FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Lightweight Real-Time Feature Monitoring

João Dias Conde Azevedo

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: André Monteiro de Oliveira Restivo, PhD

Second Supervisor: Pedro Manuel Pinto Ribeiro, PhD

Company Supervisor: Marco O. P. Sampaio, PhD

Company Supervisor: Pedro Cardoso Silva, Msc

July 24, 2020

Lightweight Real-Time Feature Monitoring

João Dias Conde Azevedo

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Prof. Luís Filipe Pinto de Almeida Teixeira
External Examiner: Prof. Alexandre Paulo Lourenço Francisco
Supervisor: Prof. André Monteiro de Oliveira Restivo

July 24, 2020

Abstract

Many real-time stream monitoring systems are static once deployed in a production environment. The engineers of those systems configure them under the assumption that future data flowing through the system roughly follows the same distribution as previously seen data. They do so consciously, to the best of their knowledge and available tools, keeping in mind that in the future they may need to reconfigure the system. Thus, even though the initial configuration of the system may be one of the best fits, over time, due to data pattern shifts, the initially deployed static system's performance gradually deteriorates. Data pattern shift detection refers to the process of finding patterns in data that do not conform to expected or usual behavior. Accurate and timely detection of data pattern deviations allows for immediate measures to be taken. Thus, the problem at hand is to determine when to reconfigure the system, *e.g.*, a Machine Learning model, based on an analysis of the drifts in the stream of data.

In this thesis, we design a method that makes use of lightweight streaming aggregations and distribution divergence functions to alert about deviations in data patterns in real-time, while in the presence of large volumes of high velocity, highly skewed and seasonal data. We develop a two-phased and two-windowed method: first, we perform a batch analysis on a reference window and then a stream analysis over a sliding streaming window. We aggregate the contents of both windows using an approximate histogram aggregation based on Exponential Moving Averages (EMAs). We do this for each event and each of its fields, also denominated features in our context. We measure the divergence between both reference and sliding window EMA-based histograms for each feature with the Jensen–Shannon Divergence function, but other distance functions could be used. For each feature, we find out the probability value of each divergence measure and apply the Holm-Bonferroni multiple test correction to find out which probability values are lowest and their associated features. We generate alerts based on a user-defined probability threshold, for each feature.

We evaluate our method through a series of tests, some with synthetic datasets and some with real data. We further split the tests into single and multi-feature analysis. Our method accurately detected the introduced anomalies in the experiments using synthetic datasets while maintaining high throughput, for single and multi-feature analysis. However, experiments with real data were not as accurate. Despite not knowing the specific reasons for that, we defer further investigation to future work and formulate a set of hypotheses that might explain it and hence are worthy of pursuing next. Our set of experiments supports the claim that sliding window aggregations and distribution divergence methods can be combined to detect data pattern shifts in streaming scenarios with constant time and memory complexity.

Keywords: data streams, monitoring, real-time, lightweight, concept drift

Resumo

Muitos sistemas de monitorização de *streams* de dados em tempo real são estáticos uma vez colocados em produção. Os engenheiros desses sistemas configuram-nos sob o pressuposto de que dados futuros que fluem pelo sistema seguirão aproximadamente a mesma distribuição que os dados vistos anteriormente. Fazem-no conscientemente, com as ferramentas disponíveis, tendo em mente que no futuro poderão precisar de reconfigurar o sistema. Assim, mesmo que a configuração inicial do sistema seja uma das melhores, ao longo do tempo, devido a mudanças nos padrões de dados, o desempenho do sistema em produção decai gradualmente. A detecção de mudança de padrões de dados refere-se ao processo de encontrar padrões em dados que não estão em conformidade com o comportamento esperado ou usual. Uma detecção oportuna e precisa dos desvios dos padrões de dados permite que medidas imediatas sejam tomadas. Assim, o problema em questão é determinar quando reconfigurar o sistema, *e.g.*, um modelo de *Machine Learning*, com base numa análise dos desvios de padrões da *stream* de dados.

Nesta tese, desenvolvemos um método que utiliza agregações de *streaming* leves e funções de divergência para alertar sobre desvios nos padrões de dados em tempo real, na presença de grandes volumes de dados que fluem a alta velocidade, são altamente *skewed* e sazonais. Desenvolvemos um método de duas fases e duas janelas: primeiro, realizamos uma análise *batch* numa janela de referência e, em seguida, uma análise *streaming* sobre uma janela deslizante. Agregamos o conteúdo de ambas as janelas usando um histograma aproximado baseado em Médias Móveis Exponenciais (MMEs). Fazemos isto para cada evento e cada um dos seus campos, também denominados *features* no nosso contexto. Medimos a divergência entre os histogramas de referência e os das janelas deslizantes baseados em MMEs para cada *feature* com a função de divergência de *Jensen-Shannon*, mas outras funções podem ser usadas. Para cada *feature*, descobrimos o valor de probabilidade para cada medida de divergência e aplicamos a correção de múltiplo teste de *Holm-Bonferroni* para descobrir quais os valores de probabilidade mais baixos e as *features* associados. Geramos alertas para cada *feature* com base num limite de probabilidade definido pelo administrador.

Avaliamos o nosso método através de uma série de testes, alguns com conjuntos de dados sintéticos e outros com dados reais. Dividimos ainda os testes em testes de análise a uma única *feature* e tests de análise com múltiplas *features*. O nosso método detectou com precisão as anomalias introduzidas nos testes com conjuntos de dados sintéticos, mantendo um alto *throughput*, tanto para análises com uma única ou com múltiplas *features*. No entanto, os testes com dados reais não foram tão precisos. Apesar de não sabermos as razões específicas para tal, passamos os próximos testes para trabalhos futuros e formulamos um conjunto de hipóteses que pretendem explicar o sucedido e, portanto, são dignas de serem exploradas em seguida. O nosso conjunto de testes apoia a afirmação de que métodos de agregação de janelas deslizantes e métodos de divergência de distribuição podem ser combinados para detectar alterações nos padrões de dados em cenários de *streaming* com complexidade temporal e de memória constante.

Acknowledgements

A lot of people contributed to this thesis and I will explicitly thank everyone, in no particular order.

I thank my academic supervisors, Professors André Restivo and Pedro Ribeiro, for their valuable insights on how to conduct a research project, namely a master thesis, and how to structure this final document. I thank them for the multiple times where they advised me on which direction to take and the multiple great ideas they contributed with. Above all, I thank them for always putting my interests first and ensuring I had the necessary conditions to research, develop and write my thesis.

I thank my company supervisors, Marco Sampaio and Pedro Silva, for the advice they gave me throughout this thesis which made it possible for me to develop this project, step by step. I thank them for the hours they spent discussing with me the next steps to take, experiments to conduct, new methods to try and for the remaining hours they spent analyzing the results with me. Finally, I thank them for the many hours spent reviewing this document.

I thank the entire Feedzai Systems Research team, namely, João Oliveirinha, Pedro Silva and Sofia Gomes for their daily support and suggestions. I thank them for the technical tips they gave me to surpass certain tasks that at first seemed impossible.

Last but certainly not least, I thank my family, friends and girlfriend for keeping me sane while I worked on this thesis during the COVID-19 lockdown.

João Dias Conde Azevedo

"Most of the world will make decisions by either guessing or using their gut.
They will be either lucky or wrong."

Suhail Doshi, Chief Executive Officer at Mixpanel

Contents

List of Figures

List of Tables

List of Algorithms

Abbreviations

CMS	Count-Min Sketch
DABA	De-Amortized Banker's Aggregator
EMA	Exponential Moving Average
FWER	Family-Wise Error Rate
HLL	HyperLogLog
HTM	Hierarchical Temporal Memory
IoT	Internet-of-Things
JSD	Jensen–Shannon Divergence
ML	Machine Learning
PDS	Probabilistic Data Structure
RDD	Resilient Distributed Dataset
RFS	Recalculate-From-Scratch
SOE	Subtract-On-Evict
SWAG	Sliding Window Aggregation
TPS	Transactions Per Second

Chapter 1

Introduction

1.1 Context

In the past decade, applications have become increasingly data-driven, placing data at the center of application design. For different use cases, the data is processed in different ways for different purposes. For instance, e-commerce platforms such as Alibaba[1] need to process a large number of daily transactions while ensuring that sales run smoothly and that products are delivered to customers' homes. Streaming services, like YouTube[2] and Netflix[3], try to guarantee that media content reaches up to millions of users simultaneously. Social networks are responsible for generating large volumes of data. Twitter[4], for example, generates more than 500 million *tweets* (posts in Twitter) per day[5]. *Tweets* may contain text, media content or both. Cybersecurity applications are yet another example of time-sensitive, data-driven applications. The entities that work in these use cases need to monitor user accesses and their actions in the network, where timely detection of intruders is critical to prevent them from tampering with the underlying system.

All of these applications generate large volumes of data over time which poses data storage and processing challenges. This data is characterized by its large volume, high velocity and high variety [1], creating large scale data management problems. Additionally, the large amounts of

[1] https://alibaba.com
[2] https://youtube.com
[3] https://netflix.com
[4] https://twitter.com
[5] https://blog.twitter.com/engineering/en_us/a/2013/new-tweets-per-second-record-and-how.html

information produced by such systems and use cases lead to the creation of multiple real-time unbounded datasets, also known as data streams. The information that flows through such a stream can be analyzed on-the-fly or stored for later processing. The former is best known as stream processing, whereas the latter is known as batch processing.

E-commerce applications have to process endless streams of credit card transactions. Live streaming services need to distribute the media content created by the *streamer* (content creator) and deliver it across multiple users. Social networks process and analyze hundreds of posts per second to find out trending keywords across their network and suggest them to users. Computer network monitoring systems aim to detect intruders and revoke their system access before they have time to corrupt the system or steal private information. All of the referred use cases rely on real-time processing of data for near real-time actions. Hence, stream processing fits these use cases better than batch processing.

As mentioned, the processing of data can be divided into two main categories: batch and stream processing. However, due to the increasing volume of data and the need for timely processing to allow companies to react to changing conditions in real-time, the demand for stream processing systems is increasing. Unlike batch processing of static datasets, where assumptions on the underlying data distributions can be made, streaming data is known for being non-stationary [2] where the value of the produced information lies in its recency [3]. *Recency* is measured under different scales for different use cases. For instance, consider the monitoring of geological data to forecast possible natural disasters such as earthquakes versus the monitoring of a computer network for intruder detection. In the former case, after the forecast of a future earthquake, governmental authorities need a couple of days to launch an evacuation plan and keep the population safe. However, in the intruder detection scenario, the decision of removing access to a user must be done as soon as possible, preferably in a few seconds. Hence, while in the first scenario information retrieved within a day would still be recent, in the second scenario, information is considered recent and relevant if delivered within seconds (or even milliseconds).

In either processing scenario, we usually want to apply operations on data, for example, to compute the maximum value, an average or a count of distinct elements. Operations that produce a single result when applied to datasets are commonly known as aggregations. In stream processing, we apply these aggregations to data streams to obtain valuable information. However, since a data stream is, effectively speaking, an unbounded dataset, if our aggregation requires a finite domain, there is the need to apply windowing techniques. A thorough analysis of stream windowing techniques is done in Chapter 2 of this document but for now it suffices to say that windowing is a mechanism to select a fine portion of an unbounded data stream, typically by defining a time or tuple-based window size, *e.g.*, events from the last three days or the last million events, respectively.

Consider the use case where we want to detect credit card fraud on the data stream of all the transactions made on Amazon[6] by every US citizen on a single day. In this case, Amazon's transaction stream will need to be monitored by another system, a fraud detection one. Such a

[6] https://amazon.com

system would have to process all incoming transactions and decide, in a fraction of a second, if a transaction is fraudulent or not. It is clear that manually monitoring such an intensive, latency-sensitive data stream is hard and error-prone, thus justifying the need for autonomous real-time data stream monitoring systems.

Many market solutions monitor streams of credit card transactions with the intent of fraud detection. In the context of this thesis, we work side by side with Feedzai[7]. In brief, the company's solution for fraud detection consists of a workflow of different components. Incoming events are fed into this workflow and processed accordingly. Workflow components can be sets of user-defined rules and/or Machine Learning (ML) models that output scores reflecting their belief on whether a transaction is fraudulent or not. ML models are configured and trained, before deployment, often with assumptions on the statistical distributions of future incoming data. Thus their model performance depends on how well those assumptions hold for future data. However, fraud patterns are not static over time, but seasonal — *e.g.*, Black Fridays, holiday shopping and product launches — and evolutionary — *e.g.*, new fraud attack methods by fraudsters. Due to these shifts in data patterns and distributions, the ML models' performance can deteriorate over periods of time. In the Machine Learning field, this performance decay caused by data pattern shifts is also known as concept drift [4, 5].

1.2 Objectives

Feedzai's data pattern shift challenge is not unique to them nor their field. Many analytical systems are static once deployed and are configured *a priori* under the assumption that future data flowing through the system will roughly follow the same distribution as previously seen data. The problem is that data pattern shifts will lead to poor performance of the previously configured system. Monitoring the data patterns themselves and alerting for changes in the underlying distribution would allow users to reconfigure the system before its performance declines.

Thus the objective of this thesis is to build a lightweight real-time system that monitors a stream of high volumes of high velocity, highly skewed and seasonal data and detects pattern shifts relative to a reference period.

1.3 Motivation

For any organization, the quality of their provided services is paramount. Therefore, it is common for companies to have auxiliary systems monitoring their services, testing their usage rate, availability and performance, just to name a few. The motivation behind this thesis is to produce a system capable of real-time monitoring a stream of data so that changes in the underlying stream are caught early, preventing a fall in the performance of another system analyzing said data stream.

We want to build a lightweight enough solution that can be integrated into existing workflows. Ensuring constant time and space complexity means that the system will remain usable for the

[7] https://feedzai.com

ever-growing volumes of data. Since such a monitoring system is not mission-critical — *i.e.*, the provided services do not rely on it — the operational cost associated with it must be kept to a minimum. Having a real-time system monitoring other real-time systems consuming as many resources as the latter would be far too costly.

1.4 Hypothesis

The question this thesis aims to answer is whether *sliding window aggregations and outlier detection methods combined can be used to detect data pattern shifts in data streaming scenarios in the presence of high volumes of high velocity, highly skewed and seasonal data in real-time and with a low memory footprint.* In this thesis, we study sliding window aggregations that replace the sliding window events in memory by a lighter representation, while still being able to provide an estimate of the aggregation value.

1.5 Methodology

The resulting data pattern shift reporting system must have the following properties:

- low memory footprint
- low latency to work in real-time
- fix a reference period as the normal state of the system
- explainable alerts

The hypothesis will be tested and evaluated on the previous criteria. Experiments will be made using multiple synthetic datasets and real data from the financial fraud space, provided by Feedzai.

1.6 Document Structure

This section outlines the content of each chapter of this thesis.

In Chapter 2 of this document, we brief the reader with the necessary background and relevant concepts needed to understand the following work, namely the topics of Large Scale Data Processing, Sliding Window Aggregations and Outlier Detection, as well as all relevant sub-topics inherent to them.

In Chapter 3, we examine state of the art methods related to the previously mentioned topics and discuss shortcomings, strengths and the applicability of such methods to our hypothesis.

In Chapter 4, we define the problem this thesis aims to solve, the issues with state of the art solutions, our proposed solution, the assumptions made with it and the research questions we intend to answer.

In Chapter 5, we discuss initial approaches and the reasons that led us to discard them, before presenting our proposed solution.

In Chapter 6, we describe the datasets used and the set of experiments that were performed to validate our method. We start with single feature experiments on synthetic data, then multi-feature analysis and finally move on to multi-feature analysis on real data.

In Chapter 7, we draw our final conclusions. We reiterate our hypothesis and research questions and provide answers to them. Finally, we conclude by summarizing our contributions and, make explicit current issues to be addressed as future work.

Chapter 2

Background

In this chapter, we discuss fundamental concepts that will help us identify the set of relevant techniques that are suitable to address the problem to be formulated and tackled in this thesis. In Section 2.1, we introduce batch and stream processing as two large scale data processing techniques. In Section 2.2, we review windowing mechanisms that essentially turn unbounded data streams to finite datasets. In Section 2.3, we introduce aggregations as a whole. Then, we specifically discuss sliding window aggregations, algorithms used for its computation and aggregation properties. In Section 2.4, we present Exponential Moving Averages as aggregators that replace the sliding window contents for a lighter representation, providing an approximate estimation of the aggregation value.

2.1 Stream Processing as a Superset of Batch Processing

Batch processing is the processing of data grouped by batches. A batch is a collection of data points that have been grouped by certain criteria. Examples of such criteria could be *"all the transactions from the past two months"*, *"the last million transactions"* or even *"all the transactions made by an user until now"*. Batch datasets are known as static data [6]. An example application of such a technique would be processing all the keywords searched for in Google's search engine[1], and then grouping the results by keyword to count how many times a keyword was used in searches (for trend analysis purposes). Between batch and stream processing, batch processing is the older paradigm. Historically, Apache Hadoop [7, 8, 9] was one of the first widely known

[1] https://www.google.com

frameworks for large-scale batch processing, created in 2005. Hadoop uses a programming model called MapReduce [10]. This paradigm applies a map function that processes key/value pairs and generates sets of intermediate ones, ultimately merging all intermediate values using a reducer function. To do this, the distributed computation framework reads and writes from disk on intermediate steps, which makes it inefficient for applications that reuse working sets of data across multiple parallel operations (*e.g.*, iterative Machine Learning algorithms). Nearly a decade later, Apache Spark [11, 12] was released as the industry accepted successor. Spark retains the scalability and fault tolerance of MapReduce but introduces Resilient Distributed Datasets (RDDs) [13]. An RDD is a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Additionally, Spark processes data in random access memory (RAM), while Hadoop MapReduce persists data to the disk after a map or reduce action. Thus, Spark achieves lower processing times.

Stream processing, on the other hand, refers to the processing of data streams. A data stream is essentially a continuous or unbounded dataset. This data needs to be processed sequentially and incrementally on an event-by-event basis. Streaming data varies widely, from logs generated by customers using mobile or web applications, sensor data from Internet-of-Things (IoT) devices to credit card transactions. These examples of infinite datasets contrast with batch processing which works on finite ones. This distinction is important since some aggregations are very hard and even impossible to compute with 100% accuracy for several use cases in an efficient manner on unbounded datasets. An example of such is counting distinct elements that flow through the data stream. An immediate solution would be to create a regular set of items or strings and have its length be the number of distinct elements processed. However, this approach is not very efficient and has a linear memory usage growth regarding the number of distinct elements. Therefore, as the number of distinct elements approaches large orders of magnitude we run out of memory.

In order to provide a streaming solution, Apache Spark created the Spark Streaming [14] module. Being a batch system in its core, Spark implements streaming as micro-batching [15]. This approach handles streaming data by grouping incoming events into very small batches. Then, to process the entire defined window of data, it joins all micro-batches into a larger one and performs the desired computations. However, this creates an artificial barrier that does not truly exist in the streaming context. More than conceptually different from a true stream processing system, processing a stream in a batch fashion, even if in small ones, presents some issues. First, since we are indeed working with batches, results will never be updated in real-time per event being processed. For example, if we use a batch size of 100 events, the system would only produce accurate results every 100 events. Usually, to handle system inactivity, these micro-batch systems also have the notion of updating the aggregations with incomplete batches, if no event has entered the system after a pre-specified time period has elapsed. As such, micro-batching was created as a natural implementation of stream processing using the available batch processing systems of the time, but can not be considered true stream processing.

The Venn diagram in Figure 2.1 [16] illustrates the relation of both processing techniques. A data stream is unbounded or infinite whereas a bounded dataset has a beginning and an end. Using

these definitions, the concept of an unbounded data stream contains that of a bounded dataset. That is, from an infinite stream, it is possible to fix left and right limits to obtain a bounded and finite dataset. This relation is visible in Figure 2.2 from the same blog post.

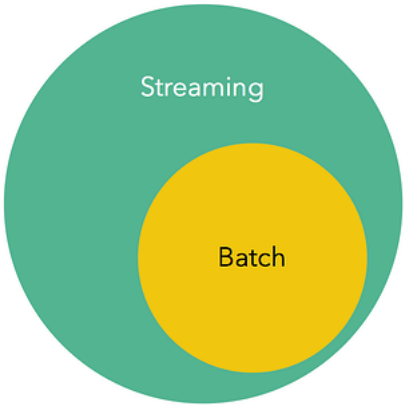


Figure 2.1: Stream processing as a superset of batch processing [16]

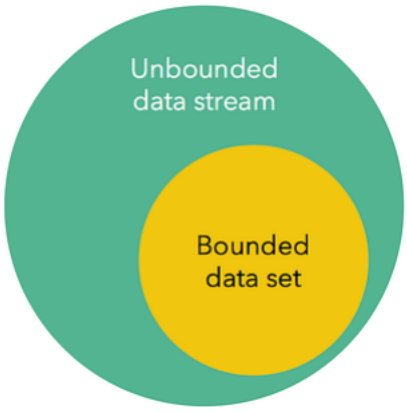


Figure 2.2: Unbounded data streams as a superset of bounded datasets [16]

In the next section, we will cover exactly how an unbounded dataset (also known as a data stream) can be converted into a bounded dataset with the use of windows.

2.2 From Unbounded Streams to Finite Datasets

In Section 2.1, we have defined a data stream as an unbounded dataset. But what exactly are the issues of working with an infinite stream of data? Why might we need to convert it into a bounded dataset and how to do so? In this section, we will analyze such issues and present sliding windows as the solution.

Computations over an unbounded dataset are possible. For example, computing the *average* of an event's field for all incoming events is computationally trivial: we need only to store the total number of events and the current total sum of all events arriving in the system. However, is

this boundless average over the stream of events useful? While in some scenarios it may be, more often than not we would like to introduce some boundaries into our computations. For instance, consider the monitoring of machines spread out across multiple racks in a large data center and its temperatures. Assume that temperatures measured across all racks are sent to a monitoring system in the form of a data stream. The average temperature of a machine over an infinite period (*i.e.*, since the machine was deployed) is of little value for detecting if the equipment is currently overheating, and to trigger the appropriate action to prevent a failure. In this scenario, an average of the temperatures per rack and for the last 5 minutes allows for a more actionable context, in this case identifying a likely-to-fail rack. To obtain such insight requires computing the average over the last 5 minutes' worth of data, effectively defining a begin and end — *i.e.*, all the data between the current timestamp in minutes *t* and *t-5*. By creating such a conceptual range, we realize that in practice we need a bounded dataset. Partitioning an infinite data stream to obtain such a finite dataset is made through the use of windows.

Botan *et al.* [17] define a window as:

Definition 1 *A window over a stream* $\mathbb{S}$ *is a finite subset of* $\mathbb{S}$

Windows can be categorized by two dimensions: the window size W and the sliding step size S. Windows can be defined as tuple or time-based. Tuple-based windows are windows whose size is defined by *number* of items contained in the window, whereas time-based windows are windows whose size is defined by the total *time* a window can store. For instance, a tuple-based window of size 10^3 tuples will store exactly 10^3 items. In contrast, a time-based window storing the last 5 minutes of events can contain a varying number of tuples but will always keep the events arrived in the last 5 minutes. A window slides over incoming data and groups it, enabling us to compute aggregations over a defined and bounded set of events. The step size S of a windowing technique is defined as the distance between consecutive windows [17]. This means that two consecutive windows W_1 and W_2 are exactly S units away. As mentioned, these units can be units of time or a number of tuples.

Windows can also be characterized by how they move across data, *i.e.*, how the step S is defined. The three most common ways are: true sliding, stepping or tumbling windows. A true sliding window is a window with a unitary step, *i.e.*, where $S = 1$. At each step, the window advances one unit, which may be one event for tuple-based steps, or one time-unit whether that is in milliseconds, seconds, days, weeks or another magnitude of time, measured based on the system clock or event time. Such a true sliding movement can be observed in Figure 2.3 with a tuple-based window of size W of three tuples and unitary step.

A stepping window behaves exactly like a true sliding one but the step size S is not unitary, *i.e.*, $S > 1$. In other words, a stepping window is defined by any window size W and steps S greater than a unit. Figure 2.4 represents a stepping tuple-based window of size W of three and a stepping size S of two.

A tumbling window is a window with a stepping size as large as the window, *i.e.*, $S = W$. Hence, tumbling windows partition the stream into non-overlapping chunks with the same size.

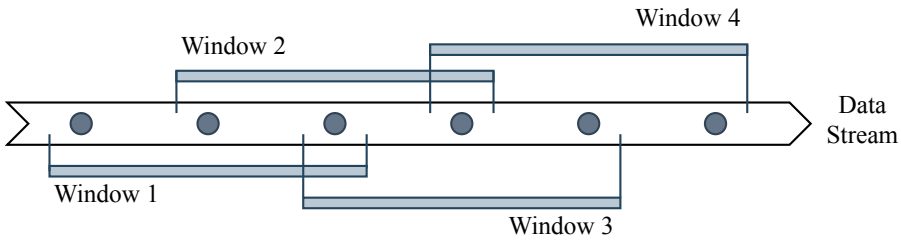


Figure 2.3: True sliding window, W=3, S=1

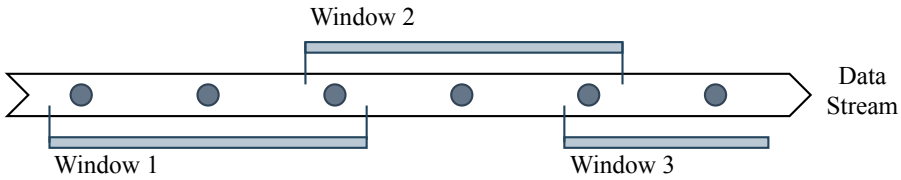


Figure 2.4: Stepping window, W=3, S=2

Since the intersection of different tumbling windows is always empty, each event is said to belong to exactly one tumbling window. An example of a tumbling window can be seen in Figure 2.5 with a tuple-based window of size W of three and an equally large step size S.

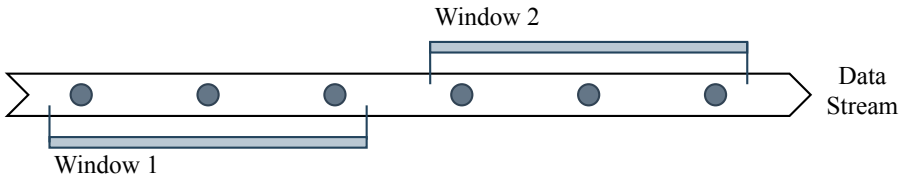


Figure 2.5: Tumbling window, W=S=3

The mathematical expression below summarizes our categorization of sliding windows.

$$Sliding\ window\ of\ size\ W,\ step\ S = \begin{cases} true\ sliding & \text{if } S{=}1 \\ stepping & \text{if } S{>}1 \\ tumbling & \text{if } S{=}W \end{cases} \tag{2.1}$$

For the scope of this thesis, we will work under a tuple-based true sliding window framework. However, despite focusing on tuple-based true sliding windows, our approach is just as valid for time-based windows and other step values as well.

2.3 Aggregations over Data

An aggregation — also known as a fold, reduce or accumulate function — is an operation that reduces a collection of values into a single one. Examples of such are the computation of dataset sums, minima, averages or the number of distinct elements (cardinality). An aggregation may be exact or approximate. Exact aggregations compute a value from a sequence of elements, where no error or loss of resolution exists. On the other hand, approximate aggregations have an error margin to the associated aggregation value which is usually controllable by adjusting a set of parameters. Approximate aggregations are most useful in situations where memory is limited, the volume of data to process is large and/or computing the exact aggregation can not be done in useful time. Thus, approximate aggregators trade precision of results for efficient time and space usage. In Section 3.3, we explore approximate aggregators that make such a trade-off.

Aggregations are present in most programming languages and frameworks. Quoting from the Java programming language documentation on the *Stream.reduce()* method [18]: *"A reduction operation (also called a fold) takes a sequence of input elements and combines them into a single summary result by repeated application of a combining operation"*. Similarly, the MapReduce [10] paradigm consists of first applying *map* steps to the collection of data and then a *reduce* one. *Map* tasks will transform each item as desired and the *reduce* task will merge the collection into a single value. Reduce methods make use of a *combine* function and recursively apply it. The *combine* function takes the current aggregation value and combines it with the next item in the set, returning the new aggregated state. Doing this recursively for each item in the collection results in a single value: the aggregate value. The *combine* function is from here onwards represented as $\oplus$ and the identity or neutral element of the aggregation as $\overline{\theta}$.

2.3.1 Sliding Window Aggregations

While some aggregations can be computed on an infinite dataset, often we want to compute these over a finite one. As we have seen in Section 2.2, we can obtain a bounded dataset from a data stream by applying windowing techniques. When doing so, we obtain a finite dataset denoted as a sliding window and compute aggregations over it. Tangwongsan *et al.* define such aggregations over sliding windows as Sliding Window Aggregations (SWAGs) [19]. As an example of a SWAG, consider the data center temperature monitoring problem from Section 2.2. In this example, we wanted to compute the average temperature per rack in the data center in the last 5 minutes. If we translate this SWAG into streaming SQL following a similar syntax to the one provided by Apache Flink [20] we would get:

```
SELECT rack_id, AVG(temperature)
FROM Temperatures
GROUP BY TUMBLE(timestamp, Time.minutes(5), rack_id)
```

In the query above, the `TUMBLE(timestamp, Time.minutes(5), rack_id)` statement groups incoming data using time-based tumbling windows of 5 minutes duration, using the `timestamp`

field to evict events not in this time interval. The resulting list of rows has `rack_id` as its key and as corresponding value a list of temperatures, from the past 5 minutes. Finally, for each rack and its list of temperatures, we compute an `AVG` aggregation, resulting in the average temperature per rack in the past 5 minutes. Note that this is very similar to any aggregation made in a standard Relational Database Management System (RDBMS). The main difference is that in standard SQL the `GROUP BY` clause includes only the columns on which the data will be grouped while in streaming SQL the `GROUP BY` also specifies the type of window used (*e.g.*, in this case, a tumbling window).

2.3.2 Sliding Window Aggregation Algorithms

A sliding window aggregation algorithm allows the computation of an exact or approximate aggregation over a sliding window. A SWAG algorithm may restrict the set of computable aggregations based on their properties. Different SWAG algorithms use different data structures for the window contents and the incremental aggregation state.

Two of the most basic SWAG algorithms are Recalculate-From-Scratch (RFS) and Subtract-On-Evict (SOE). Recalculate-From-Scratch (RFS) is the simplest and most general SWAG algorithm. As the name suggests, this algorithm recomputes the aggregation over the entire window for each update. For instance, the computation of a sum over a sliding window using RFS means that whenever the window changes all elements are summed. This approach works for every aggregation. However, given n as the window size, this algorithm takes $O(n)$ space and time complexity which makes it unfit in most scenarios of large scale data processing. Recall that the *combine* function is represented by $\oplus$ and the identity or neutral element of the aggregation by $\overline{\theta}$. Pseudocode for the Recalculate-From-Scratch algorithm is shown in Algorithm 1, assuming all functions have access to the *vals* queue which holds the window contents.

Algorithm 1 Recalculate-From-Scratch insert, evict and query methods

```
1: function QUERY
2:     agg ← θ̄
3:     for each v ∈ vals do
4:         agg ← agg ⊕ v
5:     return agg
6: function INSERT(v)
7:     vals.pushBack(v)
8: function EVICT
9:     vals.popFront()
```

Subtract-On-Evict (SOE) on the other hand, is a SWAG algorithm that incrementally computes the aggregation value. For each element inserted or evicted from the window, the aggregation value is updated without recomputing the entire aggregation like in RFS. SOE relies on expiring the evicted elements from the aggregation state. Hence, SOE only works for *invertible* aggregations, that is, a function $\ominus$ exists such that $(x \oplus y) \ominus y = x$, for all x and y. To illustrate the differences

between SOE and RFS, consider the computation of a sum aggregation over a sliding window. The total sum of the window will be the aggregation state or value. When inserting an element, the updated total window sum equals the current total sum plus ($\oplus$) the new element. The sum aggregation is considered *invertible* because, when evicting an element, the new aggregation state is the total sum computed thus far minus ($\ominus$) the evicted element. With window size of n, SOE has *O(n)* space complexity and *O(1)* time complexity. However, aggregations are not always invertible thus making this algorithm very restrictive and not always applicable. Algorithm 2 shows pseudocode for SOE assuming all functions have access to the window contents through the *vals* queue and access to the current aggregation state *agg*.

Algorithm 2 Subtract-On-Evict insert, evict and query methods

```
1: function QUERY
2:     return agg
3: function INSERT(v)
4:     vals.pushBack(v)
5:     agg ← agg ⊕ v
6: function EVICT
7:     v ← vals.popFront()
8:     agg ← agg ⊖ val
```

2.3.3 Aggregation Properties

In the previous section, we presented Subtract-On-Evict as an improvement over Recalculate-From-Scratch based on the idea of aggregation invertibility. The question then becomes if there are other aggregation properties besides invertibility that must be taken into account when analyzing general stream processing algorithms.

As it turns out, besides *invertibility*, other properties allow us to group aggregations in different families. Tangwongsan *et al.* construct Table 2.1 to summarize the aggregation families and satisfying properties [19]. To understand these properties, we must be familiar with the *combine* and *query* functions. The already mentioned *combine* function merges the incoming event with the incrementally computed aggregation state. The *query* function is responsible for returning a result for a given query according to the current aggregation state. For instance, consider once again the use case of computing a sum aggregation over a window of data. The *combine* function would be the arithmetic operator + and would merge the current aggregation state — *i.e.*, the current sum value — with the new incoming event. The *query* function would return the current aggregation state — *i.e.*, return the total sum computed. Based on the *combine* and *query* functions, the following are formal definitions of these properties. An aggregation is said to have:

1. an ***invertible combine***, if a function $\ominus$ exists such that $(x \oplus y) \ominus y = x$, for all x and y;
2. an ***associative combine***, if $(x \oplus y) \oplus z = x \oplus (y \oplus z)$, for all x and y;
3. a ***commutative combine***, if $x \oplus y = y \oplus x$, for all x and y;

4. a ***size-preserving combine***, if the aggregation state resulting from $\oplus$ always occupies the same space in memory;

5. a ***unary query***, if the query does not need any parameters — *e.g.*, a membership test with a Bloom Filter (detailed in Section 3.3.1) would require the element to test membership for as a parameter so it is not unary; returning the total sum is unary since it does not require any parameters.

	invertible combine	**associative combine**	**commutative combine**	**size-preserving combine**	**unary query**
sum-like	✓	✓	✓	✓	✓
collect-like	✓	✓	×	×	?
median-like	✓	✓	✓	×	?
max-like	×	✓	?	✓	✓
sketch-like	×	✓	✓	✓	×

Table 2.1: Aggregation properties. Checkmarks (✓), crosses (×), and question marks (?) indicate a property is true for all, false for all or false for only some aggregations of that family, respectively [19]

According to these properties, Tangwongsan *et al.* define five aggregation families [19]. These are the sum, collect, median, max and sketch-like families. The sum-like family of aggregations comprises aggregations that have an invertible, associative, commutative and size-preserving *combine* as well as a unary *query* function. Examples of such aggregations are the sum, count, mean/average and standard deviation operations. The collect-like family of aggregations has an invertible and associative *combine* function but their *query* function is not always unary. For example, keeping a string concatenation aggregation is invertible and associative but not commutative nor size-preserving. While in the string concatenation case you do not need to pass any arguments to the *query* function, in the case of another collect-like aggregation, for instance, keeping track of the *i-th* youngest element, you need to specify in your query the parameter *i* (*e.g.*, the 5th youngest element). Hence, their *query* function is not always unary. The median-like family of aggregations contains for example the percentile operation. For example, the percentile aggregation is invertible, associative and commutative but not size-preserving and requires an argument to the *query* function: the percentile to compute (*e.g.*, percentile 95 or 99). The max-like family comprises aggregations like the maximum or minimum, operations that are not invertible but associative and size-preserving, as well as unary from a *query* function perspective. The final family, the sketch-like family, comprises aggregations that are usually solved by approximate aggregators, also called probabilistic data structures or *sketches*. These aggregations can be membership queries, item frequency estimation or cardinality estimation.

The aggregation properties analyzed in this section will be used to evaluate SWAG algorithms. Different algorithms might only apply to certain families of aggregations. This way, the multiple SWAG algorithms presented throughout this thesis will be analyzed not only on their time and space complexity but also on the restrictions they impose regarding the aggregation' properties required. For example, under this framework, to implement Subtract-On-Evict (SOE), both the *combine* $\oplus$ and its inverse $\ominus$ functions must be implemented so that SOE applies $\ominus$ to expire the evicted element from the aggregation state. Since SOE only restriction is for the aggregation to be invertible it works for the *sum-like*, *collect-like* and *median-like* families in Table 2.1.

2.4 Exponential Moving Averages

We talked about sliding window aggregations and algorithms to compute them. We observed that the contents of the sliding window must be stored in memory to evict the oldest event and discount its effect from the aggregation when a new one is inserted. In this section, we analyze Exponential Moving Averages (EMAs). EMAs process each event only upon insertion. Since there is no eviction operation to perform, we do not need to store the whole window of data.

Exponential Moving Averages (also known as Exponential Weighted Moving Averages) [21, 22, 23] are a type of reduce functions applied to a sequence of elements, one at a time in an orderly fashion, to compute a weighted average, using different weights for each element. These weights decrease exponentially with the age of the observations. In other words, EMAs give higher weights to more recent data points. Figures 2.6 and 2.7 show the weights applied to past events. In a typical sliding window, events either have a weight of one (inside the window) or zero (outside the window) whereas EMAs give exponentially smaller weights to older events.

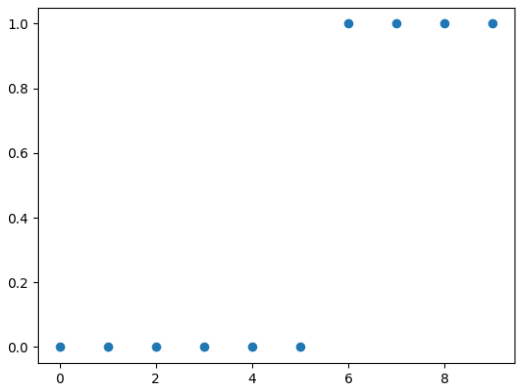


Figure 2.6: Window weights

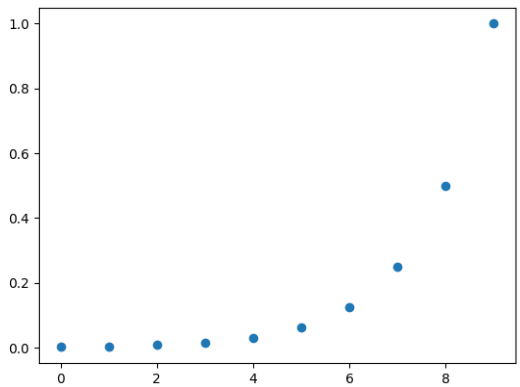


Figure 2.7: EMA weights

EMAs are recursive: the EMA value at timestamp *t* depends only on the EMA value at timestamp *t-1*. This recursive property allows us to store only one EMA value and continuously update it for each new value val_t, based on the following recursion formula:

Definition 2 $EMA_t = EMA_{t\text{-}1} * 2^{-\frac{\delta_t}{\tau_{1/2}}} + val_t$

where δ_t is the time passed between t and $t\text{-}1$ and $\tau_{1/2}$ is the time-based half-life of the EMA.

The half-life $\tau_{1/2}$ controls how fast old events are forgotten. The half-life is the amount of time that it takes for an element to see its contribution or weight reduced to half. For instance, with $\tau_{1/2} = 1s$, for each $\delta_t = 1s$ step, we reduce the weights applied to each event by half.

Note that it is also possible to express a tuple-based half-life $n_{1/2}$, *i.e.*, how many tuples or events must the EMA process before a certain element has its weight reduced by half. In this case, the definition does not take into account the time passed between events:

Definition 3 $EMA_t = EMA_{t\text{-}1} * 2^{-\frac{1}{n_{1/2}}} + val_t$

For example, with a tuple-based half-life of $n_{1/2} = 1$ tuple, for each new tuple all weights are reduced by half.

Definition 2 introduces the recurrence formula for time-based EMAs whereas Definition 3 works for tuple-based ones. In this thesis, we chose to work with tuple-based EMAs due to its simplicity, since it does not require keeping track of timestamp variations, but our approach can be adjusted to use time-based ones.

2.5 Summary

In this chapter, we introduced fundamental concepts to understand the set of methods that will allow us to test our hypothesis, presented in Chapter 3.

In Section 2.1, we introduced batch and stream processing as two large scale data processing techniques. We defined each, elaborated on the fundamental differences between both paradigms but also introduced the superset relation there is between stream and batch processing.

In Section 2.2, we discussed windowing mechanisms that allowed us to convert unbounded data streams (or infinite datasets) to finite datasets.

In Section 2.3, we introduced aggregations over datasets. We gave examples of computable aggregations over unbounded datasets at first, but then discussed sliding window aggregations, which make use of windowing techniques to obtain a finite dataset, on which the aggregation is computed. We further discussed algorithms used for the efficient computation of sliding window aggregations. Lastly, we presented a set of aggregation properties.

While sliding window aggregation algorithms keep the window contents in memory, Exponential Moving Averages, presented in Section 2.4, replace them for a lighter representation, providing an approximate estimation of the aggregation value.

In the next chapter, we review several outlier detection methods for time-series data and review state of the art sliding window aggregations and algorithms to summarize our data windows in a lighter representation.

Chapter 3

State of the Art

A data stream has a temporal dimension and the underlying process that generates data can change over time [24, 25]. In this chapter, we discuss the most relevant work for the problem of detecting data pattern shifts in real-time over true sliding windows using resource-lightweight approaches on streaming systems, presented in Chapter 1. We present a wide analysis of several categories of algorithms to be able to identify what's the region of this vast landscape of methods that may be useful to build the desired system.

3.1 Outlier Detection in Time-Series Data

Blázquez-García *et al.* review state of the art outlier detection techniques and present a taxonomy based on the main aspects of outlier detection methods [26]. From multiple reviews on these kind of methods [27, 28, 29, 30, 31] we chose to adopt the taxonomy presented by Blázquez-García *et al.* due to its focus on time-series data.

In this work, the authors define a time-series as a collection of data points sorted by time; and outliers as observations that do not follow the expected behavior. This definition is similar to the concept of outlier provided by Hawkins in 1980 where he states that an outlier is "an observation which deviates so much from other observations as to arouse suspicions that it was generated by a different mechanism" [32]. Furthermore, Blázquez-García *et al.* allude to the fact that outliers are often referred to in the literature as *"anomalies, discordant observations, discords, exceptions, aberrations, surprises, peculiarities or contaminants"* which for the purpose of their review and this thesis all refer to outliers. In summary, outlier detection refers to the process of finding anomalies in data. An anomaly (or outlier) relates to a point in time where the behavior of the

system does not match the expected one. Note that under this definition the presence of outliers or anomalies does not necessarily imply a problem.

The authors evaluate an outlier detection method on three different axes: the input data type, the outlier type to detect and the uni or multivariate nature of the method. Figure 3.1 presented by the authors in [26] summarizes these categories.

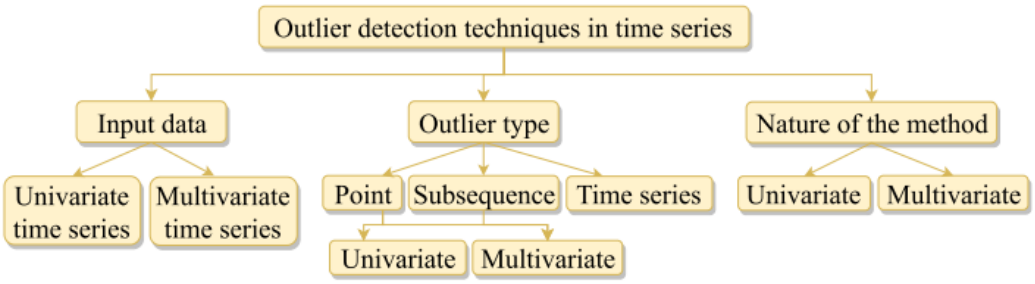


Figure 3.1: Outlier detection method taxonomy

The *input data* axis refers to the type of time-series the method uses as input. According to this criterion, an outlier detection method is said to work with a univariate time-series or a multivariate one. A univariate time-series refers to a time-series consisting of a list of values generated by a single time-dependent variable throughout time. On the other hand, a multivariate time-series has more than one time-dependent variable. Most importantly, each variable is not only time-dependent but may also have dependencies on other variables.

The *nature of the method* axis refers to the type of analysis performed by the outlier detection method. An outlier detection method may perform a univariate or multivariate analysis. A multivariate analysis considers all variables in the time-series simultaneously while univariate analysis looks at each variable independently. An outlier detection method can take as input a multivariate time-series but perform a univariate analysis, one per time-series of the multivariate input. On the other hand, to perform multivariate analysis, the input data type must be multivariate as well.

The third and final axis is the *outlier type* the method aims to report. The authors identify three distinct outlier types: point outliers, subsequence outliers and time-series outliers. A point outlier corresponds to a single data point in time that deviates from the expected value. These point outliers can be uni or multivariate, depending on the nature of the method. A univariate point outlier is a single time-dependent variable that stands out from the rest of the time-series. On the other hand, a multivariate point outlier takes into account multiple time-dependent variables in comparison to the remaining time-series. Subsequence outliers refer to a sequence of points in time that together show anomalous behavior, but may not be individually considered outliers. Similarly to point outliers, a subsequence outlier may take into account a single time-dependent variable or multiple ones. The authors classify them as univariate subsequence outliers and multivariate subsequence outliers, respectively. Lastly, time-series outliers refer to entire time-series being outliers themselves. This type of outliers can only be detected if the input data is multivariate, to be able to compare multiple time-series.

3.1.1 Univariate Point Outlier Detection

Univariate point outlier detection methods are further broken down into three categories: model-based, density-based and histogram-based (Figure 3.2) .

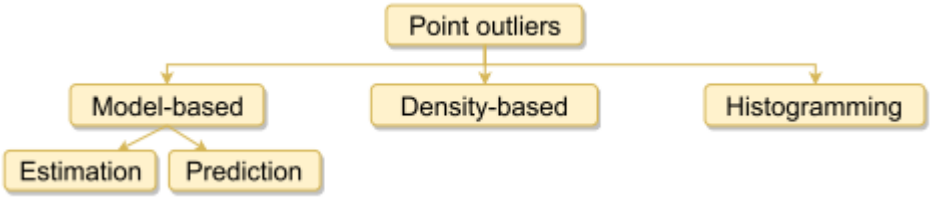


Figure 3.2: Point outlier detection in univariate time-series taxonomy [26]

The univariate point outlier detection methods based on models are the most common. For each timestamp of the time-series, the models compute the predicted value for the time-dependent variable and compare it with the observed value. If the difference between the predicted value and the observed one is above a certain threshold the data point at the current timestamp is considered an outlier. Mathematically, given the predicted value p_t, the actually observed value x_t and the user-defined threshold α, x_t is considered an outlier at timestamp t if and only if:

$$|x_t - p_t| > \alpha$$

Model-based point outlier detection methods can be one of two types: estimation or prediction models. The main difference is that estimation model-based methods use past and future data while prediction model-based methods use only past data and infer current and future values. Because of this, prediction model-based outlier detection methods can be used in real-time to process streaming time-series, while estimation based methods can only be used in batch.

Density-based univariate point outlier detection methods rely on the clustering of data points and neighbor counting. For example, nearest neighbor based methods consider a datum an outlier if it has less than k neighbors. Distance between two points is usually measured using the Euclidean distance [33]. Two points are said to be neighbors if they are close to each other — *i.e.*, the distance between them is less than or equal to a given threshold α.

The authors provide little detail on the last category, the histogram-based one. The authors state that these methods are "based on detecting the points whose removal from the univariate time series results in a histogram representation with lower error than the original".

3.1.1.1 Neighbor-Based Pattern Detection for Windows Over Streaming Data

Yang *et al.* propose *Extra-N* [34], a method for incremental detection of point outliers in sliding window scenarios. Data points are considered outliers if they have few neighbors, *i.e.*, less than a user-defined threshold α of neighbors within a user-defined range θ. Figure 3.3 depicts points

plotted in their feature space (in this case only two dimensions or features are used, x1 and x2) and a point outlier for $alpha = 2$ points and any given θ.

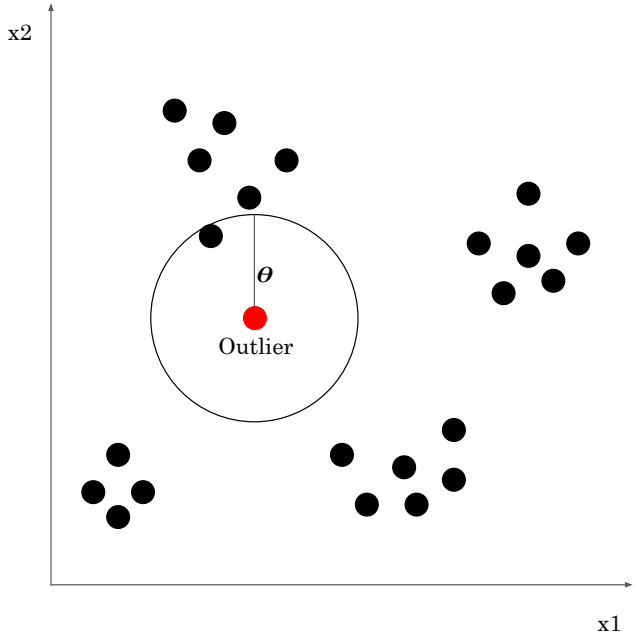


Figure 3.3: Density-based point outlier for $\alpha = 2$ and a given θ

The key idea is to predict the expiration of any data point in the sliding window and pre-handle the impact it has on its neighbors. The authors introduce the concept of *lifetime neighbor counts* or *lt_cnt*. The *lt_cnt* of a data point corresponds to a sequence of predicted neighbor counts, *i.e.*, the number of predicted neighbors a data point has in any future window where it exists. More precisely, each entry on *lt_cnt* records the number of neighbors that are known to survive in future windows.

For example, consider a window W_0 and a data point p_0 with three neighbors: p_1, p_2 and p_3. Assume p_1 expires after W_0, p_2 and p_3 expire after W_1 and p_0 expires after W_2. This implies that at W_0,

$$p_0.lt\_cnt = \{W_0 : 3, W_1 : 2, W_2 : 0\}$$

The lifetime neighbor counts (*lt_cnt*) carry enough information to compute density-based point outliers: for each data point check if the first or current element of *lt_cnt* is less than a user-defined threshold α to decide whether that data point is an outlier or not.

Method classification

Extra-N can take both uni or multivariate time-series as input data. However, it does not relate multiple time-series and it does not take into account possible dependencies between different time-dependent variables. In that sense, and according to the presented taxonomy (Figure 3.1), the method accepts uni or multivariate time-series as input but is of univariate nature. Last but not

least, Extra-N aims to detect univariate point outliers. These point outliers are identified based on the density of the region or neighborhood they belong too. Hence, we say Extra-N is a density-based univariate point outlier detection method (Figure 3.2).

Time complexity

To find outliers, Extra-N has to do a linear search through the list of points and check the neighbor counts. Hence, for outlier detection, this method shows $O(n)$ complexity with n as the number of points. Additionally, for each new data point p_i, Extra-N has to perform a range query search centered on it to find its neighbors and increment their lifetime neighbor counts by one. The runtime complexity of a range query search is $O(log(n))$ [35] with n as the number of data points.

In summary, Extra-N shows logarithmic time complexity when updating the sliding window and linear time complexity when looking for outliers, both relative to the window size.

Space complexity

Extra-N has to keep a lifetime neighbor count dictionary-like object for each window data point. For that reason, Extra-N achieves linear memory consumption relative to the number of data points in the window.

Applicability to our Hypothesis

Extra-N memory consumption grows linearly in regards to window size, so it does not fit our desired lightweight system architecture. Furthermore, updating the sliding window should be done in constant time, which is not the case for Extra-N. Hence, we will not use this algorithm.

3.1.1.2 Real-Time Stream Anomaly Detection with Hierarchical Temporal Memory

Ahmad & Purdy present an anomaly detection technique based on an online algorithm called Hierarchical Temporal Memory (HTM)[1] [36]. HTM has a few properties that are desirable for data stream anomaly detection scenarios. First, it is an unsupervised algorithm that requires no labels. This property comes in handy when monitoring a data stream in real-time, where labels for incoming data will likely be absent. Additionally, HTM adapts to noisy domains. In other words, random variations in the underlying distribution of incoming data that do not persist throughout time will be ignored. Furthermore, HTM detects both spatial and temporal anomalies — *i.e.*, changes in magnitudes of data or unusual timing for particular patterns, respectively.

HTM is an online learning algorithm, meaning that it adapts to changing statistics of the underlying data stream. In other words, if there is a sudden abrupt change in the distribution of a specific time-dependent variable, the algorithm will raise an alert. However, if an initial spike that was assumed to be an anomaly becomes frequent, then the algorithm adapts and assimilates this as "a new reality" and stops classifying it as anomalous. Unfortunately, this property is not desirable

[1] https://github.com/numenta/nupic

in our system. Our stream monitoring system must report changes relatively to an initial static configuration (reference or normality period) and should not accept a new distribution as the new standard.

Method classification

Similarly to Extra-N presented in Section 3.1.1.1, the Hierarchical Temporal Memory (HTM) algorithm accepts both uni and multivariate time-series as input data but does not perform a multivariate analysis. Hence, we say it accepts both uni and multivariate inputs but is univariate in nature. HTM focuses on finding point outliers, but unlike Extra-N, which is density-based, HTM is a model-based approach. Specifically, HTM is a prediction model-based approach for univariate point outlier detection (Figures 3.1 and 3.2).

Applicability to our Hypothesis

Online learning algorithms that adapt to data pattern shifts and eventually consider them regular patterns are not viable to our goals. We need a monitoring system that alerts data pattern shifts continuously until they match the initial configuration again. Additionally, insights produced by machine learning models are not human-understandable, leaving the user confused about why an anomaly report was made. In conclusion, we do not deem HTM applicable to our solution.

3.1.1.3 DeepAnT

Munir *et al.* present DeepAnT, a deep learning-based approach for the detection of anomalies in time-series data [37]. DeepAnT uses unlabeled data to capture and learn the distribution of the data stream used to forecast the normal behavior of the time-series. This method is unsupervised - *i.e.*, it requires no labels - which is an advantage in a streaming scenario where real-time decisions have to be made before labels are effectively-known. It is capable of detecting point anomalies in time series data with periodic and seasonal characteristics.

DeepAnT employs a Convolutional Neural Network (CNN) as its time-series predictor module. Before monitoring a data stream, the CNN model must be trained. The model can be trained with a small dataset while achieving good generalization capabilities due to the effective parameter sharing of the CNN. Additionally, the dataset used to train the model does not require labels.

Method classification

DeepAnT takes as input a uni or multivariate time-series but is a univariate method by nature — *i.e.*, it does not find correlations between multiple time-dependent variables. The model used by DeepAnT is a Convolutional Neural Network (CNN), used to make predictions on the future time-series values and measure differences with the actual observed values. Hence, DeepAnT is a prediction model-based univariate point outlier detection method (Figures 3.1 and 3.2).

Applicability to our Hypothesis

Similar to the HTM algorithm presented, DeepAnT adapts to the underlying distribution of the data stream. In other words, if the data patterns shift from the initial configuration but stabilize, DeepAnT will accept it as it is. As mentioned, our goal is to build a monitoring system that alerts data pattern shifts continuously, until they match the initial configuration again.

3.1.2 Univariate Subsequence Outlier Detection

Similarly to point outlier detection methods, Blázquez-García *et al.* [26] split subsequence outlier detection methods into uni and multivariate ones (Figure 3.1). Furthermore, the authors split univariate subsequence outlier detection methods into five different axes as seen in Figure 3.4.

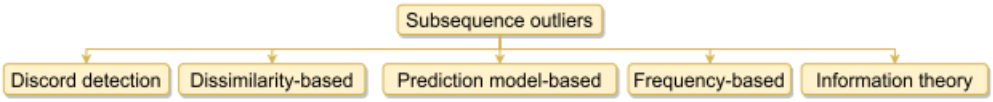


Figure 3.4: Subsequence outlier detection in univariate time-series taxonomy [26]

Methods in the discord detection category compare each subsequence in the time-series with the other subsequences within the time-series to find out the most unusual ones. Each subsequence is compared to the others using a distance metric, for instance, the Euclidean distance. The brute-force way of accomplishing this requires a pairwise comparison of all the subsequences, effectively resulting in quadratic time complexity. These methods find the most unusual subsequence in a time-series. However, as the authors put it, these methods lack a reference or normality state and hence do not accurately report outliers.

On the other hand, univariate subsequence outlier detection methods based on dissimilarity are identical to the previous discord detection based methods but use a reference of normality. The representations used for the subsequences and the reference of normality vary among methods in this class of outlier detection methods but they are based on comparison of the reference or normality subsequence with the ones being analyzed for outlier detection. Once again, the metric used to measure the distance between the reference period subsequence and the other ones depends entirely on the method and subsequence aggregation or representation. Finally, when the measured distance is above a certain threshold, these methods classify the subsequence as an outlier or an anomaly.

Prediction model-based univariate subsequence outlier detection methods assume that the reference or normality period is composed of past events. Similarly to univariate point outlier detection methods based on prediction models, these models train on past data and make predictions of the future. Subsequences that deviate from the model's prediction are classified as outliers.

Frequency-based univariate subsequence outlier detection methods also use a reference period and compute the frequency of each subsequence in the reference or normality period. Hence, a subsequence will be an outlier if it does not appear as frequently as expected in subsequent periods.

Lastly, we have information theory based methods for univariate subsequence outlier detection, which according to the authors *"are closely related to the frequency-based methods"*. This type of method not only measures the frequency of subsequences but relates it with the information carried by the subsequence. Mathematically, a subsequence S will be an outlier if

$$I(S) \times F(S) > \alpha$$

with $F(S)$ as the frequency of subsequence S, $I(S)$ as the information carried by S and α as a given threshold. According to the authors, *"I(S) is computed taking into account the number of times the symbols within S are repeated through the series"*. The authors explain that if *F(S)* is large — *i.e.*, S occurs frequently — then *I(S)* will be closer to 0.

3.1.2.1 A two window paradigm algorithm for change detection

Kifer *et al.* propose that detecting change in data streams can be reduced to testing if two windows have different underlying distributions [38]. The proposed method uses two windows, a reference W_1 and a sliding one W_2. Both windows are tuple-based, have size k and the sliding window is a true sliding one (recall Section 2.2 and the expression 2.1). The reference window W_2 works as a *reference of normality* and contains the first k points of the stream that occurred after the last detected change.

The algorithm begins by filling both windows with the first k tuples. Then it slides window W_2 by one tuple, since it is a true sliding window, for each incoming event from the data stream. In between slides, the algorithm checks for change by applying a distance function d taking as input both the reference and the sliding windows, which measures the discrepancy between both. Whenever this distance is above a certain threshold α, the algorithm raises an alert. After raising an alert, both windows are re-initialized with the next k items.

Note that after a change is detected, the reference window is updated with the next k tuples from the stream. As we have stated, we want a method that does not adapt to the data stream changes and instead keeps raising alerts until the sliding window data distribution returns to the one seen in the reference period. It is trivial to adapt this method to always use the same reference period. To do so, we simply keep the original reference window and do not update it after each alert.

This algorithm reduces change detection in data streams to testing whether two samples — *reference* and *sliding* windows — are generated by different distributions. The authors study methods for identifying differences in distribution between two samples and implement them as the d function. They provide as examples of a possible d function implementation the Kolmogorov–Smirnov Test [39] and the Wilcoxon-Signed-Rank Test [40].

Method classification

The method devised by Kifer *et al.* can take uni or multivariate input (if we use one pair of reference/target windows for each time-series) but performs a univariate analysis as it does not

find correlations between time-dependent variables. This two-windowed method measures the difference between both windows, the reference and the sliding one. Furthermore, instead of detecting point outliers, the alerts produced indicate the entire window (or subsequence) shows anomalous behavior. Hence, according to the taxonomy presented in Figure 3.4, this method is a dissimilarity-based univariate subsequence outlier detection method.

Time complexity

The authors show that it is possible to compute the distance between both windows in logarithmic time regarding window size k. Because the entire sliding window is held in memory, a window update is tantamount to removing the head of the list (element eviction) or appending a new element to the tail (element insertion), which is done in constant time. Thus, for each new event, the method evicts the oldest element, inserts the new one and computes the distance between the reference and sliding windows. In big O notation, the largest and determining factor is the computation of the distance function, done in $O(log(k))$ time, with k as the size of both windows.

Space complexity

The method makes use of two k-sized windows, thus storing $2k$ elements. Hence, memory consumption grows linearly with window size.

Applicability to our Hypothesis

In this thesis, we want to detect and alert data pattern shifts with a low memory footprint. The method proposed by Kifer *et al.* grows linearly with window size, making it unfit for large windows, our use case.

However, this two-window paradigm encodes a static normal or reference period in one window and the observed streaming values in a sliding one. Since in this thesis we want to report streaming data pattern shifts relatively to a reference period, we use a similar two-windowed model, but use lightweight and incrementally maintainable aggregations over both windows.

3.1.2.2 SAMM: Automatic Model Monitoring for Data Streams

Sampaio *et al.* present SAMM [41], an automatic model monitoring system for data streams. Similarly to the method presented in Section 3.1.2.1, SAMM uses two windows: the reference window R and the target window T. Window T contains the last n_t elements and window R the n_r elements of a reference period, which is prior to T. SAMM has two main components that interest us: the one responsible for computing the signal and the one that defines the signal threshold.

The Machine Learning (ML) model being monitored outputs a series of scores for each transaction. The authors build two histograms: one with the model scores for the reference window R and another for the target window T. The signal is a measure of similarity between both histograms H_R and H_T. The authors considered using the Kolmogorov-Smirnov [42, 39], Kuiper [43]

and Anderson-Darling [44] test but ultimately used the Jensen-Shannon Divergence (JSD) [45] to measure the similarity between both histograms. The JSD outputs a value between 0 and 1, with totally different histograms being assigned a value of 1. The signal is then computed by applying the JSD metric to both histograms H_R and H_T at each new event.

To compute the threshold for the JSD signal, the authors maintain a histogram of observed JSD values during streaming. To efficiently maintain the JSD histogram, they devise an algorithm called SPEAR which stands for Streaming Percentiles EstimAtoR. SPEAR is linear in time and space complexity regarding the number of histogram bins, which is a small and fixed constant. This way, for a new JSD value measured between the histogram H_R and histogram H_T, they compare it with the histogram and alert if above a certain percentile (*e.g.*, the 99th percentile).

Method classification

SAMM can take uni or multivariate time-series as input but performs a univariate analysis. This method is also a two-windowed method that measures the difference between reference and target sliding windows. The alerts produced report the entire target window as anomalous. According to the taxonomy presented in Figure 3.4, we classify this method as a dissimilarity-based univariate subsequence outlier detection method.

Time complexity

For each new event, SAMM updates the target window T, updates the target histogram H_T, computes the JSD between H_T and H_R, checks if this value is above a certain percentile and finally updates the histogram of JSDs with the new one.

Maintaining the target window, *i.e.*, evicting old events and inserting new ones, updating the target histogram H_T, computing the JSD between H_T and H_R and checking if it is above a given percentile are all done in constant time. Updating the JSD histogram and dynamically resizing the bins with the SPEAR algorithm takes $O(n)$ time with n as the number of bins. Because the number of bins is small and constant, the update of the JSD histogram is also done in constant time.

Overall, SAMM can process each event in constant time and is suitable for data stream monitoring.

Space complexity

SAMM keeps in memory a reference window, a target window, a reference histogram, a target histogram and a JSD histogram. The histograms can have a small and fixed number of bins so they are not the dominant factor contributing to SAMM's space complexity. Instead, because SAMM keeps two windows in memory, it stores all the events belonging to them. Assuming n_t as the size of target window and n_r as the size of the reference window, SAMM has $O(n_r + n_t)$ space complexity.

Applicability to our Hypothesis

SAMM was designed to work for small-time periods. Hence, for the authors, keeping the reference and target windows in memory was no issue. However, in our solution, we need sublinear time and space complexities to handle large windows of data. This implies we can not directly use SAMM to solve our problem.

However, SAMM has interesting properties we can use in our final solution. First, SAMM also uses a two-window paradigm: a reference and a target sliding one. This is similar to the method seen in Section 3.1.2.1, which may indicate this is a valid and tested approach to follow. Additionally, SAMM aggregates the contents of both windows as histograms and utilizes a statistical divergence test, the Jensen-Shannon Divergence, to measure the similarity (or dissimilarity) between both. This gives them a measure of divergence which they use to raise alerts, which is another great idea to test. Finally, and contrarily to the previous method (Section 3.1.2.1), SAMM does not simply threshold this divergence value: it builds a distribution of divergence values and measures percentiles for new event's divergence values. This allows a user to easier define a threshold: instead of specifying a threshold between 0 and 1 for the JSD value, the user defines only a probability value, *e.g.*, alert when the probability for a JSD value is equal or less than 1% (percentile 99th).

Even though SAMM is not directly applicable in our context, it provides us with valid and promising ideas to test.

3.2 Sliding Window Aggregation Algorithms

As discussed in Section 2.3.2, a sliding window aggregation algorithm allows the computation of an exact or approximate aggregation over a sliding window. We want to find an aggregation that is suitable to encode our reference and streaming periods so we avoid keeping the entire sliding window in memory. Furthermore, we need an efficient algorithm to incrementally maintain this aggregation. We have presented Recalculate-From-Scratch (RFS) and Subtract-On-Evict (SOE) as the two most basic generic stream processing algorithms. In this section, we analyze other more sophisticated SWAG algorithms that fit different use cases, restraining the set of computable aggregations to certain families, as defined in Table 2.1.

3.2.1 Two-Stacks

The Two-Stacks SWAG algorithm works for *associative* aggregations (recall Table 2.1) in scenarios where the sliding window has First-In First-Out (FIFO) semantics — *i.e.*, elements are evicted in the same order they were inserted. Two-Stacks [46] implements the FIFO queue using two stacks, namely the front stack F and the back stack B. Each element in each of the stacks is a pair <*val, agg*> where *val* is the item's value and *agg* is the aggregation value of everything below it on the stack. The pseudocode for Two-Stacks can be seen in Algorithm 3, assuming all functions have access to stacks F and B, with $\oplus$ as the *combine* function and $\overline{\theta}$ as the identity value of the

combine function, such that $\overline{\theta} \oplus x = x \oplus \overline{\theta} = x$. For example, if $\oplus$ is the arithmetic sum then $\overline{\theta}$ is 0. If $\oplus$ is the arithmetic multiplication then $\overline{\theta}$ is 1.

Insertion of a new item is always made on stack B. The updated aggregation state is computed using the *combine* function $\oplus$ taking as arguments the aggregation value *agg* of the top element of stack B and the item's value *val*. The pair pushed onto stack B is a pair with *val* and the new aggregation value $agg \oplus val$.

Evictions correspond to a simple pop from stack F. However, when F is empty, the elements from stack B are flipped onto stack F, re-applying $\oplus$ to make sure each element in F holds the aggregation value of everything below in the stack. Given n as the size of stack B, the occasional flip operation takes $O(n)$ time.

Computing the sliding window aggregation is done by applying the *combine* function $\oplus$ to the aggregation values of each of the stacks' top element.

Algorithm 3 Two-Stacks insert, evict and query methods

```
function GETSTACKAGGREGATIONVALUE(stack)
    if stack.isEmpty() then
        return θ̄
    else
        return stack.top().agg
function QUERY
    return getStackAggregationValue(F) ⊕ getStackAggregationValue(B)
function INSERT(v)
    B.push( {val: v, agg: getStackAggregationValue(B) ⊕ v}
function EVICT
    if F.isEmpty() then                ▷ // Flip B onto F and update top aggregation value
        while not B.isEmpty() do
            F.push( {val: B.top().val, agg: B.top().val ⊕ getStackAggregationValue(F) })
            B.pop()
    F.pop()
```

Time complexity analysis

Querying the aggregation state of the window and inserting a new element can be performed in constant time. However, while evicting an item is usually done in constant time, occasionally, a flip operation is required, consuming $O(n)$ time, with n as the size of stack B. Because this flip is occasionally performed, the time complexity of Two-Stacks is said to be *amortized* constant.

Space complexity analysis

Two-Stacks uses two stacks as auxiliary data structures. The combined size of both stacks is at most the size of the sliding window n. Hence, Two-Stacks has linear space complexity $O(n)$.

Applicability to our Hypothesis

Two-Stacks is a general sliding window aggregation algorithm to compute *associative* aggregations. However, it is linear in space and occasionally performs the computationally intensive operation of flipping stack B onto stack F, making it unfit for our use case.

3.2.2 De-Amortized Banker's Aggregator

The De-Amortized Banker's Aggregator (DABA) [46] is a SWAG algorithm that allows the computation of *associative* aggregations. This method carefully handles expensive operations and spreads them along the execution. As a result, the authors are able to turn the average-case behavior into worst-case behavior. This is commonly known as *de-amortization*, hence the name. The main idea behind DABA's design is to avoid occasional expensive operations — such as the flip operation in Two-Stacks — by gradually executing them throughout runtime.

Data Structure

DABA uses two queues, *vals* and *aggs*, both of size n, seen in Figure 3.5, which was directly taken from *"Low-Latency Sliding-Window Aggregation in Worst-Case Constant Time"* by Tangwongsan *et al.* Both queues share six pointers — F, L, R, A, B and E — which point to the same position for each queue. These pointers are always ordered as follows: $F \leq L \leq R \leq A \leq B \leq E$.

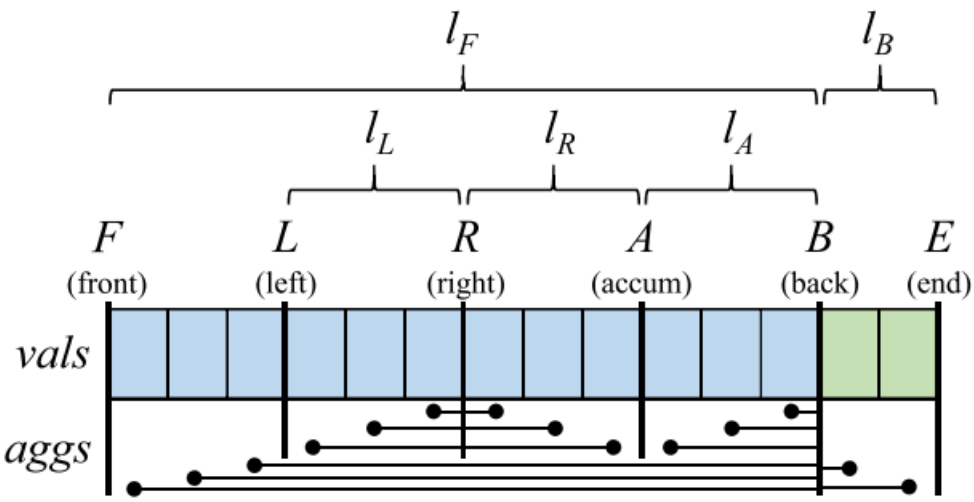


Figure 3.5: DABA data structure [46]

Queue *vals* holds the actual sliding window contents. New values are pushed in the back and old values are popped from the front of the queue. Queue *aggs* stores aggregations of sub-ranges of *vals*. Aside from pointer E, each pointer defines a sub-list and each sub-list is aggregated either to the left •— or to the right —•. If a sub-list is aggregated to the left •—, each *aggs* element will be the aggregation value of all elements in *vals* from that position to the right. On the other hand,

if a sub-list is aggregated to the right —•, each *aggs* element will be the aggregation value of all elements in *vals* from that position to the left. Values from *vals* can be aggregated to the left or to the right because DABA restricts the aggregation to be *associative* (refer to Table 2.1).

Figure 3.5 shows sub-lists l_F, l_B, l_L, l_R and l_A. Sub-list l_F is aggregated to the left for easier eviction. After an element is evicted, the new head of the queue *aggs* — pointed by F — already contains the aggregation value of all elements from *vals* to its right side. Hence, evicting an element amounts to evicting from both *vals* and *aggs* queue. Similarly, l_B is aggregated to the right to facilitate insertion. Sub-lists l_L, l_R and l_A allow for incremental reversal of the right-aggregated l_B to the left-aggregated l_F. This incremental reversal is tantamount to adjusting pointers that causes shifts in sub-list boundaries. Thus, elements of *aggs* may change from one sub-list to another and need to have its aggregation value recomputed.

Inserting a new element corresponds to pushing a new value *val* in *vals* and pushing the new aggregation value *agg* in *aggs*. The new aggregation value *agg* is computed by applying the *combine* function $\oplus$ to the back element of *aggs* and the new value *val* — *i.e.*, pushing *val* to *vals* and $aggs[E] \oplus val$ to *aggs*.

Evicting old elements is done by evicting the same number of elements from *vals* and *aggs*.

l_F and l_B combined cover the whole *vals* queue — *i.e.*, every element in the sliding window. l_F is aggregated to the left, thus *aggs[F]* contains the aggregation value of everything in l_F. On the other hand, l_B is aggregated to the right, thus *aggs[E]* stores the aggregation value of everything in l_B. Hence, the aggregation value of the entire window will be the result of $aggs[F] \oplus aggs[E]$.

Incremental Reversal of l_B to l_F

The main conceptual difference between DABA and Two-Stacks is that in Two-Stacks an occasional expensive *flip* operation takes place — flipping stack B onto stack F — and in DABA the reversal of l_B to l_F is incremental. This allows DABA to achieve constant time complexity (on average) rather than Two-Stacks' amortized constant time complexity.

Algorithm 5 shows the pseudocode presented by the authors for the DABA algorithm. In reality, after each insertion and eviction, besides the procedure already described, function *fixup* is executed. The *fixup* function performs the incremental reversal of l_B to l_F. Algorithms 4 and 5 show how such reversal is done but we will not elaborate on such procedure. However, it is important to stress out that this is the core idea of DABA: avoid occasional expensive operations, like Two-Stacks' *flip*, by spreading them out across runtime, in the form of DABA's *fixup*.

Time complexity analysis

Querying the aggregation state as described is done in constant time. Insertion and eviction of elements take constant time and invoke function *fixup*. The authors develop Theorems that prove function *fixup* is executed in constant time as well. Hence, DABA has constant time complexity.

Algorithm 4 DABA helper functions

```
1:  function GETLISTFAGGREGATIONVALUE
2:      if F == B then
3:          return θ̄
4:      else
5:          return aggs[F]
6:  function GETLISTBAGGREGATIONVALUE
7:      if B == E then
8:          return θ̄
9:      else
10:         return aggs[E - 1]
11: function GETLISTLAGGREGATIONVALUE
12:     if L == R then
13:         return θ̄
14:     else
15:         return aggs[L]
16: function GETLISTRAGGREGATIONVALUE
17:     if R == A then
18:         return θ̄
19:     else
20:         return aggs[A - 1]
21: function GETLISTAAGGREGATIONVALUE
22:     if A == B then
23:         return θ̄
24:     else
25:         return aggs[A]
```

Algorithm 5 DABA insert, evict and query methods

```
1:  function QUERY
2:      return getListFAggregationValue() ⊕ getListBAggregationValue()
3:  function INSERT(v)
4:      vals.pushBack(v)
5:      aggs.pushBack(getListBAggregationValue() ⊕ v)
6:      fixup()
7:  function EVICT
8:      vals.popFront()
9:      aggs.popFront()
10:     fixup()
11: function FIXUP
12:     if F == B then
13:         B ← E
14:         A ← E
15:         R ← E
16:         L ← E
17:     else
18:         if L == B then
19:             L ← F
20:             A ← E
21:             B ← E
22:         if L == R then
23:             A ← A + 1
24:             R ← R + 1
25:             L ← L + 1
26:         else
27:             aggs[L] ← getListLAggregationValue() ⊕ getListRAggregationValue() ⊕
    getListAAggregationValue()
28:             L ← L + 1
29:             aggs[A − 1] ← vals[A − 1] ⊕ getListAAggregationValue()
30:             A ← A − 1
```

Space complexity analysis

DABA stores the window contents in the *vals* queue. Queue *aggs* is of the same size as *vals* and stores partial aggregations over *vals*. Hence, if the window size is n, DABA uses two n-sized queues. Hence, DABA has $O(n)$ space complexity, meaning memory consumption grows linearly with window size.

Applicability to our Hypothesis

Even though DABA improves on the time complexity of Two-Stacks, it still shows linear space complexity regarding window size. In our monitoring system, we need to analyze large windows of data, thus making DABA inapplicable for our Hypothesis.

3.3 Probabilistic Data Structures

In Section 3.2 we presented some state of the art algorithms for computing sliding window aggregations (SWAGs). However, if the aggregation we desire to compute can be approximated, there may exist more memory efficient methods. These aggregators are commonly denoted as sketches or probabilistic data structures and produce results orders-of-magnitude faster than some exact approaches while keeping the memory consumption constant and providing mathematically proven error bounds. Each probabilistic data structure is used for efficient computation of one approximate aggregation, hence we say they are aggregation-specific.

Probabilistic data structures use hash functions to compactly represent a set of items. They require a single pass through the data, which is appropriate for a streaming scenario. Probabilistic data structures have constant space and time complexity [47] making them ideal aggregators to build our real-time and low memory footprint system.

3.3.1 Membership Queries and the Bloom Filter

A Bloom Filter [48] approximate aggregator allows membership queries on a set of elements. Bloom Filters answer the query: "Is element *el* in the set of seen elements so far?". Being a probabilistic data structure and approximate aggregator, Bloom Filters' answers have an error associated. *False positive* matches are possible — *i.e.*, saying *el* is in the set when it is not — but *false negatives* are not — *i.e.*, saying *el* is not in the set but it actually is. In other words, a Bloom Filter will accurately identify all items that do not belong in the set but will misclassify some items as being present in the set. Hence, a query returns either *possibly in set* or *definitely not in set*.

Data structure

A Bloom Filter is an array of bits of size m, initially all set to 0. Figure 3.6 represents a Bloom Filter with a bit array of size m=7 in its initial state.

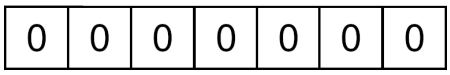


Figure 3.6: Bloom Filter bit array initial state of m=7

Element Insertion

Element insertion and query answering make use of k hash functions, each mapping elements to a position in the bit array. Adding an element is done by determining which positions should be set to 1. To that end, the new element is fed into each hash function that maps the element to an array position. The resultant k outputs are used as the k positions of the array to be set to 1. Figure 3.7 shows the insertion of element *el* in the Bloom Filter, making use of three hash functions (k=3), h_1, h_2 and h_3. The outputs of the hash functions correspond to the array indexes to set to 1.

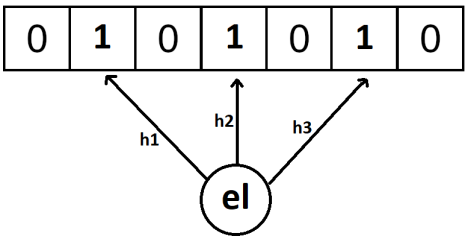


Figure 3.7: Insertion of element *el* into a Bloom Filter with m=7 and k=3

Membership Query

Performing a membership query — *i.e.*, testing if an element *el* is in the set — is done by feeding *el* into each one of the k hash functions in order to get an array of positions. If *el* was already in the set, all k positions should be 1. If any position contains a 0 then the element is *definitely not in set*, as shown in Figure 3.8.

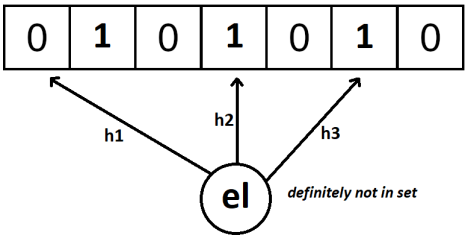


Figure 3.8: Bloom Filter membership test of element *el*. Finding at least one “0" indicates that it is not in the set.

If all positions are 1 then the element is said to be *possibly in the set*. The reason a Bloom Filter does not provide 100% certainty that *el* is in the set in case of all positions being 1 is that hash functions may give the same position for two different elements. For example, hash functions

h_1, h_2 and h_3 may map an element *el1* to positions $[1,2,3]$ and an element *el2* to positions $[4,5,6]$. This way, bits in positions $[1,2,3,4,5,6]$ are set to 1. When querying about the membership of a new element *el3*, hash functions h_1, h_2 and h_3 may map it to positions $[1,3,5]$ as exemplified in Figure 3.9. Since all of these positions are 1 because of the previous elements inserted, the Bloom Filter returns *"possibly in set"* when in reality *el3* was not in the set. This is considered a *false positive*.

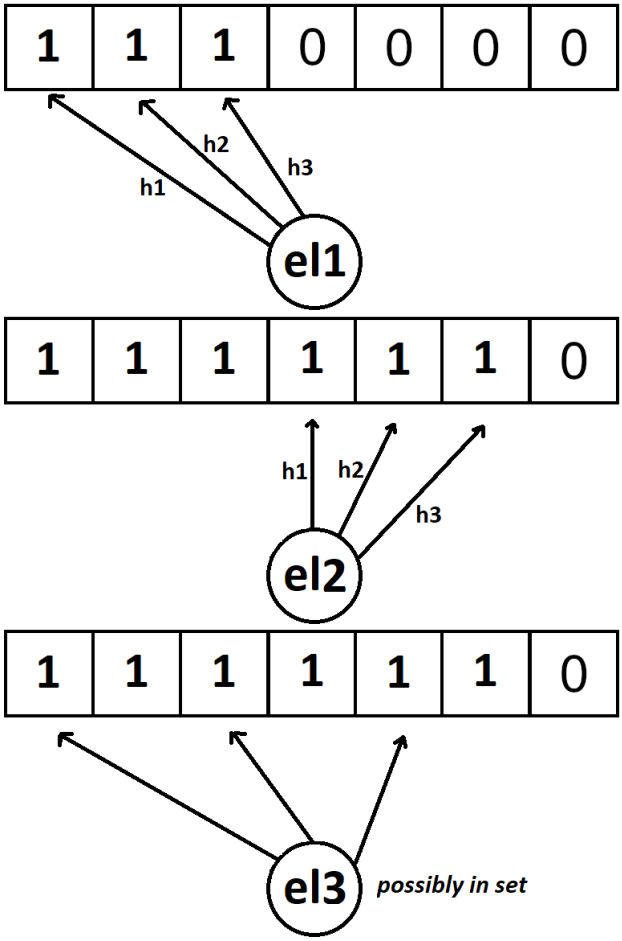


Figure 3.9: Bloom Filter false positive

False Positive rate

The false-positive rate is a function of the bit array size m, the number of hash functions k and the number of elements in the set n. It is assumed that the hash functions produce an independent and random index value for each element.

Mullin presents an analysis of the Bloom Filter in [49] where he details the false positive rate formula. The probability of one arbitrary bit being set to 1 is:

$$\frac{1}{m} \tag{3.1}$$

because there are m bits. Likewise, the probability of one bit not being set to 1 is

$$1 - \frac{1}{m} \tag{3.2}$$

Mullin states the probability that an arbitrary bit is not set to 1 after one element insertion and corresponding k bit updates is:

$$(1 - \frac{1}{m})^k \tag{3.3}$$

For n insertions, the probability a bit is not set to 1 is:

$$(1 - \frac{1}{m})^{kn} \tag{3.4}$$

Therefore, the probability that an arbitrary bit is set to 1 after n insertions is:

$$1 - (1 - \frac{1}{m})^{kn} \tag{3.5}$$

Hence, the probability of false positive of a membership query for a new element for k hash functions is given by

$$(1 - (1 - \frac{1}{m})^{kn})^k \tag{3.6}$$

Finally, this false positive rate can be approximated by

$$(1 - e^{-kn/m})^k \tag{3.7}$$

Optimal number of hash functions

We want to choose the number of hash functions k in order to minimize the false positive rate. In order to do so, we must first allocate m bits for the Bloom Filter, choosing m based on available memory. The value of k that minimizes the false positive rate is either of the nearest integers given by:

$$\frac{m}{n} ln(2) \tag{3.8}$$

Time complexity analysis

Considering a Bloom Filter with m bits and k hash functions, insertion has *O(k)* time complexity because all there is to do is run the input through all of the k hash functions and set the bits in the given positions to 1. Similarly, query answering has *O(k)* time complexity because it needs only to run the input through the k hash functions and check if all bits are set to 1. If so, it returns *possibly in set*. Otherwise it returns *definitely not in set*.

Note that the time complexity does not at all depend on the number of elements processed by the Bloom Filter and that k will be a rather small constant. Hence, the Bloom Filter is said to have constant time complexity.

Space complexity analysis

Considering a Bloom Filter with m bits, the space required is simply the array of m bits, thus $O(m)$ space complexity. Similarly to the time complexity analysis, we point out that m will be constant and that each of the array elements occupies 1 bit. Thus the Bloom Filter has a constant space complexity.

Applicability to our Hypothesis

Bloom Filters exhibit constant space and time complexity making them possible methods to test our hypothesis. Their false positive rate is tolerable in most scenarios and can be controlled by tweaking the values of m and k. However, we need to evict old events, and for that reason, Bloom Filters are of no use to us.

3.3.2 Item Frequency and the Count-Min Sketch

The Count-Min Sketch (CMS) uses two principles from the discussed Bloom Filters in Section 3.3.1. First, Bloom Filters show that precision can be sacrificed to achieve space savings. Second, they are very similar at a technical level.

A Count–Min Sketch [50] is an approximate aggregator that estimates the frequency of each element in the dataset. It is named after the two basic operations used to provide the estimate: counting (*count*) and computing the minimum (*min*).

Data structure

A CMS is represented by a two-dimensional array (or matrix) with width w and depth d. A CMS will make use of d hash functions, each associated with one row of the matrix. Each hash function will map elements to a position between 1 and w, the column of the two-dimensional array. Initially, all of the matrix positions are 0. Figure 3.10 represents the two-dimensional array used by the CMS and corresponding dimensions in its initial state.

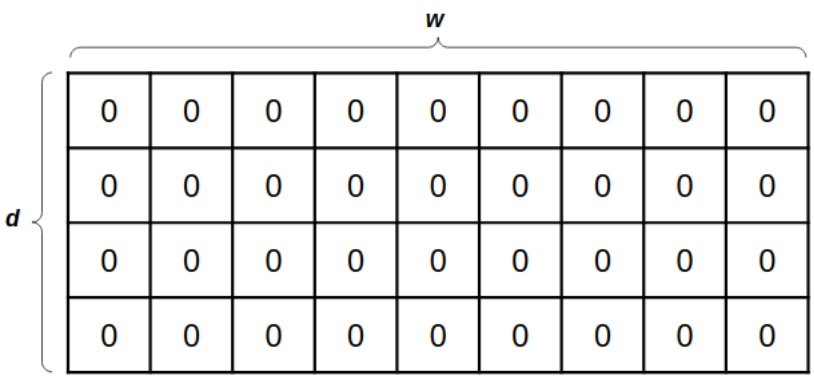


Figure 3.10: Count-Min Sketch matrix initial state of d=4 and w=9

Element Insertion

When adding an element, for each row i of the d rows in the matrix, we hash the element using that row's hash function to obtain the result j. Lastly, for all obtained pairs of i and j, increment matrix cells (i, j) value by 1, as seen in Figure 3.11. If elements have different weights these can be added instead of incrementing one unit.

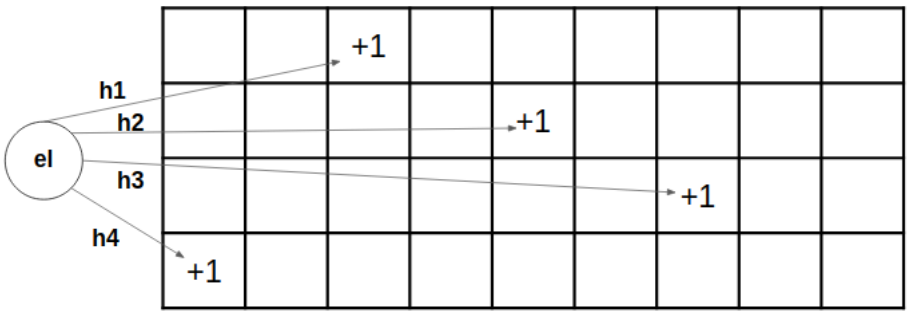


Figure 3.11: CMS mapping element *el* using d=4 hash functions to determine the column position for each row and increment it.

Frequency Query

Retrieving the minimum frequency of an element *el* can be done by taking the minimum value of all row counts for *el*. Mathematically, the frequency estimate is the minimum value of the set of all row values, for each row i of the total d rows, given by:

$$min\{matrix[i][h_i(el)]\}$$

Figure 3.12 illustrates this procedure, where the result of the query "What is the frequency of *el*?" returns "2".

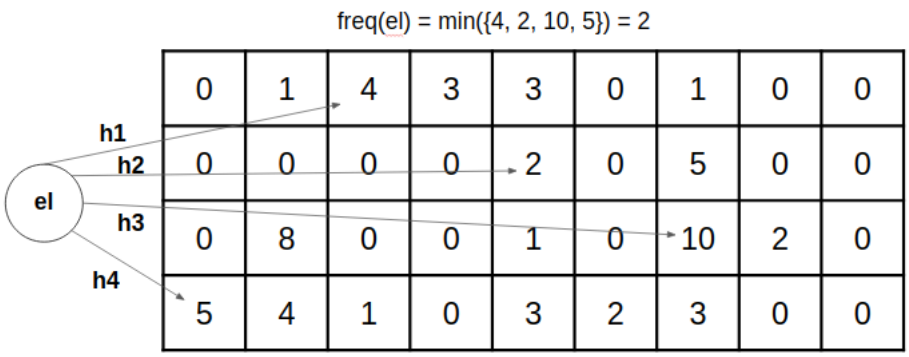


Figure 3.12: Count-Min Sketch frequency query of element *el*

Optimal number of hash functions

For an element *el*, each row will have one position incremented by *el*'s true frequency. However, multiple elements may be mapped to the same position within a row, thus overlapping their frequencies. Hence, for an element *el*, the CMS provides a frequency estimate that is greater or equal to the true frequency of *el* in the set.

The number of hash functions d to be used depends on the desired error probability. To achieve an error probability δ, d is given by the following expression:

$$d = \lceil ln\frac{1}{\delta} \rceil \tag{3.9}$$

For example, to achieve an error δ = 1%, d = 5 hash functions are necessary.

Time complexity analysis

When adding an element, the Count-Min Sketch loops through each row and applies a constant in time hash function, updating the value of the target cell. Hence, insertion loops through d rows and has time complexity of *O(d)*. Similarly, querying the frequency of an element *el* consists of computing the minimum of all row values. Since both insertion and query have constant time complexity, the CMS is constant in time.

Space complexity analysis

The Count-Min Sketch relies on a two-dimensional matrix as its only data structure. The memory used will correspond to the *wd* counts, hence *O(wd)* space complexity. Since both w and d are constants, the CMS has constant space complexity.

Applicability to our Hypothesis

Count-Min Sketches are constant in both time and space. Similarly to Bloom Filters, this makes them ideal candidates for our solution.

3.3.3 Cardinality Estimation and the HyperLogLog

A set of probabilistic counting algorithms is introduced by Flajolet *et al.* in [51] to estimate the number of distinct elements of a set — also known as the cardinality of the set. These techniques are based on *"bit-pattern observables"* in the binary representation of the hashed values. Bit-pattern observables are defined by the authors as *"patterns of bits occurring at the beginning of the (binary) S-values"*, with S as the target set.

Later on, the authors propose the LogLog algorithm [52]. In LogLog, the bit-pattern observable recorded for each item is the position of the leftmost 1-bit, denoted as P. The authors claim it is related to the total number of distinct elements in the dataset in that it is *"more or less a likely indication that the cardinality of S is at least* 2^P*"*.

HyperLogLog

A couple of years later, the same authors propose HyperLogLog (HLL) [53], an approximate aggregator to compute the cardinality of a set.

HLL uses the same bit-pattern observable as LogLog — P, the position of the leftmost 1-bit. With LogLog, the authors realize that storing only one observation produces extremely inaccurate predictions due to the high variability of a single value. The proposed solution is to apply *stochastic averaging*. As the authors state, stochastic averaging *"consists in emulating the effect of m experiments"*. Effectively speaking, the authors propose dividing the input stream into m substreams or buckets, maintaining one observable per bucket and computing P as an average of all bucket values (stochastic averaging).

Data Structure

HyperLogLog (HLL) maintains m bit-pattern observables, each being the leftmost 1-bit position, P. Since each position is a single integer, HLL maintains m integer counters, as seen in Figure 3.13. Each of these counters holds the maximum value for the bit-pattern observable seen so far. The first bit is assumed to be in position "1" and all counters begin with "0".

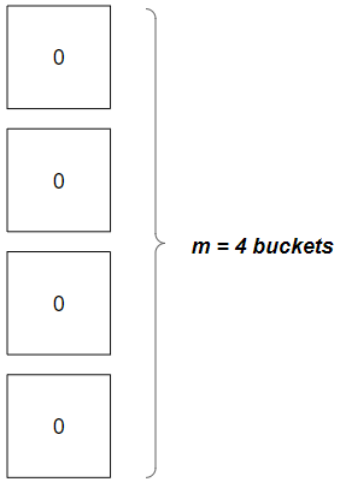


Figure 3.13: HyperLogLog with m=4 observable buckets

Element Insertion

In HyperLogLog, new elements are hashed into a binary string. The first n bits of the hashed representation will map to one of the $m = 2^n$ buckets. Effectively speaking, the first n bits work as a *bucket ID*. The bit-pattern observable P — the position of the leftmost 1-bit — is counted from the binary string discarding the first n bits. The observable P is stored in the pre-determined bucket if it is greater than the value the bucket contains.

Figure 3.14 demonstrates such procedure for a new element *el*. The hash representation of *el*, using $n = 2$ bits, maps to bucket *10*. Discarding the first $n = 2$ bits and counting from 1, the position of the leftmost 1-bit is $P = 4$. The previous value of bucket *10* was 0 and because $P = 4$ is greater than 0 the bucket value is updated.

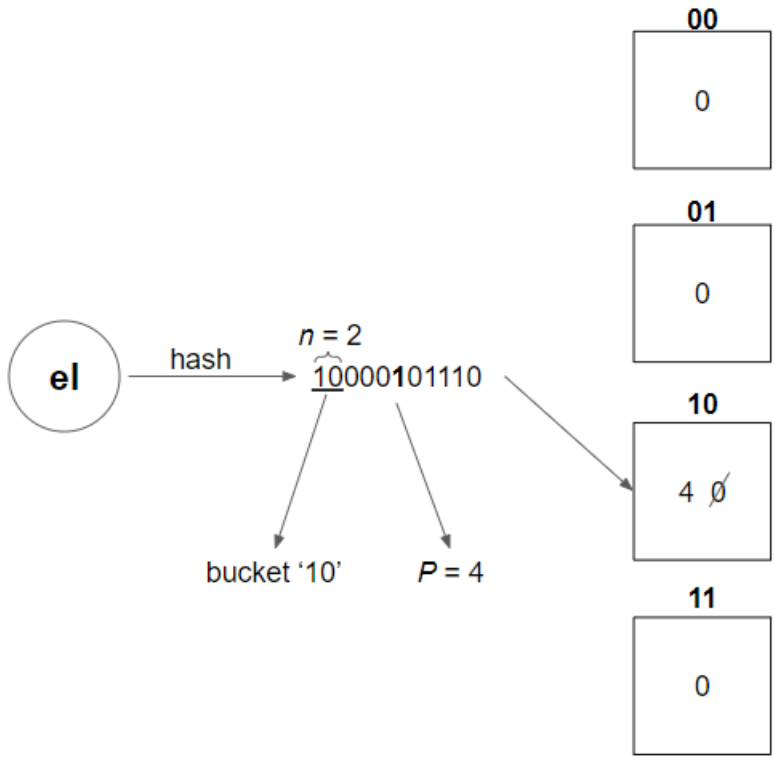


Figure 3.14: HyperLogLog *el* insertion with *m*=4 and *n*=2 bits

Cardinality Query

The cardinality of the set is estimated to be 2^P where the bit-pattern observable P is the position of the leftmost 1-bit. As discussed, HyperLogLog (HLL) differentiates from LogLog in that it applies a stochastic averaging process. Hence, the value of P will be the harmonic mean of all m buckets.

Time complexity analysis

Adding an item is tantamount to hashing it, determining the corresponding bucket based on the first n bits, counting the leftmost 1-bit position and updating the bucket value if need be. Estimating the cardinality of the set amounts to finding the maximum value P between all buckets and computing 2^P. Hence, both inserting a new element and estimating the cardinality of the set take constant time.

Space complexity analysis

Quoting Flajolet *et al.* about the memory consumption and error in the approximations obtained in their experiments in the original HyperLogLog paper [53]: "cardinalities till values over N = 10^9 can be estimated with a typical accuracy of 2% using 1.5kB (kilobyte) of storage."

HyperLogLog uses m buckets that store a single integer value, thus the space complexity is $O(m)$ where m is a rather small constant. Hence, HyperLogLog has constant space complexity.

Applicability to our Hypothesis

Similar to the previously studied approximate aggregators, HyperLogLog (HLL) has constant space and time complexity and provides an estimate of a set's cardinality. Hence, HLLs are considered valuable aggregators for our thesis.

3.4 Sliding Window Aggregations with Probabilistic Data Structures

In Section 3.3 we presented a set of probabilistic data structures constant in time and space. These structures aggregate an endless stream of data. However, when working with sliding windows there is the need not only to insert new events but to evict old ones. The probabilistic data structures described until now can aggregate new incoming data but do not address the type of temporality needed for sliding windows — *i.e.*, the ability to expire old elements from the aggregated state.

The probabilistic data structures studied can be implemented under generic sliding window aggregation algorithms like Recalculate-From-Scratch or DABA. However, doing so means we lose the main advantage of using these approximate aggregators: the constant space complexity. For example, cardinality estimation of a sliding window can be done employing the HyperLogLog heuristic of estimating cardinality as 2^p, with p as the maximum position seen for the leftmost 1-bit, using DABA 3.2.2 for the computation of p per sliding window. With this approach, despite computing the aggregation state in constant time, we have a linear in window size space complexity solution. Thus, in this section, we explore sliding window implementations of the previously discussed probabilistic data structures that keep the space and time complexity at the very least logarithmic.

3.4.1 Sliding HyperLogLog

The HyperLogLog (HLL) aggregator presented in Section 3.3.3 will only be useful to us if it can be implemented efficiently in a sliding window fashion. The HLL aggregator estimates the set's cardinality assuming it is close to 2^p, where p is the maximum position seen for the leftmost 1-bit in the binary values of all of the set items' hashes. In practice, all the aggregator has to compute is p, the maximum value of all observables. Stochastic averaging is applied by emulating the effects of m experiments using m buckets to store m bit-pattern observables.

3.4.1.1 Sliding HyperLogLog using Lists of Future Possible Maxima

The work of Chabchoub *et al.* adapts the original HLL aggregator from Flajolet *et al.* to work under a sliding window framework [54]. The main idea behind this implementation is to replace every HLL counter with a *List of Future Possible Maxima (LFPM)*.

Data Structure

The adaptation of the original HLL to work under a sliding window scenario requires the storage of additional information. The original HLL maintained m integer counters, each storing the maximum for the leftmost 1-bit position of the analyzed binary strings. The counter held the maximum value for all the seen binary strings so far but there was no way to expire old items. To allow eviction, the proposed data structure is very similar to the original m integer counters but keeps a list of values per bucket and associates a timestamp to each.

The stochastic averaging process of HLL remains unchanged: incoming items are hashed and distributed across m buckets. However, the authors propose keeping a short list of *packets* per bucket instead of a single integer count. A *packet* is defined as a pair $<t_i, p_i>$ where t_i is the arrival timestamp of the packet and p_i the position of the leftmost 1-bit in the binary hash representation of the associated value. Packets are only kept in the list if they are possible maxima over future windows. Because of that, the authors name this list a "List of Future Possible Maxima (LFPM)".

Element Insertion & Eviction

Similarly to the original HLL, incoming elements are hashed and the first n bits of the string are used to map the new element to one of the m buckets. The first n bits are then discarded and the bit-pattern observable p is computed from the remaining string.

Since each bucket now holds a LFPM instead of a single integer, the procedure for bucket update is different. Upon insertion of an element k, a packet $< t_k, p_k >$ is created with t_k as the timestamp of arrival of the new element k and p_k as the position of the leftmost 1-bit position of k's binary representation. Given the new packet $<t_k, p_k>$, the authors propose the following LFPM update method, that checks for and evicts old packets at each insertion:

- Delete *old packets*: packets that do not belong in the window anymore — *i.e.*, packets $<t_i, p_i>$ with $t_i \leq t_k$ - WindowSize
- Delete less or equally valued packets: packets that will never be a maximum — *i.e.*, packets $<t_i, p_i>$ with $p_i \leq p_k$
- Add the new packet $<t_k, p_k>$

Sliding Window Cardinality Query

Estimation of cardinality is done similarly to the original HLL cardinality estimation algorithm. To estimate the cardinality at a timestamp t, from all m buckets and all m lists, select packets that

are in the window — *i.e.*, packets <t_i, p_i> where $t_i > t$ - WindowSize. This filtering is necessary since the proposed algorithm evicts old packets only at element insertion and at the time of the query some packets may no longer belong to the window. Next, from the filtered set of packets, for each bucket and its list, compute the maximum p_i value. In the end, compute the harmonic mean between all values, just like the original HLL. The result will be the averaged bit-pattern observable value p and the cardinality is estimated to be 2^p.

Time complexity analysis

When performing insertions or evictions, the algorithm requires scanning through the Lists of Future Possible Maxima (LFPMs) to delete old packets or packets that will never be window maxima. Similarly, we need to scan the lists when estimating the cardinality to filter for packets within our window. Denote L_n as the mean size in packets for each LFPM. There are m buckets, each with a list. To scan through all lists we need to analyze $m\ L_n$ elements.

The authors prove that L_n has an upper bound of $ln(n/m)$, with n as the cardinality of the set and m as the number of buckets used. Hence, in the worst-case scenario, we need to analyze $m\ ln(n/m)$ elements. This means that a Sliding HLL implementation using Lists of Future Possible Maxima has $O(ln(n))$ time complexity, given the number of buckets m is constant.

Space complexity analysis

The memory used in the Sliding HLL aggregator implemented with Lists of Future Possible Maxima (LFPMs) amounts to the total size of the m lists. Assuming 4 byte timestamps and 1 byte bit-pattern observables for each packet, the aggregator uses $(4+1)\ L_n\ m$ bytes.

The lists are dynamic thus they shrink and grow at runtime. However, $ln(n/m)$ was proved to be an upper bound for L_n. Hence, this sliding implementation of the HLL aggregator consumes up to $(4+1)\ ln(n/m)\ m$ bytes of memory. Considering that the number of buckets m is constant, the space complexity of a Sliding HLL implementation with LFPM is $O(ln(n))$.

The authors claim that the proposed sliding version remains as accurate as of the original HLL aggregator. With an additional memory of only 35kB, it is possible to estimate the number of distinct elements in the set with a standard error of 3% in the presence of up to a million distinct elements.

Applicability to our Hypothesis

Implementing a Sliding HLL using Lists of Future Possible Maxima (LFPMs) shows logarithmic time and space complexity while providing estimates with the same accuracy as the original HLL approximate aggregator. However, the memory consumption is dynamic due to the variable size of the lists.

Despite the fact memory consumption dynamically changes during runtime, this is a valid approach to test our Hypothesis due to the worst-case scenario being a logarithmic space complexity.

However, in Section 3.4.1.2 we analyze an implementation that is not dynamic in memory during runtime making it a preferred approach.

3.4.1.2 Sliding HyperLogLog using Distance Recorder Vectors

In Section 3.4.1.1 we have analyzed a Sliding HLL implementation where each of the m buckets holds a list of values (LFPM) instead of a single integer count. However, the LFPM structure is dynamic and its size varies at runtime.

The approach proposed by Xu uses a smaller amount of memory that does not change at runtime [55]. The author proposes a special counter: a *Distance Recorder Vector (DRV)*.

Data Structure

Much like the original HLL and the LFPM Sliding implementation, the Distance Recorder Vector (DRV) approach uses m buckets. However, instead of integer counts or lists, each of the buckets stores a DRV.

A DRV is composed by $\lceil ln(n/m) \rceil$ records, with n as the cardinality of the set. Notice that *n/m* is the average number of distinct elements each bucket m is expected to process and that a binary string of size $\lceil ln(n/m) \rceil$ can represent *n/m* distinct elements. Also recall that the bit-pattern observable p used by the HLL is the position of the leftmost 1-bit. In a binary string of size $\lceil ln(n/m) \rceil$, p will range from 1 to $\lceil ln(n/m) \rceil$. The DRV is indexed exactly this way, from 1 to $\lceil ln(n/m) \rceil$, and each bit-pattern observable p will be used as an index in this vector of records. Essentially, each p will have an associated record in a DRV of a bucket.

Initially, all records in the DRV will be set to be infinity or a very large number.

Element Insertion

When an element is inserted, the same procedure as in the original HLL is followed: the element is hashed and p is computed from the hashed binary representation. Let us denote the bit-pattern observable of the new element as p_{new}. For the DRV of that bucket, we increment the value of all records by one. Next, we set position p_{new} in the DRV to 0 — *i.e.*, $\text{DRV}[p_{new}] = 0$.

Sliding Window Cardinality Query

This method does not perform element eviction. Instead, the query algorithm assumes that some records in DVR's entries might be expired.

To estimate the sliding window cardinality is to find the value p. To find the value p for a bucket, we scan the DRV of that bucket backward. For that, we start at the last element — positioned at $\lceil ln(n/m) \rceil$ — and scan towards the head of the vector. For each record, we check if it is active in the current window. To do so, we check if its value is less than the size of the window. If it is, the record is active. Otherwise, the record is inactive.

If the record is inactive we continue scanning backward. If the record is active, then the position or index currently being scanned is the value of p for that bucket. If all records are inactive, the bit-pattern observable p is 0.

Stochastic averaging is done just like in the original HLL: compute the harmonic mean of all p_m values from all m buckets. The cardinality of the sliding window is then estimated to be 2^p.

Time complexity analysis

When inserting an element, determining its bucket, hashing the element and computing p is done in constant time. However, all the $\lceil ln(n/m) \rceil$ records must have their value increased by one and this means iterating over the DRV. Setting the element of the DRV indexed by p to 0 is done in constant time as well.

Estimating cardinality is done by scanning the DRV backward until an active record is found. In the worst-case scenario, this means scanning all $\lceil ln(n/m) \rceil$ records.

Both inserting an element and estimating sliding window cardinality require a scan through $\lceil ln(n/m) \rceil$ records. This gives us a time complexity of *O(ln(n))*, if the number of buckets m is constant.

Space complexity analysis

This approach still uses m buckets but each one contains a Distance Recorder Vector (DRV). A DRV is a vector of $\lceil ln(n/m) \rceil$ elements. Hence, $m\lceil ln(n/m) \rceil$ elements must be kept in memory, giving us a space complexity of *O(ln(n))* with n as the cardinality of the set.

Applicability to our Hypothesis

Despite having the same time and space complexities, the Sliding HLL DRV approach uses a smaller amount of memory that does not change during runtime, while the LFPM requires heap resizing because the lists dynamically shrink and grow. Hence, the DRV approach is a better fit to integrate our lightweight real-time solution, if we need to perform cardinality estimation.

3.5 Summary

In this chapter, we presented state of the art outlier detection methods and classified them according to Blázquez-García *et al.* taxonomy [26]. Additionally, we presented sliding window aggregation algorithms, probabilistic data structures and their respective sliding window implementations, with the intent of researching how to incrementally maintain an aggregation in a streaming environment.

In Section 3.1, we reviewed outlier detection algorithms. However, we found most of them inapplicable in our scenario, because they were resource-intensive (both time and memory-wise), provided no measure of how divergent an outlier was or because they evolved their reference period.

In Section 3.2, we presented sliding window aggregation algorithms that we intended on using to compute our aggregations. However, their time or memory complexities did not meet our sub-linear requirements.

In Section 3.3, we discussed probabilistic data structures. Then, in Section 3.4, we explored some probabilistic data structure sliding window implementations. We explored only sliding window versions of the HyperLogLog, but sliding versions of other probabilistic data structures exist, such as the Sliding Bloom Filter [56] and the Sliding Count-Min Sketch [57]. Despite being efficient, both time and memory-wise, each probabilistic data structure only allows for one type of aggregation, and none of them happened to be of particular use to us.

In the next chapter, we formally define the problem we aim to solve and the issues with the current state of the art solutions. Then, we formulate our proposal along with the assumptions we make when approaching this problem. Finally, we pose some research questions we aim to answer by the end of this thesis.

Chapter 4

Problem Statement

With this chapter, we further discuss the problem we aim to solve with this thesis making use of concepts discussed in Chapter 2 and explaining why none of the reviewed methods from Chapter 3 works for our use case.

4.1 The Problem

Many real-time stream monitoring systems are static once deployed in a production environment. These systems are configured *a priori* and make assumptions about future streaming data based on the observable past events. The problem is that if these assumptions no longer hold in the future, the systems' performance decays. Thus, the problem at hand is to determine when to reconfigure the system based on an analysis of the drifts in the stream of data. For example, consider a Machine Learning (ML) model that monitors a data stream of credit card transactions and detects fraud. The ML model trains on a reference period and learns from this training period, *i.e.*, makes assumptions based on the analyzed data. In streaming, throughout time, the data patterns will change and the assumptions made will no longer hold (*e.g.*, buying patterns change on holidays), which leads to a decay in model performance. Our objective is to alert data pattern shifts. Our motivation is doing it so that system administrators can reconfigure the system before its performance decays. In this example, system reconfiguration is tantamount to model retraining.

4.2 State of the Art Issues

Aggregations, specifically sliding window ones, were introduced in Chapter 2 along with several windowing techniques required to monitor streaming data patterns in real-time. In Chapter 3 we presented state of the art sliding window aggregation algorithms as well as outlier detection methods. However, the reviewed methods were deemed unfit for our use case because none of them simultaneously showed all of the desired characteristics, namely:

- **Low memory consumption:** need for a low memory footprint system that avoids linear growth relative to the sliding window size used;
- **Low time complexity:** need for a time-efficient method that updates the sliding window aggregation and raises alerts in constant time;
- **Fixed reference period:** the reference window must be kept the same throughout runtime;
- **Sliding window maintainability:** possibility to evict old events and insert new ones while updating the aggregation state;
- **Explainable alerts:** have an interpretable measure of divergence to understand why an alert was raised.

4.3 Proposal

We devise a two-phased and two-windowed method: the first phase is a batch analysis that makes use of a reference window; and the second phase is a streaming analysis that makes use of a target sliding window. The batch analysis main goals are to compute the reference aggregation for the reference window and the divergence measures distribution. The stream analysis main goals are to incrementally maintain the same type of aggregation but over the streaming data and alert when the difference between the reference aggregation and the target sliding one is high.

We choose to use a histogram aggregation with constant memory usage to encode the distribution of the observed values. Furthermore, we devise a sliding window approximate histogram aggregation that is maintainable in constant time in a sliding window fashion, through the use of Exponential Moving Averages (EMAs). Hence, our EMA-like histogram aggregation is ideal for the streaming scenario because of its constant time and space complexity, for ever-growing volumes of data.

Given a user-defined reference period and a stream of data to process, we monitor each event's fields and alert if they are considered divergent relative to their reference aggregation. We provide a measure of how divergent each field is. Based on this information, we believe that a system administrator would have an easier time determining when to reconfigure the system.

4.4 Assumptions

Our feature monitoring method assumes only that the monitored features are numerical or categorical and that the events arrive in an orderly fashion during streaming. If the latter is not true, our method will still work but results may be imprecise, as our sliding window histogram aggregation relies on Exponential Moving Averages (EMAs), which give different weights to each event according to its age.

4.5 Research Questions

The goal of this thesis is to develop a lightweight stream monitoring system that detects changes in features distribution between a reference period and a sliding window in real-time. To that end, we devise a series of research questions to be answered by the end of this thesis:

(RQ1) What aggregation can we use to properly encode data distributions?

We need an aggregation to reduce the entire window contents to a lighter representation and we need one that encodes the distribution of feature values correctly so we can measure the difference between the reference aggregation and the target sliding window aggregation accurately.

(RQ2) How can we maintain such an aggregation in a sliding window fashion in real-time with constant memory usage?

We need an aggregation that shows constant space complexity so that we can scale the size of our windows as we wish without increasing memory usage. Additionally, we need to find a way of incrementally maintaining this aggregation in real-time.

(RQ3) How can we measure divergence between the reference and sliding window aggregations?

After we aggregate the contents of the reference and target sliding windows, we need a function that takes both aggregation states and returns a measure of divergence.

(RQ4) Based on a divergence measure, when should we raise an alert?

Hard-coding a threshold value for this divergence measure is cumbersome, impractical and error-prone. In other words, we need to find a way of automatically defining the divergence threshold that we use to raise alerts.

4.6 Summary

In Section 4.1, we formally defined the problem we aim to solve with this thesis: alert streaming data pattern shifts in regards to a static reference period with the intent of helping system administrators reconfigure their systems before their performance declines due to data pattern shifts.

Furthermore, we make explicit we work under time and space constraints, inherent to the streaming paradigm and to the non-critical nature of our monitoring system.

In Section 4.2, we summarize the main identified issues regarding state of the art solutions and in Section 4.3 we detail how we propose to implement our solution along with the assumptions made (Section 4.4).

In Section 1.4, we defined our hypothesis or main question to address as to whether sliding window aggregations and outlier detection methods combined could be used to detect data pattern shifts in data streaming scenarios in real-time and with a low memory footprint. To better discuss our hypothesis, we pose a series of research questions in Section 4.5.

In Chapter 5, we discuss initial approaches we thought of and reasons to why we discarded them. Finally, we present and detail each phase and component of our stream monitoring method.

Chapter 5

Lightweight Real-Time Feature Monitoring

5.1 State of the Art Outlier Detection Methods

Our first theoretical approach was to use one of the outlier detection methods presented in Section 3.1. For that, we reviewed subsequence and point outlier detection methods. Using point outlier detection methods, the main idea is to keep track of the rate of point outliers detected. For instance, keep track of how many outliers were alerted in the past hour. The decision to alert for a data pattern shift of the system would be made when comparing this rate of point outliers detected in the past hour with a user-defined threshold. On the other hand, using subsequence outlier detection methods would allow us to report the whole target sliding window as an outlier when compared to the reference window.

Challenges

The analyzed outlier detection algorithms in Section 3.1 were deemed unfit for our use case due to one or more of the following: *(a)* high memory consumption, *(b)* high time complexity, *(c)* "memory loss" and *(d)* obfuscated insights.

For this thesis, we need constant time complexity to process events one by one in real-time. Because our system is not mission-critical, we also want the solution to be as lightweight as possible, which means techniques that grow linearly with window size are not a good fit. Some of the methods discussed also suffer from what we call *"memory loss"*: they lose track of the reference window or normality period. In other words, the algorithms are online learners, meaning

they adapt to the underlying statistics of the data stream. While in some scenarios this is a much-needed feature, in our use case it is not. How clear the alerts and insights retrieved from the methods are, is also an important property. Some outlier detection methods use Machine Learning (ML) techniques to produce alerts, which makes it harder to explain to a system administrator why an alert was raised and what changed.

Conclusion

The outlier detection methods reviewed in Section 3.1 were discarded as viable solutions to our problem because of the aforementioned problems of memory and/or time complexity, *"memory loss"* and non-explainable alerts.

5.2 State of the Art Aggregation Algorithms

Dissimilarity-based univariate subsequence outlier detection methods as presented in Section 3.1.2 focus on measuring the distance between two periods of time and report an outlier when the distance is above a certain threshold. Keeping both reference and target windows in memory means our memory consumption grows linearly regarding both windows size. In an attempt to solve this, we focus on storing only an aggregation of both windows.

In Section 3.2 we reviewed algorithms to compute sliding window aggregations and in Sections 3.3 and 3.4 we discussed special aggregators and respective sliding window implementations.

In our dissimilarity-based approach, we compute sliding window aggregations and compare the aggregations' current state versus an initial reference, reporting a shift when the difference is substantial. For instance, consider the case where we monitor one single feature or variable x. Assume we define the set of aggregations to compute the mean of x and its standard deviation. Also assume we save the reference window and maintain a target sliding window and the associated average and standard deviation aggregations for x efficiently. As seen in Figure 5.1, the initial mean and standard deviation values for x presented in the reference window are of 5 and 1, respectively. Later on, in *Window 1*, we see the same values for the mean and standard deviation, of 5 and 1, respectively. This represents a *normal* case where no alert is produced.

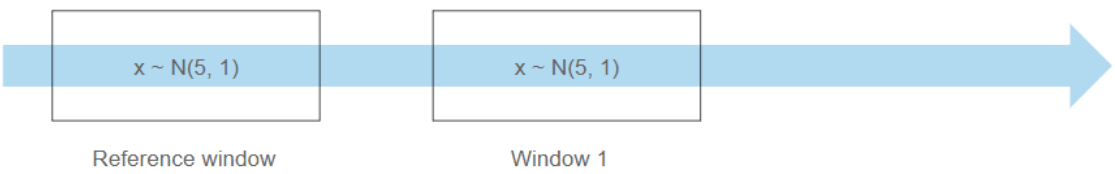


Figure 5.1: Normal state for feature x in window 1

In Figure 5.2, after further sliding steps of our target sliding window, we obtain *Window 2*. In *Window 2*, the mean and standard deviation aggregation values change from 5 and 1 to 12 and 6, respectively. In this case, we would measure the difference between the target sliding window

aggregations and the reference window ones and if above a certain threshold raise an alert. For example, if we used a threshold of $t = 6$, we would raise an alert, as at least one sliding aggregation, in this case the mean, would differ of at least t when compared to the reference value.

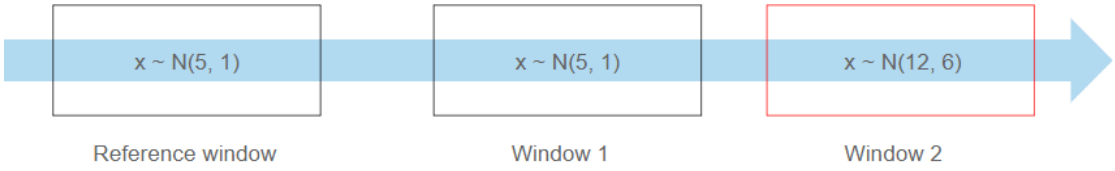


Figure 5.2: Alert state for feature x in window 2

Challenges

The first issue with this approach is that generic sliding window aggregation algorithms grow linearly in space regarding window size, such as Recalculate-From-Scratch and Subtract-On-Evict from Section 2.3.2, Two-Stacks from Section 3.2.1 and DABA in Section 3.2.2 which does not meet our sub-linear memory growth requirement.

Exponential Moving Averages (Section 2.4), Probabilistic Data Structures (Section 3.3) and their sliding window implementations (Section 3.4) can be used to solve this first issue and reduce the linear memory complexity of the system relative to sliding window size.

Another challenge when using this approach is defining the set of aggregations to compute at feature/variable level that represent the system state well enough for comparison of reference and streaming periods and change detection. To illustrate this problem, consider both time-series show in Figure 5.3. Time-series 1 and 2 have the same maximum, minimum, mean and standard deviation values for the time-based window between t_1 and t_8. Hence, if our set of aggregations chosen were the maximum, minimum, mean and standard deviation, we would not differentiate between these two different time-series.

Conclusion

This approach requires efficient sliding window aggregation state maintenance, which can be done using sliding window probabilistic data structures and/or exponential moving averages. However, the need to choose a set of aggregations to use and the deviation threshold itself ultimately led us to discard this approach.

5.3 Method: Feature Distribution Monitoring

In Section 2.1, we defined a data stream as a continuous collection of events. Each event is timestamped and contains multiple fields. In the context of this thesis, each of the event's fields are named features. Our goal is to detect data pattern shifts for each feature in a streaming fashion.

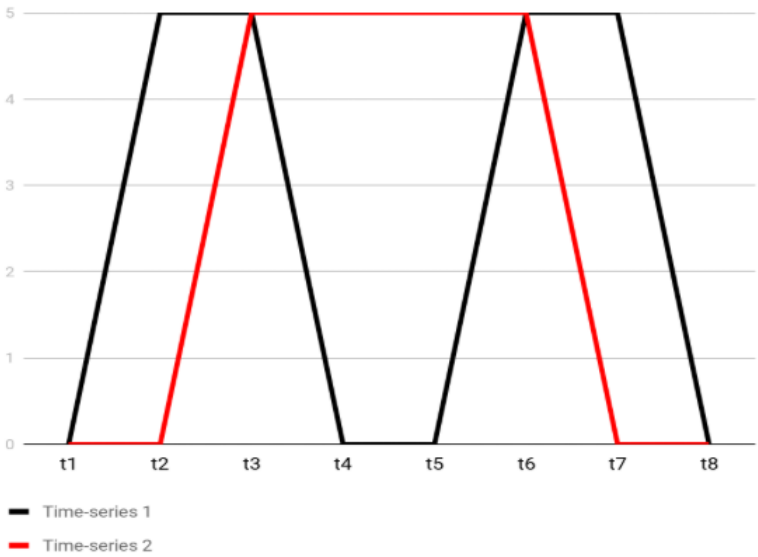


Figure 5.3: Two different time-series with same max, min, mean and standard deviation

Such shifts will be measured between a static in time or reference period versus the observed reality during the streaming phase. Figure 5.4 shows the separated reference and streaming timelines.

Figure 5.4: Reference and streaming timelines

To detect data pattern shifts, we do not need to directly compare the entire window contents of the reference and sliding windows. This approach would require storing all window contents hence making the memory consumption grow linearly with target sliding window and/or reference period size. Instead, we want to use a *(a)* fixed cost in-memory aggregation that can be *(b)* incrementally maintained during the streaming phase. Additionally, we want a streaming aggregation that *(c)* reliably encodes the distribution of each feature.

One aggregation that encodes distributions properly are histograms. Histograms are a representation of the distribution of numerical data thus being ideal to encode the distribution of feature values, satisfying *(c)*. Histograms require storing a fixed-size list of bins and corresponding counts per bin regardless of the volume of data, satisfying property *(a)*. Adding or removing an element to the histogram aggregation is tantamount to determining the correct bin and incrementing or decrementing the associated count, respectively, thus satisfying property *(b)*. Recall Section 2.3.2 where we discussed generic sliding window aggregation algorithms and remember the Subtract-On-Evict (SOE) algorithm (Algorithm 2). Computing a histogram aggregation over a sliding window of size

W using SOE implies storing all W elements in memory to know the element to remove after a new one is inserted. This violates our constant in-memory property *(a)* which is why we chose to use the Exponential Moving Averages (EMAs) analyzed in Section 2.4. EMA-like histograms computed over sliding windows do not require the storage of window contents. An EMA-like histogram is *(a)* constant in memory, *(b)* maintained in real-time and *(c)* encodes the distribution of each feature values. These EMA-like histograms are further explained in Section 5.3.1.

Since the reference window is static, *i.e.*, it is not a sliding window, we can use exact histogram aggregations. Hence, for each feature, we build an exact reference histogram aggregation. On the other hand, in streaming, we use one EMA-like approximate histogram aggregation per feature (Figure 5.5).

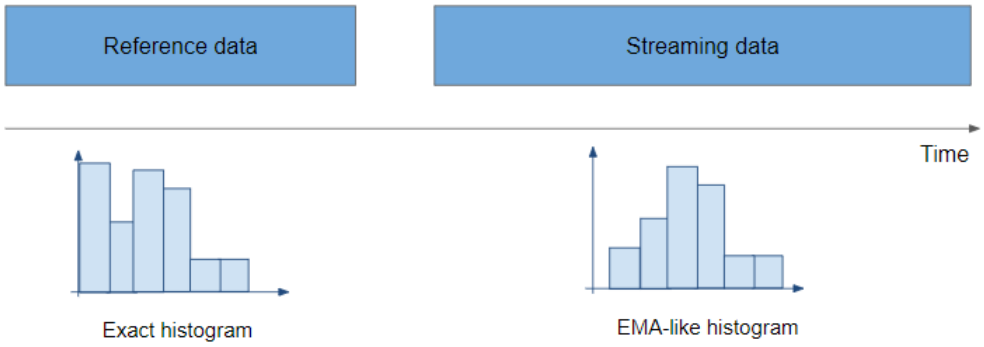


Figure 5.5: Reference and Streaming timelines and corresponding histogram aggregations

To compare each feature's reference and target sliding window histogram aggregation we need a measure of divergence. We chose to use the Jensen–Shannon Divergence (JSD) [45] but other statistical tests are possible such as Kolmogorov-Smirnov or Wasserstein [58]. The JSD takes as input two histograms and outputs a value between 0 and 1. The closer this value is to 0 the more similar the distributions are. On the other hand, a JSD value of 1 represents two different distributions. We chose to use the JSD metric due to its simplicity and because it was used in previous work successfully [41], leaving as future work other possible tests.

Given a divergence or distance value for the two aggregations (reference and target) we need a measure of normal statistical fluctuations. Since we consider the reference period as the normal state we can find the normal distribution for the divergence by sampling smaller windows of data from the reference period in a batch analysis. This allows us to define a threshold based on percentiles, which gives an estimate for the probability of a certain value occurring, rather than using a constant hand-picked threshold for the divergence value (details in Section 5.3.2).

To implement the proposed methodology, we devised a two-phase method. The first phase relies on a batch analysis that, for each feature, builds a reference histogram and computes the distribution of divergence values (details presented in Section 5.3.2). The second phase (detailed in Section 5.3.3) focuses on efficiently updating the target sliding window histogram for each feature and periodically performing a divergence test between the static reference aggregation and

the target sliding one, alerting for changes if need be. The proposed method currently works only for numerical features, that is, only for event fields or variables that are of numeric type, but can be extended to other types.

5.3.1 Exponential Moving Average Histogram

As stated, we compute histogram aggregations for the reference and target sliding windows. The computation of the reference histogram is done once in batch. Since the reference data is fixed and does not change over time, computing this histogram in batch is more efficient and easier to implement in a distributed system like Spark. In addition, a batch computation also allows us to compute an exact reference histogram. However, the target sliding window histogram is an EMA-like histogram to ensure that we are able to encode the feature distribution with constant space complexity and maintain it in real-time.

In Section 2.4 we discussed Exponential Moving Averages (EMAs) and how they compute sliding window aggregations without actually storing the window contents. Definition 3 shows how to compute an EMA using its recursive formula. An EMA state or value depends only on the previous one multiplied by an exponential decay. Recall that a tuple-based decay is given by:

$$2^{-\frac{1}{n_{1/2}}}$$

where $n_{1/2}$ is commonly denoted as a tuple-based half-life.

Algorithm 6 shows how we update our EMA histogram. We assume we have access to global *counts* and *bins* arrays and use a suppression factor of $2^{-\frac{1}{n_{1/2}}}$ with $n_{1/2}$ as the tuple-based half-life. The histogram aggregation is updated for each new streaming value `val` using the `processValue(val)` function, that increments the correct histogram bin (note that the *bins* array defines the bin boundaries while the *counts* array actually holds each bin's count) and applies the EMA suppression to every bin.

Algorithm 6 EMA histogram sliding window emulation

```
1: function GETBIN(val)
2:     return binarySearch(bins, val)
3: function PROCESSVALUE(val)
4:     bin ← getBin(val)
5:     counts[bin] ← counts[bin] + 1                ▷ // Increment correct bin
6:     for each count ∈ counts do                   ▷ // Apply EMA suppression to all bins
7:         count ← count × 2^(−1/n_{1/2})
```

After processing x tuples the oldest tuple is *smoothed* by a factor of $2^{-\frac{x}{n_{1/2}}}$. If we process $x = 4n_{1/2}$ tuples the oldest tuple would have its contribution reduced by

$$2^{-\frac{4n_{1/2}}{n_{1/2}}} = 2^{-4}$$

which is approximately 0.06. In other words, the *x-th* event would have 6% contribution towards the EMA state or value meaning older events would have even less weight.

We choose to emulate windows with four times the half-life size (or $4n_{1/2}$), meaning we discard events whose contribution is less than 6%. We set the half-life to be one-quarter of the size of the window we want to emulate. The size of the window we want to monitor varies from situation to situation, but in the limited scope of this thesis we set to monitor a period extending up to one month (at an error level of about 6%) so we set the half-life to be the number of tuples that correspond to one week worth of data. This will give more weight to the most recent week but it will include contributions decaying to 6% in the fourth oldest week.

5.3.2 Batch Analysis Phase

The goal of the batch phase will be to build our reference aggregation for each feature and get to know the distribution of divergence or distance values between the reference and EMA-like aggregations. In other words, we:

1. Obtain a reference histogram for each feature using all data in the reference period
2. Compute the distribution of distances between the reference histogram and random windows of data and corresponding histograms

This distribution of values can then later be compared to a given value of divergence for a single sample to find out how probable that value is to occur (if it is very low an alert may be raised).

Building the Reference Histogram for each Feature

First, for each feature, we compute the reference histogram from the reference period dataset. This is an exact histogram aggregation. When building it we ensure the histogram has equal counts (or equal height) for all bins which often results in different sized bins. We make no assumption on the dataset distributions and hence use equal height histograms because they adjust better to wildly varying ones. Equal height means we have more bins covering very dense regions (regions with many data points) and fewer bins in lower density regions.

Finding the Distribution of Distances for each Feature

Secondly, we aim to build the histogram that encodes the distribution of expected distance values, to later on threshold the observed distance values during the online phase. To that end, we make S samples of transactions (in Section 5.3.4 we discuss the minimum number S of samples to make), each with the same tuple-based size of the target sliding window we will use in streaming. Each sample is a contiguous block of transactions, thus preserving the order of transactions and the time-dependency property of a time-series.

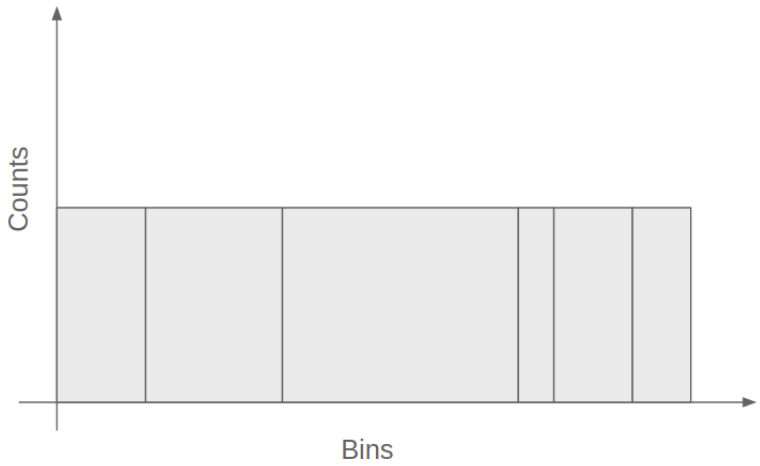


Figure 5.6: Reference equal-height histogram

For each sample, and each feature, we compute the approximated histogram using the bins computed for the reference histogram (Figure 5.7). We also add two extra bins that will cover the region to the left and the right of our histogram, respectively. In other words, we add a bin that covers the region between $-\infty$ and our first bin and another that covers values from the last bin to $+\infty$. This way, observed values outside of the reference period will be placed in these special bins, either left or right.

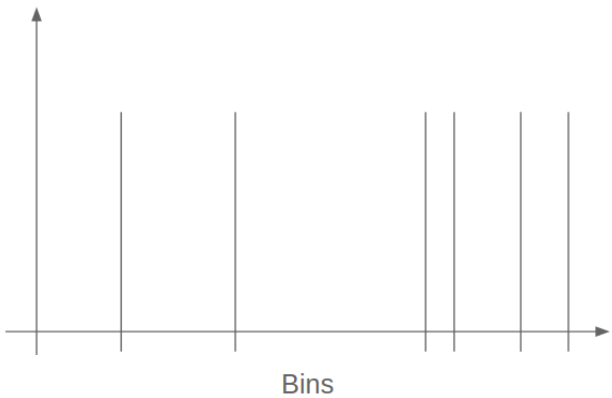


Figure 5.7: Reference equal-height histogram

Each of the histograms is an EMA-like histogram. An EMA-like histogram is essentially a collection of EMA-counts, as described in Section 5.3.1. Each sample is processed event by event and a discount factor is applied per event to each bin (EMA expiration factor), thus building an approximated histogram aggregation, using the reference bins, for each feature of that sample. This procedure is illustrated in Figure 5.8. We compute multiple sample histograms to encode distributions over smaller time periods that may exist within the large reference period. We use an approximated histogram to mimic the target approximated histogram we will incrementally

Figure 5.8: Reference equal-height histogram

maintain in the streaming environment.

For each feature, we have now S histograms, one for each sample. For each of the S histograms, we compute the distance between it and the reference histogram. Figure 5.9 illustrates this process as we apply our distance function for each sample histogram and the reference histogram, obtaining S distance measurements.

For each feature, we end up with S distance values. These S measurements are distance measurements between random samples of data and the reference period. Hence, we claim we now know the distribution of expected distance values. Given a new distance value measured between another random window and the reference window, we can compute its probability and produce alerts if the probability is below a certain threshold. Note that the histograms for each sample are EMA-like histograms (as detailed in Section 5.3.1) just like the target sliding window histogram. This ensures a certain degree of fidelity in the test we make in streaming because we measure the divergence between a random sample of data (our streaming histogram or any of the sample's histogram) and the reference one.

It is important to understand why we want to know the distribution of divergence values instead of just thresholding the divergence value itself. For instance, given that JSD values vary between 0 and 1, why not just set to alert if the measured JSD value is 0.7? The reason is that a divergence value may fall on different percentiles for different features. In other words, the same *i-th* percentile will correspond to a different divergence value for different features. Consider

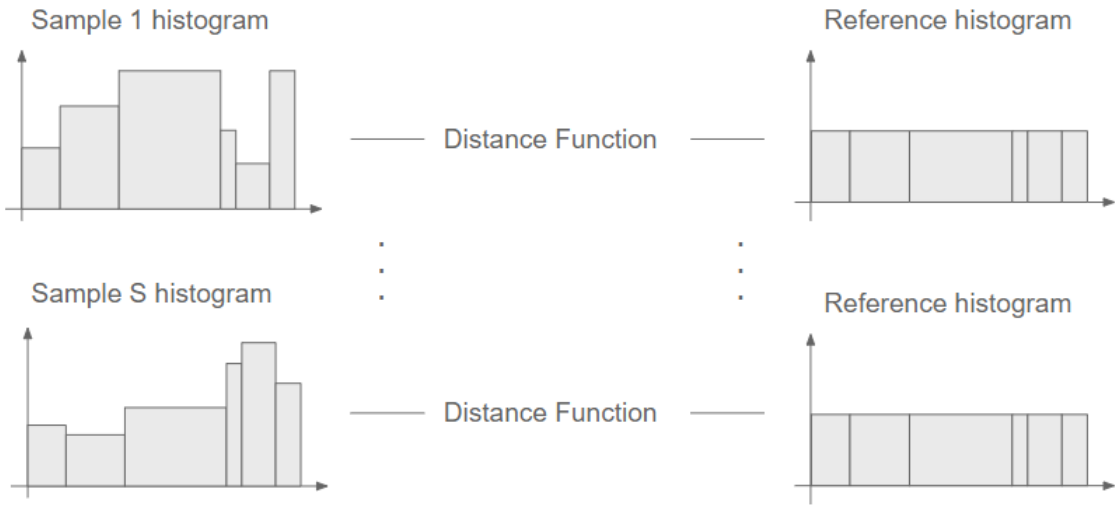


Figure 5.9: Compute as many distance values as samples, one for each sample-reference histogram pair

the following left and right-skewed distance distributions represented by Figures 5.10 and 5.11, respectively.

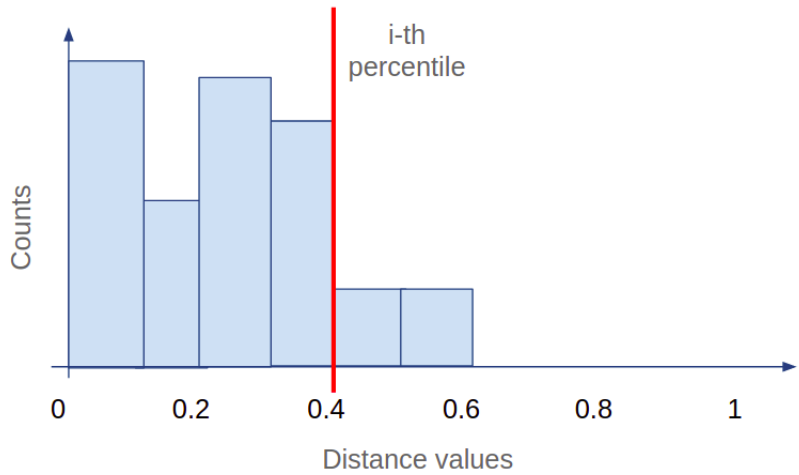


Figure 5.10: Left skewed distribution of distance values

Let's assume that the histogram from Figure 5.10 encodes the distribution of expected distance values for a feature *x1* and Figure 5.11 does the same but for a feature *x2*. Setting a user-defined threshold of α=*0.6* for both features would yield very different results. For feature *x1* we would be reporting distance values above the 100th-percentile which would not be the case for feature *x2*. Instead, we define our threshold as a percentile of the distribution and not as a hard-coded distance constant. For instance, in this case, we would define the threshold to be the *i*-th percentile, which would correspond to a threshold value of 0.4 for feature *x1* and 0.6 for feature *x2*.

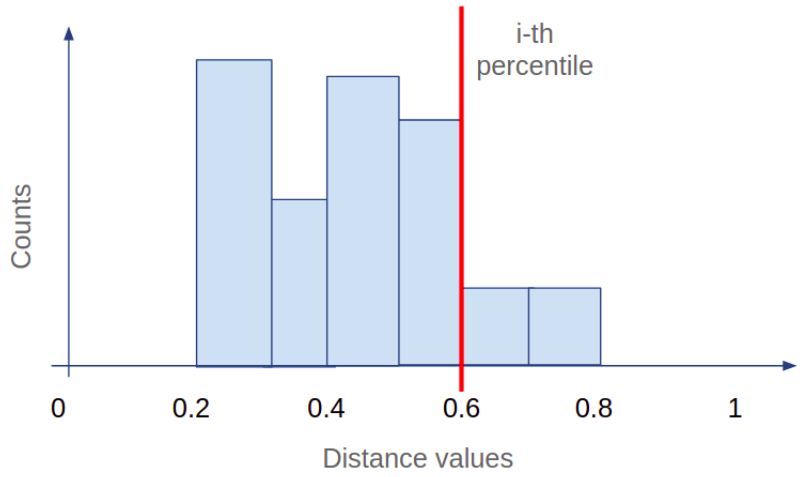


Figure 5.11: Right skewed distribution of distance values

Hard-coding a threshold value to be used for all features is cumbersome, impractical and error-prone. The distance threshold must be computed based on the distribution of distances.

Burn-In Period at System Initialization

When we boot up our system in the streaming phase we initially have empty histogram aggregations. The period of time where you discard the alerts produced by the system until it processes enough data to produce accurate reports is commonly denoted as the burn-in period. To avoid this burn-in period we propose that the target sliding window approximate histogram aggregation for each feature is initialized using the last sample's histogram for that feature.

Batch Phase Artifacts

By the end of the batch phase, we should have, for each feature:

- a reference histogram
- a list of distance values that represent its distribution
- the last sample's histogram, to be used as burn-in period for the target sliding window histogram

5.3.3 Streaming Phase

In the streaming or online phase, the system will process event by event in a true sliding window fashion (recall Section 2.2) and perform a change detection test periodically. This phase makes use of all of the batch phase artifacts, namely, for each feature, the reference histogram, the distribution of distance values and the last sample's histogram.

For each feature, we initialize the approximate histogram with the last sample's data, from the batch phase. Then, for each incoming event and each of the event's features, we update our streaming approximate histogram for that feature by adding the new value and then "cutting a slice" from the top of the histogram bars to simulate eviction, as described in Section 5.3.1. This is all we do, regarding event processing.

In order to obtain alerts, we periodically perform a multi-feature change detection test. The test is done by first computing the distance between the reference histogram for a feature x and the target sliding window histogram of the same feature. This distance value is then tested against the known distribution of distance values for feature x. In other words, we test to see if the percentile of the computed distance is in the distribution of distance values and alert if above a certain percentile, *e.g.*, the 99th-percentile. When we perform the test, for each feature x, we compute the percentile of the distance measured between the reference and target sliding window histogram aggregations. The probability value or *p-value* of a distance value is:

$$p\text{-}value_{distance_x} = 1 - percentile(distance_x)$$

For example, if we define alerts to have 1% or less probability to happen, it means the system will raise an alert for distance values above the 99th-percentile or *p-value* = 0.01. Generally speaking, we alert a feature x as divergent at a given timestamp if for a given probability threshold α:

$$percentile(distance_x) > 1 - \alpha$$

or:

$$p\text{-}value_{distance_x} < \alpha \tag{5.1}$$

Multiple Test Correction

Note that we periodically apply this test (Equation 5.1) for each feature, effectively repeating the analysis. When considering several hypotheses, the problem of multiplicity arises: the more hypotheses are checked, the higher the probability of obtaining false positives. In other words, when repeating a test multiple times the multiple test or simultaneous statistical inference problem [59, 60, 61, 62] arises which states that the more inferences are made, the more likely erroneous inferences are to occur. In the literature, many multiple test correction methods are proposed, but we opted to use the Holm-Bonferroni multiple test correction because it is a simple and easy to compute correction that can reduce the number of false negatives while having as target to control the number of false positives. Furthermore, it does not require to assume any independence between the tested hypothesis (as other methods do) which would almost surely be a wrong assumption for realistic datasets. Therefore we leave as future work the testing of other correction methods. In statistics, Family-Wise Error Rate (FWER) [63, 64] is the probability of having one or more false positives when performing multiple hypotheses tests, *i.e.*, considering one hypothesis true when it is not. The Holm-Bonferroni method [65] is one of many approaches for controlling the Family-Wise Error Rate *(FWER)* by adjusting the rejection criteria for each hypothesis.

Holm-Bonferroni Correction

The Holm-Bonferroni correction takes Equation 5.1 and corrects the right-side threshold α for each feature. This correction is done by dividing the threshold α by a correction factor, as described next.

First, we compute the *p-value* of each feature. We then order our features by *p-value*, in ascending order. Next, starting from the first feature, the one with the smallest *p-value*, and proceeding until the end of the list of features, we check if:

$$p - value_{distance_x} < \alpha/(m-k) \tag{5.2}$$

with α as the probability to reject the null hypothesis, m as the number of hypotheses to test, in our case the number of features and corresponding *p-values* and k as the 0-based index of the current feature. We stop at the first feature that fails this test, and every feature processed up until this one is considered to be a divergent feature and alerts are raised.

5.3.4 Final Remarks

Minimum Number of Samples

In Section 5.3.2 we stated we made S samples in the batch phase. But what is the optimal number of samples to make? How can we mathematically derive a formula for the number of samples to make and avoid *ad hoc* user-defined constants?

When checking for alerts we are performing one test for each feature, hence we perform $n_{features}$ tests. We apply a multiple test correction on the tests performed and want to set the FWER to α. In other words, we want to alert deviations for the full set of tests globally that have probability $\alpha < 1\%$, for example.

We use the Holm-Bonferroni correction and, for each feature, we need to reliably estimate the upper tail of the distribution which corresponds to the α / $n_{features}$ region. Each sample we do translates into a distance value and the collection of these data points builds the distribution of distance values. We want to be sure that the number of samples S that we do generates enough distance points to have a high probability of having one or more points in the upper tail region. To that end, we define the probability of not having at least one point in the upper tail of the distance distribution as γ.

In [41], the authors prove that the minimum number of samples S needed to satisfy these probability requirements is:

$$S = \left\lceil \frac{\log \gamma}{log(1 - \frac{\alpha}{n_{features}})} \right\rceil \tag{5.3}$$

Method classification

Similarly to the methods presented in Sections 3.1.2.1 and 3.1.2.2, we use a two-windowed model that can take uni or multivariate input. However, we perform a multivariate analysis of all fea-

ture's time-series with the use of the Holm-Bonferroni correction. Hence, we say our method is multivariate. We also alert entire windows or subsequences as anomalous. Thus, according to the taxonomy presented in Section 3.1, our method is a dissimilarity-based multivariate subsequence outlier detection method.

Time complexity of the streaming phase

The critical phase of our method is the stream analysis because it has to process new events and alert for possible deviations in real-time.

For each event and for each feature, we update our EMA histogram using Algorithm 6 which has to perform a binary search to find the correct bin to insert our new value and traverses all bins to apply a suppression factor. Using n as the number of bins, we have a time complexity of $O(log(n)+n)$, for each feature. However, since n is a fixed and small constant, *i.e.*, more or less 100 bins, we say that this algorithm has constant time complexity. In other words, our time complexity does not depend on the size of the emulated streaming windows.

Space complexity of the streaming phase

We store n bins per feature, so we have a space complexity of $O(n)$ per feature. Again, since n is a small and fixed constant, we claim to have constant space complexity for ever-growing volumes of data.

Chapter 6

Method Validation

In this section, we provide summary statistics for our method's performance and draw visualizations for the obtained results to validate our alarms and reason about them.

6.1 Experiments with Synthetic Data

In order to test our proposed system, we generated multiple synthetic datasets before moving on to real data. In this section, we present summary statistics for the synthetic datasets created and our generated alarms.

6.1.1 Single-Feature Analysis

As an initial step, we started by validating our system using single-feature datasets before moving to multi-feature scenarios to facilitate system validation, considering the complexities of multi-feature datasets and the problem of multiple test correction. In this section, we validate our results for single-feature synthetic datasets.

6.1.1.1 Varying Sample Sizes

We began with single-feature scenarios and by experimenting with different tuple-based half-lives $n_{1/2}$. The half-life $n_{1/2}$ parameter in turn controls the size of the samples we make in the batch phase and the target sliding window size of the streaming phase.

In Section 5.3.1, we mentioned that we choose to discard events whose contribution is less than 6%. We set a half-life $n_{1/2}$ value and then multiply it by a constant factor of 4 to get the size of the samples to make (a larger sample would contain events whose contribution was equal to or less than 6%).

All the synthetic datasets used throughout this section are a time-series where at each time-step we sample uniformly from a fixed distribution, *i.e.*, there are no time correlations in the used datasets. The reference dataset R1 used in this first experiment contained a single feature that followed a normal distribution with mean $\mu = 10$ and standard deviation $\sigma = 2$ for the 5 million rows of the dataset. Figure 6.1 shows the time-series of the single feature in dataset R1. The dataset processed in the streaming phase (*i.e.*, the target dataset) was generated with a size of 5 million events and a single feature. For the first half, *i.e.*, for the first 2.5 million events, the feature followed a normal distribution with mean $\mu = 10$ and standard deviation $\sigma = 2$, like the reference dataset. For the other half, we changed the generating distribution to be a continuous uniform one with lower bound $a = 100$ and upper bound $b = 200$. Figure 6.2 shows the time-series of the single feature in the target dataset T1 where we see the abrupt change in feature values.

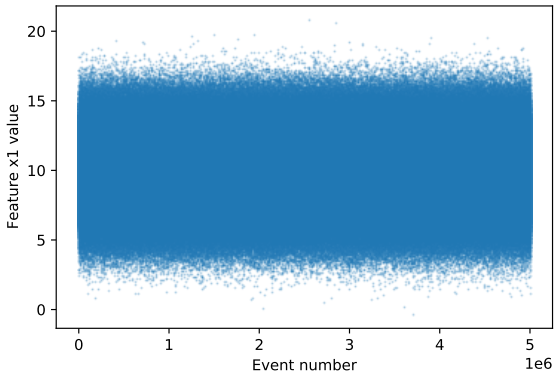


Figure 6.1: Time-series for dataset R1

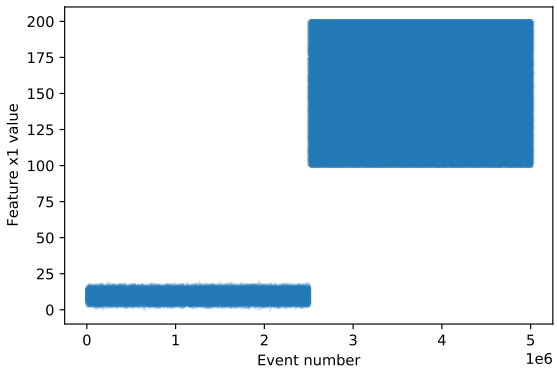


Figure 6.2: Time-series for dataset T1

We conducted three different experiments varying the tuple-based half-life $n_{1/2}$, using the reference dataset R1 and the target dataset T1. The experiment identifiers, the corresponding half-lives and the execution times are shown in Table 6.1. Notice that *SampleSize* corresponds to the size of each sample made in the batch phase and so it is four times the half-life. We used 100 bins (plus the two special infinity bins) for the reference histogram and made 1000 samples. Table 6.1

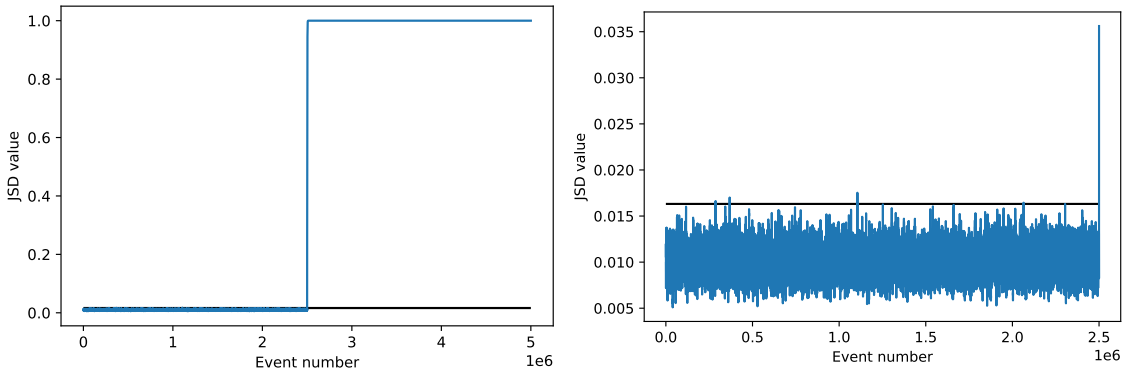
Experiment	Half-life	Sample Size	Executors	Batch execution time (seconds)
01	625	2500	300	152
02	62,500	250,000	300	2,822
03	250,000	1,000,000	300	10,082

Table 6.1: Experiments with varying half-lives

shows the execution time for the batch phase in seconds. All experiments used 300 Spark executor instances. Since our sample size depends on the half-life (four times larger), the larger the half-life, the larger the sample size. Larger samples imply longer processing times when computing the EMA-like histogram for each sample. With no surprise, we see that the larger the sample the longer it takes to finish the batch phase.

From a functional point of view, were we able to detect the abrupt change introduced in the target dataset T1? Recall that from the 2.5 millionth event forward we change the distribution from

a gaussian to an uniform one (Figure 6.2). Figure 6.3a shows the distance (JSD) values computed during streaming at each event of the 5 million events from dataset T1, using the outputs from the batch phase of experiment 01. We clearly see a change in JSD values, and we move from values close to 0 to values close to 1 (or even 1). Figure 6.3b shows the same signal but for the first half of the streaming period (essentially a zoom-in). We see our JSD values before the change were very low (magnitudes of 10^{-2}).



(a) For the whole time-series

(b) Zoom in on the first 2.5 million events

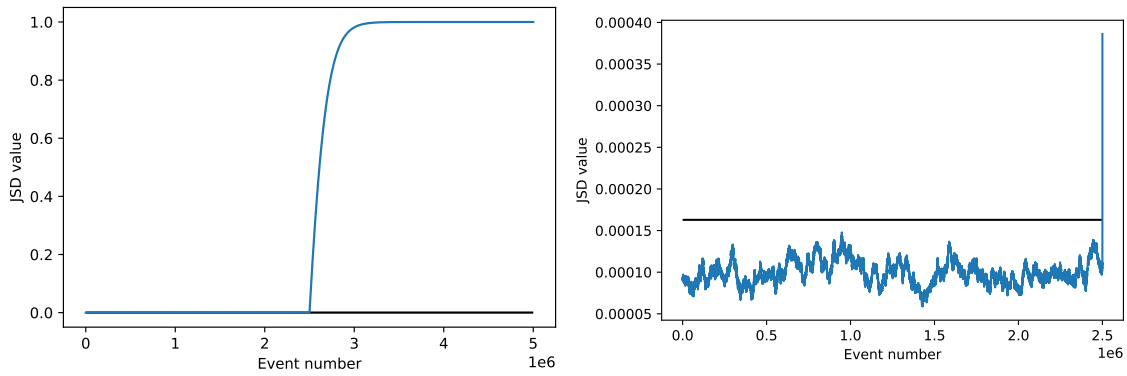
Figure 6.3: Experiment 01: JSD signal and threshold

However, the change in JSD value is very abrupt as well. Almost instantly we go from close to 0 values to 1. We also had some false-positives, *i.e.*, some JSD values above the threshold. This indicates to us that the window used is too small and sensitive to small changes, or that we can choose a more strict threshold.

In experiment 02, we use larger sampling windows. Remember our target sliding window in streaming is of the same size as the sampling windows. In the second experiment, we increase the size of the sampling and target windows by 100 times.

Figure 6.4a shows the JSD signal during streaming for dataset T1, using the outputs from the batch phase of experiment 02. Again, we see a change in signal from the middle of the streaming period forward. Figure 6.4b shows the same signal for the first half of the streaming period. Notice that this time the growth of our signal is slower. Notice also that we do not have false positives in the first half of the signal. This is because we use a larger window, which means each event has a smaller contribution to the aggregation state. Hence, outliers will have less impact on the aggregation and corresponding signal value.

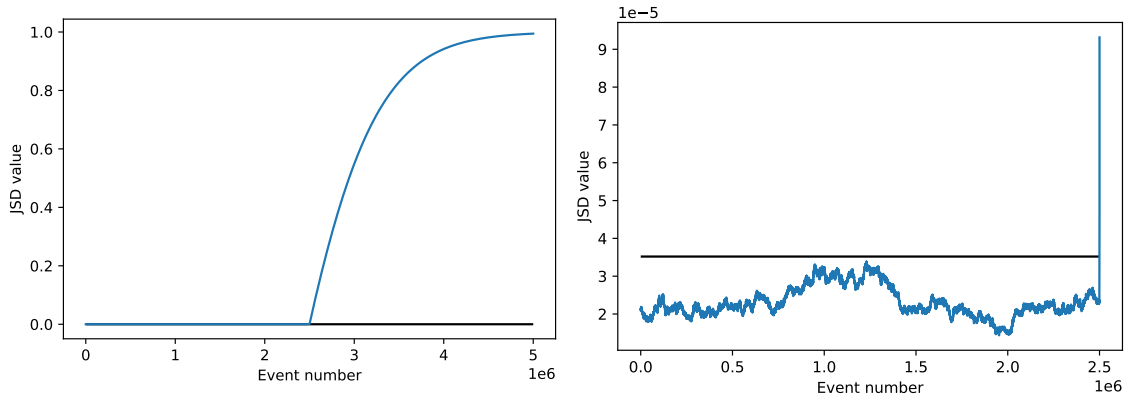
In experiment 03, we increase the sampling and target windows size once again, this time four times larger than in experiment 02. We would expect an even smoother growth of our signal and, again, no false positives. And indeed, judging by the corresponding signal computation and zoom-in, Figures 6.5a and 6.5b, respectively, the signal looks smoother and we see no false positives.



(a) For the whole time-series

(b) Zoom in on the first 2.5 million events

Figure 6.4: Experiment 02: JSD signal and threshold



(a) For the whole time-series

(b) Zoom in on the first 2.5 million events

Figure 6.5: Experiment 03: JSD signal and threshold

6.1.1.2 Multiple Distribution Changes

In this experiment, we wanted to see how our signal behaved when there were multiple change points, where each gaussian distribution was a scaled down version of the previous one. Figure 6.6 shows the time-series for the reference dataset R2 used in this experiment. It contained 5 million events with a single feature following a gaussian distribution with mean $\mu = 100$ and standard deviation $\sigma = 50$. The time-series for target dataset T2 is shown in Figure 6.7. Notice that we change distributions at 2, 3 and 4 millionth events. Notice also that each subsequent distribution produces values whose domain is a subset of the previous distribution.

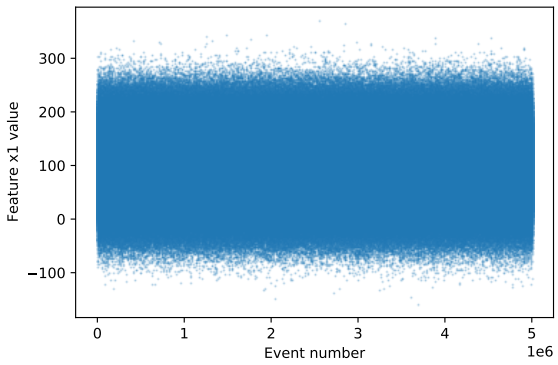


Figure 6.6: Time-series for dataset R2

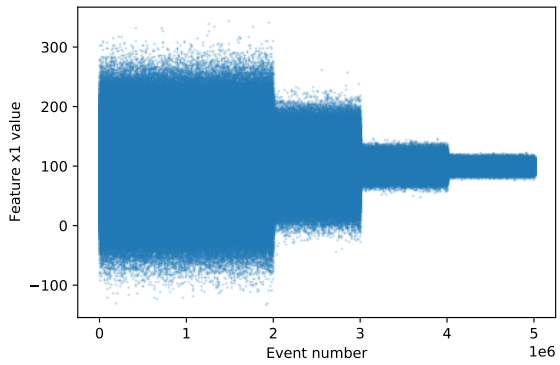
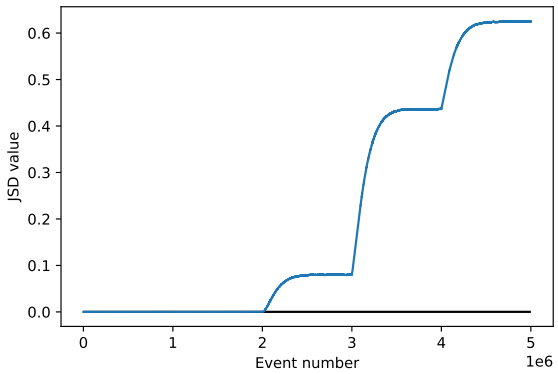


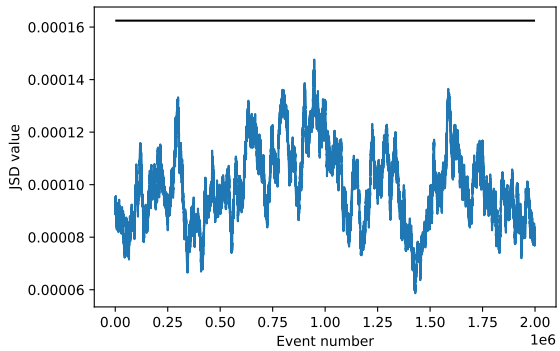
Figure 6.7: Time-series for dataset T2

The target dataset T2 begins with the same distribution as the reference dataset R2, a gaussian distribution with mean $\mu = 100$ and standard deviation $\sigma = 50$. At the 2 millionth event mark, we use another Gaussian distribution, this time with mean $\mu = 100$ and standard deviation $\sigma = 30$. At the 3 millionth event, we change the standard deviation to be $\sigma = 10$. From the 4 millionth event to the end, we use a standard deviation of $\sigma = 5$.

In this experiment, we used a half-life $n_{1/2} = 62500$ and corresponding sample size of 250,000 events, like in the previous section's experiment 02. Figure 6.8a shows the computed JSD signal for the target dataset and Figure 6.8b zooms in on the first 2 millionth events, before any distribution is changed. We observe that once again we have no false positives in the first 2 million events, which is ideal. Furthermore, we observe that the computed JSD signal varies accordingly with



(a) For the whole time-series



(b) Zoom in on the first 2 million events

Figure 6.8: JSD signal and threshold

the changes in distribution. Despite us changing the distributions to subsets of the previous ones to try and deceive our method, our signal accurately represents these changes. In Figure 6.8a, we see several growth points that correspond to the points where the generating gaussian distribution parameters changed. Hence, we are able to detect changes in distribution parameter changes and quantify how big the change is (the closer to 1 our signal is).

6.1.1.3 Multiple Distribution Changes, each a Superset of the Previous One

In this experiment, we wanted to do the reverse of the previous one. In this one, there are multiple change points but each new distribution spans a domain that is a superset of the previous one. Figure 6.9 shows the reference period used, with 5 million events and one feature following a gaussian distribution of mean $\mu = 100$ and standard deviation $\sigma = 10$. Figure 6.10 shows the target period used, also with 5 million events and one feature. Like in the previous experiment, we change distributions at the 2, 3 and 4 millionth event mark. On the other hand, unlike the previous experiment, we now make subsequent distributions cover the previous ones. We change the reference gaussian used with mean $\mu = 100$ and standard deviation $\sigma = 10$ at the 2, 3 and 4 millionth mark, to use standard deviations $\sigma = 30$, $\sigma = 40$ and $\sigma = 50$, respectively.

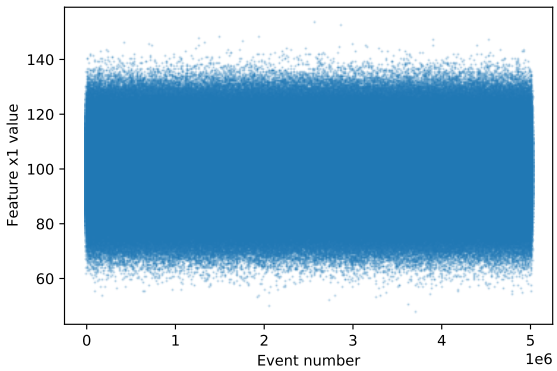


Figure 6.9: Time-series for dataset R3

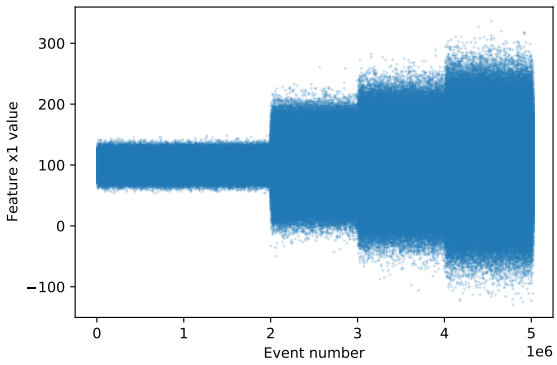
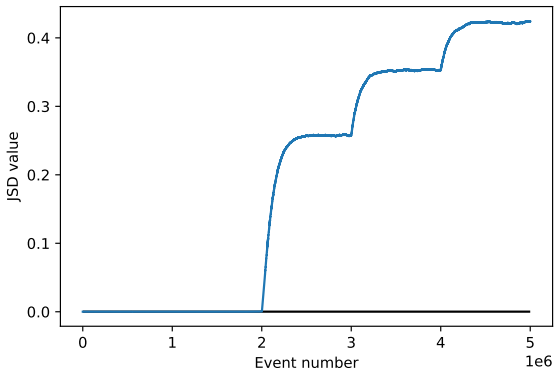
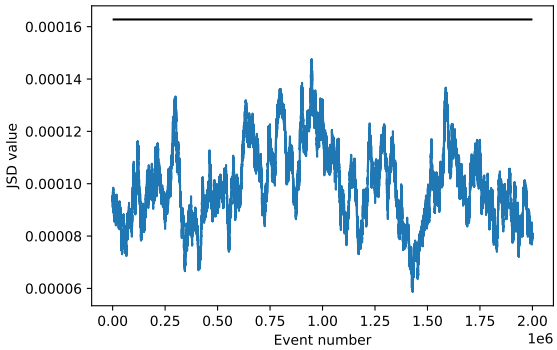


Figure 6.10: Time-series for dataset T3

Figure 6.11a shows the JSD signal for the target dataset in this experiment. Once again, we were able to detect the changes introduced at the 2, 3 and 4 millionth events. We are also able to identify bigger distribution changes, relative to the reference, by observing bigger JSD values. Figure 6.11b shows a zoomed in portion of the original signal, where we can verify that there are no false positives.



(a) For the whole time-series



(b) Zoom in on the first 2 million events

Figure 6.11: JSD signal and threshold

6.1.1.4 Return to Normality After Change

Our goal with this experiment was to check if our signal returned to its normal values (below the alert threshold) if we changed our distribution to the original reference one, as intended.

In this experiment, we used reference dataset R4 (Figure 6.12) and target dataset T4 (Figure 6.13).

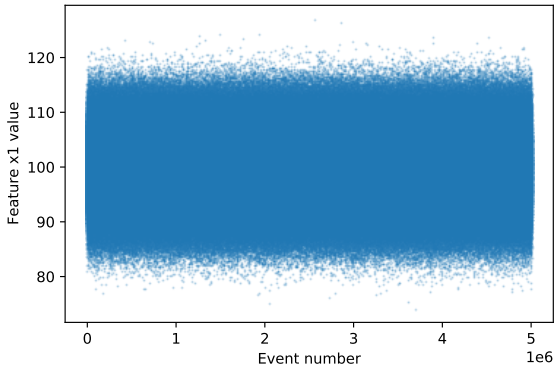


Figure 6.12: Time-series for dataset R4

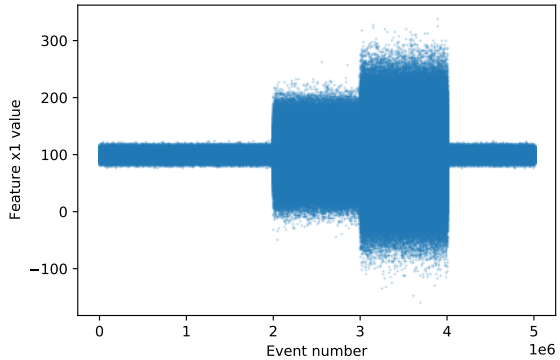


Figure 6.13: Time-series for dataset T4

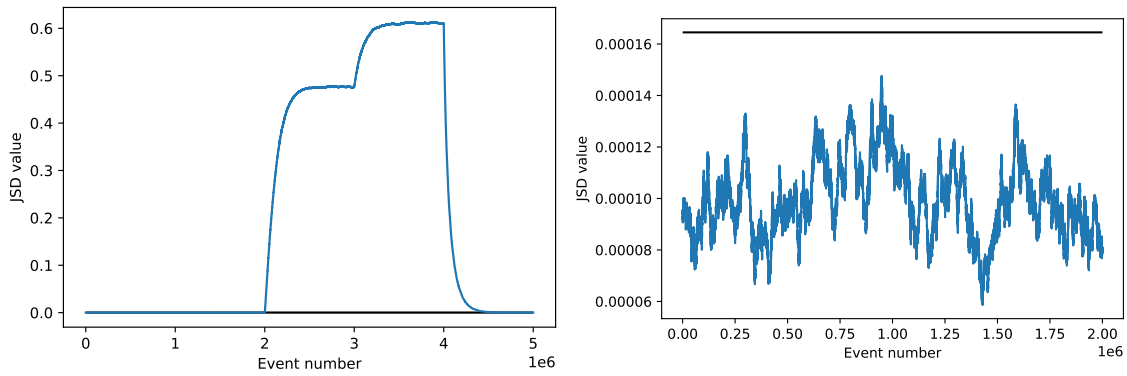
The reference dataset R4 had a single feature that followed a Gaussian distribution of $\mu = 100$ and $\sigma = 5$ for all 5 million events. Up until the 2 millionth event, the target dataset T4 followed a Gaussian distribution of $\mu = 100$ and $\sigma = 5$, similarly to the reference dataset R4. At the 2 millionth event, we change the Gaussian parameters to $\mu = 100$ and $\sigma = 30$. At the 3 millionth mark, we once again change the Gaussian parameters to $\mu = 100$ and $\sigma = 50$. Finally, on the 4 millionth event, we return to our reference distribution, a gaussian with $\mu = 100$ and $\sigma = 5$.

Figure 6.14a shows the computed JSD signal for the target period. As expected, our JSD signal increases at the 2 millionth and 3 millionth event mark, because the distribution parameters change. At the 4 millionth event, we use the original reference distribution and notice that the JSD signal returns to values below the threshold, considering it a normal state, as we intended. Figure 6.14b is a zoom in on the first 2 millionth events before any change occurs, and we see no false positives.

6.1.1.5 An Attempt to Deceive our Method

In this experiment, we attempt to deceive our method by using a different distribution from the reference period but ensuring its domain is roughly the same as the reference one.

We used the reference dataset R5 (Figure 6.15) which had 5 million events, with a single feature that followed a gaussian with $\mu = 100$ and $\sigma = 10$. Our target dataset T5 is represented in Figure 6.16. For the first 2 million events, the generating distribution used was similar to the one used in the reference dataset, a gaussian with $\mu = 100$ and $\sigma = 10$. However, at the 2 millionth event, we change the gaussian standard deviation to $\sigma = 30$. Then, at the 4 millionth event, we change the distribution type. We use an uniform distribution with lower bound $a = 80$ and upper bound $b = 120$. Note in Figure 6.16 that this last portion resembles the first one, the reference.



(a) JSD signal and threshold

(b) Zoom in on the first 2 million events

Figure 6.14: JSD signal and threshold

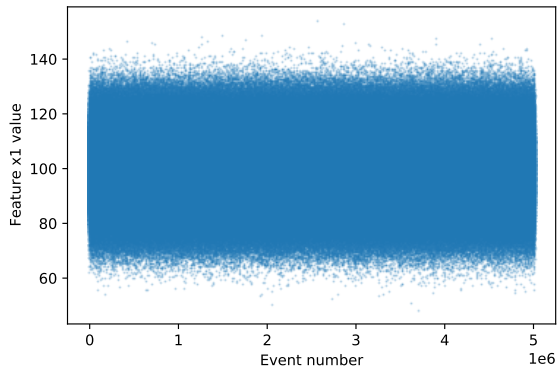


Figure 6.15: Time-series for dataset R5

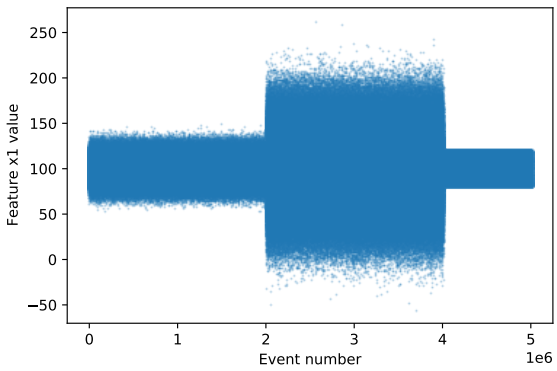


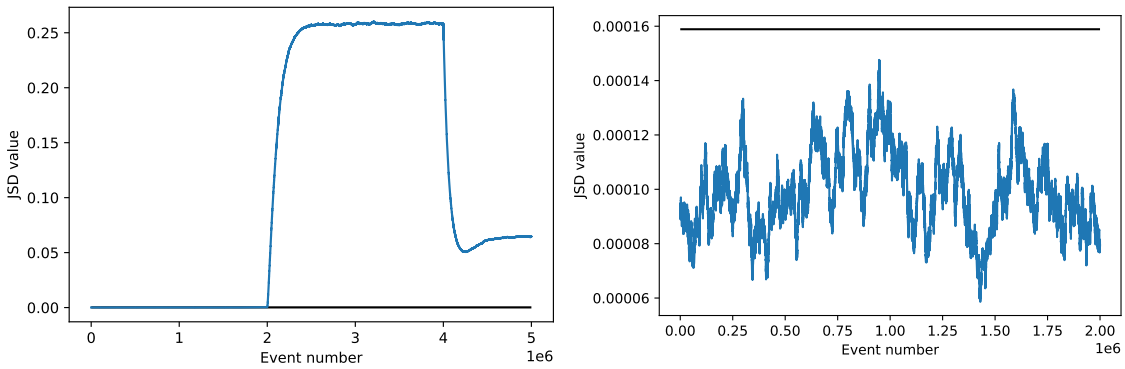
Figure 6.16: Time-series for dataset T5

Let's take a look at the signal computed for the target period. Figure 6.17a shows us the JSD signal for dataset T5 (Figure 6.16). As expected, we see a rapid growth of JSD value after the 2 millionth event, when we change the Gaussian parameters. We then change the distribution type, from a gaussian to a uniform one. Despite the latter having a domain contained within the former, we see that our signal is accurate and does not drop below the alert threshold. This is still a different distribution that we want to alert. Figure 6.17b is a zoom in of the first 2 millionth events and once again we have no false positives.

6.1.2 Multi-Feature Analysis

Before moving to real data, we performed one last experiment, using synthetic data but this time performing a multi-feature analysis. Table 6.2 shows the underlying generating distributions and parameters for each of the four features (x_1, x_2, x_3 and x_4) in the reference multi-feature dataset R6. Figures 6.18, 6.19, 6.20 and 6.21 show the reference time-series for features x_1, x_2, x_3 and x_4 on the reference dataset R6, respectively.

We used a target dataset T6 with 5 million events and the same four features (x_1, x_2, x_3 and x_4). Feature x_1 followed the reference distribution for the 5 million events of the target dataset. We did



(a) JSD signal and threshold

(b) Zoom in on the first 2 million events

Figure 6.17: JSD signal and threshold

Feature	Reference Distribution	Parameters
x_1	Gaussian/Normal	$\mu = 100$, $\sigma = 5$
x_2	Gaussian/Normal	$\mu = 100$, $\sigma = 5$
x_3	Uniform	$a = 200$, $b = 250$
x_4	Uniform	$a = 50$, $b = 125$

Table 6.2: Multi-feature reference dataset and feature distributions

not change this feature underlying distribution to serve as a control, *i.e.*, we expect no alerts for this one. Figure 6.22 shows the time-series for the target dataset T6 and feature x_1.

Feature x_2's underlying distribution changed four times and returned to normal, so we expect to see alarms with increasing divergence values but eventually no more alarms. Figure 6.23 shows the time-series for the target dataset T6 and feature x_2. Feature x_2 followed a gaussian with $\mu = 100$ and $\sigma = 5$ like in the reference period up to the 1 millionth event mark, then a gaussian with $\mu = 100$ and $\sigma = 10$ up to the 2.5 millionth event, then a gaussian $\mu = 100$ and $\sigma = 15$ until the 3 millionth event, then a gaussian $\mu = 100$ and $\sigma = 20$ until the 4 millionth event and finally returned to the original gaussian with $\mu = 100$ and $\sigma = 5$ until the last event. Table 6.3 sums up the distributions used and its parameters for each interval of the reference dataset for feature x_2.

From Event	To Event	Distribution	Parameters
0	1,000,000	Gaussian	$\mu = 100, \sigma = 5$
1,000,000	2,500,000	Gaussian	$\mu = 100, \sigma = 10$
2,500,000	3,000,000	Gaussian	$\mu = 100, \sigma = 15$
3,000,000	4,000,000	Gaussian	$\mu = 100, \sigma = 20$
4,000,000	5,000,000	Gaussian	$\mu = 100, \sigma = 5$

Table 6.3: Feature x_2 generating distributions

Feature x_3's underlying distribution changed three times and never returned to normal, so we expect to see alarms with increasing divergence values up until the end of the stream analysis.

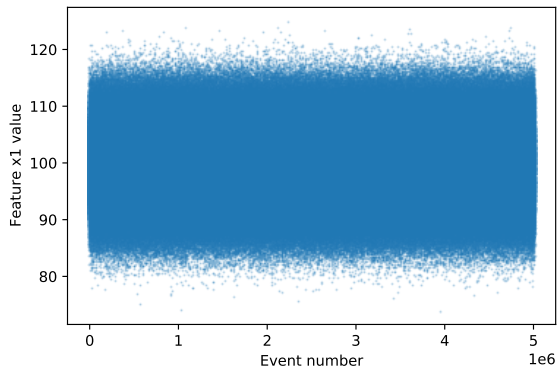


Figure 6.18: Feature x_1 reference time-series

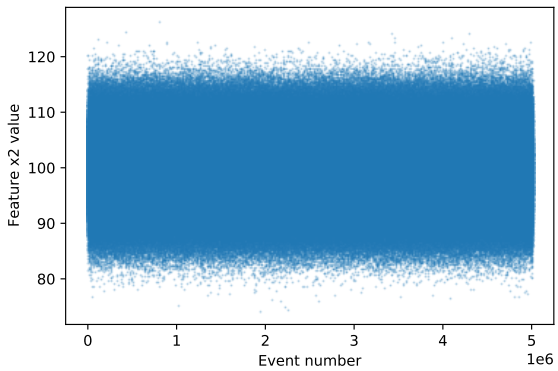


Figure 6.19: Feature x_2 reference time-series

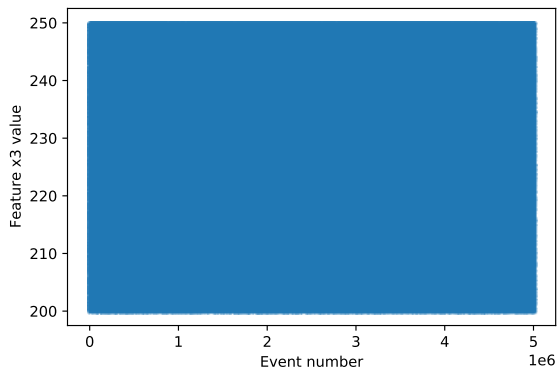


Figure 6.20: Feature x_3 reference time-series

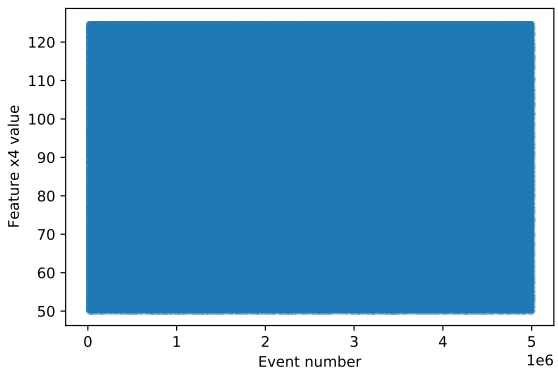


Figure 6.21: Feature x_4 reference time-series

Figure 6.24 shows the time-series for the target dataset T6 and feature x_3. Feature x_3 followed a uniform distribution with $a = 200$ and $b = 250$ like in the reference period up to the 1.5 millionth event mark, then a uniform distribution with $a = 200$ and $b = 220$ up to the 2 millionth event, then a uniform distribution with $a = 180$ and $b = 200$ until the 3.5 millionth event and finally a gaussian with $\mu = 50$ and $\sigma = 20$ until the last event. Table 6.4 sums up the distributions used and its parameters for each interval of the reference dataset for feature x_3.

From Event	**To Event**	**Distribution**	**Parameters**
0	1,500,000	Uniform	$a = 200, b = 250$
1,500,000	2,000,000	Uniform	$a = 200, b = 220$
2,000,000	3,500,000	Uniform	$a = 180, b = 200$
3,500,000	5,000,000	Gaussian	$\mu = 50, \sigma = 20$

Table 6.4: Feature x_3 generating distributions

Feature x_4's underlying distribution changed two times and returned to normal, so we expect to see no alarms towards the end of the stream analysis. Figure 6.25 shows the time-series for

the target dataset T6 and feature x_4. Feature x_4 followed a uniform distribution like the reference one, with $a = 50$ and $b = 125$ up until the 1.5 millionth event. After that, it followed a uniform distribution with $a = 125$ and $b = 200$ until the 3 millionth event where it resumed its original uniform distribution (with $a = 50$ and $b = 125$) until the end. Table 6.5 sums up the distributions used and its parameters for each interval of the reference dataset for feature x_4.

From Event	To Event	Distribution	Parameters
0	1,500,000	Uniform	$a = 50, b = 125$
1,500,000	3,000,000	Uniform	$a = 125, b = 200$
3,000,000	5,000,000	Uniform	$a = 50, b = 125$

Table 6.5: Feature x_4 generating distributions

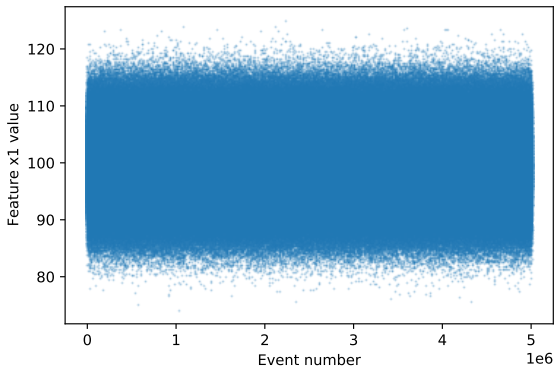


Figure 6.22: Feature x_1 target time-series

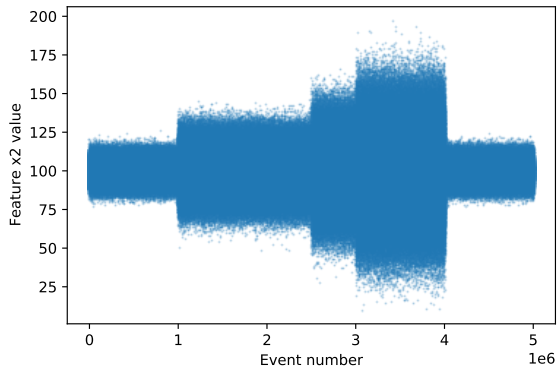


Figure 6.23: Feature x_2 target time-series

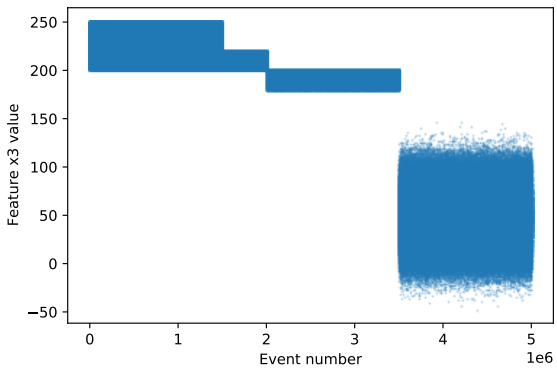


Figure 6.24: Feature x_3 target time-series

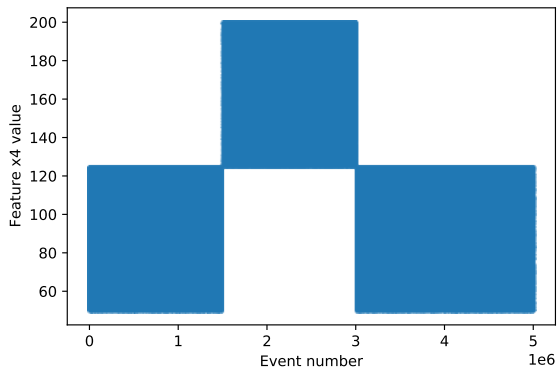


Figure 6.25: Feature x_4 target time-series

In this experiment, we used a tuple-based half-life $n_{1/2} = 62500$, *i.e.*, sampling and target windows of 250,000 tuples. We used 100 bins for the reference histogram plus the two special infinity bins. In this multi-feature scenario, we used the Holm-Bonferroni correction discussed in Section 5.3.3 and set the Family-Wise Error Rate (FWER) to 1% (FWER = 0.01). Furthermore,

instead of doing 1000 samples as we did previously in an ad-hoc way, we use Equation 5.3 to determine the minimum number of samples to make while ensuring $\gamma = 1\%$ (recall Section 5.3.4). For our four feature analysis scenario, using as inputs $\gamma = 0.01$ and $FWER = 0.01$ in Equation 5.3 tells us we need to make 1840 samples minimum. Using 300 Spark executors, each given 3G of memory, we finished the batch phase in 418 seconds.

In Section 5.3.3, we mention that, in our stream analysis, we perform our multiple hypothesis divergence tests on a fixed user-defined frequency. Finding an optimal frequency value for the test falls out of scope for this thesis, but ideally, we set it to be large enough so that we do fewer computations and achieve higher throughput but still get alerts in useful time. In this experiment, we performed the divergence test with the Holm-Bonferroni correction for every new event (frequency of 1 event). We did this to collect a list of ordered corrected p-values for each feature. A corrected p-value (derived from Equation 5.2) is given by:

$$p - value_{corrected} = p - value_{distance_x} * (m - k) \tag{6.1}$$

With this list, for each feature, we can plot the corrected p-values on each event and compare it with the FWER threshold α. Note that despite performing the test at the maximum frequency (in real-time) we still achieved a throughput of 17,921 transactions per second (TPS), even if only for four features.

In the following plots, unlike the JSD signal plots presented before, a feature is in a divergent state if its p-value is below the threshold, according to Equation 5.2. Working with corrected p-values (Formula 6.1), a feature is considered divergent if its corrected p-value is below the threshold.

In Figure 6.26 we see the plot of corrected p-values for feature x_1 and the probability threshold black horizontal line at the bottom. Recall that feature x_1's values were generated by a Gaussian distribution for both the reference period (Figure 6.18) and the target period (Figure 6.22), with the same parameters. We observe that the corrected p-values never pass below the threshold, as we expected. Hence, this feature is never alerted as divergent.

Figure 6.27 shows a plot of feature x_2's corrected p-values and the FWER threshold. Feature x_2 underlying distribution was changed four times (Table 6.3): at the 1, 2.5, 3 and 4 millionth events. The first change is consistent with the sudden drop in corrected p-value for this feature, as we cross below the threshold and enter a divergent state. Around a thousand events after the change we saw p-values of 0.2% (or 0.002). The values keep dropping as time passes and as we introduce bigger changes in distribution. We resume the reference or original distribution at the 4 millionth event. However, as we can see from a close-up on the original plot (Figure 6.28), it takes around 600,000 events for the target sliding histogram aggregation to once again resemble the reference one, resulting in a high enough p-value and thus stop alerting x_2 as divergent.

Feature x_3 was the only feature that never returned to its original distribution and is the only one that should be reported until the end as divergent. This is in fact the case as illustrated in Figure 6.29. The first change to x_3's generating distribution was introduced at the 1.5 millionth

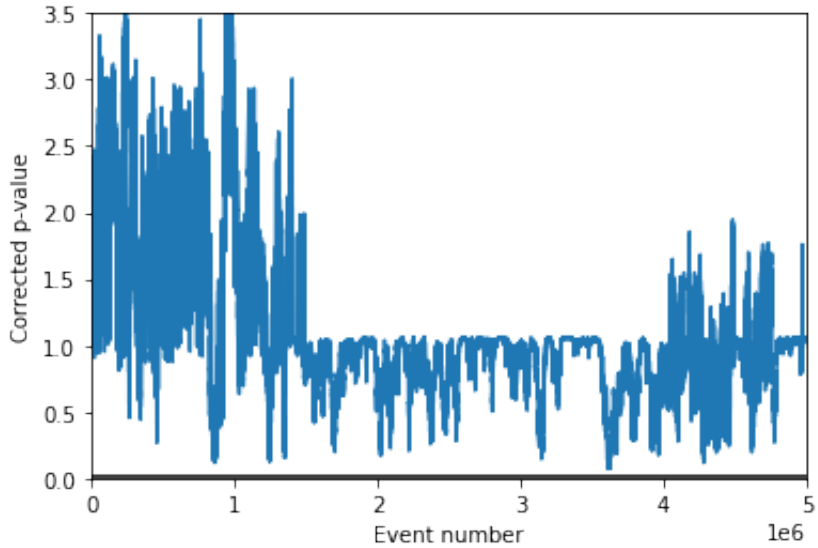


Figure 6.26: Feature x_1 corrected p-values and threshold

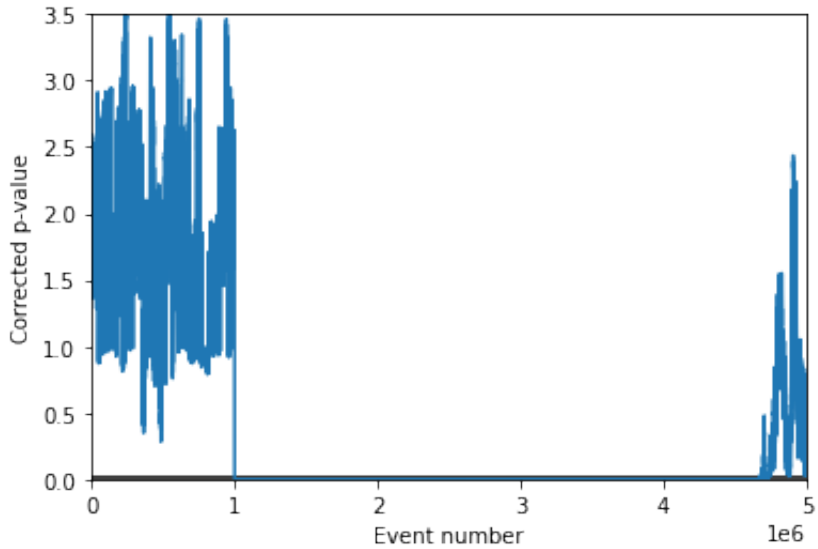


Figure 6.27: Feature x_2 corrected p-values and threshold

event and we clearly see the corrected p-value rapidly approaching values close to zero. For the rest of the dataset, this x_3's p-value remains below the threshold, as we never swap to its original distribution.

Finally, feature x_4's corrected p-value plot is given by Figure 6.30. Similarly to feature x_2, we resume the original distribution at a certain point, in this case at the 3-millionth event. However, as we can see from a close-up on the original plot (Figure 6.31), it takes around 810,000 events for the target sliding window aggregation to be considered similar enough to the reference one and stop alerting x_4 as divergent.

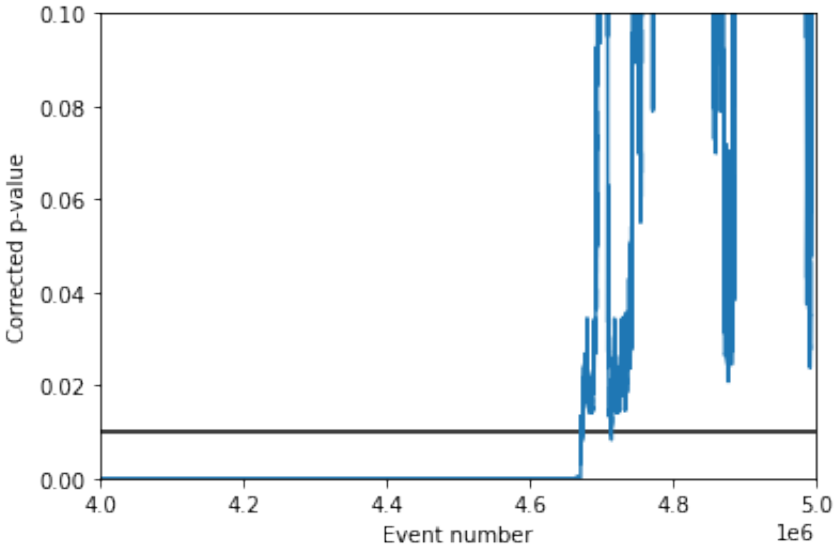


Figure 6.28: Feature x_2 zoomed in corrected p-values and threshold

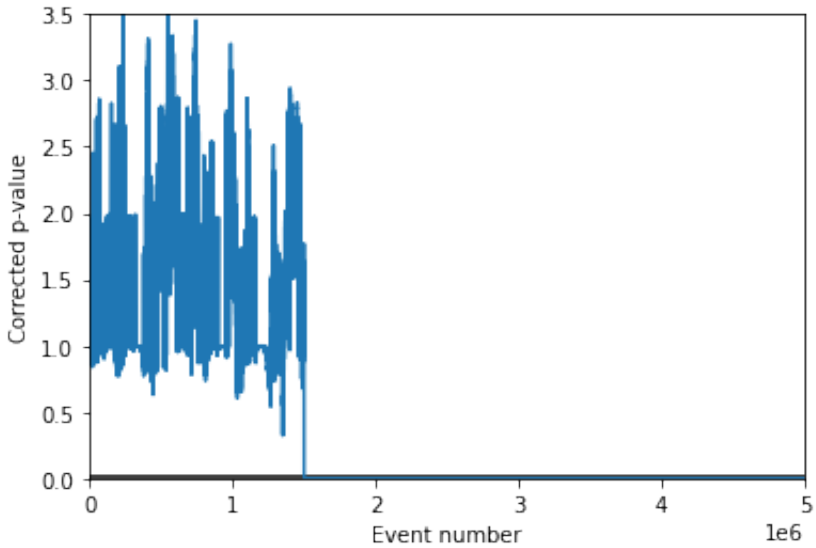


Figure 6.29: Feature x_3 corrected p-values and threshold

With this experiment, we further validate that the p-value is a measure of how divergent a feature is: the closer to zero, the lower the probability of the statistical test result and hence the more divergent a feature is considered to be. At the 1.5 millionth event, we swap both x_3 and x_4's distribution. For the first 2,535,000 events, x_4 has a lower p-value than x_3 and hence is considered more divergent. However, after that, we change x_3's distribution so aggressively that x_3 starts being reported as most divergent, with a lower p-value than x_4.

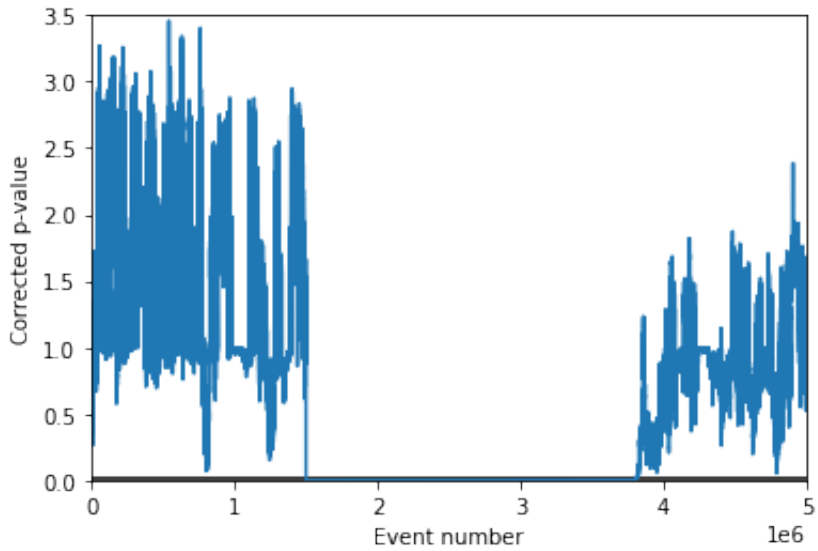


Figure 6.30: Feature x_4 corrected p-values and threshold

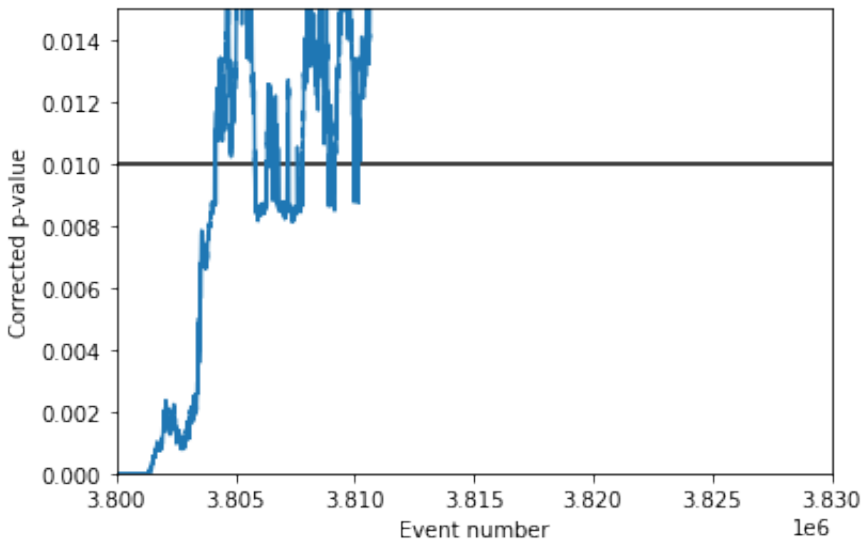


Figure 6.31: Feature x_4 zoomed in corrected p-values and threshold

6.2 Experiments with Real Data

Due to time constraints, we experimented with data from only one merchant. The merchant's entity as well as feature names will remain anonymous due to confidentiality reasons.

Feedzai provides a fraud detection solution for this merchant and had stored two contiguous in time datasets of credit card transactions that, combined, covered a period of roughly 1 year and 9 months. In this experiment, we used the older one as the reference dataset and the more recent one as the target period. Both datasets were already feature-engineered [66] and had a total of 183 numerical plus numerically encoded categorical features (*e.g.*, one-hot encoded or ordinal encoded [67]). Table 6.6 shows summary statistics for both datasets.

Dataset	From	To	Transactions
Reference	Tuesday, September 5, 2017	Friday, June 29, 2018	1,604,509
Target	Friday, June 29, 2018	Sunday, June 30, 2019	4,032,505

Table 6.6: Summary statistics for reference and target merchant datasets

In the batch phase, we used 100 Spark executors, each with 12G of memory. We determined that one week's worth of data for this merchant was on average equivalent to 37,803 events. We set the half-life to be $n_{1/2} = 37,803$ and discard events whose EMA weight is 6% or less, which is equivalent to using sampling windows of data of four times the half-life (as detailed in Section 5.3.1). Hence, our samples's windows had 151,212 events. Using Equation 5.3, we determined that for 183 features, a FWER of 1% and γ (as defined in Section 5.3.4) of 10% we needed to make a minimum of 42,137 samples. Our batch analysis took 5 hours and 34 minutes.

After the batch phase, we had the necessary inputs for the streaming phase (detailed in Section 5.3.2), namely, for each feature, the reference histograms, the list of observed divergence values and the last sample's histogram. We then used the target dataset, simulating a stream of ordered events. Our monitoring system processed the dataset event by event in sequential, time-ordered fashion, emulating a stream computation and reported divergent features with associated p-values. We performed our multiple hypotheses with the Holm-Bonferroni correction (Section 5.3.3) for every 1000 events. It took us 382,363 milliseconds (roughly 6 minutes) to process the 4,032,505 transactions, amounting to a throughput of 10,546 transactions per second (TPS).

This merchant and in particular these datasets were known to suffer a lot from concept drift. Our system produced a large list of alerts, from nearly the beginning of the target period until the end. Due to a large number of features (183), we only show some plots. Furthermore, we refer to features with anonymized identifiers, like feature x_j. In the following plots, we show the corrected p-values (Formula 6.1) for some features at each event of the time-series and the FWER threshold set at 1%. A feature is considered divergent at a given timestamp if below that threshold line.

Figure 6.32 is a plot of the reference time-series of feature x_1 and Figure 6.33 the corresponding target time-series. In Figure 6.34 we plot the corrected p-values for the each event of the target period. The reference and target periods look very similar and in fact we have very few alarms

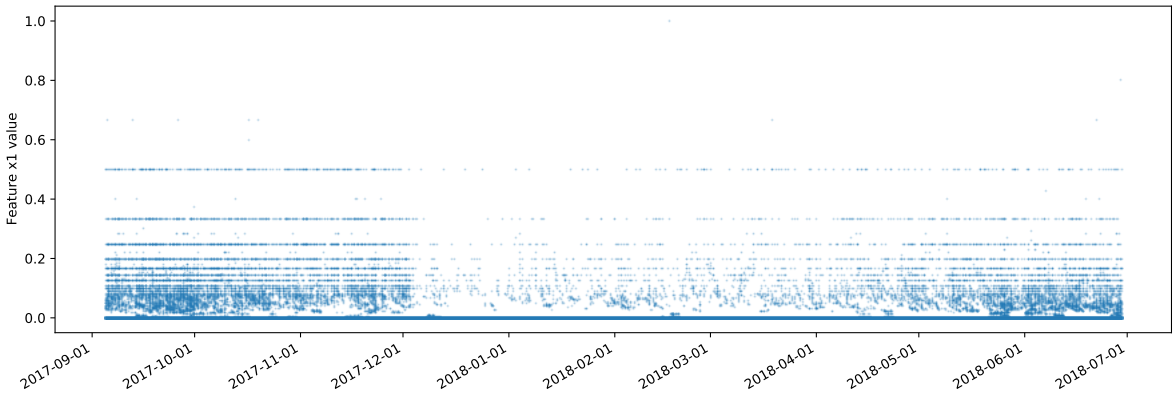


Figure 6.32: Merchant feature x_1 reference time-series

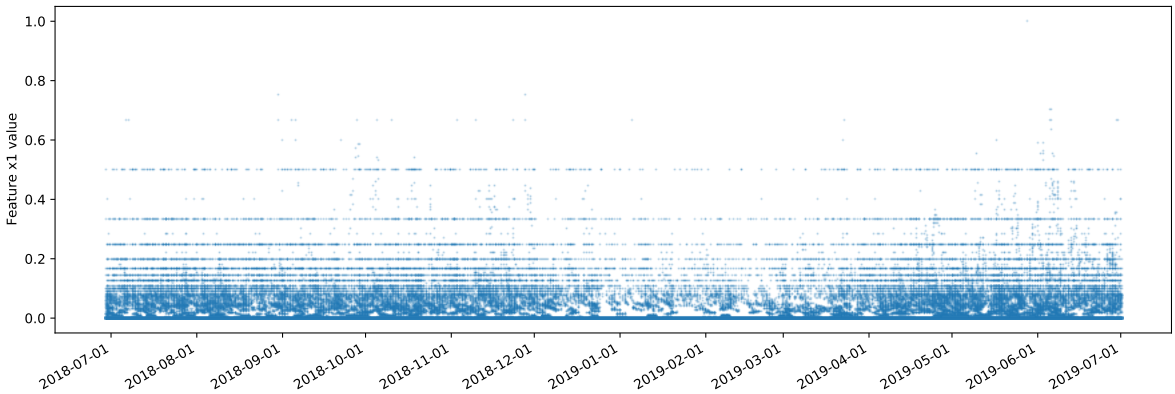


Figure 6.33: Merchant feature x_1 target time-series

for this feature, *i.e.*, the number of times the line crosses to the bottom half of the plot, imposed by the threshold line. Furthermore, the alarms appear to be roughly in the middle portion of the

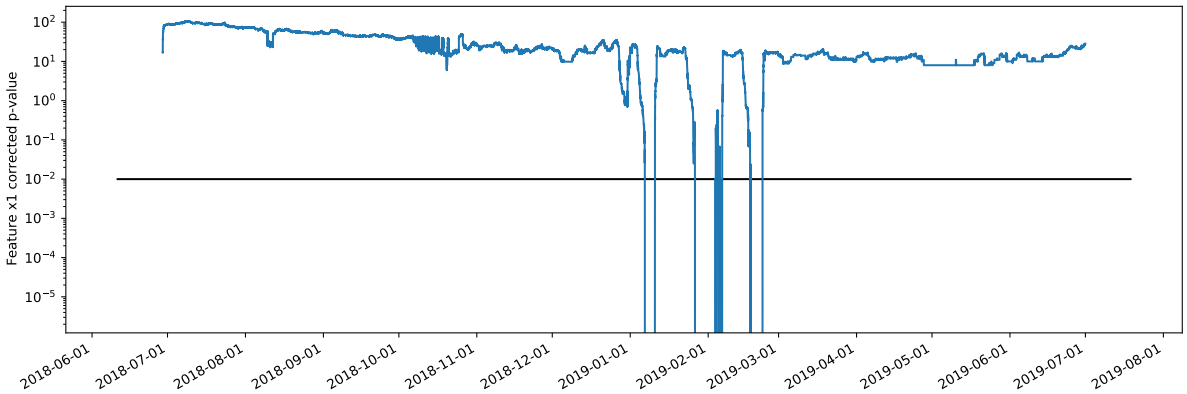


Figure 6.34: Merchant feature x_1 corrected p-values

time-series, where in fact the target period appears to have less points. We can see roughly three alarms. The first one in early January 2019 for a few days, the second one lasting about a week at the end of January 2019 and the last one in the second half of February 2019 again for a few days.

Now let us look at features x_2 and x_3. Figures 6.35 and 6.36 show feature x_2's time-series plots for the reference and target periods, respectively. Figures 6.37 and 6.38 show feature x_3's

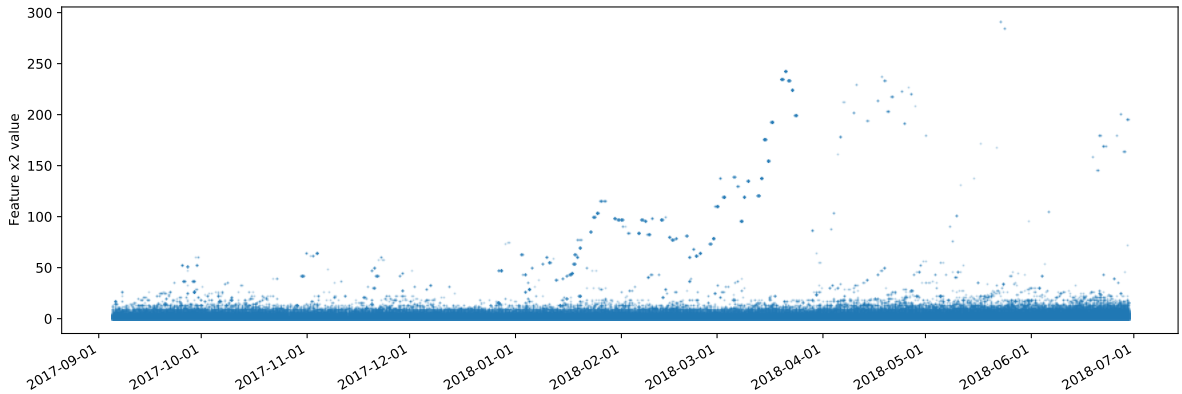


Figure 6.35: Merchant feature x_2 reference time-series

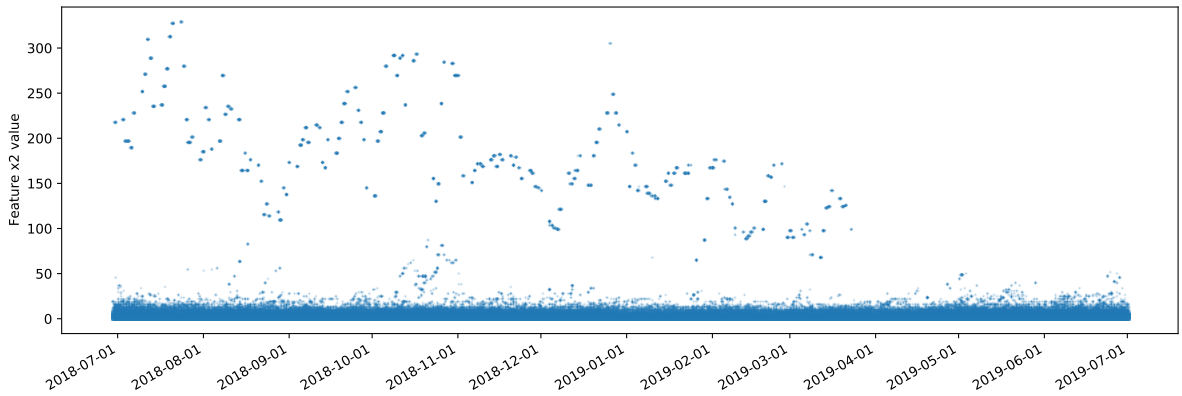


Figure 6.36: Merchant feature x_2 target time-series

time-series plots for the reference and target periods, respectively.

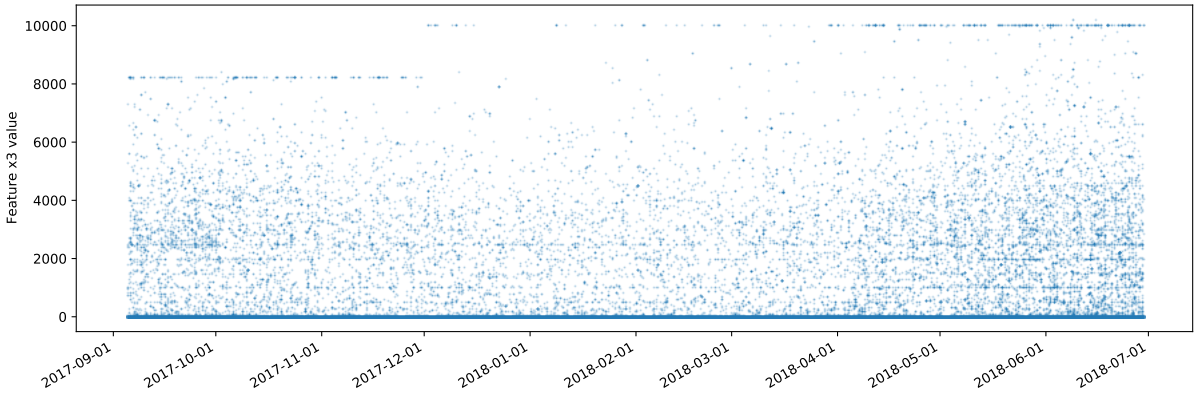


Figure 6.37: Merchant feature x_3 reference time-series

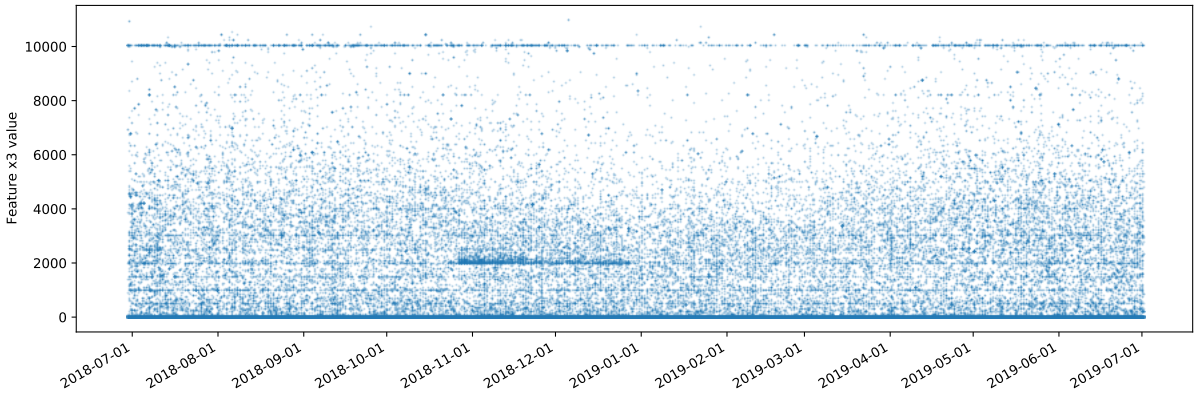


Figure 6.38: Merchant feature x_3 target time-series

Relative to x_1, features x_2 and x_3 also have very similar reference and target periods. Just like feature x_1, we expect to have some alarms but still not too many. Once again, to visualize the alarms, we plot the corrected p-values for each event of the target time-series for each feature. Figure 6.39 is a plot of feature x_2's corrected p-values and the FWER threshold of 1%. Likewise, Figure 6.40 shows the same plot but for feature x_3. The corrected p-values plot shows us once

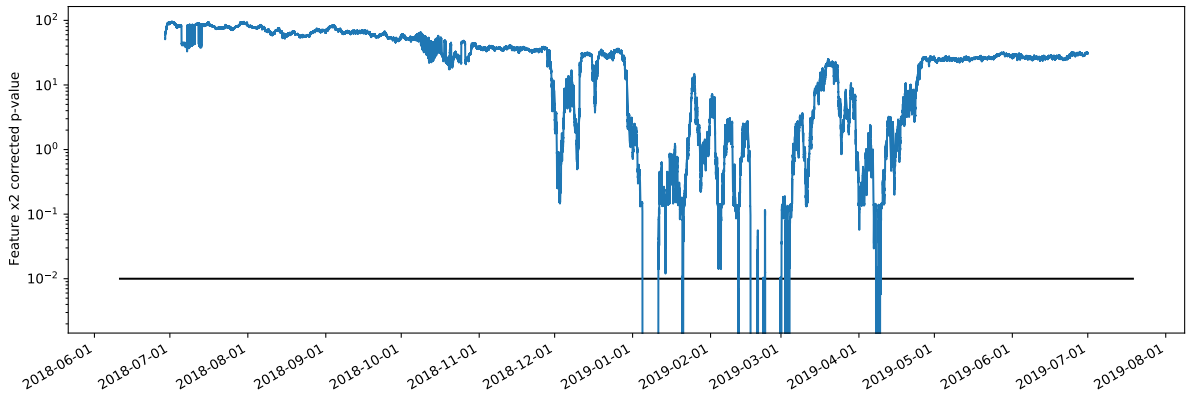


Figure 6.39: Merchant feature x_2 corrected p-values

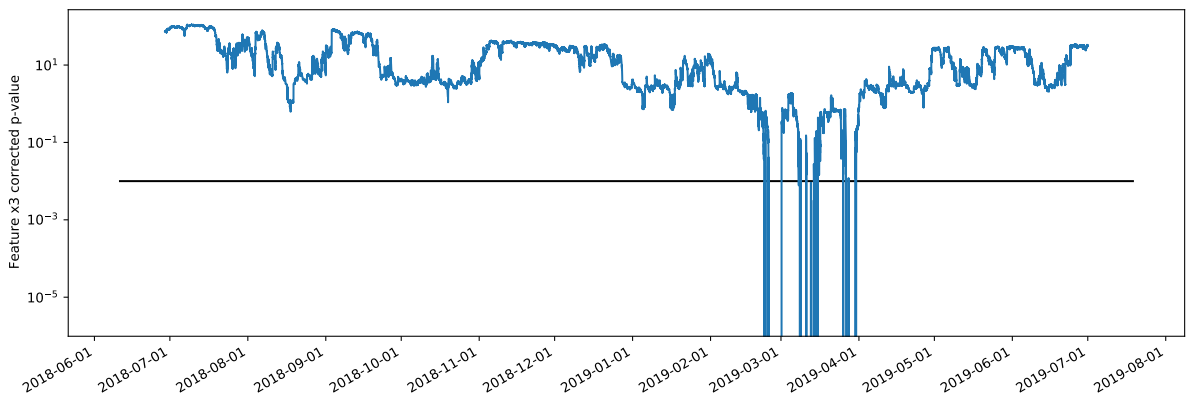


Figure 6.40: Merchant feature x_3 corrected p-values

again that both x_2 and x_3 features had some alarms but eventually rose above the threshold and resumed their reference distribution. This is expected from features whose reference and target time-series plots look so similar.

Next, we analyze a feature with very different reference and target time-series. Feature x_4's reference time-series is presented in Figure 6.41 and its target period in Figure 6.42. Notice that even if for the most part both reference and target periods values belong to a domain between 0 and 1, at a certain point in time, in the target time-series, a lot of -1 values appear. In Figure 6.43 we plot the corrected p-values for feature x_4 target time-series. Like the previous analyzed features, x_4 also drops below the threshold some times. However, unlike the other features, where feature values return to normal and alarms stopped being raised, x_4 remains below the threshold line until

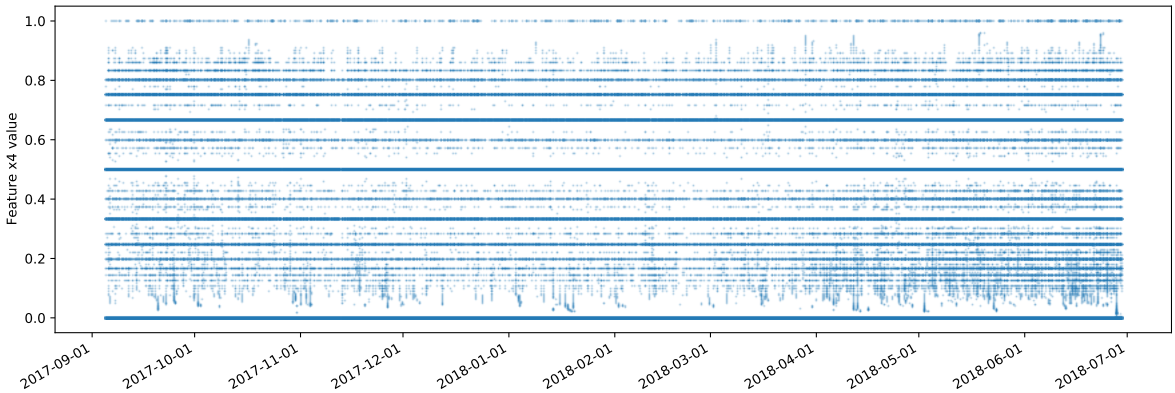


Figure 6.41: Merchant feature x_4 reference time-series

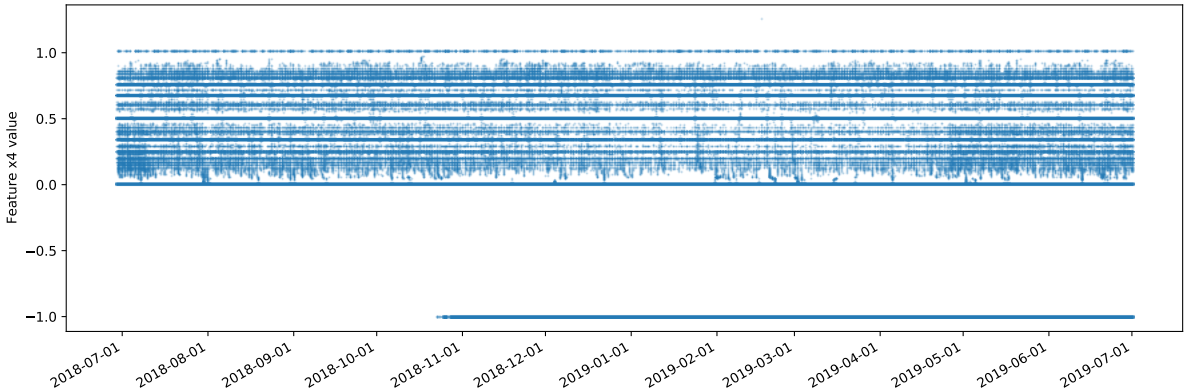


Figure 6.42: Merchant feature x_4 target time-series

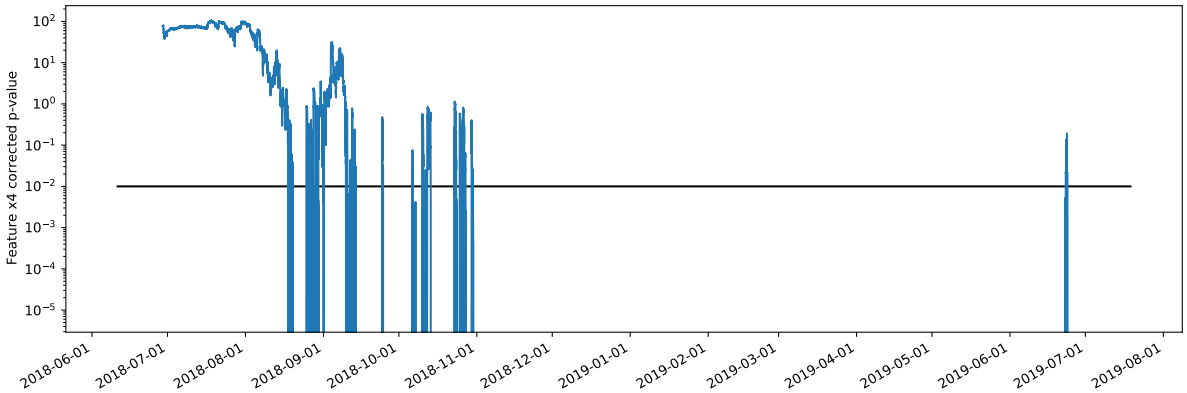


Figure 6.43: Merchant feature x_4 corrected p-values

the end of the target period, where a small spike occurs. The last time feature x_4 was ever above the threshold line (and hence considered normal) was before the appearance of the -1 values in the target period. This seems to work as expected: -1 values were so off the reference domain that for

here on out, as long as these values keep appearing, we will not stop reporting x_4 as divergent.

The -1 value is often used as a placeholder for missing values, *i.e.*, if some problem occurs and the value for a feature is missing at a given timestamp or event, -1 is used. A follow-up question that is raised is if we should include some mechanism of dealing with missing values in our method. However, we discard this idea and defend that we should not: missing values are indeed something we want to detect. For instance, if a certain device responsible for reporting one feature stops working, we want to know that. This scenario is likely to happen in a distributed Internet-of-Things (IoT) system with hundreds of devices transmitting data.

6.2.1 Same Reference and Target Periods

In this experiment, we set out to test our system in a low drift scenario. To guarantee the reference and target periods are as similar as possible, we decided to use the same dataset to work as reference and target. We chose to use the reference dataset from the previous experiment (Reference of Table 6.6) and feed it as input for both our batch and stream analysis as the reference and target periods, respectively. We expect to have less alarms for each feature but not necessarily zero.

We first show the corrected p-value plots when both reference and target periods are the same, for the previous features x_1, x_2 and x_3 in Figures 6.44, 6.45 and 6.46, respectively.

Feature x_1 has less alerts in this experiment where we use equal reference and target periods. However, the alarms last for about two months. This is unexpected as we predicted we would see no alarms or few ones with short duration. Likewise, we observe that for feature x_2 (Figure 6.45) and x_3 (Figure 6.46) we still have many alarms and some lasting up to one month. We do not

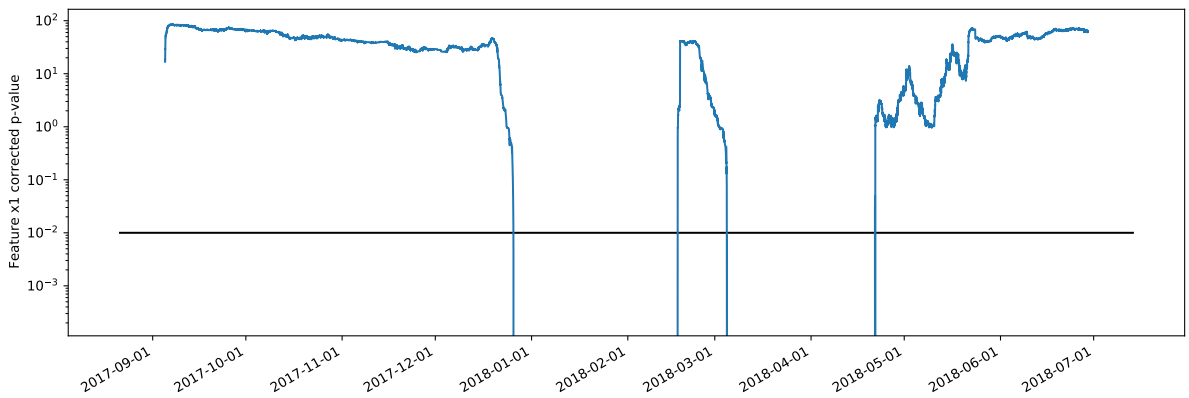


Figure 6.44: Merchant feature x_1 corrected p-values with equal reference and target periods

have a clear explanation for this and due to time constraints we leave further tests to future work. We suspect the issue may be that there is a difference between the batch and streaming phases. In batch we have subsets of events and compute EMA-like histogram aggregations based on those. In streaming, however, we have a continuous stream of data and never expire the 6% of data that in batch is not present, because it is a finite window.

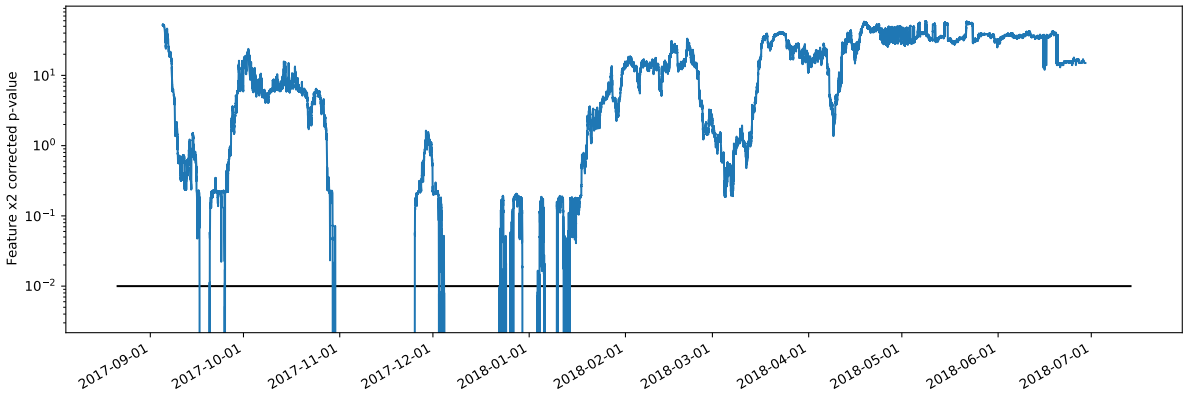


Figure 6.45: Merchant feature x_2 corrected p-values with equal reference and target periods

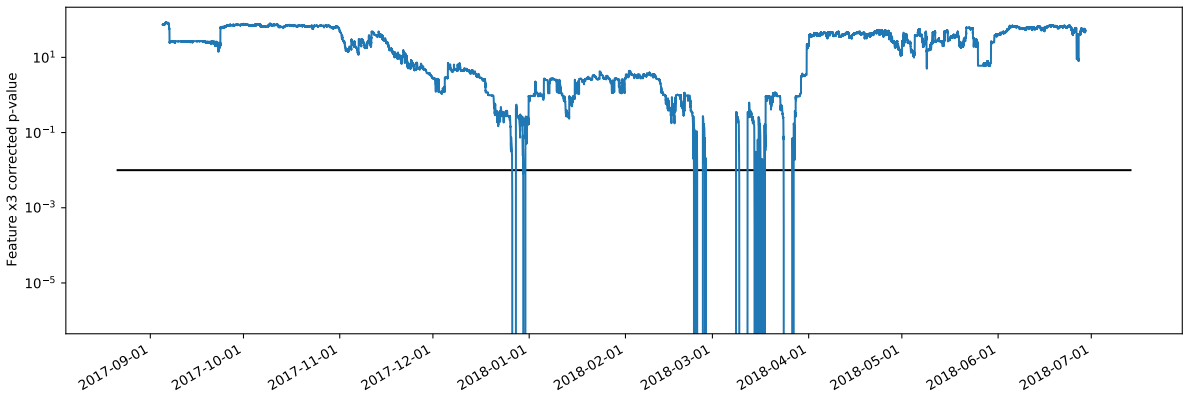


Figure 6.46: Merchant feature x_3 corrected p-values with equal reference and target periods

Now we analyze feature x_4 from the previous experiment, as this one was the one with the highest measured drift. Recall that feature x_4's target time-series (Figure 6.42) was significantly different from the reference one (Figure 6.41). Because of this, the corrected p-values plot from the previous experiment (Figure 6.43) crossed our threshold line multiple times and remained in an alert state for most of the target time-series. In this new experiment, we used the reference period twice. Because of this, the corrected p-values plot is now different. Figure 6.47 shows the corrected p-values plot for feature x_4 when we used the reference period for both phases. Using the same period twice theoretically means we have no drift at all. Hence, we expected to see no alerts. Figure 6.47 shows that we have a few alerts but significantly less than in Figure 6.43. Furthermore, in this new experiment, x_4 eventually stops being reported at all until the end of the dataset and ends at a high p-value. Recall that the higher the p-value the less divergent a feature is. Overall, we see that this feature has fewer alarms and stabilizes well above our alarm threshold which is coherent since we used the same dataset as reference and target periods.

We now introduce a new feature, x_5. Feature x_5 also suffers from a large amount of drift. In the reference and target datasets presented in Table 6.6, feature x_5 reference time-series plot is

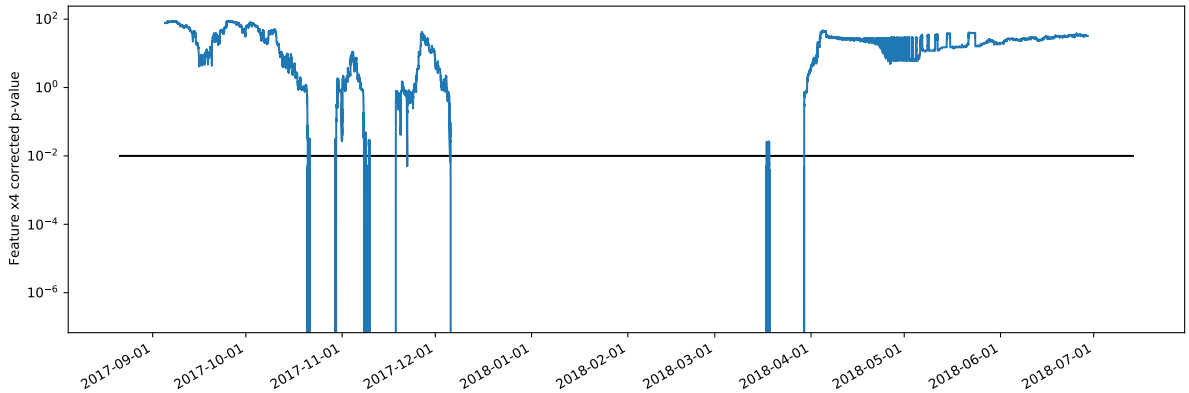


Figure 6.47: Merchant feature x_4 corrected p-values with equal reference and target periods

given by Figure 6.48 and the target time-series by Figure 6.49. The density in the upper domain

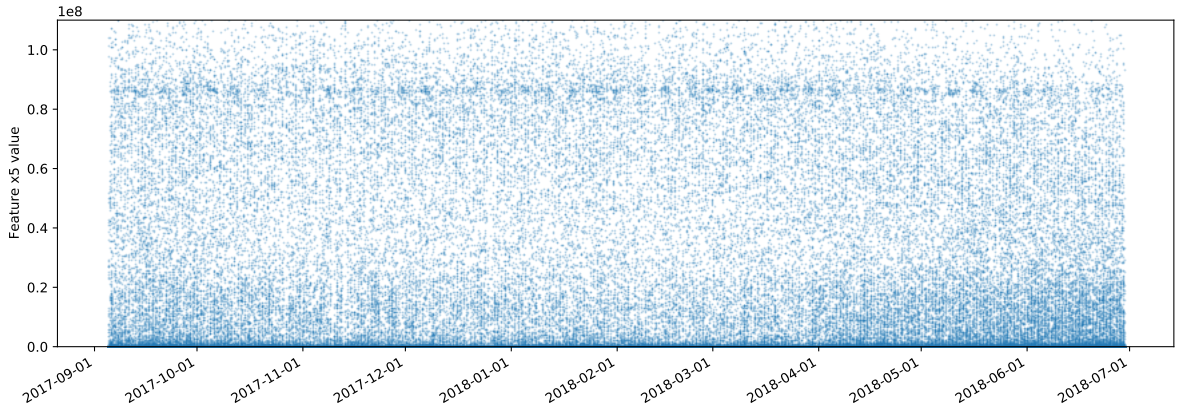


Figure 6.48: Merchant feature x_5 reference time-series

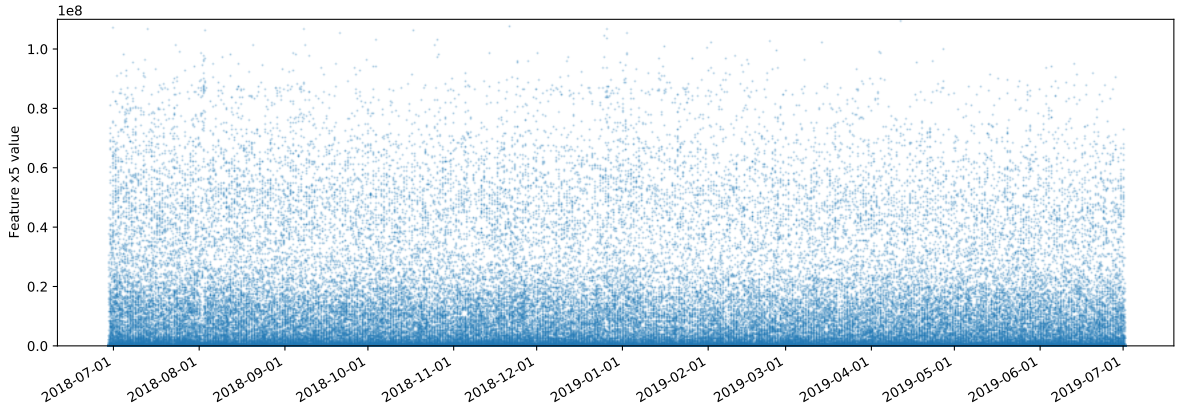


Figure 6.49: Merchant feature x_5 target time-series

of values is clearly different in both reference and target periods. Hence, in our first experiment, this feature had a corrected p-values plot with two alarm states, where the second one lasted until the end of the target period, with the lowest possible p-value (zero), represented in Figure 6.50.

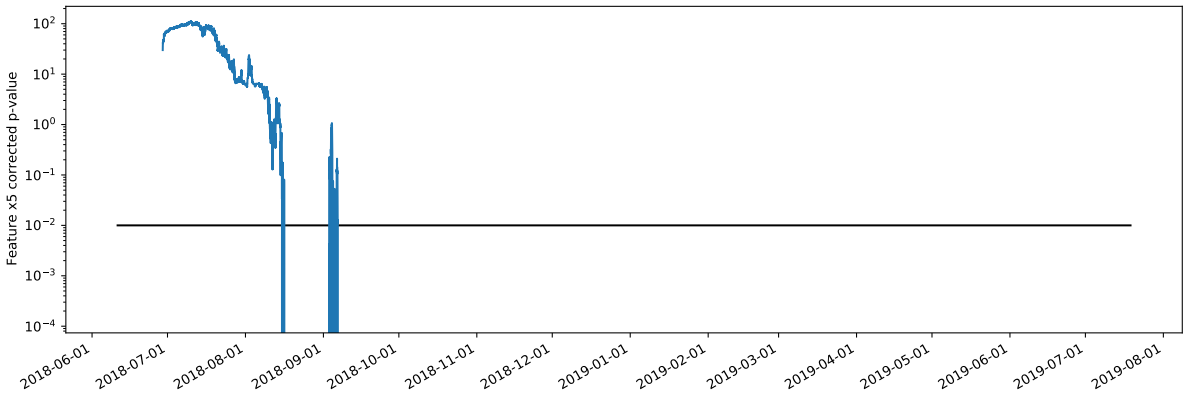


Figure 6.50: Merchant feature x_5 corrected p-values

However, when using the same reference and target periods, in this case using the time-series in Figure 6.48 twice, we achieve our lowest possible drift scenario. The difference in corrected p-values is clear. Figure 6.51 shows this plot when using equal reference and target periods. We observe that our corrected p-values never go below the threshold line, in fact, they stay well above it. This is expected behavior when using equal reference and target periods.

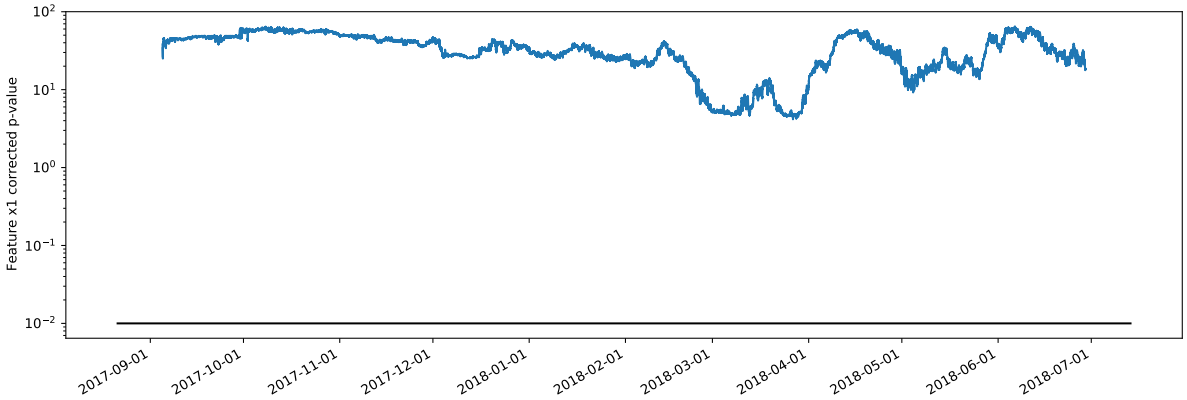


Figure 6.51: Merchant feature x_5 corrected p-values with equal reference and target periods

6.2.2 Building an Accurate Divergence Measure Distribution

Our method comprises two phases: the batch and the streaming analysis. We can not conclude in which phase an error that could explain the last experiment's results exists. To isolate both phases, we conduct this last experiment where instead of using the distribution of divergence measures built in the batch phase we use a distribution we know to be the best fit for the streaming period.

Our best fit divergence distribution was built over the streaming period. To do this, we processed the streaming events once and saved all the JSD measures. We binned these JSD values with 50,000 bins all with equal heights (or counts). Then we performed once again the streaming analysis over the same period but this time using the previously built distribution for which we know it is the best possible fit since it was computed over the same period, built event by event. We set the FWER to be 1% so we expect to see some alarms, *i.e.*, for JSD measures above the 99th percentile.

Recall that in the last experiment, for feature x_1, using the same target and reference periods, we had two large alarms that lasted months (Figure 6.44). Using the same periods but with the newly computed distribution of divergence measures we obtain the corrected p-values plot shown in Figure 6.52. This plot actually makes sense and represents expected behavior. Notice that

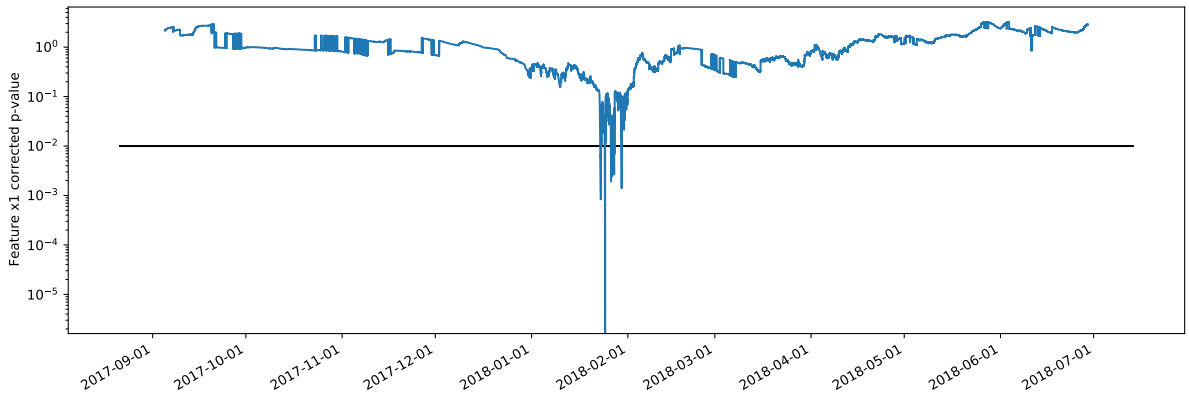


Figure 6.52: Merchant feature x_1 corrected p-values with equal reference and target periods and using the best-fit divergence distribution

we have only a few short-lasting alarms, which correspond to the JSD measures above the 99th percentile. These can be thought of as false positives. In reality, the definition of the multi-test we perform is exactly this: alert events above the 99th percentile (FWER=1%).

Figures 6.53, 6.54, 6.55 and 6.56 show the corrected p-value plots for this experiment for features x_2, x_3, x_4 and x_5, respectively. The analysis performed for feature x_1 applies for all of these features. Notice that for all plots we have very few alarms of very short durations. This coherency in results across multiple features seems to indicate that our streaming analysis is accurate and detects the outliers accurately if the divergence measure distribution used is well built.

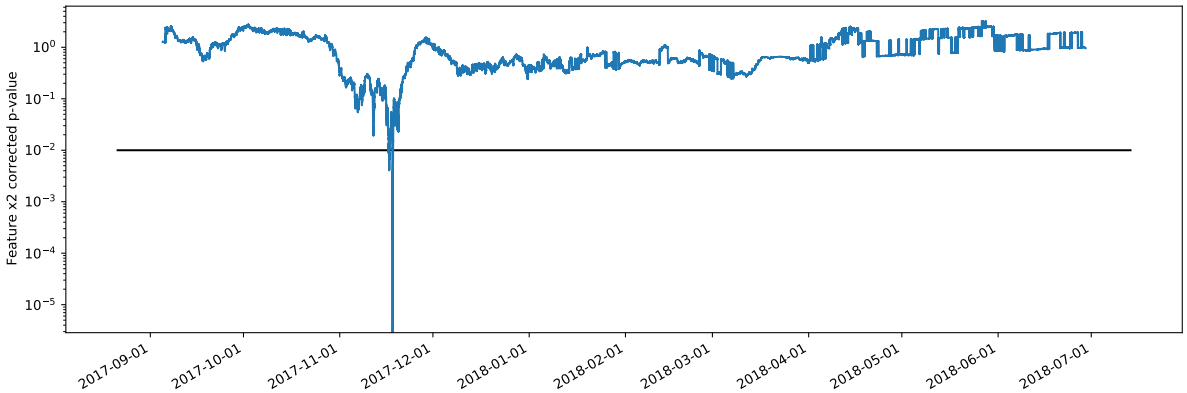


Figure 6.53: Merchant feature x_2 corrected p-values with equal reference and target periods and using the best-fit divergence distribution

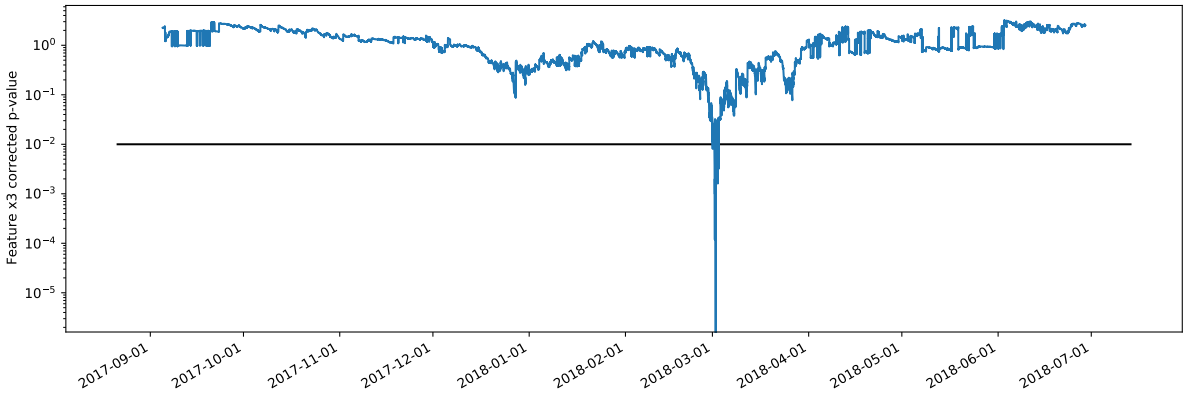


Figure 6.54: Merchant feature x_3 corrected p-values with equal reference and target periods and using the best-fit divergence distribution

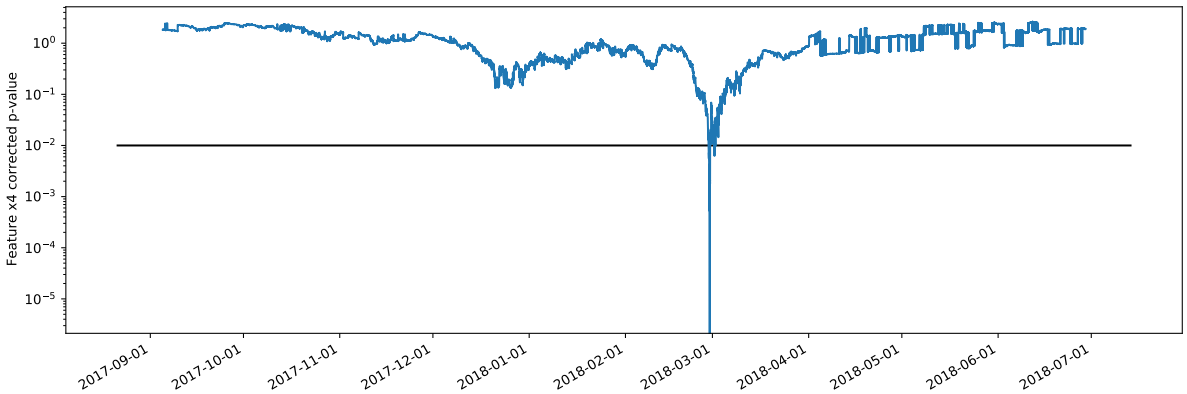


Figure 6.55: Merchant feature x_4 corrected p-values with equal reference and target periods and using the best-fit divergence distribution

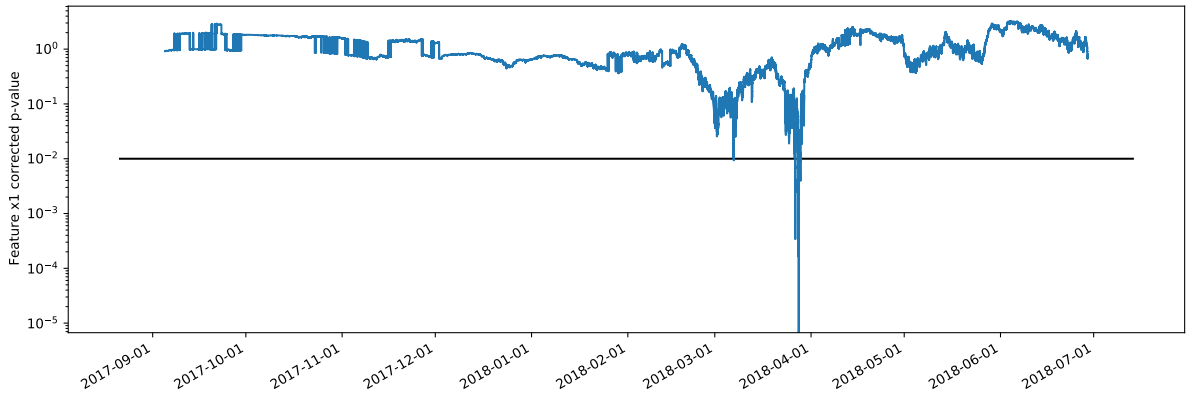


Figure 6.56: Merchant feature x_5 corrected p-values with equal reference and target periods and using the best-fit divergence distribution

6.3 Conclusions

In this section, we summarize our experimental tests, results and uncovered issues.

We began by performing single-feature analysis of synthetic data. The results obtained with this set of experiments were what we expected, in the sense that the artificially introduced anomalies in the datasets were all correctly reported. Still working with synthetic data, we performed our first multi-feature analysis. In this scenario, we applied the Holm-Bonferroni multiple test correction to the p-values of each hypothesis. Once again, each feature was correctly reported as divergent at the timestamps where we changed the underlying generating distribution.

We then moved on to real data. Due to time constraints and the scope of this thesis, we experimented only with data from one merchant. In our first experiment with real data, we ran our method with what we thought was an ideal set of parameters and produced a lot of alarms. A close analysis of the reference and target time-series of each feature revealed that some alarms indeed made sense, as this dataset contained a lot of drift.

In our second experiment with real data, we attempted to reduce concept drift to a maximum by using the same period as reference and target. To our surprise, we still produced a large number of long-lasting alarms. We defer the investigation of the root cause of the problem to future work and leave a few untested hypotheses as solutions for this issue, like the possibility that the EMA-like histograms in batch work with subsets of data and the streaming ones work with infinite amounts of data. In other words, for the streaming EMAs, the previous events never have their contribution set to zero, even though the weights tend to zero, whereas the batch EMAs worked with a finite dataset. Additionally, our sampling may have been insufficient and hence the divergence measure distribution was inaccurate. To test the first hypothesis, experiments with higher smoothing factors for streaming EMAs should be carried out. To test the second one, a lower γ parameter should be used in the batch phase, hence increasing the number of samples to be made, potentially building a more accurate distribution of JSD values.

In our final experiment, we aim to build the most accurate as possible distribution of divergence measures by building it based on the measured divergences observed in streaming. In our second streaming run, we used this distribution and observed that the alarms produced were very few and short-lasting ones. The few alarms raised correspond to divergence measures above the 99th percentile. With this experiment, we conclude our streaming analysis is correctly built but depends a lot on how accurate the divergence measure distribution is. As discussed, future tests should be carried out where more samples are made, to ensure a more accurate divergence measure distribution is computed.

Chapter 7

Conclusion

7.1 Hypothesis Revisited

In Section 1.4 we defined our hypothesis or main question to address as to whether sliding window aggregations and outlier detection methods combined could be used to detect data pattern shifts in data streaming scenarios in the presence of high volumes of high velocity, highly skewed and seasonal data in real-time and with a low memory footprint. Before discussing our hypothesis, we revisit the research questions posed in Section 4.5 and provide answers to them.

(RQ1) What aggregation can we use to properly encode data distributions?

First, we searched for a set of aggregations that would encode our distribution in the best way. Later on, we concluded that a simple histogram is better than any other set of aggregations. Histograms are a representation of the distribution of numerical data thus being ideal to encode the distribution of feature values.

In brief, the answer to RQ1 is that we can use histogram aggregations to properly encode data distributions.

(RQ2) How can we maintain such aggregation in a sliding window fashion in real-time with constant memory usage?

This research question is about two things: maintaining the sliding window aggregation in real-time and keeping its memory usage constant.

The aggregation in question is a histogram aggregation. A histogram requires storing a fixed-size list of bins and corresponding counts, regardless of the volume of data. This means that for ever-growing volumes of data our memory consumption stays the same.

However, maintaining a sliding window histogram in real-time is a harder challenge. Traditionally, sliding window maintenance requires evicting the oldest element from the window when a new one is inserted and updating the aggregation state. We could maintain the histogram in constant time using a Subtract-On-Evict (SOE) algorithm (Algorithm 2): for each new event, we would evict the oldest one and decrement the bin it belonged to and insert the new event in the window while incrementing the bin it belongs to. However, using this approach, we violate our sublinear memory consumption, because now we have to keep the entire window contents in memory, hence we grow linearly regarding sliding window size. We instead propose an EMA-like approximate histogram aggregation, in Section 5.3.1. An EMA-like histogram has constant memory complexity and can be maintained in real-time.

To summarize, EMA-like approximate histogram aggregations, defined in Section 5.3.1, have constant memory and time complexity and can be maintained in a sliding window fashion.

(RQ3) How can we measure divergence between the reference and sliding window aggregations?

Statistical tests that measure the similarity between two probability distributions are common in the literature. Tests like Kolmogorov–Smirnov, Wasserstein or Jensen–Shannon (the one we used) provide a bounded measure of distance between two probability distributions, or histograms.

In brief, since the aggregation we use is a histogram, we can use statistical tests that measure the similarity between probability distributions. We chose to use the Jensen–Shannon Divergence (JSD). The JSD takes as input two histograms and outputs a value between 0 and 1. The closer this value is to 0 the more similar the distributions are. On the other hand, a JSD value of 1 represents two different distributions. We chose to use the JSD metric due to its simplicity and leave as future work the usage of other statistical tests.

(RQ4) Based on a divergence measure, when should we raise an alert?

Statistical divergence tests provide fixed bounds for the measurement value. In the case of the Jensen–Shannon Divergence (JSD), the output ranges from 0 to 1. This research question poses the following challenge: how can we threshold a JSD value? In other words, do we alert a feature as divergent if the JSD measure is above 0.6? Or if above 0.99? How can we threshold this value?

In Section 5.3.2, we explain how we define the threshold and hence define when we raise an alert. In our batch phase, we make several samples and measure the JSD distance between the histogram aggregations of those samples and our reference histogram aggregation. We end up with a list of JSD values. Given a new JSD value, we check against the now known distribution of JSD values and alert if above a certain percentile. Note that defining a threshold percentile is

easier because we are defining a probability: alerting above the 99th percentile means alerting for hypotheses with 1% probability.

In brief, we build the distribution of divergence values in batch and use it to compare a future divergence value and find out its probability. If said divergence value has a very low probability (*e.g.*, 1% or less), then we raise an alert.

Hypothesis Discussion

In our synthetic data experiments, we were able to detect all of the introduced anomalies. Additionally, we were able to do it achieving very high streaming throughputs. In the real data experiments, once again, we managed to achieve high throughput but the produced alerts were not very accurate. We defer further investigation to future work because of the time constraints of this thesis but leave open the possibilities that our divergence measure distribution was poorly built and hence more samples need to be made in the batch phase or that the approximate EMA histogram aggregation value is more inaccurate than what we thought it would be since the EMA tail holds all past events and higher EMA smoothing factors should be used.

Overall, our set of tests supports the claim that sliding window aggregations and outlier detection methods can indeed be combined to detect data pattern shifts in streaming scenarios, with constant time and memory complexity.

7.2 Contributions

We summarize our contributions as follows:

1. a two-phased and two-windowed univariate subsequence outlier detection method, classified according to the taxonomy in Section 3.1.2 and detailed in Chapter 5, that is lightweight and works for real-time stream monitoring

2. a histogram aggregation based on Exponential Moving Averages, that replaces the entire sliding window contents by a lighter representation, resulting in a constant in time and space complexity sliding window aggregation

3. a set of synthetic datasets and experiments where the method in question accurately detected anomalies

4. experiments on real data that showed unexpected results but where we provide insights and possible hypotheses to test for future work

7.3 Future Work

First and foremost, we believe the next steps are to reason about the obtained results in the real data experiments and find the root cause for the existence of many and large alert periods. As a

possible hypothesis to test, we suggest testing with EMAs with higher smoothing factors, so that past events are forgotten faster; additionally, we suggest decreasing γ, which further increases the number of samples to make in the batch phase, improving the accuracy of the divergence measure distribution built.

In this thesis, we used the Jensen–Shannon Divergence (JSD) as a statistical test to measure the distance or divergence between our reference and target sliding window aggregation. We chose to use the JSD metric due to its simplicity and because it was used in previous work successfully [41]. We believe that future work comprises testing new distances in the literature. In particular, we believe the Wasserstein distance is worth trying next, and illustrate why with Figure 7.1 and Table 7.1. The Wasserstein distance measures the work required to transport the probability mass

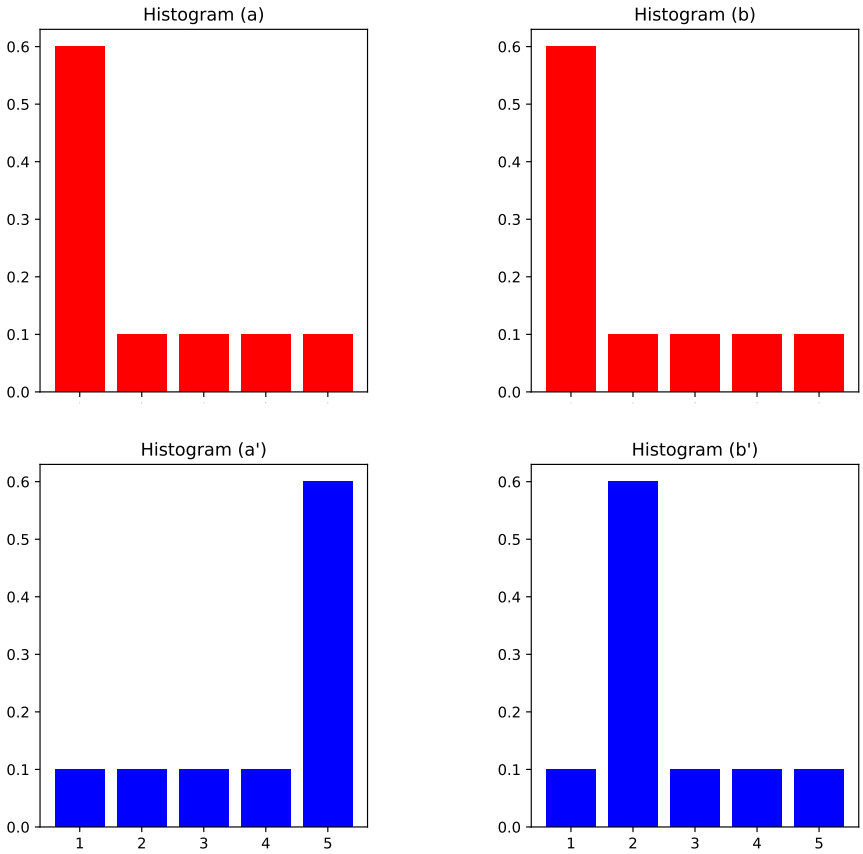


Figure 7.1: Wassertein vs Jensen–Shannon distances

from one histogram to another. This measure will be larger the further way the probability mass is. In Figure 7.1, we show two pairs of histograms, *Histogram (a)* paired with *Histogram (a')* and

Distribution #1	Distribution #2	Wasserstein Output	Jensen–Shannon Output
a	a'	2.0	0.4451
b	b'	0.5	0.4451

Table 7.1: Wasserstein and Jensen-Shannon divergences for both distribution pairs

Histogram (b) paired with *Histogram (b')*. Table 7.1 shows the output values for both pairs by each divergence function. The JSD measure is the same for both cases, whereas the Wasserstein distance is larger for the first pair because the probability mass has to be transported further.

In this thesis, we had to deal with the multiple test problem and apply a multiple test correction. In our case, we used Holm-Bonferroni due to its simplicity and because it controls the family-wise error rate with larger power than some other methods (thus providing a good trade-off between simplicity and power). There are many other multiple test correction methods, and we leave the testing of each as future work.

Besides testing new methods for distribution divergence measurement or multiple test correction methods, we have a set of parameters to work with. Experiments where only one of the parameters is changed should be carried out. Our method has lots of parameters to fine-tune and experiment with, as defined in Section 5.3.2, such as the tuple-based half-life, the size of the samples we make, the γ probability (which defines the probability of not having at least one sample divergence measure in the upper tail of the distribution) and the Family-Wise Error Rate.

Lastly, we propose another type of validation method. We mentioned in the beginning that our objective is to detect data pattern shifts before the performance of the systems that rely on the data declines. We propose a test where we train a Machine Learning model on a reference period and measure its prediction accuracy on a target period. Then, we propose running our method on reference and target periods to find out timestamps with a high number of divergent features. We propose that the model is retrained at these timestamps (tantamount to system reconfiguration) and then used to predict for the remaining target period. Finally, we propose that the accuracy should be compared to the previously observed model performance, expecting it to be higher after retraining in one of the high divergent timestamps.

References

[1] A. Mavragani, G. Ochoa, and K. P. Tsagarakis, "Assessing the methods, tools, and statistical approaches in google trends research: Systematic review," *J Med Internet Res*, vol. 20, p. e270, Nov 2018.

[2] J. Gama, *Knowledge Discovery from Data Streams*. Chapman & Hall/CRC, 1st ed., 2010.

[3] T. Kolajo, O. Daramola, and A. Adebiyi, "Big data stream analysis: a systematic literature review," *Journal of Big Data*, vol. 6, no. 1, 2019.

[4] A. Tsymbal, "The problem of concept drift: Definitions and related work," 05 2004.

[5] J. a. Gama, I. Žliobaitundefined, A. Bifet, M. Pechenizkiy, and A. Bouchachia, "A survey on concept drift adaptation," *ACM Comput. Surv.*, vol. 46, Mar. 2014.

[6] N. Martin, M. Swennen, B. Depaire, M. Jans, A. Caris, and K. Vanhoof, "Batch processing: Definition and event log identification," *CEUR Workshop Proceedings*, vol. 1527, pp. 137–140, 2015.

[7] D. Borthakur, "The hadoop distributed file system: Architecture and design," *Hadoop Project Website*, vol. 11, no. 2007, p. 21, 2007.

[8] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–10, 2010.

[9] https://hadoop.apache.org/, 2020. Online; accessed 17 June 2020.

[10] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, p. 107–113, Jan. 2008.

[11] https://spark.apache.org/, 2020. Online; accessed 17 June 2020.

[12] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, (USA), p. 10, USENIX Association, 2010.

[13] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, (San Jose, CA), pp. 15–28, USENIX, 2012.

[14] https://spark.apache.org/docs/latest/streaming-programming-guide.html, 2020. Online; accessed 17 June 2020.

[15] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, (New York, NY, USA), p. 423–438, Association for Computing Machinery, 2013.

[16] "Batch is a special case of streaming." https://www.ververica.com/blog/batch-is-a-special-case-of-streaming. Accessed: 2020-01-30.

[17] I. Botan, R. Derakhshan, N. Dindar, L. Haas, R. J. Miller, and N. Tatbul, "SECRET: A model for analysis of the execution semantics of stream processing systems," *Proceedings of the VLDB Endowment*, vol. 3, no. 1, pp. 232–243, 2010.

[18] https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/stream/package-summary.html#Reduction, 2020. Online; accessed 17 June 2020.

[19] K. Tangwongsan, M. Hirzel, and S. Schneider, "Sliding-Window Aggregation Algorithms," *Encyclopedia of Big Data Technologies*, pp. 1516–1521, 2019.

[20] https://flink.apache.org, 2020. Online; accessed 17 June 2020.

[21] J. E. Everett, "The exponentially weighted moving average applied to the control and monitoring of varying sample sizes," *WIT Transactions on Modelling and Simulation*, vol. 51, pp. 3–13, 2011.

[22] J. S. Hunter, "The Exponentially Weighted Moving Average," *Journal of Quality Technology*, vol. 18, no. 4, pp. 203–210, 1986.

[23] M. B. Perry, *The Exponentially Weighted Moving Average*. American Cancer Society, 2011.

[24] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu, "A framework for clustering evolving data streams," in *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, p. 81–92, VLDB Endowment, 2003.

[25] G. Hulten, L. Spencer, and P. Domingos, "Mining time-changing data streams," in *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '01, (New York, NY, USA), p. 97–106, Association for Computing Machinery, 2001.

[26] A. Blázquez-García, A. Conde, U. Mori, and J. A. Lozano, "A review on outlier/anomaly detection in time series data," *ArXiv*, 2020.

[27] C. C. Aggarwal, *Outlier Analysis*. Springer Publishing Company, Incorporated, 2nd ed., 2016.

[28] H. Aguinis, R. K. Gottfredson, and H. Joo, "Best-Practice Recommendations for Defining, Identifying, and Handling Outliers," *Organizational Research Methods*, vol. 16, no. 2, pp. 270–301, 2013.

[29] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Comput. Surv.*, vol. 41, July 2009.

[30] V. J. Hodge and J. Austin, "A survey of outlier detection methodologies," *Artificial Intelligence Review*, vol. 22, pp. 85–126, 2004.

[31] X. Xu, H. Liu, and M. Yao, "Recent progress of anomaly detection," *Complexity*, vol. 2019, pp. 2686378:1–2686378:11, 2019.

[32] D. Hawkins, *Identification of outliers*. Monographs on applied probability and statistics, London [u.a.]: Chapman and Hall, 1980.

[33] H. Anton, *Elementary Linear Algebra*. John Wiley & Sons, 7th ed., 1994.

[34] D. Yang, E. A. Rundensteiner, and M. O. Ward, "Neighbor-based pattern detection for windows over streaming data," *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology, EDBT'09*, pp. 529–540, 2009.

[35] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, p. 226–231, AAAI Press, 1996.

[36] S. Ahmad and S. Purdy, "Real-Time Anomaly Detection for Streaming Analytics," *CoRR*, 2016.

[37] M. Munir, S. A. Siddiqui, A. Dengel, and S. Ahmed, "DeepAnT: A Deep Learning Approach for Unsupervised Anomaly Detection in Time Series," *IEEE Access*, vol. 7, pp. 1991–2005, 2019.

[38] D. Kifer, S. Ben-David, and J. Gehrke, "Detecting change in data streams," pp. 180–191, 04 2004.

[39] Y. Dodge, *Kolmogorov–Smirnov Test*, pp. 283–287. New York, NY: Springer New York, 2008.

[40] D. Rey and M. Neuhäuser, *Wilcoxon-Signed-Rank Test*, pp. 1658–1659. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

[41] F. Pinto, M. O. P. Sampaio, and P. Bizarro, "Automatic model monitoring for data streams," 2019.

[42] F. J. M. Jr., "The kolmogorov-smirnov test for goodness of fit," *Journal of the American Statistical Association*, vol. 46, no. 253, pp. 68–78, 1951.

[43] N. H. Kuiper, "Tests concerning random points on a circle," 1960.

[44] T. W. Anderson and D. A. Darling, "A test of goodness of fit," *Journal of the American Statistical Association*, vol. 49, no. 268, pp. 765–769, 1954.

[45] J. Lin, "Divergence measures based on the shannon entropy," *IEEE Transactions on Information Theory*, vol. 37, no. 1, pp. 145–151, 1991.

[46] K. Tangwongsan, M. Hirzel, and S. Schneider, "Low-latency sliding-window aggregation in worst-case constant time," *DEBS 2017 - Proceedings of the 11th ACM International Conference on Distributed Event-Based Systems*, pp. 66–77, 2017.

[47] A. Singh, S. Garg, R. Kaur, S. Batra, N. Kumar, and A. Y. Zomaya, "Probabilistic data structures for big data analytics: A comprehensive review," *Knowledge-Based Systems*, p. 104987, 2019.

[48] Burton H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," *ACM SIGDA Newsletter*, vol. 8, no. 1, pp. 6–11, 1978.

[49] J. K. Mullin, "A second look at bloom filters," *Commun. ACM*, vol. 26, p. 570–571, Aug. 1983.

[50] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The Count-Min Sketch and its applications," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 2976, pp. 29–38, 2004.

[51] P. Flajolet and G. Nigel Martin, "Probabilistic counting algorithms for data base applications," *Journal of Computer and System Sciences*, vol. 31, no. 2, pp. 182–209, 1985.

[52] M. Durand and P. Flajolet, "LogLog Counting of Large Cardinalities," *inproceedings*, pp. 1–14, 2007.

[53] P. Flajolet, "HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm," *2007 Conference on Analysis ofAlgorithms, AofA 07*, pp. 127–146, 2001.

[54] Y. Chabchoub and G. Hébrail, "Sliding HyperLogLog: Estimating cardinality in a data stream over a sliding window," *Proceedings - IEEE International Conference on Data Mining, ICDM*, pp. 1297–1303, 2010.

[55] J. Xu, "Distributed super point cardinality estimation under sliding time window for high speed network," *CoRR*, vol. abs/1807.01527, 2018.

[56] A. Shtul, C. Baquero, and P. S. Almeida, "Age-partitioned bloom filters," 2020.

[57] N. Rivetti, Y. Busnel, and A. Mostéfaoui, "Efficiently summarizing data streams over sliding windows," in *2015 IEEE 14th International Symposium on Network Computing and Applications*, pp. 151–158, 2015.

[58] M. Hazewinkel, *The Encyclopaedia of Mathematics*, vol. 5, pp. 102–140. 01 2000.

[59] M. Aickin and H. Gensler, "Adjusting for multiple testing when reporting research results: the bonferroni vs holm methods.," *American journal of public health*, vol. 86 5, pp. 726–8, 1996.

[60] T. Dickhaus, "Simultaneous statistical inference," in *Springer Berlin Heidelberg*, 2014.

[61] R. G. Miller, "Simultaneous statistical inference," 1966.

[62] G. I. Webb and F. Petitjean, "A multiple test correction for streams and cascades of statistical hypothesis tests," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, (New York, NY, USA), p. 1255–1264, Association for Computing Machinery, 2016.

[63] A. Neumann, T. Bodnar, D. Pfeifer, and T. Dickhaus, "Multivariate multiple test procedures based on nonparametric copula estimation.," *Biometrical journal. Biometrische Zeitschrift*, vol. 61 1, pp. 40–61, 2019.

[64] A. C. Tamhane and J. Gou, "Advances in p-value based multiple test procedures," *Journal of Biopharmaceutical Statistics*, vol. 28, pp. 10 – 27, 2018.

[65] S. Holm, "A simple sequentially rejective multiple test procedure," 1979.

[66] P. Domingos, "A few useful things to know about machine learning," *Commun. ACM*, vol. 55, p. 78–87, Oct. 2012.

[67] K. Potdar, T. S. Pardawala, and C. D. Pai, "A comparative study of categorical variable encoding techniques for neural network classifiers," *International Journal of Computer Applications*, vol. 175, pp. 7–9, 2017.