n)$ is the set of $W$ data points: $o_{n-W+1}, o_{n-W+2}, \ldots, o_n$; And window size $|W| = o_n.t - o_{n-W+1}.t$

**Definition 3. Time-based window:** Given data point $o_n$ and a fixed time period $W$, the time-based window $D(n)$ is the set of $n - n' + 1$ data points: $o_{n'}, o_{n'+1}, \ldots, o_n$ where $o_n.t - o_{n'}.t = W$. And window size $|W| = W$

In time-based window, the window size $|W|$ is defined by a fixed amount of time interval where in count-based window, the window size $|W|$ is defined by the time elapsed within a given amount of data points. Based on the triggering time $T_{start}$ of each slide with respect to window size $|W|$, sliding-window can also be categorized into non-overlapping window and overlapping window. If the triggering time $T_{start}$ of each slide is in concert with the window size $|W|$, in which case $|W| = |Slide|$ where $|Slide|$ is the length of each slide, the window is called non-overlapping window and each windowed data will experience only one slide during its lifetime (see Figure 2.2 left). Otherwise, the window is called overlapping window, where each windowed data will experience $\frac{|W|}{|Slide|}$ slides during its lifetime (see Figure 2.2 right).

![Figure 2.2: Non-overlapping sliding window (left) vs Overlapping sliding window (right)](image)
In the case of **count-based window** where \(|\text{Slide}| = 1\), slides are triggered upon each individual data point’s arrival or departure, the *sliding-window* becomes fully reactive to events. We shall encounter this type of *sliding-window* in later chapters when we introduce the \(C_{LOF}\) algorithm.

### 2.3.2.6 Distance-based Outlier Detection over Data Streams (DODDS)

Since the very first distance-based method for outlier detection proposed by Knorr and Ng [25] is based on static data, several methods have been proposed to extend it to detect *distance-based outlier detection in data streams (DODDS)* [3, 10, 19, 26, 28, 32, 44]. One of the most popular methods used to detect *distance based* outliers over data streams is to use *sliding-window* to help with detection of outliers. Range queries are performed over data points in *sliding-window* to detect outliers. Based on the benchmark among all DODDS algorithms given by Luan Tran et al.[40], the MCOD algorithm introduced by M.Kontaki et al.[26] seems to have the most satisfying performance. In [26], it checks whether an outlier object has become inline due to the arrival of new data points or whether an inlier object has become outlier due to the departure of expired object using 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.

Another type of *distance-based* outlier detection model over data streams is based on the approximation of *probability density function (pdf)*, usually with *kernel density estimator* (explained shortly) [28, 32]. The distance is measured based on the density of a data point in the estimated pdf around a user defined radius. Sadik et al. first proposed a novel

---

2 Figure from [https://ci.apache.org/projects/flink/flink-docs-release-1.5/dev/stream/operators/windows.html](https://ci.apache.org/projects/flink/flink-docs-release-1.5/dev/stream/operators/windows.html)
binned implementation of kernel density estimator to detect outliers without necessity to store all observed data and outliers are detected if the distance is below a threshold (DBOD-DS) [28]. To deal with the concept drift of data streams, they further improved the method by introducing concept drift detection module to handle change of distribution in data (A-ODDS) [32]. To handle the curse of dimension problem, they also proposed a framework, called Orion [34], which addresses all the characteristics of data streams and looks for projected dimension of high-dimensional data points using evolutionary algorithm to detect outliers in multi-dimensional data streams.

Most of DODDS algorithms are based on the original definition of distance-based technique given by Knorr and Ng [25]. The other distance-based techniques in outlier detection have not been studied in the streaming context. It would be interesting to see if those methods can be extended to work in data streams. Since all DODDS methods only have access to only a portion of data, they all lack a global view of the entire dataset and in most cases, there accuracies are not guaranteed.

### 2.3.2.7 Clustering-based model over data streams

The clustering-based model is another technique to outlier detection over stream data. Two main algorithms exists for clustering-based approaches. One of them is called K-Mean clustering [15], which also usea the idea of sliding window and clusters the data in each window. 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 to further compare with data in other windows. If the candidate outlier passed a given number of windows, it is then identified as true outlier. Compared to K-Mean clustering, K-Median clustering [14] clusters each chunk of data into a variable number of clusters (from \( k \) to \( k\log(n) \) where \( n \) is the data size and \( k \) is the KNN parameter), and it passes the weighted medians found in current window into next one for testing outlierness
rather than the mean and candidate outliers. Both of these two approaches will require users’ input of value $k$ but the number of clusters in K-Median clustering is not fixed.

### 2.3.2.8 Density-based model over data streams

We discussed two density-based models for outlier detection over static data, namely LOF and LOCI. Here, another version of LOF algorithm was proposed to detect outliers over data streams. In [30], D. Pokrajac et al. presented an incremental version of the Local Outlier Factor algorithm over data streams. They gave theoretical evidence that the insertion of new data points as well as deletion of an old data point affects only a limited number of neighbours and does not depend on the total number of records in current dataset and is bounded by $O(n \cdot \log n)$ time complexity after insertion of $n$ data points.

### 2.3.2.9 Auto Regression model over data streams

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 [13]. It is very popular for time series outlier detection and its definition is given by

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

where $x(t)$ is the series under investigation, $a_i$ are the autoregression coefficients, $n$ is the order of the autoregression and $\xi(t)$ is the noise and is almost always assumed to be a Gaussian white noise. Based on this formula, we can estimate the coefficient parameters $a_i(t)$ based on the given time series of $x(t),...x(t-n)$. The model can then be used to predict future time series by defining a threshold, called cut-off limit and the data point is identified as an outlier if it is beyond this threshold.
2.3.2.10 Kernel density (Non-parametric) model over data stream

The Kernel density estimator (KDE) is a non-parametric method to estimate probability density function of random variables [35]. It has become increasing popular in recent year as an efficient way to detect outliers over data streams. The probability density function $f(x)$ is given by:

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

where $k_{h_i}(x)$ is the kernel functions with bandwidth $h_i$. The kernel functions distribute the occurrence of a data points into its neighbourhood regions and therefore, after observing enough data points, the density function can be curved. Furthermore, the bandwidth can be calculated online using Scott’s rule [35] as new data points are being observed.

Several works have been proposed to use this method for online outlier detection over data streams. A technique inspired from sensor network is mentioned in [37], where it uses a KDE to model the distribution of the sensor data. In [22], Yuni Xia et al. use GPU to accelerate kernel density estimator with helps of non-overlapping sliding window and a statistical binned summary to detect outliers in high volume and high dimensional data streams. In this method, the outlierness is considered not only based on data points in current window, but also based on historical data that are mined efficiently into bins.

2.3.2.11 Summary

Unsupervised methods does not require labelled data and they can also be modified so that the algorithms can work sequentially because they do not require a training stage. In addition, unsupervised methods do not assume a fixed data distribution, and outliers patterns can be learnt dynamically as data change. Therefore, it is considered appropriate to use unsupervised methods for outlier detection over data streams.
Most of the unsupervised outlier detection methods over data streams employ a sliding-window to keep a number of active objects in memory and delete them from memory when they expire \([3, 10, 19, 22, 26, 44]\). Some of them design the algorithms in a way that is online or incremental \([28, 30, 32]\). The usage of sliding-window can address some of the properties of data streams (like transiency, temporal notation) but not all of them. The effectiveness of sliding-window depends heavily on the choice of window size. Other algorithms refer to statistical summaries mined from historical data to help with the prediction of active data points in current window \([22]\). Keeping summaries can be helpful to some extend by considering historical data, but it does not consider the future data, to which we do not have access yet, therefore producing higher false positive rate when concept drifts occur. To address this problem, the algorithms either need to support continuous monitoring of outlier status of data points in current window over some periods \([3, 26]\) or instead of reporting immediately when outliers are detected, the detected outliers should be treated as candidate outliers and only consider them as true outliers after observing enough future data points \([14, 15]\). In \([28, 32]\), temporal information in data streams are also considered by introducing \textit{forgetting factor} \(\lambda\). Historical data that have not be changed will be gradually faded and impact less on recent data.

\subsection*{2.4 Thesis Goal}

Since the unsupervised methods of outlier detection can effectively address most of the aforementioned properties of data streams, the goal of this thesis is to develop and implement efficient algorithms that could detect type II outliers (temporal context) over data streams in real-time without labelled data. Specifically, the algorithms should be online or incremental and are capable of handling concept drift of data streams. Additionally, the algorithms will not rely on storing the entire history of observed data in secondary memory but instead, keep only an efficient summary of previous data that could fit into
primary memory and use it to help with predictions of future outputs. From the engineering perspective, we also want our solutions to be scalable to the number of data volumes with low latency on results generation so that real-time application is feasible. We propose two algorithms, both of which, are based on unsupervised methods that work in different contexts. We leave the study of supervised mining over data streams a future research interest.
Chapter 3

Cumulative Local Outlier Factor

3.1 Overview

In this chapter we will introduce our first algorithm, namely Cumulative Local Outlier Factor (C_LOF) that is based on a very popular outlier detection technique, called Local Outlier Factor (LOF), which works by calculating proximities (reachability distances) of data points and compare them with their neighbours’. In C_LOF, we use a count-based overlapping sliding window to maintain active data points and incrementally update their proximities as new data arrive or old data expire. This process works exactly in the same way as discussed in literature [30]. Furthermore, we also keep statistical summary of historical data to help predict the proximities of active data points, which gives novelty to this algorithm. Expired inlier points are clustered as virtual data points and combined with active points in the window to execute the LOF algorithm incrementally. The outlier status of active data points can be monitored and vary as new data points arrive in the sliding window. This is done in order to deal with the concept drift of data streams. Moreover, to incorporate the temporal context in data streams and identify Type II Outliers, we introduce the forgetting factor \( \lambda \) on all virtual points. In the following, we will give the formal procedures of C_LOF algorithm as well as its implementation based on actor model.
3.2 A Basic Technique

3.2.1 Local Outlier Factor

Let us first look at the classical LOF algorithm proposed by Breunig et.al in 2000 [27]. The main idea of the LOF algorithm is to assign to each data point a degree (or score) of being outlier; this degree is called \textit{Local Outlier Factor}(LOF) of the data point. The metrics measure the density of a data point compared with respect to its neighbourhood (K-nearest neighbours). The computing of LOFs for all data points typically comprise of following steps [27]:

1. For each data point \( p \), compute its \( k\)-distance\( (p) \) i.e. the distance to its \( k \)-th nearest neighbour of \( p \)

2. For each data point \( p \), find its \( k\)-distance-neighbourhood of \( p \), which contains every object \( q \) whose distance to \( p \), noted as \( d(p, q) \) is not greater than \( k\)-distance\( (p) \)

3. For each data point \( q \) in the \( k\)-distance-neighbourhood of \( p \), calculate its reachability distance with respect to data record \( p \) as follows:

\[
reach-dist(p, q) = \max(d(p, q), k\text{-distance}(q))
\]  

(3.1)

4. For each data point \( p \), calculate its \textit{local reachability density}(lr\( d \)) of \( q \) as inverse of the average reachability distance over \( k \)-nearest neighbour of \( p \):

\[
\text{lr\( d \)}(p) = \frac{1}{\sum_{k \in knn(p)} \text{reach-dist}(p, q)/k}
\]

(3.2)

5. And finally, for each data point \( p \), calculate its \textit{LOF} as ratio of average \( \text{lr\( d \)} \) over
k-nearest neighbour of \( p \) and \( lrd \) of \( p \) itself

\[
LOF(p) = \frac{1}{k} \sum_{k \in knn(p)} lrd(p)
\]

We assume that distances between each pair of data points are different and in the original publication, \( k \) was also named \( MinPts \), which means the minimum number of data points in a cluster in order to consider this cluster as inliers [27]. The outlierness is detected once the LOF value of a data point \( p \) deviates largely from the average value of LOF in the population. This is often controlled by hyper-parameters that defined the maximum threshold \( \theta \) that the algorithm can tolerate (as inliers). Based on our experiments, inliers turn to have LOF scores below 1. However, the exact value of \( \theta \) is very application-dependant and often requires tuning.

### 3.2.2 Incremental LOF

To address the challenge of applying LOF over data streams, an incremental LOF algorithm was proposed in [30]. The incremental LOF works by constantly maintaining \( k \)-distances, \( lrd \) and \( LOF \) values for all existing points and incrementally updating these values whenever a new data point is inserted or an obsolete data point is deleted. Since the static LOF algorithm has time complexity of \( O(N \cdot \log N) \), if we apply LOF algorithm \textit{iteratively} after observation of \( N \) data points, the algorithm will have a time complexity \( O(N^2 \cdot \log N) \) [27]. In [30], they proved theoretically that the insertion and deletion of data points actually only affect a limited number of existing data points (neighbours) rather than the total number of data points in dataset, and therefore the total complexity of incremental LOF algorithm can be bounded by \( O(N \cdot \log N) \). In addition, they also illustrated that the result of applying their incremental LOF algorithm is the same as the result of applying the static version of LOF algorithm after receiving \( N \) data points and it is also independent of the order of the insertions. In the following, we will show the insertion and deletion
operations of incremental LOF algorithm in details. Apart from insertion and deletion, we also introduce an update operation which happens when the position of a data point within dataset has changed.

### 3.2.2.1 Insertion

Whenever a new data point is inserted into the existing dataset, its LOF value will be computed immediately and decided whether it is an outlier. In addition, LOF values for existing data records are updated if needed. Notice that for insertion, if a data point is judged as outlier at its time of insertion, it can become an inlier as new data points are observed. On the other hand, if a data point is judged as inlier at its insertion time, it can never become an outlier unless some data points are deleted.

![Figure 3.1: The Insertion operation in Incremental Local Outlier Factor (LOF) algorithm](image)

As for the insertion part, the algorithm performs two steps: a) Insertion of new record and computation of its $k$-distance, $lrd$ and LOF values; b) maintenance, updating $k$-
distance, lrd and LOF values for affected existing points. Let’s first consider the example of inserting a new data point \( p_n \) into a dataset as shown in Fig 3.1. Assume \( k = 2 \), we need to first compute \( k\)-neighbourhood of newly inserted point \( p_n \) (points 4, 6 in Fig. 3.1 a), noted as \( k\)-NN(\( p_n \)) as well as data points \( p \) that have the new point \( p_n \) in their \( k\)-neighbourhood (points 1,3,4,6 in Fig 3.1 a), noted as \( k\)-RNN(\( p_n \)). In [30], it proves that the insertion of new point \( p_n \) may only decrease the \( k\)-distances of certain points in \( k\)-RNN(\( p_n \)). According to Eq. 3.1, when a \( k\)-distance changes for a point \( p \), the reachability distances will only be affected for point \( q \) such that \( q \in k\)-NN(\( p \)). In our example, the 2-neighbours of data point 3 are the data points 2 and 11. So reach-dist(2,3) and reach-dist(11,3) will be updated as in Fig. 3.1 b. According to Eq. 3.2, the lrd value of a point \( q \) is affected if: a) \( q \in k\)-RNN(\( p_n \)); b) For all \( p \in k\)-RNN(\( p_n \)) and \( q \in k\)-NN(\( p \)) such that \( p \in k\)-NN(\( q \)) or \( q \in k\)-RNN(\( p \)). The first refers to all the points \( q \) such that their \( k\)-neighbourhood changes (because of the inclusion of \( p_n \)). The second refers to all points \( q \) such that its reachability distance to at least one of its \( k\)-neighbours changes. In our example, points 1, 3, 4, 6 have point \( n \) in their 2 nearest neighbours (Fig. 3.1 b) and points 2, 5, 7 have their reachability distances updated among their 2 nearest neighbours (Fig. 3.1 c). So in total, points 1, 2, 3, 4, 5, 6, 7 will all have their lrd values updated (Fig. 3.1 d). According to Eq. 3.3, the LOF values of a point \( q \) should be updated if: a) lrd(\( q \)) is updated (points 1, 2, 3, 4, 5, 6, 7 in Fig. 3.1 d); b) for all \( p \) such that lrd(\( p \)) is updated, and \( p \in k\)-NN(\( q \)) (points 8, 9, 10 in Fig. 3.1 d). The details of insertion operation in incremental LOF algorithm can be summarized in Algorithm 1.

### 3.2.2.2 Deletion

Unlike insertion, the deletion operation only needs to re-compute LOF values for existing records and this may cause some inlier points to become outlier but it can never cause an outlier point to become inlier. For the deletion part, the algorithm only needs to perform the maintenance step and adjust decisions on some of existing data points based on its
Algorithm 1 Incremental LOF Insertion (Dataset S, Point points)

for all $p_n \in \text{points}$ do
    $S_{update \_ k \_ distances} = k\text{-RNN}(S, p_n) \cup p_n$;
    $\text{Insert}(S, p_n)$;
    $S_{update \_ lrd} = S_{update \_ k \_ distances}$;
    for all $p \in S_{update \_ k \_ distances}$ do
        Compute $k$-distance($S, p$);
        for all $q \in k$-NN($S, p$) do
            if $q \notin S_{update \_ lrd}$ and $p \in k$-NN($S, q$) then
                reach-dist($q, p$) = $k$-distance($S, p$);
                $S_{update \_ lrd} \cup q$;
        $S_{update \_ LOF} = S_{update \_ lrd}$;
    for all $p \in S_{update \_ lrd}$ do
        Compute $k$-NN($S, p$);
        for all $q \in k$-NN($S, p$) do
            Get/Compute reach-dist($p, q$) using Eq. 3.1;
            Update $lrd(p)$ using Eq. 3.2;
            $S_{update \_ LOF} \cup k$-RNN($p$);
    for all $p \in S_{update \_ LOF}$ do
        Get $lrd(p)$;
        for all $q \in k$-NN($S, p$) do
            Get $lrd(q)$;
        Update LOF($p$) using Eq. 3.3;
newly calculated LOF value. Similar to insertion, the deletion of a data point $p_n$ will only affect $k$-distances of data points in $k$-RNN($p_n$). Rather than decrease, their $k$-distances are increased to distances to their $k + 1$ nearest neighbours before deletion of the data point $p_n$. As for the reachability distances, when the $k$-distance value changes for a point $p$, reachability distances will only be affected for point $q$ such that $q \in (k-1)$-NN($p$) since the reachability distance from $p$ to its $k^{th}$ neighbour is already equal to their Euclidean distance [30].

![Diagram](image)

Figure 3.2: The Deletion operation in Incremental Local Outlier Factor (LOF) algorithm [30]

Consider the example in Fig. 3.2, where $k = 3$. Prior to the deletion of point $p_c$, $p_c$ is within the $k$ nearest neighbours of point $p_j$ and the radius of circle indicates the $k$-distance of point $p_j$ as shown in Fig. 3.2 a. The reachability distances from $p_j$ to any other point within the circle are defined by this value. When point $p_c$ is deleted, the $k$-distance of $p_j$ is enlarged as shown by the radius of circle in Fig. 3.2 b. Only the reachability distances between $p_j$ and two of its nearest neighbours (denoted by dashed lines) need to be updated because the reachability between $p_j$ to its third nearest neighbour remain unchanged. And they are updated to the new radius defined by the circle.

The other steps of the deletion operation, such as update of lrd and LOF values are
almost identical to the insertion counterpart. Therefore, I will ignore them for now. The
details of deletion operation in incremental LOF are summarized in Algorithm 2

\begin{algorithm}
\caption{Incremental LOF deletion (Dataset $S$, Point points)}
\For{all $p_n \in \text{points}$} {
    $S_{update,k\text{distances}} = k-RNN(S \setminus \{p_n\}, p_n)$
    \textbf{Remove}($S$, $p_n$);
    $S_{update,lrd} = S_{update,k\text{distances}}$;
    \For{all $p \in S_{update,k\text{distances}}$} {
        \textbf{Compute} $k$-distance($S, p$);
        \For{all $q \in (k-1)$-NN($S, p$)} {
            \If{$q \notin S_{update,lrd}$ and $p \in k$-NN($S, q$)} {
                $\text{reach-dist}(q, p) = k$-distance($S, p$);
            }
            $S_{update,lrd} \cup q$;
        }
    }
    $S_{update,LOF} = S_{update,lrd}$;
    \For{all $p \in S_{update,lrd}$} {
        \textbf{Compute} $k$-NN($S, p$);
        \For{all $q \in k$-NN($S, p$)} {
            \textbf{Get/Compute reach-dist}($p, q$) using Eq. 3.1;
        }
    }
    \textbf{Update} $lrd(p)$ using Eq. 3.2;
    $S_{update,LOF} \cup k$-RNN($p$);
    \For{all $p \in S_{update,LOF}$} {
        \textbf{Get} $lrd(p)$;
        \For{all $q \in k$-NN($S, p$)} {
            \textbf{Get} $lrd(q)$;
        }
    }
    \textbf{Update} $LOF(p)$ using Eq. 3.3;
}\end{algorithm}

3.2.2.3 Update

For the update operation, we refer to the positions of some points within dataset have
changed. This can get complicated as the change of position of a particular point within
dataset may cause $k$-distances of some points in dataset to decrease while it can also cause
$k$-distances of other points in dataset to increase. Other than that, the change of position
of any point will break the $K$-NN relationship that has previously established among data
points. Particularly, the $k^{th}$ neighbour of an updated point $p_n$ may change due to the
change of position of other points in dataset. The safest option for updating a point $p_n$
is to first perform the delete operation on point $p_n$, followed by an insert operation on
point $p_n$ based on its new position. While this will guarantee the correctness of the update operation, the execution maybe very time consuming, since large amount of $K$-$NN$ and $K$-$RNN$ range queries need to be executed for insert and update operations. In addition, when the change of position of a point $p_n$ is very tiny, it will unlikely to cause the $K$-$NN$ relationships among data points to change and therefore executing delete and insert operations will result in many redundant range queries. Therefore, we need to simplify the update operation when the change of position is tiny.

For the update operation, we assume that, when the change of position (in terms of Euclidian distance) of a point $p_n$ is within a threshold $\varepsilon$, for all point $p'$ in the dataset, the $k^{th}$ nearest neighbour of point $p'$ would not change. Therefore, the $k$-distances will only change for those points $p_c$ whose $k^{th}$ nearest neighbour is point $p_n$, noted as $r$-$k^th(p_n)$, and point $p_n$ itself. For points $p_c \in r$-$k^th(p_n)$, their $k$-distances should be updated based on the new Euclidian distances between $p_c$ and $p_n$. We also need to consider the reachability-distances that have been affected by such changes of $k$-distances and re-calculate $lrd$ values for those points affected just as in insertion and deletion operations. For point $p_n$ itself, we need to re-calculate its $k$-distance to its $k^{th}$ neighbour since its position has changed. Since $p_n$’s $k$-distance is updated, the reachability distances between all points in its $k$ neighbours, $k$-$NN(p_n)$ to point $p_n$ have also been updated, and therefore for all points $q \in k$-$NN(p_n)$, if $q$ satisfies $p_n \in k$-$NN(q)$ or $q \in k$-$RNN(p_n)$, its $lrd$ values should be updated. As a result of change of position of $p_n$, the Euclidian distances from every points to $p_n$ have changed. Therefore, the reachability distances from any point $q$ such that $q \notin k$-$NN(p_n)$ to $p_n$ will change. As a result of this, the $lrd$ value will change for those points $q$ such that $q \in k$-$RNN(p_n)$ and $q \notin k$-$NN(p_n)$. Apart from them, the $lrd$ value of point $p$ itself need to be updated since the Euclidian distances to every of its $k$ neighbours have changed. For update of LOF values, it is the same as insertion and deletion operations. Algorithm 3 depicts the update operation in details.
Algorithm 3 Incremental LOF Update (Dataset S, Point $p_n$)

if $\Delta p_n < \varepsilon$ then
    $S_{update,k\text{-distance}} = \text{Compute } r$-$k^{th}(p_n) \cup p_n$;
    Update($S$, $p_n$);
    $S_{update,lrd} = \text{Compute } \{[k\text{-RNN}(S, p_n) - k\text{-NN}(S, p_n)]\} \cup p_n$;
    for all $p \in S_{update,k\text{-distances}}$ do
        Compute $k$-distance($S$, $p$);
        for all $q \in k$-NN($S$, $p$) do
            if $p \in k$-NN($S$, $q$) then
                reach-dist($q$, $p$) = $k$-distance($S$, $p$);
                $S_{update,lrd} \cup q$;
        $S_{update,LOF} = S_{update,lrd}$;
    for all $p \in S_{update,lrd}$ do
        Compute $k$-NN($S$, $p$);
        for all $q \in k$-NN($S$, $p$) do
            Get/Compute reach-dist($p$, $q$) using Eq. 3.1;
            Update lrd($p$) using Eq. 3.2;
            $S_{update,LOF} \cup k$-RNN($p$);
    for all $p \in S_{update,LOF}$ do
        Get lrd($p$);
        for all $q \in k$-NN($S$, $p$) do
            Get lrd($q$);
            Update LOF($p$) using Eq. 3.3;
else
    Deletion($S$, $p_n$);
    Insertion($S$, $p_n$);
3.3 Proposed Technique

3.3.1 Overview

We design our Cumulative Local Outlier Factor (C_LOF) algorithm based on the Incremental Local Outlier Factor algorithm as mentioned in the previous section, and it consists of two steps: maintenance of active data points and maintenance of virtual data points. Moreover, we employ a count-based overlapping sliding window to maintain a chunk of active data points in memory, and dynamically update their \textit{k-distances}, \textit{lrd} and LOF metrics as new data points arrive or old data points depart. Additionally, expired data points are not thrown away immediately; instead they are clustered incrementally based on an algorithm similar to DBSCAN. For each cluster, we maintain its \textit{virtual data point} by averaging all data points and the total number of data points that fall into this cluster. Those virtual data points are combined with active data points in sliding window to execute the incremental LOF algorithm. Like active data points, virtual data points also maintain \textit{k-distances} and \textit{lrd} values but they are calculated differently. These values are calculated statistically based on all historical data in this cluster, and running the Incremental LOF will not affect \textit{k-distances} and \textit{lrd} values of virtual data points. Those information are needed in order to calculate \textit{k-distances}, \textit{lrd} and LOF values of active data points. Unlike active data points, virtual data points do not need to maintain LOF values because we do not need to make decisions of their outlierness on their own. The virtual data points contain historical information of all expired data in a efficient manner and therefore they can help with the predictions of future data points. The virtual points are also weighted based on the so-called \textit{forgetting factor} $\lambda$. The virtual points that share the same temporal context with the current windowed data should receive more weights than those which share less. This is in place in order to address the temporal notation in data streams. The following sections will describe the C_LOF algorithm in details, starting from the maintenance of active data points:
3.3.2 Maintenance of active data points

As mentioned in Section 2.3.2.5, there are two types of sliding windows: count-based window and time-based window. Furthermore, based on the relationship between window size $|W|$ and length of each slide $|Slide|$, sliding windows can be further categorized into overlapping and non-overlapping windows. Both of them can be used to maintain active data points. In our case, since LOF is a density-based outlier detection algorithm, the number of data points is of vital importance in determining the accuracy of its results. Therefore, we employ count-based window in our C_LOF algorithm since time-based window can not guarantee that the window will have enough data points to run the LOF algorithm efficiently. Furthermore, in the Incremental LOF algorithm, as dataset are updated upon each new data point’s arrival or old data point’s departure, each calculation would be overlapping with previous data and the algorithm is triggered upon each new single data point’s arrival. Therefore, we apply count-based overlapping sliding window with $|Slide| = 1$ in our C_LOF algorithm to maintain active points.

The maintenance of active points starts when a new data point is fed into the C_LOF algorithm. Every new data point coming after the first window will run the algorithm incrementally as shown in Section 3.2.2. More specifically, when a new data point arrives, it needs to first kick out the oldest data point in the window by performing the deletion operation (3.2.2.2) and then insert the new one by performing the insertion operation (3.2.2.1) in incremental LOF. Therefore, we need to maintain a queue in the sliding window so that data points can arrive and depart in a FIFO manner.

Algorithm 4 describes how our active data points in sliding window are maintained upon arrival of each new data point:
Algorithm 4 Sliding Window Maintenance (Queue window, Point $p^{new}$)

\[
\text{if } |\text{window}| < W - 2 \text{ then} \\
\quad \text{window.enqueue}(p^{new}); \\
\text{else} \\
\quad \text{if } |\text{window}| == W - 2 \text{ then} \\
\quad\quad \text{window.enqueue}(p^{new}); \\
\quad\quad \text{Non Incremental LOF}(\text{window.active.points}); \\
\quad\text{else} \\
\quad\quad \text{if } |\text{window}| == W - 1 \text{ then} \\
\quad\quad\quad \text{window.enqueue}(p^{new}); \\
\quad\quad\quad \text{Incremental LOF Insertion}(\text{window.active.points}, p^{new}); \\
\quad\text{else} \\
\quad\quad\quad \text{else} \\
\quad\quad\quad\quad p^{old} = \text{window.dequeue}(); \\
\quad\quad\quad\quad \text{Incremental LOF deletion}(\text{window.active.points}, p^{old}); \\
\quad\quad\quad\quad \text{window.enqueue}(p^{new}); \\
\quad\quad\quad\quad \text{Incremental LOF Insertion}(\text{window.active.points}, p^{new}); \\
\quad\text{else} \\
\quad\quad \text{p^{old} = window.dequeue();} \\
\quad\quad \text{Incremental LOF deletion}(\text{window.active.points}, p^{old}); \\
\quad\quad \text{window.enqueue}(p^{new}); \\
\quad\quad \text{Incremental LOF Insertion}(\text{window.active.points}, p^{new}); \\
\text{else} \\
\quad \text{Computing threshold } \theta; \\
\]

3.3.2.1 Cold Start

One thing to note is the cold start problem in the sliding window. When the algorithm starts, since there are not enough data in the very first window, the algorithm will not be triggered until the number of data points has reached the predefined value of $W$. Therefore, it will result in the initial window of data being run in a non-incremental fashion.

3.3.2.2 Dynamic threshold

In addition, we also maintain a parameter $\theta$ that defines the threshold above which a LOF value of a given point should be considered as outlier. Due to the masquerading problem mentioned earlier, rather than fixing this parameter, we adjust it dynamically based on the average LOF values of all data points we have observed and their variances. The threshold is calculated using following equation:
\[ \theta = \mu_{lof} + 3 \cdot \sigma_{lof} \]  

(3.4)

where \( \mu_{lof} \) is the average of LOF scores, and \( \sigma_{lof} \) is the standard variance of LOF scores above all points we have received so far; they are updated as new data points arrive.

### 3.3.3 Maintenance of virtual data points

Instead of throwing every historical data away right after they expire, and therefore have no impact on the algorithm, we cluster them incrementally and store only the virtual data points and the total number of data points for each cluster, so that they can fit into the memory. The virtual data points contain the position information and the proximity information (e.g. \( k \)-distances, \( lrd \) and \( LOF \) values) about all data points that fall into this cluster previously and they can affect the proximity information of active data points. More importantly, at each iteration, the calculation of virtual data points depend only on their previous states and the total number of points in the cluster. It does not depend on the whole data points in that cluster; therefore, they can be computed incrementally.

The maintenance of virtual data points starts whenever a data point expires from the active window and that point is NOT an outlier. It first needs to execute the clustering algorithm that runs incrementally to mark all active points that belongs to the same cluster as this one, and then merge its position and proximity information into the virtual data point of the cluster it belongs to. The cluster that this expired data point belongs to should (ideally) be marked by previous points that expired. Once we decide to which cluster each expired data point belongs, we first update the position of that cluster incrementally by calculating its average value, based on its previous position, the number of data points in that cluster and the current position of this expired point (as shown in Eq. 3.5).

\[
P_{\text{virtual}}^\text{new} = (p_{\text{virtual}}^{\text{old}} \cdot \text{count}^{\text{old}} + p_{\text{expired}})/(\text{count}^{\text{old}} + 1) \]  

(3.5)
If the expired data point has not been marked by any previous expired points, it will form a new cluster on its own, and the point will just stay as it is, except that its status has changed from active to virtual. That virtual point will become the first point of that cluster along with its proximity information copied directly from its previous values as active point.

The same goes for updating the proximity values for virtual data point. They are calculated based on the average $k$-distance and $lrd$ values above all points in that cluster rather than by calculating their neighbourhoods. Therefore, for each cluster (virtual point), we only need to keep the average positions, average $k$-distance values, average $lrd$ values and the count of data points in that cluster in memory. Once the information of a virtual data point is updated, we need to calculate the $k$-distances, $lrd$ and LOF values of active data points that are affected by such changes and update them accordingly. This can be achieved by performing the update operation in incremental LOF algorithm as described in Section 3.2.2.3. Before discarding this expired data point, we also need to work out the weighted $lrd$ values for each virtual point by applying forgetting factor (explained shortly) and these are the actual $lrd$ values of virtual data points that are used for computation. After all these steps, we can safely discard this point as its information has already been incorporated in that cluster.

The overall procedure for cluster maintenance step can be described by Algorithm 5 when a data point expires. Details of each steps will be explained in following:

**3.3.3.1 Incrementally Clustering**

For the clustering, we use an algorithm based on the reachability of inlier data points very similar to the widely used DBSCAN algorithm [16]. The way it works is as follows.

Consider a data point $p$ that is about to expire from current window. First we need to determine whether it is an outlier based on $p$’s LOF value. If it is an outlier, it is reported
Algorithm 5 Cluster Maintenance (Map clusters, Point $p_{\text{expired}}$)

```plaintext
if $p_{\text{expired}}$.is_outlier then
    Report $p_{\text{expired}}$ and discard
else
    if $p_{\text{expired}}$.is_marked then
        for all $n \in k$-NN($p_{\text{expired}}$) do
            if !n.is_marked then
                n.is_marked = True;
                n.cluster_num = $p_{\text{expired}}$.cluster_num;
        cluster\_update = clusters[$p_{\text{expired}}$.cluster_num];
        virtual\_point\_old = cluster\_update.virtual\_point;
        Update cluster\_update.virtual\_point using Eq. 3.5;
        ++cluster\_update.count;
        Incremental LOF Update (window.active\_points, virtual\_point\_old, cluster\_update.virtual\_point);
    else
        cluster\_new = new Cluster (virtual\_point = $p_{\text{expired}}$, count = 1);
        hash\_number = some_hash\_function (cluster\_new.id);
        clusters[hash\_number] = cluster\_new;
        points\_to\_update = Incremental Clustering (k-NN($p_{\text{expired}}$), {});
        for all $p \in$ points\_to\_update do
            if !p.is_marked then
                p.is_marked = True;
                p.cluster_num = hash\_number;
        Incremental LOF Insertion (window.active\_points, cluster\_new.virtual\_point);
    end
    for all cluster $\in$ clusters do
        Update cluster.projected\_lrd using Eq. 3.6;
        Update all LOF values affected by running kRNN(cluster.virtual\_point);
    end
else
    cluster\_new = new Cluster (virtual\_point = $p_{\text{expired}}$, count = 1);
    hash\_number = some_hash\_function (cluster\_new.id);
    clusters[hash\_number] = cluster\_new;
    points\_to\_update = Incremental Clustering (k-NN($p_{\text{expired}}$), {});
    for all $p \in$ points\_to\_update do
        if !p.is_marked then
            p.is_marked = True;
            p.cluster_num = hash\_number;
    Incremental LOF Insertion (window.active\_points, cluster\_new.virtual\_point);
    for all cluster $\in$ clusters do
        Update cluster.projected\_lrd using Eq. 3.6;
        Update all LOF values affected by running kRNN(cluster.virtual\_point);
    end
```

immediately and would not participate in the algorithm anymore. Otherwise, if $p$ an inlier, we need to first check whether it has been marked by other points that expired earlier. If it has been marked by other points, it will be assigned to the same cluster that marks this point. A range query $k$-NN($p$) will then be executed on point $p$ and, for all data point $q \in k$-NN($p$), if $q$ has not been marked by any point before, it will be marked and will be clustered into the same cluster that $p$ belongs to when it expires. Otherwise, point $q$ is left as it is. On the other hand, if $p$ has not been marked by any previous point and yet it is an inlier, it will form a new cluster starting from $p$ itself. In addition, range queries will be executed recursively on all points that $p$ can reach in its $k$ neighbourhood until no point can be reached. For each of these points $q$, it will be marked and will be clustered to the same cluster that $p$ initiated when it expires. Algorithm 6 depicts in details how incrementally clustering is achieved:

**Algorithm 6 Incremental Clustering** (Dataset $points\_to\_add$, Dataset $points\_to\_update$)

```
if for all $p \in points\_to\_add$ and $p \in points\_to\_update$ then
    Return $points\_to\_update$;
else
    $points\_to\_update = points\_to\_update \cup points\_to\_add$;
    for all $p \in points\_to\_add$ and $p \notin points\_to\_update$ do
        for all $q \in k$-NN($p$) do
            $points\_to\_add \cup q$;
        Incremental Clustering($points\_to\_add$, $points\_to\_update$)
```

### 3.3.3.2 Forgetting factor

As mentioned previously, for those virtual data points that are contained in their clusters, their $lrd$ values are calculated statistically based on historical data points in that cluster rather than based on proximity with respect to other active data points. By definition (Eq. 3.2), the $lrd$ value should reflect how dense is the region where the point is. The larger this value is, the denser that region is. For virtual data points, this value is calculated by
taking the average of all \( lrd \) values of historical data points that are contained in the same cluster.

In addition to reflect the historical density of the cluster region the virtual data point is contained in, we want the \( lrd \) value of a virtual data point to reflect the \textit{freshness} of that cluster in order to address the \textit{temporal} property of data streams. That means that, if the cluster has not been updated for a while, it should have less weight on active data points and gradually fade away from our algorithm. If that cluster has been updated recently, it should have more weight. We achieve this by introducing a \textit{forgetting factor} \( \lambda \) that are applied on \( lrd \) values of all virtual data points. The result is the weighted \( lrd \) values that are actually used for computation. Suppose we have \( n \) virtual data points that are sorted based on their last updated timestamps and their \( lrd \) values are \((lrd_1, lrd_2, ..., lrd_n)\) respectively. We will multiply them by their relative weights defined as \((\lambda^{n-1}, \lambda^{n-2}, ..., 1)\) and \( \lambda \) can be selected using the bootstrapping method used in [8]. Therefore, the projected \( lrd \) value of a virtual data point is defined as:

\[
lrd_{\text{projected}}^v = \lambda^{n-i} \cdot lrd^v
\]

where \( i \) is the rank of the virtual data point with respect to its last updated timestamp. Since the index \( i \) will be updated for all virtual points whenever a single cluster updates, it will cause all virtual points to update their projected \( lrd \) values and therefore result in updates to LOF values for all active points that are affected by such changes.

### 3.3.4 Summary

When we get the active data points from sliding window and virtual data points from statistical clusters, we need to combine them into a dataset and run the Incremental LOF algorithm to detect outliers. More specifically, we store the \textit{k-distances}, \( lrd \) and LOF values for active data points and \textit{k-distances}, \( lrd \) values for virtual data points and update...
these information incrementally (except the very first run to cold start). The active data points can only be inserted or deleted from the dataset when new data point arrives or old data point expires from sliding window respectively. This can cause the proximity ($k$-distances, $lrd$ and LOF) of other active data points to update as per in Sections 3.2.2.1 and 3.2.2.2. On the other hand, the virtual data points can only be inserted to dataset when an expired data point forms a new cluster or its position is updated when a newly expired data point falls into that cluster. The insertion of a new virtual data point will cause the proximity of other affected active data points to update as insertion of active data points. To simplify the case when an inlier data point expires and starts to form a new cluster, rather than performing delete operation in Incremental LOF and treat the expired data point as a virtual data point and insert it back to form a cluster, we just change its status from active to virtual and create the new cluster information. Therefore, no proximity information need to be updated. The position update of an existing virtual data point will cause affected active data points to update their proximity base on Algorithm 3. More importantly, any update to the virtual data point will be followed by the update of the proximity of this virtual data point (based on Eq. 3.5) and $lrd$ values of all virtual data points (based on Eq. 3.6), causing further active data points to update their proximities. In short, the proximity information of active data points are functionally determined by the position information of both active and virtual data points, and the proximity information of virtual data points that are calculated differently.
3.4 Message-passing Implementation

3.4.1 General Framework

We implemented the C_LOF algorithm using an asynchronous programming model called the *actor model*\(^1\), which is an efficient way to achieve reactive programming. The way the actor model works is by decomposing the application into different components called actors. Each actor is a light-weighted process that maintains its states internally and constantly checks for incoming messages. The only way for an actor to interact with other actors is by sending messages, through which blocking can be avoided as it happens in synchronous programming. The actor model is an distributed and concurrent computation model that is locks-free and therefore results in higher usage of CPU resources.

For the C_LOF algorithm, we design five actors that need to cooperate with each other to achieve the required computation tasks. These actors are: 1) Coordinator Actor; 2) Window Actor; 3) Cluster Actor; 4) LOF Processor Actor; and 5) K-NN Actor. These actors and their relationships are summarized in Figure 3.3

\(^1\)https://en.wikipedia.org/wiki/Actor_model
The system is triggered when a new data point $p_{new}$ comes in. If the number of data points $n$ in the system is smaller than $W - 1$, where $W$ is the number of data points in the window, the relative information of point $p_{new}$ will be inserted to actors along the path from Coordinator Actor to K-NN Actor without performing any proximity calculation. When the number $n$ reaches $W - 1$, it will trigger the LOF Processor actor to invoke the static version of LOF algorithm, performing cold start and populating corresponding $k$-distances, $lrds$ values for each active data points and send the LOF values calculated to
the Window actor, resulting the calculation of threshold as per Eq. 3.4. Any subsequent
insertion will cause the LOF Processor actor to run the incremental LOF algorithm. When
the number $n$ is equal to $W$, the system will need to first “kick out” an expired point
$p_{\text{expired}}$, based on the queue structure maintained by Coordinator actor and invoke the
deletion operation on subsequent actors. Based on the status of $p_{\text{expired}}$, this can either
trigger the maintenance of cluster steps, which further triggers the update of virtual data
point operation, or the point $p_{\text{expired}}$ is reserved and becomes the starting representative
point for a new cluster (new virtual point). If $p_{\text{expired}}$ is an outlier, it will be reported to
the Coordinator actor. When the deletion operation with all its necessary maintenance
steps is finished, the system performs the incremental insertion as in the case $n = W - 1$
described previously. The detailed procedure of this message-passing implementation of
CLOF algorithm is summarized by the sequence diagram in Figure 3.4
In the following, we will give a detailed explanation and the implementation on each of these 5 actors mentioned above, including their data structured used under the hood.

### 3.4.2 Coordinator Actor

The Coordinator Actor only maintains a queue structure that “remember” the order of insertion of active data points. When a new data point comes, it generates a unique ID (uuid) that is used to identify each data point in our actor system and sends instructions for the following actors to perform actions such as Insertion or Deletion. When point expires, it receives a decision (whether it is an outlier or not) from internal actors and
reports it to the client. The Coordinator actor serves as an interface to the client of our system and it is the only actor that interacts with other components that sit outside our application. Other than that, the Coordinator actor controls the pacing of the entire algorithm. Specifically, this means that no insertion operation is allowed before the expired data point is fully removed from the system (including all necessary maintenance steps) to avoid the concurrent modifications over data.

3.4.3 Window Actor

The Window Actor serves as the sliding window component in the C_LOF algorithm and it keeps an hash table of all LOF values of active data points in the sliding window. The keys of the hash table are the unique IDs of data points generated by coordinator actor and the values are their corresponding LOF values. Moreover, it also keeps the threshold that is used to decide the outlierness of each active data point when it expires. The threshold is adjusted dynamically as LOF values get updated based on Eq. 3.4. Additionally, it also keeps the cluster number of which, each active point may belong to. These information are processed and updated by the LOF Processor actor, as shown in the following. With these information, the Window actor can decide the workflow to perform when an data point expired based on Algorithm 6.

3.4.4 Cluster Actor

The Cluster Actor serves as the cluster component in the C_LOF algorithm that is used to calculate the weighted proximity information (k-distances and lrds) of a virtual data point stored in LOF Processor actor, and to update them if necessary. It stores a hash table that keeps the raw proximity information of virtual data points and the total number of data points (count) that fall within each cluster. The keys of the hash table are the unique IDs of clusters that are generated when new clusters come into beings. The count
of each cluster is a required information used to calculate the raw proximity information when a data point expired and merged to an existing cluster (Eq. 3.5). In addition, the Cluster actor also maintains a stack of virtual points’ ID that always put the last updated item on top. This is used to work out the weighted proximity information as mentioned earlier in Section 3.3.3.2 by multiplying forgetting factor calculated over the index of the current item in this stack.

### 3.4.5 LOF Processor Actor

The *LOF Processor Actor* is the main processor of the C-LOF algorithm. It stores the actual data points’ values from both sliding window (active points) and clusters (virtual points), as well as their *k-distances* and *lrds* using separate hash tables with both their IDs as keys. Notice that the *k-distances* and *lrds* values of virtual data points are the weighted ones that is calculated by Cluster actor. The actual (raw) proximity of virtual points are stored by Cluster actor itself. The LOF Processor actor performs four types of operations: **Insertion**, **Deletion**, **Update** and **Fading**. Each of them will result in an update of *k-distances* and *lrds* values of affected active data points. In addition, at the end of each operation, the Processor actor needs to send an message to Window actor to persist the changes of LOF values. It does so by constantly querying the *KNN actor* for *K-NN* and *K-RNN* information (explained shortly). For **Insertion**, **Deletion** and **Update** operations, the procedure is exactly the same as specified in Section 3.2.2. However, there is a slight difference in that, when finding the *k-distances*, *lrds* as well as *LOF* values to update, we need to filter out the virtual data points as any changes of data points themselves should not affect the proximity of virtual data points since their proximity information are updated separately (Eq. 3.5). For **Deletion** operation, we also need to run incremental clustering to find all active data points that belong to the same cluster as the current expired point if it is not an outlier. The **Fading** operation is performed at end of each update to existing clusters so as to apply the *forgetting factor* to
each virtual point (Eq. 3.6). Any update to the cluster will result in a change of proximity information for each virtual point (as its index changed); therefore, we need to update proximity information of all active data points that are affected by such changes.

### 3.4.6 K-NN Actor

The *K-NN Actor* is just an in-memory data store that holds all the data points (including active and virtual) in the application. It is designed to speed up the KNN and KRNN queries that are required by the application. An ideal data structure for this task is KD-Tree [7], where the overall complexity of each range query is $O(DN\log N)$ or better where $D$ is the dimension of data and $N$ is the total number of data points in the store. However, KD-Tree is not resilient to the *curse of dimension* problem and it only performs better than the naive brute-force approach when $D$ is small ($D < 20$) [7].

### 3.5 Complexity Analysis

#### 3.5.1 Overview

We will analyze the complexity of C_LOF algorithm in terms of both time and space. For the time complexity, we will first consider the case when we have observed each individual point $p^{\text{new}}$. Then we will analyze the case when we have observed $n$ data points. We also include the case for the very first $n$ observations when the algorithm initially bootstrapped (cold start). For the space complexity, as secondary memory is not required in our C_LOF, we only discuss the primary memory that is required to store all active data points and virtual data points.
3.5.2 Time Complexity

We will first discuss what happens when each individual point \( p^{new} \) is fed to the algorithm. The situation would depend on the number of data points, noted as \( N \) that are currently in the system. If \( N < W - 2 \), where \( W \) is the number of data points contained in each count-based window, the operation would only consist of time to insert new data point to data stores of each components (actors), noted as \( T_{\text{insert}} \). If we use KD-Tree as data store for KNN actor, the average time for \( T_{\text{insert}} \) would be \( O(\log N) \). Insertions to other components only take constant time as we store them use hash tables. Therefore, the time complexity of \( \text{C}_\text{LOF} \) in this case would be defined by \( T_{\text{insert}} \), where no LOF calculation actually happens.

\[
T_{\text{C}_\text{LOF}}^{N < W - 2} = T_{\text{insert}} = O(\log N)
\]

When \( N = W - 2 \), it will insert the point \( p^{new} \) as before and trigger the static LOF algorithm that runs in batch mode. Since the static LOF algorithm has time complexity \( O(N \cdot \log N) \), noted as \( T_{\text{LOF}} \) [27], the time complexity to process initial \( N \) data points to cold start the \( \text{C}_\text{LOF} \) algorithm is therefore given by:

\[
T_{\text{C}_\text{LOF}}^{N = W - 2} = T_{\text{insert}} + T_{\text{LOF}} \leq a \cdot \log N + b \cdot N \cdot \log N = O(N \cdot \log N)
\]

where \( a, b \) are constants.

When \( N = W - 1 \), the \( \text{C}_\text{LOF} \) algorithm will from now on process each future points incrementally. The time complexity of \( \text{C}_\text{LOF} \) algorithm in this case would be exactly the same as the insertion operation in incremental LOF algorithm, noted as \( T_{\text{LOF}_{\text{insert}}} \). According to [30], the insertion operation in incremental LOF algorithm has time complexity \( O(k \cdot F \cdot T_{kNN} + k \cdot F \cdot T_{kRNN} + F^2 \cdot k + T_{\text{insert}}) \), where \( k \) is the predefined number of nearest neighbours of point \( p^{new} \), \( F \) is the maximum number of reverse nearest neighbours.
of $p^{new}$, which is proportional to $k$ [30], and $T_{kNN}$ and $T_{kRNN}$ is the time complexities of $k$-NN and $k$-RNN range queries respectively. For KD Tree, time complexities of these range queries can be reduced to $O(\log N)$, as compared to the naive brute-force approach which would require $O(N^2)$ time complexity. If we denote $F = ck$, the time complexity of C_LOF algorithm when $N = W - 1$ would be bounded by:

$$T_{C_LOF}^{N=W-1} = ck^2(T_{kNN}+T_{kRNN})+c^2k^3+T_{insert} \leq 2ack^2\log N + b\log N + c^2k^3 = O(k^2\log N + k^3)$$

where $a, b, c$ are constants.

When $N = W$, the C_LOF algorithm will first need to remove the oldest data point $p^{old}$ from window by performing the delete operation in incremental LOF algorithm. Depending on the status of $p^{old}$, C_LOF will either or not trigger the cluster maintenance steps. Same as insertion operation when $N = W - 1$, the deletion operation when $N = W$ has the same time complexity. If no cluster maintenance steps are triggered (if $p^{old}$ is an outlier), C_LOF will just run the insertion operation after performing deletion operation. Therefore the time complexity in this case would be:

$$T_{C_LOF}^{(N=W)} = T_{LOF, insert} + T_{LOF, delete} = 2T_{LOF, insert} = O(k^2 \cdot \log N + k^3)$$

If the cluster maintenance steps are triggered ($p^{old}$ is an inlier), beside the insertion and deletion steps, the C_LOF first need to run incremental clustering. The time complexity of incremental clustering, noted as $T_{clustering}$ is dependant on the clustering algorithm. Based on [16], the DBSCAN algorithm has an average runtime complexity of $O(N \cdot \log N)$ and worst case runtime complexity of $O(N^2)$. Therefore, the $T_{clustering}$ should have the same bound.

$$T_{clustering} = O(N^2)$$
The update of virtual point and its proximity information takes only constant time and the update operation in incremental LOF algorithm, noted as $T_{LOF\text{-}update}$ takes as much as twice as the insertion operation $T_{LOF\text{-}insert}$.

$$T_{LOF\text{-}update} \leq 2T_{LOF\text{-}insert} = O(k^2 \cdot \log N + k^3)$$

At the end, it needs to apply the forgetting factor and calculate the projected $lrd$ values for all virtual points and update LOF values of affected active points. If we denote the number of virtual points currently in the system as $m$, and base on the definition of LOF, $m$ should be less than $\frac{N}{k}$ in worst case. For each of these virtual point, we need to run the $k$-$RNN$ range queries and find its reverse neighbours. If we denote the number of these reverse neighbours as $F = ck$ where $c$ is a constant, then for each of these reverse neighbours, their LOF values need to be updated base on Eq. 3.3, which has exactly the same time complexity as $k$-$NN$ range queries. Assuming $k$-$NN$ and $k$-$RNN$ both have the time bound of $O(\log N)$ and $m = \frac{N}{k}$, the total time complexity to apply the projected $lrd$ values on all virtual points, noted as $T_{projected\_lrd\_update}$ would be:

$$T_{projected\_lrd\_update} \leq m \cdot (\log N + ck \cdot \log N) = O(N \cdot \log N)$$

Therefore, the total time complexity of $C\_LOF$ in case $N = W$ where cluster maintenance steps are necessary has an upper bound of:

$$T_{C\_LOF}^{(N=W)\text{\_worst}} = T_{LOF\text{-}insert} + T_{LOF\text{-}delete} + T_{clustering} + T_{LOF\text{-}update} + T_{projected\_lrd\_update} = O(N^2 + k^2 \cdot \log N + k^3)$$

However, as the $T_{clustering}$ only has $O(N^2)$ in worst case, on average, it can be run in $O(N\log N)$ time. The average time complexity of $T_{C\_LOF}$ can be reduced to $O(N\log N + k^2 \cdot \log N + k^3)$ in most cases.
Next, we shall see the case when \( n \) data points are fed into the algorithm and each of them will be processed individually. We will first discuss the initial \( n \) data points that are first fed to our algorithm, where \( n = W - 1 \). In that case, the first \( n - 1 \) points will not trigger any update and they only need to be stored in the sliding window. The \( n^{th} \) data point will trigger the static LOF algorithm that runs in batch mode. Since the LOF algorithm has time complexity of \( O(n \cdot \log n) \) [27], the time complexity to process the initial \( n \) data points to cold start in C_LOF algorithm is therefore given by:

\[
T_{C_LOF}(n) = \sum_{i=1}^{n-1} T_{insert} + T_{LOF} = O(n \cdot \log n)
\]

where \( n = W - 1 \).

Later on, whenever a new point is inserted, it will trigger the C_LOF algorithm to run incrementally. As discussed previously, each insertion of individual data point has an upper bound of \( O(N^2 + k^2 \cdot \log N + k^3) \), where \( N \) here would be equal to the number of data point \( W \) in each window. Therefore after observing \( n \) data points, the time complexity of C_LOF would be:

\[
T_{C_LOF}(n) = \sum_{i=1}^{n} T_{C_LOF} \leq nW^2 + nk^2 \cdot \log W + nk^3 = O(nk^3)
\]

where \( W \) is ignored as it is a constant in a count-based window.

### 3.5.3 Space Complexity

As we see from the definition of C_LOF algorithm, the application only need to store data from sliding window, called active data points, and data that are mined from incremental clustering, called virtual data points, For each of these data points, we need to store not only its value, but also its proximity information including \( k\)-distances, \( lrds \). For active data points, we also need to store LOF values that is used to judge outlierness. The number
of data points in each window is fixed, as we use count-based sliding window. However, the cluster size may vary (number of virtual data points) after receiving $n$ data points depending on the distribution of these data. In the original definition of LOF algorithm, the parameter $k$ also means the minimum number of data points that can form a cluster [27]. Therefore, the maximum number of clusters that LOF can form after observing $n$ points is bounded by $n/k$. Assuming the count-based window has $W$ number of data points, and the number of received data points $n >> W$, the space complexity of C_LOF is therefore given by:

$$S_{C_{LOF}}(n) \leq W + (n - W)/k = O(n/k)$$

where $W$ is ignored as it is a constant in a count-based window.
Chapter 4

Cumulative Kernel Density Estimator with Retrospect

4.1 Overview

In this chapter, we will introduce our second algorithm, namely Cumulative Kernel Density Estimator with Retrospect (C.KDE.WR) that is a parameterless approach for outlier detection over high-dimensional data streams and it is extended from an existing algorithm called SOD_GPU in literature [22]. In SOD_GPU, densities of each data points are estimated over a density function, which is approximated from data points in a non-overlapping sliding window and statistical binned summary that has been mined from historical data. The density function is approximated based on the kernel density estimator method with some slight modification to work on binned summary. Outlier scores are calculated for each points in sliding window based on its density. In C.KDE.WR, we further improve SOD_GPU by introducing the novel concept drift detection module. When outliers are detected, rather than reporting them immediately, we store them and treat them as candidate outliers. They are only considered true outliers after observing enough data points to give confidence on our previous decisions. Additionally, a forgetting factor is introduced and
applied as weights to all bins in order to address the temporal notation of data streams. The use of kernel estimator together with retrospection on candidate outliers and forgetting factor on bins will help us deal with the concept drift of data streams. Therefore, the proposed $C_{KDE\_WR}$ algorithm is capable of detecting Type II outliers in data streams with temporal context. In the following, we will give detailed steps on this algorithm as well as its accelerated implementation on GPU.

4.2 Methodology

4.2.1 SOD\_GPU in General

The SOD\_GPU algorithm is based on the cumulative approximation of the probability density function $f(x)$ on current data points that are contained in a non-overlapping sliding window and statistical binned summary that is mined from historical data. Several methods exist for approximation of the probability density functions such as histogram, wavelet. In SOD\_GPU, they use kernel density estimator with Gaussian kernel to approximate the density function as it gives smooth estimations over the entire dataset [22]. For data points that have been expired from window, they are mined statistically rather than discarded immediately and therefore contribute nothing to the algorithm. They use a very popular technique in [17] to turn all historical data into binned summary in order to avoid maintaining all expired data points and to also reduce the number of evaluations required for each of them. To decide whether a data point is an outlier, they estimate the density of that point in the probability density function $f(x)$. If its density on $f(x)$ is larger than the threshold $\theta$, which has been estimated dynamically based on the average of all data points in current window, it will be considered an outlier. We shall discuss all necessary steps in SOD\_GPU as well as our novel concept drift detection module in details in following sub-sections.
4.2.2 Kernel Density Estimator

To address the varying data distribution and concept drift property of data streams, we do not assume the fixed distribution of the underlying probability density function. Rather, we estimate it dynamically as new data points have been observed. Therefore, the probability density function \( f(x) \) always reflects the most recent data distribution and it never becomes obsolete. To address the uncertain property of data streams, we use kernel density estimator to estimate the probability density function. Different from the histogram-based estimation, where the occurrence of each data point \( x_i \) will increase the density of probability density function at its corresponding bin index by 1, the kernel estimator will only increase the density at \( x_i \) by a probability of \( p(x_i) \) and the 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. Since the probability has built-in characteristics of uncertainty, the kernel estimator is capable of estimating the density function from uncertain 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. 4.1, where \( k(x) \) is called the kernel function.

\[
f(x) = \frac{1}{n} \sum_{i=1}^{n} k(x_i - x) \tag{4.1}
\]

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 very smooth estimation. The Gaussian kernel is given by:

\[
k(x, x') = \frac{1}{(2\pi)^{D/2}H} \exp \left\{ -\frac{1}{2} \left( \frac{x - x'}{H} \right)^2 \right\} \tag{4.2}
\]

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 \(+\infty\) [35]. However, the majority of density will still only be distributed to the neighbourhood of \( x' \). We use Scott’s rule [35] to calculate the bandwidth at each dimension based on the following formula:

\[
h_i = \sigma_i n^{1/D+4}
\]

where \( \sigma_i \) 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 \( f_{\text{cumulative}}(x) \) by adding the kernel estimator in the sliding window \( f_{\text{window}}(x) \) and the kernel estimator in bin summary \( f_{\text{bin}}(x) \) accordingly. Therefore,

\[
f_{\text{cumulative}}(x) = f_{\text{window}}(x) + f_{\text{bin}}(x)
\]

In particular, for bin 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 have much been reduced. To address the temporal notation of data streams, we also introduce forgetting factor on kernel estimator function of bin summary to weight each bin accordingly. The following sections will discuss these in details.

### 4.2.3 Sliding Window

Due to the unbounded nature of data streams, 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 2.3.2.5. We divide data streams into chunks of windows at regular
time intervals and apply the algorithm on each of them. Therefore, it is a time-based non-overlapping sliding window. Assume $|W|$ is the window size and $T_0$ is the starting time, the window boundaries will therefore be $T_0 + |W|$, $T_0 + 2|W|$, $T_0 + 3|W|$ and so on.

For the data points in a sliding window, they contribute directly to the density function $f(x)$ as defined in Eq. 4.1. Outliers are found over these windowed data. Therefore, for a data point $x$ that is in the current sliding window, its density over the sliding window is defined by Eq. 4.4 after applying Gaussian kernel, where $n$ is the number of data points in sliding window

$$f_{\text{window}}(x) = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{(2\pi)^{D/2}H} \exp\left\{ -\frac{1}{2} \left( \frac{x - x_i}{H} \right)^2 \right\} \quad (4.4)$$

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 previous historical data.

### 4.2.4 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 [17]. There are many implementations of binned summary in literature and we apply the one introduced in [17].

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.

### 4.2.4.1 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, $\Delta$. Therefore:

$$\Delta = \frac{[max(x_j) - min(x_j)]}{k}$$

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:

$$< I_{i1}, I_{i2}, I_{i3}, \ldots, I_{iD} >$$

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

$$B_{xi} = (I_{iD} - 1)k^{D-1} + (I_{i(D-1)} - 1)k^{D-2} + \ldots + (I_{i2} - 1)k + I_{i1} \quad (4.5)$$
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 $m << k^D$. Therefore, we are not worried about the curse of dimension problem here.

4.2.4.2 Maintain bin statistics

![Diagram of bin summary](image)

For each bin, we maintain the number of data points (noted as bin count $C_i$) and their aggregate mean value vector (noted as $M_i = < \mu_{i1}, \mu_{i2}, \ldots, \mu_{iD} >$) for the points that fall into this bin. The $\mu_{ij}$ here is the average mean value of all data points in $B_i$ at dimension $j$. Additionally, we also maintain the mean value vector $\mu$ and the standard deviation $\Sigma$.

Figure 4.1: Maintenance of bin summary [22]
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 $n^{th}$ 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 $\mu_i^n$ 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,

$$M_i^n = \frac{c_i^n \cdot \mu_i^n + C_i^{n-1} \cdot M_i^{n-1}}{c_i^n + C_i^{n-1}}$$

$$C_i^n = c_i^n + C_i^{n-1}$$

### 4.2.4.3 Density estimation

For the bin summary, its density function is calculated slightly differently than those in sliding widow defined by Eq. 4.4. 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. 4.1 for bin summary. Therefore, for a data point $x$ in the current window, its density over the bin summary is defined by Eq. 4.6 after applying Gaussian kernel, where $m$ is the number of bins in the bin summary:

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

(4.6)
The density of a data point is calculated cumulatively by applying Eq. 4.4 and Eq. 4.6 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 data streams as we want the model to always be consistent with the most up-to-date trend of data distribution.

4.2.5 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. 4.7

$$f_o = \frac{1}{f(x)} \quad (4.7)$$

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 $\xi$, given by:

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

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

4.2.6.1 Candidate outliers and Retrospect

When an outlier is detected, rather than reporting and confirming decisions immediately, they will be treated as candidate outlier and re-evaluated against future windows in order to deal with the concept drift of data streams. When a concept drift occurs, our underlying kernel estimator model needs time to adjust itself to reflect the latest change of data distribution; coining our decisions immediately may generate a large number of false positive when such changes start to emerge. Therefore, we introduce the retrospect step to re-consider our decision made on candidate outliers detected earlier.

When a candidate outlier is detected, we store it in a separate list and re-evaluate its outlier factors for a number of future windows. Each candidate outlier is associated with a rank \( r \) that denotes the number of times that they have been continuously considered as candidate outliers. Every time they are re-evaluated, we either increment their ranks if they are still candidate outliers in this window, or decrement their ranks if they are no longer candidate outliers in this window. When their ranks \( r \) reach a pre-defined limit \( R \), they will be considered as true outliers and reported. Or if their ranks keep dropping and reach zero, they will be removed from the list and therefore not considered as outliers anymore.

4.2.6.2 Forgetting factor

When calculating the density of a data point over a bin summary, what we really want is to also consider the freshness of that bin so that bins that are relatively older should gradually fade away and have less impact on deciding the densities of current data than more recent ones. That is why we introduce the forgetting factor over binned summary when calculating the kernel estimator. The introduction of the forgetting factor will help us address the temporal property of data streams. 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 old data points and the weight is decreasing exponentially from present to past [8]. We apply exponential forgetting in bin summary by storing the timestamp for each bin when it was last updated, and sort the bins according this timestamp. The relative weight between two consecutive bins are a constant called forgetting factor $\lambda$, where $0 < \lambda < 1$. The most recent bin will receive a weight 1 and the older ones will receive a weight $\lambda$ on top of the previous weight. Therefore for any particular bin, it will receive a weight according to its relative position to the most recent bin in this sorted list. The forgetting factor $\lambda$ is selected using the bootstrapping method in [8].

Let $(B_1, B_2, B_3, ..., B_m)$ be bins that have been sorted according to their last updated timestamp, $(M_1, M_2, M_3, ..., M_m)$ be their corresponding mean value vectors, and $(C_1, C_2, C_3, ..., C_m)$ be their bin counts. Their corresponding weights would therefore be $(\lambda^{n-1}, \lambda^{n-2}, \lambda^{n-3}, ..., 1)$. If we apply these weights to the kernel density estimator function defined over bin summary in Eq. 4.6, the probability density function become Eq. 4.8 after applying the exponential forgetting factor

$$f_{\text{bin(weighted)}}(x) = \frac{1}{\sum_{i=1}^{m} \lambda^{m-i}C_i} \sum_{i=1}^{m} \lambda^{m-i}C_i \exp\left\{ -\frac{1}{2} \left( \frac{x - M_i}{H} \right)^2 \right\} \quad (4.8)$$

### 4.3 Parallel Implementation

#### 4.3.1 General Framework

As we can see from Eq. 4.4 and Eq. 4.6, the kernel density estimator is very computational demanding since it evolves density estimation over the entire data points in the sliding
window and bins, plus additional bins maintenance and parameters adaption steps. The processing of each window with \( N \) data points will require the time complexity of \( O(N^2) \) since each data point needs to be compared with all other points in the window and bins in order to estimate its density. Moreover, the processing time also increase as the number of dimensions \( D \) grows. When the data is coming at a very high rate, this algorithm will not be able to give results in a timely manner and will therefore hit the bottleneck as it can not keep up with the speed of incoming data streams.

To avoid this bottleneck, we implemented the C_KDE_WR algorithm in a parallel fashion, taking advantages of GPU acceleration using NVIDIA CUDA parallel programming platform\(^1\) to speedup this algorithm. Specifically, since density estimation of each data point can be calculated independently and does not rely on other points, the density of each data point can be calculated independently and concurrently by allocating different threads in GPU. Other computational tasks such as bin index calculations, bin summary maintenance etc are run on CPU and do not interfere with the density calculation on GPU. This give us another level of parallelism since we do not have to wait for the GPU to complete density estimation before calculating and updating the bin statistics. In the following section, we will discuss the parallel implementation of C_KDE_WR algorithm in details:

### 4.3.2 Density estimation on GPU

Programming in GPU follows SIMD (Single Instruction Multiple Data) architecture. A CUDA program consists of *host functions* that runs on CPU and *kernel functions* that run on GPU. Kernel functions are executed by a given dimension of *thread blocks* and multiple thread blocks (*grids*) can be given in order to cover the entire input elements. All threads in the thread blocks have a unique ID to identify themselves and they are executed in groups.

\(^1\)Compute Unified Device Architecture: https://developer.nvidia.com/cuda-zone
of 32 threads, called *warp*. They run in parallel and usually only access one element (or section) of the input data or output data (by referring to its unique ID). Kernel functions are mostly run sequentially, and each invocation requires memory transfer back and forth from CPU to GPU. However, with the introduction of *CUDA streams*, kernel functions can also be run concurrently on different CUDA streams, meaning that the overhead of data transfer can be avoided by overlapping kernel executions on different CUDA streams. For example, when executing the *Kernel 1* on *Stream 1*, *Kernel 2* can start transferring data from host to device on *Stream 2* and, when *Kernel 1* finishes, *Kernel 2* will be ready to go with all necessary data loaded in GPU memory.

To remove ambiguities, we define *query points* as data points that we want to calculate outlier factors of, and *reference points* as data points used to approximate the probability density function $f(x)$. In our case, query points are those from the current sliding window as well as candidate outliers from previous batches. Reference points are all points from the current sliding window as well as all bins from bin summary. Note that, for each query point from the sliding window, its reference points are those points from this window except the data point itself plus all bins. Reference points for candidate outliers are simply points from window and bins. For density estimation in our proposed C_KDE_WR algorithm, we employed *four* kernel functions to calculate the final outlier factor of each data point. Whenever a old window expires and new window arrives, GPU runs these four kernel functions as defined in the following:

- Kernel 1: Computes the standard deviation and estimates the bandwidth $H$ according to Scott’s rule [35]
- Kernel 2: Computes the probability density for each query point over the sliding window using Eq. 4.4
- Kernel 3: Computes the probability density for each query point over binned summary with forgetting factor using Eq. 4.8
• Kernel 4: Computes the cumulative probability density and calculates the outlier factor for each query points using Eq. 4.7; Marks candidate outliers and updates ranks of each candidate outliers; Reports true outliers if necessary

Since Kernel 2 and Kernel 3 are independent of each other, they can be assigned to different CUDA streams to be executed concurrently. Both Kernel 2 and Kernel 3 depend on Kernel 1 and Kernel 4 depends on both Kernel 2 and Kernel 3. In CUDA, the default stream (also known as stream 0) is always synchronized with other streams and we therefore assign Kernel 1 and Kernel 4 to default streams. We apply barriers in Kernel 2 and Kernel 3 in order to synchronize Kernel 1 from the default stream.

For GPU programming, as the data need to be transferred from CPU memory (referred as host memory) to GPU memory (referred as device memory), the bandwidth overhead required for such transfer needs to be minimized. In CUDA, if a warp (with size of 32 threads) access a successive device memory of size 32, 64 or 128 bytes, the memory access is coalesced into a single fetch. We therefore transfer the input data to the column-major format ($D \times N$ data matrix). To fully utilize the power of GPU cores, we also need to pad the number of data points $N$ in a window, and block size of the thread block $T$ as multiple of warp size 32 to achieve further coalescing. Notice that the total number of threads within a thread block can not exceed 1024. The size of each data point with dimension $D$ also need to be aligned to multiple of 1, 2, 4, 8 or 16 bytes. Furthermore, to save some data transfer between CPU and GPU, we also pre-load some data into the shared memory, which is a memory space shared between all threads within the same thread block as accessing to shared memory is much cheaper than accessing of global device memory.

In the following section, we shall discuss CUDA implementation for each of these kernel functions in details:
4.3.2.1 Kernel 1

Kernel 1 is responsible for the computing of standard deviation \( \sigma_i \) and derive the bandwidth \( H \) for kernel estimations according to the Scott’s rule [35]. The computation is performed by one-dimensional thread block of size \( 1 \times T_{K1} \), where \( T_{K1} \) needs to be a multiple of 32 to achieve coalesced memory access, and one-dimensional grid of size \( 1 \times B_{K1} \), where \( B_{K1} \) is set to \( N/T_{K1} \). Each thread block computes \( T_{K1} \) data points in parallel and derive their standard deviation \( \sigma_i \) over \( D \) dimensions. At first, threads within each thread block compute the partial sum for each dimension \( D \) of data points size \( T_{K1} \) in shared memory and then update the total sum of all data points in global memory. Once the total sum is obtained, we can easily compute the mean value vector \( \theta \) and then apply the same procedure to compute the total sum of \( (X - \theta)^2 \) in order to derive the standard deviation for each dimension. We then apply the Eq. 4.3 to derive the bandwidth \( H \) used for kernel estimations.

4.3.2.2 Kernel 2, Kernel 3

Kernel 2 and 3 calculate the probability density of all query data points (windowed data and candidate outliers from previous window) over sliding window and statistically binned summary respectively. We apply the same grid and thread block layouts for these kernels as they implement the same function. The only difference is the dimensions for these layouts. For Kernel 2, the input matrices are query points matrix with dimension \( D \times (N+C) \), where \( C \) is the size of candidate outliers from previous windows, and reference points matrix with dimension \( D \times N \). For Kernel 3, the query points matrix has the same dimension as Kernel 2. However, the reference point matrix only has dimension of \( D \times M \), where \( M \) is the size of binned summary (total number of bins). The densities for both Kernel 2 and Kernel 3 will be computed using Eq.4.4 and Eq. 4.8 respectively. Note that for Kernel 3, we also need to sort the bins by their last updated timestamps and apply forgetting factor \( \lambda \) based
on their position indices in the sorted list. The output of both kernel functions are density matrices with dimension \((N + C) \times 1\), where elements are density for each query point respectively.

For each query point in these two kernels, we need to sum up densities over all reference points. Therefore, we need \((N + C) \times N\) computations in total to get the final output for Kernel 2 and \((N + C) \times M\) computations in total to get the final output for Kernel 3. Hence, we divide computations in these two kernels into \(B_x \times B_y\) dimension grid, with each consisting of a \(T \times T\) dimension thread block, where \(T = 32\) to achieve coalescing memory access, \(B_x\) for both kernels are \((N + C)/T\) and \(B_y\) for Kernel 2 is \(N/T\), while for Kernel 3 it is \(M/T\). To remove the bandwidth of memory transfer between host and device, each of these \(T \times T\) thread block copy the sub-matrices of query and reference points’ matrices into its shared memory and compute partial sums of all possible calculations over these sub-matrices. The result of these partial sums are merged and aggregated into the final density matrix stored in device global memory. Figure 4.2 illustrates the GPU layout of Kernel 2 and Kernel 3.

![Figure 4.2: GPU Grid and thread block layout for Kernel 2 and Kernel 3](22)
4.3.2.3 Kernel 4

Once we acquired the density matrices from Kernel 2 and Kernel 3, we can aggregate them into one cumulative probability density matrix for all query points. We can then apply Eq. 4.7 to derive the outlier factors. We mark the candidate outliers using the pre-defined cutting-off threshold $\theta_{\text{threshold}}$ and update their ranks. If any of their ranks reached predefined value of $R$, we report them as true outliers. For this kernel function, we divide the computation into $1 \times B_{K4}$ dimension grid and $1 \times T_{K4}$ dimension thread block. As before, the number of $T_{K4}$ needs to be multiple of 32 in order to achieve coalesced memory access. $B_{K4}$ is then set to $N/T_{K4}$ to cover all query points. Each thread block computes outlier factor over $T_{K4}$ query points in parallel and update their ranks if they are candidate outliers. We return these candidate outliers to host memory and filter out the true outliers base on their ranks.

4.3.3 Binned summary maintenance on CPU

Unlike GPU, programming on a multi-core CPU follows MIMD (Multiple Instructions Multiple Data) architecture. For update on bin 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 batch of data points from sliding window, we need to mine their statistics and update these information into our existing binned summary. Hence, we design the maintenance step on binned summary in two phases: In-window mining and Out-of-window mining.

4.3.3.1 In-window mining

During the In-window mining, we calculate statistics sufficient to data points within the current window $n$. As mentioned in Section 4.2.4.2, we need to calculate the mean value
vector $\mu^n_i$ and bin count $c^n_i$ for each of its corresponding bin $B_i$ in order to calculate its cumulative mean value vector $M^n_i$ and bin count $C^n_i$ at bin index $i$ in this window. Before calculating $\mu^n_i$ and $c^n_i$, we first need to determine the bin index $i$ for all data points in the current window using Eq. 4.5 and group them by using the keyedBy operation. We then apply the reducers for each of these indexed groups to derive the $\mu^n_i$ and $c^n_i$ 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 of this bin.

4.3.3.2 Out-of-window mining

Once we get the $\mu^n_i$, $c^n_i$ at current window, we can easily derive the next bin statistic for $B_i$, including mean value vector $M^n_i$, bin count $C^n_i$. 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.

4.4 Complexity Analysis

4.4.1 Overview

We will analysis the complexity of the proposed C_KDE_WR algorithm in terms of both time and space. For time complexity, as data are processed in batches of non-overlapping windows, we will only analyze the complexity of the algorithm whenever it processes each of these single batch. For space complexity, we will consider the memory required after processing of total number of data points of size $n$ where $n$ is ways larger than the number of data points in a single window.
4.4.2 Time Complexity

As the C_KDE_WR algorithm mainly contains two parts, namely density estimation and bins maintenance parts, we will analyze their time complexities separately.

For density estimation, as mentioned earlier, the density is estimated for each query point over reference points. The query points are the points from the current sliding window plus the candidate outliers from previous windows. For reference points, it is a cumulative one over points in sliding window and bins in binned summary. If we denote the number of data points in current window as \( N \), the dimension of data points as \( D \), the number of candidate outliers from previous window as \( C \), and the number of bins currently in system as \( M \), the time complexity to apply the Gaussian kernel function defined in Eq. 4.2 as \( T_{\text{kernel}} \), the time required for density estimation is given by:

\[
T_{\text{density, estimation}} = (N + C)(N + M)T_{\text{kernel}}
\]

since the density estimation is linear over all query points and, for each query point, it is linear over all its reference points. Notice that the time complexity for running Gaussian kernel function \( T_{\text{kernel}} \) is also linear over the dimension \( D \) of data points; we can therefore expand the \( T_{\text{kernel}} \) as:

\[
T_{\text{kernel}} = D \cdot T_{\text{kernel,d}}
\]

where \( T_{\text{kernel,d}} \) is the time complexity to apply the Gaussian kernel function defined in Eq. 4.2 at each single dimension. The final complexity of density estimation is therefore given by:

\[
T_{\text{density, estimation}} = (N + C)(N + M)D \cdot T_{\text{kernel,d}} = O(D \cdot N^2)
\]

where \( C, M \) are independent of \( N \) and \( D \) and they are treated as constants.

The bins maintenance consists of two steps: In-window mining and Out-of-window
mining. For in-window mining, we first need to calculate the bin index for each point in window, which will give a linear time complexity over data dimension $D$ for each single point and linear time complexity over total data points of size $N$. The overall complexity is therefore bounded by $O(D \cdot N)$. We then apply the `keyedBy` operation to group data points by their bin index, which is linear time over $N$ as each record would need to be traversed, and finally perform aggregations on each of these bins $(\mu_i^n, c_i^n)$, which is also linear time over $D \cdot N$ as each dimension of a single data record need to be scanned for all data points regardless of which bins they belong to.

Once we get the all statistics for in-window mining, the out-of-window mining take only constant time for each mined bin from current window data. In the worst case scenario, each data point from current window will be scattered into different bins and therefore after mining over all data points in the window of size $N$, we get $N$ mined bins to update! Therefore, this step will require linear time of data size $N$ in the worst case.

The total time complexity for bin maintenance is therefore bounded by:

$$T_{\text{bin\_maintenance}} = T_{\text{in\_window}} + T_{\text{out\_of\_window}} = aD \cdot N + bN = O(D \cdot N)$$

where $a, b$ are constants

### 4.4.3 Space Complexity

For space complexity, as each window of data are kept in primary memory temporarily and discarded after they have been absorbed into binned summary, we only consider the memory required to store all data in sliding window and candidate outliers from previous windows plus the additional memory required to keep bin statistics at any given point of time.

The number of data points $N$ in each window will vary based on the rates of incoming
streams in application if we apply \textit{time-based} sliding window. It will be a constant if we apply \textit{count-based} sliding window. To eliminate these ambiguities, we make the assumption that the average number of data points in each window is linear to $T$, where $T$ is defined by the time elapsed for each window. We can therefore denote the number of data points in each window as $c \cdot T$ where $c$ is a constant.

For candidate outliers, as they are minority groups, we by assumption, assume that the number of candidate outliers after observing $n$ data points is logarithmic to $n$. We therefore represent number of candidate outliers as $d \cdot \log n$ where $d$ is a constant.

The number of bins from previous data is bit of tricky to estimate as it depends on the distribution of data. Some data distributions might be more clustered while others might be more scattered. However, in reality, most datasets turn to be more clustered and if we assume the minimum number of data points in each of these clusters is constant $k$, the maximum total number of clusters (bins) after observing $n$ data points is given by $n/k$. For each of these bins, we need to maintain statistics such as mean value vector, bin count etc. Therefore we will need $2 \cdot n/k$ memory in total.

The total space complexity for $C\_KDE\_WR$ is therefore given by:

$$S_{C\_KDE\_WR} = c \cdot T + d \cdot \log n + 2 \cdot n/k = O(T + n)$$

where $c, d$ are constants.
Chapter 5

Experiments and Results

5.1 Overview

In this chapter, we present the experimental results of our proposed algorithms (C_LOF and C_KDE_WR) that have been applied over both synthetic and real-life datasets. Datasets are split and fed into algorithms window-by-window in order to simulate streaming environment in a real-world application. To measure the accuracy, we use different metrics including Precision, Recall and F-score etc. For C_LOF, we measure its accuracy on different parameter settings on synthetic datasets. For C_KDE_WR, we compare its accuracy with the method purposed in literature [22] in the same setup. We do not benchmark accuracies between C_LOF and C_KDE_WR because they each apply in different scenarios, one on low-dimensional data only, the other works better on high-dimensional data. We also measure performances of C_LOF and C_KDE_WR based their latencies on a multicore CPU and NVIDIA GPU respectively. In the following, we will present details of our experiments.
5.2 Datasets

5.2.1 Synthetic Dataset

For C_LOF algorithm, we only measure its accuracy on synthetic datasets using Gaussian mixture model with outlier points generated using uniform distribution given a range; this is because the original datasets (real-life video frames) used in literature [30] (which is what the C_LOF algorithm is based on) are inaccessible of the time of writing. We chose the Gaussian mixture model because it do not conform to a fixed distribution and its data pattern can change over time to simulate real data streams.

We conducted the experiments in two different settings. In one setting, we want to measure the ability to detect concept drift of our model and we therefore generated two 2-dimensional Gaussian distributions with different means but same variances (noted as Synthetic 1 dataset in experiments). In the second setting, we want to know if our model is resilient to the masquerading problem as mentioned in Section 2.2.3.6 and we therefore generate two 2-dimensional Gaussian distributions with same mean but different variances (noted as Synthetic 2 dataset in experiments). Each of the Gaussian distribution consists of 200 data records and they may appear at different point in time. 20 outlier points are generated randomly within a range specified that deviate largely from these distributions. Data records are fed to the algorithm 100 at a time so the window size is set at 100.

5.2.2 Real-life Dataset

For C_KDE_WR algorithm, we measure its accuracy on both synthetic datasets and real-world datasets. The synthetic datasets are generated using Gaussian mixture model same as we used in C_LOF algorithm. We also use two real-world datasets obtained from UCI machine learning library \(^1\) to measure how C_KDE_WR performs. The first dataset is the

\(^1\)http://archive.ics.uci.edu/ml/datasets.html
KDDCup99 network dataset for the intrusion detector learning task. The second is the “Covertype forest cover” dataset for cover type prediction task in forest. Both of these datasets are originally designed for classification tasks. In our case, we choose classes with minority instances as “outliers” points. For KDDCup99 dataset, we choose points belong to “normal”, “smurf” and “neptune” as inliers. Points belongs to other classes that have less than 1000 instances are considered to be outliers. The total dataset consists of 3.8M records, each with 11 attributes and among which, 2K records are outliers. For Covertype dataset, points belongs to classes “Spruce-Fir” and “Lodgepole Pine” were chosen as inliers. Only points from minority classes, such as “Cottonwood/Willow” were considered to be outliers. The resulting dataset consists of around 500K records with 10 attributes and among which, 500 records are outliers.

5.2.3 Streaming Environment

We take advantages of Apache Flink \(^2\) framework to create the streaming environment in our testing. In Flink, data streams can be simulated from different sources including sockets, networks, file systems etc. In our case, we just read the static files on disk and Flink will generate data streams from input files automatically. Additionally, Flink also supports windowing data streams into different chunks based on time, count and session. We apply count-based window as it guarantees enough data points for processing. Therefore, datasets are divided into fixed number of instances in each window chunks and are fed to the algorithms one at a time

\(^2\)https://flink.apache.org/
5.3 Evaluation Criteria

5.3.1 Accuracy Evaluation

To evaluate the accuracy, we use the same metrics as the binary classification task in machine learning. 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, which is widely used for accuracy evaluation in binary classification.

*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:

\[
F_{\text{score}} = \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}}
\]

We measure the accuracy of C_LOF against different parameters of \( k \) on two synthetic dataset settings generated from mixture of Gaussians as mentioned before: One is fixing on variance but varying mean, the other is fixing on mean but varying on variance. We want to show that C_LOF can detect concept drift and overcome the masquerading problem in outlier detection. As for C_KDE_WR, we compare its accuracy with the SOD_GPU algorithm purposed in [22] on the two real-life datasets as mentioned previously. We want to demonstrate that the proposed C_KDE_WR algorithm has further increased its accuracy by reducing false positives compared to its counterpart, SOD_GPU due to its novelties introduced.
5.3.2 Performance Evaluation

We use the latencies to measure the performances of these two algorithms. Latency is given by the number of time required to process a given number of data points. As two algorithms are implemented differently, we need to measure their latencies differently. For C_LOF, as it is implemented on a multi-core CPU using actor model, we will compare its latencies by different number of threads in the thread pool. In the CUDA programming on GPU, the majority of latency comes from the memory transfer between CPU to GPU. This can be affected by many factors such as whether the memory accesses is coalesced, whether shared memory is used, how the grids and blocks are being laid out etc. In this application, we will only measure the performance of C_KDE_WR by latencies on different dimensions of block size. The block size of a kernel function can affect whether the memory access on GPU is coalesced and therefore it has a huge impacts on the performance of the application.

All the experiments were taken on a machine with 2.6 GHz Intel Core i7 quad cores CPU with 256 KB L2 Cache per core and 8GB of host memory, and NVIDIA Geforce GTX 650 graphic cards with 768 CUDA cores and 2GB of device memory. The version of CUDA installed on the machine was 8.0.

5.4 Benchmark and Results

5.4.1 C_LOF

5.4.1.1 Accuracy

We test the accuracy of our C_LOF algorithm against different parameters of $k$, which by definition of LOF, is the minimum number of points in order to be considered as a cluster [27]. We set this parameter $k$ from 3 to 10 and test the corresponding accuracy metrics on
two synthetic datasets generated from mixture of Gaussians as per in Section 5.2.1. Note that setting $k$ to 1 and 2 would not have any practical meaning and therefore we do not measure C_LOF on these two settings. The following figures illustrate the results of these experiments:

Figure 5.1: Accuracy of C_LOF on different parameters of $k$ on Synthetic dataset 1

Figure 5.2: Accuracy of C_LOF on different parameters of $k$ on Synthetic dataset 2

The results indicate that the accuracy of C_LOF on both datasets peaked at parameter
settings of \( k \) at around 5. The precision metric seems to be stabilized on both synthetic datasets after \( k = 5 \), where it is equal to 1. However, both the recall and F-Score metrics have descended after \( k = 5 \) on synthetic dataset 1, while these metrics fluctuate after \( k = 5 \) on synthetic dataset 2. Overall, the C_LOF performs better on synthetic dataset 1 than on synthetic dataset 2, and on synthetic dataset 1, the F-Score can even reach over 0.8 at its highest setting. The results of these metrics show that the purposed C_LOF algorithm can accurately detect outlier points over data streams with concept drift with very low or even zero false positives on a proper parameter settings (e.g. \( k = 5 \)). However, there is still room for improvement on synthetic dataset 2 whose outlier points are masquerading inside inlier points.

5.4.1.2 Performance

As we implemented C_LOF in multi-core CPU using actor model, we measure the performance of C_LOF by the average time it takes to process one window of data, which in this case is 100, with different number of threads in the thread pool. The following table shows the result of this profiling:

<table>
<thead>
<tr>
<th>Time ( \text{Time(Avg)} )</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>1.27s</td>
<td>0.97s</td>
<td>0.91s</td>
<td>0.92s</td>
</tr>
<tr>
<td>Min</td>
<td>0.95s</td>
<td>0.87s</td>
<td>0.88s</td>
<td>0.79s</td>
</tr>
<tr>
<td>Max</td>
<td>2.03s</td>
<td>1.43s</td>
<td>1.22s</td>
<td>1.30s</td>
</tr>
</tbody>
</table>

From the table, we can see that from threads size of 4 to 8, there is a significant drop on the time it takes to process data in C_LOF. However, after threads size 8, there is not much improvement on latencies, or even become worse as the number of threads increase. This is because of the hardware limitation on the number of available processing units (cores) in CPU. As number of the threads increases, each processor may be assigned multiple threads; to process them concurrently, some techniques need to be used, such as time-slicing etc,
which may introduce some overheads.

5.4.2 C_KDE_WR

5.4.2.1 Accuracy

We test the accuracy of our C_KDE_WR algorithm with the SOD_GPU algorithm purposed in [22] on 2 real-life datasets as per in Section 5.2.2 and 2 synthetic datasets generated from mixture of Gaussian as per in Section 5.2.1. We record the accuracy metrics of C_LOF and SOD_GPU accordingly on all these datasets and compare their accuracies. The following figures show the results of these comparisons:

Figure 5.3: Accuracy comparison between C_KDE_WR and SOD_GPU on Synthetic dataset 1
Figure 5.4: Accuracy comparison between C_KDE_WR and SOD_GPU on Synthetic dataset 2

Figure 5.5: Accuracy comparison between C_KDE_WR and SOD_GPU on KddCup99 dataset
Figure 5.6: Accuracy comparison between C_KDE_WR and SOD_GPU on CoverType dataset

The figures shows that there is a significant difference between the C_KDE_WR and the SOD_GPU algorithms on synthetic datasets in terms of their accuracies. Specifically, C_KDE_WR outperform SOD_GPU in terms of its precision and overall F-Score metrics, but not on sensitivity (recall). This means that C_KDE_WR has reduced its false positives but also increased its false negatives at the same time. One possible reason for this is that some candidate outliers points staged in C_KDE_WR may be retained in memory forever and never get a chance to be output (rank bouncing back and forth).

On the other hand, we did not notice any significant improvements in accuracies on real-life datasets, especially on KddCup99 dataset where results are almost identical, although C_KDE_WR is slightly better with only 1% advantage in F-Score. The reason for this may because of the fact that the size of the KddCup99 dataset (3.8M records) is much larger than the sizes of those synthetic ones (only 200 records), making any improvement on results unnoticeable. For CoverType dataset, there are improvements but still, such improvements are not significant.
5.4.2.2 Performance

For all GPU applications, one of the major and critical performance bound is memory bandwidth and throughput. As mentioned earlier, we employed the $32 \times 32$ dimension thread block and took advantages of shared memory provided in CUDA to reduce the global memory bandwidth in Kernel 2 & 3. To see the effects of having different thread block layouts on memory latencies, we tried three different block sizes on Kernel 2 & 3, one with $32 \times 32$ dimension, the other $16 \times 16$ dimension and the last $8 \times 8$ dimension. They all process a same dataset of size 1024 and dimension of 64 with shared memory access. The following table shows the profiling results of these options given by nvprof:

Table 5.2: Latency of Kernel 2 & 3 with different block size

<table>
<thead>
<tr>
<th>Latency \ Block size</th>
<th>$8 \times 8$</th>
<th>$16 \times 16$</th>
<th>$32 \times 32$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time(%)</td>
<td>98.49%</td>
<td>98.21%</td>
<td>98.52%</td>
</tr>
<tr>
<td>Time(Avg)</td>
<td>187.32ms</td>
<td>181.25ms</td>
<td>181.65ms</td>
</tr>
<tr>
<td>Min</td>
<td>185.68ms</td>
<td>178.95ms</td>
<td>180.31ms</td>
</tr>
<tr>
<td>Max</td>
<td>189.46ms</td>
<td>182.64ms</td>
<td>182.69ms</td>
</tr>
</tbody>
</table>

As we can see from the table, there is a significant improvement on the latency between $8 \times 8$ thread block and $16 \times 16$ thread block but the change between $16 \times 16$ and $32 \times 32$ thread blocks was barely noticeable. There is due the the hardware limitation where only a maximum of 1024 threads are available per block in the provided NVIDIA GPU. As the number of the block size get closer to this maximum available threads per block, the change of latency will become increasingly un-noticeable.
Chapter 6

Conclusion and Future Works

6.1 Overview

In this thesis, we have proposed two online algorithms that are capable of detecting outliers over data streams. They both work by keeping a chunk of data points in memory at any point in time. This approach has been widely-used and well-known as *sliding window* in many streaming applications. Further to that, both algorithms keep statistical summaries of previous data to help predict the outlier scores of future points. The presence of statistical summaries also avoids storing entire historical data and therefore efficiently reduce the number of evaluations required on computation. Both algorithms employ weighting scheme on observed points by introducing a so-called *forgetting factor*. These two algorithms differ in terms of the way outlier score has been calculated and how the sliding window is used.

In C_LOF, the outlier score is calculated based on proximity and reachability distances between each data point with respect to its neighbours. This outlier score is termed as Local Outlier Factor. The sliding window is overlapping, which means each window in the time series will have large overlapping components with its neighbouring windows. This makes C_LOF work in an incremental fashion, as in each window we only need to process changes with respect to its previous window. As the algorithm is incremental by nature, it
is ideal for event-based and asynchronous implementation, which is why we use the actor model by message-passing. The result of this implementation is to make the algorithm scalable and possible for real-time results as well as resilient to failures.

Unlike C_LOF, C_KDE_WR calculates outlier score based on density of each data point over an approximation probability density function defined by kernel density estimator. It adopts non-overlapping window, which means every window in the time series will have completely different components compared to any other windows. This is exactly the same as batch processing except that we divide the original input into different chunks and process one after the other. The batch processing makes C_KDE_WR an ideal fit for parallelizing computation, which is what the CUDA framework in GPU programming is designed for. We use CUDA to accelerate the C_KDE_WR by defining different kernel functions with different block size on CUDA streams to achieve designed latencies for real-time results.

6.2 Conclusions

As shown by the experimental results, both algorithms can detect outliers over data streams accurately on different type of datasets with low number of false negatives, although with higher number of false positives on some real-life datasets. The C_LOF algorithm can detect outliers on synthetic datasets where inlier points are governed by Gaussian distribution where both concept drift and masquerading may occur. The C_KDE_WR algorithm outperforms the SOD_GPU algorithm in terms of accuracy, which is the state of the art by time of writing, on the exact same real-life datasets used for testing. It achieves this by lowering the number of false positives compared to SOD_GPU and therefore increasing the precision and F-score metrics. This further prove that our novel concept drift detection module added to the original methodology is effective. Both algorithms are scalable and therefore their throughputs can be increased as the number of computational resources
(CPU or GPU cores) grows. Armed with these, we can conclude that our algorithms can be applied in streaming applications to detect anomalies in real-time.

6.3 Future Works

Although the results seem promising, there are several directions for future works and improvements on this thesis. We will list some of them to give some ideas on how this work can be extended:

- Though we managed to drop the number of false positives in \(C_{KDE\_WR}\), its number is still high in some cases. This relies heavily on the efficiency of the pre-processing on raw datasets and the chosen of parameters. Many of them are application-dependant based on their precise definition of outlier. A more efficient preprocessing step for datasets from different domains is yet to be found.

- The computational complexity of the \(C_{LOF}\) algorithm is very high especially as dimension of data increases. This is due to the heavily applied \(K\-NN\) and \(K\-RNN\) queries used (especially in the clustering step). An efficient (or approximation) algorithm for clustering (based on reachability distances) and \(K\-RNN\) is to be developed in order to decrease the overall complexity of \(C_{LOF}\).

- Both proposed algorithms are developed to detect Type II outliers on temporal (time) context. It would be interesting to develop an outlier detection algorithm to detect Type II outliers on other contexts (e.g. locations), or even composite of them (e.g. time and locations). For instance, what would be an outlier temperature in Australia in December?

- Algorithm for detecting Type III outliers are barely found in the literature and therefore this area has much to be researched. The application results in such area would
rewarding. For instance, how to find a group of criminals who are trying to commit crimes collaboratively based on their behaviours, each of which may appear normal?

- We did not perform testing for C_LOF algorithm on real-life datasets. It would be interesting to see how C_LOF performs in real-life datasets, with proper preprocessing steps.

We hope this thesis can serve as a basis and give some ideas to future researchers on how to efficiently pull out outliers from data streams without any prior knowledge and secondary memory, and apply it into real-world applications to help find abnormalities in their data.
References


## APPENDICES

**Synthetic Dataset 1**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2.02662221</td>
<td>2.16625604</td>
<td>2.02013640</td>
<td>2.11741427</td>
<td>2.00652985</td>
</tr>
<tr>
<td>1.52834701</td>
<td>2.47014988</td>
<td>2.07025033</td>
<td>0.40492356</td>
<td>3.24261505</td>
</tr>
<tr>
<td>1.82511751</td>
<td>2.50790293</td>
<td>2.85526151</td>
<td>3.86733802</td>
<td>1.87672956</td>
</tr>
<tr>
<td>2.00871367</td>
<td>1.70943871</td>
<td>2.31880291</td>
<td>1.79864692</td>
<td>1.62720723</td>
</tr>
<tr>
<td>0.19705529</td>
<td>1.75354320</td>
<td>1.88072168</td>
<td>1.89384225</td>
<td>2.01245614</td>
</tr>
<tr>
<td>2.36565290</td>
<td>2.02046650</td>
<td>1.69870343</td>
<td>2.00864094</td>
<td>2.56456685</td>
</tr>
<tr>
<td>0.23681830</td>
<td>1.36230616</td>
<td>2.01635394</td>
<td>1.63477107</td>
<td>2.00337756</td>
</tr>
<tr>
<td>1.89065496</td>
<td>1.96219543</td>
<td>1.81390035</td>
<td>2.03697736</td>
<td>1.81558119</td>
</tr>
<tr>
<td>2.07701626</td>
<td>2.05066363</td>
<td>2.28723879</td>
<td>1.86271892</td>
<td>1.84712861</td>
</tr>
<tr>
<td>1.57578209</td>
<td>2.26546929</td>
<td>2.60554178</td>
<td>1.96605251</td>
<td>1.99049368</td>
</tr>
<tr>
<td>0.23388230</td>
<td>2.00303924</td>
<td>1.91632524</td>
<td>2.48973742</td>
<td>1.93694650</td>
</tr>
<tr>
<td>0.21040125</td>
<td>2.01236129</td>
<td>1.47933373</td>
<td>2.3027392</td>
<td>2.04726663</td>
</tr>
<tr>
<td>0.22893275</td>
<td>2.17096552</td>
<td>0.30919946</td>
<td>1.76525760</td>
<td>2.37713855</td>
</tr>
<tr>
<td>2.42026462</td>
<td>1.54525048</td>
<td>0.02268919</td>
<td>-0.70940147</td>
<td>2.97069167</td>
</tr>
<tr>
<td>1.14835985</td>
<td>2.20697132</td>
<td>2.26884992</td>
<td>1.51616422</td>
<td>2.10277712</td>
</tr>
<tr>
<td>2.50596335</td>
<td>1.44706077</td>
<td>-2.32447218</td>
<td>1.25262910</td>
<td>2.06852887</td>
</tr>
<tr>
<td>2.40188467</td>
<td>1.82250281</td>
<td>1.63015610</td>
<td>1.89586613</td>
<td>2.44038106</td>
</tr>
<tr>
<td>1.79452937</td>
<td>2.23940800</td>
<td>2.32082810</td>
<td>0.83009292</td>
<td>2.27844174</td>
</tr>
<tr>
<td>0.16486609</td>
<td>1.84758864</td>
<td>2.10916875</td>
<td>2.10791407</td>
<td>1.49520994</td>
</tr>
<tr>
<td>1.49759891</td>
<td>2.05476908</td>
<td>1.69050680</td>
<td>0.16271611</td>
<td>2.82192773</td>
</tr>
<tr>
<td>2.06337160</td>
<td>2.06887542</td>
<td>2.07721517</td>
<td>1.37817464</td>
<td>2.43669363</td>
</tr>
<tr>
<td>2.12611333</td>
<td>1.13491307</td>
<td>-2.72273100</td>
<td>-2.02639457</td>
<td>2.16841922</td>
</tr>
<tr>
<td>1.87863688</td>
<td>1.69251751</td>
<td>1.81486723</td>
<td>1.67412948</td>
<td>2.00954450</td>
</tr>
<tr>
<td>2.69228249</td>
<td>1.75705579</td>
<td>2.53938911</td>
<td>2.73921182</td>
<td>2.40813517</td>
</tr>
<tr>
<td>1.78888158</td>
<td>2.03206012</td>
<td>2.32612926</td>
<td>2.07967778</td>
<td>2.47219010</td>
</tr>
<tr>
<td>1.79808899</td>
<td>1.90016017</td>
<td>2.56197143</td>
<td>2.39002492</td>
<td>2.04865138</td>
</tr>
<tr>
<td>1.96496191</td>
<td>1.90428656</td>
<td>3.95229623</td>
<td>2.32959129</td>
<td>2.18971635</td>
</tr>
<tr>
<td>2.02932307</td>
<td>2.37359816</td>
<td>2.24699944</td>
<td>1.85426536</td>
<td>1.96589774</td>
</tr>
<tr>
<td>2.21807086</td>
<td>1.73733449</td>
<td>2.23964106</td>
<td>2.17708916</td>
<td>1.70930691</td>
</tr>
</tbody>
</table>
Synthetic Dataset 2

2.02662221,2.16625604,0, 2.02013640,2.11741427,0, 2.00652985,1.52834701,0
2.47014988,2.07025033,0
0.40492356,3.24261504,1
1.82511751,2.50790293,0, 2.85526151,3.86733802,1
1.87672956,2.00871367,0
1.70943871,2.31880291,0, 1.79864692,1.62720723,0, 1.97055291,1.75354320,0
1.88072168,1.89384225,0
2.01245614,2.36565290,0
2.02046650,1.69870343,0, 2.00840942,2.56456685,0
2.36818307,1.36230616,0
2.01635294,1.63477107,0, 2.00337756,1.89065496,0
1.96219543,1.81390035,0
2.03697736,1.81558119,0
1.80680692,1.91778079,0, 2.35913572,1.34419542,0
1.75027321,2.14664458,0, 1.87701626,2.05066363,0
2.28723879,1.86271892,0, 1.84712861,1.57578209,0, 2.26546929,2.60551478,0
1.96605251,1.99049368,0
2.33888230,2.00303924,0, 1.91632524,2.48973742,0, 1.93694650,2.10404125,0
2.01236129,1.47933737,0
2.30273792,2.04726663,0
2.28932752,2.17096552,0
3.09199461,1.76525760,1
2.37713855,2.24026462,0
1.54525048,0.02268919,1
-0.70940147,2.97069167,1
1.48359858,2.20697132,0
2.26884992,1.51616422,0
2.10277712,2.50596335,0
-1.44706077,-2.32447218,1
2.52629210,2.06852887,0
2.40188467,1.82250281,0
1.63015610,1.89586613,0
2.44038106,1.79452937,0, 2.23940800,2.32082810,0
1.83009292,2.27844174,0
1.64866095,1.84758864,0
2.10916875,2.10791407,0
1.49520994,1.49759891,0
2.05476908,1.69050680,0
KddCup99 Dataset

Please refer to http://archive.ics.uci.edu/ml/datasets/kdd+cup+1999+data

Covertype Dataset

Please refer to https://archive.ics.uci.edu/ml/datasets/covertype