UNIVERSITY OF OKLAHOMA
GRADUATE COLLEGE

OUTLIER EXPLANATIONS FOR DATA STREAMS: OUTLYING ATTRIBUTES AND ROOT CAUSE FACTORS OF OUTLIERS

A DISSERTATION
SUBMITTED TO THE GRADUATE FACULTY
in partial fulfillment of the requirements for the Degree of
DOCTOR OF PHILOSOPHY

By
EGAWATI PANJEI
Norman, Oklahoma
2024

OUTLIER EXPLANATIONS FOR DATA STREAMS: OUTLYING ATTRIBUTES AND ROOT CAUSE FACTORS OF OUTLIERS

A DISSERTATION APPROVED FOR
THE SCHOOL OF COMPUTER SCIENCE

BY THE COMMITTEE CONSISTING OF

Dr. Le Gruenwald, Chair

Dr. Qi Cheng

Dr. Sanjana Mudduluru

Dr. Theodore Trafalis

Acknowledgements

I would like to express my gratitude to Dr. Le Gruenwald for her unwavering support and insightful guidance. I am grateful for her constructive feedback and the countless hours she spent reviewing my work, which significantly improved the quality of this research. Beyond her academic support, her caring nature and understanding have made this journey more manageable and fulfilling. She has always been there to offer a wise counsel, for which I am deeply thankful.

Also, I would like to thank Dr. Qi Cheng, Dr. Theodore Trafalis, and Dr. Sanjana Mudduluru for their support. I extend my sincere appreciation to them for taking the time to serve in my dissertation committee.

I am grateful for the University of Oklahoma as well as the School of Computer Science students, staffs and faculties. I am thankful for Dr. Sridhar Radhakrishnan for his support at the beginning of my PhD journey, granting me an opportunity to be a graduate teaching assistant throughout these years.

To Dortje Tandirura, Carlos Sanchez, Von and Marge Worten, cousins, nephews, nieces, aunts, uncles, and friends, thank you for your prayers, support, and belief in me. Your love and understanding fuel me on this journey.

To God, be the glory, great things He has done.

Table of contents

List of Tables

List of Figures

List of Symbols

Symbol	Interpretation
$\mathbb{S} = \{S_1, S_2, \dots, S_m\}$	A set of m data streams where each number of {1, 2, ...m} also represents each stream's ID
m	Total number of data streams
S_j	A data stream j
TW	Tumbling Window
TW_j	An active tumbling window of a data stream j
tws	Tumbling window size
T_p	A starting time of the p-th tumbling window
T_{p+1}	An ending time of the p-th tumbling window
d_j	Total number of attributes in a data stream j
X	An attribute or a variable
$\mathcal{D}_j = \{X_{j1}, X_{j2}, \dots, X_{jd_j}\}$	A set of attributes in a data stream j
D	Total number of attributes of all m data streams, $D = d_1 + d_2 + \cdots + d_m$
$\mathbf{o}_i^j$	An outlier data point i in data stream j, $\mathbf{o}_i^j \in \mathbb{R}^{d_j}$
$\widehat{\mathbf{o}}_i^j$	Estimated normal values of $\mathbf{o}_i^j$
$\mathbf{O^j}$	All outliers found in S_j with timestamps within the start and end time of TW_j. $\mathbf{O^j}$ is a matrix whose columns represent the attribute values and rows represent the outlier data points. The first and the last rows represent the oldest and the latest outliers in TW_j respectively.
$\widehat{\mathbf{O}}^{\mathbf{j}}$	A matrix representing estimated normal values of $\mathbf{O^j}$
$\mathbf{Q}$	A matrix of size D × k, representing k-eigenvectors, $\mathbf{Q} \in \mathbb{R}^{D \times k}$
$\mathbf{Q_j}$	A segment of $\mathbf{Q}$ that is associated with S_j. $\mathbf{Q_j}$ is a matrix of size $d_j \times k$.
$\mathbf{Z}$	A matrix of size D × k, used to update $\mathbf{Q}$
$\mathbf{x_{aligned}}$	A tuple of D combined values of data points from all streams that share the same timestamp, $\mathbf{x_{aligned}} \in \mathbb{R}^D$
AlignedData	A key value set where a key represents a timestamp, and its corresponding value represents $\mathbf{x_{aligned}}$
$\mathbb{C}_j = \{C_1, \dots, C_{l_j}\}$	A set of lj clusters linked to data stream j
lj	Number of clusters in a data stream j

num_points_{aj}	The number of data points absorbed by a cluster $C_a \in \mathbb{C}_j$
$\mathbf{centroid_{aj}}$	The centroid of a cluster $C_a \in \mathbb{C}_j$, $\mathbf{centroid_{aj}} \in \mathbb{R}^{d_j}$
Est	The estimator component of EXOS
$\mathcal{C}f_j$	An independent function j that clusters data in the active tumbling window at stream j
$\mathcal{G}_j$	An independent function j associated with stream j that forms a local context of each outlier based on the output of Est and $\mathcal{C}f_j$ and then computes the outlying contribution score of the outlier's attributes
N_j	The number of data points in the active tumbling window of data stream j (TW_j)
$\lvert \mathbf{O^j} \rvert$	The number of outliers in TW_j
$n_{samples}$	The maximum number of auxiliary data points generated for the inlier or outlier classes
I	The maximum number of iterations required to initialize clusters of each stream
nepochs	The number of iterations to train a linear SVM using a gradient descent algorithm
G	A causal graph represented by a set of vertices V and a set of edges E
V	A set of vertices or variables, e.g $V= \{X_1, X_2, ..., X_d\}$
E	A set of edges in G, e.g $E=\{(X_1, X_j), ..., (X_j, X_d)\}$
|V|	The total number of vertices or variables in the causal graph
|E|	The total number of edges in the causal graph
X_b	A target vertex in G
$P(X_i)$	A parent of vertex X_i
$\lvert P(X_i) \rvert$	The number of parents of X_i
nroots	the number of root vertices
NT_i	A noise term variable corresponding to vertex X_i
$A(X_b)$	A list whose members consists of X_b 's ancestors and X_b itself
$\lvert A(X_b) \rvert$	The size of $A(X_b)$
$f_{X_i}(P(X_i), NT_i)$	A functional causal model at vertex X_i
FCM	Functional Causal Model, representing all $f_{X_i}(P(X_i), N_i)$ of a causal graph
g(.)	An outlier scoring function
SW	Sliding window

SW^T	A current sliding window
ΔT	A sliding window size in given time unit
δT	A slide size in given time unit
|W|	The number of data points in a sliding window
|slide|	The number of data points in a Slide
M	The number of samples in to generate from each fitted functional causal model
AFDS	Active FCM related Data Structure
max_id,	The maximum number of AFDS' IDs
Active FCM	The first part of AFDS represented as a hash map with keys and values correspond to slide IDs and time period respectively
Models	The second part of AFDS represented as a hash map with keys correspond to slide IDs and each slide ID points to a set of functional causal model of each vertex in G respectively
NT Samples	The third part of AFDS represented as a hash map with keys correspond to slide IDs and each slide ID points to a set of M noise term values of each vertex in G respectively
Scorers	The fourth part of AFDS represented as a hash map with keys and values correspond to slide IDs and fitted outlier scoring functions respectively
TargetRMSEs	A key value set in which a key represents an ID of an FCM, and the key's corresponding value represents the root mean squared error (RMSE) of prediction function at a target vertex X_b
n_o	The number of outliers obtained from the outlier queue when a window slide
NT	A matrix of size $M \times \| A(X_b)\|$ storing M noise term values of $\| A(X_b)\|$ variables
Iter	The number of iterations to train functional causal model
NDistTypes	the number of possible data distribution types to represent a noise term of a vertex
kneighbor	the number of neighbors used in KL divergence computation

Abstract

Data streams, which are continuous sequences of timestamped data points, necessitate real-time monitoring due to their time-sensitive nature. They have special characteristics such as the notion of infinity (continuous arrival of data points and unbounded volume of data) and concept drift. Detecting outliers which are data points significantly different from the rest of the data in the dataset, is crucial in many data stream applications. For example, in network security and credit card transaction monitoring, real-time detection of outliers is vital, as these outliers often signify potential threats. However, investigating detected outliers usually requires significant time and effort from the users. Therefore, providing real-time outlier explanations is equally important, as it enables users to gain insights and shorten their investigation time. When an outlier is detected as a multidimensional object, the investigation time of the outlier is equivalent to the number of outlier attributes. Hence providing an outlier explanation in the form of the set of attributes responsible for the outlier abnormality (also known as the outlying attributes) is necessary. In some applications such as cloud monitoring, expert users spend considerable time and effort to investigate the root cause factors of an outlier detected in the front-end service. The investigation of the root cause factors involves examining the front-service all the way to the backend services. Providing an outlier explanation in the form of root cause factors of the outlier is important to minimize user effort in identifying the reasons for their occurrences. There exist techniques that discover outlying attributes and root causes factors of outliers for data streams. However, they do not simultaneously address the characteristics of data streams, especially for those involving the notion of infinity and concept drift.

This dissertation proposes two outlier explanation algorithms, EXOS and Ocular, for discovering outlying attributes and root cause factors of outliers, respectively. EXOS is designed for discovering outlying attributes of multi-dimensional outliers in data streams. Unlike other existing techniques, EXOS leverages cross-correlations among data streams, accommodates varying data stream schemas and arrival rates, and effectively addresses challenges related to the unbounded volume of data and concept drift. The algorithm provides real-time explanations based on the local context of the outlier, derived from a time-based tumbling window. Ocular is an algorithm designed to identify root cause factors of point outliers in continuous real-time data streams. It utilizes a user-provided normal causal graph, which depicts the causal relationships or dependencies between variables in a system. When the value of a variable at a particular timestamp is detected as an outlier, Ocular employs this causal graph to identify the variables responsible for the anomalous value of the target variable. The algorithm simultaneously addresses inherent characteristics of data streams: the notion of time, the notion of infinity, and concept drift.

Extensive theoretical and empirical analyses have been conducted to evaluate the performance of EXOS and Ocular using both real and synthetic datasets. The evaluation results show that, on average, EXOS achieves a 45.6% better F1 Score and is 7.3 times lower in explanation time compared to existing outlying attribute algorithms. Additionally, Ocular outperforms current root cause identification algorithms by 170% in F1 Score on average, while maintaining comparable or lower explanation times.

CHAPTER 1

Introduction

Data streams are infinite sequences of data, each with an associated timestamp [1–3]. Data streams are time-sensitive and demand real-time monitoring. To illustrate, consider financial trading applications, where stock prices constantly fluctuate, rendering the value of a stock at a specific time critical, yet it quickly becomes outdated. This example underscores how data within streams are fleeting, possessing relevance for only a finite period of time before being either discarded or archived. In streaming environments, data flows continuously from various sources, resulting in an unbounded volume of data for analysis also known as the notion of infinity. Moreover, in many stream applications, there exist concept drifts where the underlying data distribution changes over time, further complicates data stream analysis [4–8].

An outlier in a dataset is a data point that has a significantly different value compared to other data points in the dataset [9]. The increasing demand for real-time analytics has created a need to apply real-time outlier detection over data streams [10, 11]. The ability to identify outliers in data streams is incredibly important as they often indicate problems that require further exploration. For example, detected outliers in sensor data could point towards equipment failure, while spotting them in financial transactions might suggest fraudulent activities. In these instances, it falls upon users to investigate the identified outliers in order to take appropriate actions. The investigation process can be expedited when the detected outliers are accompanied by explanations or insights that shed light on the reasons behind their occurrences [12–14]. However, providing explanations for

outliers in data streams is challenging due to the special characteristics of data streams. These characteristics include their infinite nature and the presence of concept drift, which causes the underlying data distribution to change over time. These factors complicate the analysis and necessitate advanced techniques to ensure accurate and timely explanations.

In some data stream applications such as Structural Health Monitoring (SHM) [15, 16], an outlier is often detected as a multidimensional object. Siddique et al. [12] mention that the investigative work of the analyst is roughly related to the number of outlier's attributes. If each outlier has an explanation as to which attributes are responsible for its detection, the analysts can verify the finding faster and take further action. This explanation is also known as *outlying attributes.* Furthermore, in some data stream applications, users often spend hours uncovering the *root cause factors* behind the occurrence of outliers [13]. For example, in microservice monitoring, when an unusual response time at the front-end service is detected as an outlier, Software Reliability Engineers (SREs) can take hours to analyze not only the front-end service but also all connected backend services to identify which service is causing the outlier at the front-end service [13]. An anomalous CPU utilization at one of the backend services propagates its effect to the front-end service and affect its response time. The service with unusual CPU utilization is deemed the root cause of the outlier in the front-end service. Clearly, providing these types of outlier explanations can help analysts speed up their investigation process. However, not much research has been conducted to discover outlier explanations in data streams, particularly those that identify outlying attributes or root cause factors of point outliers while also managing the continuous arrival and unbounded volume of data streams, as well as concept drift. To fill

this gap, we propose EXOS and Ocular, which respectively discover outlying attributes and root cause factors of outliers in data streams.

EXOS generates outlying attributes in multi-dimensional data streams. It considers the potential data correlation within a data stream and across data streams to estimate a local context of each outlier while handling different data arrival rates among the data streams. It forms the local context incrementally while handling both the notion of infinity and concept drift. Once the local context is formed, EXOS use the hyperplane that separate the local context from the outlier to determine which attributes responsible for the abnormality of the outlier. On the other hand, Ocular finds the root cause factors of a variable's anomalous value with particular timestamp in each data stream independently; thus, it does not need to be concerned about whether data streams have different arrival rates. Ocular employs a user given normal causal graph to find outliers' root causes. The vertices in the causal graph represent variables or attributes of the system where the outliers are detected while the directed edges denote causal relationships (with the source vertex causing changes in the destination vertex). Ocular utilizes a time-based sliding window to deal with the continuous arrival of data points. It provides a mechanism to update the model associated with the vertices in the causal graph only when detecting changes in data distribution.

In this chapter, we first provide the background information on data streams and outliers. Next, we describe outlier explanations in terms of outlying attributes and outlier root causes and their role in data stream applications. We then discuss the research challenges in developing algorithms that can generate these types of outlier explanations

in data streams. We finally state our research objectives and contributions and present the dissertation outline.

1.1 Data Streams

Data streams are infinite sequences of data points, each has a timestamp (referred to as *the notion of time*) [1, 3, 17] indicating the time when the data point is either collected or recorded. A Wireless Sensor Network (WSN) [18] is a widely used application of data streams, comprising interconnected nodes that can self-configure. A WSN node commonly has three elements: a wireless radio transceiver, one or more sensors and a microprocessor. A transceiver allows communication between neighboring nodes in the WSN. Sensors within the node measures the attributes of the physical environment. For example, for environmental monitoring purpose, sensors measure wind speed, wind angle direction, temperature, and humidity, etc. The measurements along with the timestamps when they are collected are continuously sent to the node's microprocessor to process.

In streaming applications such as a WSN, data flows continuously from single or multiple sources (e.g., sensors), resulting in an unbounded volume of data for analysis. Data streams are thus said to have *the notion of infinity* [1]. Practical constraints such as memory limitations render it impractical to store this infinite amount of data in memory for processing purposes [19]. As data points arrive from various sources within a WSN, the arrival rate of each data stream can vary [1]. For instance, one sensor may omit data points every second while others may exhibit a different frequency such as every minute or every other minute. We refer to the data streams with the same arrival rates as *concurrent data streams* and the varying arrival rates as *non-concurrent data streams*.

Data stream sources such as sensors in a WSN, can produce *multi-dimensional data points*, with the type and number of attributes potentially varying across streams [1]. Moreover, these streams may exhibit *cross-correlation*, signifying the attributes in one stream correlate with those in other streams [18, 20]. For instance, measurements from a weather sensor in an area may correlate with measurements from other sensors in neighboring areas. Additionally, data distribution in a stream can change over time, a phenomenon known as *concept drift* [4–8]. For example, weather patterns can change, causing shifts in temperature, precipitation, humidity, and other meteorological attributes.

Network traffic management is another popular application of data streams, where network packet header information is monitored across multiple routers to analyze traffic flow patterns [21, 22]. For example, the backbone network of an Internet Service Provider (ISP) [22] monitors packet traces collected from various links in the network, such as customer links, which connect each customer's network to the ISP's network, and backbone links, which connect two routers within the ISP's backbone network. Each link produces streams of packet traces whose headers may include the following information: the IP address of the packet sender, the IP address of the packet destination, the unique identification of the packet, the length of the packet, and the time when the packet was recorded. Each packet header can be considered a multi-dimensional data point in this network monitoring system. Data points arrive continuously at highly variable rates and can only remain in memory for a limited amount of time before they are replaced with new incoming data points.

1.2 Outliers

Hawkins [23] defines an outlier as "an observation which deviates so much from other observations as to arouse suspicions that it was generated by a different mechanism." Outliers are “patterns in data that do not conform to a well-defined notion of normal behavior” [9]. Outliers are also called anomalies, abnormalities, aberrations, contaminants, deviants, discordant observations, exceptions, peculiarities, or surprises in some applications [9, 24]. Outliers can be identified in the datasets by applying outlier detection algorithms. Chandola et al. [9] categorize outliers into three types:

- *point outliers*

  A data point or instance that deviates significantly from the rest of the data is termed a point outlier. For example, the two points x_1 and x_2 in Figure 1-1 are the examples of point outliers. This dissertation focusses on this outlier category.

- *collective outliers*

  A collective outlier is the collection of anomalous data points where the individual data points may not be anomalous by themselves.

- *contextual outliers*

  A contextual outlier refers to a data point that is anomalous in a specific context (but not otherwise). For example, 100 F temperature in Oklahoma in February is an outlier but not in August.

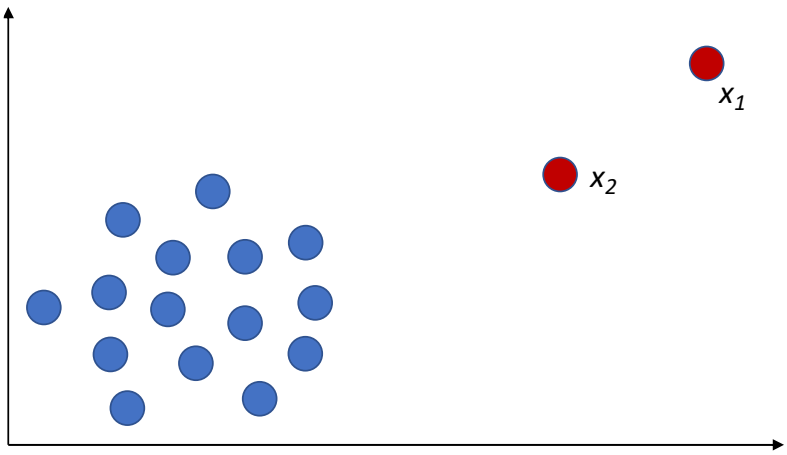


Figure 1-1 Example of two-point outliers in a dataset

1.3 Outlier Explanations

For applications to benefit more from the results of the outlier detection process, the results should be explainable. To this end, the process should include two tasks: outlier detection and outlier explanation. The Merriam-Webster dictionary defines explanation as “the act or process of explaining." To explain is “to make known" or “to give the reason for or cause of" or “to make something plain or understandable." In [25], Miller argues that “explainable artificial intelligence can benefit from existing models of how people define, generate, select, present, and evaluate explanation.” In the context of outlier detection, the outlier explanation task provides guidance for users in investigating detected outliers.

In this dissertation, we focus on two kind of outlier explanations: outlying attributes of outliers and root cause factors of outliers.

1.3.1 Outlying Attributes of Outliers

Outlying attributes of an outlier refer to all members of a feature subspace where the outlier is highly deviated from the normal data or, in other words, they refer to the attributes

that contribute the most to the abnormality of the outliers. There are two major interpretations of this outlier explanation [14]. Some algorithms [12] interpret outlying attributes as the attributes in the smallest subspace where the outlier score is greater than a threshold, while others [26, 27] interpret them as all members of a non-empty set of attributes where each member has an outlying contribution score higher than a threshold. We use the later interpretation to define outlying attributes in Chapter 3.

Imagine human analysts have to deal with tens or hundreds of attributes as in the case of e-commerce or healthcare systems. Whenever a data point is flagged as abnormal by an outlier detector, analysts need to manually go through the feature space to identify the subset of attributes responsible for the detection to verify whether the data object is a true outlier. Researchers in [12] state that the amount of effort that users need to investigate an outlier is roughly related to the number of attributes or features associated with the outlier. Knowing the subspace where an outlier stands out can reduce the amount of work/time analysts need to judge the status of the outlier. In addition, since less time is required to examine an outlier, analysts can review more flagged points during a time period.

1.3.2 Root Causes Factors of Outliers

Consider a system with multiple variables or attributes where each variable may causally affect others, meaning its value at time t affects the values of other variables after t. If a value of a variable X_b at a particular time is deemed to be an outlier, a root cause factor of the outlier is a variable X_a in the system that initiated unusual value or change and has a causal effect chain leading to X_b. A causal effect chain refers to a sequence of events where each subsequent event is directly cause by the preceding one. Note that the variable

X_b may have multiple causal effect chain related to it; hence its anomalous value at time t could be caused by more than one variable.

For instance, suppose a microservice monitoring system [28] gathers data on web response time (Web RT), queries per second of web users (Web QPS), middleware response time (Mid RT), queries per second of a middleware (Mid QPS), CPU utilization of an underlying container (CPU), and memory load (Mem Load). These variables are interconnected in a causal graph, as illustrated in Figure 1-3. For example, Web QPS affects CPU utilization, and CPU utilization in turn affects Web QPS.

Imagine a scenario where an anomalous value of Web RT is detected at a specific time t. Upon investigation, it is discovered that prior to this anomaly, there was an unusually high value of Web QPS. This elevated Web QPS directly influenced CPU utilization and memory load, which subsequently impacted Web RT. Since the anomalous Web RT was triggered by the unusually high Web QPS, Web QPS is identified as the root cause factor responsible for the outlier observed at time t in Web RT.

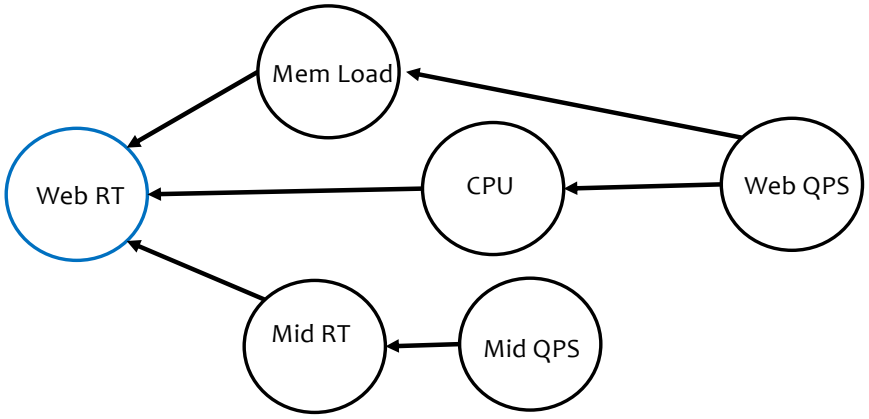


Figure 1-2 A causal graph of a microservice system

1.4 Data Stream Applications of Outlying Attributes and Root Cause Factors of Outliers

In this section, we discuss some data stream applications where the outlier explanations in terms of outlying *attributes* and *root cause factors of outliers* play an important role.

1.4.1 Intrusion Detection

Milenkoski et al. [29] state that "Intrusion detection is a common cyber security mechanism whose task is to detect malicious activities in the host or network environments." From a computer security perspective, a system/environment is considered secure if it has the properties of confidentiality, integrity, and availability of its data and services. Any attempts to violate these security properties are considered as attacks or intrusion. Outlier detection is applicable in the intrusion detection systems (IDS) because intrusions are different from the expected behaviors of the system [9].

Typically, an IDS needs to deal with a massive volume of data that arrives in streaming fashion. To stop ongoing attacks, the detection of malicious activities requires a timely reaction. An IDS typically deals with multi-dimensional data. For example, Avritzer et al. [30] monitor so-called performance signatures to trigger alerts of five types of security attacks: denial of service (DOS), SQL injection, man-in-the-middle (MITM), buffer overflow, and stack overflow. The performance signatures employ system usage-based attributes, such as CPU percentage, number of active threads, interface received bytes per sec, swap percentage, number of TCP connections established, number of TCP resets, interface transmitted bytes per second, virtual memory usage, working set in bytes, and memory percentage. The authors assume that "the performance of the well-behaved system

can be measured such that performance signatures of several types of attack can be identified." This assumption implies that when an alert has an explanation of the *outlying attributes* type that aligns with a known attack, the IDS can inform users of the type of the attack. However, when the *outlying attributes of an alert* do not fit any known attacks, analysts can use the information to either flag the alert as a false alarm or investigate it further to define a new type of attack.

1.4.2 Fake News Detection

Fake news is "news stories that are false: the story itself is fabricated, with no verifiable facts, sources or quotes" [31]. It is part of the larger environment of misinformation and disinformation. According to the Merriam-Webster dictionary, misinformation is "incorrect or misleading information," while disinformation is "false information deliberately and often covertly spread (as by the planting of rumors) in order to influence public opinion or obscure the truth." In recent years, fake news articles have increased through social media. It is very concerning, especially during the COVID-19 pandemic, because fake news can cost lives [32].

The spreading of fake news can be regarded as an anomalous behavior in social networks [33, 34] Fake news tends to have poor grammar, contains bad language, and refers to vague or untraceable sources. In addition, it is often posted by bogus accounts that use misleading names, images, or bogus web addresses [32]. These are some factors that social media users can use to spot fake news. They can also be used as input features for a fake news detection algorithm. For example, in [34], Li et al. categorize the factors into four feature types: (i) *text content features*, such as the number of positive sentiment words, whether the news contains question marks, etc.; (ii) *propagation features*, such as the

number of comments, number of likes, and number of retweets; (iii) *image feature* indicating whether the image in the article is tampered; and (iv) *user features*, which are features related to the users who publish the news, such as account age, follower-friend ratio, number of tweets, etc. These features are combined into a features vector used as input for an autoencoder. An autoencoder is an unsupervised neural network that has been used for anomaly detection in other applications [35, 36]. The autoencoder will determine whether a news article is fake or real. A fake news watch group or a community note group can use this information to warn people. However, the autoencoder is a black-box model [37]. To build trust, it is crucial to explain why a news article is flagged fake. Thus, we can use a decision tree-based explainer [38] to explain the black-box model. This technique generates an explanation of the *outlying attributes* type that tells which relevant features the autoencoder uses to flag a fake news article.

1.4.3 Fraud Detection

Fraud detection uncovers malicious or criminal activities in organizations such as credit card companies, banks, insurance companies, online auctions, telecommunication companies, etc. Outlier detection algorithms have been used widely for fraud detection, and some survey papers have a lengthy discussion about them [9, 39, 40], but can a fraud detection application benefit from the outlier explanation task?

Fraud detection, for example, credit card fraud detection, deals with millions of transactions every day, and most are legitimate transactions. However, it was reported by Javelin Institute [41] that in 2014, one in six legitimate cardholders experienced at least one decline because of suspected fraud. This false positive detection impacted merchants because following the decline, the customers reduced their patronage and even stopped

shopping with the merchants where their cards were declined. To mitigate this situation, most companies have human analysts monitoring all transaction alerts 24/7 [42].

A transaction is represented by a number of attributes, such as merchant-related attributes (unique id, bank of the merchant, type of the merchant, country), transaction-related attributes (amount, timestamp, currency, presence of a customer), terminal-related attributes (device type, how data is input into the terminal, service or not), etc. [42]. Fraud detection flags a transaction based on those attributes. Therefore, if an explanation about *the outlying attributes of a suspected fraudulent transaction* is made available, the analyst can make the decision faster.

1.4.4 Structural Health Monitoring

Structural Health Monitoring (SHM) is "the process of implementing a damage detection strategy for aerospace, civil or mechanical engineering infrastructure" [43]. Early detection of damages, such as cracks and corrosion, can reduce maintenance costs and prevent catastrophic events [44]. SHM combines sensor technologies and digital twins, i.e., virtual representations, to enable continuous observations of the structures of interest. For example, SHM for civil infrastructure (e.g., buildings, bridges, dams) employs hundreds of sensors monitoring environmental conditions, such as temperature, humidity, and wind speed. It also includes sensors monitoring structural response, such as acceleration, deflection, and strain [45].

Outlier detection algorithms have been applied in SHM [44, 45]. For example, Bigoni and Hesthaven [44] employ a one-class classifier outlier detection algorithm for each sensor so that it is possible to locate the damage on the structure of interest. They extract damage-sensitive engineering-based features from the raw signals generated by each

sensor and use them as inputs for the outlier detection algorithm. The detector will tell whether any subsequent data object belongs to a group of what is considered as healthy signals (inlier), or it is an outlier. Should the engineers be provided with the information of which features are responsible for the detection of an outlier (*the outlying attributes*), they can verify the finding faster and take action to fix the damage.

1.4.5 Cloud Service Monitoring

Nowadays, more and more business applications are being created directly on the cloud to leverage the numerous benefits of cloud-native systems, such as horizontal scalability and continuous delivery [13, 28, 46]. The cloud systems often use a microservice-oriented architecture to allow easy abstraction, reuse, and independent component scaling. The cloud service providers monitor the performance of their microservices to make sure they remain reliable overtime. A microservice failure can worsen user experience and bring economic loss [28].

Outlier detection has been applied to monitor unusual activity in the microservice system. An unusual response time at a front service may indicate a failure that is propagated from other services; however, locating the *root cause factors of anomalies* or incident has always been a difficult task for site reliability engineers (SREs) [46]. To illustrate, let us consider Figure 1-3 which presents a simplified example depicting relationships and dependencies among different components of a social media application, including the frontend service (Client App) and the rest of the backend services. Imagine a scenario where users encounter unusual delays when refreshing their social media feeds. In this case, these outliers may be attributed to latency issues within the feed generation service, media metadata service, and media storage service, as these services are interconnected.

However, the true causal explanation or root cause of these outliers might be traced back to a delay specifically within the media storage service. This delay, in turn, directly impacts the media metadata service, consequently, leading to unusual latency within the feed generation service, ultimately affecting the overall user experience. Techniques such as CloudRanger [13] and MicroCause [28] can be applied to find the *root cause factors of outliers* detected at the Client App.

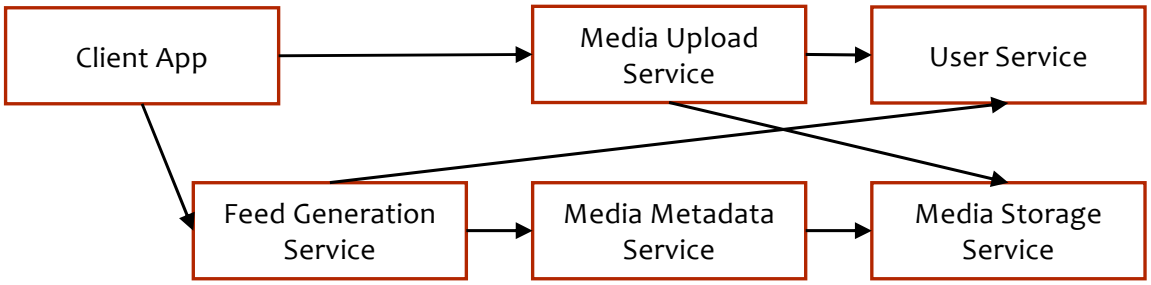


Figure 1-3 A toy example of social media microservices and their dependencies

1.5 Research Challenges

In this section, we first discuss the common research challenges that both types of outlier explanation algorithms, generating outlying attributes and generating outlier root cause factors, need to address due to the special characteristic of data streams. We then discuss the additional research challenges that are applied specifically to each of the two types of algorithms.

1.5.1 Research Challenges due to the Characteristics of Data Streams

a) Challenge DS 1: *Dealing with the notion of time*

Stream data have timestamps associated with them, providing a temporal context for each data point. Data points should be processed in the order of their timestamps. Furthermore, data points need to be processed based on their own temporal contexts. To generate outlier explanations, outliers must be compared with other data points that occurred within a time period semantically related to their timestamps. For example, if an outlier has a timestamp of 10 am, June 15, 2024, its explanation should come from a time period not far from 10 am, June 15, 2024. Furthermore, to determine the reason for its occurrence, the outlier needs to be compared with events that took place before that timestamp.

b) Challenge DS 2: *Dealing with the notion of infinity (unbounded volume of data)*

The notion of infinity in data streams implies that at any given time a random access to the entire data set is impossible. Due to memory limitation, only a fraction of the data points can be stored in memory for a period of time before being replaced by new incoming data points. Outlier explanation algorithms should be designed to process data points in the memory as quickly as possible. To do that limiting the number of passes on the data points and applying parallelism will help achieve this goal.

c) Challenge DS 3: *Dealing with different data arrival rates*

Data streams may have different arrival rates. For instance, one source of data streams can emit data points every one second while the other can emit data points

every five seconds. When processing data points, outlier explanation algorithms should be able to accommodate this case, especially when considering the cross-correlation among multiple streams to generate outlier explanations.

d) Challenge DS 4: *Dealing with concept drift*

The data distribution in a data stream can change over time because of changes in the environment, shifts in trend, etc. Hence, outlier explanation algorithms should not assume any fixed data distributions. When these algorithms use models fitted with some initial data, the algorithm should provide a mechanism to update the model over time to handle concept drift effectively.

1.5.2 Additional Research Challenges for Generating Outlying Attributes

a) Challenge OutAtt 1: *Limiting subspace search*

To find a subset of attributes responsible for the outlierness of an anomalous object, one can examine all the possible combinations of attributes and compute the object's outlier score in every attribute space. However, this approach is infeasible when the number of attributes is large. Some techniques limit the search space by using a heuristic approach; however, the approach does not guarantee to find the optimum subspace, which is the subspace where the outlier is the most anomalous [47, 48]. Instead of searching the subspace, some techniques [26, 27] depend on the local neighborhood of each outlier to extract its outlying attributes, and some other techniques [38, 49] use interpretable models to measure the contribution of each attribute to the abnormality of the object.

b) Challenge OutAtt 2: *Generating readily interpretable output*

In layman's terms, Occam's Razor principle [50] can be stated as "the simplest explanation is almost always the best." This principle implies that a technique that generates outlying attributes should minimize the number of features or attributes included in its output. Some existing techniques [12, 48] generate a list of outlying attributes, while some other techniques [49, 51] list all the original attributes and their corresponding contribution scores. The latter requires more effort from users to set a contribution threshold to determine the outlying attributes. Furthermore, even though some techniques [12, 48] produce a set of outlying attributes, users can further benefit if the explanation output also includes the outlying attributes' values. Of course, the output's description should be presented in a human-interpretable way.

1.5.3 Additional Research Challenges for Generating Root Cause Factors of Outliers

a) Challenge RC 1: *Providing counterfactual*

According to Pearl & Mackenzie [52], there are three levels on the ladder of causation: (i) association or correlation, (ii) intervention, and (iii) counterfactual. Association or correlation between variables suggests that there is a relationship between them. However, it does not indicate the direction or nature of the relationship. Intervention helps to explore the causal relationship more directly. Intervention means intervening or changing one variable while holding the other variables constant to observe the change it causes in other variables. Counterfactual is the highest level in the ladder of causality. It allows us to explore the question "What if things had been different." Providing counterfactuals when discovering

root cause factors of outliers will help us establish causality convincingly. However, defining the counterfactuals for outliers is not a trivial task. How one could assess what would have happened when the outliers did not take place or when the causal factors had different outcomes.

b) Challenge RC 2: *Incorporating user knowledge about variables' causal dependency*

Expert users, for example in microservice systems, understand the dependency between different services (variables). Leveraging this knowledge can enhance the identification of root causes for outliers detected within specific variables. We deal with the question how one can incorporate this expertise into the input of algorithms for effectively analyzing and addressing the root cause factors of anomalies within the system. What kind of the data structure should be used to represent the knowledge?

1.6 Research Objectives and Contributions

This research aims to develop effective and efficient outlier explanation algorithms tailored for data streams. We focus on designing algorithms for two types of outlier explanations:

1. outlying attributes of outliers
2. root cause factors of outliers

Below we describe our contributions.

Research Contribution 1. We propose EXOS, a real-time online outlier explanation algorithm identifying outlying attributes of multi-dimensional outliers in data streams. To

deal with the notion of infinity (Challenge DS 2), EXOS generates outlying attributes using time-based tumbling [53] where a memory buffer is used to store a sequence of timestamped data of each data stream in the main memory. When a specified time period expires, i.e., when the window slides, all the data stored in the buffers are replaced with the new arriving sequence of data. Whenever a new window arrives, EXOS applies single-pass incremental computation on the data in the window, allowing it to process the data fast.

EXOS is a local neighborhood-based outlier explanation technique. This approach allows EXOS to limit the subspace search (Challenge OutAtt 1). EXOS defines a local context for each outlier as a set of inlier neighbors and uses the local context to identify the outlier's outlying attributes. The local context is summarized from the tumbling window relevant to the outlier's timestamp (Challenge DS 1). The formation of the local context involves considering cross-correlations among data streams, with eigenvectors initially generated using offline data. EXOS handles the case when data points from different sources have different arrival rates (Challenge DS 3). To account for concept drift in data streams (Challenge DS 4), the eigenvectors are updated whenever new windows from all data streams are available. The update is done using a single-pass eigendecomposition. We conduct a time and space complexity analysis of EXOS. Moreover, we run experiments using real-world and synthetic datasets, comparing EXOS with existing algorithms in terms of average precision, recall, F1-score, and average explanation time. The experiments reveal that, on average, EXOS achieves up to an 86% higher average F1 score and 29.6 times lower explanation time than existing algorithms.

Research Contribution 2. We propose Ocular, an algorithm that provides the root cause factors for point outliers for a data stream in real time. Ocular finds the root cause factors of a point outlier based on the counterfactual question "*Would the data point not have been an outlier had we assigned rather 'normal' causal mechanisms at other variables of the system instead of the observed mechanism associated with the outlier?*" (Challenge RC 1). It incorporates user's knowledge of the normal causal dependency between variables in the system in the form of a directed acyclic graph (DAG) (Challenge RC 2). Ocular ensures that each outlier is explained using a model fitted with data points whose timestamps are earlier than the outlier's timestamp (Challenge DS 1). The model is initialized offline, but during the online process, the model is checked for necessary updates due to concept drift (Challenge DS 4). To handle the notion of infinity, Ocular utilizes a time-based sliding window where a group of data points remain in memory until a new group arrives (Challenge DS 2). Ocular is applied on an independent data stream; hence it does not have to account for different data stream arrival rates. It also employs parallel processing to ensure faster computation. We conduct a time and space complexity analysis of Ocular. Running experiments using synthetic and real-world datasets, we show that Ocular outperforms the state-of-the-art algorithms in terms of F1 score while achieving faster or similar average explanation time.

1.7 Organization of the Dissertation

The remainder of the dissertation is structured as follows. Chapter 2 reviews existing techniques for identifying outlying attributes and root cause factors of outliers. Chapter 3 presents EXOS, our proposed technique for discovering outlying attributes in data streams,

its complexity analysis, and experimental results comparing its performance with other existing algorithms. Chapter 4 presents Ocular, our proposed technique for identifying root cause factors of outliers in data streams, provides its complexity analysis, and experimental results comparing its performance with other existing algorithms. Finally, Chapter 5 provides conclusions and future research directions.

CHAPTER 2

Literature Review on Outlier Explanations

In this chapter, we present a review of the existing techniques that find *outlying attributes* of *outliers* (Section 2.1) and *root cause factors of outliers* (Section 2.1.3). In each section we discuss whether they address the research challenges described in Section 1.5, Chapter 1.

2.1 Techniques to Find Outlying Attributes of Outliers

The outlying attributes of outliers are those attributes in the outliers that are responsible for the abnormality of the outliers. There exist two types of techniques that find outlying attributes: one is for an individual outlier while the other is for a group of outliers. We review these two types of techniques in this section.

2.1.1 Techniques to Find Outlying Attributes of an Individual Outlier

Some of the techniques for finding the feature subspace contributing most to the abnormality of an individual outlier are outlier detection model-agnostic, and some are specific to the outlier detection model. These techniques fall into the following categories: (i) subspace search-based, (ii) local neighborhood-based, (iii) decision tree-based, (iv) layer-wise relevance propagation (LRP), (v) entropy-based reward, and (vi) game theory-based. We describe the techniques in Sections 2.1.1.1 - 2.1.1.6. The discussion on how the

techniques address the research challenges explained in Section 1.5 is given in Section 2.1.1.7.

2.1.1.1 Subspace Search-Based Techniques

The techniques in the subspace search-based category find the outlying attributes of each individual outlier by searching the subspace. To avoid examining all the possible subsets of attributes/features, two methodologies are introduced:

- *heuristic-based subspace search*

  This methodology limits the subspace search using a heuristic process that is not guaranteed to find an optimal subspace, yet sufficient enough. This includes bottom-up beam search that find the outlying attributes by first examining a single attribute and then proceeding to the higher dimension subspace.

- *branch and bound subspace search*

  This methodology finds the optimal outlying attributes by building a tree whose nodes represent the subset of attributes. In the worst case, it may require exploring all possible permutations of the subspace.

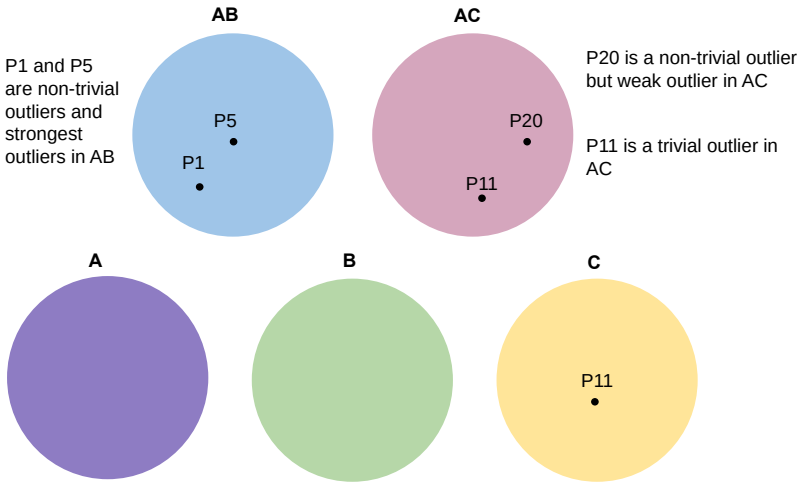


Figure 2-1 Illustration of strong, weak and trivial outliers when examining 3 attributes, A, B, and C

We now describe some techniques for the aforementioned methodologies.

Heuristic-based subspace search. Given a multidimensional data set, an object that is identified as an outlier in a sub-space may be considered as normal in another sub-space. Knorr and Ng [54] propose an algorithm that detects outliers in the subsets of attributes with lower cardinality and then examining the higher cardinality. To explain why an object is an outlier in a multidimensional data set, Knorr and Ng [54] propose to find the attribute spaces in which there is an outlier and label the outlier as strongest, weak, or trivial outlier. Figure 2-1 shows an illustration of strongest, weak, and trivial outliers in the 3D space {A, B, C}. P1 and P5 are non-trivial outliers in the subspace AB because they are not the outliers in the subspace A or the subspace B. They are also the strongest outliers in AB because there is no other anomalous point in the subspace A or B. P20 is a weak outlier in the subspace AC because there is another outlier point P11 in the subspace C. P11 is a trivial outlier in the subspace AC because it is also an outlier in the subspace C.

In other words, the process to find the outlying subspace of an outlier follows the heuristic process of finding where the outlier stands out as the strongest outlier. If the algorithm finds an object **o** as an outlier in the 3D subspace {A, B, C} and **o** is not an outlier in any of the subsets (A, B, C, AB, AC, and AB), then **o** is called the strongest outlier in ABC. Here, we find ABC as the outlying attributes of **o**. Thus, identifying the strongest outlier also means finding the outlying attributes of the outlier. This technique can be applied only to distance-based outliers.

Refout [55] randomly draws subspaces of dimensionality d1 without replacement and adds them into the subspace pool SP_1 until $|SP_1|$ reaches a threshold. It then applies an outlier detection model to all the subspaces in SP_1. Refout normalizes the outlier scores so that they are comparable among different subspaces. It saves the normalized scores for every object in every subspace and ranks all objects according to their maximum outlier scores over all the subspaces in SP_1. The top m ranked objects are then considered for subspace refinement. For each top ranked object, Refout extracts a set of outlier scores which is then used by a refine function to extract a refined subspace individually for each object. The function performs a bottom-up beam search to find a subspace of length d_2. The search starts from one dimensional possible candidate. In each iteration, it computes the quality of the subspace candidate, i.e., the p-values expressing how well the subspace candidate separates the outlier score population and ranks them. It keeps a list of all candidates ranked by their qualities but only uses the high-quality candidates to construct higher dimensional candidates. The search stops when it is not possible to form a higher dimensional candidate. If the number of attributes in the final candidate S' is less than d_2, the attribute of the candidates in the list will be added to the final candidate until $|S'| = d_2$.

S' is then added to the refined subspace pool SP_2. Finally, it applies the outlier detection model to all the subspaces in SP_2 on all objects in the dataset. It outputs the outlier score and best subspace description for each object in the dataset.

Lookout [56] provides outlying attributes of each outlier as a set of so-called focus plots. A focus plot is 2-dimension scatter plot of all data points where the x-axis and y-axis represent a pair of attributes. Lookout needs the complete data set consisting of inliers and outliers and the set of all possible pair of attributes **P** as inputs. For each pair of attributes in **P**, an isolation forest (iforest) model [57] is constructed using the complete data set and the pair of attributes. The iforest model is then used to recompute the outlier scores of every outlier. Lookout records the outlier score for every possible pair of attributes of each outlier. It then runs a lazy greedy heuristic [58] to find the set of n focus plots that maximize the total maximum outlier scores of outliers represented through the n plots. The outlying attributes of each individual outlier are the focus plots that have the highest outlier score.

Branch and bound subspace search. Siddiqui et al. [12] study the problems of computing and evaluating sequential feature explanations (SFE) for density-based outliers. A SFE is generated for every outlier by estimating the probability density function f(x) (PDF). Given a set of N data points $x_1, x_2, \ldots, x_N$ in d dimensions, the probability density-based anomaly detector ranks the data points according to their joint PDF values. A data point having the lowest joint PDF value is given the highest outlier score by the detector. Note that in the PDF based detector, the threshold based on which a point is detected as an outlier can be derived from a function parameterized by a percentile value α. Given a projection of the dataset on a subset of attributes E, the threshold function $\tau(E, \alpha)$ may generate different threshold values when it applies to different E. The SFE objective

function is defined as the minimum number of attributes that must be disclosed to the analysts so that they can confidently judge a detected point as a true anomaly. Formally, given an SFE E, the smallest prefix SP is defined as the smallest number of attributes in E such that a detected data point x having the PDF value less than a threshold generated by the function $\tau(E, \alpha)$. This definition is written as $SP(x, E, \alpha) = \min k: f(x_{E_k}) < \tau(E, \alpha)$. The SFE objective is defined as finding E that minimizes the expected value of SP denoted by ESP with respect to a prior $p(\alpha)$. Since the true value of the percentile parameter α is unknown, the authors use a discrete distribution over α that assigns $p(\alpha)$ to realistically small values. Mathematically, $ESP(x, E) = \sum_{\alpha} SP(x, E, \alpha)p(\alpha)$, and the objective of SFE is $\underset{E}{\operatorname{argmin}}\, ESP(x, E)$.

The authors also propose an algorithm based on a branch and bound search tree to optimize ESP. This algorithm starts by creating an empty root that has d child nodes which represent every attribute of a data point **x**. Those nodes are stored in a priority queue and each node has an upper bound value which is computed from the joint PDF when **x** is projected on the subset of attributes represented by the node. In each iteration, a node in the priority queue is expanded when it has the smallest upper bound value. Each child node is added to the priority queue if its lower bound value is smaller than the current smallest upper bound; otherwise, it is pruned. The lower bound is computed as $\sum_{\alpha} \hat{t}_{\alpha} p(\alpha)$ where $\hat{t}_{\alpha} = \min\left(\left\{j: j \leq i, f\left(\mathbf{x}_{E_j}\right) < \tau_j(E, \alpha)\right\} \cup \{i\}\right)$, $f\left(x_{E_j}\right)$ is the joint PDF value of a data point x in the SFE E of length j, and $\tau_j(E, \alpha)$ is the threshold value of a given percentile value α. The iteration stops when the priority queue is empty, or the number of nodes expanded reaches the maximum threshold.

2.1.1.2 Local Neighborhood-Based Techniques

Several techniques depend on the local neighborhood of the outlier to derive its outlying attributes. LODI [59] is an algorithm that finds an optimal 1-dimensional subspace to rank and interpret local outliers in multidimensional data sets. For each instance in the data set, LODI heuristically selects the optimal set of nearest neighbors (referred to as the neighboring set) with the maximum information potential using quadratic entropy. Each instance in the data set is a vector in a D–dimensional space, where each dimension represents an attribute. After selecting the neighboring set for each instance, LODI trains a binary classifier to find an optimal 1-dimensional subspace (Opt1D) that maximally separates an instance from its neighboring set.

LODI ranks all instances based on the relative difference between the statistical distance of each instance and that of its neighboring set along the direction of its optimal subspace. The top M instances with the highest numerical outlier ranks are selected as outliers. For each outlier, the projection of its neighboring set over Opt1D is the linear combinations of its original attributes f_1 to f_D. The absolute coefficients within the eigenvector Opt1D correspond to the weights of the original attributes. A user-defined parameter λ (0,1) is used to select the top d absolute coefficients in w for each outlier. Note that $d << D$ and each outlier is associated with a small set of attributes f_1 to f_d. The attributes with large corresponding weights in w are the most important ones to identify outliers. LODI provides explanations for entropy-based outliers.

Similar to LODI, Micenková et al. [60] propose a local based outlier explanation. For a detected outlier **o** in the data set, the technique first simulates n artificial instances, oversampled from the normal distribution centered on **o**. These instances are denoted as

the outlier class. The technique then creates an inlier class by subsampling n instances from the inliers in the data set. A binary classifier is used to separate the outlier class from the inlier class. Then, a standard attribute selection technique is used to determine the so-called explanatory subspace for the corresponding outlier. The explanatory subspace for an outlier is an attribute subset where the outlier has the highest deviation from other points and at the same time, the dimensionality in the subspace is low. This technique can be applied for outliers detected by an arbitrary outlier detection algorithm (outlier detection model-agnostic).

LGOP [61] focuses on identifying and interpreting local outliers using a graph-based model to capture the local geometry. LOGP builds a global graph by finding k nearest neighbors of every object in the dataset. Two objects $\mathbf{x_i}$ and $\mathbf{x_j}$ have a non-negative weighted edge if $\mathbf{x_i}$ is among the k nearest neighbors of $\mathbf{x_j}$ and vice versa. The weight reflects how similar $\mathbf{x_i}$ and $\mathbf{x_j}$ are. If the points are identical then the weight is close to 1, while if they are very dissimilar, the weight is close to 0. For each data instance $\mathbf{x_i}$, LGOP extracts from the global graph a neighboring subgraph X_i (a $D \times k$ matrix) which comprises the vertices corresponding to the k nearest neighbors of $\mathbf{x_i}$. LGOP then maps data points in every neighboring subgraph to a lower dimensional space and, at the same time, preserves the local geometrical structure. The mapping is done through a singular value decomposition of X_i. The matrix resulted from the computation of the SVD is used for eigen decomposition. Since eigen values and eigen vectors are going in pairs, the first eigen vector corresponding to the largest eigen value is used to find the discriminative features by ordering the absolute values of the leading eigen vector decreasingly. The difference in

coefficients between relevant and irrelevant features are expected to be larger at least by a factor of two than most of the differences between two ordered relevant features.

COIN [51][62] also finds outlying attributes of each outlier based on its local neighborhood. Given a data set $\mathcal{X}$ consisting of normal instances and outliers identified by an arbitrary outlier detector, COIN first identifies the local context (C_i) of each outlier $\mathbf{o_i}$. The local context C_i refers to the nearest normal neighbors of $\mathbf{o_i}$ based on the L_2 norm. It also expands $\mathbf{o_i}$ into a hypothetical outlier class $\mathcal{O}_i$ using the synthetic sampling method to balance the outlier data with the inlier C_i data. COIN then segments C_i into different clusters $\{C_{i,1}, C_{i,2}, \dots, C_{i,L}\}$ using K-means clustering or hierarchical clustering. In order to determine the number of clusters of C_i, COIN adopts the measure of so-called prediction strength [62] which is a method used to quantify the number of groups/clusters in the dataset. Any cluster $C_{i,l}$ having the number of members less than 3% of the total number of C_i members is ignored in the training phase, in which outlier and inlier classes are trained using Support Vector Machine (SVM) [63] to find the optimum weights $w_{i,l}$ that maximize the margin between the outlier class and the local context $C_{i,l}$. The length of the vector weight $w_{i,l}$ is equal to the number of attributes of the data set such that every element in the $w_{i,l}$ corresponds to an attribute a_m. The significance of attribute a_m for each $C_{i,l}$ is described as $s_{i,l}(a_m) = |w_{i,l}[m]|/\gamma_{i,l}^m$, where $|w_{i,l}[m]|$ is the absolute value of the element in vector $w_{i,l}$ at index m and $\gamma_{i,l}^m$ is the average distance along the m axis between an instance in $C_{i,l}$ and its nearest neighbors. The value of $s_{i,l}(a_m)$ is used to compute the significance of the attribute a_m for each outlier $\mathbf{o_i}$ that is defined as $s_i(a_m) = \frac{\sum_l |C_{i,l}| s_{i,l}(a_m)}{|C_{i,l}|}$.

The attributes with the large value of $s_i(a_m)$ are considered as the outlying attributes of the outlier $\mathbf{o_i}$.

MICOES [64] forms inlier and outlier classes to generate outlying attributes like LODI and COIN, yet it utilizes incremental computation to generate outlying attributes to deal with the unbounded volume in data streams. It consists of two components: (1) the micro-clusters that are updated whenever a data object arrives and run in parallel with the outlier detector, and (2) the explainer that is only evoked when an object is identified as an outlier by the outlier detector.

MICOES depends on the neighborhood of an outlier with a particular timestamp ($\mathbf{o_t}$) to find the attributes responsible for the outlierness of $\mathbf{o_t}$. When a data point arrives, MICOES absorbs the data point into a micro-cluster using a single pass approach. It then incrementally updates to the statistics of the micro-cluster, yet it does not keep the original data point. When $\mathbf{o_t}$ is detected as an outlier, MICOES finds the relevant micro-clusters whose centroids lie within the maximum distance threshold from $\mathbf{o_t}$. It treats the centroids of the relevant micro-clusters as data points. MICOES also creates an outlier class by generating some points sampled from a normal distribution centered on $\mathbf{o_t}$. Suppose MICOES finds *n* micro-clusters near $\mathbf{o_t}$, it then uses *n* linear models to find *n* hyperplanes that separate each of the micro-cluster from the outlier class. The weights associated with the hyperplanes represent the contributions of the attributes to the abnormality of $\mathbf{o_t}$. MICOES applies Lasso regularization on each of the linear models such that the attributes that do not contribute to form the decision boundary will have zero weights. It then computes each attribute's contribution score by taking the average of its corresponding

absolute weight value from all the hyperplanes. The attributes with non-zero contribution scores are considered as the outlying attributes.

2.1.1.3 Decision Tree-Based Techniques

The outlying attributes of an individual outlier can be generated from decision trees. We now describe the techniques in this category.

Kopp et al. [65] introduce Explainer, an algorithm to develop Sapling Random Forests (SRF), an ensemble of specifically trained binary classification trees. SRF explains outlier detected by any arbitrary anomaly detector. The approach splits the data set into two disjoint sets Set_a and Set_n containing anomalous and normal samples accordingly. For each anomaly $\mathbf{x_a}$, the approach selects a training set consisting of the anomaly as one class and a randomly chosen subset of the normal samples as the other class. The size of the training set depends on the user input. Multiple training sets are generated for each anomaly by repeating the sampling process for the other class. They are then used to train multiple binary decision trees or saplings to classify each anomaly. The growth of the sampling is stopped if the leaf containing $\mathbf{x_a}$ is pure, or if the height is over a threshold. This threshold is estimated from the average heights of single samplings trained per anomaly. Saplings/decision trees for each anomaly $\mathbf{x_a}$ are used to derive decision rules from the splitting conditions and attribute thresholds at each node on the path from the root to the leaf containing $\mathbf{x_a}$. The following conjunction (c) of these rules is used to explain $\mathbf{x_a}$: $c = (x_{j1} > \Theta_1) \wedge (x_{j2} > \Theta_2) \wedge \ldots \wedge (x_{jd} > \Theta_d)$ where j1…. jd are attributes and $\Theta_1 \ldots \Theta_d$ are attribute thresholds used in the splitting conditions of the saplings. For anomalies with multiple saplings, the algorithm aggregates the conjunctions of these rules by grouping the rules per attribute set. Thus, unlike other algorithms, SRF produces rules with attributes

that are most important and discriminative for detecting anomalies. The typical height of the saplings in SRF is between 1 and 3. Thus, the length of the association rules derived from theses saplings are short, leading to compact and intuitive explanations of anomalies. In their later work [65], the authors form these rules extracted from SRF as association rules $r_f(x)$, as follows: $r_f(x) = x_{j1} > \Theta_1 \wedge x_{j2} > \Theta_2 \wedge \ldots \wedge x_{jd} > \Theta_d \rightarrow x \in Set_a$. Here, the conjunctions of the rules are the antecedents and a classified anomaly x is the consequent. The rules' support, lift and confidence are used to select the desired sets of association rules to explain individual anomalies.

EXAD [66] identifies the outlying attributes of the outliers detected using a neural network (LSTM/Autoencoder). Raw data collected from the system/application traces does not always present a sufficient feature set. Deep learning is used because it addresses feature engineering and anomaly detection in the same mechanism. The neural network is approximated by a decision tree to provide explanations about outliers. An explanation is formed by the paths starting from the root of the tree leading to the leaves marked as outliers. The final explanation is presented as a disjunctive normal form (DNF) of atomic predicates of the form (v o c) where v is a feature, c is a constant, and o is one of the operators {<, <=, =, >=, >}. If later another object in the test set satisfies the condition in the DNF, then the object can also be labeled as an outlier. Hence, the explanation can be used to predict future outliers.

2.1.1.4 Layer-wise Relevance Propagation (LRP) Techniques

Amarasinghe et al. [67] propose a technique focusing on predicting outliers using Depth Neural Network (DNN) and provides explanations about features relevant in making the prediction. Before the DNN model is deployed to predict outliers in data streams, it is

built offline using some labelled training set. The DNN model is trained for multi-class classification where some of the classes represent the anomalous category.

During the offline training phase, the relevance of each input feature to the DNN prediction on each class is computed using Layer-wise Relevance Propagation (LRP). LRP uses backward propagation to compute the relevance of each dimension of the data point in every layer. It starts from the output layer (the highest layer), hidden layer, and all the way to the input layer (the lowest layer). In other words, the relevance of the lower layer is computed based on the relevance of the higher layer. The features' relevance scores of each class are computed by summing all the feature relevance scores of the data points belonging to the class in the input layer and dividing it by the number of the class members.

Keep in mind that LRP is done during the DNN training phase. When a data point is classified as an outlier by the deployed DNN model, the outlying attributes of the data point are derived from the relevant features of anomalous class in which the outlier is classified.

2.1.1.5 Entropy-Based Reward Techniques

EXstream [68] applies an entropy-based distance function to measure the reward for every feature and ranks every feature based on the reward to find outlying attributes for each outlier. This algorithm provides explanations for user annotated outliers in Complex Events Monitoring (CEP). Consider a stream producer generating events of multiple types. Each event type follows a schema consisting of set of attributes including a time-stamp attribute. CEP receives events stream continuously and monitor the events using user-defined queries.

Users can interact with the CEP monitoring system through the provided visualizations. The visualization usually shows the time stamp on the X-axis and one of the derived event's

attributes on the Y-axis. Each visualization represents a partition which is the result of a user-defined query monitored by a Hadoop job. Users can drag and draw rectangles on the visualization to annotate the abnormal interval (I_A) and the regular interval (I_R) that demonstrate the normal behavior. Given the I_A and I_R, the relevant attributes/features for the abnormal behavior are generated from the events that take place during the I_A and I_R.

EXstream assumes that the CEP system archives the raw data streams. EXstream computes the reward for every single feature using a distance-based function which measures the difference between the feature's values in the I_A and I_R. It then ranks the features based on their rewards. Sharp changing between successive features in the ranking indicates that the features that rank below the sharp drops are not significant for explanations. The insignificant features are filtered out. The algorithm identifies and purges the so-called false positive features which are the features that demonstrate similar behavior in other partitions without abnormal indication. It searches the archived streams to find similar partitions. Such partitions should be the results generated by the same query and monitored by the same Hadoop program on the same dataset. The retrieval is fast because EXstream maintains a record of partitions. Once it discovers the related partitions, it aligns the annotated region IA and IR to each partition on temporal-based or point-based. It selects the alignments in which two partitions have the smallest relative difference. Alignments map the annotation to all related partitions.

EXstream uses hierarchical clustering to label the intervals in the related partitions. If a period is placed in the same cluster as the annotated anomaly, then it is labeled as an outlier. It then recomputes the entropy-based distance for each feature left and filter out the false positive ones. It uses pairwise correlation to identify similar features. Each feature is

represented as a node. Two nodes are connected when their pairwise correlation exceeds a threshold. Connected nodes are considered as a cluster and EXstream selects one node from each cluster. Once the final selection is done, the conjunction normal form (CNF) is used to connect the selected features. The value boundaries of the selected features during the abnormal interval (I_A) are used as constants in the CNF, for example $feature_1 > 10 \wedge feature_5 < 5$. The CNF represents the complete explanations. Furthermore, when a future event stream satisfies the condition of the CNF, it can be classified as an anomalous event.

2.1.1.6 Game Theory-Based Techniques

Takeshi [69] proposes a technique using the Shapley values of the reconstruction errors of PCA to explain outliers detected using PCA. The Shapley value is a method from game theory to distribute the gain of a game to the players. In the context of PCA-based anomaly detection, a gain is an outlier score which is the reconstruction error of the PCA, and a player is a feature used by the model. To compute the contribution of an individual feature, the algorithm needs the following inputs: the principal component matrix **Q**, the observation noise variance, a data point **x**, a target feature's index j, and the total number of Monte Carlo iterations.

The algorithm approximates the Shapley value of an individual feature using Monte Carlo approximation. The feature i's Shapley value is initially set to 0. In every iteration, an order of feature indices D is drawn randomly from the set of permutations of {1, … ,d} where d is the total number of features. The Shapley value of the current iteration is computed by subtracting the value function of the set of feature indices that precede i in the order D from the value function of the set of feature indices that precedes i in the order D union i. The Shapley value of the current iteration is then divided by the total number of

iterations and the result is added to the feature i's Shapley value. Features whose Shapley' values are higher than a threshold, are considered outlying attributes of **x**.

2.1.1.7 Discussion of the Surveyed Techniques on Outlying Attributes of an Individual Outlier

We now discuss the aspects of the techniques presented in Sections 2.1.1.1 - 2.1.1.6 based on the research challenges previously described in Section 1.5 in Chapter 1. We first discuss how the existing techniques handle the challenge related to the outlying attributes in general and then we explain whether they address the challenges related to data streams.

- Challenge (OutAtt1): *Limiting subspace search*

  Searching the subspace (Section 2.1.1.1) to find the optimum subset of attributes responsible for the abnormality of outliers is an NP-hard problem. Knorr and Ng's technique [54] and Refout [48] apply some heuristics to find relevant subspaces that represent the outlying attributes for an individual outlier; however, they do not guarantee to find the optimum subspace which is the one where the outlier is the most abnormal compared to all other subspaces. On the other hand, Lookout [56] uses a submodular objective to quantify the explanation quality and guarantees finding the optimum 2-dimensional subspace. The algorithm scales linearly with the number of input outliers. SFE [12] adopts four greedy-based approaches and a branch-and-bound approach to find the outlying attributes. In all the approaches, SFE needs multiple scans of the entire dataset in order to compute the joint PDF for a single outlier object. SFE's branch-and-bound approach prunes the search space, and it is aimed to find the optimum subspace. Nevertheless, their experiments show that the greedy approach is preferred because it is less computationally intensive and yields similar results to the

branch-and-bound approach in actual evaluations. Other techniques presented in Sections 2.1.1.2 - 2.1.1.6 do not rely on the subspace search. However, they do deal with some overhead. For example, for every single outlier, COIN [51] requires finding so-called local context and training classifiers based on it to find the outlying attributes.

- Challenge (OutAtt 2): *Generating readily interpretable output*

  Most of the subspace search-based techniques described in Section 2.1.1.1 produce a set of original attributes relevant to the abnormality of each outlier. However, they do not include the attributes' values that characterize the outlier. Lookout [56] provides pictorial explanations. It shows scatterplots of 2-dimension subspace where users can easily tell whether each outlier is isolated from other objects.

  The local-neighborhood-based techniques (Section 2.1.1.2), the layer-wise relevance propagation (LRP) technique (Section 2.1.1.4), and the game theory-based technique (Section 2.1.1.6) generate real number values associated with each original attribute. The values represent the contribution of each attribute to the abnormality of an outlier where the non-contributing attributes are supposed to have zero or minimal values. This kind of output requires users to define a threshold to limit the number of attributes of non-zero weight they want to examine further.

  The decision tree-based techniques (Section 2.1.1.3) and the entropy-based reward technique (Section 2.1.1.5) use the disjunctive normal form (DNF) or conjunctive normal form (CNF) to represent the outlying attributes of an outlier. This kind of output does not require users to specify a threshold; however, users' efforts to verify the result are equivalent to the number of rules in the DNF/CNF.

Most of the techniques for generating outlying attributes described in Sections 2.1.1.1 - 2.1.1.6 are for static or regular data. Therefore, they do not handle any challenges related to tdata streams (Challenges DS 1- 4 Section 1.5). However, techniques such as EXAD[38], EXstream [64], MICOES [64], and MacroBase [64] are designed for data streams.

- Challenge (DS 1): *Dealing with the notion of time*

  EXAD, EXstream, MICOES, and MacroBase process data points on the sequential order based on their arrival timestamp.

- Challenge (DS 2): *Dealing with the notion of infinity*

  To deal with the notion of infinity, MICOES utilizes incremental computation to form inlier and outlier classes. It only needs the summary or the statistics of the arriving data points to generate outlying attributes. Both EXstream and MacroBase also employ incremental computation. However, EXAD does not need to update its model when data points arrive, allowing it to generate outlying attributes as soon as it sees an outlier.

- Challenge (DS 3): *Dealing with different data arrival rates*

  EXAD, EXstream, MICOES, and MacroBase operate independently for each data stream. Thus, they do not need to be concerned about whether the data streams have the same arrival rate or not.

- Challenge (DS 4): *Dealing with concept drift*

  EXAD uses a decision tree, estimated from a neural network trained offline, to generate outlying attributes of each detected outlier online. However, it does not provide a mechanism to update the decision tree when the concept drift happens. In contrast, EXstream updates the temporal context needed to find the outlying attributes over time to capture the changes in the data distribution. Similarly, MICOES also absorb arriving

data points into micro-clusters, and the continuous update on the micro-cluster state captures changes in the data distribution. MacroBase produces outlying attributes using a prefix tree of frequent itemsets updated periodically, allowing it to handle concept drifts.

2.1.2 Techniques to Find Outlying Attributes of a Group of Outliers

Finding the outlying attributes of a group of outliers is basically finding the similarity among outliers. We review the techniques proposed for this type of outlier explanation in this section. The main methodologies used by these techniques to generate the explanations can be summarized as (i) Apriori-based, (ii) Attribute reduction-based, and (iii) Subspace clustering-based. They are described in Sections 2.1.2.1 - 2.1.2.3. The discussion on how the techniques address the challenges explained in Section 1.5 Chapter 1 is given in Section 2.1.2.4.

2.1.2.1 Apriori-based outlying attributes

Chen et al. [70] propose an algorithm to find outlying attributes in distance-based outliers. They first identify outliers in multi-dimensional spaces using the distance-based outlier detection algorithm proposed by Knorr and Ng [47][72] and then group the detected outliers based on the times they were observed (G1 to Gn for n observed times, while each group Gi has a different number of outliers). For each group of outliers (Gi), the algorithm generates a knowledge set using the Mine-Knowledge Set algorithm, which resembles the apriori association rule data mining algorithm [73]. A knowledge set is the minimum

subspace in which an object is detected as a non-trivial outlier (an outlier is non-trivial in an attribute set $\mathcal{A}_{\mathrm{p}}$ if it is not an outlier in a proper subset of $\mathcal{A}_{\mathrm{p}}$ [54]). Before generating knowledge sets, the algorithm assumes that the users have categorized data attributes into three types: identification (I), indication (C), and observation (O). Identification attributes are the attributes that together uniquely identify each object in the data set. Indication attributes are the attributes whose values are used to calculate the category and behavior rate. Observation attributes are the attributes that are used to group objects based on the time when the objects were observed. The construction of knowledge sets is based on indication attributes. Each outlier found in an observation may have several knowledge sets. Any pair of outlier groups {Gi,Gj} is considered categorically similar if the number of outliers in Gi sharing the knowledge sets with the outliers in Gj exceeds some predefined threshold. Thus, categorical similarity indicates a high percentage of common categories of outliers between Gi and Gj. Any pair of outlier groups {Gi,Gj} is considered behaviorally similar if the number of categorically similar outliers in Gi and Gj lying very close to each other exceeds some predefined threshold. Any pair of outlier groups {Gi-1,Gi} are considered similar if they are both categorically and behaviorally similar. Thus, n groups of outliers (G1, G2, …, Gn) are similar if every two consecutive groups are similar. The groups that are categorically and behaviorally similar are considered as one big group. The outlying attributes of this big group are the knowledge sets that make the members categorically similar.

2.1.2.2 Attribute Reduction-based

Huang and Yang [71] propose a technique to find sets of attributes having most of the detected outliers. The proposed method consists of (i) identifying outliers in the total

attribute space (the outlier detection task), (ii) finding the Knowledge Attribute Subspace (KAS), which is the set of attributes where an identified object has abnormal values, and (iii) detecting the key KAS (KKAS), which is the sets of attributes having a high percentage of outliers. The algorithm to find KAS is similar to the method to find the knowledge set proposed in [72] which we described in the apriori-based outlying attributes technique in Section 2.1.2.1.

To find KKAS, the algorithm needs to compute the outlying partition similarity which is the sum of the weighted percentages of the cardinality of outliers found in a KAS compared to the cardinality of outliers found in the full attribute space. The outlying partition similarity for each KAS is calculated using the equation:

$$\text{outlying partition similarity} = w_1 f_{sup} + w_2 f_{con} + w_3 f_{inc}$$

where $f_{sup} = \frac{card(XO_s \cap XO_d)}{card(XO_s \cup XO_d)}, f_{con} = \frac{card(XO_s \cap XO_d)}{card(XO_s)}, f_{inc} = \frac{card(XO_s \cap XO_d)}{card(XO_d)}$, w_1, w_2, w_3 are weights defined by the user, $card(XO_d)$ is the cardinality of outliers detected in the full attribute space, $card(XO_s)$ is the cardinality of outliers detected in a given KAS, $card(XO_s \cap XO_d)$ is the cardinality of outliers found both in a given KAS and in the full attribute space, and $card(XO_s \cup XO_d)$ is the cardinality of outliers found in a given KAS or the full attribute space.

The concept of maximum outlying partition is based on the fact that if an object is an outlier in the entire attribute space, its projection in a subspace is not always an outlier and vice versa. The algorithm determines KKAS by finding the maximum outlying partition similarity among KAS. KKAS is the set of attributes that contains most of the outliers. In other words, KKAS is the set of outlying attributes for the group representing most of the detected outliers.

2.1.2.3 Subspace Clustering Approach

We now describe XPACS [73] whose goal is to identify few small clusters of outliers in an arbitrary attribute subset. XPACS finds the groups/clusters of outliers following three steps. Step 1 is to apply subspace clustering in order to group outliers into a set of hyper-rectangles. In an attribute subset of length k, a hyper-rectangle R has k sides where each side represents a value interval/range of an attribute. The subspace clustering starts by initializing candidates of hyper-rectangles R in 1-dimension using kernel density estimation (KDE). Outliers could be concentrated in more than one interval in each attribute, meaning there could be more than one candidate hyper-rectangle in every attribute. To find the multiple intervals in an individual attribute, the subspace clustering algorithm varies the quantile thresholds.

The algorithm progresses level by level. If a candidate hyper-rectangle in k-dimension meets the mass threshold which is the minimum number of outliers required to form a cluster, it will be examined further; otherwise, it will be discarded. The candidate hyper-rectangle R is then checked to see whether the number of normal points included is less than the maximum number of inliers allowed. If yes, it is then included in the set of pure hyper-rectangles in k-dimension denoted as $\mathcal{R}^{\mathrm{k}}_{\mathrm{pure}}$; otherwise, it is included in a non-pure set denoted as $\mathcal{R}^{\mathrm{k}}_{\neg\mathrm{pure}}$. All candidate hyper-rectangles in $\mathcal{R}^{\mathrm{k}}_{\mathrm{pure}}$ are joined into the set of hyper-rectangles denoted by $\mathcal{R}$. The next candidates of hyper-rectangles in k+1-dimension are generated by joining $\mathcal{R}^{\mathrm{k}}_{\mathrm{pure}}$ and $\mathcal{R}^{\mathrm{k}}_{\neg\mathrm{pure}}$. Two hyper-rectangles in k-dimension ($\mathrm{R}^{\mathrm{k}}_{\mathrm{a}}$ and $\mathrm{R}^{\mathrm{k}}_{\mathrm{b}}$) can form a candidate of hyper-rectangle in k+1-dimension if they both have the same interval (side) in k-1 attributes.

Step 2 of the algorithm refines the set of rectangles found in Step 1 into a set of axis-aligned hyper-ellipsoid called pack. A pack is defined as a set of data points whose squared distance from the center is equal to or less than 1. Formally, a pack formed in an attribute subset of length k is defined as $p_k = \{x|(x-c_k)M_k^{-1}(x-c_k) \leq 1\}$ where c_k represents the center of the hyper-ellipsoid and M_k is a diagonal matrix of $k \times k$ size. When a hyper-rectangle R is transformed into a hyper-ellipsoid (pack), the goals are to: (i) keep the outliers that are already in R denoted as x_i, (ii) exclude normal data points denoted as x_l and (iii) include more outliers that are not in R denoted as x_j. A pack found in an attribute subset of length k is represented by a conjunction of k attribute rules where each rule is an attribute value interval $\left(c^{(z)} - \text{radius}_z, c^{(z)} + \text{radius}_z\right)$. They can be used to predict future outliers.

The same outliers can be covered by multiple packs generated in Step 2. Hence in Step 3, the goal is to find packs that describe outliers in the dataset as briefly as possible while avoiding packs that cover largely overlapping groups of outliers or contain too many inliers. Step 3 uses the random greedy algorithm by Buchbinder et al. [74] to select a subset of packs that yields the fewest bits. The optimum packs selected in Step 3 represent the outlying attributes for each group of outliers. They can be used to predict future outliers.

2.1.2.4 Discussion of the Surveyed Techniques on Outlying Attributes of Groups of Outliers

We now discuss the aspects of the techniques presented in Sections 2.1.2.1 - 2.1.2.3 based on the research challenges explained in Section 1.5 in Chapter 1.

- Challenge (OutAtt 1): *Limiting subspace search*

XPACS [73] (Section 2.1.2.3) limits the subspace search by pruning some subspaces that do not meet the mass threshold. The other two techniques apply some heuristics so that they do not have to check all possible combinations.

- Challenge (OutAtt 2): *Generating readily interpretable output*

  The algorithms by Chen et al. [70] and Huang and Yang [71] are both outputting the set of outlying attributes which are easily understood. XPACS [73] (Section 2.1.2.3) uses the conjunctive normal form (CNF) to represent each group of outliers and uses a greedy approach to determine the shortest CNF. CNF is considered as an interpretable machine learning output [37].

All of the techniques for generating outlying attributes described in Sections 2.1.2.1 - 2.1.2.3 are for static or regular data. Therefore, they do not handle any challenges related to the data streams (Challenges DS 1- 4 Section 1.5).

2.1.3 Summary of the Properties of Outlying Attribute Discovery Techniques

In Table 2-1, we summarize the properties of the reviewed techniques that generate outlying attributes. Our summary is based on the following aspects:

i. the name of the proposed techniques
ii. the referenced papers (including the year they were published)
iii. whether the techniques generate outlying attributes for an individual outlier or a group of outliers
iv. whether they are applied to static data or data streams

v. whether they handle the data streams-related challenge (DS 1): the notion of time)

vi. whether they the notion of infinity (DS 2)

vii. whether they handle different data arrival rates (DS 3)

viii. whether they handle concept drift (DS 4)

ix. whether the handle the additional research challenge related to outlying attributes (OutAtt 1): limiting subspace search

x. whether they generate readily interpretable output (OutAtt 2)

xi. whether they apply parallelism

Data stream is an example of Big Data, characterized by the large volume, remarkable velocity, variety of data formats from multiple sources, and variability of data [75]. Parallel processing is crucial as we deal with Big Data [75, 76]. Big data is characterized by a large volume, remarkable velocity, variety of data formats gathered from multiple sources, and uncertainty. We can accelerate outlier explanations on Big Data using parallelism, achieved by running the outlier explanation task on numerous computing cores [77, 78], GPU, or distributed system. Thus, we include parallelism in Item xi.

We first summarize Table 2-1 based on the reviewed outlying attribute techniques in this Chapter. Table 2-1 shows that most of the reviewed techniques generate outlying attributes for each individual outlier in static data. Only four techniques are designed for data streams. These four techniques address most of the research challenges related to data streams; however, since none of them leverage cross-correlation, they do not account for different data arrival rates of data streams (DS 3). All the reviewed techniques address

additional research challenges related to generating outlying attributes. Only three reviewed techniques use parallelism.

Furthermore, at the last row of Table 2-1, we include our proposed technique, EXOS (explained in Chapter 3). EXOS is designed to discover outlying attributes for each individual outlier in data streams. It deals with the notion of time, infinity, and concept drift. It takes advantage of cross-correlation to handle data streams with different arrival rates. It employs a similar approach to local neighborhood-based techniques that do not rely on finding all possible feature subspaces (OutAtt 1). For each individual outlier, EXOS produces an outlying attribute contribution score for each original attribute of the outlier. The score of an attribute represents its contribution to the abnormality of the outlier. This output is interpretable (OutAtt 2) because, given a threshold, any attributes with outlying attribute contribution scores lower than the threshold are considered non-outlying attributes. Finally, it is clear that EXOS is the only technique that covers all the research challenges described in Section 1.5, Chapter 1. In addition, EXOS also applies parallelism.

Table 2-1 Summary of the properties of the outlying attribute discovery techniques

Name	Year, Reference	Outliers are explained as	Data Type	Data Stream Related Challenges				Outlying Attributes Challenges		Apply Parallelism
				DS 1	DS 2	DS 3	DS 4	OutAtt 1	OutAtt 2	
_	1999, [79]	Individual	Static					√	√	
LODI	2013, [80]	Individual	Static					√	√	
_	2013, [26]	Individual	Static					√	√	
Refout	2013, [48]	Individual	Static					√	√	
Explainer	2014, [81]	Individual	Static					√	√	
LOGP	2014 , [27]	Individual	Static					√	√	
EXstream	2017, [82] 2021, [83]	Individual	Stream	√	√		√	√	√	√
COIN	2018, [51]	Individual	Static					√	√	
EXAD	2018, [38]	Individual	Stream	√	√			√	√	
MacroBase	2018, [84] 2021, [83]	Individual	Stream	√	√		√	√	√	
_	2018, [85]	Individual	Static					√	√	
Lookout	2019, [56]	Individual	Static					√	√	
SFE	2019, [49]	Individual	Static					√	√	
_	2019, [12]	Individual	Static					√	√	
MICOES	2021, [64]	Individual	Stream	√	√		√	√	√	√
_	2003, [72]	Group	Static					√	√	
KKAS	2011, [71]	Group	Static					√	√	
XPACS	2018, [73]	Group	Static					√	√	√
EXOS (proposed in Chapter 3 of this dissertation)		Individual	Stream	√	√	√	√	√	√	√

2.2 Techniques to Find Root Cause Factors of Outliers

There exist techniques [13, 46, 86–88] that find root causes factors of outliers. Techniques such as CloudRanger [13], MicroCause [28] and DyCause[46] are proposed for data stream applications especially for cloud service monitoring. All these techniques find the root cause factors of outliers based on a causal graph. A causal graph is a graphical representation used to model and visualize causal relationships between variables in a system. We categorize the recent techniques for discovering the root cause factors of outliers based on how the causal graph is defined: (i) data-driven based causal graph, and (ii) user-defined causal graph.

2.2.1 Data-Driven based Causal Graph

CloudRanger [13] is used in microservices monitoring for cloud systems where frontend services relate to other backend services. An outlier which is unusual latency is observed in a frontend service during a time period or window. To find the root causes of the outlier, CloudRanger constructs a so-called impact graph or anomalous causal graph based on the correlation between services. It uses the PC algorithm [89], a causal discovery algorithm to form the impact graph for outliers using data from the window. Once the impact graph is formed, CloudRange calculates the correlation among nodes or vertices in the graph to determine the so-called impact relationship between services. The impact relationship affects how often a vertex/service in the impact graph is visited during a random walk. CloudRanger ranks the services based on their visit frequency in descending order. The *k* most visited services are most likely to be the root causes of the outlier.

MicroCause [28] is designed to localize the root causes of failures in microservice monitoring. Operators continuously monitor a microservice's key performance indicators (KPIs) and metrics to ensure a high quality of user experience. A KPI, such as the average response time of web users, directly reflects the quality of service. Metrics, such as CPU utilization of a container, queries per second, and middleware response time, indicate the status of a microservice's underlying components. When a KPI is detected as an outlier, the microservice is considered to have failed. MicroCause is employed to identify the root cause factors of the outlier, focusing on the top k metrics most likely responsible for the KPI failure.

Suppose a KPI failure occurs at 3:15 pm on March 15, 2024. Immediately, MicroCause is activated and uses time series data of the KPI and metrics from 3:10 pm to 3:15 pm on March 15, 2024 to construct a failure/outlier causal graph. The KPI and each metric are represented as vertices in this causal graph. The construction of the failure causal graph employs an improved version of the PC algorithm for time series data [28]. In parallel with the failure causal graph construction, MicroCause also employs a univariate anomaly detection to measure the anomalous degree of each metric during the period of time.

MicroCause then performs random walks on the failure causal graph using Temporal Cause-Oriented Random Walks (TCORW) [28]. During TCORW, the root cause contribution scores of the microservices' metrics are computed, taking into account the anomalous degree of each metric. The metrics with the top k scores are identified as the root causes of the failure.

Pan et al. propose DyCause to identify the root cause factors of microservice failures experienced by applications (user space) utilizing a microservice system. They argue that

existing techniques rely heavily on monitoring metrics gathered by the server side (kernel side) of the microservice system. The microservice system may be used by multiple applications. Each application collects the operational status of the kernel services they utilize and periodically sends their logs through APIs to the DyCause system. DyCause converts these logs into various time series representing the performance metrics of the kernel services. When an application detects a failure (outlier), DyCause employs SPOT to find an abnormal interval in each metric time series. SPOT utilizes extreme value theory (EVT) to generate a threshold-based extreme value distribution of the past data. Each metric has its own threshold, and DyCause selects abnormal intervals only for the metrics that meet this threshold. The abnormal interval contains a window of data before and after the outlier's timestamp.

Even though the failure is triggered by an application, the detection of the abnormal interval of kernel metrics is done for all applications to ensure DyCause covers all possible causes. For each application, DyCause employs the discovered abnormal metric intervals to find dynamic causality curves using the Granger causality test. Dynamic causality curves provide a propagation path showing how an anomaly in one service propagates through other services during the abnormal interval. These curves are then used to construct a weighted local dependency graph. Each vertex in the graph represents a service. If one service affects another, there is an edge with an arrow connecting the first service to the second. The weight on each edge, ranging between 0 and 1, represents the causality strength between the services.

Since the clues that a single application can provide are limited, DyCause pools insights from other applications. It fuses the application's local dependency graph with those of

other applications. It computes the similarity between the application that triggered the outlier and other applications using the Pearson correlation coefficient. It then refines the local graph weights based on the similarity values. DyCause applies backward breadth-first search (BFS) on the refined graph to find root cause factors (services) that triggered the outlier. During the backward BFS, DyCause estimates service abnormality scores by considering the path correlation strength from the abnormal front-end service (the application) and the Pearson correlation coefficient. DyCause outputs the services ranked by their abnormality scores from high to low. The higher the score, the higher the chance that a service is a root cause factor of the outlier.

2.2.2 User-Defined Causal Graph

Applying data-driven causal discovery-based algorithms such as the PC algorithm requires a large amount of data to form the true causal graph. However, constructing anomalous causal graphs for outliers purely from data may not be accurate since the amount of anomalous data points is usually small. Hence, some recent techniques such as CausalRCA [87] and EasyRCA [88] use a different approach. They assume that causal graphs that show the dependencies between variables in normal situations are given by domain experts.

EasyRCA [88] works for timestamped dataset. It is intended to find the root causes of a collective outlier which is the collection of anomalous data points where the individual data points may not be anomalous by themselves [90]. The algorithm inputs the data set containing inliers and outliers during a time period, the summary causal graph of the normal time series, the maximum lag between a cause and an effect, the starting timestamp of the collective outliers in each of the causal graph vertices, and the number of data points

in the collective outlier. It separates the dataset into the normal and anomalous regimes. The normal regime consists of all inliers that occurred before the collective outliers (anomalous regime) and the outlier. It uses the given causal graph to find an anomalous subgraph. It then finds the sub-root vertices and time-defying vertices of the outlier. A sub-root vertex is a root vertex in the anomalous subgraph. A time-defying vertex is a vertex on which the collective outliers have appeared earlier than on its parents. A set of the sub-root vertices together with the time-defying vertices is deemed to be the root causes of the collective outliers. The root causes set also includes other vertices whose direct effect in the normal regime is different than the anomalous regime.

Budhatoki et. al. [87], propose CausalRCA to find root causes of point outliers in a static dataset. CausalRCA does not consider the timestamps of the data points. A point outlier refers to the anomalous individual data instance with respect to the data set. Given a causal graph with the premise that the graph follows a Functional Causal Model (FCM), inliers, point outliers, and a target vertex in which the point outliers are detected, the algorithm will find the vertices that cause the target vertex value to be anomalous. FCM is a mathematical function that describes the causal dependency between variables in a system [91–94]. It describes how the value of a variable depends on the values of other variables in the model. The procedure for determining whether a particular vertex X_i is a root cause of an outlier is based on the counterfactual question "*Would the data point at the target vertex not have been an outlier had we assigned rather 'normal' causal mechanisms at X_i instead of the observed mechanism associated with the outlier?*" [87].

CausalRCA takes a causal graph, inlier dataset, and point outliers as inputs. It follows **six steps** to generate the root causes of each outlier. Note that each instance in the inlier and outlier dataset is a jointly observed value of each vertex in the causal graph.

***Step 1*:** It samples *m* normal data to train or fit the FCM. If a vertex is a root in the causal graph, it uses the sampled data associated with the vertex to find the vertex's distribution type and its corresponding parameters. If a vertex has parents, the algorithm uses its parents as independent variables and the vertex itself as a dependent variable in the prediction model (e.g., linear regression model). It trains the model with the parents' sampled data as the predictor and the vertex's sampled data as the target. It then fits the noise term of the vertex using the difference between the estimated value of the vertex generated by the prediction model and the actual value of the vertex in the sampled data.

***Step 2*:** It applies the noise term of each vertex trained in Step 1 to generate M Noise Samples and corresponding M Vertex Samples. The Noise Samples represent M noise term values of vertices while the Vertex Samples represent M values of vertices. The Noise Samples and Vertex Samples generations are done iteratively from the root to the leaf vertices.

***Step 3*:** It fits the statistical outlier scoring function of the target vertex using the values of the target vertex in the Vertex Samples.

***Step 4*:** For each outlier, it generates a tuple of outlier noise using the prediction model of each vertex trained in Step 1. If a vertex is a root, the corresponding outlier noise value is generated from the noise term.

***Step 5*:** Given the outlier noise in Step 4, starting from the root to the leaf vertices, the algorithm estimates the vertex values of the outlier noise using the FCM fitted in Step 1. It

then computes the outlier score of the estimated target vertex's value using the outlier scoring function from Step 3.

***Step 6*:** It uses the FCM, Noise Samples, outlier scoring function, outlier noise, and outlier score generated in Steps 1, 2, 3, 4, and 5 respectively, to estimate the Shapley value of each vertex for the outlier. Each vertex's Shapley value represents the contributions of the vertex as the root cause. The vertices with the *k* highest absolute Shapley values [87, 95] are the root cause factors of the outliers.

2.2.3 Discussion of the Surveyed Techniques on Root Cause Factors of Outliers

We now discuss the aspects of the techniques presented in Sections 2.2.1 - 2.2.2 based on the research challenges explained in Section 1.5 in Chapter 1. We first discuss how the existing techniques handle the challenges related to root cause factors of outliers and then we explain whether the address challenges related to data streams.

- Challenge RC 1: *Providing counterfactual*

  CausalRCA [87] is the only reviewed technique that provides a counterfactual question on finding the root cause factors of an outlier. To determine whether a variable (vertex) is a root cause factor of an outlier x_n, CausalRCA asks this counterfactual question "Would the event x_n have not been an outlier had we assigned rather 'normal' mechanism at the variable instead of the existing mechanism associated with the outlier x_n?".

- Challenge RC 2: *Incorporating user knowledge about variables' causal dependency*

CausalRCA [87] and EasyRCA [88] require a user-defined causal graph in their inputs. This causal graph describes the normal system variables' dependency. Both CausalRCA and EasyRCA assume that the causal graph follows functional causal models in which each child variable (vertex) in the causal graph is a function of its parents' variables.

CausalRCA [87] and EasyRCA [88] are proposed for static data. They do not handle any challenges related to data streams except for the notion of time where EasyRCA processes data points based on their timestamps. However, CloudRanger [13], MicroCause [28] and DyCause [46] are proposed for data stream applications.

- Challenge (DS 1): *Dealing with the notion of time*

  CausalRCA, MicroCause, and DyCause consider the outlier's timestamp to find its root cause factors; hence they handle the notion of time. CausalRCA and MicroCause generate anomalous causal graph with data points that arrive before the outlier. However, DyCause still requires some data points with timestamps after the outlier's occurrence. Even though EasyRCA is applied for static data, it finds so called normal and anomalous regimes based the start and end timestamp of a collective outlier.

- Challenge (DS 2): *Dealing with the notion of infinity*

  CloudRanger, uses a time-based window to process data points, allowing it to run in real time and handle the notion of infinity. In contrast, MicroCause does not explicitly describe how it collects and processes the arriving data points, suggesting that it might find the root cause factors of outliers in an offline setting. DyCause collects logs sent by applications that use a microservice system. However,

DyCause assumes that all necessary logs are available to examine the microservice metrics when an application triggers a failure (outlier). Hence, we can infer that neither MicroCause nor DyCause deals with the notion of infinity.

- Challenge (DS 3): *Dealing with different data arrival rates*

  CloudRanger and MicroCause operate on the server side of the cloud service monitoring. The data points they examine, can be considered to arrive from one system (one source). Hence, they do not need to be concerned about different arrival rates. However, DyCause handles logs arriving from multiple applications (multiple source). It describes that the applications send the log periodically, meaning the data arrival rates are fixed.

- Challenge (DS 4): *Dealing with concept drift*

  In CloudRanger, MicroCause, and DyCause, all models, including the anomalous causal graphs, are generated whenever an outlier is detected. In other words, the generation of these models takes place over time, allowing these techniques to always reflect the latest distribution of their data and thus be immune from concept drift.

2.2.4 Summary of the Properties of the Techniques that Discover the Root Cause Factors of Outliers

In Table 2-2, we summarize the properties of the reviewed techniques that discover root cause factors of outliers. Our summary is based on the following aspects:

i. the name of the proposed techniques

ii. the referenced papers (including the publication year)

iii. the outlier types they are originally proposed for

iv. whether they are applied to static data or data streams

v. how the causal graphs used to find the root causes are constructed

vi. whether they handle the data streams-related challenge (DS 1): the notion of time)

vii. whether they the notion of infinity (DS 2)

viii. whether they handle different data arrival rates (DS 3)

ix. whether they handle concept drift (DS 4)

x. whether the handle the additional research challenge related to the outlier's root cause factor generation (RC 1): providing counterfactual

xi. whether they incorporate user knowledge about variables' causal dependency (RC 2)

xii. whether they apply parallelism

Recall in Section 2.1.3, we explain why parallelism is important for data streams. It basically helps speeding up the computation process.

We first summarize Table 2-2 based on the reviewed techniques for identifying the root cause factors of outliers discussed in this chapter. Table 2-2 shows that only one reviewed technique is intended for point outliers, while the other four are designed for collective outliers. Two techniques are applied to static data, and the other three are for data streams. Three techniques construct causal graphs using collected log data, while the other two employ user-defined causal graphs. All the reviewed techniques, except CausalRCA, deal with the notion of time. DyCause is the only technique that addresses the notion of infinity. Three techniques handle concept drift. Only one technique, CausalRCA, provides

counterfactual-based root cause factors. Two techniques incorporate user knowledge about variables' causal dependencies. Finally, three techniques apply parallelism.

At the bottom row of Table 2-2, we introduce our proposed technique Ocular (further detailed in Chapter 4), which identifies root cause factors of point outliers in data streams. Compared to the reviewed techniques, Ocular addresses all challenges related to data streams except DS 3 (handling varying data arrival rates). Notably, none of the reviewed techniques address DS 3. Ocular operates independently on each data stream, thus disregarding differences in data arrival rates. Furthermore, Ocular identifies outlier root cause factors based on counterfactual analysis (RC 1) and incorporates user knowledge regarding causal dependencies among variables (RC 2). Evidently, Ocular stands out as the sole technique for generating outlier root cause factors that addresses most of the research challenges outlined in Section 1.5 of Chapter 1. Additionally, Ocular applies parallelism.

Table 2-2 Summary of the properties of the techniques that discover root cause factors of outliers

Name	Year, Ref	Outlier Type	Data Type	Causal Graph	Data Stream Related Challenges (DS)				Root Cause Factors Related Challenges (RC)		Parallelism
					1	2	3	4	1	2	
CloudRanger	2018, [13]	Collective Outlier	Stream	causal graph constructed from log data	√	√		√			
MicroCause	2020, [28]	Collective Outlier	Stream	causal graph constructed from log data	√			√			√
CausalRCA	2022, [87]	Point Outlier	Static	user-defined causal graph					√	√	√
EasyRCA	2023, [88]	Collective Outlier	Static Timeseries Data	user-defined causal graph	√					√	
DyCause	2023, [46]	Collective Outlier	Stream	causal graph constructed from log data	√			√			√
Ocular (proposed in Chapter 4 of this dissertation)		Point Outlier	Stream	user-defined causal graph	√	√		√	√	√	√

CHAPTER 3

Discovering Outlying Attributes of Outliers in Data Streams: EXOS

Data streams, continuous sequences of timestamped data points, necessitate real-time monitoring due to their time-sensitive nature. In various data stream applications, such as network security and credit card transaction monitoring, real-time detection of outliers is crucial, as these outliers often signify potential threats. Equally important is the real-time explanation of outliers, enabling users to glean insights and thereby shorten their investigation time. The investigation time for outliers is closely tied to their number of attributes [12], making it essential to provide explanations that detail which attributes are responsible for the abnormality of a data point, referred to as outlying attributes. However, the notion of time, notion of infinity, and concept drift of data streams pose challenges for discovering the outlying attributes of outliers in real time. In response, in this Chapter, we propose EXOS, an algorithm designed for discovering the outlying attributes of multi-dimensional outliers in data streams.

EXOS leverages cross-correlations among data streams, accommodates varying data stream schemas and arrival rates, and effectively addresses challenges related to the notion of time, notion of infinity and concept drift. The algorithm is model agnostic for outlier detection and provides real-time explanations based on the local context of the outlier, derived from time-based tumbling windows. This Chapter also provides a complexity analysis of EXOS and an experimental analysis comparing EXOS with existing algorithms.

The evaluation includes assessing their performance on both real-world and synthetic datasets in terms of average precision, recall, F1-score, and explanation time.

The remainder of this Chapter is organized as follows: Section 3.1 formally defines the problem with the underlying assumptions; Section 3.2 provides an overview of EXOS; Section 3.3 explains the algorithm in detail; Section 3.4 presents a time and space complexity analysis of EXOS; and Section 3.5 discusses the experimental performance evaluation.

3.1 Preliminaries

This section formally defines the problem of finding outlying attributes of an outlier in data streams.

Definition 3.1 (Data Stream). A data stream denoted as S is an infinite sequence of data points $\{\mathbf{x}_i | i \geq 0\}$. Each $\mathbf{x}_i$ is a tuple of length $d + 1$ denoted as $\mathbf{x}_i = \langle x_1, x_2, \ldots, x_d, t \rangle$ where d is the number of attributes, x_r is the value of the *r*-th attribute, and t is the associated timestamp when the tuple is recorded or collected.

We consider a set of m concurrent or non-concurrent data streams $\mathbb{S} = \{S_1, \ldots, S_m\}$ where $m \geq 1$ and each number 1, ..., m also represents a data stream ID. Each data stream has a corresponding outlier detector. The *i*-th data point in a data stream j (S_j), is denoted as $\mathbf{x}_i^j$. We denote $\mathbf{x}_i^j$ as $\mathbf{o}_i^j$ when $\mathbf{x}_i^j$ is detected as an outlier. The attributes in S_j can differ from those in S_k. The total number of attributes in S_j is denoted as d_j. The set of attributes in S_j is denoted as $\mathcal{D}_j = \{X_{j1}, X_{j2}, \ldots, X_{jd_j}\}$. The total number of attributes of all m data streams is denoted as $D = d_1 + d_2 + \cdots + d_m$.

Data streams arrive continuously and are inherently unbounded; hence, keeping all data in the memory for real-time processing is impossible. To address this challenge, EXOS employs a time-based tumbling window [53] for each data stream. A tumbling window (TW) is a fixed size window that segment data streams into non-overlapping contiguous chucks of time. It serves as a buffer where all data points in the buffer remain available for a duration of time which is determined by the window size. When the current window expires, and a new window starts to receive new set of data points that arrive in the next period of time. To illustrate, Figure 3-1 depicts expired and active tumbling windows of window size 15 minutes. The data points from the timestamps 7:45 am-8:00 am are held in the memory for 15 minutes, after which they are replaced by the data points from the timestamps 8:00 am-8:15 am. We formally define a tumbling window in Definition 3.2.

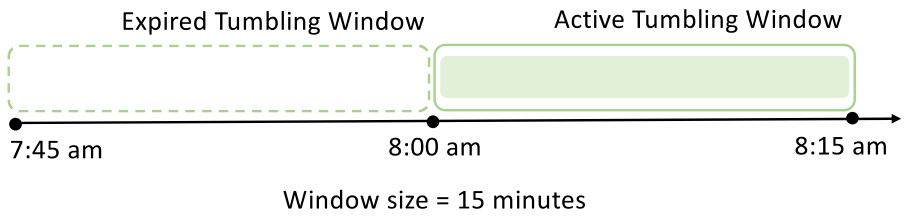


Figure 3-1 Tumbling window's illustration

Definition 3.2 (A Tumbling Window) A tumbling window (TW) can be mathematically defined by the following components:

i. window size, denoted as tws, which is a constant representing the length of each TW

ii. the start time of the p-th TW, denoted as T_p, which can be expressed as $T_p = T_0 + p$ tws, where T_0 is the start time of the first or initial TW. In this expression, $T_0 + p$ tws represents the timestamp after p times window size.

iii. the end time of the p-th TW, denoted as T_{p+1}, which can be expressed as $T_{p+1} = T_p + tws$. In this expression, $T_p + tws$ means the timestamp after one window size.

A data point **x** with timestamp t belongs to the p-th TW if $T_p \leq t < T_{p+1}$. In other words, the p-th tumbling window contains a sequence of data points whose timestamps are between T_p inclusively and T_{p+1} exclusively.

We refer multiple data streams with different data arrival rates as non-concurrent data streams. However, we assume that all data streams have the same tumbling window size, and they are all initialized at the same time.

Definition 3.2 (Outlying Attributes) Given an outlier o, a set of *d* dimensions (attributes) $\mathcal{D} = \{X_1, X_2, \ldots, X_d\}$ where $\mathbf{o} \in X_1 \times X_2 \times \cdots \times X_d$, an outlying contribution score function $h: \mathcal{D} \rightarrow \mathbb{R}$ that generates a real-value quantifying the contribution of each attribute to the abnormality of **o**, and an outlying contribution score threshold $\gamma \geq 0$, outlying attributes of $\mathbf{o} = \{X_i \in \mathcal{D} \mid h(X_i) > \gamma\}$.

Problem Definition: Given a set of *m* multi-dimensional data streams $\mathbb{S} = \{S_1, S_2, \ldots, S_m\}$, an outlier $\mathbf{o}_i^j$ identified by an arbitrary outlier detector at data stream j, an outlying contribution function h , and an outlying contribution score threshold $\gamma \geq 0$, the problem is to find the outlying attributes of $\mathbf{o}_i^j$ accurately and efficiently.

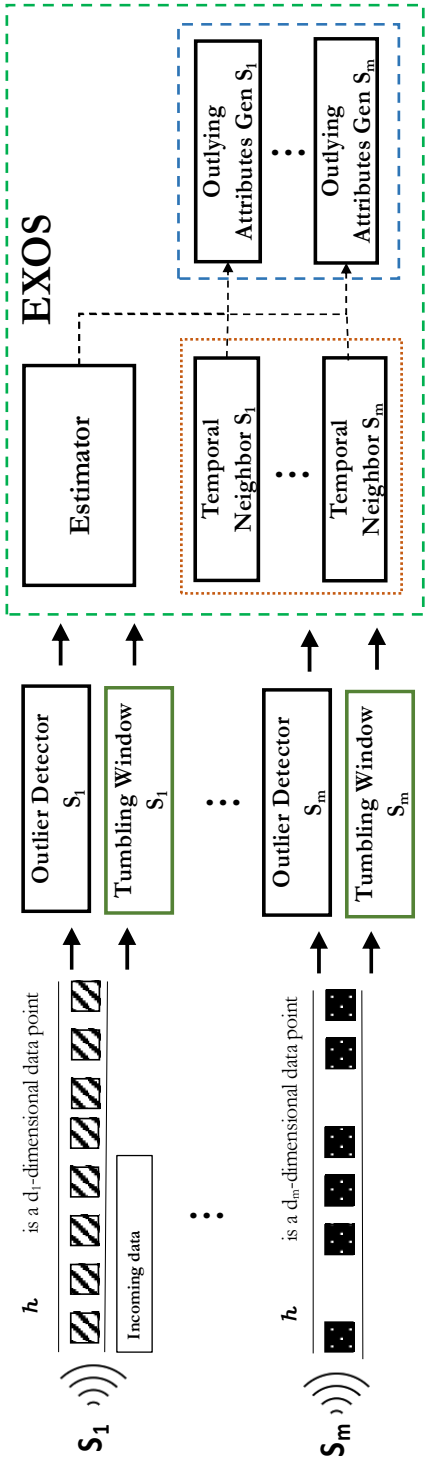


Figure 3-2 The interaction between outlier detectors, tumbling windows, and EXOS

3.2 The EXOS Overview

EXOS assumes each data stream is associated with an outlier detector and a tumbling window. Figure 3-2 illustrates the interaction between the outlier detectors, tumbling windows, and EXOS. In this figure, data stream 1 (S_1) is associated with an outlier detector and a tumbling window (TW), as are the other data streams. Recall that EXOS utilizes tumbling windows (TWs) to manage the continuous arrival and unbounded volume of data streams (the notion of infinity). EXOS assumes that all TWs across all data streams have the same window size. In each stream, arriving data points are consumed by the corresponding outlier detector and tumbling window. At the end of the active period of the TWs, EXOS absorbs the data points within the active TWs into its Estimator and Temporal Neighbor components. The Estimator component absorbs data points from the active TWs of all streams, while the Temporal Neighbor Component absorbs the data points from each active TW of data stream j into its corresponding independent temporal neighbor function.

While absorbing data points from the active TWs, EXOS will also check whether the outlier detectors identify any outliers during within the duration of the active TWs. If any outliers are detected, EXOS will activate its third component: Outlying Attributes Generator. This component takes the outputs from the Estimator and Temporal Neighbor components as its inputs and proceeds to find the outlying attributes of each outlier detected in each stream.

Like COIN [51][62], EXOS generates the outlying attributes of each outlier based on its local neighborhood (local context). This approach allows EXOS to avoid the subspace

search. Based on the definition that an outlier is a data point that is significantly different than the rest of the data points, COIN uses the closest normal data points (inliers) to an outlier to serve as the outlier's local context. However, its neighborhood computation requires storing the entire set of data points, which is impractical in a data stream environment. Additionally, the notion of time implies that neighborhood computation should consider the timestamp of the outlier. Thus, identifying the local context for each outlier in a stream setting is not straightforward.

To handle neighborhood computation in the stream setting, EXOS estimates the normal value of each outlier using a model that captures the correlation among the attributes within the data stream where the outlier is detected as well as across other data streams (cross-correlation). The model is updated periodically, specifically when EXOS absorbs new data points from active TWs, to ensure it captures changes in data distribution (concept drift). This estimation of the normal values of each outlier and the model update are performed by the Estimator component. While Estimator is doing its job, the other component of EXOS, the Temporal Neighbor, clusters data points from each data stream independently in an incremental way. The clusters do not retain the original data points but only maintain the summary or statistics of the absorbed data points.

The Outlying Attributes Generator component takes as input the estimated normal value of each outlier and statistics derived from clusters within the data streams. It operates with m parallel independent functions, each dedicated to a specific data stream. Each function j within this component is responsible for constructing a local context for each outlier from data stream j (S_j), using both the estimated normal value of the outlier and the clusters identified within S_j. Once the local context is established, the function determines

the outlier's class. Subsequently, the Outlying Attributes Generator function trains a linear classifier to differentiate the local context from the outlier class. The weights assigned to each attribute in the classifier's decision boundary are then used to identify which attributes constitute outlying attributes of the outlier.

3.3 The EXOS Algorithm

Algorithm 3.1 shows the EXOS algorithm. It consists of the initialization phase (Lines 1-2) and the Online Phase (Lines 3-24). The initialization phase consists of two tasks: (i) initializing eigenvectors **Q** and (ii) setting the list of the initial clusters' centroids in all streams (Line 2). EXOS makes use of the cross-correlation characteristics of data streams to determine the outlying attributes of outliers in each stream. It captures the cross-correlation by applying a single-pass incremental Principal Component Analysis (PCA)-based technique to the data in the active windows. This technique requires the initialization of *k* eigenvectors or principal components **Q** before it runs online.

We apply naive PCA [96] on offline data to generate the initial eigenvectors **Q** (Line 1). The offline data consist of initial data of all data streams. Recall that the total number of attributes of all data stream is D. The Naive PCA first obtains a D×D correlation matrix by calculating the correlation coefficients between each pair of attributes in the offline data. It then applies an eigenvalue decomposition to the correlation matrix. The decomposition process factorizes the correlation matrix into its eigenvalues and eigenvectors. Eigenvalues indicate the amount of variance explained by each principal component (or eigenvector). Larger eigenvalues correspond to components that capture more variance. Eigenvectors represents the directions in the original attribute space that capture maximum variance.

Selecting k first principal components means selecting the first k eigenvectors associated with the k largest eigenvalues. To this end, the initial $\mathbf{Q}$ is a $D \times k$ eigenvectors matrix where $\mathbf{Q} \in \mathbb{R}^{D \times k}, k < D$.

Moreover, the offline data are also clustered using the k-means clustering algorithm [97] (Line 2). The clusters' centroids are stored in the AllClust data structure to be used by the Temporal Neighbor Clustering component. AllClust is a list of m sets of clusters of m data streams. Each cluster in the set of clusters is represented by its centroid and the number of data points it has.

The Online Phase indicated in Lines 3-23. Before a new tumbling window starts, EXOS consumes data points from all m active tumbling windows and store it into a data structure called TWinS (line 6). TWinS is a list of m groups of data points, where a group j contains data points from data stream j's active tumbling window. Recall that all active tumbling of all data streams has the same tumbling window size, starting time, and ending time. EXOS also obtains all outliers detected during the period of all active tumbling windows and stores it in a data structure called AllOutliers (Line 7). AllOutliers is a key value set, where a key represents a data stream's ID and its corresponding value represent a set of outliers whose timestamp within the duration of the active tumbling window of the particular stream ID.

Lines 8-19 show how the three key components of EXOS, (i) Estimator *Est* (Line 9), (ii) Temporal Neighbor Clustering (Line 10-13) and (iii) Outlying Attributes Generator (Lines 15-19), are connected. The *Est* component uses the eigenvectors $\mathbf{Q}$ to estimate the normal values of outliers. It incrementally updates $\mathbf{Q}$ whenever it absorbs data points from all active tumbling windows to ensure that the changes of data distribution, known as the

concept drift, are reflected in **Q**. Running concurrently with *Est*, the Temporal Neighbor component consists of *m* parallel clustering functions, $\mathcal{C}f_1, \mathcal{C}f_2, \ldots, \mathcal{C}f_m$. Each $\mathcal{C}f_j$ incrementally absorbs each data point in the active tumbling window of S_j (TW_j) into a cluster by selecting the cluster whose centroid has the closest distance to the data point.

To deal with the issue of unbounded volume of data under the constraints of limited memory, EXOS does not keep the original data in the clusters. It only retains the centroids and the number of data points the clusters. The Outlying Attribute Generator component also features *m* parallel functions, $\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_m$. Each $\mathcal{G}_j$ corresponds to a data stream j (S_j). It uses the estimated normal values of outliers from *Est* and the latest centroids from $\mathcal{C}f_j$ to identify the outlying attributes of each outlier $\mathbf{o}_i^j$ detected in data stream j.

Algorithm 3.1: **EXOS** (𝕊, k, D, γ, ds_attrs, tws, init_data, n_clusters, outlier_queues)
Input: 𝕊 the set of *m* concurrent data streams, *k* the number of eigenvectors, *D* the total number of attributes, γ the outlying contribution threshold, *ds_attrs* the list of the attributes in each stream, *tws* the window size, *init_data* the offline data, *n_clusters* the list of number of clusters in each data stream, *outlier_queues* the list of *m* queues storing the outliers of the *m* streams.
Output: F_j the set of outlying attributes of each outlier in each data stream j
Initialization 1 **Q** := initialize_eigen_vectors (init_data) 2 AllClust := initialize_set_of_clusters (init_data, n_clusters) 3 *end_ts* := current_time () 4 *start_ts* := end_ts – tws **## Start the Online Phase** 5 **while (true):** ## repeat indefinitely 6 TWinS = get_active_tumbling_windows_data_points (𝕊, start_ts, end_ts) 7 AllOutliers = get_outliers_from_each_stream (outlier_queues, start_ts, end_ts) 8 **do in parallel** **##The Estimator Component** Q, *est_list* := **Estimator** (TWinS, k, D, **Q**, tws, AllOutliers, *arrival_rates, ds_attrs*) 9 **##The Temporal Neighbor Component** 10 **parallel for** j **in** [1: m]

```
                AllClust[j]:= TemporalNeighborClusteringAtSj (TWinS[j],AllClust[j],
11                                                    n_clusters[j])
12              end parallel for
13          end do in parallel
            ## The Outlying Attribute Generator Component
14          parallel for j in [1: m]
15              Ôj := est_list[j] ## get estimated normal values of the outliers in stream j
16              Oj := get_outlier_values (AllOutliers[j])# get outlier values in stream j
17              OutAttrListj := OutlyingAttrGeneratorAtSj (Oj, Ôj, AllClust[j], γ, ds_attrs[j])
                ## return the outlying attributes of the outliers in the current window of data
                 stream j
18              yields(OutAttrListj)
19          end parallel for
20          while (end_ts + tws > current_time()) : wait() end while
21          end_ts = current_time (); start_ts = end_ts – tws
22      end while
```

We now describe the three key components of EXOS.

3.3.1 The Estimator Component

The Estimator Component, denoted as *Est*, is responsible for approximating the normal value $\hat{\mathbf{o}}_i^j$ of each outlier $\mathbf{o}_i^j$ in each tumbling window. This task is accomplished by leveraging the attribute correlations within the data stream j (S_j) and considering the inter-attribute correlations across other data streams. To illustrate this process, imagine a scenario involving three-source data streams depicted in Figure 3-3. In this scenario, the data streams S_1, S_2, and S_3 have three attributes (X_{11}, X_{12}, X_{13}), two attributes (X_{21}, X_{22}), and two attributes (X_{31}, X_{32}), respectively, and their data points arrive every 1, 2, and 4 minutes, respectively. The data are observed between the time frame of 8:00 am and 8:15 am, during which the outlier detector at S_1 identifies an outlier at 8:03 am. Figure 3-3.a) and Figure 3-3.b) describe the data points in the three streams before and after estimating the normal values of outliers, respectively. The normal data points in each stream are

represented by small circle shapes without any text. The outlier detected at 8:03 am in S_1 is represented as a small red circle and denoted as $\mathbf{o}_{t+2}^{1}$ in Figure 3-3.a) and b). Its corresponding normal value estimation is the small dotted green circle denoted as $\hat{\mathbf{o}}_{t+2}^{1}$ in Figure 3-3.b).

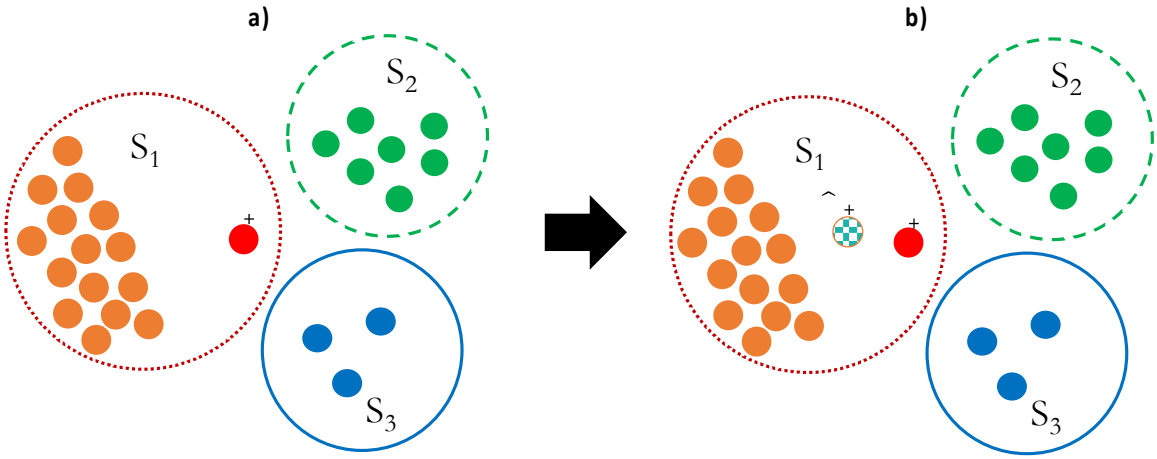


Figure 3-3 Illustration of three data streams and their outliers

In the quest to find the estimated normal values of outliers within an active tumbling window, each time an old tumbling window expires and a new one starts, *Est* aligns the data from active tumbling windows of all streams according to their timestamp and arrival rate. For instance, in the previously described scenario, the data alignment yields a 7×15 matrix depicted in Figure 3-4. The matrix showcases the data points from data streams S_1, S_2, and S_3 between 8:01 am and 8:15 am. Its rows represent the attributes from each stream while the matrix columns represent the timestamp steps of the active tumbling windows. The total number of the timestamp steps is determined by the data stream with the fastest arrival rate. In our previously described scenario, we have 15 timestamp steps from t to t+14 since S_1's data points arrive every one minute. It is important to note that when dealing

with a single data stream, there is only one active tumbling window. Hence, in this case, the data alignment contains all data points within the active tumbling window.

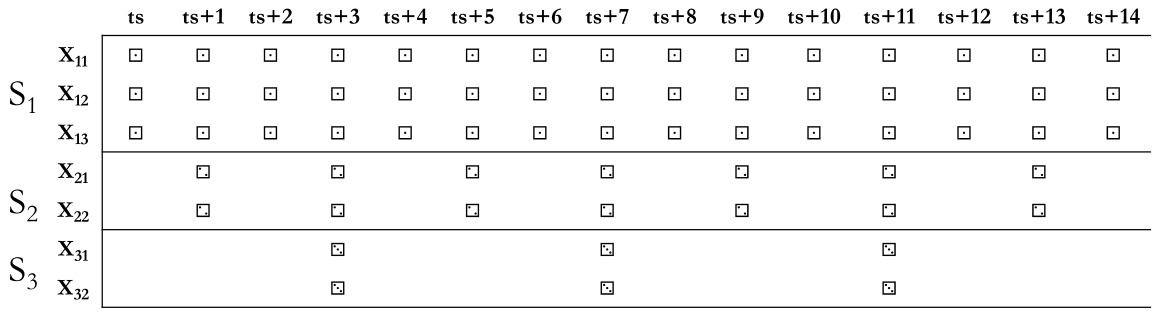


Figure 3-4 An example of data alignment results from three data streams of 3, 2, and 2 attributes in Figure 3-3.

In the situation where the arrival rates among all data streams are the same or when dealing with a single stream, *Est* utilizes all the aligned data from the active tumbling windows as the input to the PCA-based algorithm. However, when the data stream arrival rates differ from each other, *Est* selectively employs the aligned data corresponding to the timestamp steps that contain the data points from all streams. For instance, in our scenario, *Est* selects the data points associated with the timestamp steps t+3, t+7, and t+11 as the input to the PCA-based algorithm. Let us denote each selected aligned data point as $\mathbf{x_{aligned}} \in \mathbb{R}^{D}$. The vector representation of $\mathbf{x_{aligned}}$ is given by:

$$\mathbf{x_{aligned}} = \begin{bmatrix} x_{11} \\ x_{12} \\ x_{13} \\ x_{21} \\ x_{22} \\ x_{31} \\ x_{32} \end{bmatrix}$$

where $x_{11}, x_{12}, x_{13}, x_{21}, x_{22}, x_{31}$ and x_{32} represent the values of attributes X_{11}, X_{12}, X_{13}, X_{21}, X_{22}, X_{31}, and X_{32}, respectively.

Est uses a PCA-based approach which is primarily used to transform high-dimensional data into lower-dimensional space while preserving the most important information [98]. PCA achieves this by identifying orthogonal axes (also known as principal components or eigenvectors) along which the variance in the data is maximized. It focuses on capturing the variance in the data, with the first eigenvector explaining the most variance, the second explaining the second most, and so on. Each eigenvector is a linear combination of the attributes. In the process of identifying the eigenvectors, PCA takes into account the covariance between attributes. Attributes with higher covariance will have a stronger influence on the eigenvectors. In this sense, PCA can indirectly reveal the relationships among attributes, especially if there are strong correlations between them. High positive or negative correlations are reflected in the eigenvectors.

Whenever *Est* handles a new active tumbling window, its PCA part derives *k* eigenvectors (principal components) from the selected aligned data of the active tumbling window. The continuous update on *k* eigenvectors allows *Es*t to adapt to concept drift in the data streams. Arguably, *Est* can use the naive PCA to generate the eigenvectors in every window. However, the naive PCA ignores the data seen in the previous windows and requires multiple passes on the aligned data, which is not suitable to handle real-time continuous data streams with limited memory. Hence to update the *k* eigenvectors, we adopt DBPCA, a single-pass eigendecomposition algorithm described in (Li et al., 2016). Our adaption of DBPCA is described in the following paragraph and also shown in Algorithm 3.2 Lines 5-8.

Let the k eigenvectors be $\mathbf{Q}$, the matrix of size $\mathrm{D} \times \mathrm{k}$ where $\mathbf{Q} \in \mathbb{R}^{\mathrm{D} \times \mathrm{k}}, \mathrm{k} < \mathrm{D}$. From our previously described scenario and given *k*=1, $\mathbf{Q}$ then contains one eigenvector as follows:

$$\mathbf{Q} = \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \\ e_5 \\ e_6 \\ e_7 \end{bmatrix}$$

The values $e_1, e_2, e_3, e_4, e_5, e_6$ and e_7 correspond to the attributes X_{11}, X_{12}, X_{13}, X_{21}, X_{22}, X_{31}, and X_{32}, respectively. When a new tumbling window starts, the DBPCA part of *Est* initializes a zero matrix $\mathbf{Z}$ of size $\mathrm{D} \times \mathrm{k}$ and then for each selected aligned data point $\mathbf{x_{aligned}} \in \mathbb{R}^{\mathrm{D}}$, it incrementally updates Z using the following equation:

$$\mathbf{Z} = \mathbf{Z} + \frac{1}{\mathrm{N}} \mathbf{x_{aligned}} \mathbf{x_{aligned}}^{\mathrm{T}} \mathbf{Q}$$

where N is the number of selected aligned data points, $\mathbf{x_{aligned}}^{\mathrm{T}}$ is the transpose of $\mathbf{x_{aligned}}$, and $\mathbf{Q}$ is the matrix containing the *k* eigenvectors from the combined active m tumbling windows. After conducting a single-pass iteration over all selected aligned data points to update $\mathbf{Z}$, *Est* then computes the new $\mathbf{Q}$ using the Q-R decomposition of $\mathbf{Z}$.

After obtaining the updated $\mathbf{Q}$, *Est* duplicates it and retains the new eigenvectors $\mathbf{Q}$ in memory to serve as the initial $\mathbf{Q}$ for the subsequent window. The duplicated $\mathbf{Q}$ is then sliced into m distinct segments: $\mathbf{Q_1}, \cdots, \mathbf{Q_j}, \cdots \mathbf{Q_m}$. In our example, the copied $\mathbf{Q}$ is segmented into three parts: $\mathbf{Q_1}, \mathbf{Q_2}$, and $\mathbf{Q_3}$. Each segment corresponds to a specific stream as follows:

$$\mathbf{Q_1} = \begin{bmatrix} e_1 \\ e_2 \\ e_3 \end{bmatrix}, \mathbf{Q_2} = \begin{bmatrix} e_4 \\ e_5 \end{bmatrix}, \text{ and } \mathbf{Q_3} = \begin{bmatrix} e_6 \\ e_7 \end{bmatrix}$$

Each segmented eigenvector associated with data stream j ($\mathbf{Q_j}$) is then utilized to estimate the normal value of each outlier $\mathbf{o}_i^j$ in data stream j (S_j), using the following matrix operation:

$$\widehat{\mathbf{o}}_i^j = \mathbf{o}_i^j \mathbf{Q_j} \mathbf{Q_j^T}$$

In our previous example, the estimated normal value of the outlier detected at 8:03 am in data stream S_1 is calculated as follows:

$$\widehat{\mathbf{o}}_{t+2}^1 = \mathbf{o}_{t+2}^1 \mathbf{Q_1} \mathbf{Q_1^T} = [x_{11} \quad x_{12} \quad x_{13}] \begin{bmatrix} e_1 \\ e_2 \\ e_3 \end{bmatrix} [e_1 \quad e_2 \quad e_3]$$

Furthermore, when there is more than one outlier in the TW_j (the active tumbling window of a data stream j), the outliers can be consolidated into a matrix $\mathbf{O^j} \in \mathbb{R}^{\#o^j \times d_j}$ where $\#o^j$ is the total number of outliers in S_j in its active tumbling window and d_j is the number of attributes in S_j. The estimated normal values of the outliers in the TW_j become:

$$\widehat{\mathbf{O}}^{\mathbf{j}} = \mathbf{O^j} \mathbf{Q_j} \mathbf{Q_j^T}$$

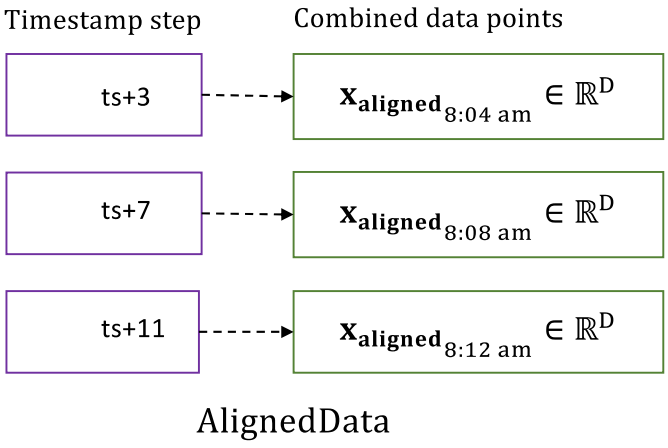


Figure 3-5 Illustration of the AlignedData data structure based on the example in Figure 3-4

Algorithm 3.2 summarizes how *Est* works. In Line 1, the algorithm computes the total number of timestamp steps required to align the data points from the data streams' active windows. It then processes the active windows to get the selected aligned data points (AlignedData**)** (Line 2). The AlignedData data structure of our example in Figure 3-4 is depicted in Figure 3-5.

Algorithm 3.2: **Estimator** (TWinS, k, D, **Q**, tws, AllOutliers, arrival_rates, ds_attrs)

Input: *TWinS* A list of m groups of data points, where a group j obtains data points from data stream j's active tumbling window, *k* the number of eigenvectors, D the total number of attributes, **Q** the previous eigenvectors, tws the tumbling window size, AllOutliers the key-value set, where a key represents a data stream's ID and its corresponding value represents a set of outliers whose timestamp with the duration of the active tumbling window of the particular stream ID, *arrival_rates* the list containing the arrival rates of all data streams. *ds_attrs* the list containing the numbers of attributes of all data streams

Output: updated eigenvectors **Q**, list of estimated normal values of outliers *est_list*

```
    ## compute the total number of time stamp steps for aligned data
1   n_ts_step = tws /min(arrival_rates)
    ## align data points from all data streams' active tumbling windows
2   AlignedData = get_selected_aligned_data (TWinS, n_ts_step)
3   N̄ = |AlignedData| ## get the number of selected timestamp steps
4   Initialize a zero matrix Z of size D × k
5   for each ts_step in AlignedData do
6       x_aligned = AlignedData[ts_step]
7       Z = Z + (1/N) x_aligned x_aligned^T Q
8   end for
9   Q = QR-decomposition of Z
    ## copy and divide eigenvectors into m segments
10  Q_segments = slice_eigenvectors(Q, ds_attrs)
11  Set est_list as a list of length |TWinS|
12  parallel for 1 ≤ j ≤ |TWinS| do:
```

```
13      Q_j = Q_segments[j]
14      O^j  =  get_outlier_matrix (AllOutliers[j])
15      Ô^j = O^j Q_j Q_j^T
16      est_list[j] = Ô^j
17  end parallel for
18  return Q, est_list
```

Lines 4-9 are the DBPCA approach responsible for updating the eigenvectors matrix **Q** based on the combined data points from all streams in AlignedData. The DBPCA goes through each **x_aligned** $\in \mathbb{R}^D$ in AlignedData to update **Z** (Line 7) and then computes the new **Q** (Line 9). The copied new **Q** is sliced into *m* segments (Line 10) and each segment of **Q** is then used to estimate the normal values of the outliers in its corresponding data stream (Lines 11-17). The algorithm returns the new eigenvectors matrix **Q** and the list of the estimated normal values of the outliers.

3.3.2 The Temporal Neighbor Clustering Component

The temporal neighbor clustering component is a set of *m* independent functions $\mathcal{C}f_1, \mathcal{C}f_2, \ldots, \mathcal{C}f_m$ that runs in parallel with the estimator component. Each $\mathcal{C}f_j$ receives data points from the active tumbling window of data stream j (TW_j) and integrates each data point into a cluster. In regular or static dataset, the neighborhood computation of an outlier requires all data points to be available. However, this approach is impractical for data streams, where data points continuously arrive, making it impossible to store all data points in the memory. To address this issue, the temporal neighbor clustering function for data stream j ($\mathcal{C}f_j$) has the role to maintain statistics of data points without storing them all. These cluster statistics are later used to form a local context for each outlier detected in the

same tumbling window. By using cluster statistics updated from the same tumbling window in which the outlier is detected, we ensure that the outlier's local context (the selected neighborhood of the outlier) is both relevant and timely.

We now describe how the temporal neighbor clustering function for data stream j ($\mathcal{C}\mathrm{f_j}$) works. As depicted in Figure 3-6, each data point from the active tumbling window of data stream j ($\mathrm{TW_j}$) is grouped into a set of clusters $\mathbb{C}_\mathrm{j} = \{\mathrm{C}_{1j}, \dots, C_{l_j}\}$ such that $C_{aj} \cap C_{bj} = \emptyset$ for all $a \neq b$ and the number data points in $\mathrm{TW_j} = the\ number\ of\ data\ points\ that\ all\ clusters \bigcup_{aj=1}^{lj} \mathrm{C_{aj}}$ absorb. $\mathbb{C}_\mathrm{j}$ acts as the temporal neighbors of outliers that belongs in $\mathrm{TW_j}$. The number of clusters in $\mathbb{C}_\mathrm{j}$ is denoted as lj. Due to the unbounded amount of data points in data streams and the limited memory, the clusters cannot retain the data points in the memory. Hence, each cluster $C_{aj} \in \mathbb{C}_\mathrm{j}$ only keeps the number of points in the cluster ($\mathrm{num\_points_{aj}}$) and C_{aj}, and the centroid of the cluster ($\mathbf{centroid_{aj}} \in \mathbb{R}^{\mathrm{d_j}}$).

The distance between a data point $\mathbf{x} \in \mathrm{TW_j}$ and a cluster's centroid $\mathbf{centroid_{aj}}$ can be represented as a distance function $d(\mathbf{x}, \mathbf{centroid_{aj}})$. By excluding the timestamp from **x**, the distance function denoted as $d(\mathbf{x}, \mathbf{centroid_{aj}})$ is formulated as $d(\mathbf{x}, \mathbf{centroid_{aj}}) = \|\mathbf{x} - \mathbf{centroid_{aj}}\|$. The data point **x** is assigned to a cluster $\mathrm{C_{aj}} \in \mathbb{C}_\mathrm{j}$ such that the distance between **x** and $\mathbf{centroid_{aj}}$ is minimized.

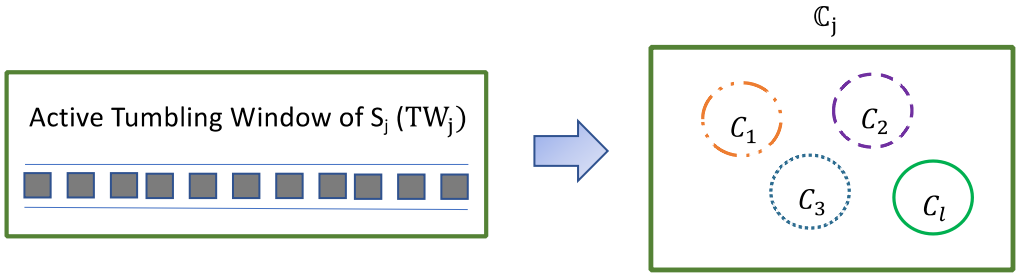


Figure 3-6 Grouping data points in the active tumbling window W_j into a set of clusters

Algorithm 3.3 delineates the operations within an independent temporal neighbor clustering function $\mathcal{C}f_j$. This function takes the active tumbling window of data stream j (TW_j) and the initial centroids of all clusters in the window as inputs. It starts by setting each cluster's centroid to its corresponding initial centroid and setting the total number of data points in each cluster to one (Lines 1-3). Given the continuous arrival of data points necessitating a single-pass computation, forming clusters of data points in the window is done incrementally. The sequential *k*-means algorithm [100] is employed to update the set of clusters $\mathbb{C}_j$ (Lines 4-8). The algorithm iterates through each data point from TW_j. For each iteration, it assigns a data point **x** to the nearest cluster (Line 5). Subsequently, it updates the properties of the assigned cluster, by incrementing its total number of data points num_points_{aj} by one (Line 6) and updating its $\mathbf{centroid_{aj}}$ (Line 7) using the formula:

$$C_{aj}.\mathbf{centroid_{aj}} = C_{aj}.\mathbf{centroid_{aj}} + \frac{\mathbf{x} - C_a.\mathbf{centroid_{aj}}}{C_{aj}.num\_points_{aj}}.$$

When the algorithm finishes iterating over each data point once, it returns the set of clusters $\mathbb{C}_j$.

Algorithm 3.3: **TemporalNeighborClusteringAtS$_j$**(TW$_j$, init_centroids, l_j)

Input: TW$_j$ the active window of S_j, *init_centroids* the list of initial centroids of clusters, l_j the number of clusters in data stream j

Output: $\mathbb{C}_j$ the set of lj clusters of the active window

1 set $\mathbb{C}_j$ as a set of lj empty clusters $C_{1j}, \dots, C_{lj}$

2 set num_points$_{aj}$ of each $C_{aj} \in \mathbb{C}_j$ to 1

3 set centroids of each $C_{aj} \in \mathbb{C}$ to its corresponding init_centroids

4 **for x** $\in$ TW$_j$ **do**

5 ## assign **x** into the closest centroid
argmin$_{C_{aj} \in \mathbb{C}_j}$ $\boldsymbol{d}(\mathbf{x}, \mathbf{centroid_{aj}})$

6 ## update the total number of points the selected cluster C_{aj}
C_{aj}.num_points$_{aj}$ = C_{aj}.num_points$_{aj}$ + 1

7 ## update the centroid of the selected cluster C_a
C_{aj}.**centroid$_{aj}$** = C_{aj}.**centroid$_{aj}$** + $\frac{\mathbf{x} - C_a.\mathbf{centroid_{aj}}}{C_a.\text{num\_points}_{aj}}$

8 **end for**

9 **return** $\mathbb{C}_j$

Incorporating the sequential k-means algorithm, allows each independent function $\mathcal{C}f_j$ of the Temporal Neighbor Component to conduct a single pass incremental computation. As we can see, each data point in the active tumbling window of data stream j is processed one at a time. One could argue that the process of assigning a data point into a cluster in $\mathbb{C}_j$ in Line 5, requires lj distance computation between the data point and the centroids of lj clusters that can be done in parallel. Thus, the process can be seen as a single pass. Furthermore, unlike traditional k-means algorithm which require multiple set over the dataset to update the centroids of the clusters, the sequential k-means algorithm updates the centroids immediately after processing each individual data point. By processing each

data point individually and updating the centroids immediately, the sequential k-means algorithm avoids the need to store the entire dataset in the memory.

3.3.3 The Outlying Attribute Generator Component

The outlying attribute generator component is a set of m independent functions $\{\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_m\}$. Each $\mathcal{G}_j$ corresponds to a data stream j (S_j) and receives the inputs produced by the estimator component *Est* and the temporal neighbor component $\mathcal{C}f_j$. It is activated whenever a tumbling window associated with data stream j (TW_j) contains one or more outliers. Est provided the estimated normal value of each outlier detected in TW_j while $\mathcal{C}f_j$ provided the clusters of data points from TW_j. Together these inputs are used by $\mathcal{G}_j$ to form a local context of each outlier detected in TW_j, generating outlying attributes of each outlier.

We now describe how the outlying attribute generator function for data stream j ($\mathcal{G}_j$) forms the local context of an outlier $\mathbf{o_i^j}$ detected in TW_j and then generates outlying attributes for the outlier. We call the local context of the outlier $\mathbf{o_i^j}$ as the inlier class of the outlier.

Figure 3-7 depicts how the inlier class is formed for an outlier $\mathbf{o_i^j}$. The inlier class is generated by finding a cluster $C_{aj} \in \mathbb{C}_j$ whose centroid $\mathbf{centroid_{aj}}$ is the closest to the estimated normal values of the outlier ($\widehat{\mathbf{o}}_\mathbf{i}^\mathbf{j}$). We denote by $dist_{C_{aj}}^{i}$ the distance between the centroid of cluster C_{aj} ($\mathbf{centroid_{aj}}$) and $\widehat{\mathbf{o}}_\mathbf{i}^\mathbf{j}$. Recall that the cluster does not keep the data points it has absorbed, but it has the information about the number of data points absorbed by the cluster (num_points_{aj}). Initially, the inlier class only has two members:

centroid$_{aj}$ and $\hat{\mathbf{o}}_i^j$. The additional members are then added by generating *a*uxiliary data points whose distances to **centroid$_{aj}$** is less than $dist_{C_{aj}}^{i}$. The total number of generated *a*uxiliary data points is equal to num_points$_{aj}$ multiplied by the number of attributes in data stream j (d_j).

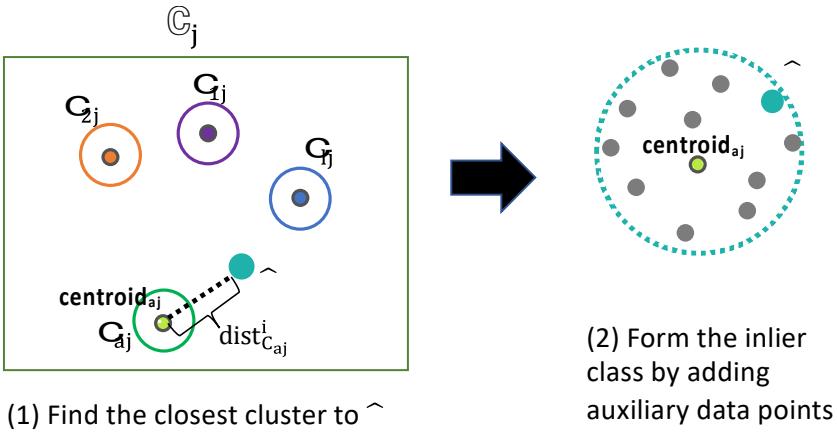


Figure 3-7 Forming the local context (inlier class) of an outlier $\mathbf{o}_i^j$

The process of generating the auxiliary data points is done by first creating a covariance matrix of size $d_j \times d_j$ by multiplying $dist_{C_{aj}}^{i}/3$ with an identity matrix of size d_j by d_j as follows:

$$\mathbf{Cov}_{inlier} = dist_{C_{aj}}^{i}/3 \cdot \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} = \begin{bmatrix} dist_{C_{aj}}^{i}/3 & 0 & \cdots & 0 \\ 0 & dist_{C_{aj}}^{i}/3 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & dist_{C_{aj}}^{i}/3 \end{bmatrix}$$

The decision to divide the distance by three is based on the empirical rule that in normally distributed data, nearly all of the data points fall within three standard deviations of the mean [101]. The multivariate normal data generator requires three input parameters: 1) a mean of d_j dimension; 2) a $d_j \times d_j$ covariance matrix, and 3) a number of data points to generate. We use $\mathbf{centroid_{aj}}$, $\mathbf{Cov}_{inlier}$, and $num\_points_{aj} * d_j$, as the parameters 1), 2), and 3) for the multivariate normal data generator, respectively.

So far, we have shown the creation of the local context of an outlier $\mathbf{o_i^j}$ that we also call the inlier class of $\mathbf{o_i^j}$**.** The inlier class of $\mathbf{o_i^j}$ consists of the estimated normal value of the outlier ($\widehat{\mathbf{o}}_\mathbf{i}^\mathbf{j}$**)** the centroid of the closest clusters $\widehat{\mathbf{o}}_\mathbf{i}^\mathbf{j}$ and auxiliary data points. All the members of the inlier class have the same attributes as the data points in stream j. For the purpose of training a classifier that separates the outlier and its local context, a label of value 0 is added to each member of the inlier class.

After generating the inlier class, the outlying attribute generator function for data stream j ($\mathcal{G}_j$) form an outlier class for the outlier $\mathbf{o_i^j}$**.** Figure 3-8 illustrates how the outlier class is formed. The outlier class is created by generating multivariate normal data centered around $\mathbf{o_i^j}$. To ensure that the members of the outlier class do not overlap with those of the inlier class, the standard deviation denoted as σ_o is set to be less than one-third of the distance between $\widehat{\mathbf{o}}_\mathbf{i}^\mathbf{j}$ and $\mathbf{o_i^j}$. The $d_j \times d_j$ covariance matrix of the outlier class is generated as follows:

$$\mathbf{Cov}_{outlier} = {\sigma_o}/{3} \cdot \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} = \begin{bmatrix} {\sigma_o}/{3} & 0 & \cdots & 0 \\ 0 & {\sigma_o}/{3} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & {\sigma_o}/{3} \end{bmatrix}$$

Similar to the inlier class data generation, to generate the outlier class we use $\mathbf{o_i^j}$, $\mathbf{Cov}_{outlier}$, and $num\_points_{aj} * d_j$ as the parameters 1), 2), and 3) for the multivariate normal data generator, respectively.

The generation of the multivariate data explained in the previous paragraph creates the outlier class of $\mathbf{o_i^j}$ that consists of $\mathbf{o_i^j}$ and auxiliary data points. All members of the outlier class have the same attributes as the data points in stream j. Additionally, each member of the outlier class has a label with a value of 1.

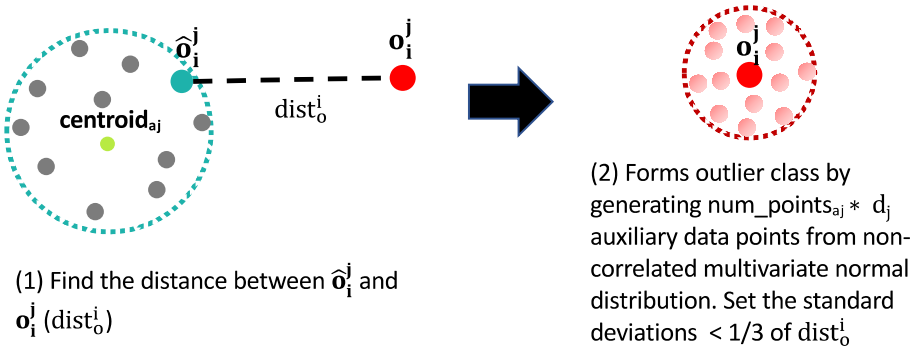


Figure 3-8 Forming the outlier class of an outlier $\mathbf{o_i^j}$

After forming the inlier and outlier classes, the subsequent step involves determining a decision boundary or hyperplane that separates both groups (illustrated in Figure 3-9) using a linear classifier. Specifically, we train a linear support vector machine (SVM) [102] using data from the inlier and outlier classes. The decision boundary is a linear combination of the data attributes where each attribute corresponds to a coefficient or a weight. The classifier applies lasso regularization [103] such that the resulting hyperplane has zero weights on any attributes that are not relevant in forming the boundary.

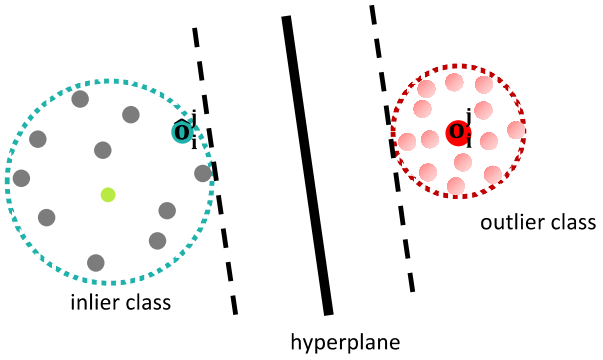


Figure 3-9 Finding the decision boundary that separates the inlier class from the outlier class

Given $\mathcal{D} = \left\{X_{j1}, X_{j2}, \dots, X_{d_j j}\right\}$ the set of attributes in S_j and d_j the total number of attributes, the hyperplane that separates the inlier class from the outlier class of $\mathbf{o_i^j}$ is formulated as $w_{j0}^i + w_{j1}^i X_{j1} + w_{j2}^i X_{j2} + \cdots + w_{jd_j}^i X_{jd_j} = 0$. The attribute weights associated with $\mathbf{o_i^j}$ on the hyperplane are expressed as a vector $\mathbf{H}_{\mathbf{o_i^j}} \in \mathbb{R}^{d_j}$ where:

$$\mathbf{H}_{\mathbf{o_i^j}} = \begin{bmatrix} w_{j1}^i \\ w_{j2}^i \\ \vdots \\ w_{jd_j}^i \end{bmatrix}$$

Once $\mathbf{H}_{\mathbf{o_i^j}}$ is obtained, the algorithm stores the absolute values of the hyperplane's weights in $\left|\mathbf{H}_{\mathbf{o_i^j}}\right| \in \mathbb{R}^{d_j}$ where:

$$\left|\mathbf{H}_{\mathbf{o_i^j}}\right| = \begin{bmatrix} |w_{j1}^i| \\ |w_{j2}^i| \\ \vdots \\ |w_{jd_j}^i| \end{bmatrix}$$

$\left|\mathbf{H}_{\mathbf{o}_i^j}\right| \in \mathbb{R}^{d_j}$ is then used to calculate the outlier's outlying contribution scores stored in the vector $\mathbf{f}_{\mathbf{o}_i^j} \in \mathbb{R}^{d_j}$ by first computing the sum of all absolute weights in $\left|\mathbf{H}_{\mathbf{o}_i^j}\right|$.

$$\text{sum\_w}_{\mathbf{o}_i^j} = \sum_{b=1}^{d_j} |w_{jb}^j|$$

$$\mathbf{f}_{\mathbf{o}_i^j} = \frac{\left|\mathbf{H}_{\mathbf{o}_i^j}\right|}{\text{sum\_w}_{\mathbf{o}_i^j}}$$

Each attribute in $\{X_{j1}, X_{j2}, \ldots, X_{jd_j}\}$ corresponds to an outlying contribution score in $\mathbf{f}_{\mathbf{o}_i^j}$. The attributes having an outlying contribution score higher than a given threshold γ are determined as the outlying attributes of $\mathbf{o}_i^j$.

Algorithm 3.4 outlines the process by which $\mathcal{G}_j$ determines the outlying attributes of each outlier within the active window of data stream j. The algorithm initializes an empty list OutAttrList_j designated to store the outliers' outlying attributes (Line 1). It then iterates over each outlier (Lines 2-16). Within each iteration, the algorithm forms the inlier and outlier classes (Lines 3-4) and then uses data from those classes to determine a hyperplane that separates the inlier class from the outlier class (Lines 5-6). The absolute values of the attributes' weights on the hyperplane are then used to compute the outlying contribution scores of the outlier's attributes (Lines 7-8). The identification of the outlying attributes involves comparing each attribute's outlying contribution score with a predetermined threshold. If its outlying contribution score exceeds the threshold, the corresponding attribute is added to the list of outlying attributes of $\mathbf{o}_i^j$ ($F_{o_i^j}$) (Lines 9-15). Finally, the algorithm returns OutAttrList_j, which is a list of $|\mathbf{O}^j|$ outlying attribute lists, where $|\mathbf{O}^j|$ corresponds to the number of outliers within TW_j, the active window of data stream j.

Algorithm 3.4: **OutlyingAttrGeneratorAtS$_j$**($\mathbf{O_j}, \widehat{\mathbf{O}}_\mathbf{j}, \mathbb{C}_j, \gamma, \mathcal{D}_j$)

Inputs: $\mathbf{O^j}$ all outliers detected within the active tumbling window (TW_j) of data streams j (S_j), $\widehat{\mathbf{O}}^\mathbf{j}$ the estimated normal values of all outliers in TW_j, $\mathbb{C}_j$ the set of cluster of data points in TW_j, γ the threshold of outlying contribution score, and $\mathcal{D}_j$ the list of the attributes in S_j

Output: OutAttrListj the list of the list of outlying attributes of all outliers within TW_j

```
create an empty list OutAttrList_j
for each o_i^j in O^j and ô_i^j in Ô^j do
    inlier_class = generate_inlier_class(C_j, ô_i^j)
    outlier_class = generate_outlier _class(o_i^j, ô_i^j)
    classifier = LinearClassifier(requlariza tion=l_1)
    hyperplane = classifier.train(inlier_class, outlier_class)
    |H_{o_i^j}| = absolute(hyperplane.weights)

    sum_w_{o_i^j} = Σ_{b=1}^{d_j} |w_jb^i|
    f_{o_i^j} = |H_{o_i^j}| / sum_w_{o_i^j}
    create an empty list F_{o_i^j}
    for 1 ≤ b ≤ |D_j| do:
        if f_{o_i^j}[b] > γ do:
            F_{o_i^j}.append(D_j[b])
        end if
    end for
    OutAttrList_j.append(F_{o_i^j})
end for
return OutAttrList
```

3.4 Complexity Analysis

In this section, we discuss the time and space complexity of EXOS based on the following parameters:

- N_j, the number of data points in the active tumbling window of data stream j (TW_j), where $N_j = {tws}/{arrivalrate_j}$, tws is a constant representing the tumbling window size, and $arrivalrate_j$ is the data arrival rate at stream j. We also assume that N_j is the number the data points representing each data stream during the initialization phase.
- $|\mathbf{O^j}|$ the number of outliers in TW_j
- D, the total number of attributes of all data streams. $D = \sum_{j=1}^{m} d_j$ where d_j is the number of attributes in data stream j (S_j)
- k, the number of eigenvectors
- m, the number of data streams
- lj the number of clusters in S_j, the number of clusters in each stream can vary
- I, the maximum number of iterations required to initialize clusters of each stream
- $n_{samples}$, the maximum number of auxiliary data points generated for the inlier or outlier classes.
- nepochs, the number of iterations to train a linear SVM

3.4.1 Time Complexity of EXOS

We now discuss the time complexity of EXOS' initialization and Online Phases.

3.4.1.1 Time Complexity of The Initialization Phase

The initialization phase of EXOS involves two tasks: (1) initializing *k* eigenvectors **Q** (Algorithm 3.1, Line 1) and (2) initializing lj number of clusters in each data stream

(Algorithm 3.1, Line 2). Task 1 uses N_j combined data points of D attributes as input to the naive PCA algorithm [96]. Naive PCA requires $O(N_j D^2)$ time complexity to compute a covariance matrix of $D \times D$ size, $O(D^3)$ for eigen decomposition of the covariance matrix, and $O(D \log D)$ to sort k eigenvalues and select k eigenvectors to form **Q**. Overall the time complexity for Task 1 is $O(N_j D^2 + D^3 + D \log D)$.

Task 2 can be done in parallel since creating lj initial clusters for a data stream does not depend on another stream. The cluster initialization for S_j is done using the k-means clustering algorithm that requires $O(N_j lj I d_j)$ time complexity [104].

The overall time complexity of the Initialization Phase is $O(N_j D^2 + D^3 + D \log D) + O(Nj\ l_j\ I\ d_j) = O(N_j D^2 + D^3 + D \log D + Nj\ l_j\ I\ d_j)$. Assuming that the number of data points N_j is larger than the total number of attributes D, we can summarize the time complexity to $O(N_j D^2 + N_j lj I d_j)$.

3.4.1.2 Time Complexity of The Online Phase

The complexity of the EXOS' Online Phase is divided into three parts: (1) Estimator, (2) Temporal Neighbor Clustering, and (3) Outlying Attribute Generator.

3.4.1.2.1 Time Complexity of The Estimator

We compute the time complexity of the Estimator, *Est*, based on the active windows of all data streams. *Est* computes the number of timestamp steps needed to align data points from m streams (Algorithm 3.2, Line 1). This process has $O(m)$ time complexity and is followed by $O(mN_j)$ to align and select data points (Algorithm 3.2, Line 2). Notice that the number of selected aligned data points is equal to N_j when the arrival rates of all streams are equal, otherwise, it is less than N_j.

Est takes $O(1)$ to get the total number of data points in AlignedData (Algorithm 3.2, Line 3). It takes constant complexity because AlignedData is a key-value set or hashmap data structure and getting the total number of data points is the same as getting the size of AlignedData**.** Recall that in a typical hashmap implementation, the size (the number of key-value pairs) is tracked directly as an attribute of the hashmap. This means the size attribute requires no iteration or complex computation, just a simple read or lookup at the stored value.

Est initializes a zero matrix $\mathbf{Z}$ of size $D \times k$ which takes $O(Dk)$ time complexity (Algorithm 3.2 Line 4). It goes through every data point $x \in \mathbb{R}^D$ in the selected aligned data and updates $\mathbf{Z} = \mathbf{Z} + \frac{1}{N}\mathbf{x_{aligned}}\mathbf{x_{aligned}}^T\mathbf{Q}$. Multiplying $\mathbf{x_{aligned}}\mathbf{x_{aligned}}^T$result in a matrix $D \times D$ and the time complexity is $O(D^2)$. Multiplying the resulting matrix with $\mathbf{Q}$ takes $O(D^2 k)$ and adding the result to Z takes $O(D k)$. Overall updating $\mathbf{Z}$ for N_j times has $O(N_j (D^2 + D^2 k + D k))$ time complexity (Algorithm 3.2, Lines 5-8). After updating $\mathbf{Z}$, the next step is to update $\mathbf{Q}$ using QR Decomposition of $\mathbf{Z}$ (Algorithm 3.2, Line 9). The standard QR Decomposition uses the Householder Transformation algorithm [105]. It involves a series of reflections to transform each column in $\mathbf{Z}$ into an upper triangular form. The reflection involves vector and matrix multiplication operations that use $O(D^2)$. Since there are k columns in $\mathbf{Z}$, the overall time complexity for QR decomposition is $O(D^2 k)$.

Copying the updated Q takes $O(D k)$ and dividing the copied Q into m segments and storing them in a list of length m requires $O(m)$ operations. Hence the time complexity for the function slice_eigenvectors is $O(D k + m)$ (Algorithm 3.2, Line 10).

It takes $O(m)$ to initialize est_list (Algorithm 3.2, Line 11). In parallel, each slice of $\mathbf{Q}$, denoted as $\mathbf{Q}_j$, is used to estimate the outliers detected in Wj. The operation $\widehat{\mathbf{O}}^{\mathbf{j}} = \mathbf{O}^{\mathbf{j}}\mathbf{Q_j}\mathbf{Q_j^T}$

requires $O(d_j k)$ to transpose Q_j, $O(|\mathbf{O^j}| d_j k)$ for the $\mathbf{O^j Q_j}$ multiplication and $O(|\mathbf{O^j}| k d_j)$ to multiply the intermediate result with $\mathbf{Q_j^T}$. Overall the estimation process takes $O(d_j k + 2(|\mathbf{O^j}| d_j k))$ time complexity for each data stream (Algorithm 3.2, Line 12-17)

In summary, *Est* requires: $O(m) + O(m N_j) + O(1) + O(D k) + O(N_j (D^2 + D^2 k + D k)) + O(D^2 k) + O(D k + m) + O(m) + O(d_j k + 2(|\mathbf{O_j}| d_j k)) = O(1 + 3m + m N_j + 2D k + D^2 k + N_j D k + N_j D^2 + N_j D^2 k + d_j k + 2(|\mathbf{O_j}| d_j k))$ to process all its tasks for all data streams.

Since $N_j >> D$, $D > k$, $N_j >> |\mathbf{O^j}|$ and $D >> d_j$, the dominant term here is $O(N_j D^2 k)$, thus the time complexity for *Est* is $O(N_j D^2 k)$.

3.4.1.2.2 Time Complexity of Temporal Neighbor Clustering

The Temporal Neighbor Clustering component runs m $\mathcal{C}f_j$ functions in parallel. Each $\mathcal{C}f_j$ is associated with a data stream j. It takes $O(lj)$ to set lj empty clusters, $O(lj)$ to initialize num_points_{aj}, the number of data points of every cluster, and $O(lj\ d_j)$ to set $\mathbf{centroid_{aj}}$ of each cluster (Algorithm 3.3, Lines 1-3). For each data point in the active tumbling window of data stream j (TW_j), $\mathcal{C}f_j$ requires $O(lj\ d_j)$ to assign the data point into a cluster, $O(1)$ to update the assigned cluster's num_points_{aj}, and $O(d_j)$ to update its $\mathbf{centroid_{aj}}$. Considering there are N_j data points in the TW_j, the time complexity for updating the clusters is $O(N_j (lj\ d_j + d_j + 1))$ (Algorithm 3.3, Lines 4-7). In summary, the operations in $\mathcal{C}f_j$ have a time complexity of $O(lj) + O(lj) + O(lj\ d_j) + O(N_j (lj\ d_j + d_j + 1))$, where the dominant term is $O(N_j\ lj\ d_j)$, resulting in time complexity $O(N_j\ lj\ d_j)$ for $\mathcal{C}f_j$.

3.4.1.2.3 Time Complexity of Outlying Attribute Generator

The outlying attribute generator component executes m $\mathcal{G}_j$ functions simultaneously. Each $\mathcal{G}_j$ takes $O(1)$ to initialize an empty list OutAttrList_j**.** It identifies the outlying attributes of each $\mathbf{o_i^j}$ in $\mathbf{O^j}$ by first generating $n_{samples}$ auxiliary data points to form an inlier class for $\mathbf{o_i^j}$ based on the $\widehat{\mathbf{o}}_\mathbf{i}^\mathbf{j}$. The process of forming the inlier class involves $O(lj)$ time to find the closest cluster to $\widehat{\mathbf{o}}_\mathbf{i}^\mathbf{j}$, $O(d_j^2)$ time to generate a covariance matrix of size $d_j \times d_j$, and $O(n_{samples}d_j^2)$ time to generate $n_{samples}$ multivariate normal data [106]. Consequently, forming the inlier class requires a total of $O(lj) + O(d_j^2) + O(n_{samples}d_j^2)$. It takes the same time complexity to form the outlier class.

After forming the inlier and outlier classes, the next step is to determine the hyperplane that separates both classes using Linear SVM. Both inlier and outlier classes have $n_{samples}$ data points; thus, the total number of training data is $2n_{samples}$. The process of training a linear SVM using Stochastic Gradient Descent (SGD) [107] takes $O(\text{nepochs } 2n_{samples}\ d_j)$, where nepochs, $2n_{samples}$, and d_j represent the number of iterations in SGD, the total number of training samples for the linear SVM, and the number of attributes in data stream j, respectively. After the hyperplane is identified, $\mathcal{G}_j$ computes the absolute values of the hyperplane weights in $O(d_j)$ time. It then takes $O(d_j)$ to compute the outlying contribution scores of the attributes, $O(d_j)$ to check whether the outlying contribution score of each attribute exceeds the threshold for outlying attributes, and $O(d_j)$ to store the outlying attributes of o_i^j,

In total to generate the outlying attributes of an outlier $\mathbf{o_i^j}$, $\mathcal{G}_j$ needs $O(lj) + O(d_j^2) + O(n_{samples}d_j^2) + O(\text{nepochs } 2n_{samples}\ d_j) + O(d_j) + O(d_j) + O(d_j) + O(d_j) = O(lj + 4\ d_j + d_j^2$

+ $n_{samples}d_j^2$ + nepochs $2n_{samples}$ d_j). Considering the dominating term, the time complexity to generate the outlying attributes of an outlier $\mathbf{o_i^j}$ is O(nepochs $2n_{samples}$ d_j). Thus, for all outliers in TW_j, $\mathcal{G}_j$ requires $O(|\mathbf{O^j}|$ nepochs $2n_{samples}$ d_j) time complexity.

3.4.1.2.4 The Overall Time Complexity of The Online Phase

EXOS takes O(m N_j D) to absorb data points within the active tumbling windows of all data streams (Algorithm 3.1, Line 6). Additionally, it needs $O(|\mathbf{O^j}|d_j)$ time to get outliers from each stream in parallel (Algorithm 3.1, Line 7). As the estimator and temporal neighbor clustering components run in parallel (Algorithm 3.1, Lines 8-13), their time complexity is $O(\max(N_j D^2 k, N_j lj\ d_j))$. The outputs of these components serve as the inputs to the outlying attributes generator component which operates m parallel functions (Algorithm 3.1, Lines 14-19), each demanding $O(|\mathbf{O^j}|$ nepochs $2n_{samples}$ d_j).

The overall time complexity of the EXOS's Online Phase sums up to $O(m\ N_j\ D + |\mathbf{O^j}|\ d_j + \max(N_j\ D^2 k, N_j\ lj\ d_j) + |\mathbf{O^j}|$ nepochs $2n_{samples}$ d_j). Given that $N_j >> D$, $N_j >> |\mathbf{O^j}|$, $N_j >> lj$, $n_{samples} > d_j$, and $D > d_j$, we can simplify the overall time complexity of the Online Phase to $O(N_j D^2 k + |\mathbf{O^j}|$ nepochs $2n_{samples}$ $d_j)$.

3.4.2 Space Complexity of EXOS

We now discuss the memory space complexity of EXOS' initialization and Online Phases. When dealing with data streams, it is important to keep relevant data structures in the memory. Accessing data from memory is almost instantaneous, whereas I/O operations involve reading data from or writing data to disk, which is significantly slower. By performing operations in the memory, the algorithm can process data more quickly, which is crucial for real-time applications.

3.4.2.1 Space Complexity of the Initialization Phase

To get the initial eigenvector matrix **Q** (Algorithm 3.1, Line 1), EXOS applies the Naive PCA algorithm. This algorithm employs D x D matrix for correlation matrix derived from the initial data, where D represents the total number of attributes in all data streams. The algorithm then decomposes the correlation matrix using eigen decomposition into a list of eigenvalues of size D and a matrix of eigenvectors of size D x D. Thus, to construct the initial eigenvectors **Q,** EXOS requires $O(D^2)$ space to store the correlation matrix, $O(D^2)$ to store the eigenvectors, and O(D) to store the eigenvalues. Taking the maximum term, the space complexity required by Naive PCA is $O(D^2)$. Note that in Algorithm 3.1, Line 1, the function that calls Naive PCA only returns the first k eigenevectors. Thus **Q** is a D x k matrix that requires $O(D \times k)$ space.

EXOS stores a list of m sets of clusters of m data streams denoted as AllClust (Algorithm 3.1, Line 2). To initialize AllClust, EXOS runs the k-means clustering algorithm on each data stream j using N_j initial data points. Each data point has d_j dimension. The k-means algorithm groups data points into lj clusters. It needs $O(N_j\ d_j)$ space to store data points in the clusters, $O(N_j\ lj)$ space to store the distance between all data points to all clusters, $O(lj\ d_j)$ space to store centroids, Hence, the space complexity for k-means in all data streams is $O(m\ (N_j\ d_j + N_j\ lj + lj\ d_j))$. Recall that in Algorithm 3.1, Line 2, the function that employs the k-means algorithm on each data stream only return the centroids of the constructed clusters. Thus, the AllClust itself only uses $O(m\ lj\ d_j)$ space.

The total space complexity of the Initialization Phase is $O(D^2 + m\ (lj\ d_j + N_j + N_j\ d_j))$.

3.4.2.2 Space Complexity of the Online Phase

During the Online Phase (Algorithm 3.1 Lines 5-22), since there are m data streams, EXOS maintains m buffers or tumbling windows denoted as TWinS which store data within a period of time and replace the data when their period expires (Algorithm 3.1, Line 6). Each buffer denoted as TW_j corresponds to a data stream j and requires approximately $O(N_j d_j)$ space. Recall that d_j is the number of attributes in stream j and Nj is the number of data points in TWj which is calculated as follow $N_j = {}^{tws}/_{arrivalrate_j}$, where tws is a constant representing the tumbling window size and $arrivalrate_j$ is the data arrival rate at stream j.

Throughout the Online Phase, EXOS uses $O(m\ |\mathbf{O^j}|\ (d_j+1))$ space to store all outliers detected within m tumbling windows of all streams (Algorithm 3.1, Line 7). Recall that $|\mathbf{O^j}|$ is the number of outliers in all data streams, and each outlier in a stream j has d_j attributes and a timestamp associated with it. In addition to the space needed to store the outliers, $O(m|\mathbf{O_j}|(d_j))$ space is also needed to keep the corresponding estimated normal values of the outliers in the *est_list* data structure (Algorithm 3.1, Line 9). Furthermore, EXOS maintains $O(D\ k)$ space for D x k eigenvectors **Q** updated in the memory continuously (Algorithm 3.1, Line 9). EXOS maintains AllClust which stores $O(m\ lj\ d_j)$ or $O(L\ d_j)$ centroids of total L clusters in all data streams.

The space complexity of each key component of EXOS is further discussed in the following sections.

3.4.2.2.1 Space Complexity of Estimator

The space complexity analysis for *Est* involves considering various operations. *Est* takes $O(1)$ space for to keep the number of time steps (n_ts_step) needed to align data points from m tumbling windows (Algorithm 3.2, Line 1). It uses up to $O(m\ N_j\ D)$ space to store a key value set of the aligned data, AlignedData (Algorithm 3.2 Line 2), and $O(1)$ for variable $\overline{N}$ that represents the total number of aligned data points (Algorithm 3.2 Line 3).

Furthermore, EXOS utilizes $O(Dk)$ space for initializing D x k zeros matrix **Z** (Algorithm 3.2 Line 4). When iterating over each data point to update **Z**, it requires $O(D)$ space for $\mathbf{x_{aligned}} \in \mathbb{R}^D$, $O(D^2)$ space to keep the intermediate result of $\mathbf{x_{aligned}}\mathbf{x_{aligned}}^T$operation, and $O(D\ k)$ space for multiplying the intermediate result with the eigenvectors **Q** and adding the result to **Z** (Algorithm 3.2 Lines 5-8).

The Q-R decomposition can be implemented by modifying **Z** in place. The space complexity for Algorithm 3.2 Line 9 is $O(1)$ since it does not require an additional memory proportional to the input size. The process of slicing the eigenvectors **Q** (Algorithm 3.2 Line 10) requires $O(D\ k)$ space to duplicate **Q** and store it into m parts. The parallel procedure to estimate the normal values of outliers (Algorithm 3.2 Lines 12-17) requires approximately $O(m\ N_j\ (k+D))$ space.

Summing up all the space complexities of Est's operations, we get $O(1) + O(m\ N_j\ D) + O(1) + O(D\ k) + O(D^2 + D\ k) + O(1) + O(D\ k) + O(m\ N_j\ (k+D)) = O(3 + m\ N_j\ D + 3\ D\ k + D^2 + m\ N_j\ (k + D))$. Considering $N_j >> D$, the upper bound on the space required by Est is $O(m\ N_j\ (k+D))$ as it is the most significant term in the sum.

3.4.2.2.2 Space Complexity of Temporal Neighbor Clustering

We now discuss the space complexity analysis for the Temporal Neighbor Clustering Component based on the operation needed by its m independent functions $\mathcal{C}f_1, \mathcal{C}f_2, \ldots, \mathcal{C}f_m$. Each $\mathcal{C}f_j$ is associated with a data stream j that has d_j attributes and lj clusters. A $\mathcal{C}f_j$ takes $O(lj\ d_j)$ space to set lj initial centroids of all clusters (each $\mathbf{centroid_{aj}} \in \mathbb{R}^{d_j}$) and O(lj) space for all variables that keep the number of data points absorbed by the clusters (Algorithm 3.3, Line 1-2). Additionally, when iterating over each data point **x** to assign it to a cluster, $\mathcal{C}f_j$ requires $O(lj\ d_j)$ space when computing the distance of **x** to lj clusters and $O(d_j)$ space for updating the centroid of the closest cluster to **x** (Algorithm 3.3, Line 4-8).

In total, a $\mathcal{C}f_j$ uses $O(lj + 2\ lj\ d_j + d_j)$. Thus, the overall space required by the Temporal Neighbor Clustering component is $O(m(lj + 2\ lj\ d_j + d_j))$. Taking the most dominant term, the space complexity of the Temporal Neighbor Clustering component is $O(m(2\ lj\ d_j))$.

3.4.2.2.3 Space Complexity of Outlying Attribute Generator

To analyze the space complexity of the Outlying Attribute Generator component, we need to examine the space complexity of m independent functions $\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_m$. Each $\mathcal{G}_j$ corresponds to a data stream j.

Recall that in each active tumbling window associated with a strem j, TW_j, the total number of outliers detected is denoted as $|\mathbf{O}^j|$. In a $\mathcal{G}_j$, each outlier $\mathbf{o_i^j}$ has an inlier class and an outlier class. Each class has $n_{samples}$ auxiliary data points of $d_j + 1$ attributes (the original attributes of the outlier + the label of the data points). Thus the $\mathcal{G}_j$ function requires

$O(n_{samples}(d_j + 1))$ space to store the generated inlier class (Algorithm 3.4, Line 3) and $O(n_{samples}(d_j + 1))$ space for an outlier class (Algorithm 3.4, Line 3).

During the training of a linear classifier (Algorithm 3.4, Line 5) to find the decision boundary or hyperplance that separates the inlier and outlier classes, the classifier algorithm requires $O(2n_{samples}(d_j + 1))$ space to copy and store the entire data points from both the inlier and outlier classes, $O(d_j)$ to keep the weights associated with each attribute, and $O(1)$ space for the bias term. These are general space requirements for the linear classifier. Depending on the implementation of the classification algorithm, additional space might be needed for other parameters such as learning rate, etc [108].

The hyperplane generated by the linear classifier for each outlier $\mathbf{o_i^j}$ is kept by $\mathcal{G}_j$ in a vector of size d_j; thus, it takes $O(d_j)$ space for the hyperplane (Algorithm 3.4, Line 6). Additionally, $\mathcal{G}_j$ needs $O(d_j)$ space to keep the absolute values of attribute weights of the hyperplane (Algorithm 3.4, Line 7). It also uses $O(d_j)$ space to store the outlying contribution scores of the attributes in a vector $\mathbf{f}_{\mathbf{o_i^j}}$ of size d_j (Algorithm 3.4, Line 8). Finally, to store the outlying attributes of $\mathbf{o_i^j}$, $\mathcal{G}_j$ uses $O(d_j)$ space for a list named $F_{o_i^j}$ (Algorithm 3.4, Line 9-14).

In summary, the upper bound of $\mathcal{G}_j$'s space complexity for each outlier $\mathbf{o_i^j}$ is $O(2n_{samples}(d_j + 1))$ because this is the most dominant term. Thus, to generate outlying attributes of all $|\mathbf{O}^j|$ outliers detected in an active tumbling window of stream j, $\mathcal{G}_j$ requires $O(|\mathbf{O}^j|2n_{samples}(d_j + 1))$ space. We can conclude that the Outlying Attribute Generator's space complexity is $O(m|\mathbf{O}^j|2n_{samples}(d_j + 1))$.

3.5 Performance Evaluation

We conduct comprehensive experiments to evaluate the performance of EXOS. In this section, we first discuss our experimental setup and then provide our experimental results and analysis.

3.5.1 Experimental Setup

The experimental setup consists of (i) competing algorithms, (ii) datasets, (iii) performance metrics, and (iv) software and hardware.

3.5.2 Competing Algorithms

Chapter 2 shows that most of the existing techniques that discover outliers' outlying attributes are designed for static or non-stream data. The static data algorithms do not handle any research challenges related to data stream discussed in Chapter 1. They do not deal with the notion of time (Challenge DS 1), the notion of infinity (Challenge DS 2), varying data arrival rate especially for multiple data streams (Challenge DS 3), nor concept drift (Challenge DS 4) where the underlying distribution or statistical properties of the data can change over time [7, 8, 109].

There exist techniques proposed for data streams such as EXAD [38], EXstream [82, 83], MICOES [64], and MacroBase [83, 84]. These techniques handle the notion of time and the notion of infinity of the data streams. EXAD uses a decision tree estimated from a neural network trained offline to generate outlying attributes of each detected outlier online. The outlying attributes of an outlier are denoted as a disjunctive normal form of features formed by a path starting from the root to the leaf where the outlier resides.

Nevertheless, EXAD's decision tree is only good if the data distribution does not change. MICOES forms inlier and outlier classes to generate outlying attributes like COIN [51] that is intended for data streams, yet it utilizes incremental computation to generate outlying attributes. Its micro-cluster components adapt to changes in data distribution and thus allow the explanation to also adapt. MacroBase produces outlier explanations using a prefix tree of frequent itemsets updated periodically, allowing it to handle concept drifts. EXstream [82, 83] generates outlier explanations based on the temporal context of the outliers, which is updated over time and thus captures the change in the data distribution. These existing data stream-based outlying explanation algorithms: EXAD, EXstream, MICOES and MacroBase, operate independently for each data stream. Thus, they do not need to take care whether the data streams have the same arrival rate or not.

We compare EXOS to four existing algorithms: COIN [51], MICOES [64], MacroBase [83, 84], and EXstream [82, 83]. Like EXOS, all these algorithms are agnostic to the algorithm used for outlier detection. EXstream was originally proposed for COIN is one of the algorithms using inlier neighbors of an outlier as a local context to find the outlier's outlying attributes. It is designed for static data, yet we run it in batches to compare it with EXOS. The other three algorithms are proposed for data streams. EXstream is intended for interval or collective outliers, yet it can also be applied for point outliers. The source code of each algorithm is publicly available. We exclude EXAD since the published work does not provide a detailed explanation and implementation of the algorithm.

In Table 3-2 we present a summary of the properties of EXOS and the competing algorithms. Recall that in Chapter 2 Table 2-1, we have provided detail information about the competing algorithms. Here we repeat some information related to the data stream

research challenges discussed in Chapter 1. Notably, EXOS stands out as the sole algorithm tailored for data streams, simultaneously addressing the notion of time, the notion of infinity, concept drift, and while taking advantage of the cross-correlation of the data streams. Additionally, like the other four algorithms: COIN, MICOES, Macrobase, and EXstream, EXOS handle both research challenges related to outlying attributes: limiting subspace search (Challenge OutAtt 1) and generating readily interpretable output (Challenge OutAtt 2). EXOS applies a local neighborhood-based approach that does not require it to search all possible combination of attribute subspace to find outlying attributes of an outlier. It provides outlying contribution score for each outlier's attribute that can be easily understood. Attributes with zero or very small outlying contribution scores are not considered as the outlying attributes.

Table 3-1 Summary of the properties of the competing algorithms

Algorithm	Data Type	OutAtt 1	OutAtt2	DS 1	DS 2	DS 3	DS 4	Cross-Correlation
COIN [51]	Static Data	Yes	Yes	No	No	No	No	No
MICOES [64]	Data Stream	Yes	Yes	Yes	Yes	No	Yes	No
MacroBase [83, 84]	Data Stream	Yes	Yes	Yes	Yes	No	Yes	No
EXstream [82, 83]	Data Stream	Yes	Yes	Yes	Yes	No	Yes	No
EXOS (our algorithm)	Data Stream	Yes	Yes	Yes	Yes	Yes	Yes	Yes

OutAtt 1: limiting subspace search
OutAtt 2: providing readily interpretable output
DS 1: notion of time
DS 2: notion of infinity
DS 3: different data arrival rates
DS 4: concept drift

3.5.2.1 Datasets

We use three real-world datasets that reflect multi-source data streams: (1) Intel, (2) Microtremor, and (3) AMPds2. The Intel dataset [110] was gathered from 54 sensors deployed in the Intel Berkeley Research Lab between February 28th and April 5th, 2004. Each sensor sent time-stamped data with attributes: humidity, temperature, light, and voltages every 31 seconds. Since the sensors started sending measurements on different dates, we simulated data gathered between March 1st -31st, 2004 from Sensors 7, 9, 23, 25, 26, 29, 36, 38, and 44.

The Microtremor dataset [111] was collected from 33 seismic recording stations in Sacramento-San Joaquin Delta Region California. The ambient microtremor data were collected using temporary broadband seismometers over continuous time intervals ranging between 2 and 2.8 hours. After examining the metadata, we decide to use the data collected from 4 stations with site IDs, 2030, 2031, 2032, and 2033, as their data from these sites were collected on the same day while the data from the other sites were collected on different days.

The AMPds dataset [112] contains electricity, water, and natural gas measurements at one-minute intervals in a residential house in Canada. There are 21 power meters, 3 water meters, and 2 natural gas meters. Each meter sent a total of 1,051,200 readings for 2 years (April 2012 to March 2014). Power, water, and gas have 11, 2, and 3 attributes, respectively.

These datasets do not have the ground truth information for outliers and their outlying attributes. For performance evaluation purposes, we synthetically generate that

information. Specifically, we inject outliers into the dataset by randomly selecting around 1% of the original data points to be outliers. For each selected outlier, we randomly select several attributes from the dataset to be the outlying attributes and set the value of each of those attributes to be far away from its mean value. For example, suppose in the Intel dataset, data point #100 is chosen as an outlier and temperature as the outlying attribute. The temperature value of data point #100 is then set to either *max(temperature) + delta * standard deviation(temperature)* or *min(temperature) - delta * standard deviation(temperature)*, where delta > 3 based on the empirical rule which states, 99.7% of the observed data points lie within three standard deviations of the mean in normal distribution [101]. With a delta greater than three, we ensure that the generated abnormal values for a particular attribute are significantly deviated from the rest of the values of the attribute. The summary of the datasets used for the evaluation is provided in Table 3-2.

Table 3-2. Summary of the Datasets for Outlying Attributes

Datasets	**# Streams**	**Total number of data points**	**# Outliers injected**	**# Attributes per stream**
Microtremor	4	7 M	70 K (1%)	3
AMPds2	25	26M	260 K (1%)	2, 3, 11
Intel	9	401K	4.5K (1.12%)	4
Synthetic 1	5	50K	500 (1%)	10
Synthetic 2	4	40K	400 (1 %)	10

In addition, in Table 3-3, we list two synthetic datasets: Synthetic 1 and Synthetic 2. These datasets consist of five and four groups of data whose attributes are correlated. Each group represents a data stream and has 10 attributes. To ensure that the data streams are correlated, we first generate a correlation matrix using a technique described in [113]. This

technique allows us to specify whether the data attributes within and across data streams are highly or weakly correlated. The correlation matrix is then used to generate multivariate normal data. We inject outliers into the dataset in the same way as we do for the real datasets.

3.5.2.2 Performance metrics

We measure the effectiveness of the algorithms in finding outlying attributes using average precision, average recall, and average F1 score. For each outlier, we compute the numbers of True Positives (TP), False Positives (FP), False Negatives (FN), and True Negatives (TN) of its outlying attributes. We then use the information to compute the following metrics:

$$\text{Precision} = \frac{TP}{TP+FP}$$

$$\text{Recall} = \frac{TP}{TP+FN}$$

$$\text{F1 Score} = \frac{2*Recall*Precision}{Recall+Precision}$$

For example, suppose tuple #5 in the Intel dataset is an outlier with the ground truth outlying attributes = {light}. If EXOS decides its outlying attributes = {light, voltages}, the data point #5's confusion matrix is as follows:

d = 4	Predicted: No	Predicted: Yes
Ground Truth: No	TN = 2	TP = 1
Ground Truth: Yes	FN = 0	FP = 1

Given the confusion matrix, we can compute the outlying attribute's precision, recall, and F1 score of the outlier. Precision = $\frac{1}{1+1} = 0.5$, recall = $\frac{1}{2+0} = 0.5$, F1 score = $\frac{2*0.5*0.5}{0.5+0.5} =$

0.5 . We compute the precision, recall, and F1 score of each outlier and then aggregate the average of each metric.

In addition to measuring the accuracy of the algorithms, we also consider their efficiency. Given that we are dealing with a continuous arrival of data, we need to ensure that the algorithms can keep up with the incoming data promptly. Therefore, we also measure the average time the algorithms take to identify the outlying attributes of an outlier, which we refer to as the average explanation time, as part of our evaluation process.

3.5.2.3 Software and Hardware

We implement EXOS in Python 3 and simulate the multi-source data streams in parallel using its multiprocessing package. Our source code is available on a GitHub repository [114]. We use the Python implementations of MacroBase and EXstream that are provided for the Exathlon benchmark [83]. COIN's implementation is available in [115] while MICOES' implementation is available in [116]. We add some helper functions to the competitive algorithms so that they can run in parallel as we simulate multiple data streams. The experiments were conducted on a MacBook Pro: macOS Catalina, Processor 2.7 GHz Quad-Core Intel Core i7, Memory 16 GB 1600 MHz DDR3.

3.5.3 Performance Results and Analysis

In this section, we report the overall performance of the algorithms on each dataset and study the impact of window size, data arrival rate, and concept drift on the performance of each algorithm. Finally, we investigate how some parameters of EXOS affect its performance.

3.5.3.1 Overall Performance

We first evaluate the algorithms by simulating each dataset described in Section 3.5.2.1 as concurrent data streams where all data streams have the same arrival rate which is 0.1 second. Each algorithm has access to the tumbling windows. In each data stream, we set the window size to accommodate 1,440 data points for Intel and AMPds2 and 1,000 data points for Microtremor and Synthetic.

Table 3-3 shows that EXOS has the highest average precision for Intel and AMPds2. EXstream gets the best average precision for Microtremor and Synthetic 2 while COIN wins on Synthetic 1. When it comes to average recall, MacroBase wins for AMPds2, Synthetic 1 and 2, while COIN dominates for Intel. For Microtremor, the four algorithms achieve an average recall of 0.99, while EXstream 0.87. Obviously, there is a tradeoff between precision and recall among these algorithms. Precision measures the extent of error by False Positives, while recall deals with the error caused by False Negatives. F1 score balances those scores. EXOS achieves the best average F1 score in all the datasets except for AMPds2. The average F1 score of EXOS are 35%, 82%, 7%, 28%, and 69% better than those of COIN, 39%, 78%, 21%, 52%, and 78% better than those of MICOES, 44%, 75%, -10%, 86%, and 82% better than those of MacroBase, and 29%, 2%, 12%, 46%, and 57% better than those of EXstream for the Intel, Microtremor, AMPds2, Synthetic 1, and Synthetic 2 datasets, respectively. Compared to COIN, MICOES, Macrobase, and EXstream which operate independently on each data stream, EXOS stands out as the sole algorithm that incorporates the cross-correlation among all data streams, leveraging it to estimate the normal values of outliers. This unique approach elucidates why EXOS achieves a balanced average precision and recall, distinguishing it from other

algorithms that may demonstrate high average precision but lower average recalls. Additionally, although COIN and MICOES similarly utilize the weights of a linear classifier's hyperplane to determine outlying attributes of an outlier, their average F1 score is comparatively lower than that of EXOS. This emphasizes the effectiveness of EXOS's strategy in considering the cross-correlation among data streams.

Table 3-3 Algorithm Performance Evaluation

Datasets	Algorithms	Avg precision	Avg recall	Avg F1 Score	Avg Explanation Time (second)
Intel	COIN	0.49	**0.99**	0.63	0.042
	MICOES	0.5	0.91	0.61	0.020
	MacroBase	0.49	0.92	0.59	0.016
	EXstream	0.85	0.59	0.66	0.011
	EXOS	**0.95**	0.84	**0.85**	**0.010**
Microtremor	COIN	0.33	**0.99**	0.49	0.045
	MICOES	0.33	**0.99**	0.50	0.089
	MacroBase	0.34	**0.99**	0.51	0.007
	EXstream	**0.87**	0.87	0.87	**0.003**
	EXOS	0.83	**0.99**	**0.89**	**0.003**
AMPds2	COIN	0.43	0.86	0.54	0.045
	MICOES	0.39	0.80	0.48	0.020
	MacroBase	0.56	**0.89**	**0.64**	0.097
	EXstream	0.91	0.42	0.52	**0.007**
	EXOS	**0.94**	0.48	0.58	**0.007**
Synthetic 1	COIN	**1.00**	0.52	0.64	0.072
	MICOES	0.47	0.79	0.54	0.055
	MacroBase	0.3	**1.00**	0.44	0.599
	EXstream	0.98	0.45	0.56	0.050
	EXOS	0.84	0.87	**0.82**	**0.047**
Synthetic 2	COIN	0.42	0.86	0.54	0.076
	MICOES	0.51	0.58	0.51	0.234
	MacroBase	0.34	**1.00**	0.50	0.629
	EXstream	**0.99**	0.44	0.58	0.093
	EXOS	0.88	0.97	**0.91**	**0.058**

Table 3-3 shows that for all the datasets, EXOS has the fastest average execution time for generating outlying attributes for outliers. EXOS is 4.2, 15, 6.4, 1.5, and 1.3 times faster than COIN, 2, 29.6, 2.8, 1.1, and 4 times faster than MICOES, and 1.6, 2.3, 13.8, 12.7, and 10.8 times faster than MacroBase for the Intel, Microtremor, AMPds2, Synthetic 1, and Synthetic 2 datasets, respectively. Even though EXOS and EXstream have similar average execution times, EXOS outperforms EXstream in the average F1 Scores for all the datasets. These results underscore that while considering data streams cross-correlation introduces added complexity to EXOS, executing the Estimator component along *m* functions in the Temporal Neighbor Clustering component in parallel helps speed up the average outlier explanation time on each window.

We also delve deeper into how the means of the average F1 score and explanation time differ among the algorithms. Employing a One-way ANOVA [117] followed by a pairwise Tukey's Honest Significant Difference (HDS) test [118], we aim to discern statistical differences in the performance means of the algorithms. The null hypothesis of the One-way ANOVA posits no significant difference between the means of the overall algorithms' performance.

We ran experiments ten times on each dataset and recorded the results. The One-Way ANOVA test on the average F1 score yields a remarkably small p-value (less than 0.05), providing grounds to reject the null hypothesis. With the understanding that there are statistically significant differences between the means of the algorithms' average F1 Scores, we proceed to Tukey's HDS to pinpoint the significant differences between each pair of the algorithm means.

The Tukey's HDS results are presented in Figure 3-10, where the mean difference (meandiff) is calculated by subtracting the mean of group 1 from group 2 (group 2 - group 1). The p-adj column signifies the adjusted p-value, indicating the probability of observing a result as equal to, or more extreme than, the one obtained if the null hypothesis is true. A smaller p-adj suggests stronger evidence against the null hypothesis. In this context, the null hypothesis is 'there is no significant difference between the performance means of a pair of algorithms.' The lower and upper columns denote the confidence intervals for the differences between the performance means of the compared algorithms. If the confidence interval excludes zero, it lends support to the rejection of the null hypothesis. The reject column denotes whether the null hypothesis is rejected (True) or not (False).

In Figure 3-10 (a), the mean difference of the average F1 score between COIN (group 1) and EXOS (group 2) is 0.242, indicating that EXOS's mean is higher than COIN's. The small, adjusted p-value (-0.0) and the mean difference confidence interval excluding zero lead to the rejection of the null hypothesis, signifying a significant difference in the mean of COIN and EXOS average F1 Scores. Conversely, there are three comparisons where there is insufficient evidence to reject the null hypothesis: COIN vs Macrobase, COIN vs MICOES, and Macrobase vs MICOES. This implies that COIN, Macrobase, and MICOES exhibit similar average F1 Scores. Notably, when comparing EXOS (group 1) to EXstream, Macrobase, or MICOES (group 2), the negative meandiff value indicates that EXOS's average F1 score means are greater than those of the other three algorithms.

The null hypothesis for Figure 3-10 (b) is "there is no significant mean difference in the average explanation time of a pair of algorithms." Four comparisons COIN vs Macrobase, EXOS vs Macrobase, EXstream vs Macrobase, and Macrobase vs MICOES

plausibly reject the null hypotheses. The outcomes confirm that only Macrobase has a higher average explanation time, while the other algorithms demonstrate comparable average explanation time.

(a) Tukey's HDS on the algorithms average F1 Score

group1	group2	meandiff	p-adj	lower	upper	reject
coin	exos	0.242	-0.0	0.1923	0.2917	True
coin	exstream	0.07	0.0013	0.0203	0.1197	True
coin	macrobase	-0.032	0.3945	-0.0817	0.0177	False
coin	micoes	-0.04	0.1792	-0.0897	0.0097	False
exos	exstream	-0.172	-0.0	-0.2217	-0.1223	True
exos	macrobase	-0.274	-0.0	-0.3237	-0.2243	True
exos	micoes	-0.282	-0.0	-0.3317	-0.2323	True
exstream	macrobase	-0.102	0.0	-0.1517	-0.0523	True
exstream	micoes	-0.11	0.0	-0.1597	-0.0603	True
macrobase	micoes	-0.008	0.992	-0.0577	0.0417	False

(b) Tukey's HDS on the algorithms average explanation time

group1	group2	meandiff	p-adj	lower	upper	reject
coin	exos	-0.031	0.7774	-0.1048	0.0428	False
coin	exstream	-0.0232	0.9098	-0.097	0.0506	False
coin	macrobase	0.2136	0.0	0.1398	0.2874	True
coin	micoes	0.0276	0.8425	-0.0462	0.1014	False
exos	exstream	0.0078	0.9984	-0.066	0.0816	False
exos	macrobase	0.2446	-0.0	0.1708	0.3184	True
exos	micoes	0.0586	0.1902	-0.0152	0.1324	False
exstream	macrobase	0.2368	-0.0	0.163	0.3106	True
exstream	micoes	0.0508	0.3248	-0.023	0.1246	False
macrobase	micoes	-0.186	0.0	-0.2598	-0.1122	True

Figure 3-10 The Tukey's HDS test results on the algorithms' overall performance

COIN, EXstream, Macrobase, and MICOES algorithms explain outliers in each data stream independently, without the necessity to process data points from all streams collectively. Consequently, they can be applied directly to *m* data streams by running *m* instances of the algorithms. On the contrary, EXOS utilizes cross-correlation between data

streams to estimate normal values of outliers, necessitating data points from all streams to be processed together in its Estimator component. To showcase the performance of these algorithms on non-concurrent data streams, we run the Synthetic 2 dataset as non-concurrent data streams. We set the window size to ten minutes and simulated each data point in S1, S2, S3, and S4 to arrive every 1, 2, 4, and 3 seconds, respectively. The performance results are reported in Table 3-4.

Table 3-4. The algorithms' performance on Synthetic 2 simulated as non-concurrent data streams

Algorithm	Avg precision	Avg recall	Avg F1 Score	Avg Explanation Time (second)
COIN	0.350	0.683	0.435	0.411
MICOES	0.354	0.727	0.443	0.332
Macrobase	0.258	**1.000**	0.391	1.994
EXstream	**0.988**	0.517	0.643	**0.284**
EXOS	**0.933**	**0.987**	**0.947**	0.331

In Table 3-4, it is evident that EXOS stands out as the sole algorithm with the average precision, average recall, and average F1 score surpassing 0.9. While EXstream attains the highest average precision, it concurrently records the lowest average recall. Conversely, Macrobase achieves the perfect average recall but at the expense of the lowest average precision. In this particular scenario, the algorithms are ranked from the highest to the lowest average F1 Score as follows: EXOS, EXstream, MICOES, COIN, and Macrobase. In terms of average explanation time in this scenario, the ranking from the fastest to the slowest is as follows: EXstream, EXOS, MICOES, COIN, and Macrobase.

3.5.3.2 The Impact of the Window Size and Data Points' Arrival Rate

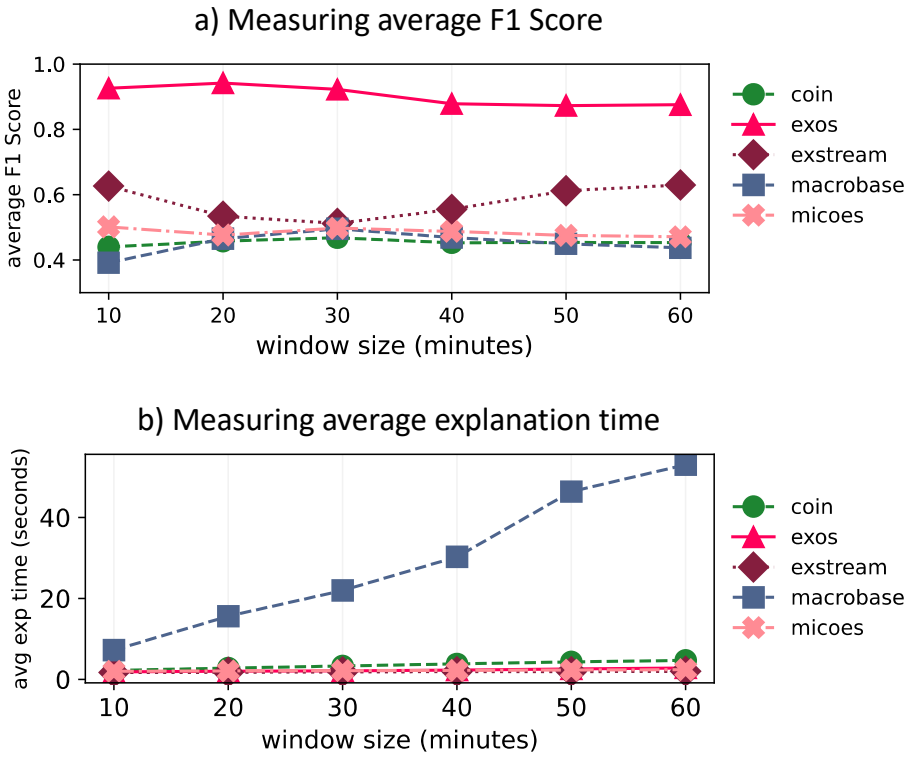


Figure 3-11. The impact of the window size on the performance of the algorithms

The number of data points within a window is contingent on both the window size and the arrival rate of data points. Using the Synthetic 1 dataset, which represents four data streams, we explore how the window size affects the algorithm performance. We vary the window size from ten to sixty minutes, simulating each data point to arrive every five seconds. Figure 3-11 describes the average F1 Scores of the algorithms. Notably, EXOS consistently achieves the highest score irrespective of the window size. However, with an increase in the window size, its F1 Scores show a slight decrease. This suggests that, in EXOS, a higher proportion of inliers compared to outliers does not necessarily lead to a more accurate explanation of outlying attributes. Among all algorithms, Macrobase has the

highest average explanation time. The explanation time for Macrobase increases as the window size expands. In contrast, the average explanation time for the remaining algorithms remains relatively low.

(a) Tukey's HDS on the algorithms average F1 Score

group1	group2	meandiff	p-adj	lower	upper	reject
coin	exos	0.4487	0.0	0.395	0.5024	True
coin	exstream	0.124	0.0	0.0703	0.1777	True
coin	macrobase	-0.0029	0.9999	-0.0566	0.0508	False
coin	micoes	0.0307	0.4652	-0.023	0.0844	False
exos	exstream	-0.3247	0.0	-0.3784	-0.271	True
exos	macrobase	-0.4516	0.0	-0.5053	-0.3979	True
exos	micoes	-0.4181	0.0	-0.4718	-0.3643	True
exstream	macrobase	-0.1269	0.0	-0.1806	-0.0732	True
exstream	micoes	-0.0933	0.0003	-0.147	-0.0396	True
macrobase	micoes	0.0335	0.3774	-0.0202	0.0872	False

(b) Tukey's HDS on the algorithms average explanation time

group1	group2	meandiff	p-adj	lower	upper	reject
coin	exos	-1.2475	0.9987	-14.7715	12.2765	False
coin	exstream	-1.675	0.996	-15.199	11.849	False
coin	macrobase	25.5602	0.0001	12.0363	39.0842	True
coin	micoes	-1.3323	0.9984	-14.8563	12.1917	False
exos	exstream	-0.4275	1.0	-13.9515	13.0965	False
exos	macrobase	26.8078	0.0	13.2838	40.3317	True
exos	micoes	-0.0848	1.0	-13.6088	13.4392	False
exstream	macrobase	27.2352	0.0	13.7113	40.7592	True
exstream	micoes	0.3427	1.0	-13.1813	13.8667	False
macrobase	micoes	-26.8925	0.0	-40.4165	-13.3686	True

Figure 3-12. The Tukey's HDS test result on the algorithms' performance when window sizes vary

We also explore whether the performance of the algorithms significantly differs with varying window sizes. Conducting the One-Way ANOVA test on the average F1 score produces a notably small p-value (less than 0.05), prompting the rejection of the null

hypothesis which suggests no difference among the means of the algorithms' average F1 Scores. Recognizing statistically significant variations between the means of the algorithms' average F1 Scores, we employ Tukey's HDS to identify significant mean differences between each pair of the algorithms.

In Figure 3-12 (a) the difference in the average F1 Scores between COIN (group 1) and EXOS (group 2) is 0.4487, indicating that EXOS's mean exceeds COIN's. Recall that the null hypothesis for the Tukey's HDS test is "there are no statistically significant differences between the means of a pair of algorithms' average F1 score". With the adjusted p-value of 0.0 and the confidence interval excluding 0 in its range, it is plausible to reject the null hypothesis. However, there is insufficient evidence to reject the null hypothesis for three comparisons: COIN vs Macrobase, COIN vs MICOES, and Macrobase vs MICOES. These results suggest similar average F1 Scores among COIN, Macrobase, and MICOES despite varying window sizes. Notably, when comparing EXOS (group 1) with EXstream, Macrobase, and MICOES (group 2), the negative mean difference values indicate that EXOS's average F1 score surpasses those of these three algorithms.

In Figure 3-12 (b) only four comparisons support rejecting the null hypothesis. These comparisons are COIN vs Macrobase, EXOS vs Macrobase, EXstream vs Macrobase, and Macrobase vs MICOES. This outcome confirms that only Macrobase has a higher average explanation time, while the other algorithms have similar time performance when the arrival rate is five seconds, and the window sizes vary between ten to sixty minutes.

Utilizing the same dataset, we investigate the influence of the data point's arrival rate by altering it between one and ten seconds, with a fixed window size of thirty minutes. The outcomes are depicted in Figure 3-13 and Figure 3-14. The arrival rate signifies the time

interval to await a new data point after the preceding one has arrived. For instance, an arrival rate of 10 seconds implies a data point arrives every ten seconds. With an increasing arrival rate, there is a reduction in the number of data points within the fixed-size window.

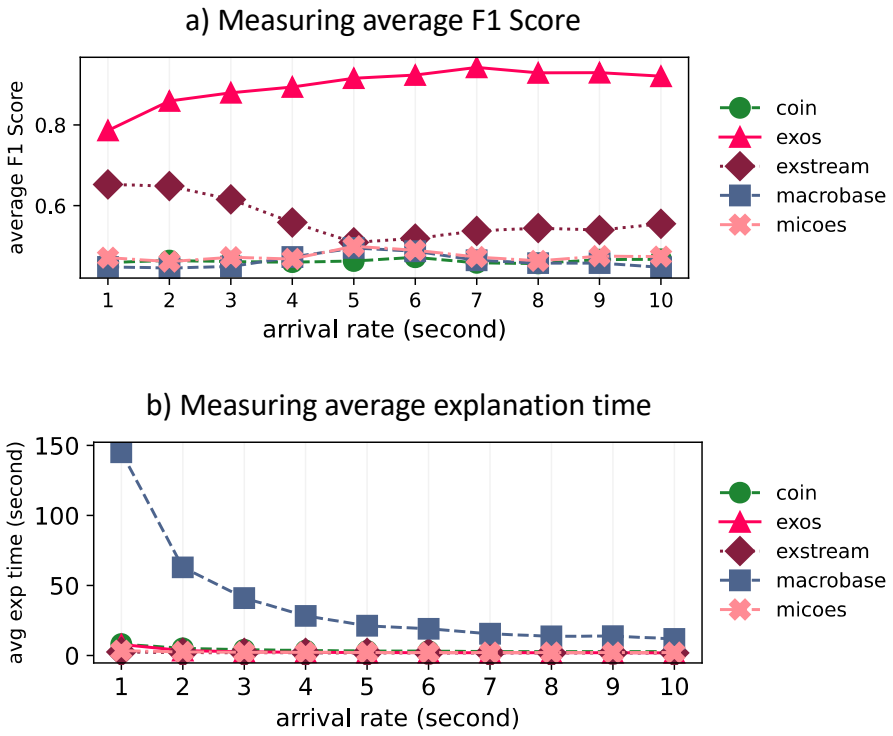


Figure 3-13. The impact of the arrival rate of data points on the performance of the algorithms

Figure 3-13 illustrates that EXOS attains the highest average F1 score across all variations of the arrival rate. This finding supports the notion that, for achieving a superior average F1 score, EXOS does not necessitate an increase in data points but rather requires a better understanding of the local context for each outlier. More data points do not inherently translate to an improved local context. Notably, Macrobase demonstrates the lengthiest average explanation time in all arrival rate variations. However, as it takes longer

for a new data point to arrive, the average explanation time is going down. This can be attributed to the decrease in the number of data points within the window as it takes longer for a data point to arrive, resulting in a shorter average explanation time.

(a) Tukey's HDS on the algorithms average F1 Score

group1	group2	meandiff	p-adj	lower	upper	reject
coin	exos	0.4349	-0.0	0.3933	0.4764	True
coin	exstream	0.1051	0.0	0.0635	0.1466	True
coin	macrobase	-0.0002	1.0	-0.0418	0.0414	False
coin	micoes	0.0119	0.9246	-0.0296	0.0535	False
exos	exstream	-0.3298	-0.0	-0.3714	-0.2882	True
exos	macrobase	-0.4351	-0.0	-0.4766	-0.3935	True
exos	micoes	-0.4229	-0.0	-0.4645	-0.3814	True
exstream	macrobase	-0.1053	0.0	-0.1468	-0.0637	True
exstream	micoes	-0.0931	0.0	-0.1347	-0.0516	True
macrobase	micoes	0.0121	0.9202	-0.0294	0.0537	False

(b) Tukey's HDS on the algorithms average explanation time

group1	group2	meandiff	p-adj	lower	upper	reject
coin	exos	-1.0568	0.9999	-24.4646	22.351	False
coin	exstream	-1.8923	0.9994	-25.3001	21.5155	False
coin	macrobase	33.4102	0.0018	10.0024	56.818	True
coin	micoes	-1.5273	0.9997	-24.9351	21.8805	False
exos	exstream	-0.8355	1.0	-24.2433	22.5723	False
exos	macrobase	34.467	0.0012	11.0592	57.8748	True
exos	micoes	-0.4705	1.0	-23.8783	22.9373	False
exstream	macrobase	35.3025	0.0009	11.8947	58.7103	True
exstream	micoes	0.365	1.0	-23.0428	23.7728	False
macrobase	micoes	-34.9375	0.001	-58.3453	-11.5297	True

Figure 3-14. The Tukey's HDS test results on the algorithms' performance when arrival rates vary.

In Figure 3-14 (a), only three pairwise comparisons COIN vs. Macrobase, COIN vs. MICOES, and COIN vs. Macrobase vs. MICOES, suggest no significant mean difference in the average F1 score between the algorithms. This aligns with the observations from

Figure 3-13 (a) showing COIN, Macrobase, and MICOES exhibit comparable average F1 Scores. Figure 3-14 (b) further validates that Macrobase exhibits significantly higher mean differences in average explanation time compared to other algorithms. Additionally, it reveals no significant mean differences in the average explanation time of COIN, EXOS, EXstream, and Macrobase.

3.5.3.3 The Impact of Concept Drift

We investigate the impact of data distribution changes, known as concept drift, on the performance of the algorithms. We simulate four data streams, each having ten attributes, eleven windows, 1K data points per window, and the same arrival rate. Our experiments focus on two key questions: (Q1) Does the performance of the algorithms alter when a concept drift occurs in a specific window?, and (Q2) How does the frequency of concept drift occurrences affect the algorithms' performance?

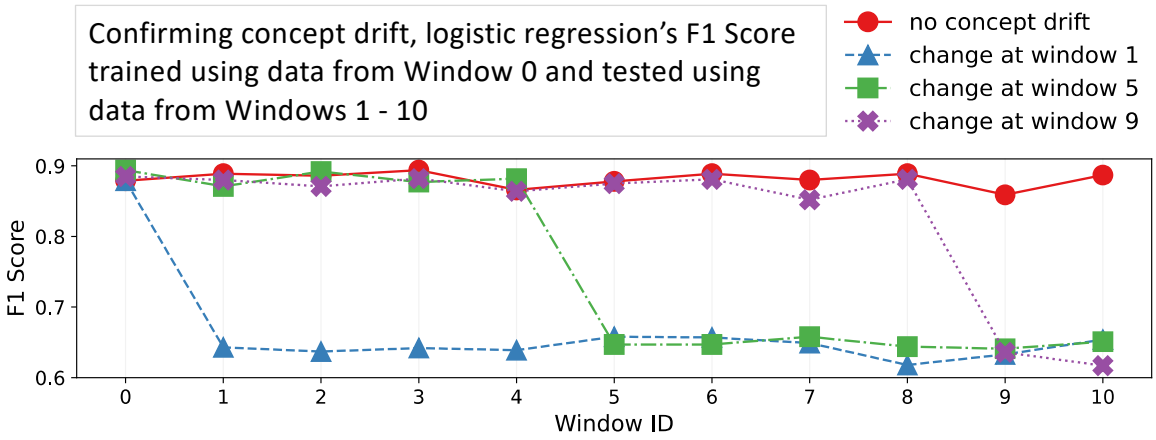


Figure 3-15. Using a static data-based classifier to confirm whether a dataset indeed has concept drifts

To address Question (Q1), we simulate the synthetic datasets with varying locations of concept drift occurrences, i.e., varying the IDs of the windows where concept drifts occur. Following the similar approaches used to verify that concept drifts indeed occur in a dataset [109, 119], we confirm the presence of concept drift by adding binary classification labels to the datasets and then training a logistic regression classifier using the data in Window 0 before testing it using the data in Windows 1 through 10. It is worth noting that this classification model is designed for static data and thus is expected to perform similarly across all the windows in the absence of concept drifts. However, as shown in Figure 3-15, once there is a change in the data distribution in Window *i*, the classifier performance begins to decline, indicating the presence of concept drifts in the datasets.

Figure 3-16 demonstrates the performance of the outlying attribute algorithms when the location of a concept drift occurrence is varied from Window 0 to Window 10. EXOS achieves the highest average F1 Scores, followed by EXstream, MacroBase, MICOES, and COIN. However, MacroBase has the highest average explanation time, while other algorithms perform similarly. The consistent performance is maintained by all the algorithms regardless of the concept drift locations as their model and/or temporal context are updated in each window. Thus, we can conclude that the performance of the algorithms is unaffected by the locations of concept drifts.

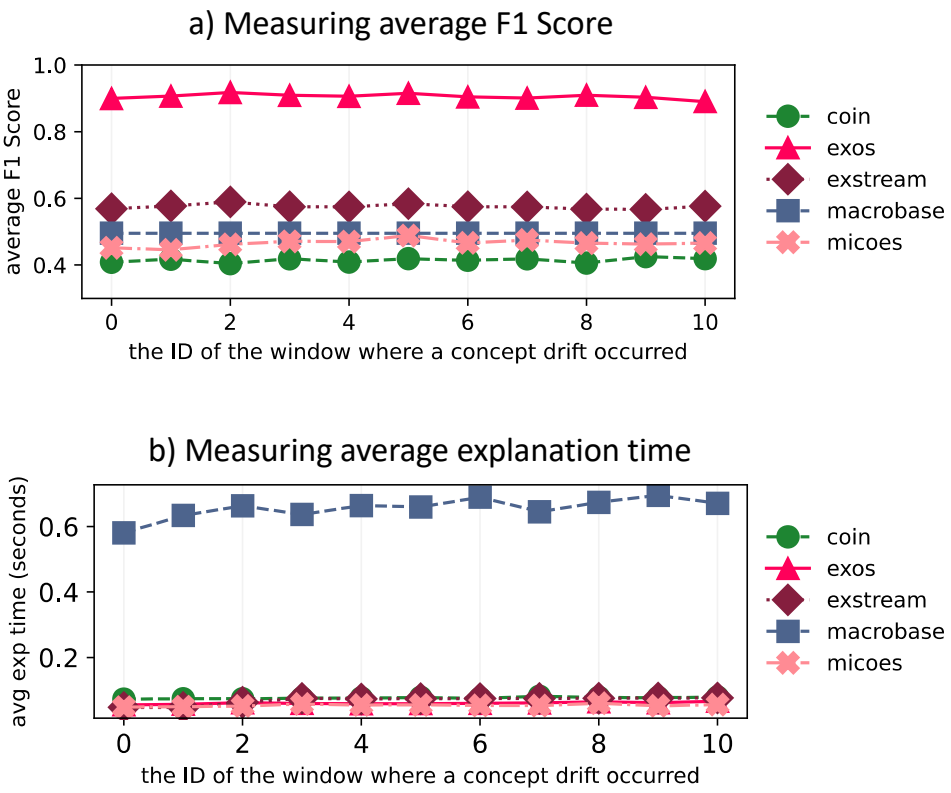


Figure 3-16. The impact of the location of concept drift on the algorithms' average F1 Score and explanation time

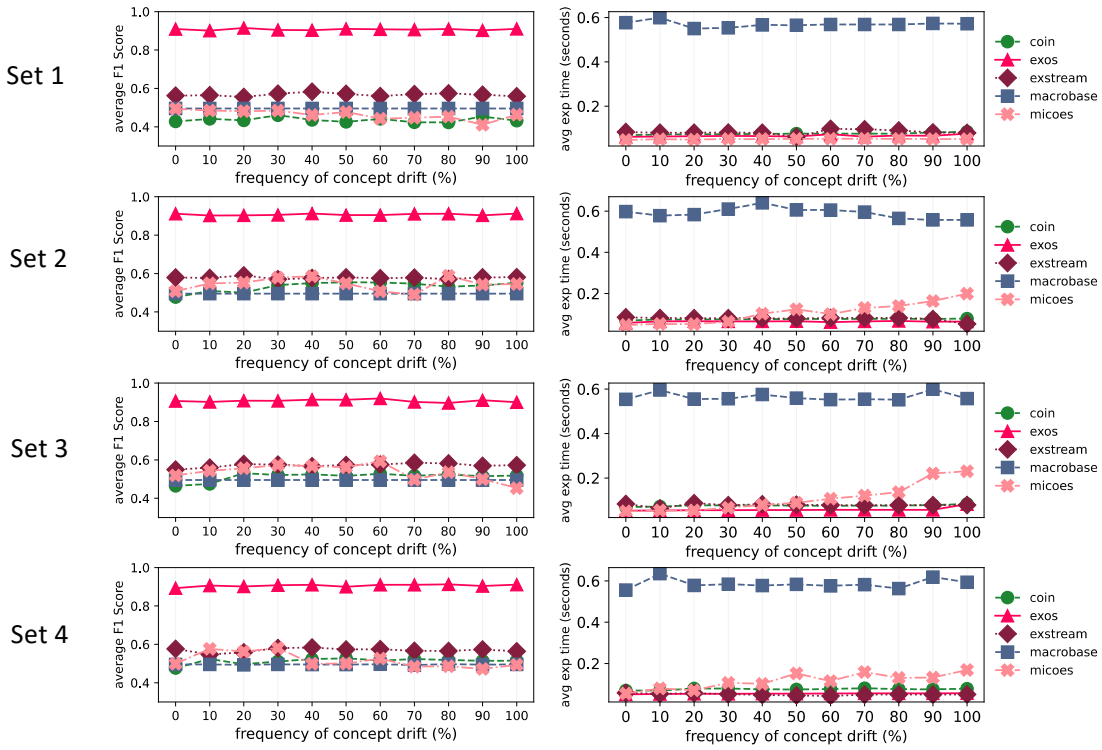


Figure 3-17. The Impact of the frequency of concept drifts on average F1 score and explanation time

To answer Question (Q2), we vary the occurrences of concept drifts across eleven windows ranging from 0% to 100%. A 0% concept drift indicates that Windows 1-10 have the same data distribution as Window 0. A 50% concept drift means from Window 1 through Window 10, there are five windows where the data distribution changes. A 100% concept drift means the data distribution changes every time the window slides. We run experiments on four synthetic datasets, Sets 1-4. Sets 1, 2, and 3 are composed of multivariate Gaussian data. In Set 1, a change in the mean vectors occurs during specific windows while the covariance matrices remain unchanged. In contrast, in Set 2, only the covariance matrices change while the mean vectors stay constant. Finally, both the mean vectors and covariance matrices change simultaneously in Set 3.

Set 4 has mixed data distributions consisting of randomly chosen multivariate Gaussian, Beta, Gamma, and Exponential distributions. For instance, when we set the percentage of concept drifts to be 50%, Windows 0-2 can have a Gaussian distribution, while Window 3 may use a gamma distribution. Likewise, Windows 4-7 can be exponential, Window 8 beta and finally, Windows 9-10 are gamma distributed.

Figure 3-17 displays the performance of the algorithms on Sets 1-4, with EXOS having the best average F1 score. All the algorithms have similar average explanation times while MacroBase has the highest one. The frequency of concept drifts hardly has any effect on the performance of EXOS, EXstream, Coin, and MacroBase. With EXOS, the eigenvectors used to estimate the normal values of the outliers in a window are updated when the window slides. The same update applies to its temporal context component, which, together with the estimated normal values of the outliers, forms the inlier and outlier classes. Therefore, even when a window data distribution differs from that in the previous window,

the EXOS components are constantly adjusted, allowing it to maintain its performance. A similar window-sliding update on the explanation model also applies to the other algorithms. Even though COIN is intended for static data, we run it in batches such that the subsets of data points used to build its outlier explanation model are continually updated as the window slides. The average F1 score slightly fluctuates on MICOES, indicating that the temporal context formed using denstream-based micro-clusters in a window still mixes with some micro-clusters from the previous window. MICOES's average explanation time also slightly increases in Sets 2-4 when the frequency of concept drifts is getting higher. This explanation time increase relates to the maintenance of its micro-clusters. When the data distribution changes more often, the state of its micro-clusters also changes, requiring more time to maintain/update.

3.5.3.4 The Impact of EXOS-Specific Parameters

We study the impact of two EXOS user-defined parameters using the synthetic dataset where the window size and arrival rate of each stream are set to be the same such that each window contains 1000 data points.

Impact of the number of eigenvectors (k). To generate the estimated normal value of each outlier found in each window, EXOS needs the estimated eigenvectors (principal components) whose size depends on *k*. *k* can be varied from 1 to D (the total number of attributes). In our study, we vary *k* from 1 to 10. Figure 3-18 tells us that *k* has an impact on the average precision, recall, and F1 score. As *k* increases, the average recall also increases on average by 1.89% but the average precision and F1 score decrease on average by 3.73% and 0.84% respectively. Unsurprisingly, as *k* gets larger, the average explanation time gets longer on average by 3.6% as the EXOS' estimation component uses more

eigenvectors (i.e., the size of the D x k eigenvectors matrix **Q** is getting larger). Therefore, to provide the explanations quickly, a small value of *k* such as *k*=1 would be preferred.

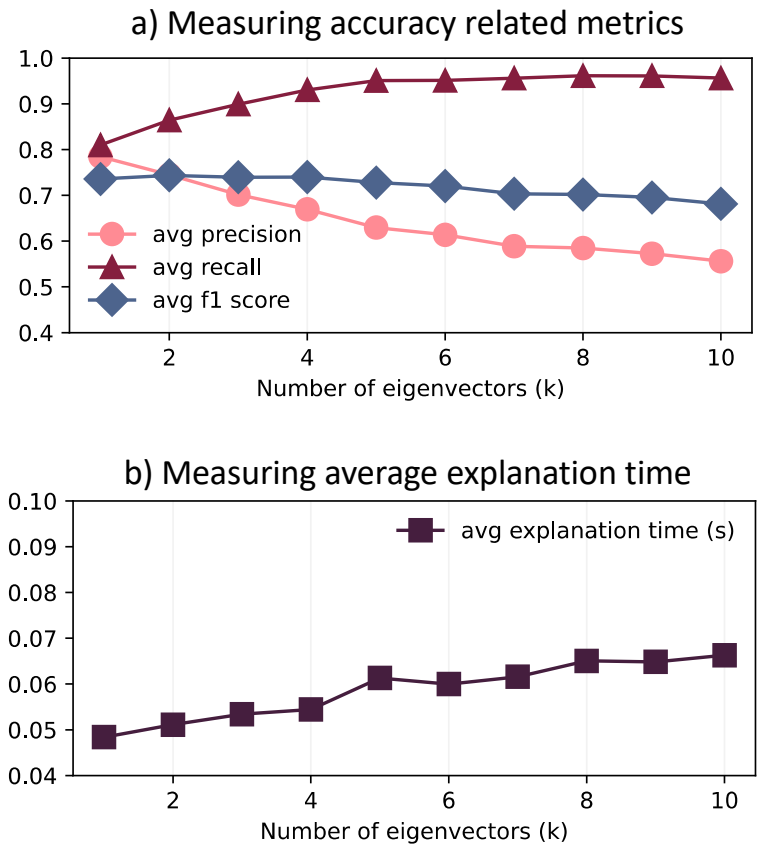


Figure 3-18. Impact of the number of eigenvectors (k) on performance of EXOS

Impact of the outlying contribution score threshold (γ). The threshold γ guides EXOS in identifying attributes considered as the outlying attributes of an outlier after the generating decision boundary that separates the inlier class from the outlier class for that outlier. γ = 0.01 indicates that only attributes that have the outlying contribution scores greater than 1% are considered the outlying attributes. Figure 3-19 demonstrates the average precision slightly increases on average by 1% as γ increases by 0.01. A higher γ facilitates the reduction of False Positive outlying attributes. Conversely, the average recall

experiences a slight decline on average by 1% with increasing γ, as higher γ values contribute to more False Negative outlying attributes. Nevertheless, the average F1 score remains steady as the reduction of False Positive is counterbalanced by the increase in False Negative. Additionally, as the γ value increases by 0.01, the average explanation time is slightly increased by 2%. Therefore, a small value of γ such as γ=0.0 would be preferred.

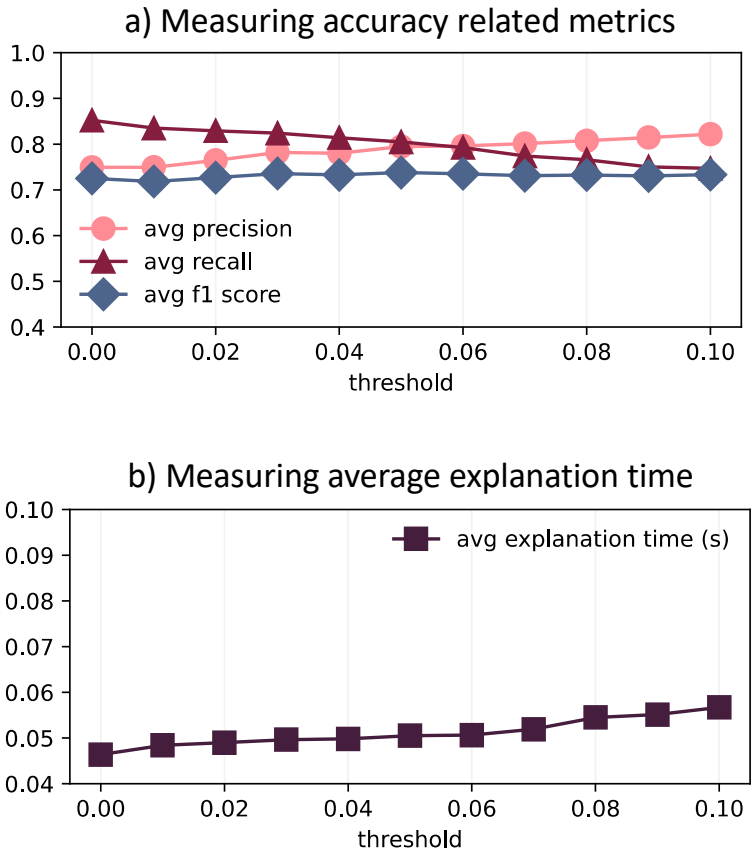


Figure 3-19. Impact of the outlier attribute contribution threshold on the performance of EXOS

CHAPTER 4

Discovering Root Cause Factors of Outliers in Data Streams: Ocular

Outlier explanation is a growing research area [14, 120]. In Chapter 3, we present EXOS a technique that generate outlier explanations in the form of outlying attributes for each individual outlier. Knowing the outlying attributes helps speeding up users' investigation on the outlier. However, it is important to note that outlying attributes may not always provide a complete causality explanation or uncover the root cause factors behind the outliers. In application such as cloud service monitoring, Software Reliability Engineers (SRE) spends hours to find the root cause factors of outliers [13]. To illustrate this, consider Figure 4-1, which shows a simplified example of the relationships and dependencies among various components of an online store application, including the frontend service (Client App) and the other backend services (Web Service, Order Service, Payment Service, Shipping Service, and Database Service). Consider a scenario where a user experiences unusual delay or slow response time while checking out a cart. This anomaly or outlier might be attributed to the latency issues within the web service, order service, shipping service, and payment service as these services are interconnected. However, the root cause factors of the outlier might be trace back to a delay within the payment service. This delay then directly impacts the order service and shipping service, leading to the unusual increased latency within the web service and negatively impacting the user experience.

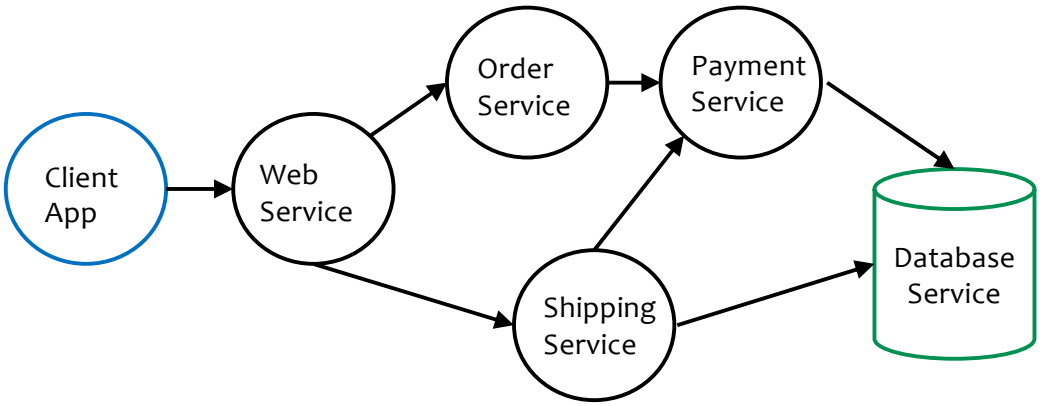


Figure 4-1 A simple example of an online shopping services and their dependencies

Some works [13, 28, 86–88] have proposed techniques to find the root cause factors of outliers including CloudRanger [13], MicroCause [28], and DyCause [46] which are tailored to data stream applications. These techniques determine the root cause factors of outliers by generating anomalous causal graphs using data-driven-based causal discovery approaches [89]. Generation of an anomalous causal graph requires a large amount of data including inliers and outliers. In real-time scenarios, depending on the data arrival rate and the memory's storage capacity, there might too few outliers available to conduct conditional independence tests on all factors or variables of the system, which can obstruct the creation of an accurate causal graph. Additionally, as noted in [88], even with sufficient data, these methods often depend on strong assumptions that should be validated by domain experts. This validation process is time consuming, leading delays in real-time outlier analysis.

In contrast, Budhatoki et al. [87] introduce CausalRCA, a method that generates root cause factors of outliers using a user-defined normal causal graph, eliminating the need for validation associated with the data-driven causal discovery method. A causal graph is

depicted as a directed acyclic graph (DAG) in which an arrow from one vertex (variable) to another reflects the influence flow from a parent to a child vertex [121]. A causal graph follows a functional causal model (FCM) [91, 93], which is a mathematical model that represents each vertex in the graph. Figure 4-1 can be seen as a causal graph of the microservices by changing the direction of each arrow. For example, instead of having an arrow from ClientApp to Web Service, the arrow should be in the opposite direction. CausalRCA assumes that an outlier is observed in the target vertex, e.g., the frontend service (Client App) in Figure 4-1, and its root cause factors can be determined by examining the upstream vertices, e.g., the backend services. CausalRCA finds the root cause factors of point outliers based on the FCM trained using normal data.

Notably, CausalRCA is designed for static data scenarios, where the amount of data is finite and the timestamp indicating when a data object is collected or arrived is not considered. Unlike static data, streaming data continuously arrive and thus their volume is unbounded/infinite. In addition, they include the notion of time where each data point including outliers has a timestamp associated with it. A root cause generation for an outlier should be based on FMC fitted with data points that occurred before the outlier. Furthermore, due to the memory size limitation, older data in the memory need to be replaced by new incoming data. We cannot assume that after an outlier is detected, data relevant to the outlier are still available in the memory.

To deal with the aforementioned challenges, we propose Ocular, an outlier explanation algorithm that is based on CausalRCA but possesses the capability to identify root cause factors of outliers in continuous real-time data streams. Unlike CausalRCA, Ocular finds the root cause factors of outliers online in real-time data streams and addresses the issues

of unbounded volume (notion of infinity), notion of time, and concept drifts of data streams. To evaluate the performance of Ocular, in this chapter we also present its time and space complexity and its accuracy and execution time compared with those of existing algorithms.

We organize the rest of this chapter as follows. In Section 4.1, we present the background on Causal Graph and Functional Causal Model. We then define the problem with underlying assumptions in Section 4.2. In Section 4.3, we review CausalRCA and explain why we need Ocular for data streams. We discuss the details of Ocular in Section 4.4 and Ocular's complexity analysis in Section 4.5. Finally, in Section 4.6 we discuss our experimental setup and results.

4.1 Causal Graph and Functional Causal Model

A causal graph provides a graphical representation of causal relationships between variables or events (i.e., vertices) [121]. It visually depicts the structure and direction of causality. It offers a qualitative view of causal connections, shedding light on the dependencies and mechanisms within a system. Causal graphs can be constructed based on prior knowledge of the applications, expert opinions, or data-driven causal discovery techniques [46, 90, 92, 94] .

In the context of timestamped data, a time series causal graph is utilized, which is a type of DAG with infinite vertices [122, 123] as depicted in Figure 4-2 a). Although Figure 4-2 a) consists of possibly infinite vertices (shown as triple dot ...), it has four main variables: X_1, X_2, X_3, and X_4. An arrow from one vertex to another indicates that a change in the source vertex causes a change in the destination vertices. For example, the value of

variable X_4 at time t is influenced by the value of variable X_3 at time t-1. In turn, the value of X_3 at time t-1 is affected by X_2 at t-2, which is influenced by X_1 at t-3 and so on.

The infinite vertices and edges in time series causal graph in Figure 4-2 a) can be condensed into a summary causal graph as shown in Figure 4-2 b) where $X_1 \rightarrow X_2$ means X_1 has a causal effect on X_2 (i.e the changes on X_1 cause the changes on X_2 or X_2 is a function of X_1). For simplicity, we refer to the summary causal graph as the causal graph throughout the rest of this Chapter.

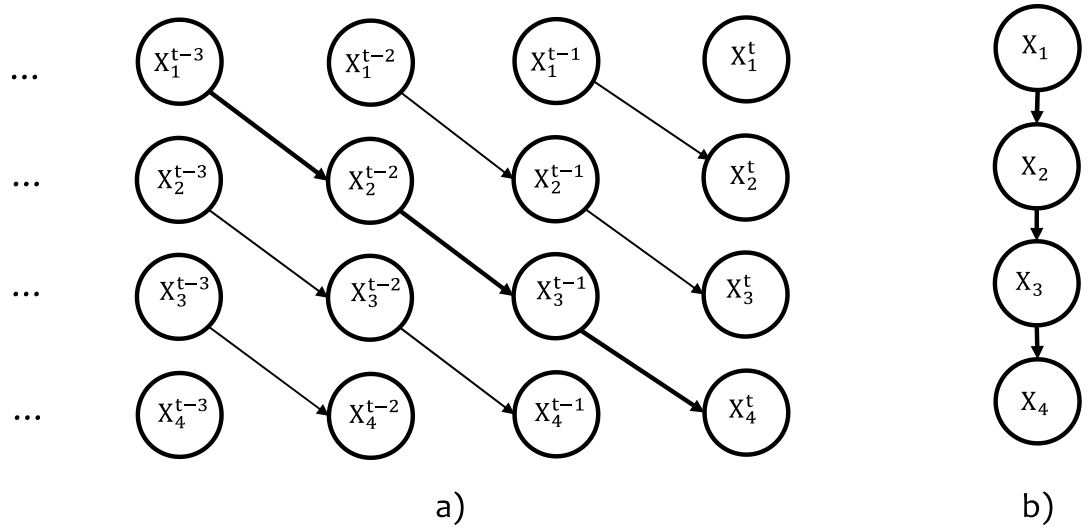


Figure 4-2 a) An example of a time series causal graph of four observed variables and b) its corresponding summary causal graph

A causal model goes beyond a causal graph by incorporating mathematical or statistical representations of these relationships [121]. It encompasses the causal graph's structure and includes quantitative information about the strength and nature of the causal effects. Causal effects refer to the impact or change in an outcome, i.e., the value of a child vertex, that can be attributed to a specific cause or intervention. By assigning mathematical equations

or functions to each vertex in the graph, a causal model specifies how changes in one vertex led to changes in others.

Functional Causal Model (FCM) or structural equation model (SEM) is a type of causal model [91–94]. In an FCM, each variable is represented by a mathematical function that describes how its value depends on the values of other variables in the model. These functions capture the causal relationships and dependencies between variables by specifying how changes in the parent vertices (independent variables) lead to changes in the child vertex (dependent variable). The corresponding FCM of the causal graph in Figure 4-2 can be written as follows:

$$\begin{aligned} \mathrm{X}_1 &= \mathrm{NT}_1 \\ \mathrm{X}_2 &= f_{\mathrm{X}_2}(\mathrm{X}_1, \mathrm{NT}_2) \\ \mathrm{X}_3 &= f_{\mathrm{X}_3}(\mathrm{X}_2, \mathrm{NT}_3) \\ \mathrm{X}_4 &= f_{\mathrm{X}_4}(\mathrm{X}_3, \mathrm{NT}_4) \end{aligned} \qquad \text{(Equations 4.1)}$$

where $f_{\mathrm{X_i}}$ represents a function that captures how the child vertex $\mathrm{X_i}$ is generated from the parent vertex $\mathrm{X_{i-1}}$. $\mathrm{NT_i}$ denotes the noise term at $\mathrm{X_i}$ and is assumed to be independent of $\mathrm{X_{i-1}}$. The noise term is introduced to accommodate the fact that other than the observed variables $\mathrm{X_{i-1}}$, there could be unobserved or unknown factors that also influence $\mathrm{X_i}$. For example, suppose we want to find out how the number of hours students spend studying affects their exam scores. In a simple deterministic model, we might represent it as *'Exam Score = f(Study Hours)'*. However, the real world is more complex than that, and various factors can introduce variability and uncertainty into this causal relationship. The noise term NT represents all the unobservable or uncounted factors that can influence the exam

score but are not explicitly modeled. The FCM can then be written as *'Exam Score = f(Study Hours, NT)'.*

4.2 Preliminaries

This section defines the problem of generating root cause factors of point outliers in data streams.

Due to the unbounded volume of a data stream and memory constraints, it is not feasible to store an entire data stream in memory for processing. Therefore, windowing is often used in data stream processing to partition the stream into finite chunks for analysis [10, 53, 124]. In Chapter 3, we use the time-based tumbling windows since we need to update eigenvectors and form clusters incrementally to determine the local contexts of outliers. We want to ensure that each arriving data point is only absorbed by the EXOS' components once. The time-based tumbling window is ideal for this scenario since it divides the data stream into non-overlapping, contiguous time interval of fixed duration. Each data points belong to exactly one window, with no overlap between consecutive windows.

However, in this Chapter, we use a time-based sliding window to discover the root cause factors of point outliers. A sliding window allows overlapping of data points between consecutive windows, meaning a data point can belong to more than one consecutive windows. This overlapping enables fine-grained and more detailed analysis, allowing Ocular to detect a small but significant change that might be missed with non-overlapping windows. Since Ocular updates its FCM only when concept drift happens and not periodically, using sliding window is preferable.

A sliding window (Definition 4.1) is defined by both a fixed window size (in hours, minutes, second, or etc) denoted as ΔT and a slide size denoted as δT (in hours, minutes, second, or etc). A window size refers to the duration of the window, while slide size refers to the time interval by which the window moves forward. The window slides constantly, allowing for continuous processing of the data stream.

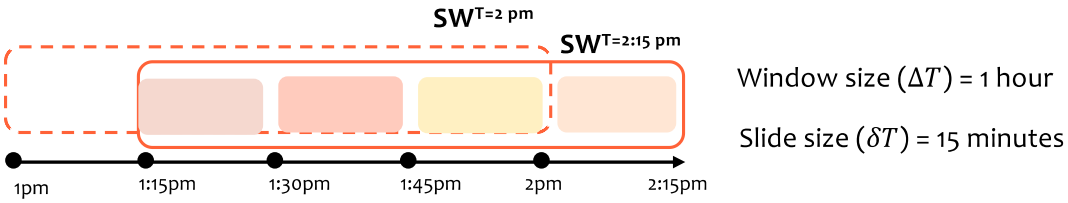


Figure 4-3 Sliding window illustration

Definition 4.1 **(Time-based Sliding Window).** Let SW^T denotes the window at time T, ΔT be the window size in given time unit, and δT be the slide size in given time unit. A sliding window SW^T is defined as: $SW^T = \{\mathbf{x} \mid T - \Delta T < \text{timestamp}(\mathbf{x}) \leq T\}$, where $T - \Delta T$ indicates the timestamp that is ΔT units before T. This means that SW^T includes all data points of the data stream (S) whose timestamps t are within the intervals $(T-\Delta T, T]$. The window slides forward by δT at each step. If T_p represents the time at the p-th window position, then $T_{p+1} = T_p + \delta T$. Consequently, the $SW^{T_{p+1}} = \{\mathbf{x} \mid T_{p+1} - \Delta T < \text{timestamp}(\mathbf{x}) \leq T_{p+1}\}$.

Figure 4-3 depicts a sequence of sliding windows of window size ΔT = 1 hour and sliding size δT = 15 minutes. In this figure, the current sliding window at time T = 2:15 pm ($SW^{2:15\ pm}$) contains any data points that have timestamps between 1:15 pm and 2:15 pm. If we look closely $SW^{2:15\ pm}$ has four intervals (Slides): (1:15 pm, 1:30 pm], (1:30 pm,

1:45 pm], (1:45 pm, 2 pm], and (2 pm, 2:15 pm]. The first three Slides of $SW^{2:15\ pm}$ contains data points that overlapped with the previous sliding window $SW^{2\ pm}$.

The process of finding the root cause factors of outliers relies on the causal graph defined below.

Definition 4.2 **(Causal Graph).** Let G = (V, E) be a graph where the set of vertices V= $\{X_1, X_2, ..., X_D\}$ depicts the set of variables in a system and E be the set of directed edges representing the causal relationships between the variables. Depending on the database design, those variables can be considered as properties of the same entity or different entities. A causal graph is a directed acyclic graph where each directed edge that connects vertices X_a and X_b, $(X_a, X_b) \in E$, represents a causal relationship from variable X_a to variable X_b, meaning the value of variable X_a affects the value of variable X_b.

Definition 4.3 **(Root cause factors of an Outlier).** Given a causal graph, G, with a set of vertices V= $\{X_1, X_2, ..., X_d\}$, an outlier value of a target vertex X_b with timestamp *t* denoted as o_t, and the values of all variables or vertices in DAG, the root cause factors of o_t are defined with the following steps:

- Let $A(X_b)$ be the list whose members consists of X_b 's ancestors and X_b itself, $A(X_b) = \{X_i \in V \mid \text{there exists a directed path from } X_i \text{ to } X_b \text{ in } G\} \cup \{X_b\}$
- Let h be the function that computes the causal contribution values of vertices in $A(X_b)$ to outlier o_t. The h function's parameters consist of the causal graph G, $A(X_b)$, the values of variables in $A(X_b)$, and o_t. It returns a key value set denoted as $CCVo_t$ where the key represents a variable X_a and the value represent ccv_{X_i}, a causal contribution value of the variable X_a.

- The root cause factors of o_t, are defined as the variables that corresponds to the highest values in $CCVo_t$.

***Problem Definition*.** Given a data stream, a causal graph, and an outlier, find the root cause factors of the outlier.

Moreover, it is imperative to acknowledge the underlying assumptions concerning the causal model:

- Instead of using a time series causal graph with infinite vertices for timestamped data, we employ a summary causal graph (or causal graph for clarity) as a part of the Ocular's input. The causal graph, DAG, comes with the functional causal model (FCM) such that each variable or vertex X_a in the DAG is a function of its parents $P(X_i)$ and a noise term NT_a.

$$X_a = f_{X_a}(P(X_a), NT_a)$$

- The root cause factors of an outlier of variable X_b are discovered from the variables that belong to the set of ancestors of X_b including X_b ($A(X_b)$). It implies that the discovery process uses any objects with the properties of $A(X_b)$ with timestamps older than of the outlier.

Furthermore, unlike EXOS that takes advantage of data streams' cross-correlation to find the outlying attributes of an out, Ocular operates independently on each data stream where outliers are detected. For the remainder of this Chapter, we address an outlier value of a target vertex X_b at time t as an outlier o_t for simplicity.

4.3 The CausalRCA Algorithm

In [87], Budhatoki et. al. present CausalRCA an algorithm that find root cause factors factors of outliers in static data. They assume that the causal graph that shows the dependencies among variables (i.e vertices) is given and the corresponding values of all vertices are jointly observed. In their scenario, point outliers are detected in a target vertex and the algorithm finds the root cause factors of each outlier by computing how each noise term NT_a of each vertex X_a in the causal graph affects the outlier's score.

CausalRCA relies on the assumption that the causal graph follows FCM where a child vertex is a function of its parents and noise term. The child vertex can then be recursively written as a function of all its ancestor noise terms. For example, the FCM in Equations 4.1 can be written as follows:

$$\begin{aligned} \mathrm{X}_1 &= \mathrm{NT}_1 \\ \mathrm{X}_2 &= f_2(\mathrm{NT}_1, \mathrm{NT}_2) \\ \mathrm{X}_3 &= f_3(\mathrm{NT}_1, \mathrm{NT}_2, \mathrm{NT}_3) \\ \mathrm{X}_4 &= f_4(\mathrm{NT}_1, \mathrm{NT}_2, NT_3, \mathrm{NT}_4) \end{aligned} \qquad \text{(Equations 4.2)}$$

Consequently, the function of a vertex X_a can be written as $X_a = f_j(NT_{root}, ..., NT_a)$.

Based on the assumption that we can recursively resolve the children in terms of their parents until we reach the root vertex, CausalRCA asks a counterfactual question to tell whether X_a is a root cause of an outlier value o of the target vertex. The counterfactual question is *"If we assigned rather a normal mechanism at X_a by randomizing NT_a, would o have not been an outlier?"*. If indeed after randomizing NT_a, o remains an outlier, then X_a is not a root cause. We can see that the noise terms play an important role in determining the root cause factors of an outlier.

CausalRCA takes a causal graph, inlier dataset, and point outliers as inputs. As we have described in Chapter 2, CausalRCA follows **six steps** to generate the root cause factors of each outlier. For clarity, we repeat the description of those Steps below. Note that each instance in the inlier and outlier dataset is a jointly observed value of each vertex in the causal graph.

***Step 1*:** It samples *m* normal data to train or fit the functional causal model (FCM). If a vertex is a root in the causal graph, it uses the sampled data associated with the vertex to find the vertex's distribution type and its corresponding parameters. If a vertex has parents, the algorithm uses its parents as independent variables and the vertex itself as a dependent variable in the prediction model (e.g., linear regression model). It trains the model with the parents' sampled data as the predictor and the vertex's sampled data as the target. It then fits the noise term of the vertex using the difference between the estimated value of the vertex generated by the prediction model and the actual value of the vertex in the sampled data.

***Step 2*:** It applies the noise term of each vertex trained in Step 1 to generate M Noise Samples and corresponding M Vertex Samples. The Noise Samples represent M noise term values of vertices while the Vertex Samples represent M values of vertices. The Noise Samples and Vertex Samples generations are done iteratively from the root to the leaf vertices.

***Step 3*:** It fits the statistical outlier scoring function of the target vertex using the values of the target vertex in the Vertex Samples.

***Step 4*:** For each outlier, it generates a tuple of outlier noise using the prediction model of each vertex trained in Step 1. If a vertex is a root, the corresponding outlier noise value is generated from the noise term.

***Step 5*:** Given the outlier noise in Step 4, starting from the root to the leaf vertices, the algorithm estimates the vertex values of the outlier noise using the FCM fitted in Step 1. It then computes the outlier score of the estimated target vertex's value using the outlier scoring function from Step 3.

***Step 6*:** It uses the FCM, Noise Samples, outlier scoring function, outlier noise, and outlier score generated in Steps 1, 2, 3, 4, and 5, respectively, to estimate the Shapley value [87, 95] of each vertex for the outlier. The Shapley value is a method from game theory to distribute the gain of a game to the players. Each vertex's Shapley value represents the contributions of the vertex (the player) as the root cause. The vertices with the highest absolute Shapley values are the root cause factors of the outlier.

CausalRCA is designed for static data only. Applying CausalRCA to continuous real-time data streams presents several challenges. While one might initially consider using it as a batch algorithm, there are key issues to address. Firstly, CausalRCA lacks a mechanism for considering timestamps within the dataset (it does not consider the notion of time). Consequently, there is a need to incorporate a method that handles the propagation of values from ancestor vertices to target vertices at specific time points. Additionally, CausalRCA does not account for the chronological context required to determine the causality of an outlier. In other words, explaining an outlier at time *t* necessitates that the FCM used to generate the Noise Samples and the outlier scoring function must be trained or fitted using the data that predates time *t*, rather than data occurring after it. In addition,

training a new FCM when the window slides will only be necessary if a concept drift happens when a new batch of data arrives. However, if the data distribution change only takes place occasionally, fitting a new FCM whenever a new slide arrives will just add unnecessary overhead. Moreover, with the continuous arrival and unbounded volume of data streams (the notion of infinity), it becomes infeasible to retain all data in memory continuously due to the limitation of memory size. Consequently, semantically relevant data needed to explain an outlier at time *t* may no longer be available in memory when the algorithm finishes explaining the previous batch of outliers. The question then arises: *How can these specific requirements be addressed?* In response to these challenges, in this chapter, we propose Ocular that is based on CausalRCA, yet tailored to continuous real-time data streams and addresses their notion of time, notion of infinity, and concept drifts.

4.4 Ocular: Outlier Causality Explanation for Data Streams

4.4.1 The Overview of Ocular

When talking about causality explanation, logically, an anomalous event or object (i.e., the effect) is caused by some events that happen before the outliers. Hence, the root cause factors an outlier with timestamp t should be generated using functional causal model (FCM) fitted with data points that predates t. However, the memory capacity limitation implies that data points can only be stored in the memory for a short amount of time before they are replaced with new incoming data points. Thus, it is a challenge to ensure the outlier is explained with the right FCM.

We propose four facets of the Active FCM related Data Structures (AFDS) to ensure that the root cause factors of each outlier are derived using the FCM fitted with data points whose timestamps predates the outlier's timestamp. Recall that CausalRCA generates an FCM for each outlier. However, if data distribution does not change (no concept drift happens), the same FCM can be used to explained outliers. Thus, AFDS maintains the information about the starting time and the end time of a generated FCM along with its related information such as the noise term values of each vertex and the outlier scoring function of the target vertex from time to time. We will explain the AFDS further in Section 4.4.2.

As explained in Section 4.2, we adopt sliding windows to deal with the continuous arrival and unbounded volume of data streams (notion of infinity) on each stream. Sliding window allows overlapping data points between consecutive windows, such that when the previous window expires, the current window still contains some data points seen in the previous window. This detail is useful to decide whether there is a change of data distribution (concept drift). If concept drift happens, Ocular needs to train or fit a new functional causal model (FCM). The new FCM and its related information is then stored in the Active FCM related Data Structures (AFDS).

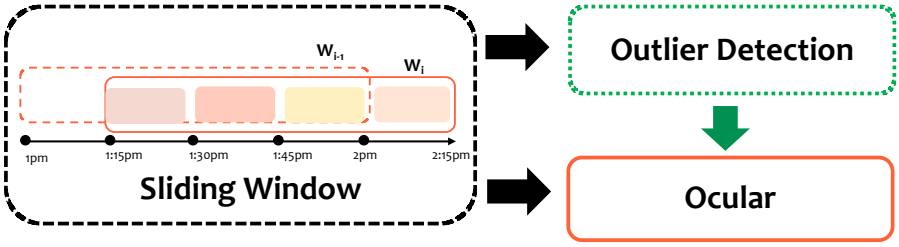


Figure 4-4 Sliding Window, Outlier Detection, and Ocular Interactions

AFDS is created in the Offline Phase or initialization phase in which Ocular uses static dataset that contains the values of all variables in the causal graph along with their timestamp to train an initial FCM. Ocular then runs on Online Phase, where it discovers the root cause factors of each outlier value of the target vertex found in each sliding window.

Figure 4-4 describes the interactions between the sliding window process, outlier detection, and Ocular during the Online Phase. In this example, the window size is 1 hour, and the sliding size is 15 minutes. Suppose the last time the window slide was at 2 pm. At 2:15 pm a new slide arrives, both the outlier detection and Ocular will absorb the data from the latest or current sliding window. Ocular uses the data to check whether it needs to update the AFDS. We call this process 'Concept Drift Checker and AFDS Updates'. At the same time, Ocular will also generate the root cause factors for the point outliers detected in the previous window using the process 'Root Causes Generation' .

Before we explain further about 'Root Causes Generation' and 'Concept Drift Checker and AFDS Updates', note that the sliding window process, outlier detection, and Ocular are independent processes (shown in Figure 4-4). They transfer information through message queues (shown as bolded arrows in Figure 4-4). A message queue is an inter-process communication (IPC) mechanism that allows information or data exchange between two processes.

Figure 4-5 depicts the parallel processes of discovering root cause factors of each outlier value of the target vertex and checking concept drift and updating AFDS. The 'Root Causes Generation' process requires the set of outliers' values the target vertex X_b identified in the previous sliding window (e.g. $SW^{T=2\ pm}$), the causal graph (DAG), and AFDS.

Meanwhile, the 'Concept Drift Checker and AFDS Updates' process needs data points from current sliding window (e.g. $SW^{T=2:15pm}$), DAG, and AFDS.

Inside the 'Root Causes Generation' process, Ocular goes through the following steps for each outlier value of the target vertex identified during the previous sliding window ($SW^{T=2\ pm}$): Step (1): Get the timestamp of the outlier and use it to determine which identifier in the AFDS to use to find the root cause factors of the outlier; Step (2): Compute the outlier noise term values that include the noise values of the target vertex and its ancestor vertices, and in parallel, use the FCM from the selected AFDS' identifier to generate the outlier score of each permutation of $A(X_b)$ which is a set containing the target vertex X_b and its ancestor vertices; Step (3): For each vertex in $A(X_b)$, compute the difference between the outlier score when a vertex is included and when it is not. Do the computation for each pair of permutations in Step (2) and aggregate the results to determine the causal contribution value of the vertex; and Step (4): In descending order, sort the vertices based on their causal contribution values. The first *k* vertices are the root cause factors of the outlier.

Inside the 'Concept Drift Checker and AFDS Updates' process, Ocular checks whether there is a change in the data distribution by computing the prediction error of the target vertex X_b using all values of variables X_b and its parents stored in the current sliding window (e.g. $SW^{T=2:15pm}$). If the prediction error of X_b is less than a threshold, meaning no concept drift, Ocular will update the end time of the latest FCM in the AFDS. Otherwise, it will train a new FCM using the values of all variables in the current window and then add the new FCM and its related information into the AFDS.

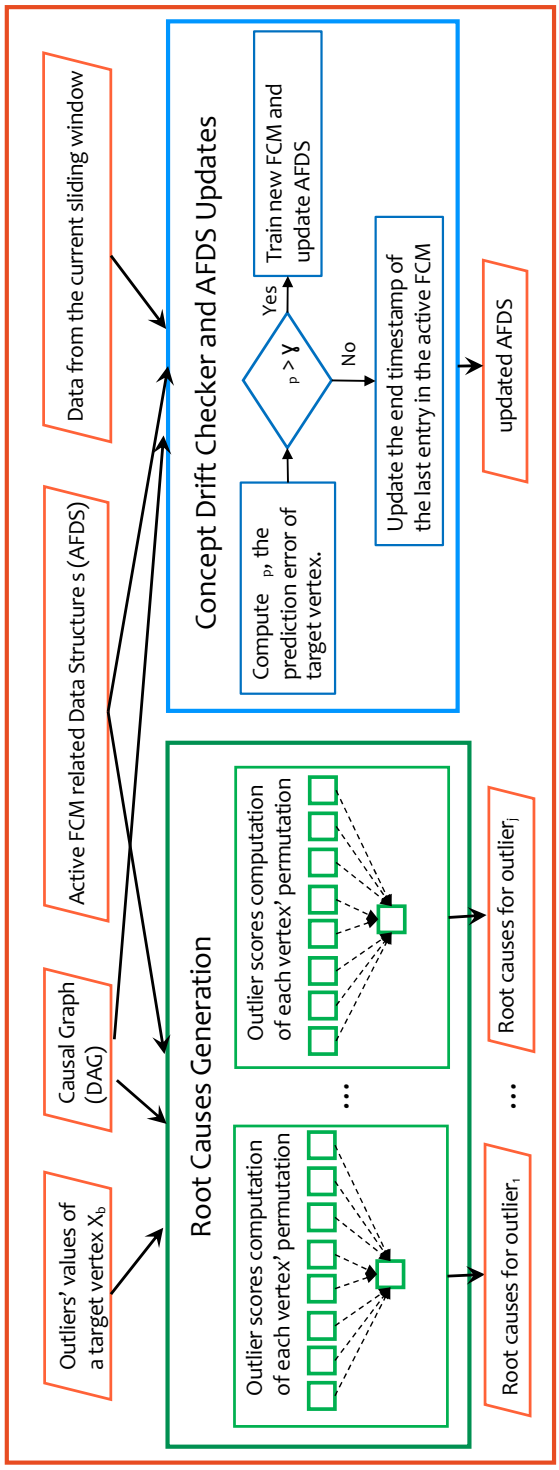


Figure 4-5 The Ocular framework (Online Phase)

4.4.2 The Active FCM related Data Structures (AFDS)

CausalRCA does not consider the notion of time of the data points which is important in data streams. To be able to answer the counterfactual question *"If we assigned rather a normal mechanism at a vertex X_j by randomizing the noise term NT_j, would o have not been an outlier?"*, we want to ensure that outlier value of the target vertex X_b at time *t* denoted as o_t will be explained by employing the noise terms generated using the FCM fitted with data whose timestamps predates the timestamp of the outlier. Let's consider Figure 4-6 as an example where at 2:15 pm a new slide just arrives and the outlier detection tells us that the value of the target vertex X_4 at 2 pm, 1:40 pm, and 1:16 pm are outliers. The root cause factors of the outliers o_{2pm}, $o_{1:40pm}$, and $o_{1:16pm}$ should be generated using the noise terms generated from FCM fitted before 2 pm, 1:40 pm and 1:16 pm respectively.

One possible approach could be to train a new FCM every time a new Slide arrives (i.e a new sliding window formed), ensuring that we have different noise terms corresponding to each Slide. However, considering we need to fit each vertex's functional model, this approach is inefficient. If the underlying data distribution in the current sliding window SW^{T_p} remains similar to that in the previous sliding window $SW^{T_{p-1}}$, that is no concept drift occurs when the window slides, then there is no need to train a new FCM. Instead, we should only fit a new FCM when a concept drift occurs. To address this requirement, we propose the use of the in-memory data structures AFDS that allow us to efficiently manage the FCMs and their corresponding noise term samples.

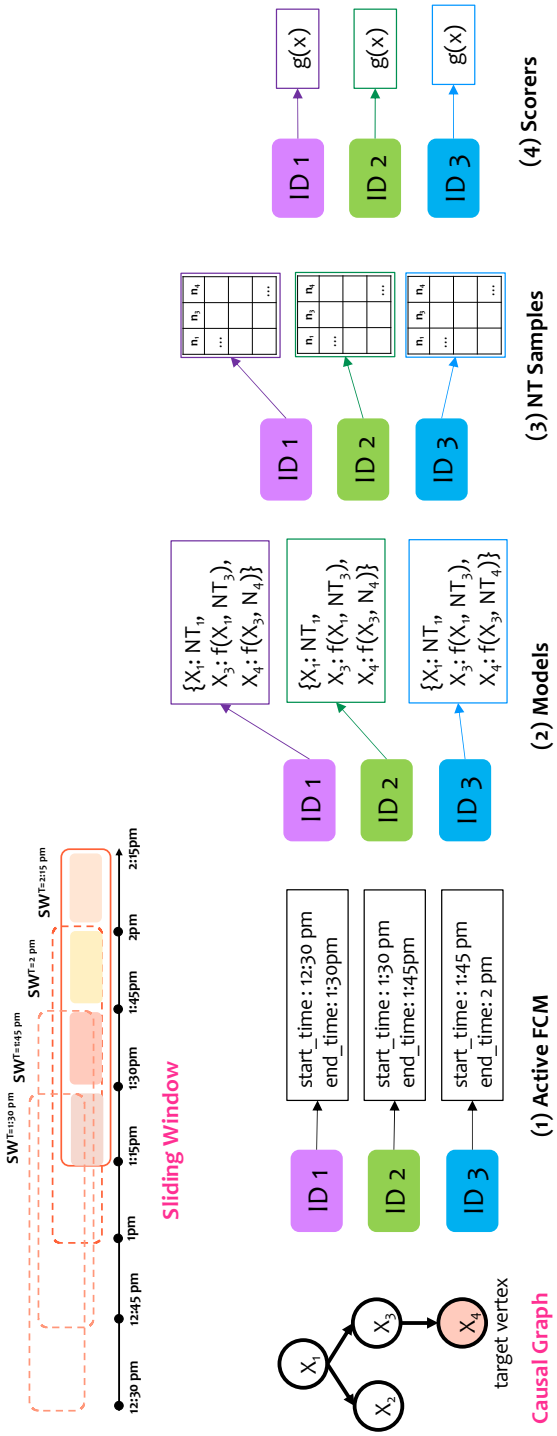


Figure 4-6 Four facets of the Active FCM related Data Structures (AFDS)

As shown in Figure 4-6, the first part of the AFDS is *Active FCM*, which acts as a hash map. It maintains unique identifiers (IDs) as keys, each pointing to the start and end times of the data with which an FCM is trained or fitted. This structure enables us to track the time period for which an FCM remains valid. The second and third parts, the *Models* and *NT Samples*, share the same keys as the *Active FCM*. In the *Models*, each ID corresponds to an FCM, while in the *NT Samples*, each ID represents a set of *M* samples of noise term values generated using FCM corresponding to that ID. Lastly, we have the *Scorers*, which facilitates the computation of outlier scores in the target vertex X_b. The *Scorers* is a key value set where a key corresponds to an ID in the ActiveFMC and the key's associated value represents a fitted outlier scoring function of the target vertex X_b. This outlier scoring function is used during the computation of root cause factors of outliers. Hence each fitted outlier scoring function should corresponds to an FCM.

The *Active FCM's* ID is initialized when an FCM is trained for the first time during an Offline Phase (described further in Section 4.4.3.1). It is incremented whenever a concept drift is identified within an active sliding window, which we will describe in Section 4.4.3.3.

4.4.3 The Ocular Algorithm

The Ocular algorithm as outlined in Algorithm 4.1, consists of two phases: (1) Offline or Initialization Phase which is conducted before Ocular starts receiving incoming data points and (2) Online Phase which is a phase where Ocular continuously process arriving data points to discover root cause factors of any identified outliers. Ocular operates on a data stream, requiring initial data to initialize AFDS, the specification of the sliding window's size (T) and its slide size (s), a causal graph corresponding to the data stream

system, a target vertex in the causal graph (X_b), the number of samples (M) to be generated by a functional causal model (FCM) in each causal graph's vertex denoted as M, the outlier scoring function (g) determining whether a data point is an outlier in the target vertex, and the FCM prediction error threshold (γ) of the target vertex, dictating FCM updates when a new window slides.

In the Offline Phase (Algorithm 4.1, Line 1), the algorithm finds a list that consists of all the ancestors of the target vertex Xb and Xb itself ordered based on the topological order of the causal graph starting from the root to the target vertex (Algorithm 4.1, Line 1). The sorted vertex ancestor list omits any vertices that have no path to the target vertex since they do not affect the root cause factor discovery. Given the initial data, causal graph, sorted vertex ancestor list, number of samples stored in *NT Samples*, and outlier scoring function, the algorithm then initializes the four facets of the AFDS: *Active FCM, Models, NT Samples,* and *Scorers* (Algorithm 4.1, Line 2). It also initializes the TargetRMSEs data structure used to keep track of the RMSEs of the *Models*. TargetRMSEs is a key value set in which a key represents an ID of an FCM, and the key's corresponding value represents the root mean squared error (RMSE) of prediction function at the target vertex X_b. We describe the AFDS initialization further in Section 4.4.3.1.

Prior to running the Online Phase indefinitely, Ocular sets the start time and the end time of the sliding window (Algorithm 4.1, Lines 3-4). It also initializes the sliding window queue and the outlier queue serving as communication channels with the sliding window and outlier detection processes respectively (Algorithm 4.1, Lines 5-6). Recall that the sliding window process, outlier detector, and Ocular operate as independent processes; therefore, queues are used for the inter process communication.

Algorithm 4.1: **OCULAR**(S, init_data, ΔT, δT, causal_graph, X_b, M, g, γ)

Input: a data stream S, initialization data init_data, window size ΔT with a unit of time, slide size δT in a unit of time, causal_graph, target vertex X_b, number of samples M, outlier scorer g, target vertex error prediction threshold γ

Output: root cause factors of each outlier in every sliding window

Initialization or Offline Phase

```
## get a list consisting of the target vertex's ancestors and the
target vertex itself based on the topological order ##
```

1 sorted_A(X_b) =get_topological_order_of_Xb_ancestors_and_itself(causal_graph, X_b)

```
## initializing the four facets of Active FCM related Data
Structure (AFDS) and TargetRMSE, the called function is described
in Algorithm 4.2, Section 4.4.31 ##
```

ActiveFCM, Models, NTSamples, Scorers, TargetRMSE = AFDS_Initialization

2 (init_data, causal_graph, X_b, M, g, sorted_A(X_b))

3 end_ts = current_time() `# the end timestamp of the initial sliding window`

4 start_ts = end_ts - ΔT `# the start timestamp of the initial sliding window`

```
## initializing a queue as an inter-process communication between
the sliding window process and Ocular. The queue is used to get
data points from in a current sliding window ###
```

5 W_queue = initialize_sliding_window_queue(start_ts, end_ts, ΔT, δT)

```
## initializing a queue used to get the information about outliers
detected in each sliding window from the outlier detector ##
```

6 outlier_queue = initialize_outlier_queue (start_ts, ΔT, δT)

Online Phase

7 **while (true):** ## repeat indefinitely

```
## get the current sliding window (SW^T), the end time of  SW^T
   (end_ts), the p-th number of the  SW^T ##
```

8 SW^T, end_ts, latest_slide_number = W_queue.get(start_ts, ΔT, δT, S)

```
## get outliers from the queue detected between the starting
   timestamp of the current sliding window and the end time of
   the preceding sliding window ##
```

9 $outliers_T$ = outlier_queue.get(start_ts, end_ts, δT)

```
## get the root cause factors of outliers_T, the called function
   is described in Algorithm 4.5, Section 4.4.3.2 ##
```

yield root_causes = online_ocular (SW^T, start_ts, end_ts, ΔT, δT, $outliers_T$, causal_graph, Xb, ActiveFCM, Models, NTSamples, Scorers, sorted_A(X_b), TargetRMSEs, M, g, γ,

10 latest_slide_number)

11 curr_time = current_time()

12 **if** (end_ts + δT < curr_time)**:** `## wait until a new Slide arrive`

13 wait until current_time equal to end_ts + δT

14 **else:** `## if the explanation taking longer than` δT

15 n_missing_sliding_window = floor((curr_time - end_ts)/ δT)

16 end_ts = end_ts + (n_missing_sliding_window * δT)

17 **end if else**

```
18      start_ts = end_ts - ΔT
19  end while
```

The Online Phase operates continuously (Algorithm 4.1, Lines 7-19). Upon the arrival of the new set of data in the sliding window, it retrieves the latest data from the window queue (Algorithm 4.1, Line 8). It also collects outliers with timestamps falling within the start time of the current sliding window and the end time of the preceding sliding window (Line 9). Subsequently, it invokes the online_ocular function that uses the information of the outliers and the latest sliding window along with the mutable AFDS data structures, Target RMSE, causal graph, X_b, M, g, and γ to find the causal explanation of the outliers and at the same time to update AFDS (Algorithm 4.1, Line 10). For a detailed explanation of the online_ocular algorithm, refer to Section 4.4.3.2.

Furthermore, in each iteration of the Online Phase, Ocular checks whether the time spent finding the root cause factors of each outlier does not exceed the slide size (Algorithm 4.1, Lines 11-18). Due to the continuous arrival and unbounded volume of the data stream, the sliding window buffer consistently discards the oldest slide when a new one arrives. If the online_ocular function takes longer than the slide size, the algorithm misses the opportunity to accurately explain outliers that happened within the discarded slides. Therefore, it is important to run the tasks of explaining outliers and updating AFDS in parallel, ensuring that the running time of online_ocular is less than the slide size.

We are now describing the details of the Offline Phase and the Online Phase of the Ocular algorithm.

4.4.3.1 AFDS Initialization (The Offline Phase)

Before running the Online Phase illustrated in Figure 4-5, Ocular needs to initialize the AFDS in the Offline Phase given in Algorithm 4.2.

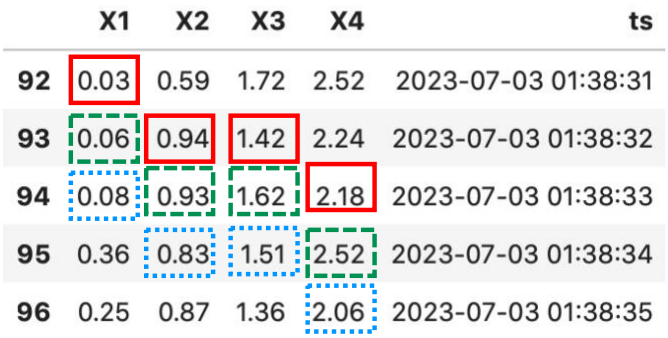
Algorithm 4.2: FCM_Initialization (init_data, causal_graph, X_b, M, g, sorted_A(X_b))

Input: init_data, a set of tuples, each tuple consist of the values of each variable in the causal graph and a timestamp, the user given normal causal graph (causal_graph), target_vertex (X_b), number of samples of noise term values (M), outlier scoring function (g), the set that consisted of the ordered ancestor of X_b and X_b itself sorted_A(X_b)

Output: The four facets of Active FCM related Data Structure (AFDS): (1) ActiveFCM, (2) Models, (3) NTSamples, and (4) Scorers, and TargetRMSEs.

- ActiveFCM is a key-value set where a key is a functional causal model (FCM)'s ID and a value is a FCM's start and end time stamp.
- Models is a key-value set where a key is a FCM's ID and a value is a FCM.
- NTSamples is a key-value set where a key is a FCM's ID and a value represents a group M NoiseTerm values of all variables or vertices in the causal graph.
- Scorers is a key-value set where a key is a FCM's ID, and a value represents a fitted outlier scoring function.
- TargetRMSE is a key-value set where a key is a FCM's ID and a value represents the root mean square error of $f_{X_b}(P(X_b), NT_b)$ that represents X_b as a function of its parent vertices $P(X_b)$ and its noise term NT_b.

```
1  ## aligned init_data according to the causal graph
   init_data = process_timeseries (init_data, causal_graph, target_vertex)
2  ## initialize an empty key value set (HashMap) for each AFDS
   ActiveFCM={}, Models={}, NTSamples={}, Scorers={}, TargetRMSE = {}
3  ## set ID = 1
   ActiveFCM[1] = {start_time : time.now(), end_time: time.now() }
4  ## initialize Models by fitting an FCM using init_data, the called
      function is described in Algorithm 4.3 ##
   Models[1] = fcm _fitting(init_data, causal_graph, target_vertex, sorted_A(Xb))
5  ## generate M noise term values and M predicted values of all
      variables in the causal graph using the initial FCM generated in
      Line 4, the called function is described in Algorithm 4.4 ##
   NTSamples[1], X̂= generate_ noise_term_samples (Models[1], causal_graph,
                                                  sorted_A(Xb), M)
6  ## fit the outlier scoring function using the predicted values of
      the target vertex Xb ##
   Scorers[1] = g.fit(X̂[Xb])
7  ## compute the root mean squared error between the actual values of
      the target vertex Xb and the predicted values of Models[1][Xb]##
   TargetRMSE[1] = compute_target_vertex _rmse (init_data[Xb], Models[1][Xb])
8  return ActiveFCM, Models, NTSamples, Scorers, TargetRMSE
```



	X1	X2	X3	X4	ts
92	0.03	0.59	1.72	2.52	2023-07-03 01:38:31
93	0.06	0.94	1.42	2.24	2023-07-03 01:38:32
94	0.08	0.93	1.62	2.18	2023-07-03 01:38:33
95	0.36	0.83	1.51	2.52	2023-07-03 01:38:34
96	0.25	0.87	1.36	2.06	2023-07-03 01:38:35

Figure 4-7 Aligning the dataset of four vertices based on the causal graph

The Initialization or Offline Phase uses an offline dataset (init_data) to fit the functional model of each vertex in the causal graph that has a path to the target vertex. The offline data set consists of the values of each variable in the causal graph that was stored in a database. It can be seen as a set of tuples; each tuple is denoted as $<x_1, x_2, ..., x_d, t>$ where d is the number of vertices in the causal graph, x_i is the value of the i-th vertex and t is the associated timestamp of each value in the tuple. Before fitting the dataset into each vertex's functional model, we need to process the dataset so that the value of each vertex would be aligned according to its parent-child relationship depicted in the causal graph (Algorithm 4.2, Line 1). For example, referring to the causal graph having four vertices and X_4 as the target vertex in Figure 4-6 and the dataset snapshot in Figure 4-7, X_1's value at row 92 will be aligned with the values of X_2, X_3, and X_4 at rows 93, 93, and 94 respectively. The new tuple uses the timestamp of the target vertex. In this example, the new tuple of row 92 becomes <0.03, 0.94, 1.42, 2.18, 2023-07-03 01:38:33>. Notably, if in the summary causal graph, a vertex affects the target vertex in more than one path, we use the shortest path to decide which value of the vertex to align with the target vertex's value.

The ActiveFCM's ID is initialized to 1, and its corresponding start and end times are set to the current timestamp (Algorithm 4.2, Line 2). The FCM for the Models of ID 1 is trained and then the noise term values are generated and stored in the NTSamples of ID 1 (Lines 3-4). We explain how to fit FCM and generate noise term values in Algorithms 4.3 and 4.4. In addition to the noise term values, the predicted values of the target vertex, $\widehat{\mathbf{X}}[X_b]$, are also generated and used to fit the outlier scoring function (Algorithm 4.2, Lines 5-6). The algorithm computes the root mean squared error (RMSE) of $f_{X_b}(P(X_b), NT_b)$ that represents the target vertex X_b as a function of its parent vertices $P(X_b)$ and its noise term NT_b. $f_{X_b}(P(X_b), NT_b)$ is shown as Models[1][X_b]) in Algorithm 4.2, Line 7. The RMSE calculation is done by computing the difference between the predicted values of $f_{X_b}(P(X_b), NT_b)$ and the actual values of X_b which is represented as init_data[X_b] in Line 7.

Fitting the Functional Causal Models (Algorithm 4.3). This algorithm is used during the offline and Online Phases of Ocular. It takes a dataset (**data**), a causal graph (causal_graph), a target vertex (X_b), and the list of X_b' ancestors and X_b itself ordered from the roof to X_b as inputs. Training a functional model of a vertex requires the data of its parent vertices unless it is a root vertex. Hence, we need sorted_A(X_b) so that the iteration is always started from the root vertex and continued down to the target vertex (Algorithm 4.3, Lines 2 -14).

If a vertex is a root, its distribution type which is also its noise term's distribution type is determined using a stochastic model that allows to sample from any parametric distributions implemented in Scipy [125]. The data associate with the vertex is used to estimate the underlying data distribution of its noise term. The data distribution could be

a Gaussian distribution with some mean and standard deviation, a beta distribution with some alpha or beta parameters, or any other types of data distributions. The stochastic model is employed to find the most suitable distribution from several possible distribution types. When a data distribution is fitted, a number of samples which is equal to the number of the vertex's values are generated. The generated samples and original vertex's values are then used to estimate KL divergence using the k-nearest neighbors [126]. If the KL divergence of a fitted distribution is smaller than a threshold, then the distribution is selected to represent the vertex.

The functional causal model of a non-root (child) vertex has the information about its prediction model and its noise term. When fitting the prediction model of each vertex in the causal graph, the data of the vertex's parents are used as the predictor or independent variables while the data of the vertex are used as the target or outcome of the function (e.g., linear regression model). For example, to train X_4's function in Figure 4-6, we need data from X_3 as the predictors and data from X_4 as the outcomes. Recall that during the initialization phase, we use init_data which consist of multiple tuples that can be represented as tabular data in which a column header represents a variable or vertex's name and the rest of the row in the column represents the variable's values. The differences between the predicted values and the actual values of a child vertex are used to determine its noise term distribution (Algorithm 4.3, Lines 11-12). The process of selecting the most suitable distribution for the noise term of the child vertex is the same as of the root vertex described in the previous paragraph.

Algorithm 4.3: **fcm_ fitting**(data, causal_graph, X_b, sorted_A(X_b))

Input: **data** the initial dataset, causal_graph the causal graph, X_b target_vertex, sorted_A(X_b) the list of X_b' ancestors and X_b itself ordered from the roof to X_b

Output: fcm

```
1  fcm = {}
2  for each vertex in sorted_A(Xb)
3    ## get the parent vertices of the vertex
     parents = causal_graph.parents (vertex)
4    if vertex is root ## no parents
5      ## find data distribution of the root vertex using a
       stochastic model ##
       fcm[vertex] = find_distribution (data[vertex])
6    end if
7    else
8      predictors = data[parents] ## get the values of the parent vertices
9      outcomes = data[vertex]    ## get the values of the vertex
10     ## train a prediction model which a functional causal model of
       the vertex ##
       fcm[vertex] = train_prediction_model (predictors, outcomes)
11     ## predict the values of the vertex based on the values of its
       parents ##
       estimated_outcomes = fcm[vertex].predict (predictors)
12     ## find data distribution of the noise term of the vertex
       using a stochastic model ##
       fcm[vertex].noise_term = find_distribution (outcomes - estimated_outcomes)
13   end else
14 end for
15 return fcm
```

Generating the Samples of Noise Term Values and the Samples of Predicted Vertex Values (Algorithm 4.4). As explained in Section 4.4.2, the Active FCM related Data Structures (AFDS) includes storing the noise term samples of each vertex and the outlier scoring function. Algorithm 4.4 shows how the noise term and vertex values are generated. It starts by initializing two empty data frames to store the noise term and vertex values. Each data frame's columns consist of all vertices in the sorted A(X_b) (Algorithm 4.4, Lines 1-2). Like Algorithm 4.3, it will iterate each vertex in the sorted A(X_b) from the root to the target vertex. The noise and predicted vertex samples of the root are generated

by drawing M random samples from the noise term distribution of that vertex (Algorithm 4.4, Lines 5-8). For each non-root vertex, its noise term values are set by randomly generating M samples from the noise term distribution. The noise term values are then used together with the vertex's parents' values to generate M samples of the predicted vertex values (Algorithm 4.4, Lines 9-14).

Algorithm 4.4: **generate_ noise_term_samples**(fcm, causal_graph, sorted_A(X_b), M)

Input: functional causal model of all vertices (fcm), the user given normal causal graph (causal_graph), the list of X_b' ancestors and X_b itself ordered from the roof to X_b (sorted_A(X_b)), number of samples to generate (M)

Output: samples of noise term values **NT**, samples of predicted vertex values $\widehat{\mathbf{X}}$

```
1  initialize NT as an empty data frame with columns of sorted_A(X_b)|
2  initialize X̂ as an empty data frame with columns of sorted_A(X_b)|
3  for each vertex in sorted_A(X_b)
4    parents = causal_graph.parents (vertex)
5    if not parents ## vertex is root
6      ## get the noise term distribution
       noise_distribution = fcm[vertex].noise_term
7      ## generate M random values for the root vertex
       X̂[vertex] = NT[vertex] = noise_distribution.draw_random_samples (M)
8    end if
9    else
10     ## get M values of the vertex'parents
       parent_values = X̂[parents]
11     ## generate M random noise term values for the vertex
       NT[vertex] = fcm[vertex].noise_term.draw_random_samples (M)
12     ## get M predicted values of the vertex using its parents'
       values and its noise term values
       X̂[vertex] = fcm.predict(parents_values, NT[vertex])
13   end else
14 end for
15 return NT, X̂
```

4.4.3.2 The Online Ocular Algorithm

We now describe the online_ocular function where in parallel, Ocular finds the root cause factors of outliers whenever the sliding window slides and also updates the Active FCM related Data Structures (AFDS). Algorithm 4.5 describes the online_ocular function as shown in Figure 4-5.

Online Ocular (Algorithm 4.5). Before executing two tasks in parallel, Ocular aligns the data from current sliding window and the outliers identified between the start time of the current sliding window and the end time of the proceeding sliding window. Each outlier detected by the outlier detector, only has a value corresponding to the target vertex and its timestamp. The data alignment is done based on to their parent-child vertex dependency (Algorithm 4.5, Line 1) which is the same as the alignment for the initial dataset (Figure 4-7). After the data alignment, each aligned outlier is represented as a tuple of b+1 values where each value corresponds to each vertex in the list of X_b's ancestors and X_b itself ordered based on the topological order from the roof vertices to X_b (sorted_A(X_b)). The aligned outlier tuple also contains the timestamp information.

If the outlier set is not empty, the algorithm will in parallel find the root cause factors of the outliers and check whether the AFDS needs to be updated (Algorithm 4.5, Lines 2-7). If not, it will only do the latter (Algorithm 4.5, Lines 8-10).

The method of finding the root cause factors of outliers is described in Algorithm 4.6. Each outlier will be processed in parallel where at first the algorithm will determine which *Active FCM* ID the outlier will use based on its timestamp (Algorithm 4.6, Lines 3-4). Let's consider Figure 4-6 as an example where at 2:15 pm a new slide just arrives and the outlier detection identifies the values of the target vertex X_4 at 2 pm, 1:40 pm, and 1:16 pm are

outliers. In this scenario, o_{2pm}'s root cause factors are generated using the AFDS of ID 2 because its FCM is fitted right before the 1:45-2 pm Slide that contains o_{2pm} arrives. The same reason applies to $o_{1:40pm}$ whose root cause factors are generated using the AFDS of ID 1. $o_{1:16pm}$ is located in the oldest slide of the current sliding window and since the AFDS of ID 1 time period ranges from 12:30 pm to 1:30 pm, $o_{1:16pm}$'s root cause factors are generated using the AFDS of ID 1.

Algorithm 4.5: **online_ocular** (SW^T, start_ts, end_ts, ΔT, δT, outliers, causal_graph, X_b, ActiveFCM, Models, NTSamples, Scorers, sorted_A(X_b), TargetRMSEs, M, g, γ)

Input: the current sliding window (SW^T), window size (ΔT), slide size (δT), SW^T's start timestamp (start_ts), a set of outlier values of the target vertex X_b detected between the start time of the current sliding window and the end time of the proceeding sliding window (outliers), the user given normal causal graph (causal_graph), target vertex X_b, four facets of AFDS (ActiveFCM, Models, NTSamples, Scorers), the list of X_b's ancestors and X_b itself ordered based on the topological order from the roof vertices to X_b (sorted_A(X_b)), the key value set in which a key represents an ID of an FCM, and the value represents the root mean squared error (RMSE) of prediction function at the target vertex X_b (TargetRMSEs), number of samples to generate based on the selected FCM for each vertex in sorted_A(X_b)(M), outlier scoring function (g), target vertex error prediction threshold γ

Output: root cause factors of each outlier detected between the start time of the current sliding window and the end time of the proceeding sliding window

```
1  # align data in SW^T and outlier values of the target vertex X_b
   normal_data, outlier_data = process_timeseries(SW^T , outliers,  causal_graph, X_b,
   sorted_A(X_b))
2  if |outliers| > 0
3    do in parallel
4        ### get the root cause factors of the outliers, the called
         function is described in Algorithm 4.6 ###
         yield root_causes = root_causes_generation (outlier_data, causal_graph, X_b,
         ActiveFCM, Models, NTSamples, Scorers, sorted_A(X_b))
5        ### call the concept drift checker and get the updated four
         facets of AFDS and TargetRMSEs, the called function is
         described in Algorithm 4.9, Section 4.4.3.3 ###
         ActiveFCM, Models, NTSamples, Scorers, TargetRMSEs =
         concept_drift_checker_and_active_fcm_ds_updates(normal data,
         causal_graph, X_b, sorted_A(X_b), M, g, ActiveFCM, Models, NTSamples, Scorers,
         TargetRMSEs, start_ts, δT, γ)
```

```
6    end do in parallel
7  end if
8  else
9      ### if no outliers, get the updated four facets of AFDS and
       TargetRMSEs, the called function is described in Algorithm
       4.9, Section 4.4.3.3 ###
       ActiveFCM, Models, NTSamples, Scorers, TargetRMSEs =
       concept_drift_checker_&_active_fcm_ds_updates(normal data, causal_graph, Xb,
       sorted_A(Xb), M, g, ActiveFCM, Models, NTSamples, Scorers, TargetRMSEs,
       start_ts, s, γ)
10 end else
```

Algorithm 4.6: **root_causes_generation**(**outlier_data**, causal_graph, X_b, ActiveFCM, Models, NTSamples, Scorers, sorted_A(X_b), k)

Input: **outlier_data** the set of aligned tuples of outlier values at the target vertex and its ancestor vertices, causal_graph, target vertex X_b, four facets of Active FCM related Data Structures (ActiveFCM, Models, NTSamples, Scorers), sorted_A(X_b) the list of X_b's ancestors and X_b itself ordered based on the topological order from the roof vertices to X_b

Output: root cause factors of each outlier

```
1  root_causes = {}
2  ### process each aligned_outlier tuple whose size is the same as
   the size of sorted_A(Xb)+ 1 ###
   parallel for each aligned_outlier in outlier_data
3    ### get the timestamp of the aligned_outlier
     outlier_ts = get_outlier_timestamp(aligned_outlier)
4    ### select the right ID to explain the outlier ###
     ID = get_active_fcm_id(outlier_ts, ActiveFCM)
5    ### get the functional causal model of the selected ID ###
     fcm = Models.get(ID)
6    ### get M x | A(Xb)| matrix of noise term values of all vertices
     in sorted_A(Xb)###
     NT = NTSamples.get(ID)
7    ### get the fitted outlier scoring function of the selected ID
     ###
     outlier_scorer = Scorers.get(ID)
8    ### compute the outlier noise which is a tuple whose size is the
     same as the size of sorted_A(Xb), call Algorithm 4.7
     outlier_noise = get_outlier_noise (aligned_outlier, fcm, causal_graph,
     sorted_A(Xb))
9    ### compute the outlier's vertex contribution scores of all
     variables in sorted_A(Xb)
```

```
        vertices_contribution = compute_vertex_contribution (outlier_noise, NT, fcm,
        outlier_scorer, sorted_A(X_b), causal_graph, X_b)
10      ### sort the vertices based on their vertex contribution scores
        from the highest to the lowest
        root_causes[outlier_ts] = sort (vertices_contribution)
    end parallel for
11  return root_causes
```

4.4.3.2.1 Getting the Outlier Noise of Each Aligned Outlier

After the ID is obtained, the algorithm will get the corresponding FCM, M samples of noise term values of all vertices in sorted_A(X_b) (the list of X_b's ancestors and X_b itself ordered based on the topological order from the roof vertices to X_b), and fitted outlier scoring function (Algorithm 4.6, Lines 5-7). The noise term values of the aligned outlier or in short, outlier noise, is then determined using Algorithm 4.7 (Algorithm 4.6, Line 8). The outlier noise is generated for each vertex from the root to the target vertex (Algorithm 4.7, Lines 2-12). For a root, its outlier noise value is the same as the vertex's value (Algorithm 4.7, Lines 4-7). However, for a child vertex, its noise value is estimated by first predicting its value using its parents' values and then subtracting the predicted value from its actual value (Algorithm 4.7, Lines 7-10).

Algorithm 4.7: **get_outlier_noise** (**aligned_outlier**, fcm, causal_graph, sorted_A(X_b))

Input: a tuple whose items corresponds to the variable in sorted_A(X_b) (**aligned_outlier**), the functional causal model of the selected AFDS' ID (fcm), the user given normal causal graph (causal_graph), the list of X_b's ancestors and X_b itself ordered based on the topological order from the roof vertices to X_b (sorted_A(X_b))

Output: outlier_noise, a tuple containing the noise term value of each variable in the **aligned_outlier,** without the timestamp

```
1   set outlier_noise as a tuple with the same length of aligned_outlier without the
    timestamp
2   ### iterate over each vertex in sorted_A(X_b) ###
    for each vertex in sorted_A(X_b))
```

```
3     ### get the parent vertices of the vertex ###
      parents = causal_graph.parents(vertex)
4     if not parents ## root vertex
5       ### when a vertex is a root, its corresponding noise value is
        the same as the value of the vertex of the aligned_outlier ###
        outlier_noise[vertex] = aligned_outlier[vertex]
6     end if
7     else
8       ### if a vertex is a child vertex, first predict the value of
        the vertex by using the values of its parents in the
        aligned_outlier as an input to the vertex's fcm prediction
        function ###
        pred_value = fcm[vertex].predict(aligned_outlier[parents])
9       ### next, compute the noise term value of the child vertex by
        finding the difference between pred_value and the actual value
        of the vertex in aligned_outlier
        outlier_noise[vertex] = fcm[vertex].compute_noise( pred_value,
                                          aligned_outlier[vertex])
10    end else
11  end for
12  return outlier_noise
```

4.4.3.2.2 Computing the Contribution Score of Each Vertex of Causing an Outlier at the Target Vertex

After acquiring the outlier noise, the subsequent step involves computing the contribution of each vertex of causing the value of the target vertex at a particular timestamp identified as an outlier (Algorithm 4.6, Line 9). This procedure is outlined in Algorithm 4.8, which begins by generating **est_o** a tuple of the estimated vertex values of the outlier noise (Algorithm 4.8, Line 1). This tuple **est_o** has the same size with the **outlier_noise** tuple. This computation is done similarly when $\widehat{\mathbf{X}}$, the matrix of the predicted values of the sorted $A(X_b)$, is generated in Algorithm 4.4, Lines 3-14. Only this time, the outlier noise tuple is already generated. The outlier score of the estimated outlier value of the target vertex is then computed (Algorithm 4.8, Line 2).

Algorithm 4.8: **compute_vertex_contribution** (**outlier_noise**, **NT,** fcm, outlier_scorer, sorted_A(X_b), causal_graph, X_b)

Input: a tuple containing the noise term value of each variable in the **aligned_outlier,** without the timestamp (**outlier_noise**), M x |A(X_b)| matrix of noise term values of all vertices in sorted_A(X_b) (**NT**)**,** the functional causal model of the selected AFDS' ID (fcm), the fitted outlier scoring function of the selected AFDS'ID (outlier_scorer), the list of X_b's ancestors and X_b itself ordered based on the topological order from the roof vertices to X_b (sorted_A(X_b)), the user given normal causal (causal_graph), the target vertex (X_b)

Output: vertices_contribution, a key-value set in which a key represents a vertex in sorted_A(X_b) and its corresponding value represents the vertex's contribution score of causing an outlier in the target vertex X_b at a particular timestamp

1 ### given an **outlier_noise** tuple, get a new tuple that contains the estimated values of all vertices in sorted_A(X_b) for the corresponding **outlier_noise. est_o** has the same size with **outlier_noise** ###
est_o = estimate_outlier_values(**outlier_noise**, fcm, causal_graph, X_b)
2 ### compute the outlier score of the target vertex's value in **est_o** ###
oscore = outlier_scorer.compute(**est_o**[X_b])
3 ### generate a set that contains all possible combinations ###
vertex_participation_set = generate_set(sorted_A(X_b))
4 ### initialize an empty HashMap to store participation scores of each item in vertex_participation_set ###
ps = {}
5 **parallel for** each *vpt* in vertex_participation_set:
6 ### shuffle a copy of **NT** along with the values from **outlier_noise** based on the valued of vpt, the size of **shuffled_N** is the same as **NT**'s size ###
shuffled_N = shuffle_samples_noise_term(**NT, outlier_noise**, *vpt*)
7 ### estimate the M target vertex's values based on the **shuffled_N** ###
est_X_b = estimate_X_b_values(fcm, causal_graph, X_b, **shuffled_N**)
8 ### compute an IT score of all M target vertex's values in est_X_b
ps[*vpt*] = IT_score_computation(outlier_scorer, est_X_b,, oscore)
9 **end parallel for**
10 ### compute the contribution of each vertex of causing an outlier value at the target vertex at a particular time based on the result stored in the participation_scores HashMap ###
vertices_contributions = compute_shapley_value(participation_scores, sorted_A(X_b)))
11 **return** vertices_contributions

The next step is generating a set of tuples that we call the "vertex participation set" (Algorithm 4.8 Line 3). Each element in this set is a tuple with a length of the size of sorted_A(X_b) (|A(X_b)|), where the value of each tuple element (vertex) is either 0 or 1. The value of 0 of a vertex represent a non-participation vertex meaning the vertex's noise term values will be randomized. The value of 1 of a vertex indicates the vertex as a participation vertex meaning its noise term values will not be randomized. The randomization of the noise term values of a vertex is related to the counterfactual question we want to find out *"If we assigned rather a normal mechanism at a vertex X_a by randomizing the noise term NT_a, would o have not been an outlier?"*. In other words, if the noise term values of a vertex X_a is randomized, would the outlier value of the target vertex at a particular timestamp remain as an outlier? If it remains as an outlier, then X_a is not the root cause factor of the outlier.

Taking Figure 4-6 as a reference, with the target vertex set as X_4 and sorted_A(X_b) as {X_1, X_3, X_4}, the maximum number of items in the vertex participation set is 2^3, which can be enumerated as follows: {<0, 0, 0>, <0, 0, 1>, <0, 1, 0>, <0, 1, 1>, <1, 0, 0>, <1, 0, 1>, <1, 1, 0>, <1, 1, 1>}. In this representation, 0 signifies the non-participation of a vertex, while 1 indicates its participation. For instance, <1, 0, 0> implies that X_1 is a participation vertex while X_3 and X_4 are not. Thus, the noise term values of X_3 and X_4 are going to be randomized while X_1 is not.

Following the generation of the vertex participation set, the next step is to calculate the IT Score of M estimated values of the target vertex of each item in the vertex participation set (Algorithm 4.8, Lines 4-9). The IT Scores computation is done independently for each item in the vertex participation set. Firstly, the noise term values,

NT, are shuffled based on the values of *vpt*, the vertex participation tuple (Algorithm 4.8, Line 6). If the value of a vertex in the *vpt* is 1, it indicates that the corresponding vertex values in **NT** are substituted with the values from the outlier noise. As illustrated in Figure 4-8, when a vertex participation tuple = <1, 0, 0>, it means all values of X_1 in **NT** are replaced with the value of X_1 from the outlier noise while the values of X_2 and X_3 are jointly shuffled. Using the shuffled **NT**, the corresponding est_X_b, the M estimated values of the target vertex, are then computed (Algorithm 4.8, Line 7). The computation is the same as the process of generating $\hat{\mathbf{X}}$[target vertex] in Algorithm 4.4, Lines 3-14.

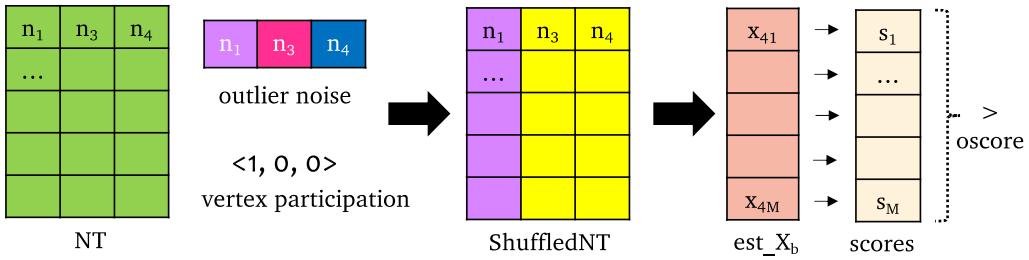


Figure 4-8 Calculating a vertex participation score

The est_X_b is then used as an input to the IT score's function which computes the participation score of the particular *vpt* (Algorithm 4.8, Line 8). The IT Scores function starts by computing the score of every value in the est_X_b using the fitted outlier scoring function (outlier_scorer). In total there are M outlier scores of M estimated target vertex's values. Then, the function calculates *NHigherOutlierScore* which is the total number of the outlier scores of all estimated target vertex values in est_X_b that is higher the outlier score of the estimated outlier value of the target vertex (oscore) generated in Line 2. The

participation score of the particular *vpt* also known as the IT Score of the *vpt* is finally computed by dividing NHigherOutlierScore by M.

Once we have the participation scores of all members of the vertex participation set, it is time to calculate the contribution of each vertex in the sorted $A(X_b)$ of causing the outlier at the target vertex X_b (Algorithm 4.8, Line 9). The contribution of a vertex X_a is measured by taking the average of all the differences between the participation scores (*ps*) when X_a is participating and when it is not. For example, the contribution of X_1 from Figure 4-6 is computed by taking an average of: {(*ps*[<1, 0, 0>] - ps[<0, 0, 0>]) ; (*ps*[<1, 1, 0>]- *ps*[<0, 1, 0>]) ; (*ps*[<1, 0, 1>]- ps[<0, 0, 1>]) ; and (*ps*[<1, 1, 1>] -[<0, 1, 1>])}. The vertices are then sorted in descending order based on their contributions (Algorithm 4.8, Line 10). The vertex with the highest contribution score of causing the outlier at the target vertex is deemed as the main root cause factor of the outlier. However, depending on the specific needs or context of the application, the analyst has the flexibility to consider not just the single highest-contributing vertex, but the top *k* vertices with the highest contribution scores as the root cause factors of the outlier.

4.4.3.3 Concept Drift Checker and AFDS Updater (The Online Phase)

As shown in Figure 4-5 and Lines 5 and 9 in Algorithm 4.5, while explaining outliers online, Ocular is also running a process for checking whether the underlying data distribution in the new sliding window has changed. We are now describing how the process runs (Algorithm 4.9).

Algorithm 4.9: **concept_drift_checker_and_active_fcm_ds_updates**(**aligned_inliers**, causal_graph, X_b, sorted_A(X_b), M, g, ActiveFCM, Models, NTSamples, Scorers, TargetRMSEs, start_ts, δT , γ max_ID)

Input: a set of aligned tuples of normal data values at the target vertex and its ancestor vertices from the current window (**aligned_inliers**), the user given normal causal graph (causal_graph), target vertex X_b , the list of X_b's ancestors and X_b itself ordered based on the topological order from the roof vertices to X_b (sorted_A(X_b)), the number of noise term samples to generate (M), outlier scoring function (g), four facets of AFDS (ActiveFCM, Models, NTSamples, Scorers), , the key value set in which a key represents an ID of an FCM, and the value represents the root mean squared error (RMSE) of prediction function at the target vertex X_b (TargetRMSEs), start time of the current sliding window (start_ts), slide size with a time unit (δT) , target vertex's error prediction threshold γ, the maximum number the AFDS can keep (max_ID)

Output: The updated four facets of Active FCM related Data Structure (AFDS): (1) ActiveFCM, (2) Models, (3) NTSamples, and (4) Scorers, and TargetRMSEs.

- ActiveFCM is a key-value set where a key is a functional causal model (FCM)'s ID and a value is a FCM's start and end time stamp.
- Models is a key-value set where a key is a FCM's ID and a value is a FCM.
- NTSamples is a key-value set where a key is a FCM's ID and a value represents a group M NoiseTerm values of all variables or vertices in the causal graph.
- Scorers is a key-value set where a key is a FCM's ID and a value represents a fitted outlier scoring function.
- TargetRMSE is a key-value set where a key is a FCM's ID and a value represents the root mean square error of $f_{X_b}(P(X_b), NT_b)$ that represents X_b as a function of its parent vertices $P(X_b)$ and its noise term NT_b.

```
1  ## get the latest ID of AFDS
   latest_ID = max(ActiveFCM.IDs)
2  ## get FCM of the latest ID of AFDS
   fcm = Models[latest_id]
3  ## get target vertex's RMSE of the latest ID of AFDS
   target_rmse = TargetRMSEs[latest_ID]
4  ## get the parent of the target vertex
   target_parents = causal_graph.parents(X_b)
5  ## predict the values of the target vertex based on its parents'
   values ##
   predicted_target_val = fcm[X_b].predict(aligned_inliers[target_parents])
6  ## compute RMSE between the predicted values of the target vertex
   and its actual values ##
   new_rmse = rmse_compute (aligned_inliers [target_parents]),
                                    predicted_target_val)
7  ## compute the change of the new RMSE and the RMSE of the lastest
   ID of AFDS
   change = abs((new_rmse - target_rmse)/target_rmse)
```

```
8  ## no concept drift if the change is smaller than the given
   threshold ##
   if change <= γ # no concept drift
9    ## when no concept drift detected just update the end timestamp
     of the latest AFDS's ID ##
     ActiveFCM[latest_ID].end_date = start_ts + δT
10 end if
11 else
12   ## concept drift is detected, need to get a new ID for the AFDS
     new_ID = latest_ID + 1
13   ## add the new ID into the AFDS along with its start and end
     timestamp ##
     ActiveFCM[new_ID] = {start_time : start_ts, end_time: start_ts + δT }
14   ## fit or train a new FCM using the aligned_outliers data using
     algorithm 4.3
     Models[new_ID] = fcm _fitting(aligned_inliers, causal_graph, target_vertex,
     sorted_A(X_b))
15   ## generate M Noise Term Samples using Algorithm 4.4
     NTSamples[new_ID], X̂= generate_ noise_term_samples (Models[new_ID],
     causal_graph, sorted_A(X_b), M)
16   Scorers[new_ID] = g.fit(X̂[X_b])
17   TargetRMSE[new_ID] = new_rmse
18 end else
19 if latest_ID > max_ID:
20   remove the item from active FCM related data structure that corresponds to the
     oldest ID
21 end if
22 return ActiveFCM, Models, NTSamples, Scorers, TargetRMSEs
```

The concept drift checker & AFDS updater process runs in parallel with the outliers' root cause factors generator. It first gets the latest ID from the *Active FCM* and then uses the latest ID to get the corresponding FCM and target vertex's root mean squared error (RMSE) (Algorithm 4.9 Lines 1-3). Recall that the normal dataset is from the most recent window. Intuitively, when an anomalous event or object takes place, its explanation explained by the recent events that occurred before the anomaly. For example, in the urban setting context, a carnival party in an adjacent area may cause crowds to flow toward the anomalous region, or a serious accident nearby may cause vehicles to avoid an area.

Moreover, the current window and the previous window only have one slide difference. Thus, using the dataset from the two most recent slides in the current window to check whether there is a change in data distribution is sufficient. For instance, the period of the current window in Figure 4-6 is between 1:15 pm to 2:15 pm with a sliding size equal to 15 minutes. The dataset used to check whether there is a concept drift happens at 2:15pm is the dataset from the slides 2-2:15 pm and 1:45-2pm. The process computes the X_b's predicted value by using the data corresponding to X_b's parents as the features or independent variables in the fcm's prediction model. It then finds the new RMSE between the target vertex' predicted values and the actual values (Algorithm 4.9 Lines 4-6). If the relative change between the new and previous RMSE values is less than or equal to the error threshold γ, then there is no concept drift. In this case, Ocular does not need to train a new FCM since the latest FCM has been trained with the same data distribution. Thus, it only needs to update the end date of the latest ID to ensure the latest FCM will be used to explain the outliers in this current Slide (Algorithm 4.9 Lines 7-10). Otherwise, a concept drift occurs, and thus the algorithm needs to train a new FCM since data distribution has changed. The algorithm creates a new ID for the new FCM, generates noise term samples and fits a new outlier scorer. It also adds the new RMSE to the *TargetRMSE* data structure (Algorithms 4.9 Lines 11-18). Recall that TargetRMSEs is a key value set in which a key represents an ID of an FCM, and the key's corresponding value represents the root mean squared error (RMSE) of prediction function at the target vertex X_b.

Finally, the process also checks whether it needs to remove the oldest ID and its corresponding information from the AFDS. This is important because data streams arrive

continuously, and it is impossible to keep all data structures in memory. By default, we set the maximum number of IDs to $(\Delta T / \delta T) + 1$ where ΔT and δT are the window size and the slide size with time units, respectively (Algorithms 4.9 Lines 19-21).

4.5 Ocular Complexity Analysis

In this section, we discuss the time and space complexity of Ocular. The analysis is based on the following aspects:

- |V|, the number of vertices in the causal graph (G). For simplicity, we assume that the number of vertices in $A(X_b)$, the list of a target vertex (X_b)'s ancestors and X_b itself, is the same as |V|
- |E|, the number of edges in in the causal graph
- $|P(X_i)|$, the number of parents of a vertex X_i
- nroots, the number of root vertices
- M, the number of noise term samples
- max_id, the maximum number of AFDS' IDs to keep in the memory
- n_o, the number of outliers obtained from the outlier queue when the window slides
- |W|, the number of data points in the sliding window. We assume that the number of data points required to initialize the functional causal model (FCM) is equal to |W|.
- |slide|, the number of data points in a Slide
- Iter, the number of iterations to train a model
- NDistTypes, the number of possible data distribution types representing a noise term of a vertex

- kneighbor, the number of neighbors used in KL divergence computation

4.5.1 Time Complexity of Ocular

We discuss the time complexity of Ocular for the Offline Phase and Online Phase.

4.5.1.1 Time Complexity of the Offline Phase

In the Offline Phase that is used to initialize Active FCM related Data Structures (FCM), Ocular first calls the *get_topological_order_of_Xb_ancestors_and_itself* function (Algorithm 4.1 Line 1). This function requires O(|V| + |E|) time to identify $A(X_b)$, all ancestors of the target vertex X_b using depth-first search (DFS) [127]. It also use DFS to create a topological list that consist of all the nodes in the causal graph from the root vertices to the leaf vertices in O(|V| + E). Finally, to get sorted_$A(X_b)$, Ocular iterates over each element in the topological list and compare it with the element in $A(X_b)$. This iteration takes O(|V|) time. Thus, taking the dominant terms, the overall time complexity of the function in Algorithm 4.1, Line 1 is **O(|V| + |E|)**.

In Algorithm 4.1, Line 2, Ocular calls the function AFDS_initialization described in Algorithm 4.2. In this function, Ocular first processes the values of each vertex according to the causal graph (Algorithm 4.2 Line 1). Assuming the number of initial data representing each vertex is equal to the typical number of data points in the sliding window (|W|) and all vertices in G belong to the sorted_list $A(X_b)$, the process of shifting |W| values of a vertex according to the vertex's distance from the target vertex takes O(|W|) time. In total, processing the time series data of all vertices into a set of tuples (data frame) takes **O(|V| |W|)** time (Algorithm 4.2 Line 1).

Initializing an empty key value set for Active FCM involves O(1) time complexity (Algorithm 4.2 Line 3). For initializing *Models* (Algorithm 4.2 Line 4), Ocular calls the fcm_fitting function (Algorithm 4.3). This function initializes an empty key value set in O(1) time (Algorithm 4.3 Line 1). It then iterates over each vertex in the sorted_A(X_b). Recall that a vertex X_i has $|P(X_i)|$ number of parents; thus, in each iteration, the function takes $O(|P(X_i)|)$ time to get a parent of a vertex X_i (Algorithm 4.3 Line 3). If a vertex does not have a parent, it means the vertex is a root. Subsequently, the if condition on checking whether a vertex is root on Algorithm 4.3 Line 4 takes O(1) time. If a vertex is a root, its distribution type which is also its noise term's distribution type is determined using a stochastic model that allows to sample from any parametric distributions implemented in Scipy [125]. The stochastic model finds the suitable distribution from NDistTypes possible distribution types. The time complexity of fitting a distribution into |W| vertex's values is generally O(|W|) for simple distributions like normal or exponential and higher for more complex distributions like gamma and beta, potentially up to $O(|W|^2)$. Once, a data distribution is fitted, |W| samples are generated from the distribution which takes O(|W|). The generated |W| samples and original |W| root vertex's values are then used to estimate KL divergence using the k-nearest neighbors [126]. The estimation of KL divergence takes O(2|W|log|W| + knearest |W|) [3] and it is used to find the best fitting parametric distribution of a given set of samples. Overall, finding the most suitable data distribution of the root vertex in Algorithm 4.3, Line 5 takes O(NDistTypes ($|W|^2$ + |W| + 2|W|log|W| + knearest |W|)). Since $|W|^2$ > knearest |W| > 2|W|log|W| > |W|, this time complexity can be simplified as O(NDistTypes $|W|^2$).

For each child vertex, getting **predictors** of size |W| by |P(X_i)| takes O(|W||P(X_i)|) time (Algorithm 4.3 Line 8) and getting **outcomes** of size |W| requires O(|W|) time (Algorithm 4.3 Line 8). Both the **predictors** and **outcomes** data are used to train the prediction model of X_i ($f_{X_i}(P(X_i))$). Assuming that $f_{X_i}(P(X_i))$ is trained using a gradient descent algorithm, then the training process requires O(Iter|W||P(X_i)|) time [128] (Algorithm 4.3 Line 9). After training the prediction model of X_i, Ocular estimates the values of X_i based on its parent's values store in **predictors**. The estimation process applies each row in predictors as an input to $f_{X_i}(P(X_i))$. The total time to get |W| values in **estimated_outcomes** is O(|W||P(X_i)|) (Algorithm 4.3 Line 9). Ocular then finds the noise distribution type of the child vertex X_i by first finding the difference between |W| **estimated_outcomes**'s values and |W| **predictors**'values in O(|W|) time (Algorithm 4.3 Line 11). The |W| differences are then used to find the X_i's noise term distribution (Algorithm 4.3 Line 12). The process of finding the most suitable noise term distribution of a child vertex X_i is the same as of a root vertex described in the previous paragraph, which takes O(NDistTypes $|W|^2$) time.

Finally, the overall time complexity of Algorithm 4.3, fcm_fitting, which is employed in Algorithm 4.2, Line 4 is : O(|V|(1 + |P(X_i)| + 1) + nroots (NDistTypes $|W|^2$) + (|V|-nroots)(|W||P(X_i)| + |W| + Iter |W||P(X_i)| + |W||P(X_i)| + |W| + NDistTypes $|W|^2$)). Considering |W| > |V| > |P(X_i)| and the number of child vertices (|V|-nroots) > the number of root vertices (nroots), we can conclude fitting the prediction model and noise term of child vertices as the dominant terms. Hence, time complexity Algorithm 4.3, fcm_fitting is **O((|V|-nroots)(Iter |W||P(X_i)| + NDistTypes $|W|^2$))**.

To initialize *NTSamples* (Algorithm 4.2 Line 5 and Algorithm 4.4), Ocular needs O(|V|) to initialize an empty dataframe **NT** and O(|V|) to initialize and empty data frame

$\widehat{\mathbf{X}}$ (Algorithm 4.4 Lines 1-2). It then processes each vertex in sorted_A(X_b) and takes O(|P(X_i)|) to get the parents of a vertex X_i and O(1) to decide whether X_i is a root or child vertex (Algorithm 4.4 Lines 4-5). If X_i is a root, drawing M random samples for **NT**[X_i] and M random samples for $\widehat{\mathbf{X}}$[X_i] requires O(M) and O(M) respectively (Algorithm 4.4 Lines 6-7). If X_i is a child vertex, the algorithm takes O(M|P(X_i)|) to get the value of $\widehat{\mathbf{X}}$[parents], O(M) to set M values of the noise term **NT**[X_i] and O(M(|P(X_i)|+1)) to predict the value of $\widehat{\mathbf{X}}$[X_i] (Algorithm 4.4 Lines 10-12). Overall, the time complexity for initializing *NTSamples* is O(|V| + |V| + |V||P(X_i)| + |V| + nroots(2M) + (|V|-nroots)(M|P(X_i)| + M + M(|P(X_i)| + 1))) **or** O(3|V| + |V||P(X_i)| + 2 nroots M + (|V|-nroots)(M|P(X_i)| + M + M(|P(X_i)| + 1))). Considering |M| > |V| > |P(X_i)| and the number of child vertices (|V|-nroots) > the number of root vertices (nroots), the dominant terms of the time complexity of for Algorithm 4.4 is O((|V|-nroots)(M|P(X_i)| + M + M(|P(X_i)| + 1)). This time complexity of Algorithm 4.4 which is called in Algorithm 4.2, Line 5 can be further simplified to **O((|V|-nroots) M |P(X_i)|)**.

The time complexity for initializing *Scorers* (Algorithm 4.2 Line 6) depends on the type of the outlier scoring function g. In our implementation we apply Median CDF Quantile Scorer [129] as g. This outlier scoring function is a linear function that requires **O(M)** time to store M predicted target vertex's values.

In order to initialize TargetRMSE of ID 1 (Algorithm 4.2 Line 6), Ocular compute the root mean squared error (RMSE) between |W| actual target vertex's values and |W| predicted values of the prediction model at the target vertex. This computation involves O(|W|) time to compute squared errors of each pair of actual and predicted values, O(|W|) time to sum up all squared errors and dividing the sum by |W| to get mean squared error

(MSE), and finally O(1) time to take the squared root of the MSE to get RMSE. Therefore, the overall time complexity of computing RMSE is **O(|W|)**.

In total, Offline Phase takes **O(|V| + |E| + |V| |W| + 1 + (|V|-nroots)(Iter |W||P(X_i)| + NDistTypes $|W|^2$) + (|V|-nroots) M |P(X_i)| + M + |W|)** time. Taking the dominant terms, we can simplify the time complexity of Offline Phase as **O((|V|-nroots)(Iter |W||P(X_i)| + NDistTypes $|W|^2$ + M |P(X_i)|)).**

4.5.1.2 Time Complexity of the Online Phase

We now analyze the time complexity used by Ocular during the Online Phase (Algorithm 4.1 Lines 7-19). Since this phase runs indefinitely, the analysis is based on the active sliding window. Each time the window slides, Ocular gets the latest slide and drops the oldest slide from the window in O(|V||slide|) time (Algorithm 4.1 Line 8). It also obtains outliers in O(n_o) time from the outlier queue (Algorithm 4.1 Line 9).

Ocular runs the function online_ocular (Algorithm 4.1 Line 10, Algorithm 4.5). The function first processes data points from all vertices in sorted_A(X_b) including aligning outlier points in the target vertex with the corresponding ancestor vertices in O(|W| |V|). Ocular used the processed outlier data as input to (1) outlier root cause factors generation (Algorithm 4.5 Line 4) and processed normal data as input to (2) concept drift checker and AFDS updates (Algorithm 4.5 Line 5 or Line 9). We analyze the time complexity of these two parallel parts in the following sub-sections.

4.5.1.2.1 Time Complexity of Outlier Root Cause Factors Generation

The root cause factors generation of outliers are outlined in Algorithm 4.6. It starts by initializing an empty HashMap to store the output of this procedure in O(1) time

(Algorithm 4.6 Line 1). It then examines each outlier in parallel to determine its root cause factors (Algorithm 4.6 Lines 2-10).

Recall that each part of AFDS is a key value set; thus for each outlier, Ocular takes O(1) to get the timestamp of the outlier, O(1) to obtain *Active FCM*'s ID of the particular outlier, O(1) to retrieve FCM from *Models* for the corresponding ID, O(1) to get noise term samples **NT** from the *NT Samples* of the ID, and O(1) to get outlier scoring function from Scorers for the corresponding ID (Algorithms 4.6 Lines 3-7).

Ocular then runs Algorithm 4.7 to generate the outlier's noise values. Setting **outlier_noise** as a list of length |V| takes O(|V|) time (Algorithm 4.7 Line 1). The algorithm iterates over each vertex X_i in sorted_A(X_b). Recall that sorted_A(X_b) is a list of the target vertex (X_b)'s ancestors and X_b itself ordered based on the topological order from the roof vertices to X_b. In each iteration, it gets | P(X_i)| parents of X_i in O(|P(X_i)|) time and checks whether X_i is a root or child vertex in O(1) time. If X_i is a root, it sets the value of **outlier_noise**[X_i] in O(1) time. If X_i is a child vertex, it first finds the prediction value of X_i based on the values of **outlier_noise**[P(X_i)] in O(|P(X_i)|) time. It then computes the value of **outlier_noise**[X_i] in O(1). Overall, generating outlier noise takes O(|V| + |V||P(X_i)| + nroots + (|V|-nroots)(|P(X_i)|+1)). Since nroots + (|V|-nroots) = |V|, the time complexity of Algorithm 4.7 that is called in Algorithm 4.6, Line 8, can be simplified as O(|V||P(X_i)|).

Ocular computes the vertices' contribution score of the particular outlier (Algorithm 4.6 Line 9) in Algorithm 4.8. In this function, Ocular starts by generating **est_o**, a tuple of size |V| that store the estimated values of variables computed from **outlier_noise**. To initialize **est_o**, Ocular requires O(|V|) time. It then iterates over each vertex X_i in (sorted_A(X_b)). In each iteration, it gets | P(X_i)| parents of X_i in O(|P(X_i)|) time. If X_i is a

root vertex, Ocular will set the coresponding value X_i in **est_o** to be the same as the value of X_i in **outlier_noise**. This process takes O(1) time. If X_i is a child vertex, Ocular takes $O(P(X_i)|)$ time to get the corresponding values of X_i' parents in **est_o**. It then computes the estimated value of X_i by inputting the values of X_i' parents in **est_o** and the value of X_i in **outlier_noise** into the functional causal model of X_i**.** This process takes $O(1+|P(X_i)|)$ time. Overall, assigning estimated values of variables into **est_o** requires $O(|V| + |V|\ |P(X_i)| + |V| + nroots + (|V|-nroots)(1+|P(X_i)|)\ )$ time. Using the dominant terms, the complexity of generating **est_o** (Algorithm 4.8 Line 1) can be simplified as $O(|V|\ |P(X_i)|)$.

Ocular then computes the outlier score (oscore) of the corresponding target vertex's value in **est_o** (**est_o[**X_b**]**) in O(M) time (Algorithm 4.8, Line 2). Recall that when fitting the outlier scoring function g using Median CDF Quantile Scorer [129], the fitted g, stores M samples of M predicted target vertex's values. Median CDF Quantile Scorer computes oscore of **est_o[**X_b**]** by counting the stored values that are smaller and greater than **est_o[**X_b**]**. The counts are useful to find the quantile **est_o[**X_b**]** in respect of the M stored samples. If **est_o[**X_b**]** lies in the tail of the distribution, its outlier score is expected to be large. Hence, computing oscore in Algorithm 4.8, Line 2 takes O(M) time.

Subsequently, Ocular generates the vertex participation set in $O(2^{|V|})$ time (Algorithm 4.8 Line 3). It computes the participation score of each member of the set in parallel (Algorithm 4.8 Lines 5-9), where it takes O(M |V|) to shuffle **NT**, a matrix of size M by |V|**,** $O(|V|\ M\ |\ (P(X_i)|+1))$ to estimate the values of the target vertex based on the values of shuffled **NT**, and O(M) to compute the participation score of the member of the participation set using IT score. Computing the vertices' contribution score is computing the Shapley values where a vertex represents a player in the game theory (Algorithm 4.8

Lines 10). The exact computation of the Shapley values involves evaluating the worth of each coalition containing a particular player. There are |V| possible players and $2^{|V|-1}$ coalitions to consider. Hence, the time complexity of computing the Shapley values is $O(|V|2^{|V|-1})$. The overall time complexity of computing the vertices' contribution scores of an outlier in Algorithm 4.8 is $O(|V|\ |P(X_i)| + M + 2^{|V|} + M\ |V| + |V|\ M\ |\ (P(X_i)|+1) + M + |V|2^{|V|-1})$. Taking the dominant terms, the time complexity of Algorithm 4.8 can be simplified as $O(|V|2^{|V-1|})$ time.

Summing up, the overall time complexity of Algorithm 4.6 is $O(1 + 1 + 1 + 1 + 1 + |V||P(X_i)| + |V|2^{|V|-1})$. Using the dominant terms, the time complexity of the Outlier's Root Cause Factor Generation process is $\mathbf{O(|V|2^{|V|-1})}$. We do not multiply the time complexity with n_o, the number of outliers obtained from the outlier queue when the window slides, because the root cause generation of each outlier runs in parallel.

4.5.1.2.2 Time Complexity of Concept Drift Checker & AFDS Updater

The Concept Drift Checker & AFDS updater component described in Algorithm 4.9, starts by getting the most recent ID from *ActiveFCM* which is a key value set data structure in O(1) time, getting the FCM for the corresponding ID in O(1) time, and getting the target vertex's RMSE is O(1) time (Algorithm 4.9 Lines 1-3). Ocular obtains $|P(X_b)|$ number of parents of the target vertex X_b in $O(|P(X_b)|)$ (Algorithms 4.9, Line 4). It uses the latest two Slides of the current window to predicts 2|slide| values of X_b using 2|slide| values of X_b' parents applied on the latest FCM. This prediction takes $O(2|slide||P(X_b)|)$ time (Algorithms 4.9, Line 5).

Subsequently, Ocular computes the new RMSE between 2|slide| predicted X_b values and 2|slide| actual X_b values in O(2|slide|) time (Algorithms 4.9, Line 6). It then computes

the absolute difference between the new RMSE and the RMSE obtained by the most recent FCM model in O(1) time (Algorithm 4.9 Line 7). Checking whether concept drift has happened based on the change of the RMSE takes O(1) time (Algorithm 4.9 Line 8). If no concept drift, it only requires O(1) time to update the end timestamp of the latest ID in *Active FCM* (Algorithm 4.9 Line 9).

If concept drift occurs, Ocular sets a new ID in O(1) time, adds the new ID as a new key, and sets its corresponding value: start and end time in O(1) time (Algorithm 4.9 Lines 12-13). A new causal model is trained (Algorithm 4.9 Line 15) using Algorithm 4.3 and added to the *Models* in $O((|V|\text{-nroots})(\text{Iter } |2\text{Slide}||P(X_i)| + \text{NDistTypes } |2\text{Slide}|^2))$. Please refer to our explanation in Section 4.1.1.1 on how the time complexity of Algorithm 4.3 is derived.

Generating the noise term samples for the corresponding new ID in *NTSamples* (Algorithm 4.9 Line 15) is done using Algorithm 4.4 and takes $O((|V|\text{-nroots}) M |P(X_i)|)$. Please refer to our explanation in Section 4.1.1.1 on how the time complexity of Algorithm 4.4 is derived. Furthermore, as described in Offline Phase, training the outlier scoring function for the new ID and saving the new RMSE take O(M) and O(M), respectively (Algorithm 4.9 Lines 16-17). Removing the oldest ID when the number of IDs has reached the maximum takes O(1) for each data structure in AFDS.

Overall, when there is no concept drift, the time complexity of the Concept Drift Checker and AFDS Updater component is $O(1 + 1 + 1 + |P(X_b)| + 2|\text{slide}||P(X_b)| + 2|\text{slide}| + 1 + 1 + 1)$. Taking the dominant terms, the time complexity of Algorithm 4.9 without concept drift is $\mathbf{O(2|slide||P(X_b)|)}$.

However, when there is concept drift, the time complexity becomes $O(1 + 1 + 1 + |P(X_b)| + 2|slide||P(X_b)| + 2|slide| + 1 + 1 + 1 + 1 + (|V|\text{-}nroots)(Iter\ |2Slide||P(X_i)| + NDistTypes\ |2Slide|^2) + (|V|\text{-}nroots)\ M\ |P(X_i)|) + M + M + 1)$. Taking the dominant terms, the time complexity of Algorithm 4.9 with concept drift is **$O((|V|\text{-}nroots)(Iter\ 2|Slide||P(X_i)| + NDistTypes\ (2|Slide|)^2 + M\ |P(X_i)|))$.**

4.5.1.2.3 The Overall Time Complexity for the Online Phase

In summary, the Root Cause Generator component concurrently identifies the root cause factors of each outlier with a time complexity of $O(|V|2^{|V|})$. The time complexity of the Concept Drift Checker & AFDS Updater component is expressed as $O(2|slide||P(X_b)|)$ when no concept drift or $O((|V|\text{-}nroots)(Iter\ 2|Slide||P(X_i)| + NDistTypes\ (2|Slide|)^2 + M\ |P(X_i)|))$ otherwise. The time complexity of the Online Phase varies based on whether outliers are detected before a new Slide arrives. In the absence of detected outliers, it relies on the Concept Drift Checker & AFDS Updater component. Conversely, if outliers are identified, the time complexity becomes $O(|V|2^{|V|})$.

4.5.2 Space Complexity of Ocular

Ocular requires the causal graph information during the offline and Online Phases. The causal graph represented as the adjacency list, takes $O(|V| + |E|)$ space where $|V|$ is the number of vertices and $|E|$ is the number of directed edges in the causal graph. In addition, the sorted list of the vertices starting from the root vertices to the target vertex, sorted_A(X_b), uses $O(|V|)$ memory space.

During the Online Phase, Ocular needs to read data from the sliding window. The sliding window buffer requires $O(|W|\ |V|)$ space to store $|W|$ values of $|V|$ vertices

(Algorithm 4.1 Line 8). Ocular needs $O(2n_o)$ space for obtaining n_o number of anomalous values of the target vertex and their n_o number corresponding timestamps from the queue (Algorithm 4.1 Line 9). To keep the aligned outlier data and normal data of all vertices, Ocular uses $O(n_o |V|)$ space and $O(|W| |V|)$ space respectively (Algorithm 4.5 Line 1).

Ocular initializes AFDS during Offline Phase and keeps it in the memory throughout the online process. Recall that the maximum number of IDs AFDS can accommodate is denoted as max_id, The space complexity of the AFDS is computed as follows:

- O(max_id) space to keep track of *ActiveFCM*'s IDs and their corresponding start time and end time.
- In *Models*, each ID points to a set of FCM where each vertex X_i keeps the weight of its parent vertices, $P(X_i)$, and its noise term N_i in $O(|P(X_i)| + 1)$ space. Since there are total |V| number of vertices, each corresponding ID in *Models* requires $O(|V| (|P(X_i)| + 1))$ space. Overall, *Models* requires $O(\text{max\_id } |V| (|P(X_i)| + 1))$ space.
- In *NTSamples*, each ID represents M samples of noise term values, requiring O(M |V|) space. Overall *NTSamples* requires O(max_id M |V|) space.
- In *Scorers*, each ID points to an outlier scoring function. The space complexity depends on the number of hyperparameters of the outlier scoring function. For example, the Median CDF Quantile Scorer requires keeping O(M) samples. In total *Scorers* uses O(max_id M) space.
- In addition, Ocular takes O(max_id) space to keep the target RMSE.

During the root cause generation (Algorithm 4.6), for each outlier, Ocular requires O(|V|) space to keep the outlier noise (Algorithm 4.7) and $O(2^{|V|})$ space to compute the

vertices' contribution score (Algorithm 4.8). Thus, for all outliers obtained from the queue when a window slides, Ocular needs $O(n_o\,(|V|+2^{|V|}))$ space.

In parallel with the root cause generation process, Ocular runs the concept drift detector and AFDS updater (Algorithm 4.9). This process requires O(2|slide|) space to store the predicted value of the latest two Slides data, O(1) for its RMSE, and O(1) for the change percentage between old and new RMSE. If there is a concept drift (Algorithm 4.9 Lines 11-18), not only that Ocular need to update the AFDS, but it also takes O(M |V|) space to estimate the values of all vertices denoted as $\hat{\mathbf{X}}$ (Algorithm 4.9 Line 15) and O(M) space to update the outlier scoring function (Algorithm 4.9 Line 16).

Overall, Ocular requires $O(|V| + |E| + |V| + |W|\,|V + 2n_o + n_o\,|V| + |W|\,|V| + max\_id + max\_id\,|V|\,(|P(X_i)| + 1) + max\_id\,M\,|V| + max\_id\,M + max\_id + n_o(|V|+2^{|V|}) + 2|slide| + 1 + 1 + M\,|V| + M)$ memory space. The dominant term is $O(n_o\,(|V|+2^{|V|}))$ which is required when computing the vertex contribution scores of outliers obtained from the outlier queue when a Slide arrives.

4.6 Performance Evaluation

This section discusses our experimental setup, results, and analysis of running Ocular and existing algorithms to compare their performance on finding outlier root cause factors using synthetic and real-world datasets.

4.6.1 Experimental Setup

The experimental setup consists of (i) competing algorithms, (ii) performance metrics, and (iii) software and hardware.

4.6.1.1 Competing Algorithms

We compare Ocular with four existing algorithms, CausalRCA [87], EasyRCA [88], CloudRanger [13], and DyCause [46]. CausalRCA is designed for static data, hence it does not consider the charateristics of data streams. EasyRCA [88] works for timestamped datasets but is not designed for real-time data streams. Since both CausalRCA and EasyRCA do not consider the unbounded volume of continuous data streams, we run them as batch processing over the sliding windows. CloudRanger and DyCause are proposed to find root cause factors of outliers in data stream applications. These two algorithms are applied in microservices monitoring for cloud systems where frontend services relate to other backend services. An outlier which is an unusual latency is observed in a frontend service during a time period or window.

EasyRCA, CausalRCA, and Ocular assume that the causal graph that represents the causality dependency between variables in the system is given. CloudRanger and DyCause require a causal graph formed using a data-driven causal discovery technique such as the PC algorithm [89] or an improved version of the PC algorithm for time series [28]. Although CloudRanger, EasyRCA, and DyCause are designed to find the root cause factors of collective outliers, we can view a point outlier as a collective outlier of one data point. We present the property summary of these competing algorithms and Ocular in Table 4-1.

4.6.1.2 Software and Hardware

We implement Ocular and a data stream simulator using Python 3.8. Our source code is available at [130] . The source code for CausalRCA is available in the PyWhy Python package [129] , while the Python implementation of EasyRCA and CausalRanger are

available at [131]. DyCause implementation is available at [132]. We incorporate these algorithms into our data stream simulator. We run our experiments on a MacBook Pro: macOS Catalina, Processor 2.7 GHz Quad-Core Intel Core i7, Memory 16 GB 1600 MHz DDR3.

4.6.1.3 Evaluation Metrics

We measure the effectiveness of the algorithms using their F1-Score and time efficiency. Given that we are dealing with continuous arrivals of data, we need to ensure that the algorithms can keep up with the incoming data in a timely manner. Therefore, we also measure the average time needed by the algorithms to explain or generate the root cause factors for an outlier.

Table 4-1 The summary of the properties of the outlier root cause factors algorithms

Name	Data Type	Outlier Type	Causal Graph	DS 1	DS 2	DS 4	RC 1	RC 2
CloudRanger Wang et al. 2018 [13]	data stream	collective outlier	causal graph constructed from log data	Yes	Yes	Yes	No	No
CausalRCA Budhatoki et al. 2022 [87]	static data	point outlier	user-defined	No	No	No	Yes	Yes
EasyRCA Assaad et al. 2023 [88]	static timeseries data	collective outlier	user-defined	Yes	No	No	No	Yes
DyCause Pan et al. 2023 [46]	data stream	collective outlier	causal graph constructed from log data	Yes	No	No	No	No
Ocular (ours)	data stream	point outlier	user-defined	Yes	Yes	Yes	Yes	Yes

DS 1: notion of time
DS 2: notion of infinity
DS 4: concept drift
RC 1: providing counterfactual
RC 2: incorporating user's knowledge about variables' dependency into the algorithm

4.6.2 Performance Results on Synthetic Datasets

Although there are benchmark datasets for point outlier detection, as far as we know, real-world datasets that include ground truth for the root causes of point outliers are not publicly accessible. Hence, we generate synthetic datasets to perform some arrays of experiments. Each synthetic dataset is generated according to the summary causal graph. We vary the number of vertices in each causal graph and randomly select different types of data distribution for the noise term in each vertex. We set a leaf vertex as a target vertex and randomly select any vertices that have a path to the target vertex as the root cause factors of each outlier. In each synthetic dataset, the percentage of the outliers generated ranges from 1% to 3% of the entire dataset. Our scripts for synthetic data generation are available at [130]. We simulate each dataset as a data stream by running a Python based TCP Client Server on our machine. Data points are continuously sent from the client to the server based on the given arrival rate. The server side handles the sliding window process. The simulation ends when no more data points available at the client.

Table 4-2 Dynamic parameters

Dynamic Parameters	Value Range	Default Value
Number of vertices	3 - 12	6
Window size	2 - 16 minutes	8 minutes
Slide size	1 - 4 minutes	2 minutes
Data arrival rate	0.1 - 1 second	0.5 second

Parameters. We study the impact of four dynamic parameters, which are the number of vertices in the causal graph, window size, slide size, and data point arrival rate, on the performance of the algorithms. As shown in Table 4-2, when we conduct experiments to

study the impact of a dynamic parameter, we vary its values within its value range, and we keep other dynamic parameters at their default values.

In addition to the dynamic parameters, we also keep the static parameters that remain unchanged throughout all our experiments. We set M, the number of samples in the set of noise term values to be 1500 and γ, RMSE change threshold to be 0.1 We follow the Shapley Config setup from CausalRCA implementation [129], such that when the number of vertices is six or fewer, the algorithms will generate all possible tuples ($2^{|V|}$) in the vertex participation sets, with "|V|" being the number of vertices in the set of the target vertex and its ancestors. Conversely, if the number of vertices exceeds six, the algorithm employs early stopping, avoiding the need to examine all the tuples.

We now present our experimental results.

4.6.2.1 Experiment Result based on the Dynamic Parameter's Default Values

Table 4-3 The algorithms' performance on the Synthetic dataset with six vertices

Algorithms	average F1 Score	average explanation time (second)
CloudRanger	0.242	0.753
DyCause	0.333	**0.011**
EasyRCA	0.398	2.050
CausalRCA	0.292	1.366
Ocular (ours)	**0.889**	0.197

Table 4-3 presents the experimental results using the default values for the dynamic parameters, where the number of vertices is six, the window size is 8 minutes, the slide size is 2 minutes, and the data arrival rate is 0.5 seconds. A leaf vertex is set as the target vertex while varying the vertices causing anomalous values at the target vertex. Ocular

achieved the highest average F1 score of 0.889, while the other algorithms' average F1 Scores were less than 0.4. Ocular's average F1 score was 267%, 167%, 123%, and 204% higher than those of CloudRanger, DyCause, EasyRCA, and CausalRCA, respectively. In this experiment, outliers were scattered across windows and identified as point outliers. CloudRanger and DyCause rely on the number of outliers with consecutive timestamps to construct anomalous causal graphs and generate root cause factors. Consequently, their accuracy may be compromised when dealing with point outliers. EasyRCA employs a user-provided normal causal graph; however, during its root cause discovery process, it relies on anomalous and normal regions. The anomalous region should consist of several outliers with consecutive timestamps, but in the case of point outliers, there is only one outlier, rendering EasyRCA's output inaccurate. Ocular is based on CausalRCA, with both applying counterfactual explanation and employing a user-provided normal causal graph. However, Ocular ensures that the root cause factors of an outlier are derived from noise term values generated from a functional causal model (FCM) fitted with data from timestamps preceding the outlier's timestamp. In contrast, CausalRCA does not consider the notion of time.

In terms of average explanation time, Ocular achieves the second fastest explanation time. Ocular is 3, 10, and 7 times faster than CloudRanger, EasyRCA, and CausalRCA, respectively. Although DyCause achieves the fastest average explanation time, its average F1 score is only 0.333, making Ocular the better option with an average F1 score of 0.889. Ocular is expected to have a faster average explanation time than CausalRCA because Ocular does not need to build a new FCM every time the window slides. Ocular initializes

an FCM during the Offline Phase and only creates a new one when concept drift is detected. In contrast, CausalRCA is designed to train a new FCM for every point outlier.

4.6.2.2 The Impact of the Number of Vertices

We vary the number of vertices in the causal graph on two types of causal graphs: (a) all other vertices in the causal graph have an influence on the target vertex, and (b) some vertices have no direct path to the target vertex. The F1 Score results of these two types are shown in Figure 4-9. Ocular and CausalRCA have the same F1 Score on Type (a) regardless of the number of vertices; however, on Type (b), Ocular has a better F1 Score than CausalRCA for the datasets with more than five vertices. Both Ocular and CausalRCA can easily find the root cause factors of an outlier value of the target vertex when the number of vertices is small. However, CausalRCA has trouble finding root causes when there are vertices in the causal graph that does not belong to the ancestors of the target vertex. Ocular ensures that the root cause discovery of an outlier value at the target vertex only involves itself and its ancestor vertices. Furthermore, unlike CausalRCA that does not consider the notion of time, Ocular discovers the root cause factors of outliers using noise terms values of their ancestor vertices, generated from FCM fitted with the data taking place before the outlier. Overall, Ocular shows the highest F1 Score in both types. DyCause's F1 Score drops when the number of vertices is larger than three and four on Type (a) and Type (b) respectively and remains the same afterward. Interestingly, DyCause achieves the perfect F1 score when the number of vertices is equal to three in Type (b). This occurs because that dataset only has the root vertex as the root cause and DyCause tends to include any root vertices that lead to the target vertex in its solution. EasyRCA's

F1 score decreases as the number of vertices increases while CloudRanger's fluctuates on both types. Overall, Ocular achieves the highest F1 Score in both types.

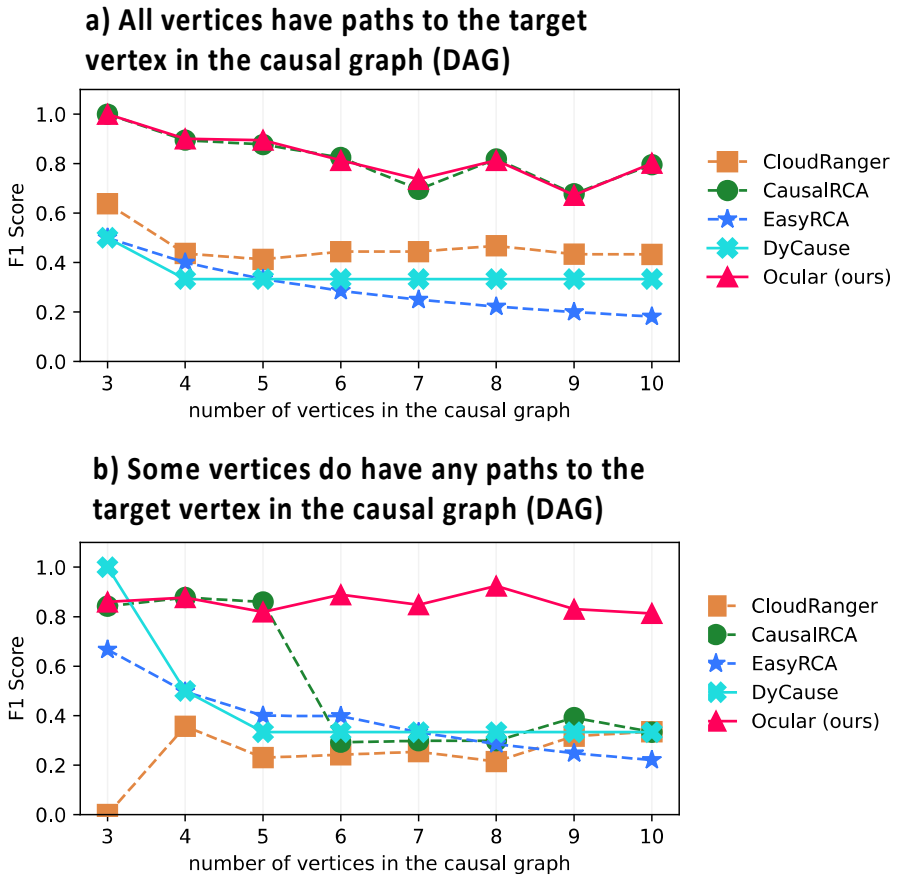


Figure 4-9 The impact of the number of vertices on F1 Score

In terms of the average explanation time (Figure 4-10), Ocular outperforms CausalRCA in both types. It is expected that in both algorithms, as the number of vertices increases, the average outlier explanation time also increases. This is because as the number of vertices increases the number of prediction models to train also increases linearly. Furthermore, both algorithms rely on computing the contribution score of each vertex based on its participation also known as the Shapley value [87, 95]. The total number of

tuples in the vertex's participation set can be exponential in terms of the number of target vertex and all its ancestors. Both Ocular and CausalRCA limit the number of tuples in the vertex participation set to examine using the techniques in [133, 134]. They only examine $2^{|V|}$ tuples when the number of vertices is fewer than or equal to six. However, the set is still growing as the number of vertices increases. Ocular has a shorter average explanation time than CausalRCA because it does not have to fit the functional causal model whenever the window slides. It relies on the concept drift detector to tell when to update the models. In comparison to CloudRanger, DyCause, and EasyRCA, Ocular exhibits a similar average explanation time for cases where the number of vertices is below 10. However, it demonstrates a slightly longer average explanation time when the number of vertices exceeds 8.

Based on the results in Figure 4-9 and Figure 4-10, we can determine that Ocular is the most preferable algorithm in terms of F1 Score and average explanation time regardless of the number of vertices. To confirm our result, we conduct a one-way ANOVA to assess whether there are significant differences between the means of the algorithms' performance. The one-way ANOVA on both F1 Scores and average explanation time yields very small p-values (0.000), indicating significant differences between the means of the algorithms' F1 Scores and average explanation time. Consequently, we proceed with a post-hoc analysis using Tukey HSD to check whether the performance means of each pair of algorithms are significantly different.

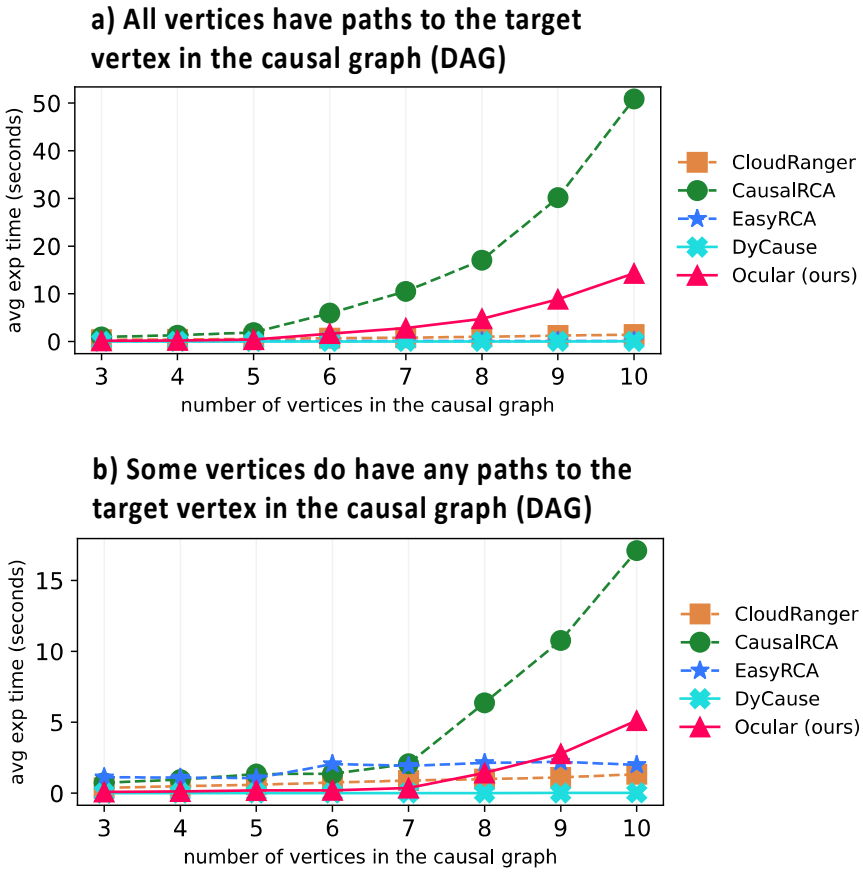


Figure 4-10 The impact of the number of vertices on average explanation time

Figure 4-11 a) shows that most pairs of algorithms reject the null hypothesis of "no significant difference between the F1 score means of the algorithm pair," except for CloudRanger vs. DyCause, CloudRanger vs. EasyRCA, and DyCause vs. EasyRCA. The pairs involving Ocular indicate that Ocular has 0.183, 0.4736, 0.4733, and 0.5203 higher F1 score means compared to CausalRCA, CloudRanger, DyCause, and EasyRCA, respectively.

In terms of average explanation time, all algorithms have similar average explanation times except for CausalRCA. In Figure 4-11 b), only the algorithm pairs involving CausalRCA reject the null hypothesis of "no significant difference between the average

explanation times of the algorithm pair." CausalRCA is the slowest algorithm when the number of vertices to examine increases. On average, it is 0.2906, 0.2903, 0.3373, and 0.183 seconds slower than CloudRanger, DyCause, EasyRCA, and Ocular respectively. Considering the FI Score and average explanation time result, we can conclude that Ocular is the best option.

group1	group2	meandiff	p-adj	lower	upper	reject
causalrca	cloudranger	-0.2906	0.0	-0.4608	-0.1204	True
causalrca	dycause	-0.2903	0.0	-0.4604	-0.1201	True
causalrca	easyrca	-0.3373	0.0	-0.5074	-0.1671	True
causalrca	ocular	0.183	0.0281	0.0128	0.3532	True
cloudranger	dycause	0.0003	1.0	-0.1699	0.1705	False
cloudranger	easyrca	-0.0467	0.943	-0.2169	0.1235	False
cloudranger	ocular	0.4736	0.0	0.3034	0.6438	True
dycause	easyrca	-0.047	0.9417	-0.2172	0.1232	False
dycause	ocular	0.4733	0.0	0.3031	0.6435	True
easyrca	ocular	0.5203	0.0	0.3501	0.6905	True

a) Tukey's HDS on the algorithms F1 scores

group1	group2	meandiff	p-adj	lower	upper	reject
causalrca	cloudranger	-9.9404	0.0	-13.6913	-6.1895	True
causalrca	dycause	-10.7752	0.0	-14.5262	-7.0243	True
causalrca	easyrca	-9.8872	0.0	-13.6381	-6.1362	True
causalrca	ocular	-7.8432	0.0	-11.5941	-4.0922	True
cloudranger	dycause	-0.8348	0.973	-4.5858	2.9161	False
cloudranger	easyrca	0.0533	1.0	-3.6977	3.8042	False
cloudranger	ocular	2.0973	0.5388	-1.6537	5.8482	False
dycause	easyrca	0.8881	0.9662	-2.8628	4.639	False
dycause	ocular	2.9321	0.2029	-0.8188	6.683	False
easyrca	ocular	2.044	0.5642	-1.7069	5.7949	False

b) Tukey's HDS on the algorithms' average explanation time

Figure 4-11 Tukey's HDS results on the means of the algorithms' performance when the number of vertices is varied

4.6.2.3 The Impact of the Sliding Window-related Parameters

We examine the impact of sliding window related parameters on the performance of the algorithms by varying the window size, slide size, and data arrival rate. The experiments are conducted on the causal graph of six vertices where two of the vertices do not have a direct path to the target vertex.

4.6.2.3.1 The Impact of the Window Size

We vary the window size between 2 to 16 minutes and set the slide size and data arrival rate to 2 minutes and 0.5 second, respectively. Figure 4-12 shows that the F1 Scores of Oculars, CausalRCA, DyCause, and EasyRCA remain constant irrespective of the changes in the window size. Ocular consistently achieves F1 Scores above 0.8 in all the test cases while other algorithms achieve the F1 Scores below 0.4. The result indicates the robustness of Ocular in identifying root cause factors of point outliers. Even though Ocular is based on CausalRCA, Ocular achieves more than double the F1 Score of CausalRCA. This result signifies that using FCM fitted with data points taking place before the outliers improves the accuracy of the outliers' root cause factors. The lower F1 Scores of DyCause, EasyRCA, and CloudRanger can be attributed to the fact that they are designed to identify root cause factors of a group of outliers that occur consecutively in time, without any normal (inlier) data points between them. These algorithms rely on the number of outliers to generate more accurate representation of the anomalous causal graph. Thus, when they deal with point outliers where each group only contains one outlier, the generated anomalous causal graph may not be accurate.

The average explanation times of Ocular, CausalRCA, DyCause, and CloudRanger exhibit no significant variations with changes in window size. In contrast, EasyRCA's

explanation time closely aligns with that of CloudRanger with a 2-minute window size but escalates with a 4-minute window size, stabilizing as the window size further increases. DyCause achieves the lowest average explanation time since it applies some optimization techniques to minimize computational overhead. Ocular follows in the second place after DyCause yet their difference is small (0.19 second difference on average). However, in terms of the F1 score, Ocular outperforms DyCause more than 100%. Hence, regardless of the window size, Ocular remains the most preferable algorithm to find root cause factors of point outliers in data streams in terms of F1 Score and average explanation time. It has the highest F1 Score and also maintains low average explanation time.

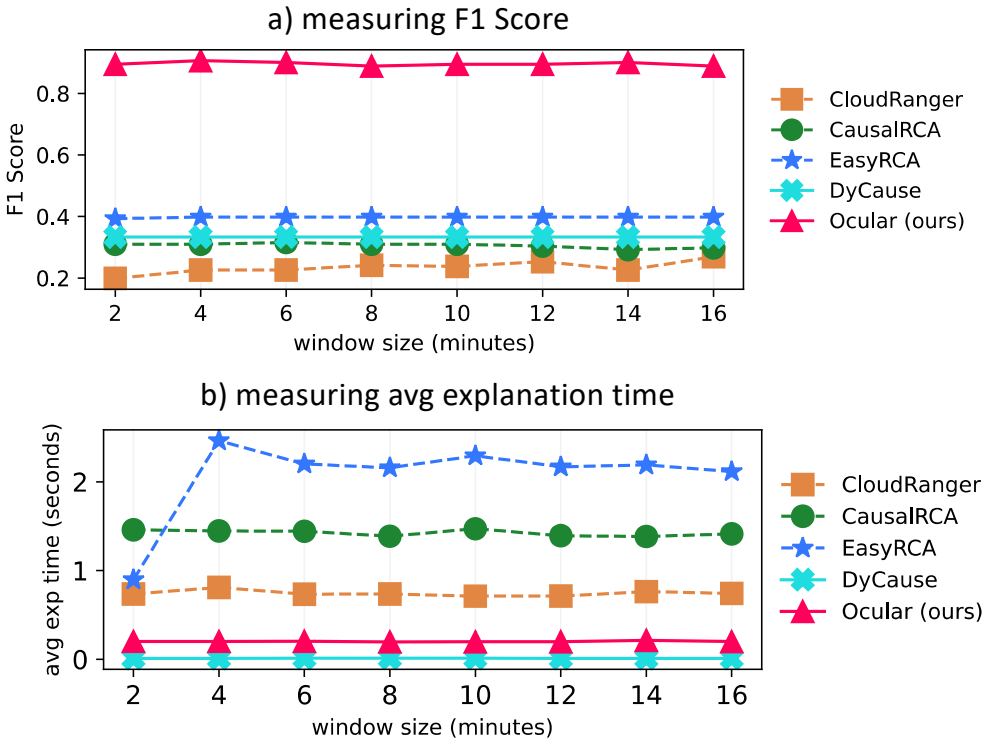


Figure 4-12 The impact of the window size

To confirm our results, we conduct statistical analysis using One-way ANOVA followed by Tukey's HDS. We run 24 experiments for each algorithm. The null hypothesis of the One-way ANOVA posits no significant difference between the means of the algorithms' performance. The One-Way ANOVA test on the average F1 score yields a remarkably small p-value (0.000) signifying that we can plausibly reject the null hypotheses. Since the means of the algorithms' F1 Scores are significantly different, we proceed to Tukey's HDS to pinpoint the significant differences between the F1 score means between each pair of the algorithms. For each pair, the null hypothesis is "the F1 score means of the pair are not significantly different".

Figure 4-13 a) shows that only in the pairs involving Ocular, the adjusted p-values are 0.0 and the confidence intervals (lower and upper values) do not include 0. Hence for each pair: CausalRCA vs Ocular, CloudRanger vs Ocular, DyCause vs Ocular, and EasyRCA vs Ocular, the null hypotheses can be plausibly rejected. Recall that the mean difference (meandiff) is computed by subtracting the mean of group1 from the mean of group2 in the Tukey HDS report (Figure 4-13). In these four pairs, Ocular is considered as group2 and their meandiff are all positive. Thus, we can conclude Ocular has significant higher F1 Score means compared to the other algorithms. This result consistent with the result in Figure 4-12.

Figure 4-13 b) shows that all algorithm pairs have sufficient evidence to reject the null hypotheses. Even though DyCause has the smallest mean average explanation time, Ocular is still preferable. All pairs have adjusted p-values of 0.0, except for DyCause vs. Ocular, which has an adjusted p-value of 0.0197. Furthermore, the smallest absolute average explanation time difference is between DyCause and Ocular. When choosing between a

0.0197-second average explanation time advantage by DyCause and a higher 0.5629 F1 score advantage by Ocular, Ocular still prevails. DyCause has very low F1 Score which is only 0.3 on average while Ocular has 0.9 F1 score on average.

a) Tukey's HDS on F1 Scores

group1	group2	meandiff	p-adj	lower	upper	reject
causalrca	cloudranger	-0.0714	0.8858	-0.2848	0.142	False
causalrca	dycause	0.027	0.9967	-0.1863	0.2404	False
causalrca	easyrca	0.0912	0.7601	-0.1222	0.3046	False
causalrca	ocular	0.5899	0.0	0.3765	0.8033	True
cloudranger	dycause	0.0984	0.7049	-0.115	0.3118	False
cloudranger	easyrca	0.1626	0.2221	-0.0508	0.376	False
cloudranger	ocular	0.6613	0.0	0.4479	0.8747	True
dycause	easyrca	0.0642	0.9197	-0.1492	0.2775	False
dycause	ocular	0.5629	0.0	0.3495	0.7762	True
easyrca	ocular	0.4987	0.0	0.2853	0.7121	True

b) Tukey's HDS on average explanation time

group1	group2	meandiff	p-adj	lower	upper	reject
causalrca	cloudranger	-0.6812	0.0	-0.8514	-0.5111	True
causalrca	dycause	-1.4137	0.0	-1.5838	-1.2435	True
causalrca	easyrca	0.6366	0.0	0.4664	0.8067	True
causalrca	ocular	-1.2228	0.0	-1.393	-1.0527	True
cloudranger	dycause	-0.7325	0.0	-0.9026	-0.5623	True
cloudranger	easyrca	1.3178	0.0	1.1476	1.4879	True
cloudranger	ocular	-0.5416	0.0	-0.7118	-0.3714	True
dycause	easyrca	2.0502	0.0	1.8801	2.2204	True
dycause	ocular	0.1909	0.0197	0.0207	0.361	True
easyrca	ocular	-1.8594	0.0	-2.0295	-1.6892	True

Figure 4-13 Tukey's HDS results on the means of the algorithms' performance when window sizes are varied

4.6.2.3.2 The Impact of the Slide Size

We vary slide sizes between 1 and 4 minutes and set the window size and data arrival rate to 8 minutes and 0.5 seconds, respectively. In terms of F1 score, Ocular consistently achieves the highest performance regardless of the slide size, followed by EasyRCA with

only 50% of Ocular's F1 Scores. The other three algorithms consistently have F1 Scores less than 0.3. In this experiment, as the slide size increases, the number of data points in the active window remains the same because the window size and data arrival rate remain constant. The result shows that the algorithms' F1 Scores are not affected by the slide size.

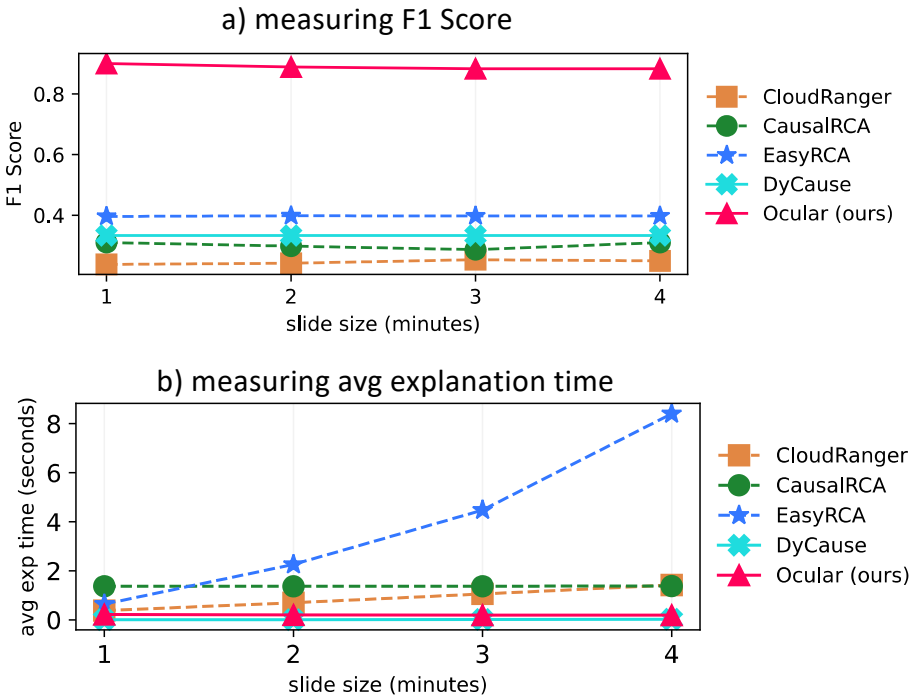


Figure 4-14 The impact of the slide size

EasyRCA shows an increasing average explanation time as the slide size increases. This result could be attributed to its need to separate the dataset into normal and anomalous regimes. When the window waits longer to slide (with a higher slide size), the normal regime, which consists of all inliers that occurred before an outlier, will contain more data points, taking longer to find the time-defying vertices of the outlier. The rest of the algorithms, including Ocular, have constant average explanation times, following similar

patterns to that in the previous section (Section 4.6.2.3.1) when the window sizes are varied.

Furthermore, we run the One-way ANOVA and Tukey HDS significant tests to confirm our results. We run 12 experiments for each algorithm. The One-way ANOVA test concludes that there is a significant difference in the algorithms' performance (rejecting the null hypothesis). Thus, we apply Tukey HDS whose results are shown in Figure 4-15.

In Figure 4-15 (part a), only the pairs of algorithms that include Ocular are found to have a statistically significant difference according to the Tukey HSD test. In other words, the null hypothesis, which typically states that there is no difference between the means of the compared groups, was rejected only for the algorithm pairs that involve Ocular, indicating that these pairs show a statistically significant difference. On average, Ocular's F1 score is higher than those of CausalRCA, CloudRanger, DyCause, and EasyRCA by 0.5877, 0.6433, 0.5556, and 0.4915, respectively. Moreover, consistent with the results in Figure 4-13, only four pairs of algorithms involving EasyRCA have sufficient evidence to reject the null hypotheses in Figure 4-14 (part b). On average, EasyRCA has significantly higher explanation time by 2.57, 3.05, 3.93, 3.74 seconds compared to CausalRCA, CloudRanger, DyCause, and Ocular, respectively. Thus, EasyRCA is the slowest algorithm. All four other algorithms (CausalRCA, CloudRanger, DyCause, and Ocular) have similar low average explanation time. Finally, based on the Tukey HDS test on F1 Score and average explanation time, it is evident that Ocular is the superior choice regardless of the slide sizes.

a) Tukey's HDS on F1 Scores

group1	group2	meandiff	p-adj	lower	upper	reject
causalrca	cloudranger	-0.0556	0.9869	-0.368	0.2569	False
causalrca	dycause	0.0322	0.9984	-0.2803	0.3446	False
causalrca	easyrca	0.0963	0.907	-0.2162	0.4087	False
causalrca	ocular	0.5877	0.0	0.2752	0.9002	True
cloudranger	dycause	0.0877	0.9319	-0.2248	0.4002	False
cloudranger	easyrca	0.1518	0.6489	-0.1607	0.4643	False
cloudranger	ocular	0.6433	0.0	0.3308	0.9558	True
dycause	easyrca	0.0641	0.9777	-0.2484	0.3766	False
dycause	ocular	0.5556	0.0001	0.2431	0.868	True
easyrca	ocular	0.4915	0.0004	0.179	0.8039	True

b) Tukey's HDS on average explanation time

group1	group2	meandiff	p-adj	lower	upper	reject
causalrca	cloudranger	-0.4848	0.9104	-2.0762	1.1065	False
causalrca	dycause	-1.3599	0.1279	-2.9513	0.2315	False
causalrca	easyrca	2.5731	0.0003	0.9818	4.1645	True
causalrca	ocular	-1.1715	0.245	-2.7628	0.4199	False
cloudranger	dycause	-0.8751	0.5347	-2.4664	0.7163	False
cloudranger	easyrca	3.058	0.0	1.4666	4.6493	True
cloudranger	ocular	-0.6866	0.7417	-2.278	0.9047	False
dycause	easyrca	3.933	0.0	2.3417	5.5244	True
dycause	ocular	0.1884	0.9972	-1.4029	1.7798	False
easyrca	ocular	-3.7446	0.0	-5.3359	-2.1532	True

Figure 4-15 Tukey's HDS results on the means of the algorithms' performance when slide sizes are varied

4.6.2.3.3 The Impact of the Data Arrival Rate

We vary the data arrival rate from 0.1 second to 1 second and set the window size to 8 minutes and the slide size to 2 minutes. As the data arrival rate decreases, the number of data points in an active window gets smaller. Figure 4-16 shows that the F1 Scores of all algorithms remain constant regardless of the data arrival rate, supporting our findings in Sections 4.6.2.3.1 and 4.6.2.3.2 that the number of data points does not affect F1 Scores. However, the number of data points does impact EasyRCA's average explanation time. With slower data arrival rates, EasyRCA's average explanation time decreases because the

algorithm processes fewer data points to form and analyze a normal regime for each outlier. The average explanation time of the other algorithms is not affected by data arrival rates.

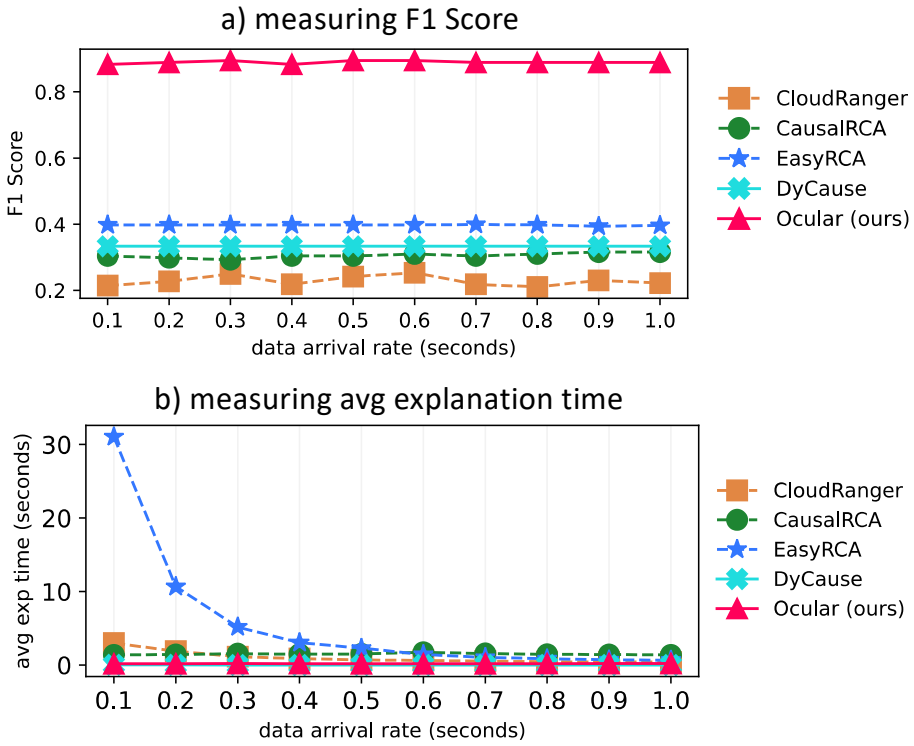


Figure 4-16 The impact of data arrival rates

Figure 4-16 shows that Ocular has the most desirable performance with F1 Score higher than 0.8 compared to the other algorithms that achieve 0.4 or lower F1 Scores. To confirm the result, we run 30 experiments for each algorithm. Using data from those experiments, we conduct One-way ANOVA which concludes that there is a significant difference in the algorithms' performance (rejecting the null hypothesis). We then apply Tukey HDS to find whether there is a significant difference between the means of each pair of the algorithm. The results are shown in Figure 4-17.

a) Tukey's HDS on F1 Scores

group1	group2	meandiff	p-adj	lower	upper	reject
causalrca	cloudranger	-0.0772	0.786	-0.2647	0.1102	False
causalrca	dycause	0.0275	0.9943	-0.16	0.2149	False
causalrca	easyrca	0.0917	0.6596	-0.0957	0.2792	False
causalrca	ocular	0.5836	0.0	0.3962	0.7711	True
cloudranger	dycause	0.1047	0.5363	-0.0827	0.2922	False
cloudranger	easyrca	0.1689	0.0987	-0.0185	0.3564	False
cloudranger	ocular	0.6609	0.0	0.4734	0.8483	True
dycause	easyrca	0.0642	0.8782	-0.1232	0.2517	False
dycause	ocular	0.5561	0.0	0.3687	0.7436	True
easyrca	ocular	0.4919	0.0	0.3045	0.6794	True

b) Tukey's HDS on average explanation time

group1	group2	meandiff	p-adj	lower	upper	reject
causalrca	cloudranger	-0.4904	0.9905	-3.4174	2.4366	False
causalrca	dycause	-1.4853	0.6275	-4.4123	1.4417	False
causalrca	easyrca	4.1996	0.0011	1.2726	7.1266	True
causalrca	ocular	-1.2998	0.7358	-4.2268	1.6272	False
cloudranger	dycause	-0.9949	0.8812	-3.9219	1.9321	False
cloudranger	easyrca	4.69	0.0002	1.763	7.617	True
cloudranger	ocular	-0.8094	0.9405	-3.7364	2.1176	False
dycause	easyrca	5.6849	0.0	2.7579	8.6119	True
dycause	ocular	0.1855	0.9998	-2.7415	3.1125	False
easyrca	ocular	-5.4994	0.0	-8.4264	-2.5724	True

Figure 4-17 Tukey's HDS results on the means of the algorithms' performance when data arrival rates are varied

In Figure 4-17 (part a), only the pairs of algorithms that include Ocular are found to have a statistically significant difference according to the Tukey HSD test. In other words, the null hypothesis, which typically states that there is no difference between the means of the compared groups, is rejected only for the algorithm pairs that involve Ocular, indicating that these pairs show a statistically significant difference. In these pairs, the Ocular algorithm has a higher average F1 score compared to each of the other algorithms. Specifically, Ocular's F1 score is higher by 0.5836, 0.6609, 0.5561, and 0.4915 when

compared to CausalRCA, CloudRanger, DyCause, and EasyRCA, respectively. This indicates that Ocular outperforms these other algorithms in terms of the F1 score by the given amounts. Furthermore, aligned with the findings in Figure 4.15, in Figure 4-16 (part b), only four pairs of algorithms that include EasyRCA show rejected null hypotheses in the Tukey HDS test. EasyRCA has significantly higher average explanation time compared to the other algorithms, indicating it is the slowest algorithm. In contrast, the remaining four algorithms (CausalRCA, CloudRanger, DyCause, and Ocular) demonstrate similarly low average explanation time. Considering both the F1 Score and average explanation time, it is clear that Ocular stands out as the best option.

4.6.3 Performance Results on Real World Datasets

There exist dataset benchmarks for point outlier detection, but to the best of our knowledge, real-world datasets with ground truth for the root cause factors of point outliers are not publicly available. Hence, we use time series datasets whose causal graphs' ground truth is available and add outliers into them. We conduct experiments on the FMRI (Functional Magnetic Resonance Imaging) datasets that contain BOLD (Blood-oxygen-level dependent) of 28 different underlying brain networks [123, 135, 136]. Each network (dataset) comprises neural activity data from up to 50 regions (vertices), with each region represented by a time series containing 50 to 5000 data points. While these datasets lack outliers, each possesses its ground truth causal graph. To replace 3% of inliers with outliers, we designate the leaf vertex in the causal graph as the target vertex and determine the linear regression function for each vertex. To inject an outlier, we randomly select a vertex with a path to the target vertex as a root cause and set its value as anomalous. The outlier value

of the root cause is propagated to the target vertex using the causal graph structure and the fitted regression functions along the path.

Table 4-4 The algorithms' performance on the FMRI datasets

Algorithms	average F1 Score	average explanation time (second)
CloudRanger	0.173	1.19
DyCause	0	**0.018**
EasyRCA	0.355	3.478
CausalRCA	0.618	9.446
Ocular (ours)	**0.714**	3.071

We simulate each FMRI dataset with injected outliers as a data stream. The data arrival rate, window size, and slide size are set to be 0.5 seconds, 8 minutes, and 2 minutes, respectively. Table 4-4 presents the simulation results, revealing that Ocular achieves the highest average F1 Score compared to other algorithms. Specifically, Ocular outperforms CloudRanger, EasyRCA, and CausalRCA by 312%, 101%, and 16%, respectively, in terms of F1 Score. DyCause, on the other hand, shows an average F1 Score of 0. The low F1 Scores of CloudRanger, DyCause, and EasyRCA can be attributed to their approach of constructing anomalous causal or propagation graphs based on available normal and anomalous data points. When the anomalous region (period) of outliers consists of only one anomalous timestamp, the result of the anomalous propagation graph might not be accurate due to insufficient outlier points to build it. In contrast, CausalRCA and Ocular do not require building such graphs; instead, they randomize noise term values to identify the root cause factors of point outliers. Furthermore, Ocular achieves a higher average F1 Score than CausalRCA because its root cause factor generation involves noise term values generated by FCM fitted with data points occurring before the outlier. In contrast,

CausalRCA lacks this step. The FCM fitting in Ocular can occur significantly earlier than the outlier timestamp, but a concept drift checker ensures that the FCM maintains the same data distribution as the recent data points leading up to the outlier. Conversely, running CausalRCA on a batch does not guarantee that its noise term values originate from FCM fitted with data points preceding the outlier. Therefore, Ocular yields a higher F1 Score than CausalRCA by 16%.

Table 4-4 shows that DyCause has the lowest average explanation time. However, since its average F1 score is 0, we cannot recommend it. CloudRanger is 2.5 times faster than Ocular, yet its average F1 Score of 0.173 is 312% lower than Ocular's F1 Score of 0.714. Even though both CausalRCA and Ocular rely on computing the contribution score of each vertex based on its participation also known as the Shapley value [87, 95]. CausalRCA is slower than Ocular because it needs to train a new FCM anytime a new batch of data points arrive. CausalRCA is also the slowest algorithm in this setup. Thus, based on the F1 Score and average explanation time metrics, Ocular is the best option for discovering root cause factors of point outliers in data streams.

CHAPTER 5

Conclusion and Future Research

5.1 Discovering Outlying Attributes of Outliers in Data Streams

In Chapter 3, we present EXOS, an algorithm to discover outliers' outlying attributes in real-time data streams that considers the characteristics of data streams. To address the challenge of the continuous arrival and unbounded volume of data streams (the notion of infinity), EXOS employs time-based tumbling windows where data points reside in memory until they expire, replaced by a new set of data points. To ensure timely processing of the data before they leave the active tumbling window, EXOS updates the eigenvectors or principal components model that captures the cross-correlation of data streams and clusters data points of each data stream in parallel. Both processes involve single-pass computation on the data points.

To capture the cross-correlation among data streams, EXOS's Estimator component needs to update the principal component model using data points from all active windows of the data streams. However, the data streams can have the same arrival rates (concurrent data streams) or varying arrival rates (non-concurrent data streams). EXOS handles concurrent or non-concurrent data streams by aligning data points in the active windows based on their timestamp steps. It then selectively employs aligned data corresponding to the timestamp steps that have data points from all streams, to update the principal component model.

EXOS determines the outlying attributes of each outlier by forming a local context using the principal components model and the closest inlier cluster. Adaptability to changes in data distribution, known as concept drift, is inherent in EXOS, as the model and the clusters are updated whenever a new window slides.

Experimental results demonstrate that EXOS is a promising technique for discovering outlying attributes of outliers in real-time data streams. It is shown to be effective when applied to data streams with both identical and varying arrival rates, achieving the best average F1 score (with one exception) while maintaining a competitive average explanation time. Overall, on average EXOS achieves 45.6% higher average F1 Score. EXOS has 1.3 to 29.6 times faster average explanation time than the other algorithms except when Synthetic 2 Dataset is simulated as non-concurrent data streams where EXstream is 1.2 times faster.

For future work, we can extend EXOS to address additional characteristics of data streams, specifically transiency and uncertainty. Transiency refers to the diminishing importance of data points over time, raising the question of how to incorporate this factor into the discovery of an outlier's outlying attributes. Currently, EXOS treats all data points in the active window with equal importance, but the significance of a data point may vary based on its temporal proximity to an outlier. Simultaneously, addressing uncertainty in data streams arising from errors or environmental factors presents an intriguing avenue for further research, especially regarding its impact on the accuracy of outliers' outlying attributes.

5.2 Discovering Root Cause Factors of Outliers in Data Streams

In Chapter 4, we present Ocular, an online algorithm to identify root causes of point outliers in real-time data streams. Ocular applies counterfactual inspired by CausalRCA. To deal with the unbounded volume of data streams (the notion of infinity), Ocular uses the time-based sliding window where data points remain in the memory until a new slide arrives. We propose four facets of the Active FCM-related Data Structures (AFDS) to ensure that the root causes of each outlier are generated with the FCM trained with data that chronologically occurred before the outlier (the notion of time). Since data streams are subject to concept drift, we propose a mechanism to detect when a concept drift takes place by monitoring the FCM's prediction error. This mechanism allows the algorithm to update the AFDS only when the concept drift occurs. To ensure that the Ocular can process data before they disappear from the active window, the process of generating the root causes of outliers and the process of detecting concept drifts are run in parallel.

The experiments on synthetic and real datasets show that Ocular has the highest F1 Score compared to the four existing algorithms: CausalRCA, CausalRange, DyCause, and EasyRCA. It also achieves the shortest average explanation time compared to the algorithms whose F1 Scores are higher than 0.5, which are DyCause and EasyRCA when the number of vertices is fewer than five in Synthetic datasets and CausalRCA in all datasets.

For future research, we plan to extend Ocular to address additional characteristics of data streams including transiency and uncertainty. Transiency refers to the decreasing significance of data points as time passes. As data age, their explanatory power in

explaining outliers diminishes. Even when a data point is in the same window as an outlier, its significance to the outlier could be less than a more recently arrived data point. This leads to the question: "How can we incorporate transiency into root cause analysis?" Simultaneously, uncertainty in data streams may stem from errors in data acquisition equipment or environmental factors. This uncertainty has the potential to impact the precision of outlier causality explanations, given that the training causal model heavily relies on the available data. Consequently, these characteristics present intriguing opportunities for further research. In addition, currently Ocular is applied on every data stream independently. Leveraging the relationships among data streams might be beneficial and warrant further research.

References

1. Sadik, Md.S., Gruenwald, L.: Research Issues in Outlier Detection for Data Streams. SIGKDD Explorations. 15, 33–40 (2014)
2. Yoon, S., Shin, Y., Lee, J.G., Lee, B.S.: Multiple Dynamic Outlier-Detection from a Data Stream by Exploiting Duality of Data and Queries. In: Proceedings of the ACM SIGMOD International Conference on Management of Data (2021)
3. Krempl, G., Žliobaite, I., Brzeziński, D., Hüllermeier, E., Last, M., Lemaire, V., Noack, T., Shaker, A., Sievi, S., Spiliopoulou, M., Stefanowski, J.: Open Challenges for Data Stream Mining Research. SIGKDD Explor. Newsl. 16, 1–10 (2014). https://doi.org/10.1145/2674026.2674028
4. Yu, K., Shi, W., Santoro, N.: Designing a Streaming Algorithm for Outlier Detection in Data Mining - an Incremental Approach. Sensors (Switzerland). 20, (2020). https://doi.org/10.3390/s20051261
5. Gomes, H.M., Read, J., Bifet, A., Barddal, J.P., Gama, J.: Machine Learning for Streaming Data: State of the Art, Challenges, and Opportunities. SIGKDD Explor. Newsl. 21, (2019)
6. Ditzler, G., Roveri, M., Alippi, C., Polikar, R.: Learning in Nonstationary Environments: A Survey. IEEE Comput Intell Mag. 10, (2015). https://doi.org/10.1109/MCI.2015.2471196
7. Lu, J., Liu, A., Dong, F., Gu, F., Gama, J., Zhang, G.: Learning under Concept Drift: A Review. IEEE Trans Knowl Data Eng. 31, (2019). https://doi.org/10.1109/TKDE.2018.2876857
8. Wang, S., Minku, L.L., Chawla, N., Yao, X.: Learning in the Presence of Class Imbalance and Concept Drift. Neurocomputing. 343, (2019). https://doi.org/10.1016/j.neucom.2019.01.080
9. Chandola, V., Banerjee, A., Kumar, V.: Anomaly Detection: A Survey. ACM Comput. Surv. 41, 15:1-15:58 (2009)

10. Tran, L., Mun, M.Y., Shahabi, C.: Real-time Distance-based Outlier Detection in Data Streams. In: Proceedings of the VLDB Endowment (2020)
11. Yoon, S., Lee, J.-G., Lee, B.S.: NETS: Extremely Fast Outlier Detection from a Data Stream via Set-based Processing. In: Proceedings of the VLDB Endowment (2019)
12. Siddiqui, M.A., Fern, A., Dietterich, T.G., Wong, W.-K.: Sequential Feature Explanations for Anomaly Detection. ACM Transactions on Knowledge Discovery from Data (TKDD). 13, 1–22 (2019)
13. Wang, P., Xu, J., Ma, M., Lin, W., Pan, D., Wang, Y., Chen, P.: CloudRanger: Root Cause Identification for Cloud Native Systems. In: 2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID). pp. 492–502 (2018)
14. Panjei, E., Gruenwald, L., Leal, E., Nguyen, C., Silvia, S.: A Survey on Outlier Explanations. The VLDB Journal. 31, 977–1008 (2022). https://doi.org/10.1007/s00778-021-00721-1
15. Carden, E.P., Fanning, P.J.: Vibration Based Condition Monitoring: A Review. Struct Health Monit. 3, 355–377 (2004)
16. Worden, K., Manson, G.: The Application of Machine Learning to Structural Health Monitoring. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences. 365, 515–537 (2006)
17. Yoon, S., Lee, J.G., Lee, B.S.: Ultrafast Local Outlier Detection from a Data Stream with Stationary Region Skipping. In: Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (2020)
18. Oreilly, C., Gluhak, A., Imran, M.A., Rajasegarar, S.: Anomaly Detection in Wireless Sensor Networks in a Non-Stationary Environment. IEEE Communications Surveys and Tutorials. 16, (2014). https://doi.org/10.1109/SURV.2013.112813.00168
19. Salehi, M., Leckie, C., Bezdek, J.C., Vaithianathan, T., Zhang, X.: Fast Memory Efficient Local Outlier Detection in Data Streams. IEEE Trans

Knowl Data Eng. 28, 3246–3260 (2016). https://doi.org/10.1109/TKDE.2016.2597833

20. Shahid, N., Naqvi, I.H., Qaisar, S. Bin: Characteristics and Classification of Outlier Detection Techniques for Wireless Sensor Networks in Harsh Environments: a Survey. Artif Intell Rev. 43, (2015). https://doi.org/10.1007/s10462-012-9370-y
21. Silveira, F., Diot, C.: URCA: Pulling out Anomalies by their Root Causes. In: 2010 Proceedings IEEE INFOCOM. pp. 1–9 (2010)
22. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and Issues in Data Stream Systems. In: Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems. pp. 1–16. Association for Computing Machinery, New York, NY, USA (2002)
23. Hawkins, D.: Identification of Outliers. In: Monographs on Applied Probability and Statistics (1980)
24. Aggarwal, C.: Outlier Analysis. In: Springer New York (2017)
25. Miller, T.: Explanation in Artificial Intelligence: Insights from the Social Sciences. Artif Intell. 267, (2019). https://doi.org/10.1016/j.artint.2018.07.007
26. Micenková, B., Ng, R.T., Dang, X.-H., Assent, I.: Explaining Outliers by Subspace Separability. In: 2013 IEEE 13th International Conference on Data Mining. pp. 518–527 (2013)
27. Dang, X.H., Assent, I., Ng, R.T., Zimek, A., Schubert, E.: Discriminative Features for Identifying and Interpreting Outliers. In: 2014 IEEE 30th International Conference on Data Engineering. pp. 88–99. IEEE (2014)
28. Meng, Y., Zhang, S., Sun, Y., Zhang, R., Hu, Z., Zhang, Y., Jia, C., Wang, Z., Pei, D.: Localizing Failure Root Causes in a Microservice through Causality Inference. In: 2020 IEEE/ACM 28th International Symposium on Quality of Service, IWQoS 2020 (2020)

29. Milenkoski, A., Vieira, M., Kounev, S., Avritzer, A., Payne, B.D.: Evaluating Computer Intrusion Detection Systems: A Survey of Common Practices. ACM Comput Surv. 48, (2015). https://doi.org/10.1145/2808691
30. Avritzer, A., Tanikella, R., James, K., Cole, R.G., Weyuker, E.: Monitoring for Security Intrusion using Performance Signatures. In: WOSP/SIPEW'10 - Proceedings of the 1st Joint WOSP/SIPEW International Conference on Performance Engineering (2010)
31. Desai, S.: Research Guides: Fake News, Lies and Propaganda: How to Sort Fact from Fiction: What is Fake News, https://guides.lib.umich.edu/fakenews
32. Fleming, N.: Coronavirus Misinformation, and How Scientists Can Help to Fight It. Nature. 583, (2020). https://doi.org/10.1038/d41586-020-01834-3
33. Yu, R., Qiu, H., Wen, Z., Lin, C., Liu, Y.: A Survey on Social Media Anomaly Detection. ACM SIGKDD Explorations Newsletter. 18, 1–14 (2016). https://doi.org/10.1145/2980765.2980767
34. Li, D., Guo, H., Wang, Z., Zheng, Z.: Unsupervised Fake News Detection Based on Autoencoder. IEEE Access. 9, (2021). https://doi.org/10.1109/ACCESS.2021.3058809
35. Feng, Q., Dou, Z., Li, C., Si, G.: Anomaly Detection of Spectrum in Wireless Communication via Deep Autoencoder. J Supercomput. 73, 3161–3178 (2017). https://doi.org/https://doi.org/10.1007/s11227-017-2017-7
36. Luo, Z., Xiong, Y., Zuo, R.: Recognition of Geochemical Anomalies using a Deep Variational Autoencoder Network. Applied Geochemistry. 122, (2020). https://doi.org/10.1016/j.apgeochem.2020.104710
37. Guidotti, R., Monreale, A., Ruggieri, S., Turini, F., Giannotti, F., Pedreschi, D.: A Survey of Methods for Explaining Black Box Models. ACM Comput Surv. 51, (2018). https://doi.org/10.1145/3236009
38. Song, F., Diao, Y., Read, J., Stiegler, A., Bifet, A.: EXAD: A System for Explainable Anomaly Detection on Big Data Traces. In: 2018 IEEE International Conference on Data Mining Workshops (ICDMW). pp. 1435–1440 (2018)

39. Akoglu, L., Tong, H., Koutra, D.: Graph based Anomaly Detection and Description: a Survey. Data Min Knowl Discov. 29, 626–688 (2014)
40. Abdallah, A., Maarof, M.A., Zainal, A.: Fraud Detection System: a Survey. Journal of Network and Computer Applications. 68, (2016). https://doi.org/10.1016/j.jnca.2016.04.007
41. Pascual Al, Marchini Kyle, Dyke Van Aleia: Overcoming False Positives: Saving the Sale and the Customer Relationship, https://javelinstrategy.com/research/overcoming-false-positives-saving-sale-and-customer-relationship
42. Wedge, R., Kanter, J.M., Veeramachaneni, K., Rubio, S.M., Perez, S.I.: Solving the False Positives Problem in Fraud Prediction using Automated Feature Engineering. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) (2019)
43. Farrar, C.R., Worden, K.: Structural Health Monitoring: A Machine Learning Perspective. (2012)
44. Bigoni, C., Hesthaven, J.S.: Simulation-based Anomaly Detection and Damage Localization: an Application to Structural Health Monitoring. Comput Methods Appl Mech Eng. 363, (2020). https://doi.org/10.1016/j.cma.2020.112896
45. Mao, J., Wang, H., Spencer, B.F.: Toward Data Anomaly Detection for Automated Structural Health Monitoring: Exploiting Generative Adversarial Nets and Autoencoders. Struct Health Monit. 20, (2021). https://doi.org/10.1177/1475921720924601
46. Pan, Y., Ma, M., Jiang, X., Wang, P.: DyCause: Crowdsourcing to Diagnose Microservice Kernel Failure. IEEE Trans Dependable Secure Comput. (2023). https://doi.org/10.1109/TDSC.2022.3233915
47. Knorr, E.M., Ng, R.T.: Algorithms for Mining Distance-Based Outliers in Large Datasets. In: Proceedings of the 24th International Conference on Very Large Data Bases (1998)

48. Keller, F., Müller, E., Wixler, A., Böhm, K.: Flexible and Adaptive Subspace Search for Outlier Analysis. In: Proceedings of the 22nd ACM International Conference on Information & Knowledge Management (2013)
49. Takeishi, N.: Shapley Values of Reconstruction Errors of PCA for Explaining Anomaly Detection. In: 2019 International Conference on Data Mining Workshops (ICDMW). pp. 793–798 (2019)
50. Borowski, S.: The Origin and Popular Use of Occam's Razor, https://www.aaas.org/origin-and-popular-use-occams-razor
51. Liu, N., Shin, D., Hu, X.: Contextual Outlier Interpretation. In: Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-18). pp. 2461–2467 (2018)
52. Pearl, J., Mackenzie, D.: The Book of Why, The New Science of Cause and Effect. Basic Books Hachette Book Group (2018)
53. Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernández-Moctezuma, R.J., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E., Whittle, S.: The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-order Data Processing. In: Proceedings of the VLDB Endowment (2015)
54. Knorr, E.M., Ng, R.T.: Finding Intensional Knowledge of Distance-Based Outliers. In: VLDB (1999)
55. Keller, F., Müller, E., Wixler, A., Böhm, K.: Flexible and adaptive subspace search for outlier analysis. Proceedings of the 22nd ACM international conference on Information & Knowledge Management. (2013)
56. Gupta, N., Eswaran, D., Akoglu, L., Faloutsos, C., Shah, N.: Beyond Outlier Detection: LookOut for Pictorial Explanation. In: ECML PKDD (2019)
57. Liu, F.T., Ting, K.M., Zhou, Z.-H.: Isolation Forest. 2008 Eighth IEEE International Conference on Data Mining. 413–422 (2008)
58. Leskovec, J., Krause, A., Guestrin, C., Faloutsos, C., VanBriesen, J., Glance, N.: Cost-effective outbreak detection in networks. In: Proceedings of the 13th ACM SIGKDD international conference on knowledge discovery and data mining. pp. 420–429. ACM (2007)

59. Dang, X.-H., Micenková, B., Assent, I., Ng, R.T.: Local Outlier Detection with Interpretation. In: ECML/PKDD (2013)
60. Micenková, B., Ng, R.T., Dang, X.-H., Assent, I.: Explaining Outliers by Subspace Separability. 2013 IEEE 13th International Conference on Data Mining. 518–527 (2013)
61. Dang, X.H., Assent, I., Ng, R.T., Zimek, A., Schubert, E.: Discriminative features for identifying and interpreting outliers. In: 2014 IEEE 30th International Conference on Data Engineering. pp. 88–99. IEEE (2014)
62. Tibshirani, R., Walther, G.: Cluster Validation by Prediction Strength. Journal of Computational and Graphical Statistics. 14, 511–528 (2005)
63. Cortes, C., Vapnik, V.: Support-Vector Networks. Mach Learn. 20, 273–297 (2004)
64. Panjei, E., Gruenwald, L., Leal, E., Nguyen, C.: Micro-clusters-based Outlier Explanations for Data Streams. In: The 1st Workshop on Anomaly and Novelty Detection, Explanation and Accommodation (ANDEA), co-located with 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)). pp. 1–8 (2021)
65. Kopp, M., Pevný, T., Holena, M.: Interpreting and clustering outliers with sapling random forests. In: ITAT (2014)
66. Song, F., Diao, Y., Read, J., Stiegler, A., Bifet, A.: EXAD: A System for Explainable Anomaly Detection on Big Data Traces. 2018 IEEE International Conference on Data Mining Workshops (ICDMW). 1435–1440 (2018)
67. Amarasinghe, K., Kenney, K., Manic, M.: Toward Explainable Deep Neural Network Based Anomaly Detection. 2018 11th International Conference on Human System Interaction (HSI). 311–317 (2018)
68. Zhang, H., Diao, Y., Meliou, A.: EXstream: Explaining Anomalies in Event Stream Monitoring. In: EDBT (2017)
69. Takeishi, N.: Shapley Values of Reconstruction Errors of PCA for Explaining Anomaly Detection. 2019 International Conference on Data Mining Workshops (ICDMW). 793–798 (2019)

70. Chen, Z., Tang, J., Fu, A.W.-C.: Modeling and efficient mining of intentional knowledge of outliers. Seventh International Database Engineering and Applications Symposium, 2003. Proceedings. 44–53 (2003)
71. Huang, B., Yang, P.: Finding Key Knowledge Attribute Subspace of Outliers in High-Dimensional Dataset. Expert Syst Appl. 38, 10147–10152 (2011). https://doi.org/https://doi.org/10.1016/j.eswa.2011.02.077
72. Chen, Z., Tang, J., Fu, A.W.-C.: Modeling and Efficient Mining of Intentional Knowledge of Outliers. In: Proceedings of the 7th International Database Engineering and Applications Symposium. pp. 44–53 (2003)
73. Macha, M., Akoglu, L.: Explaining Anomalies in Groups with Characterizing Subspace Rules. Data Min Knowl Discov. 32, 1444–1480 (2018)
74. Buchbinder, N., Feldman, M., Naor, J., Schwartz, R.: Submodular Maximization with Cardinality Constraints. In: Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 1433–1452 (2014)
75. Tsai, C.-W., Lai, C.-F., Chao, H.-C., Vasilakos, A.: Big Data Analytics: a Survey. J Big Data. 2, 1–32 (2015)
76. Amato, A.: On the Role of Distributed Computing in Big Data Analytics. In: Distributed Computing in Big Data Analytics. pp. 1–10. Springer International Publishing, Cham (2017)
77. Dutta, K.: Distributed Computing Technologies in Big Data Analytics. In: Distributed Computing in Big Data Analytics. pp. 57–82. Springer International Publishing, Cham (2017)
78. Koutris, P., Salihoglu, S., Suciu, D.: Algorithmic Aspects of Parallel Data Processing. Foundations and Trends in Databases. 8, (2018). https://doi.org/10.1561/1900000055
79. Knorr, E.M., Ng, R.T.: Finding Intensional Knowledge of Distance-Based Outliers. In: Proceedings of the 25th International Conference on Very Large Data Bases (1999)
80. Dang, X.-H., Micenková, B., Assent, I., Ng, R.T.: Local Outlier Detection with Interpretation. In: ECML/PKDD (2013)

81. Kopp, M., Holena, M.: Evaluation of Association Rules Extracted during Anomaly Explanation. In: ITAT (2015)
82. Zhang, H., Diao, Y., Meliou, A.: EXstream: Explaining Anomalies in Event Stream Monitoring. In: International Conference on Extending Database Technology (EDBT) (2017)
83. Jacob, V., Song, F., Stiegler, A., Rad, B., Diao, Y., Tatbul, N.: Exathlon: A Benchmark for Explainable Anomaly Detection over Time Series. Proc. VLDB Endow. 14, 2613–2626 (2021). https://doi.org/10.14778/3476249.3476307
84. Abuzaid, F., Bailis, P., Ding, J., Gan, E., Madden, S., Narayanan, D., Rong, K., Suri, S.: MacroBase: Prioritizing Attention in Fast Data. ACM Trans. Database Syst. 43, (2018). https://doi.org/10.1145/3276463
85. Amarasinghe, K., Kenney, K., Manic, M.: Toward Explainable Deep Neural Network Based Anomaly Detection. In: 2018 11th International Conference on Human System Interaction (HSI). pp. 311–317 (2018)
86. Zhang, C., Zhou, Z., Zhang, Y., Yang, L., He, K., Wen, Q., Sun, L.: Netrca: An Effective Network Fault Cause Localization Algorithm. In: ICASSP 2022 - 2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). pp. 9316–9320 (2022)
87. Budhathoki, K., Minorics, L., Bloebaum, P., Janzing, D.: Causal structure-based root cause analysis of outliers. In: Chaudhuri, K., Jegelka, S., Song, L., Szepesvari, C., Niu, G., and Sabato, S. (eds.) Proceedings of the 39th International Conference on Machine Learning. pp. 2357–2369. PMLR (2022)
88. Assaad, C.K., Ez-Zejjari, I., Zan, L.: Root Cause Identification for Collective Anomalies in Time Series given an Acyclic Summary Causal Graph with Loops. In: Ruiz, F., Dy, J., and van de Meent, J.-W. (eds.) Proceedings of The 26th International Conference on Artificial Intelligence and Statistics. pp. 8395–8404. PMLR (2023)
89. Spirtes, P., Glymour, C., Scheines Richard: Causation, Prediction, and Search. MIT Press, Cambridge, Mass. (2000)

90. Han, J., Kamber, M.: Outlier Detection. In: Data Mining: Concepts and Techniques. Morgan Kaufman Publishers (2012)
91. Pearl, J.: An Introduction to Causal Inference. International Journal of Biostatistics. 6, (2010). https://doi.org/10.2202/1557-4679.1203
92. Pearl, J.: Causal Inference in Statistics: an Overview. Stat Surv. 3, (2009). https://doi.org/10.1214/09-SS057
93. Zhang, K., Wang, Z., Zhang, J., Schölkopf, B.: On Estimation of Functional Causal Models: General Results and Application to the Post-nonlinear Causal Model. ACM Trans Intell Syst Technol. 7, (2015). https://doi.org/10.1145/2700476
94. Glymour, C., Zhang, K., Spirtes, P.: Review of Causal Discovery Methods based on Graphical Models. Front Genet. 10, (2019). https://doi.org/10.3389/fgene.2019.00524
95. Aas, K., Jullum, M., Løland, A.: Explaining Individual Predictions when Features are Dependent: More Accurate Approximations to Shapley Values. Artif Intell. 298, 103502 (2021). https://doi.org/https://doi.org/10.1016/j.artint.2021.103502
96. Shlens, J.: A Tutorial on Principal Component Analysis, http://arxiv.org/abs/1404.1100
97. MacQueen, J.: Some Methods for Classification and Analysis of Multivariate Observations. In: Proceedings of the fifth Berkeley Symposium on Mathematical Statistics and Probability (1967)
98. Jollife, I.T., Cadima, J.: Principal Component Analysis: A Review and Recent Developments. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences. 374, (2016). https://doi.org/10.1098/rsta.2015.0202
99. Li, C.L., Lin, H. Ten, Lu, C.J.: Rivalry of Two Families of Algorithms for Memory-restricted Streaming PCA. In: Proceedings of the 19th International Conference on Artificial Intelligence and Statistics, AISTATS 2016 (2016)
100. Ackerman, M., Dasgupta, S.: Incremental clustering: The Case for Extra Clusters. In: Advances in Neural Information Processing Systems (2014)

101. Mendenhall, W.M., Sincich, T.L.: Chapter 2 Descriptive Statistics. In: Statistics for Engineering and the Sciences. pp. 25–75 (2016)
102. Boser, B.E., Guyon, I.M., Vapnik, V.N.: Training Algorithm for Optimal Margin Classifiers. In: Proceedings of the Fifth Annual ACM Workshop on Computational Learning Theory (1992)
103. Tibshirani, R.: Regression shrinkage and selection via the lasso: A retrospective. J R Stat Soc Series B Stat Methodol. 73, (2011). https://doi.org/10.1111/j.1467-9868.2011.00771.x
104. Zaki, M.J., Meira JR, W.: Representative-based Clustering. In: Data Mining and Analysis Fundamental Concepts and Algorithms. pp. 334–335. Cambridge University Press (2014)
105. Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: 2.10 QR Decomposition. In: Numerical Recipes The Art of Scientific Computing. pp. 102–103 (2007)
106. Thomas, D.B., Luk, W.: Multivariate Gaussian Random Number Generation Targeting Reconfigurable Hardware. ACM Trans Reconfigurable Technol Syst. 1, (2008). https://doi.org/10.1145/1371579.1371584
107. Price, J., Wong, A., Yuan, T., Mathews, J., Olorunniwo, T.: Stochastic Gradient Descent, http://optimization.cbe.cornell.edu/index.php?title=Stochastic_gradient_descent
108. Scikit Learn: Support Vector Machines, https://scikit-learn.org/stable/modules/svm.html#svm
109. Das, S.: Best Practices for Dealing with Concept Drift, https://neptune.ai/blog/concept-drift-best-practices
110. Bodik, P., Hong, W., Guestrin, C., Madden, S., Paskin, M., Thibaux, R.: Intel Lab Data, http://db.csail.mit.edu/labdata/labdata.html, (2004)
111. Buckreis, T., Winders, A., Wang, P., Brandenberg, S., Stewart, J.: Microtremor Data Collected in Sacramento-San Joaquin Delta Region of California., https://doi.org/10.17603/ds2-dk6t-8610, (2021)

112. Makonin, S.: AMPds2: The Almanac of Minutely Power dataset (Version 2), https://doi.org/10.7910/DVN/FIE0S4, (2016)
113. Hardin, J., Garcia, S.R., Golan, D.: A Method for Generating Realistic Correlation Matrices. Ann Appl Stat. 7, 1733–1762 (2013)
114. Panjei, E.: exos, https://github.com/egawati/exos
115. Liu, N.: Contextual-Outlier-Interpreter, https://github.com/ninghaohello/Contextual-Outlier-Interpreter
116. Panjei, E.: micoes, https://github.com/egawati/micoes
117. Mendenhall, W.M., Sincich, T.L.: Chapter 14 The Analysis of Variance for Designed Experiments. In: Statistics for Engineering and the Sciences. pp. 742–836 (2016)
118. Abdi, H., Williams, L.J.: Tukey's Honestly Significant Difference (HSD) Test, (2010)
119. Gu, F.: Concept Drift Detection for Machine Learning with Stream Data, https://opus.lib.uts.edu.au/bitstream/10453/140165/2/02whole.pdf, (2019)
120. Yepmo, V., Smits, G., Pivert, O.: Anomaly Explanation: A Review. Data Knowl Eng. 137, (2022). https://doi.org/10.1016/j.datak.2021.101946
121. Hitchcock, C.: Causal Models. In: Zalta, E.N. and Nodelman, U. (eds.) The Stanford Encyclopedia of Philosophy. Metaphysics Research Lab, Stanford University (2023)
122. Mastakouri, A.A., Schölkopf, B., Janzing, D.: Necessary and Sufficient Conditions for Causal Feature Selection in Time Series with Latent Common Causes. In: Proceedings of Machine Learning Research (2021)
123. Assaad, C.K., Devijver, E., Gaussier, E.: Survey and Evaluation of Causal Discovery Methods for Time Series. Journal of Artificial Intelligence Research. 73, (2022). https://doi.org/10.1613/JAIR.1.13428
124. Poepsel-Lemaitre, R., Kiefer, M., von Hein, J., Quiané-Ruiz, J.-A., Markl, V.: In the Land of Data Streams Where Synopses Are Missing, One Framework to Bring Them All. Proc. VLDB Endow. 14, 1818–1831 (2021). https://doi.org/10.14778/3467861.3467871

125. SciPy: Statistics (scipy.stats), https://docs.scipy.org/doc/scipy/tutorial/stats.html
126. Wang, Q., Kulkarni, S.R., Verdú, S.: Divergence Estimation for Multidimensional Densities via k-Nearest-neighbor Distances. IEEE Trans Inf Theory. 55, 2392–2405 (2009)
127. Developers, N.: NetworkX, Network Analysis in Python, https://networkx.org/
128. Zadeh, R.: Distributed Algorithms and Optimization, https://stanford.edu/~rezab/classes/cme323/S15/notes/lec11.pdf
129. Sharma, A., Kiciman, E., others: DoWhy: A Python Package for Causal Inference, https://www.pywhy.org
130. Panjei, E.: Outlier Causality in Data Streams, https://github.com/egawati/outlier_causality
131. Assaad, C.K.: Code Python EasyRCA, https://github.com/ckassaad/EasyRCA
132. Pan, Y.: DyCause, https://github.com/PanYicheng/dycause_rca
133. Štrumbelj, E., Kononenko, I.: Explaining Prediction Models and Individual Predictions with Feature Contributions. Knowl Inf Syst. 41, 647–665 (2014). https://doi.org/10.1007/s10115-013-0679-x
134. Janzing, D., Minorics, L., Bloebaum, P.: Feature Relevance Quantification in Explainable AI: A Causal Problem. In: Chiappa, S. and Calandra, R. (eds.) Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics. pp. 2907–2916. PMLR (2020)
135. Smith, S.M., Miller, K.L., Salimi-Khorshidi, G., Webster, M., Beckmann, C.F., Nichols, T.E., Ramsey, J.D., Woolrich, M.W.: Network Modeling Methods for FMRI. Neuroimage. 54, (2011). https://doi.org/10.1016/j.neuroimage.2010.08.063
136. Nauta, M., Bucur, D., Seifert, C.: Causal Discovery with Attention-Based Convolutional Neural Networks. Mach Learn Knowl Extr. 1, (2019). https://doi.org/10.3390/make1010019