Vrije Universiteit Amsterdam

Master Thesis

# *Telosian*: a real-time anomaly detection algorithm optimized for concept drift

**Author:** Iker Antonio Olarra Maldonado (2730059)

*1st supervisor:* Rob van der Mei
*daily supervisor:* Erik Meeuwissen & Puck de Haan (TNO)
*2nd reader:* Anil Yaman

*A thesis submitted in fulfillment of the requirements for VU Master of Science degree in Business Analytics*

September 21, 2023

*"The Telosian organ of cognition is housed inside a segmented body that buds and grows at one end while withering and shedding at the other. Every year, a fresh segment is added at the head to record the future; every year, an old segment is discarded from the tail, consigning the past to oblivion."*

*from* An Advanced Readers' Picture Book Of Comparative Cognition, *by Ken Liu*

## Abstract

The increasing amount of available data and new attacks appearing in the cyber security sector require new methods to timely detect new threats. Additionally, the concept drift in the data poses a new challenge as algorithms need to be updated constantly to maintain their performance. Existing methods use different techniques to update their model. However, they lack the ability to update according to the specific amount of drift observed. This can result in poor detection performance or unnecessary computational effort. In this research we address this issue by proposing *Telosian*, an unsupervised real-time algorithm capable of adapting to concept drift. We show that measuring the amount of drift to perform a custom update to the algorithm results in a more efficient use of resources without degrading performance. The proposed method achieves comparable results to state-of-the-art methods while using fewer resources.

To Aitana, Antonio and Lorena.

## Acknowledgements

First, I would like to say special thank you to Puck for her constant support during the development of this thesis as well as the multiple “long sentence alerts” when reviewing this work.

Second, I want to thank Erik for his guidance and advice during the entire process.

Third, I want to thank Dr. Gábor Horváth, one of the authors of [19], for sharing the implementation of the *BWOAIF* algorithm and his availability to solve my questions about his research and disposition to hear my comments and suggestions. He was of great help in the development of this project.

Finally, I want to thank Dr. Dustin Mink and Dr. Sikha Bagui for their disposition in solving my questions and addressing my observations regarding the UWF-Zeek dataset [9].

# Contents

# List of Figures

# List of Tables

# 1

# Introduction

The rapid increase in data availability over the last decades has allowed us to develop new ways of extracting information across different sectors. For example, industrial machinery can now automatically emit a warning when it needs maintenance with the aid of real time data generated by sensors [30], or loyalty programs can profile their customers in a more detailed way by combining multiple data sources and training new algorithms with them [33]. However, as data keeps being generated, the methods we use to exploit it must handle the difficulties a data-driven world brings. The abundance of data, its fast generation and the limited resources to process it, pose six main challenges to be addressed when building high performance data driven models: (1) scaling the models to process large amounts of data, (2) handling streams of data in real-time, (3) exploiting unlabeled data, (4) addressing concept drift, (5) efficient use of resources and finally, (6) updating models in a proportionate manner to maintain their performance over time. These six challenges are especially present in the cyber security sector, due to its dynamic and fast data generation nature. Below, we will further explain the causes and importance of these challenges. Additionally, we discuss the advantages and shortcomings of methods that have been proposed to address them.

The first challenge is **scalability**, which results from the great amount of data that must be processed. Existing data-processing methods, like distance-based anomaly detection, have either poor performance or require significantly longer execution times [25]. Being able to design models capable of handling large sets of data is of great importance, otherwise tremendous amounts of information could be ignored due to lack of appropriate methods to exploit them. This tremendous amount of data is generated at great speeds, which poses the second challenge: the need to process this continuous data stream in a timely manner. More and more systems produce large amounts of data every second, requiring methods capable of handling **streaming data** [10]. This challenge is closely related to the third one, which is the **lack of labeled datasets**. The speed at which data is created makes it difficult to obtain up to date and reliable labeled datasets. As a result, current methods will have a significant advantage if they are able to obtain satisfactory results without a labeled dataset [39], i.e., in an unsupervised manner. Furthermore, the systems that generate data evolve quickly, changing the characteristics of the output they produce. This is known as **concept drift** and failing to overcome this challenge can result in solutions that degrade over time, until they become obsolete [24]. The fifth challenge is the necessity to produce methods that **use resources in an efficient manner**. To process the great sequences of data points that are available, minimising storage consumption and usage of CPU is of paramount importance [10]. Finally, the last challenge is to **update the model in a proportional manner**. Most existing methods take one of two approaches: either a partial but constant update of the models [19], or a total retrain of the model when certain conditions are met [14]. An update that is performed according to how the data is changing, could result in a more efficient use of the resources and greater performance, since the update is tailored to the type of concept drift.

The aforementioned challenges are evident in the cyber security context. For example, the scalability challenge is apparent, considering that in 2024 it is expected that 22 billion internet of things devices will be in use [32]. On the other hand, streaming-data has a lot of relevance considering the speed at which cyberattacks are done. According to Cisco, there were the more than 23,000 cyber security threats per second in 2018 [39], which illustrates the pace at which data is generated. Furthermore, algorithms need to be able to handle data-streams to be able to quickly detect such threats and allow supporting systems to mitigate them in a few instants. In relation to this, the time-varying nature of streaming data makes concept drift a common phenomenon that must be taken into account [14]. Furthermore, the constant appearance of new attacks also produces changes in the data and require for constant updates of existing models. This also makes it difficult to develop a good cyber security dataset [9], since the data must be processed as it arrives and data preparation or labeling tasks can be time consuming. This results in threats not being identified in time. Moreover, many methods recreate models from scratch in order to update their models [19]. Examples are the *iForestASD* algorithm [14] which trains a new *IForest* algorithm every time concept drift is detected, or the HS-Trees algorithm [37], which resets the learnt parameters with every new batch of data. This approach can be inefficient when concept drift is small or nonexistent. Besides, computer networks may present small gradual changes due to an organic growth of the user base for example, but might also experience sudden changes caused by a new type of attack. A model must be able to handle both scenarios by performing a proportional adaptation as new data arrives [19]. This way, when small changes occur, the model will perform smooth updates, while greater changes trigger a sudden update.

Anomaly detection is a very effective way to address the lack of labeled data and can be modified to address the rest of the challenges, so it is worth exploring in this thesis. Its goal is to separate the normal data instances from the abnormal ones, which might be generated by deviating mechanisms and could therefore be of interest to us. These mechanisms might be a disease that causes abnormal symptoms like in [27], where anomaly detection is used to analyze images and detect skin disease, or like in [31] where it is used to detect adversarial attacks and remove them from the data when training Deep Learning models for natural language processing. The importance of anomaly detection lies in its flexibility to detect new outliers instead of learning specific types [42]. This is of great importance when dealing with data that changes over time. Besides, unsupervised anomaly detection algorithms do not require labeled datasets, which are often hard to procure. Making anomaly detection techniques powerful tools to exploit large amounts of unlabeled data [39].

Nevertheless, overcoming these challenges is not trivial and some of the most common unsupervised anomaly detection methods are insufficient to handle data streams [10]. Existing machine learning methods tackle a number of these challenges. For example, *iForest* [25], an unsupervised isolation-based anomaly detection algorithm, made great progress in creating fast and scalable solutions that do not require labeled data. Others, like [42], investigated the detection of new attacks (*zero-day attacks*) with anomaly detection by removing a type of attack from the training data. This solution partially addresses the concept drift issue. However, the models were not designed to handle streaming data. Later, modified versions of static-data based algorithms were proposed to handle streaming unlabeled data at high speeds while still detecting anomalies effectively [6]. Nevertheless, concept drift was not taken into account. Later, the idea of a sliding window was also used in later works to address changes in the distribution of the data. Although this was proven to be useful, it had the disadvantages such as requiring full retraining of the model which may lead to inefficient use of resources and insensitivity towards some types of concept drift [19]. This insensitivity was addressed with the *BWOAIF* algorithm proposed in [19], which leveraged the speed and scalability of *iForest* to design a more efficient streaming algorithm which handles data drift by using *iForest* in a sliding window. This approach improved accuracy and resource efficiency in comparison to existing methods such as the *iForestASD* algorithm[14]. However, the initial parameters of the algorithm determine the size of the update throughout its execution. Hence the size of the update remains constant regardless of the magnitude of the concept drift[1]. Thus, the challenge of designing an algorithm that is constantly updated according to the characteristics of the concept drift and performing a **proportional update** has not yet been overcome to the best of our knowledge. For that reason, in this research we attempt to answer the question: How can anomaly detection algorithms adapt to concept drift in real-time unsupervised detection of unseen attacks in cyber security?

In summary, we want to develop a method with the following main characteristics, such that it can be used to successfully exploit streaming data:

1. Scales to process large amounts of data.
2. Handles streams of data in real-time.
3. Is unsupervised.
4. Accounts for concept drift.
5. Utilizes resources efficiently.
6. Performs a proportional update.

In this work, we propose the algorithm *Telosian*, which has the aforementioned characteristics and has a comparable performance to existing methods designed for unsupervised cyber security streaming data. The main contribution of this research is adapting the *BWOAIF* algorithm [19] to include a mechanism that measures concept drift and performs a proportional update. This allows *Telosian* to adapt to different types of concept drift and use the computational resources in a more efficient manner, while maintaining detection performance.

[1]The algorithm also includes a weighting method that allows to give more or less influence to some trees and accelerate or slow down the adaptation. However, the number of replaced regressors, which constitute the basis of the algorithm, remains constant

# 2

# Background

The goal of this research is to propose an algorithm capable of performing unsupervised anomaly detection in streams of cyber security data, taking concept drift into account. To achieve this, a number of challenges need to be addressed, which require the investigation of multiple concepts and methods. In this section, we introduce all relevant concepts and methods: First, the advantages of using anomaly detection and methods to perform it. Second, the *iForest* algorithm, an unsupervised anomaly detection algorithm which will be the basis for the new algorithm presented in this research. Third, concept drift, the main challenge this research will address. Fourth, the *BWOAIF* algorithm, a state of the art anomaly detection algorithm adapted for concept drift which will serve as basis and benchmark for our proposed algorithm. Fifth, the *NNDVI* algorithm which is used to detect concept drift. Together, these five parts conform the building blocks for *Telosian*, the concept drift optimized anomaly detection algorithm we propose in this research.

## 2.1 Anomaly detection

In this section, we first explain what anomaly detection is, then what methods are used to perform it, followed by its application in the field of cyber security. The importance of anomaly detection relies on the capacity to quickly identify anomalous patterns in data without the need of previous knowledge of the nature of the anomalous observations. Being able to identify these anomalies reduces the amount of information that needs to be carefully analyzed to model the data and results in a more efficient use of resources. In the cyber security sector, for example, anomaly detection is used to develop more flexible and efficient solutions to detect new attacks [6]. Anomaly detection is also of paramount importance in fraud detection to infer complexities and dynamic changes in criminal behaviour [18]. In the medical sector, anomaly detection is used to predict the appearance of a certain medical conditions or diagnose already existing conditions [16]. To achieve these goals, anomaly detection attempts to find observations in the dataset that are particularly different from the rest. Anomalies are observations that do not conform with a defined normal behaviour in a specific context or dataset, for example, these anomalous observations could be generated by intrusions in a system, fraudulent behaviour or a medical condition. However, they are not necessarily harmful. For example, a change in policy or service might produce anomalous observations which do not pose a threat [6]. Nevertheless, anomalous instances of the data are usually worth being investigated to uncover new information as they can provide critical information about a system [5]. In this section, we will discuss different methods that can be used to detect anomalies.

### 2.1.1 Anomaly detection methods

One of the challenges in anomaly detection is to define what normal behaviour entails. What is normal or not can be highly dependent on the method used to perform anomaly detection. In this subsection we will explore some of the methods that have been proposed to find anomalous data points within a dataset. These methods are mainly divided in: density-based methods, clustering methods, and isolation-based methods [10] and they have a different approach in identifying the data which can be useful in different contexts.

Firstly, clustering- and density-based anomaly detection methods rely on the computation of distances between data points in feature space, in order to find hidden patterns. Clustering methods partition the dataset into clusters, whose elements share similar characteristics. Once the clusters are determined, points that do not belong to any of these clusters or are the furthest from them, according to a pre-defined metric, are labeled as anomalies [42]. The *k-Mediods* algorithm, for example, represents each cluster by its most centric object and identifies anomalies by checking which points are the furthest from the center. Density-based methods, on the other hand, group subsets of points that are close to each other according to a chosen distance measure and then identify the areas with lower density [38] in order to find anomalies. The Local Outlier Factor (LOF) algorithm [11] is an example of a density based method. It works by computing an anomaly score based on the relative proximity of a point to the rest of the points. It does this in two steps; first, using the distance between a point and its neighbors, and then, the distance of the neighbors to their respective neighbors [7]. This allows it to determine to which extent a point is a local outlier. Both k-Mediods and LOF are non-parametric algorithms and therefore do not assume a distribution of the data. Density-based methods are more sensitive towards local outliers, while clustering-based algorithms have a more global view of the data and therefore are more suited to identify global anomalies [10]. Although implementations of these methods have shown great accuracy (for example [12] leveraged the *K Nearest Neighbor* clustering algorithm in combination with *Support Vector Machines* to accurately classify intrusions), these approaches require pairwise distances calculation to compare the data instances, which makes them ineffective in large datasets, as they do not scale very well [39].

Contrarily, isolation-based methods rely on the assumption that anomalies are few, different and that the values of their attributes are different from those of normal instances of the data [25]. Hence, by separating the data based on individual attributes, isolation-based methods can identify anomalies without the need of computing distances, which reduces their complexity and allows them to process greater amounts of data [38]. In [35], for example, the isolation-based algorithm *iForest* was used to quickly identify anomalies in a database of over 300 million data instances, which shows the potential of this kind of method. Nevertheless, in the end the anomaly detection method used should be selected based on the data we are trying to exploit [10]. This means that it is relevant to explore the specific characteristics of the cyber security sector to determine the most suitable methods.

### 2.1.2 Anomaly detection in cyber security

In cyber security, anomaly detection has gained importance as conventional methods have become insufficient to detect the increasing number of incidents [6]. Although there already exist mechanisms to detect known attacks, such as rule-based intrusion detection systems, these mechanisms lack the flexibility needed to discover novel types of attacks [39]. It has been recognized that cyber defense requires the capabilities of artificial intelligence to address the new threats that arise everyday [22]. Algorithms such as Random Forest, Naive Bayes and Neural Networks have been successfully utilized in the cyber security context. In [6] we can see a summary of the good performance of such algorithms on multiple benchmark datasets. However, as stated before, quality labeled datasets are uncommon in the cyber security field, which poses a major obstacle for supervised algorithms. Additionally, to make it possible for such algorithms to detect new attacks, examples of these attacks must be recorded, labeled and used to retrain the algorithm. This is time consuming and requires human labeling. Anomaly detection algorithms, however, not only have the capacity to detect novel attacks, but can also uncover patterns in the data and potentially use this information to make other detection systems more robust. In [35], for example, anomaly detection is combined with expert feedback to increase accuracy and reduce the number of false positives in future runs. However, the models used in the cyber security sector often deal with two major challenges: lack of high quality datasets [6] and large quantities of data being continuously generated [10].

First, the data used is of great importance in machine learning applications. However, most of the publicly available datasets in the cyber security domain have quality issues. These issues range from data being outdated, the lack of examples of some of the existing attacks, class imbalance and noisy or non existent class labels [6]. Labeled data is a main issue, as generating labels is time consuming, which makes the development of supervised algorithms not feasible in most cases [35]. Secondly, the continuous streams of data being generated requires algorithms to be able to deal with data efficiently. Anomaly detection in streaming data must deal with fast generated data, meaning that the volume is infinite, which makes it impossible to store the whole volume of data and therefore rules out the traditional off-line methods that follow this approach [14]. However, the assumption that anomalies are rare and show deviating behavior holds for static anomaly data sets, but also for streaming data [38]. This means that the traditional methods based on the assumption that anomalies are rare and deviating could be useful for streaming data. Nevertheless, these methods must be adapted to be able to optimise the usage of CPU and reduce the memory consumption [10] before being applied to streaming data.

Nowadays, the amount of data available in many sectors has increased considerably. To transform this data into valuable information it is necessary to analyze it in an meaningful way. However, the enormous quantities of data make it impossible to analyze every record manually. For example, in the insurance sector, an expert could take up to 2 weeks to analyze around 50 claims [40]. Few models are able to handle the great amount of data in combination with the real time processing necessity [10]. A number of algorithms have been proposed or adapted to address the streaming and concept drift anomaly detection challenges. In the next subsection we highlight some algorithms used for this purpose and their main contributions.

### 2.1.3 Review of anomaly detection methods for streaming

An algorithm that aided in the effort of quickly detecting anomalies was *iForest* [25], an unsupervised isolation-based anomaly detection algorithm, which made great progress in creating fast and scalable solutions that do not require labeled data. Others, like [42], investigated the use of the *iForest* and *KMeans* algorithms to detect new attacks (*zero-day attacks*) by removing a type of attack from the training data and then testing the model's ability to detect the unseen attacks included in the dataset. This solution partially addresses the concept drift issue, however, the models were not designed to handle streaming data. Later, modified versions of the *KMeans* algorithm such as *KMedioids* and the *ADMIT* algorithms were proposed to handle streaming unlabeled data at high speeds while still detecting anomalies effectively [6]. Nevertheless, concept drift was not taken into account.

A common way to adapt traditional algorithms to deal with streaming data and concept drift, is by incorporating a sliding window [10], which allows the algorithm to work on an up-to-date subset of the data. The idea is to use the algorithm designed for static data sets, and apply it to intervals on a stream (sliding windows). This way, the algorithm is able to be trained and make predictions on the most up-to-date data, resulting in reduced storage and faster running times.

In [10], for instance, it was investigated how to adapt *KMeans* for real time detection on streaming data by using a sliding window and monitoring incoming data to detect when significant changes in the data arose and then update the algorithm accordingly. This idea of a sliding window, was also used in later works to address changes in the distribution of the data. An example is [14], where the *iForest* algorithm was modified to handle concept drift by monitoring the number of anomalies and training a new *iForest* when the proportion of anomalies surpassed a predefined threshold. This adaptation obtained a slightly lower accuracy than other static methods[1], but with a more efficient use of resources, since only a fraction of the data is used. Although this was proven to be useful, it had the disadvantage of requiring full retraining of the model which may lead to inefficient use of resources and insensitivity towards some types of concept drift [19].

[1]It must be noted that the algorithm was tested on static data, which explains the lower accuracy.

This insensitivity was addressed with the *BWOAIF* algorithm proposed in [19], which leveraged the speed and scalability of *iForest* to design a more efficient streaming algorithm which handles data drift by using *iForest* in a sliding window and replacing older regressors with new ones trained on the most recent data. This approach not only improves the accuracy of the algorithm in comparison to [14], but it also uses computing resources more efficiently as it avoids full retraining. However, the initial parameters of the algorithm determine the size of the update throughout its execution. Hence the size of the update remains constant regardless of the magnitude of the concept drift[1]. Thus, the challenge of designing an algorithm that is constantly updated according to the characteristics of the concept drift and performing a **proportional update** has not yet been overcome to the best of our knowledge.

In conclusion, in the cyber security context, algorithms that do not require labeled data, are isolation-based and easily scalable are suitable choices to detect harmful behaviour, as they address the main challenges present in this context. Specifically, the *iForest* algorithm has characteristics which makes it a good choice for cyber security data due to its capacity to scale, detect multiple types of anomalies and deal with high-dimensional data. Nevertheless, there is still a need to create more robust algorithms that can adequately detect attacks in the presence of concept drift. In the next section we will describe how *iForest* works, its main characteristics and good practices and how these are beneficial in the cyber security context.

## 2.2 *iForest*

*iForest* [25], is an unsupervised ensemble model designed to detect anomalies by isolation. It uses recursive random partitioning of the data to determine which instances are the most abnormal. It is built under the assumption that anomalies are few and different and therefore easier to isolate than “normal” instances of the data. The main advantages that make *iForest* relevant for this research are that it **does not require a labeled dataset** and that it is **highly scalable** [25]. In this section, we first explain the algorithm and then describe the characteristics that provide it with the aforementioned advantages.

[1]The algorithm also includes a weighting method that allows to give more or less influence to some trees and accelerate or slow down the adaptation. However, the number of replaced regressors, which constitute the basis of the algorithm, remains constant

The algorithm consists of two stages; first, the training of the algorithm and later the computation of an anomaly score over the data instances.

### 2.2.1 Training the algorithm

To train the algorithm on a dataset of $n$ points where each point is composed of $Q$ features or attributes, an ensemble of independent *iTrees* is generated from samples (without replacement) of the data. To construct a single *iTree*, first, an attribute is chosen from the data and then a random split value is selected between the minimum and maximum value of this attribute. The split results in two partitions of the original dataset which are subsequently divided by selecting another random attribute and split value for each sub-dataset. This process is repeated recursively on each partition until one of the two termination conditions is met: 1. every resulting partition has a single unique value, or 2. a predefined maximum number of splits (*iTree* height) is reached. Following this process results in an *iTree* where the most anomalous points are closer to the root of the tree, as these are more easily isolated. The following pseudo code describes the steps required to build an *iTree* (Algorithm 1):

**Algorithm 1** iTree

```
Require: $X$ - input data, $e$ - current tree height, $l$ - height limit.
  if $e \geq l$ or $|X| \leq 1$ then
      return $externalNode\{Size \leftarrow |X|\}$
  else
      $Q \leftarrow$ list of attributes in $X$
      Select a random attribute $q$ form $Q$
      Select a random split point $p$ from the interval [min($X[q]$),max($X[q]$)].
      $X_l \leftarrow filter(X, q < p)$
      $X_r \leftarrow filter(X, q \geq p)$
      return $internalNode\{Left \leftarrow (X_l, e+1, l),$
                            $Right \leftarrow iTree(X_r, e+1, l),$
                            $SplitAtt \leftarrow q,$
                            $SplitValue \leftarrow p\ \}$
  end if
```

As an example, lets consider Figure 2.1 where the rectangles describe a dataset with two features, one represented by the x-axis and another one represented with the y-axis. We will use this to illustrate the process described in Algorithm 1. First, in Figure 2.1 (a) all the data instances belong to the same partition. A blue and red point are highlighted in the

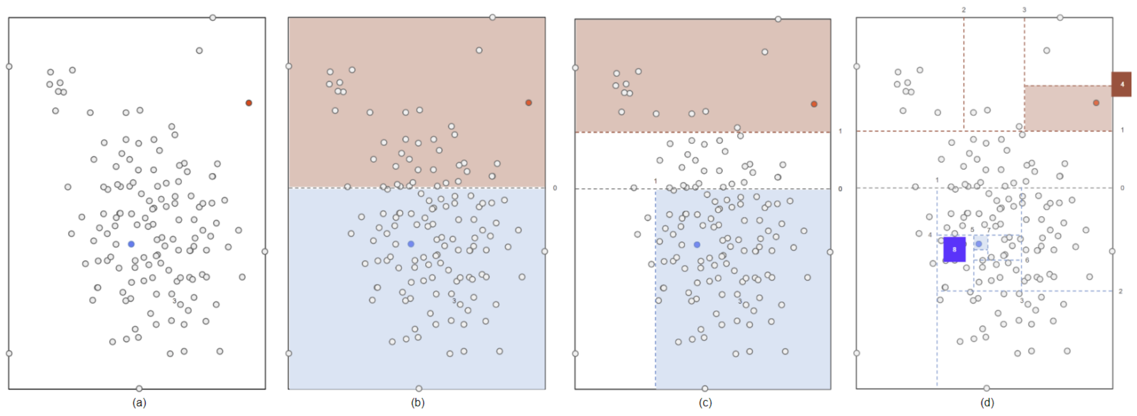

**Figure 2.1:** Example of how an *iTree* is built.

figure, as these will be the focus on the example (only these partitions will be expanded but it is assumed that the partitions that do not contain the points are equally separated until the algorithm is completed for every point). In (b) a random feature (y-axis) is selected and the data is partitioned in two by a random threshold (represented by a dashed line). Then the process is repeated to arrive at step (c) where there are now 4 partitions. The process continues until we arrive to image 2.1 (d) where both points have been isolated. It is important to notice that the red point was isolated in only 4 steps, as it is more separated from the rest of the observations, while the blue point took 8 steps, as it is more centered in the cluster. Note that each partition results in two more partitions, which is why this process can be represented as a binary tree. Using an *iTree*, the path length of a point is the number of edges in the tree it traverses until it reaches an external node. This path length is later used to compute the score using an *iForest*.

To build an *iForest*, $T$ *iTrees* are created from $T$ independent samples of size $\psi$ taken without replacement from the dataset $X$. These trees are used in an ensemble to assign an anomaly score to each instance of the full dataset $X$. The process to determine the anomaly score will be explained in the next subsection.

### 2.2.2 Anomaly score inference for *iForest*

Once the trees are trained, every point in the dataset traverses every tree to obtain the path length $h_t(x)$ of the point $x$ in the *iTree* $t \in \{1, \ldots, T\}$. Then all $T$ path lengths are averaged for each point. After this procedure, a normalization step is performed on the average tree length. Since the structure of each *iTree* is the same as a Binary Search Tree (BST), the average path length of a an unsuccessful search in BST is used to calculate the anomaly score [25]. The average path length is calculated as follows:

$$c(n) = 2H(n-1) - (2(n-1)/n). \tag{2.1}$$

Where $H(i)$ corresponds to the harmonic number, $c(n)$ is the estimate of the average path length $h(x)$ to isolate all $n$ data instances within a node.

Once the estimate of the average path length, $\mathbb{E}(h_t(x))$, of the data instance $x$ over all the *iTrees* $t$ is obtained, the anomaly score $s(x, n)$ is calculated as follows:

$$s(x, n) = 2^{-\frac{\mathbb{E}(h(x))}{c(n)}}. \tag{2.2}$$

With this anomaly score, there are two scenarios:

1. When there are evident anomalies within the data, $s(x, n)$ will be close to 1 when $x$ is an anomaly and $s(x, n)$ will be smaller than 0.5 when $x$ is a normal point.

2. When there are no anomalies within the dataset, $s(x, n)$ will be very similar to 0.5 for all points.

### 2.2.3 Hyperparameter guidelines for *iForest*

To give more insight in the workings of the *iForest* algorithm, we will discuss the different hyperparameters that it uses. Namely the following:

- $\psi$ - **Sub-sampling size**, which controls the training data size. According to [25], increasing the value of this parameter after a certain threshold yields no additional improvement in performance, whilst increasing the training time. The experimentation performed in [25] suggests that using $\psi = 256$ results in a good performance across multiple datasets. The authors mention the importance of this parameter to alleviate *swamping* and *masking* issues. *Swamping* happens when normal instances are classified as anomalies because they are close to anomalies. *Masking* refers to the existence of groups of anomalies, which makes them appear as normal points. In [25] it was found that taking a sub-sample increases accuracy, so the value of $\psi$ should be chosen carefully and according to the dataset.

- $t$ - **Number of *iTrees*** used in the ensemble. According to [25], a value of $t = 100$ is usually enough for a satisfactory detection.

### 2.2.4 Advantages of *iForest*

The main advantages of the algorithm are: first, its focus on finding anomalies rather than profiling normal points, second, its capacity to scale, and finally, its capacity to detect multiple types of anomalies. Below we will explain how the design of the algorithm results in this advantages and what these advantages entail.

#### 2.2.4.1 Focus on anomalies

To start, the focus on anomalies is of great importance. Many existing anomaly detection approaches separate abnormal points from the rest by constructing a profile of normal instances and then identifying the records that do not conform to such a profile as outliers [25]. The disadvantage is that these methods are optimized to profile normal instances and the detection of anomalies comes as a consequence of such profiling. The choice of the method (and intrinsic definition of what a normal point is), can ultimately bias what an anomaly is, as this comes as a secondary task. This might result in a high amount of false positives or the detection of too few anomalies [25]. Examples of such bias are the cluster-based and density-based methods discussed in Section 2.1.1, where the former is more suited for global anomalies while the latter is more effective with local anomalies. Each of the methods might ignore the anomalies detected by the other, hence reducing their detection capabilities. The *iForest* algorithm, on the other hand, is not dependent on a definition of what a normal point is, making it more effective and versatile in the detection anomalies.

#### 2.2.4.2 Efficient use of resources

Another advantage of the algorithm is that it is **scalable**. It has this advantage because of the initial subsampling task, the simple operations used to isolate anomalies, its high capacity of running tasks in parallel and its low memory requirement.

Firstly, subsampling brings the advantage that, since the trees are trained on a smaller subset of the data, their training time is significantly reduced. It is important to highlight that the number of points used for training is determined by the number of trees and the subsampling size, which maintains a constant training time regardless of the size of the original dataset.

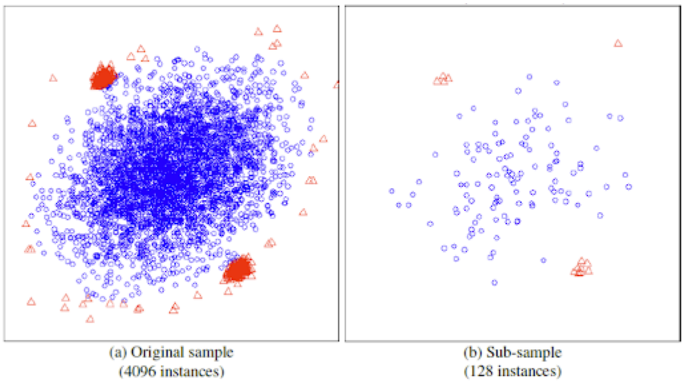

**Figure 2.2:** Subsampling with *Isolation Forest*. Figure taken from [25].

Secondly, another important characteristic of the algorithm is the efficient use of computational resources, resulting from the simplicity of its operations during training. For example, creating each level of an *iTree* (partitioning the data) requires three basic operations: first, finding the minimum and maximum points $a$ and $b$, then, computing a random value $t$ in the interval $[a, b]$ and, finally, doing an inequality comparison of the type $x < t$. None of these operations entail involved mathematical operations. Furthermore, the complexity of the process of building an *iTree* only depends on the size of the data set, which gives it linear time complexity [25]. Thus, it is very fast to run on large datasets.

Thirdly, the model is highly parallelizable for two main reasons: First, comparing the values in a partition to the randomly selected threshold can be performed simultaneously. Second, every tree is independent from the others, so they can be trained and evaluated in parallel.

Finally, the algorithm has a low memory requirement as it only needs to record the attribute used in each partition, the selected threshold and the number of points per partition. During training, because of the subsampling, the algorithm only requires a fraction of the total data and once the training is complete the data used for training is no longer needed.

The combination of the previous characteristics make this algorithm computationally inexpensive and parallelizable which make it **scalable**. This is one of the desired qualities for a model capable of successfully exploiting streaming data.

#### 2.2.4.3 Capacity to detect different types of anomalies

The *iForest* algorithm has the capacity to detect different types of anomalies due to the following three factors [25]: First, its capacity to ignore irrelevant attributes, secondly, its ensemble nature and, finally, its subsampling step.

First, all attributes of the data have the same probability of being chosen to perform the split on. However, if instances are easily isolated by a subset of attributes, these attributes only need to be chosen few times to isolate a point. Less relevant attributes, on the other hand, can be chosen multiple times without contributing much in isolating instances. Since the number of elements in a partition affects the anomaly score (more elements in a partition results in a lower anomaly score), the observations with greater anomaly scores will be influenced more by the attributes that differentiate them the most from the rest. As a result, irrelevant attributes become less important and the algorithm becomes robust to irrelevant attributes and the need of feature selection is reduced.

Second, since the algorithm is an ensemble of *iTrees* trained on different sub-samples of the data, it is very unlikely that multiple types of anomalies will be present in the same sample. This results in *iTrees* that become experts in different types of anomalies and therefore allow the ensemble to detect a broad amount of them.

Thirdly, sub-sampling has a positive impact on the ability to detect anomalies by preventing *swamping* and *masking*. *Swamping* refers to normal instances wrongfully being labeled as anomalies due to their proximity to abnormal points. Contrarily, *masking* occurs when anomalies in a cluster are mistaken as normal points due to the high density of the cluster, although the cluster is isolated from the rest of the data [25]. Figure 2.2 shows the effect of sub-sampling where anomalies are represented in red, and normal points in blue. First, (a) illustrates *swamping*, where normal points (blue) can be close to the anomalies and thus be mistaken as such. In the same figure, *masking* is also present as there are two anomaly clusters present which hide the anomalous nature of their data instances. Figure 2.2(b) on the other hands, illustrates how both *swamping* and *masking* are addressed using sub-sampling. Firstly, the clusters are smaller in size, reducing the masking effect. Additionally, the normal observations are better separated from the anomalies, as the points between them are fewer, making the differences between the normal points and the anomalies more evident. As a result, the final subset is more suitable to train the algorithm.

In summary, the design of the *iForest* algorithm allows it to detect a broad range of anomalies without the need of labeled data. It also requires very little or no feature selection, which is desireable for data sets where there is not time to perform preprocessing. And finally, its low CPU and memory requirements combined with its ensemble nature make the algorithm highly scalable. All these characteristics make *iForest* a good fit for anomaly detection tasks in the context of cyber security.

### 2.2.5 Limitations

Among the main limitations of the *iForest* algorithm are its inability to accommodate categorical variables and the high number of false positives generated. Firstly, since the algorithm relies on selecting thresholds that depend on the minimum and maximum of a certain attribute, the process can only work for numerical ordered data. This reduces its usefulness for some data sets in cyber security where categorical data is present, for example when flags are part of the attributes of the data. Another downside is that, despite the capacity to identify anomalies, the algorithm may still produce a great number of false positives. However, this is common in unsupervised anomaly detection methods [42] and it has been shown that combining unsupervised methods with supervised approaches when possible could ameliorate this issue [5]. Another solution is to incorporate expert feedback when updating the algorithm [35].

## 2.3 Concept drift

Concept drift is one of the main challenges that comes with streaming data [10] and addressing it is of paramount importance as it can turn high performing models into outdated low-accuracy models. In this section we will explain what concept drift is, how it can affect the performance of models and finally what methods exist to overcome the difficulties it poses.

### 2.3.1 Introduction to concept drift

Concept drift refers to how the underlying distribution of the data changes over time. When it occurs, the learnt patterns from past data do not occur in new data, leading to a degradation in the performance of systems that leverage these learnt patterns [14]. Concept drift has been the cause of decreased performance in many information systems such

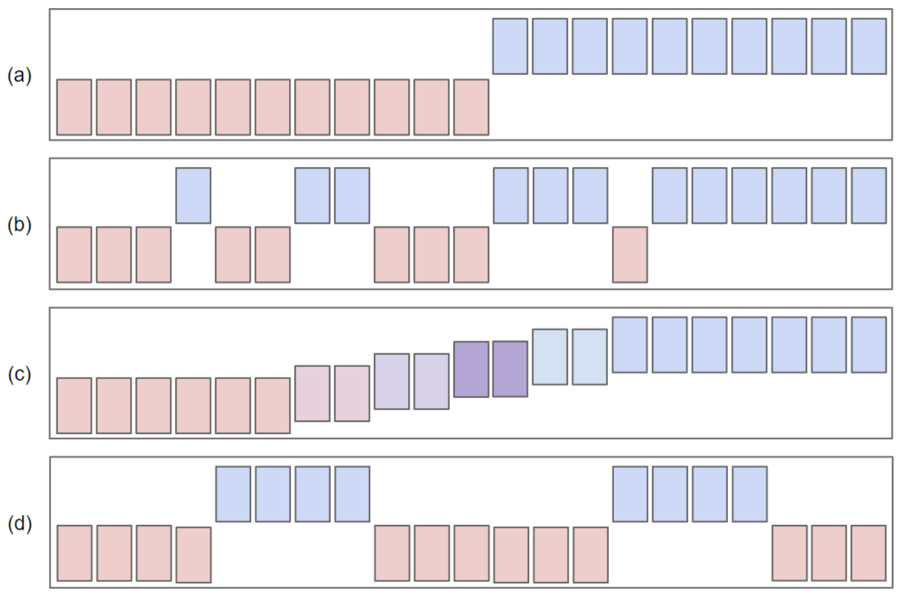

**Figure 2.3:** Types of concept drift [26].

as early warning systems and decision support systems [27]. Addressing this challenge is of great importance to make more reliable data-driven solutions. It is critical to include methods to detect concept drift, so models can be updated over time without their performance being affected.

Concept drift can occur in a number of ways that can be grouped in the following four categories: sudden drift, gradual drift, incremental drift and reoccurring concepts [26]. In sudden drift (Figure 2.3.a), the underlying distribution of the data changes in a short period of time. Meanwhile, gradual concept drift (Figure 2.3.b) occurs when the previous concept is progressively replaced by the new one. Moreover, incremental drift (Figure 2.3.c) is when the previous drift starts to transform into the new one with a smooth transition. Lastly, recurring concepts (Figure 2.3.d) refer to cases when old concepts reoccur after some time. It is important that models are able to detect different kinds of concept drifts in order to be effective. In the next subsection we will go through some methods used to detect concept drift.

### 2.3.2 Concept drift detection

There are a number of ways of achieving concept drift detection, however, most of the existing methods are supervised approaches. For instance, in [21], out of 59 surveyed methods, only 2 constitute unsupervised concept drift detection methods. This poses an issue since, as discussed earlier, labeled data sets are rare in the cyber security sector, so unsupervised techniques must be used to detect concept drift.

Three approaches to address concept drift in an unsupervised way are: assuming that drift exists, tracking a specific statistic within a time window, and measuring the discrepancies between different batches of data. The first approach simply assumes that concept drift exists and periodically updates the models. This approach is followed by [37], where the model is updated with every new batch and is thus kept up to date. Although simple and effective, it can result in excessive training as the models are replaced regardless of the existence of drift and also, patterns learnt by the model could be prematurely discarded.

On the other hand, the statistic tracking approach consists of following specific statistics of the data to determine if there is concept drift or not. The error rate-based drift detection models are examples of this approach [26]. These models work by measuring the error rate of the base learners within a sliding window and if the error rate increases or reduces significantly with respect to previous windows, then it is assumed that concept drift occurred. In [14] and [23], an unsupervised version of this method is leveraged by tracking the average anomaly rate of an anomaly detector per batch and triggering a drift alarm when it is significantly different from other batches. Although effective, this method is not robust for all types of concept drift [19] and relies on the accuracy of the underlying anomaly detector.

Finally, a more robust method that relies on measuring the differences between subsets of data, where samples taken from a reference batch are compared to samples in a future window [24]. The main idea behind this approach is that, although the batch changes, each batch should be statistically similar to the rest when there is no drift. If the aggregation of the dissimilarities between samples is statistically different, the alarm will be triggered. This approach is more sensitive to local drifts and is also application independent, since this partition can be taken from any type of streaming data [17] and is the main idea behind the *NNDVI* algorithm. In 2.5, we will delve into the *NNDVI* algorithm and explain its advantages.

## 2.4 Bilateral-Weighted Online Adaptive Isolation Forest

The Bilateral-Weighted Online Adaptive Isolation Forest (*BWOAIF*) [19] extends the *iForest* algorithm for use on streaming data. Since it uses the *iForest* algorithm as a basis, it inherits its **scalability** and **unsupervised** characteristics. However, there are also some modifications which allow it to **handle streaming data**, **deal with concept drift** and make more **efficient use of the resources**. The *BWOAIF* model leverages the ensemble nature of the *iForest* algorithm to partially update itself by replacing old *iTrees* with new ones trained on the most recent data. Additionally, a weighting scheme is applied during the anomaly score calculation. This results in an ensemble method where the most recent *iTrees* have a stronger influence on the computation of the anomaly score, thus allowing it to deal with concept drift. Below we will explain the algorithm, give some guidelines for parameter tuning and, finally, explain which characteristics result in the mentioned advantages.

### 2.4.1 Training the algorithm

The training of the algorithm is one of the most important changes in the *BWOAIF* algorithm with regards to the *iForest*. Constantly retraining *BWOAIF* is what allows it to address concept drift. It performs this by replacing a portion of the $T$ *iTrees* that conform the ensemble. The *BWOAIF* algorithm performs the partial update by using non-overlapping data batches of size $B$ and replacing the oldest $E$ trees with newly trained $E$ trees each batch. The data is selected from a receptive window of size $W = B \cdot T/E$. This ensures that the data is diverse enough to train the new trees. This way, the size of the ensemble $T$ remains constant but the algorithm is constantly updated with new data. Figure 2.4 illustrates the process of tree replacement as new data arrives. For each batch of data, the steps described in Algorithm 2 are performed to update the ensemble.

### 2.4.2 Anomaly score inference for *BWOAIF*

To allow for a more flexible adaptation to concept drift, the anomaly score inference is also modified. To do so, the *iForest* is segmented in $K$ sets of $E$ trees, where every set corresponds to the timestamp when it was created. Then, the conventional anomaly score

**Algorithm 2** *BWOAIF* algorithm

```
Require: $D = \{d_0, d_1, \ldots, d_{N-1}, d_N\}$ - Data, $B$ - Batch size, $T$ - Total number of trees,
    $E$ - Trees to be updated each batch.
  $W \leftarrow B \cdot T/E$
  $t \leftarrow W$
  Train an iForest $F$ sampling from $D_W = \{d_0, \ldots, d_{W-1}\}$
  Compute the anomaly scores for $D_{t+B} = \{d_t, \ldots, d_{t+B-1}\}$ using the iForest $F$.
  while $t < N$ do:
      $t \leftarrow t + B$
      Delete the oldest $E$ trees from $F$ based on the timestamp $t$.
      Train $E$ new trees sampling from the receptive window $\{d_t, \ldots, d_{\min(t+B-1,N)}\}$.
      Add the $E$ new trees to the ensemble $F$ with timestamp $t$.
      Compute the anomaly scores for $D_{t+B} = \{d_t, \ldots, d_{\min(t+B-1,N)}\}$ with the iForest $F$.
  end while
```

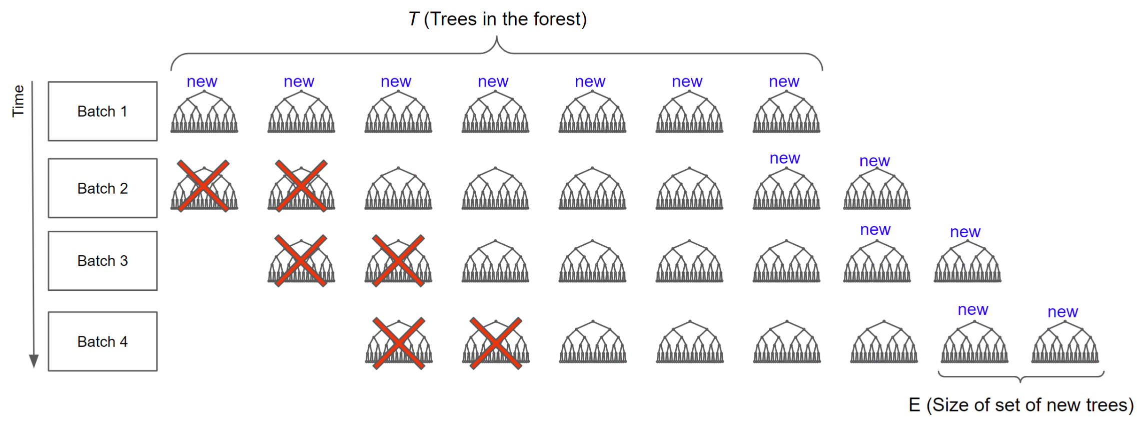

**Figure 2.4:** Update mechanism for the *BWOAIF* algorithm with $T = 7$ trees and $E = 2$ new trees per batch.

is computed over every one of this batches. This results in $K$ anomaly scores $s_k(x^{(t)})$ where $x^{(t)}$ is the sample $t$ of the dataset. Then these are aggregated using bilateral weighting:

$$S(x^{(t)}) = \frac{1}{S}\sum_{k=1}^{K} s_k(x^{(t)}) \cdot e^{-\frac{(s_k(x^{(t)})-s_1(x^{(t)}))^2}{2\sigma_v^2}} e^{-\frac{(k-1)^2}{2\sigma_a^2}}, \tag{2.3}$$

with a normalization constant:

$$S = \sum_{k=1}^{K} e^{-\frac{(s_k(x^{(t)})-s_1(x^{(t)}))^2}{2\sigma_v^2}} e^{-\frac{(k-1)^2}{2\sigma_a^2}}. \tag{2.4}$$

In Equation (2.3), the first exponential term is responsible for reducing the influence of sets of trees which have been built from a very different batch than the most recent ones. Note that this is done by comparing the anomaly scores of the most recent batch ($s_1(x^{(t)})$) and those of the other batches ($s_k(x^{(t)})$). When the difference is very large, the exponential term becomes smaller and hence the influence of batch $k$ in the overall anomaly score is reduced. The parameter $\sigma_v$ controls the effect of this term. On the other hand, the second term is concerned with giving a greater weight to the more recent trees, $k = 1$ being the most recent batch. Similarly, the parameter $\sigma_a$ controls the effect of this term.

### 2.4.3 Hyperparameter guidelines for *BWOAIF*

In [19], a set of guidelines is listed to adequately tune the algorithm. These are summarized below:

- $B$ - **Number of samples in each batch** The size of each batch should be such that the data is approximately stationary. If this information is unavailable, set similar to $\psi$.

- $E$ - **Number of trees** The authors recommend setting it between 100 and 200, but keeping it close to $\psi$.

- $\psi$ - **Sub-sampling size** The authors recommend using similar values as in the *iForest* algorithm (values around 256) while also keeping the value of the parameter similar to the parameter $E$.

- $T$ - **Number of *iTrees*** Should be based on the receptive window $W = B \cdot T/E$. A sufficiently high number of $T$ is advised to provide smooth anomaly scores. With periodic concept drifts, $W$ should cover at least two periods.

- $\sigma_a$ - **Parameter controlling suppression of old trees** Normally, it is safe to set this number high [1].

- $\sigma_v$ - **Parameter punishing dissimilar trees** This parameter should be set according to the dataset, but low values should be avoided because it will give small weights to most of the ensemble, resulting in mostly the $E$ updated trees contributing to the overall score. However, very high values disable the capacity of the model to adapt to sudden changes as newer trees are weighted the same as older ones. It is recommended to choose a value in the range of $[0.05, 0.1]$ so that new trees have enough influence without discarding older trees.

### 2.4.4 Advantages of *BWOAIF*

The algorithm maintains some of the advantages of the *iForest* algorithm, but its modifications give it three additional advantages: Ability to handle **streaming data**, capacity to account for **concept drift** and an **efficient use of the resources**.

#### 2.4.4.1 Ability to handle streaming data

To deal with streaming data, the algorithm must combine the efficient use of the resources with the ability to address concept drift. To do so, it uses diverse mechanisms, starting with the use of a sliding window, then a replacement of the elements of the ensemble and finally a new weighting scheme. Each of these mechanisms contribute to the capacity of the model to exploit streaming data and will be explained below.

First of all, the algorithm uses a sliding window which limits the amount of data that has to be processed. This has the advantage of a lower memory use, compared to algorithms that keep all the data in memory. Additionally, it allows the model to focus only on the most up-to-date data.

Secondly, the replacement of *iTrees* for newly trained ones ensures that at least a portion of them are trained on the most recent data, providing the capacity to account for **concept drift** while using less memory than if all *iTrees* were kept.

Finally, the weighting of the anomaly scores allows the user to determine the speed at which the algorithm adapts to concept drift. This is done by giving more weight to more

[1]The original paper [19] is unclear about the optimal value for this parameter

recent trees when needed, or by giving more weight to trees of the ensemble which make similar predictions to those in the most recent batch. This mechanism allows the model to adjust to different types of concept drifts.

#### 2.4.4.2 Efficient use of the resources

The sliding window and replacement of old trees reduce the memory requirement of the algorithm. However, to account for concept drift the algorithm is constantly being updated, which results in additional CPU requirement. Nevertheless, in [19], it was shown that this constant retraining of the algorithm resulted in higher accuracy and use of fewer trees than other methods such as the *iForestASD*, which incurs in a full retrain when concept drift is detected [14]. This makes *BWOAIF* an algorithm that has a more **efficient use of resources** than other methods, as fewer trees are required for training and only the data in the current window is maintained while the size of the ensemble remains constant.

Summarizing, out of the 6 characteristics listed in 1, *BWOAIF* addresses five, the only missing characteristic being the proportional update. It inherits the scalability and unsupervised characteristics from the *iForest* and introduces the capacity to deal with streaming data and concept drift by using a sliding window, updating the ensemble and using a special weighting scheme when computing anomaly scores. Finally, the sliding window and partial update of the algorithm result in a more efficient use of the resources.

### 2.4.5 Limitations

One of the limitations of this algorithm is that the parameters that determine the speed of the update are set at the beginning of the the execution. While the weighting scheme and replacement of trees help it address different types of concept drift, with sudden drift only a portion of the trees will be trained on the new data. This can be an issue because, even if the obsolete trees are ignored, updated trees might be too few to converge to an adequate score and therefore not enough to leverage the benefits of ensemble models. On the other hand, with small or no concept drift, the algorithm will still perform the same update size, resulting in unnecessary computations. An update scheme that is dependent on the characteristics of the concept drift might be able to improve the performance of the algorithm.

## 2.5 Nearest neighbor-based density variation identification

The Nearest neighbor-based density variation identification (*NNDVI*) algorithm [24] is an unsupervised concept drift detection algorithm. It relies on comparing dissimilarities between partitions taken from different batches of the data and has several advantages: it is robust to high dimensional data, it is sensitive to regional drift, and it can give a measure of how large the concept drift is. The algorithm has two main components: first, the data modeling component builds a representation of the data instances which compares their critical information. Then, a distance function is used to quantify the dissimilarity between two datasets. In the following subsections we will explain both components and then state how the algorithm can be incorporated into the *iForest* algorithm.

### 2.5.1 Data modeling with *NNDVI*

The first step of this model is creating a representation of the data which can be later compared with other datasets. A commonly used method to represent data instances is to group similar data instances into partitions and then use that partitioning scheme to group the other dataset. By comparing the differences between the partitions we can have a dissimilarity measure for the datasets. An example of this approach is comparing histograms, where data instances are grouped in bins and then we can compare the empirical density of two datasets by calculating a dissimilarity measure between the resulting histograms. However, this method has the problem that if the partitions are not chosen correctly, then similar items will be assigned to different partitions and the comparison will not be reliable. To prevent this from occurring, the *Nearest Neighbor-based partitioning schema* (*NNPS*) is proposed in [24].

The *NNPS* assumes that closely located data instances are related to each other. A way of measuring such a relationship is by expanding each data instance into a hypersphere which preserves more information than grouping data instances into partitions. Then the similarity between two instances will be the intersection between their hypershperes. This can be seen in Figure 2.5. In the figure, each point $d_i$ was expanded to a hypershpere and this resulted in three partitions, one corresponding to the intersection ($p_3$), and the other two to the non overlapping sections of the hyperspheres ($p1$ and $p2$). The similarity between both points is calculated as $\frac{|q_3|}{|q_1|+|q_2|+|q_3|}$ which corresponds to the area of the intersection divided by the sum of the area of all three partitions.

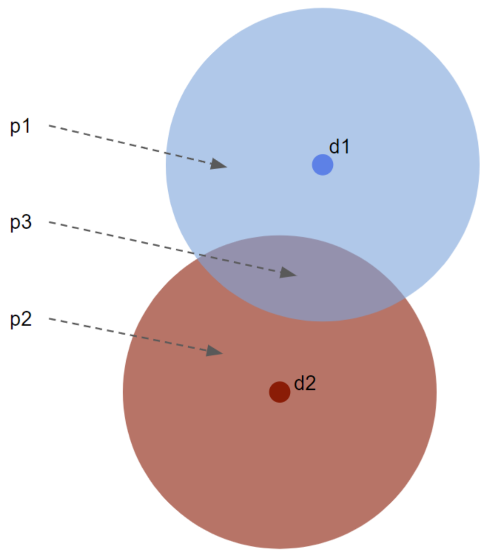

**Figure 2.5:** Example of hyperspheres and their intersection.

This approach is quite straight forward when dealing with two dimensions, but as the number of dimensions increases, so does the difficulty of calculating the intersections between hyperspheres. For this reason the *NNPS* uses the $k$-nearest neighbors of a point to generate its hypershpere and make the algorithm robust for high dimensional data. To generate the hyperspheres, the algorithm generates a *multiset* of particles which can later be used to estimate the difference between two datasets. However, to understand how these *multisets* are formed, we first need to explain the notion of *instance particle* and *instance particle group* which are the elements that form a *multiset*. For this, we will take the dataset shown in Figure 2.6 (a).

*Instance particle.* An instance particle is the expansion of a given point $d_i \in D$, where $D$ is the dataset. This expansion $P_{d_i}$ is the pair $(d_i, K_{d_i})$ where $K_{d_i}$ is the set of neighbors of $d_i$. To help visualize this, Figure 2.6 (b) shows each point of the dataset connected to its 4-nearest neighbors (a point $d_i$ is not considered a neighbor of itself) by a line matching the color of the node. Once the neighbors of each point are established, the *instance particle* of point 1, for example, can be obtained, which is $P_{d_1} = (d_1, \{d_2, d_3, d_5\})$ as shown in Figure 2.6 (c). This can be obtained in a similar fashion for all points of the dataset $D$.

*Instance particle group.* Once the instance particles are defined, we can obtain the *instance particle group* of point $d_i$, $P(d_i)$, which consists of the instance particles $d_j$ which

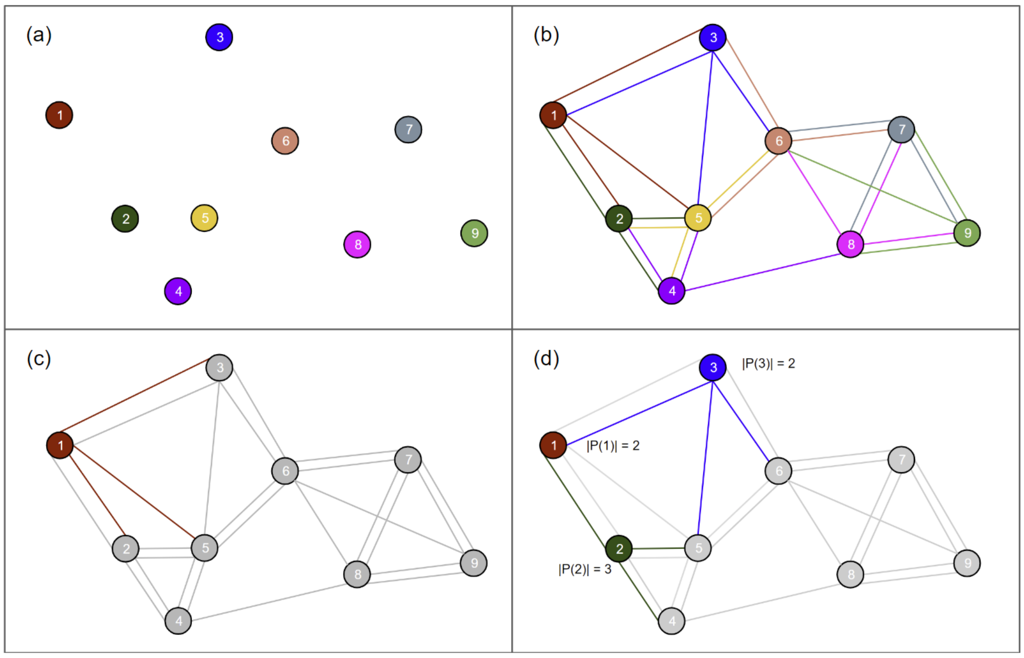

**Figure 2.6:** Example of how a multiset is built to model the data in the *NNDVI* algorithm.

contain $d_i$ in their neighbor set $K_{d_j}$. This is defined as:

$$P(d_i) = \{P_{d_j} \mid d_i \in K_j, j = 1, \dots, |D|, i = 1, \dots, |D|\}. \tag{2.5}$$

Figure 2.6 (d) illustrates an *instance particle group* where $P(d_1) = P_{d_2}, P_{d_3}$ as $d_i$ is one of the nearest neighbors for both $d_2$ and $d_3$.

Now that we have the previous definitions, we can define a *multiset* $M_{d_i}$ of particle $d_i$. This multiset will consist of pairs formed by an instance particle and a weight. It is defined as:

$$M_{d_i} = \{(P_{d_j}, m(P_{d_j})) \mid P_{d_j} \in P(d_i), m(P_{d_j}) = \frac{Q}{|P(d_j)|}\}, \tag{2.6}$$

where $Q$ is the lowest common multiple of $\{|P(d_j)| \; \forall \; d_j \in$D$\}$, which is the set of sizes of the instance particle groups. Figure 2.6 (d) displays an example of the sizes of instance particle groups. Note that with this weighting scheme, we obtain a lower weight when the *instance particle group* is greater.

Now that we have defined a *multiset*, we have the main element used to measure the dissimilarity between two datasets using the distance function which we will define and explain in Section 2.5.2.

### 2.5.2 Distance function

After defining the *multiset*, we have a basic element we can use to compare two datasets. The idea is to use the intersection between two *multisets* to measure their similarity. The distance $\delta_{nnps}$ between two multisets $M_A$ and $M_B$ is defined as:

$$\delta_{nnps}(M_A, M_B) = \frac{1}{|P(M_A) \cup P(M_B)|} \times \sum_{p_{d_i} \in P(A) \cup P(B)} \frac{|\mathbb{I}_{M_A}(P_{d_i}) - \mathbb{I}_{M_B}(P_{d_i})|}{\mathbb{I}_{M_A}(P_{d_i}) + \mathbb{I}_{M_B}(P_{d_i})}. \tag{2.7}$$

Where $\mathbb{I}_S$ is an indicator function which returns the weight of an element if it belongs to a multiset or zero otherwise. Mathematically it is defined as:

$$\mathbb{I}_S(x) = \begin{cases} m_S(x), & \text{if } x \in S \\ 0 & \text{otherwise} \end{cases} \tag{2.8}$$

This distance is a weighted sum of the intersection of the elements within the multisets and will allow us to see how similar two multisets are. A major advantage of this distance is that it follows a normal distribution. This will allow the algorithm to perform a tailored statistical significance test for the distance and with enough evidence, assess the existence of drift. Since the distance is normally distributed, then we can perform a $z$-test with the null hypothesis being that drift is not present. Now that we have explained the data modeling procedure and the distance function we can proceed to explain the algorithm.

### 2.5.3 Using *NNDVI*

The goal of the algorithm is to compare two data sets $D_1$ and $D_2$. However, since the algorithm relies on the calculation of distances, to prevent heavy computations, samples $S_1$ and $S_2$ are taken from the data sets respectively. Then, the particle groups are obtained and we proceed to estimate the distance $\delta$ between the data sets using Equation (2.7). This process is repeated $s$ times with different multisets and an estimation of the distribution of the distance is obtained. With this distribution, we perform a statistical test on $\delta$ using a $Z$-test. If the null hypothesis is rejected, we assume the existence of drift. Algorithm 3 illustrates the steps to complete this process.

### 2.5.4 Advantages of the *NNDVI*

Apart from being an unsupervised algorithm, *NNDVI* has the following three qualities that make it compatible with the *iForest* algorithm: First, it is robust to high dimensional data, second, is sensitive to regional drift, and finally it can give a measure of how large the concept drift is. Below we explain these advantages and mention why they are relevant to combine it with a *iForest* based algorithm.

**Algorithm 3** *NNDVI* algorithm

```
Require: $S_1$ and $S_2$ - Samples of the datasets, $s$ - Number of shuffles of the samples, $\alpha$
  significance level, $k$ number of neighbors.
  $D = S_1 \cup S_2$, joining the samples.
  Obtain the $k$-nearest neighbors for each data instance.
  Build the particles for each data instance.
  Compute the distance $\delta_{nnps}(S_1, S_2)$
  for $i \dots s$ do:
      $S_1', S_2'$ = shuffle($S_1$), shuffle($S_2$)
      Compute the distance $\delta_{nnps}(S_1', S_2')$
      Store the distance $\delta_{nnps}$ in a list $\Delta_{nnps}$
  end for
  Estimate the distribution of $\Delta_{nnps} \sim N(\mu, \sigma^2)$
  Perform the $z$-test
  if drift is detected then
      return True
  else
      return False
  end if
```

#### 2.5.4.1 High dimensionality

The data representation model of *NNDVI* allows it to extract important information about the data without the need of complex calculations which the hypersphere approach would require. Also, this abstraction transforms high dimensional data into more basic structures (the *multisets*) which can be compared to one another with relative ease thanks to the distance function.

This makes the algorithm robust to high dimensional data, which is an important characteristic as this allows it to exploit large datasets and potentially deal with non-relevant features in the data. Additionaly, this makes it compatible with algorithms which already achieve good performance with high dimensional data, for example, the *iForest* algorithm.

#### 2.5.4.2 Sensitive to regional drift

Another important characteristic of the algorithm is its capacity to detect small changes in the characteristics of the data. This is achieved, because the expansion of data instances in the data modeling stage allows *NNDVI* to have increased sensitivity to changes in the distribution [24]. Also, the statistical test allows to not only detect drift, but also to measure to what extent this change is significant.

This was tested empirically in [24], where regional drifts were introduced in a synthetic dataset to test the capacity of the algorithm to detect them. In these experiments, *NNDVI* outperformed the commonly used method, Kullback-Leibler distance, in measuring regional drift. The sensitivity to regional changes allow the algorithm to detect a wider range of drifts faster than other methods, which is a desirable characteristic.

#### 2.5.4.3 Drift measure

Finally, another interesting characteristic is that the *NNDVI* algorithm not only detects drift but is able to produce a measure of how large it is. This is done through the $\delta_{nnps}$ distance which returns a number between 0 and 1.

In conclusion, the *NNDVI* algorithm has characteristics which make it a reliable method to detect different kinds of drift. Additionally, it also has properties that make it compatible with algorithms such as the *iForest*. In Chapter 3 we explain how both algorithms can be used in conjunction to create an algorithm able to address concept drift.

# 3

# Telosian

## 3.1 Telosian

The *Telosian* algorithm we propose in this work, aims to extend the capabilities of the *BWOAIF* algorithm while maintaining its advantages over previous methods. *Telosian* also uses the *iForest* algorithm as core element, thus maintaining the scalability and unsupervised properties. It also uses the update mechanism proposed in [19] for the *BWOAIF*, replacing a fraction of the total trees. This allows it to handle concept drift and use the resources in a more efficient manner as it avoids a full retrain. However, the *BWOAIF* algorithm uses an incremental and progressive update, which is not optimized for regional concept drift [24]. To solve this, the *Telosian* algorithm incorporates the *NNDVI* algorithm to measure concept drift in every new batch and perform an update to the algorithm proportional to the amount of drift. This has two main advantages: first, it optimizes the algorithm for regional drifts and second, it only trains the necessary trees for every batch. Below we will explain the algorithm an delve into the expected advantages.

## 3.2 Training *Telosian*

The training of *Telosian* follows the same logic as the *BWOAIF* algorithm, replacing a portion of the ensemble to update the algorithm. However, unlike *BWOAIF*, the size of the update performed by *Telosian* is not constant and depends on the amount concept drift instead. To perform this update, *Telosian* considers non overlapping data batches of size $B$ and replaces the oldest $E$ trees with newly trained trees each batch, keeping the size of the ensemble $T$ constant. The number of trees to be replaced is determined by the concept drift score $\nu \in [0, 1]$ every batch, by using the *NNDVI* algorithm. The drift score of the batch is used as input for a function $\tau(\nu) \in [0, T]$ that determines the number of trees to be updated. The *receptive window* is defined as $W = B \cdot T/E$ analogous to *BWOAIF* (Section 2.4). Algorithm 4 describes the process in more detail.

**Algorithm 4** *Telosian* algorithm

```
Require: $D = \{d_0, d_1, \dots, d_{N-1}, d_N\}$ - Data, $B$ - Batch size, $T$ - Total number of trees,
  $E_0$ - Trees to be updated in the first batch.
  $W \leftarrow B \cdot T/E$
  $t \leftarrow W$
  Train an iForest $F$ sampling from $D_W = \{d_0, \dots, d_{W-1}\}$
  Train the NNDVI algorithm sampling from $\{d_t, \dots, d_{\min(t+B-1,N)}\}$.
  Compute the anomaly scores for $D_{t+B} = \{d_t, \dots, d_{t+B-1}\}$ using the iForest $F$.
  while $t < N$ do:
    if $t \neq W$ then
      Estimate the concept drift $\nu$ using the NNDVI algorithm on the window
  $\{d_t, \dots, d_{\min(t+B-1,N)}\}$.
      Update $E = \tau(\nu)$ according to the drift score.
    else
      $E \leftarrow E_0$
    end if
    $t \leftarrow t + B$
    Delete the oldest $E$ trees from $F$ based on the timestamp $t$.
    Train $E$ new trees sampling from the receptive window $\{d_t, \dots, d_{\min(t+B-1,N)}\}$.
    Add the $E$ new trees to the ensemble $F$ with timestamp $t$.
    Compute the anomaly scores for $D_{t+B} = \{d_t, \dots, d_{\min(t+B-1,N)}\}$ with the iForest $F$.
  end while
```

### 3.2.1 Tree update function for *Telosian*

To obtain the number of trees to be updated, we defined a special function. The goal of the function is to perform small changes when the concept drift is small, while making very large changes when it is high. The function takes the drift score $\nu$ as argument. The total number of trees $T$ is also used but is considered a constant as it remains unchanged throughout the execution of the algorithm. An initial approach was to simply multiply the total number of trees by $\nu$ to get $E$. However, this approach resulted in a really noisy $E$ and a great number of trained trees. A step function was chosen instead, as similar concept drifts resulted in the same number of trees. This also prioritized small changes as

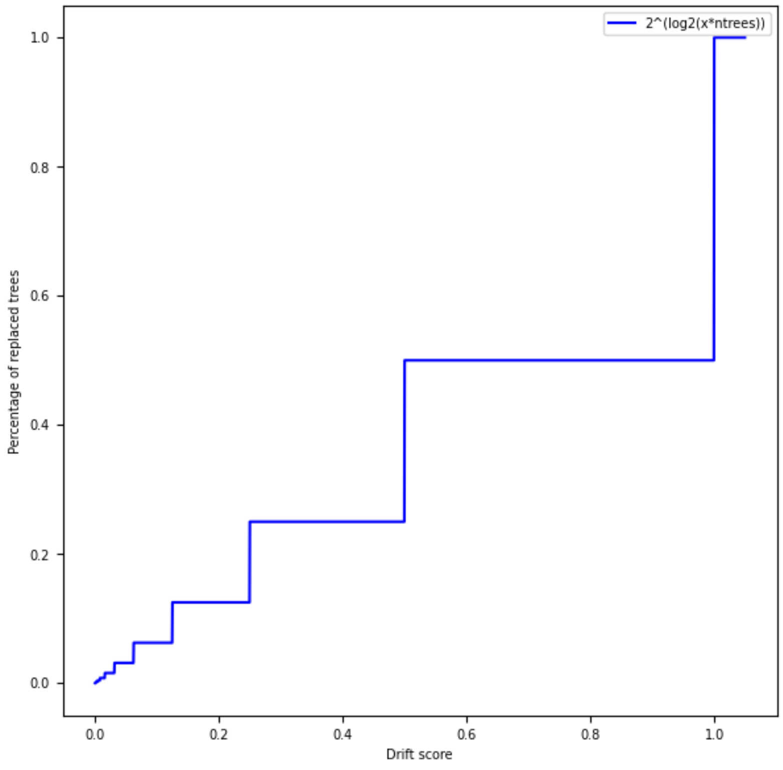

**Figure 3.1:** Function to determine the number of trees to be replaced according to the level of concept drift.

a full retrain is not only expensive but also means that the algorithm will discard many of the already trained regressors. This resulted in a function that will change at most half of the trees, but that requires only small concept drift values to change a portion of the trees so the algorithm constantly updates itself. The resulting function is the following:

$$\tau(\nu) = 2^{\lfloor \log_2 (\nu T) \rfloor} \tag{3.1}$$

### 3.2.2 Hyperparameters for *Telosian*

This algorithm uses similar parameters as the *BWOAIF* algorithm. Nevertheless, some of them are used in a slightly different manner which is specified below.

- $B$ - **Number of samples in each batch** This parameter also determines the size of the reference window to estimate concept drift using *NNDVI*.
- $\psi$ - **Sub-sampling size** This parameter determines the number of observations used to train each tree but also the samples taken from the reference window to determine

concept drift in each batch.

- $T$ - **Number of *iTrees***

- $E_0$ - **Initial number of new trees** This parameter replaced the parameter $E$ from *BWOAIF* and is only used on the first batch as $E$ is determined according to the concept drift score after the first batch.

- $\sigma_a$ - **Parameter controlling suppression of old trees**. This parameter should be set as described in Section 2.4.

- $\sigma_v$ - **Parameter punishing dissimilar trees**. This parameter should be set as described in Section 2.4.

### 3.2.3 Advantages of *Telosian*

The algorithm keeps the advantages explained earlier for the *iForest* and *BWOAIF* algorithms, but is more robust in its response to concept drift thanks to the addition of the *NNDVI* algorithm. From this we expect two main advantages: First, the performance of the algorithm should improve in comparison to *BWOAIF* and *iForest* as it should respond to concept drift in a more effective manner. Second, in most cases we expect the number of trained trees to be lower than that of *BWOAIF* as it only uses the necessary amount of trees instead of a fixed quantity. In Chapter 4 we will test the algorithm to see if this expectations are met and to have a better understanding of the algorithm.

# 4

# Experimentation

In this section, the *Telosian* algorithm for anomaly detection in streaming data with concept drift was tested. For comparison purposes, our own implementation of the *BWOAIF* and *iForest* algorithms were also tested. For the experimentation, several data sets were selected. These vary in the sector, size, number of attributes and amount of concept drift. This allowed us to test the model under various circumstances. We start by describing the data that was used for the experiment, followed by a description of how the experimentation was performed, a short description of the implementation of the algorithms and finally the main metrics that were recorded during the execution process. This information will allow us to obtain the results that will be analyzed in Chapter 5

## 4.1 Data

The data on which the algorithms is tested is of great relevance. Selecting non-relevant data sets would result in misleading results of the algorithm. For example, in [14], the proposed algorithm was designed for concept drift but was only tested on static data sets, which did not allow to test the true capacities of the algorithm. Additionally, it is important to allow other researchers to compare the existing results. For that reason the algorithms were also tested on commonly used anomaly detection data sets for comparison purposes. Furthermore, the algorithm has some limitations that prevent it from being used in some kinds of data sets, for instance, those with predominantly categorical variables. In this section we will list and describe some of the most important characteristics of the chosen data sets and mention other datasets that were considered but ultimately discarded for their incompatibility with the current research.

| Dataset | N | Features | Anomalies | Drift score | Source |
|---|---|---|---|---|---|
| SMD | 1,416,825 | 38 | 2.08% | 0.73 | [20] |
| Thyroid disease | 72,000 | 21 | 7.42% | 0.18 | [34] |
| Bank marketing | 41,188 | 62 | 11.27% | 0.13 | [29] |
| KDDCUP Http | 567,498 | 3 | 0.39% | 0.28 | [1] |
| KDDCUP Smtp | 95,156 | 3 | 0.03% | 0.17 | [1] |
| Shuttle | 49,097 | 8 | 7.15% | 0.13 | [4] |

**Table 4.1:** Summary of datasets used for the experimentation.

A total of 6 data sets were used. They all share some characteristics such as class imbalance and having only numeric attributes. Nevertheless, the sector which they belong to, size, number of attributes and their concept drift varies. This is desirable as it allows to test the algorithms under different circumstances. Table 4.1 lists the chosen data sets along with some of their main characteristics. Below, we will further describe each data set and state the motivation to include it in the experiment.

**Server Machines Data.** (`SMD`) The dataset corresponds to the cyber security sector and is comprised of 5 weeks of data collected from servers from an undisclosed large internet company. This dataset is comprised of 28 subsets which were obtained from 28 different machines. These subsets are to be trained and tested separately as described in [20]. This is the main dataset used for this research as it contains time dependent real world cyber security data with various types of concept drift, making it a suitable dataset to test the algorithms. This dataset can be found in [36] and it contains 38 features, a total size of 1,416,825 observations (about 47,000 for every subset) and an anomaly ratio of 2.08%. No preprocessing steps were required for this dataset.

**Thyroid Disease.** (`annthyroid`) This dataset corresponds to the medical sector and describes patients where some have hypothyroid and the rest (majority class) do not. The dataset can be downloaded from [34]. It was included to allow future research to compare the performance of the algorithm with models proposed in the future. It is comprised of 21 attributes, 7,200 patients (data instances), anomaly ratio of 7.42% and no significant concept drift. No preprocessing steps were used for this dataset.

**Bank Marketing.** (`bank-additional`) This dataset describes clients of a Portuguese bank who were contacted by phone to subscribe to a term deposit. The clients who subscribed to the product conform the minority class and the rest did not subscribe to the product. It was included to test the algorithms in a frequently cited dataset that can be used as benchmark. The dataset is available in [29] and consists of 62 features, 41,188 data instances and 11.27% anomaly ratio. This dataset does not have a significant data drift. No preprocessing tasks were needed for this dataset.

**KDDCUP'99.** This dataset consists of connections being made in a simulated military network. The anomalies are attacks or intrusions to the network. This data set was included because it is a common benchmark for anomaly detection related to cyber security, which makes it highly relevant for this research. The data can be downloaded from [1]. Only a subset was taken and then further divided as described in [41]. This dataset was also used to reproduce the results presented in [25] and [19] and hence given the same preprocessing steps. For preprocessing, the main steps were: First, select only those records with a value of 1 for the `logged_in` attribute. After that, all records flagged as `normal.` were labeled as normal points and all those with a different label were labeled as anomalies. Then, two subsets of the dataset were taken: The first being what we will call the *Http* dataset (`kddcup_http`) which had the `http` value for the `service` attribute and, secondly, the *Smtp* dataset (`kddcup_smtp`) which had the `smpt` value for the same attribute. Finally, for both data sets, only the following four attributes were kept (`duration`, `src_bytes`, `dst_bytes`, `flag`). This resulted in the *Http* dataset having 567,498 records of which 0.39% represented anomalies, and the *Smtp* dataset having 95,156 records of which 0.03% represent anomalies. Both data sets are also static and therefore have no concept drift.

**Shuttle.** (`shuttle_formatted`) This dataset belongs to the physical science sector and describes the radiator positions of a NASA space shuttle. The main reason of including this dataset was reproducing the results shown in [25] and hence we performed the same preprocessing steps: First, class 1 was transformed into class 0 indicating normal points. After that, the classes 2,3,5,6 and 7 were joined into a single class with the 1 flag indicating that those records are anomalies. This resulted in a dataset consisting of eight features, 49,097 records and an anomaly ratio 7.15%. This is a static dataset and therefore there is no concept drift.

### 4.1.1 Other data sets

In this subsection we quickly mention other data sets that were considered but ultimately not used for this research. The main goal of this subsection is to mention some of the expected difficulties when working with these data sets, but also to encourage other researchers to address some of the limitations of the algorithm.

**UGR'16.** This data set was designed to allow algorithms to train on a data set that considers the cyclostationary nature of traffic data and also has the advantage that it captures many different profiles of clients rather than focusing on, for example, a university network [28]. It is a collection of network traces for a time frame of 4 months. Although it has characteristics like its extended duration and variety of profiles and attacks included that make it attractive to the cyber security sector, this data set was discarded for this research. The reason for this was that its features are predominantly categorical which cannot be leveraged by the *iForest* algorithm to accurately detect anomalies.

**LITNET-2020.** The data set consists of real-world data taken from an academic network with 12 types of attacks over a period of 10 months [13]. The extension of the period and range of attacks makes it a suitable to train anomaly detection models. However, only a small portion of the features are continuous (such as received and sent packages and bytes, or time duration of the transmission). Unfortunately, these features did not provide enough information for the *iForest* algorithm to correctly detect anomalies. Hence, it was not used in this work.

**UWF-ZeekData22.** This dataset is one of the most recent cyber security data sets at the moment of writing of this work. It is based in the MITRE ATT&CK framework, was generated using real-world adversarial techniques and can be used to detect the adversary behaviour that leads to an attack [9]. Although this data set has characteristics that make it a good fit for this work (such as the large amount of records, its real-world data and updated examples of attacks) it was still being developed during our research process. Nevertheless, we thank the authors who were in contact with us and helped us in further understanding the data set. When the data set becomes available, it could be of great value to test the algorithm proposed in this research.

The aforementioned data sets have desireable characteristics that are beneficial to studying anomaly detection in the cyber security context. Nevertheless, some of their characteristics prevent them from being used with the algorithm proposed in this work. The ideal data set to test our algorithm would be one that has the following characteristics:

- Have predominantly continuous features.
- Contain data taken over an extended period of time (so that concept drift is present).
- Include real-world data traffic.
- Have a wide range of up to date attacks.
- Be labeled so that the accuracy of the algorithm can be assessed.
- Contain data instances taken in chronological order and with low latency (to simulate real-time).
- Include a large number of data instances (so we can test the scalability of the algorithm).

A data set with these characteristics would be ideal to test the real capabilities of the algorithm proposed in this research. However, to the best of our knowledge, there is not a public data set fitting this description, available at the moment.

## 4.2 Implementation

In this section, we will discuss the implementation of the three algorithms used for the experiments. First, we will describe the main components of the implementation, then we will proceed to explain how it was ensured that the implementation was correct and finally summarize the hardware specifications.

### 4.2.1 Main components of the implementation

The three algorithms have a few components in common. For this reason, it was decided to develop our own implementation of each of the algorithms so that the common components could be shared between all three algorithms. Figure 4.1 shows the relationship between the main components of the algorithm. Note that the arrows do not represent a sequence but dependence on the components used for each stage.

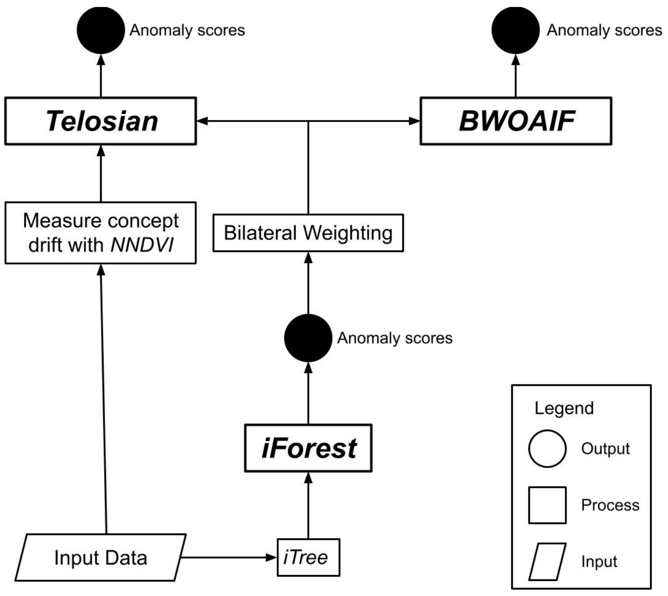

**Figure 4.1:** Graph showing the main components of the implementation and their relationship.

We will explain briefly each of these components and how they were implemented in `Python`. The code base is available in the repository [3].

***iTree*** This first component was defined as a `class` and its goal is to generate the *iTree* from a subset of the data. Additionally, once the tree is generated (trained) it can be used to obtain an anomaly score for each data instance.

***iForest*** This component is also a `class` whose instances consist of a list of *iTrees* and implements the *iForest* algorithm. The *iForest* class generates multiple *iTrees* and aggregates their anomaly scores to compute a global anomaly score, which is the final output of the *iForest* algorithm. It is important to mention that the tree generation and computing on the scores were implemented to be run in parallel and leverage the multiple cores of the hardware. Since this component is used by all three algorithms, they inherit the parallelism.

**Bilateral weighting** The bilateral weighting is a `function` which takes as input the anomaly scores generated by the *iForest* algorithm as well as the parameters mentioned in Algorithm 2. This function is used for both, the *Telosian* and *BWAOIF* algorithms.

**Measuring concept drift with *NNDVI*** The component that measures drift is only used by *Telosian* and it is what differentiates it from the other two algorithms. This component is a function which takes as input the training data and returns a score which will be later used to determine the size of the update for the *Telosian* algorithm. The amount of concept drift is computed by the *NNDVI* algorithm. In this case, since the algorithm was already implemented and the code base available publicly in [2], there was no need for a full implementation.

***Telosian*** This component is in charge of performing the update scheme necessary for the *Telosian* algorithm. It uses the *iForest* instance and updates it according to the output obtained from the measuring of the concept drift. It uses the bilateral weighting function to generate the final anomaly score of the algorithm.

***BWOAIF*** This component updates the *iForest* instance according to the update scheme of the *BWOAIF* algorithm. It also uses the bilateral weighting function to generate the final anomaly score.

### 4.2.2 Quality control of the implementation

To ensure that the algorithm were correctly implemented, they were ran on benchmark data sets to replicate published results. Note that this was only done for the *iForest* algorithm since the other two algorithms do not have comparable published results as they are our own implementation. Nevertheless, in the case of the *BWOAIF* algorithm, we were able to compare our own `Python` implementation with the original `Cython` implementation used in [19]. This was thanks to the authors who proposed *BWOAIF*, whom provided us with their code. It must be noted that their code base was not used for our implementation (we developed our own code base for the *BWOAIF* algorithm). However, we used their implementation to make sure that the results obtained by both, the original `Cython` and our own `Python` implementation, obtained the same results. Furthermore, as additional quality checks, it was tested if the algorithms followed the expected theoretical behaviour which consisted mainly in the following points:

- Have anomaly scores around 0.5. The algorithms were expected to have an average anomaly score close to 0.5 as described in Section 2.2. This condition was met by all three algorithms.

- The anomaly score stayed within the defined limits. The anomaly scores are a value between 0 and 1 where 0 is the least anomalous point and 1 the most anomalous point. All algorithms produced values in this range.
- The path lengths of the trees stayed within the theoretical limits. The theoretical limit of the path length of a tree is known and all three algorithms were tested to ensure that the produced values stayed within this limits.

Both the *Telosian* and *BWOAIF* algorithms use the *iForest* implementation internally and share the same weighting scheme, which makes them comparable as they are built from the same components.

### 4.2.3 Hardware specifications

All the experiments were ran in the same server. The hardware specification of the server are summarized in Table 4.2.

| | |
|---|---|
| **Memory** | 256GB |
| **Processor** | Intel(R) Xeon(R) Gold 6244 CPU @ 3.60GHz |
| **Cores** | 32 |

**Table 4.2:** Hardware used for the experiments.

## 4.3 Algorithm comparison design

To compare the performance of all the different algorithms in equivalent conditions, the following experiment was performed. First, the user determines the following parameters as a `json` file located in the repository [3] on the route `conf/experiment_config.json`:

- `data sets` (`list`): The data sets on which the algorithms will be tested.
- `psi` (`list`): The different values of subsampling which will be used. The chosen values were {128, 256, 512, 1024}.
- `ntrees` (`list`): The different values for the number of trees. The values used for the experiment are: {128, 256, 512, 1024, 2048}.
- `n_new_trees` (`list`): The different values for the number of new trees. The values used for the experiment are: {128, 256, 512, 1024}.

- `batch_size` (`list`): The batch size to be used for the streaming process. The batch sizes were selected from the following values: {512, 1024, 2048, 4096}.

For the weighting of the anomaly scores of the *BWOAIF* and *Telosian* algorithms, the values for the weighting parameters are $\sigma_a = 1000$ and $\sigma_v = 0.05$ following the best practices stated in [19].

This resulted in a total of `|data sets| * |psi| * |ntrees| * |algorithms|` parameter combinations. Nevertheless, some of them are not feasible (for instance those where the number of new trees exceed the the total number of trees) and therefore were discarded, leaving us with a total of 17,824 combinations. The different configurations were used to perform a grid search to find the best combination of parameters for every algorithm so they can be compared later. Each dataset was divided into train (50%) and test (50%) sets. First, the initial ensemble was trained over the whole train set and then tested on batches taken from the test set. In the case of the *iForest*, the ensemble remains the same during the execution, while for the other algorithms every batch was added to the sliding window and used for training after the predictions on the test set were computed. This way the test data remains unseen to the algorithm during the anomaly score computation phase.

The information recorded during each execution was:

1. AUC. The area under the Receiver Operating Characteristics curve (hereafter AUC), is a metric often used for imbalanced classification problems. The Receiver Operating Characteristics (ROC) curve is obtained by plotting the False Positive rate against the True Positive Rate resulting from a classifier. Each of the points of the curve is associated with a classification threshold [8]. The area under the ROC curve can be interpreted as the probability that a random chosen positive instance will be ranked higher than a negative instance [15], so values closer to 1 are preferred while values closer to 0.5 correspond to a classifier that chooses randomly. This metric has proven to be more consistent than accuracy and makes it easier to discriminate between classifiers [8]. Additionally, it has been widely used for *iForest*-related algorithms (for example [25], [19], [14]). Hence, it was chosen as the main performance metric for this research. It was computed over the whole stream and also individually for every batch.

2. Number of updated trees. For every batch, the number of updated trees is also computed. For the *iForest* and *BWOAIF* algorithms the value remains constant, but for the *Telosian* algorithm it will be determined depending on the concept drift.

3. Minimum, maximum and mean anomaly score. For every batch, the total anomaly scores are aggregated into the minimum, maximum and mean.

4. Execution time. For every execution the computation time was also recorded.

Once all the executions were completed, the results were analyzed. The main takeaways from this process are presented in Chapter 5.

# 5

# Results

In this section we will comment on the performance of the algorithms during the experimental phase described in Chapter 4. First, we will present the overall results obtained by the algorithms *iForest*, *BWOAIF* and *Telosian*. After that, we will analyze the effect of each parameter and understand the effects it has on each algorithm across the different datasets. Also, we will compare *BWOAIF* and *Telosian* in a more detailed manner. Finally, we review the execution time per batch to assess if the algorithm can be used in a real-time context.

## 5.1 Overall results

Table 5.1 shows the best Area Under the ROC curve obtained after performing grid-search for hyper-parameter tuning for all algorithms and datasets. The highest score for each dataset is indicated in bold. In the case of the *Telosian* and *BWOAIF* algorithms, the total number of retrained trees is included (for *iForest* no trees were retrained). For each dataset, the average drift score per batch is also shown.

From Table 5.1 we can make three main observations: First, *BWOAIF* obtained the highest AUC score across most of the different datasets. Second, *Telosian* uses fewer trees in all but one of the datasets. Finally, the greatest difference between *iForest* and the other algorithms is present when the concept drift is the highest. Below we comment on why this behaviour occurs.

First, *BWOAIF* has the highest scores but with little difference with respect to *Telosian*. This shows that there is not necessarily a great advantage at using either of the concept

| **Dataset** | **Drift score** | **AUC** | | | **Trees** | |
|---|---|---|---|---|---|---|
| | | *iForest* | *BWOAIF* | *Telosian* | *BWOAIF* | *Telosian* |
| SMD | 0.730 | 0.804 | **0.865** | 0.862 | 169,856 | 122,288 |
| annthyroid | 0.177 | 0.656 | 0.653 | **0.669** | 1,920 | 224 |
| bank-additional | 0.128 | 0.714 | **0.721** | 0.719 | 5,376 | 3,200 |
| kddcup_http | 0.284 | **1.00** | 0.997 | 0.997 | 17,792 | 13,184 |
| kddcup_smtp | 0.173 | 0.869 | 0.900 | **0.904** | 1,536 | 1,792 |
| shuttle_formatted | 0.127 | 0.995 | 0.995 | 0.995 | 6,144 | 2,432 |

**Table 5.1:** Summary of the results of the three algorithms on all datasets.

drift adapted algorithms (*Telosian* and *BWOAIF*). Nevertheless, they both seem to outperform the *iForest* algorithm even when the concept drift is small. This gives an important advantage as it suggests that both algorithms would be effective in static and in changing streams.

Second, in most cases, the *Telosian* algorithm has a lower number of trees, with the difference being greater when the concept drift is small. This shows that *Telosian* requires fewer trees than *BWOAIF* to achieve similar results. However, the *iForest* algorithm has similar performance in some of the data sets without the need to retrain any of the trees, which makes it even more efficient.

Third, the *Telosian* and *BWOAIF* algorithms both show significant increase in performance on the *SMD* dataset which has the greatest concept drift. This is an expected behaviour as they are able to adapt to changes in the data and gives them a significant advantage over *iForest.*

Concluding the initial remarks, the concept drift adapted algorithms achieve a similar or better performance than *iForest* on static datasets and a significant superiority in the dataset with greater drift, which makes them better choices if the drift of the dataset is not known. Furthermore, *Telosian* has comparable performance to the *BWOAIF* algorithm while being more efficient regarding the number of trees updated, which makes it a good choice as we obtain the same results with fewer resources. Nevertheless these conclusions are drawn from very general results. In the next section we will do a more detailed analysis of these results by analyzing specific data sets and parameters to see if these general observations hold up in more particular scenarios.

## 5.2 Effect of the hyperparameters

In this section we will focus on how different parameters influence the performance of the three algorithms. To compare, for each parameter we found the value that resulted in the highest AUC and investigated if the behaviour was consistent among the different datasets. The purpose of this section is to explain the found trends, explain why some cases deviate from this behaviour and finally give some guidelines to set the parameters based on the observed trends. In each case, we selected 4 figures that are representative of the different behaviours that resulted from testing different values for each parameter. Each case will be discussed within each subsection. Note that guidelines for the *iForest* algorithm are not in the scope in this paper as they have already been studied in [25].

### 5.2.1 Effect of the number of total trees

The parameter $T$ determines the size of the ensemble, but also has an effect on the size of the sliding window. Additionally, in the case of the *Telosian* algorithm, the number of updated trees per batch ($E$) is determined by the value of this parameter in conjunction with the measured concept drift. In the experiments, *Telosian* and *BWOAIF* show similar behaviour as the original *iForest* algorithm, where the AUC starts to converge after around 500 trees [25]. In most cases, increasing the number of trees resulted in an increase in performance for the *BWOAIF* algorithm. For the *Telosian* algorithm, on the contrary, keeping a smaller number of trees seemed more convenient, as more trees did not improve the AUC significantly but increased the use of resources. This can be seen in Figure 5.1. Nevertheless, *Telosian* seems to be more robust to changes in this parameter than *BWOAIF* as the changes are smaller. This might be due to the fact that increasing the number of trees does not determine the portion of the ensemble that is replaced for *Telosian*, while for *BWOAIF* algorithm it has an impact on the portion of updated trees. Additionally, the fact that *Telosian* converges earlier could explain why the overall number of trees is generally smaller for this algorithm in comparison to *BWOAIF*. For this reason, we advise to use values slightly below 500 for *Telosian* and use values closer to 2000 for the *BWOAIF* algorithm. Nevertheless, further analysis should be done to see the effects of further increasing the number of trees.

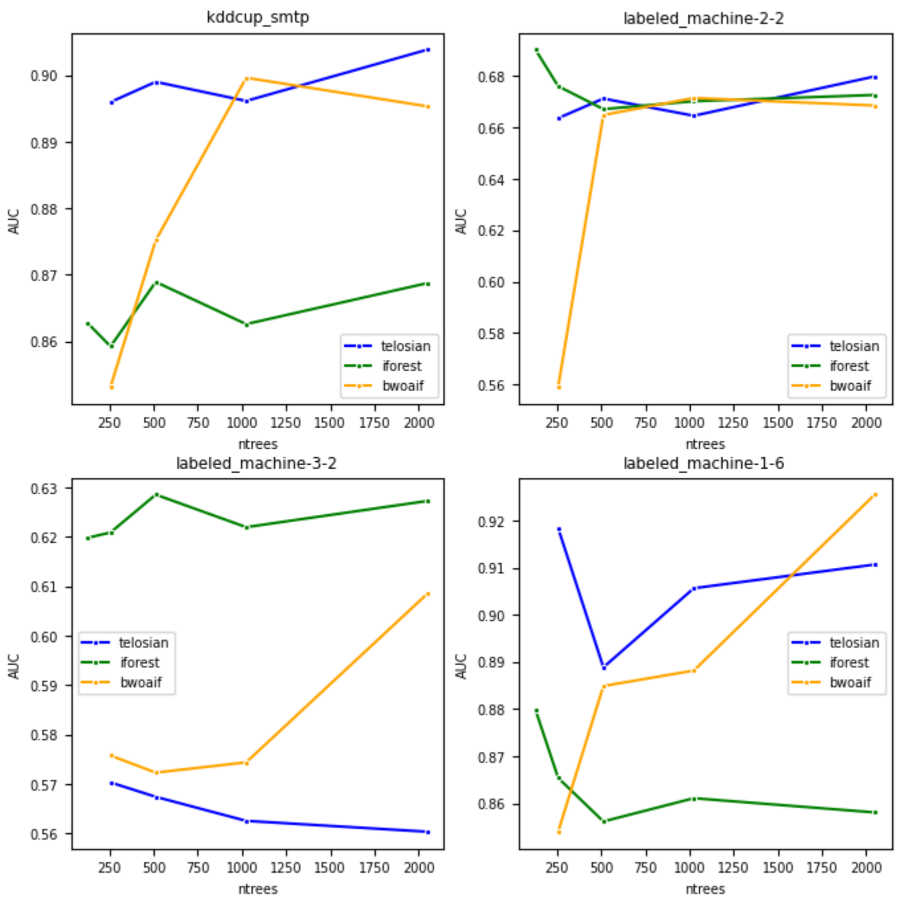

**Figure 5.1:** The AUC obtained with different values for $T$, the total number of trees.

### 5.2.2 Effect of the retrained trees per batch

The number of new trees $E$, determines which portion of the ensemble will be updated each batch. This parameter is not part of the *iForest* algorithm and for the *Telosian* algorithm, it is only used for the initial batch so it is expected that it will have a smaller influence on *Telosian* than on *BWOAIF*. The plots on the left side of Figure 5.2 illustrate how $E$ has a greater impact on *BWOAIF* than *Telosian*. For *BWOAIF*, increasing the trees to be replaced degraded performance, while for *Telosian* it resulted in either small changes or an evident decrease like the one showed in the top right corner of fig. 5.2. Finally, on the right bottom plot, we see a greater decrease in the performance for *Telosian* when the dataset has low concept drift. This is most likely influenced because the initial batch was trained on 50% of the data, and initially replacing a large number of trees results in discarding trees trained on a bigger portion of the data. Therefore, in the case of the *BWOAIF* algorithm, a small number of updated trees is preferred, while for the *Telosian* algorithm, the influence of the parameter is less, but values between 200 and 600 result in better performance.

### 5.2.3 Effect of the subsampling size

The subsampling size $\psi$, determines how many observations will be sampled to train each *iTree* in all three algorithms. However, in the case of *Telosian*, it serves a second purpose, which is the number of points that will be sampled each batch to determine the amount of concept drift. In most cases, a greater value of $\psi$ resulted in an improvement of the algorithm. However, the improvement on the AUC score is smaller as $\psi$ is increased. This is evident in the top plots of Figure 5.3. However, it must be noted that on the right lower plot, there is a decrease in performance as the value of $\psi$ is greater. This also occurred in other static datasets with a smaller number of features. In this cases, a large subsampling size can have a negative impact as it reduces the capacity of the algorithms to deal with *masking* and *swamping* (explained in Section 2.2). On the left lower plot we see this behaviour but we can appreciate a significantly greater impact on the *iForest* algorithm. This suggests that updating the trees might make the concept drift adapted algorithms more robust to the value of $\psi$. An explanation for this is that the weighting scheme incorporated in *Telosian* and *BWOAIF* gives more importance to trees that have similar anomaly scores to trees trained using the latest batch. It is also worth noticing that the subsampling size is directly related to the height of the trees, so a small

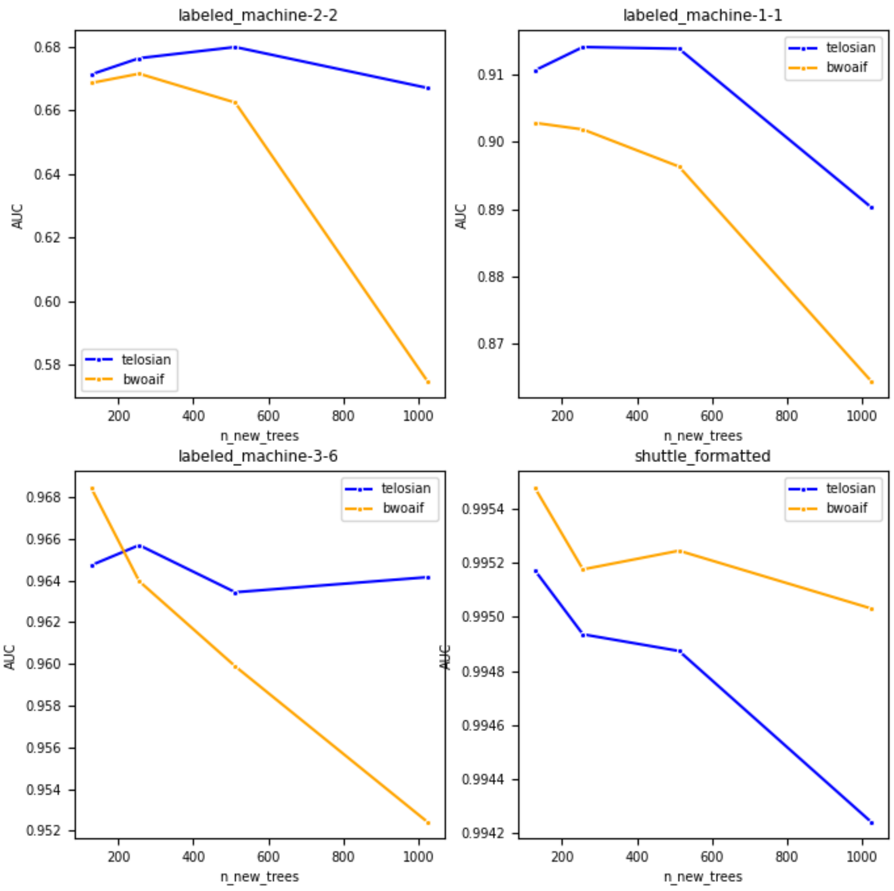

**Figure 5.2:** The AUC obtained after testing different values for $E$, the number of retrained trees.

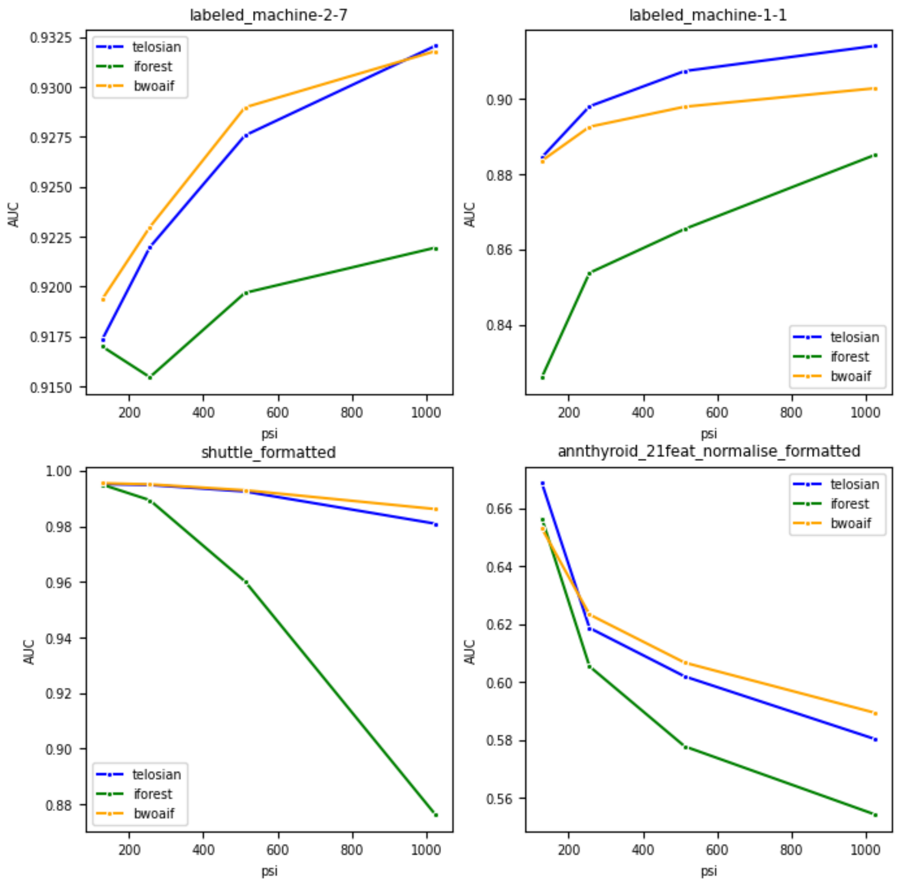

**Figure 5.3:** The AUC obtained when changing the subsampling size $\psi$.

sampling size when there are many attributes in the dataset could result in trees that are too shallow to explore multiple attributes. This could also be the reason we see an increase in performance when the subsampling size is large with datasets with many attributes (top row of Figure 5.3) and a decrease in performance when the number of attributes is lower (bottom row of Figure 5.3). Hence, we recommend using values of $\psi$ similar or greater to 1000 for both *Telosian* and *BWOAIF*. However, if the number of attributes is small, values of $\psi$ of 250 or lower should be used.

### 5.2.4 Effect of the batch size

The batch size determines the number of observations that must accumulate before the algorithm is updated. This parameter can be set by the user as we assume that the observations are continuous and arrive with high frequency. Hence, it is realistic for the user

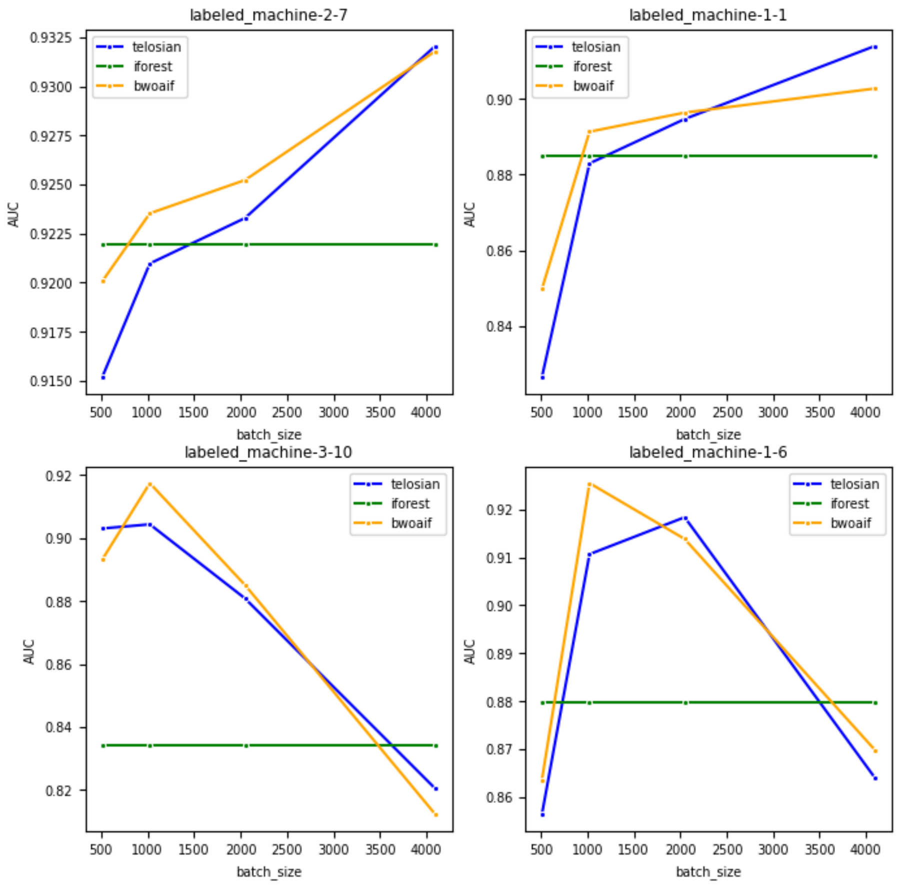

**Figure 5.4:** The AUC obtained with various batch sizes $B$.

to determine how many observations to include per batch. In the case of the *iForest* algorithm, this parameter does not have any effect because no trees are updated. For *Telosian* and *BWOAIF* on the other hand, we see that both algorithms have similar behaviour when changing the batch size. Although there does not seem to be an evident value for the parameter which maximizes the AUC, values between 600 and 2500 observations per batch result in higher AUC scores. This can be seen in Figure 5.4, where, although there is no evident common behaviour, we can see that both algorithms might be sensitive to large batches. It is important to note that the batch size not only determines the moment a new update will be triggered, but it also impacts the amount of data that will be kept in memory. Thus, setting this parameter to a small value will result in lower memory requirement.

## 5.3 *BWOAIF* vs. *Telosian*

In this section we will compare both the *BWOAIF* and *Telosian* algorithms to understand why they achieve similar results and also explain the differences. For this purpose we will focus on the *SMD* dataset, since the sub-datasets that compose it have the same features and are generated by similar systems. Nevertheless, they have different types of anomalies and behaviours which will allow us to test the algorithm under diverse conditions. After running *Telosian* and *BWOAIF* on the 28 sub-datasets, there were two main takeaways: first, the similarity in the AUCs obtained over the datasets, and second, their different performance with different degrees and types of concept drift. In the next subsections we will delve into these two topics.

### 5.3.1 Similarity in AUC

An evident result from the experiments is the similar performance of *Telosian* and *BWOAIF*. This can be seen in Table 5.1 with the aggregated AUC and in more detail in Figure 5.5 where it is evident that the algorithms have very similar performance for all datasets dispite the difference in number of trees needed. This is probably caused by the fact that both algorithms share two main elements: having *iForest* as core element and sharing the same weighting scheme.

Firstly, since *iForest* is the basis of both algorithms, they obtain similar results in the anomaly score computation stage, with the number of trees being the main difference between the algorithms. However, as seen in Section 5.2.1, *iForest* converges very quickly with a low number of trees so this is a reason we do not see a big difference in the AUCs, despite the number of trees.

Secondly, the weighting scheme borrowed from *BWOAIF* is designed to give a greater weight to trees similar to the most recently added trees. This means that in both algorithms, the most recent batch will heavily influence the weighted anomaly score. Since the new trees would be trained in similar data for both algorithms, we can expect similar anomaly scores despite the difference in their update schemes.

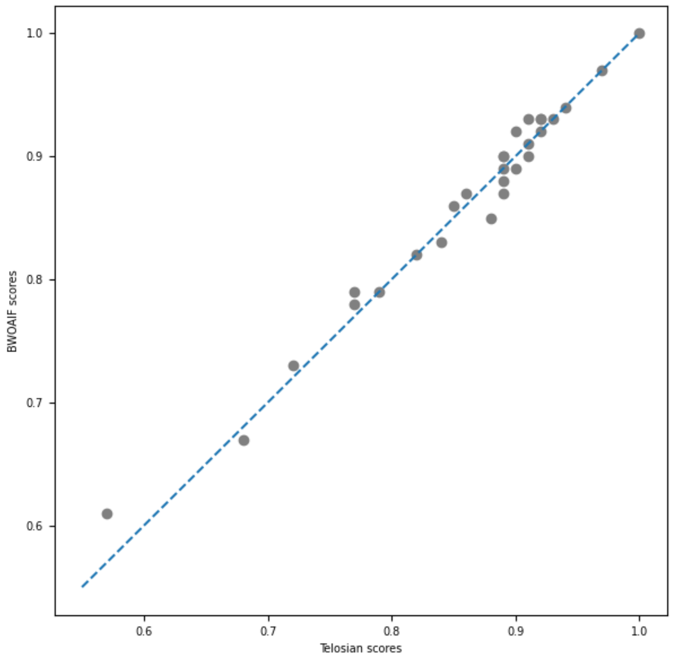

**Figure 5.5:** Similarity of the AUCs between *Telosian* and *BWOAIF* (dashed line corresponds to the $x = y$ line).

### 5.3.2 Difference in concept drifts

Despite the similarity in the AUC scores between both algorithms, the cause of the greatest difference between the algorithms was investigated. These differences, although small, can be attributed to an extent to the different types of drift present in the datasets.

To test if this was indeed the case the datasets were separated, creating a set where *Telosian* obtained higher AUC scores and another where *BWOAIF* obtained the higher AUCs. Then, an additional metric was computed to describe the datasets and understand the differences in performance. This metric called *mean difference* is the average of the differences between the drift of consecutive batches. Thus, the difference at time $t$ is defined as $d_t = |\nu_{t+1} - \nu_t|$ which will allow us to see, to an extent, how uniform or noisy the drift is for each dataset. Additionally, the *mean drift* was also included which is the average of the drift scores per dataset.

After computing this metric for all sub-datasets, the mean drift score and mean difference was taken. First, for the sub-datasets where *Telosian* obtained higher AUC scores than *BWOAIF*, and then, where the latter performed better than the former. The result is

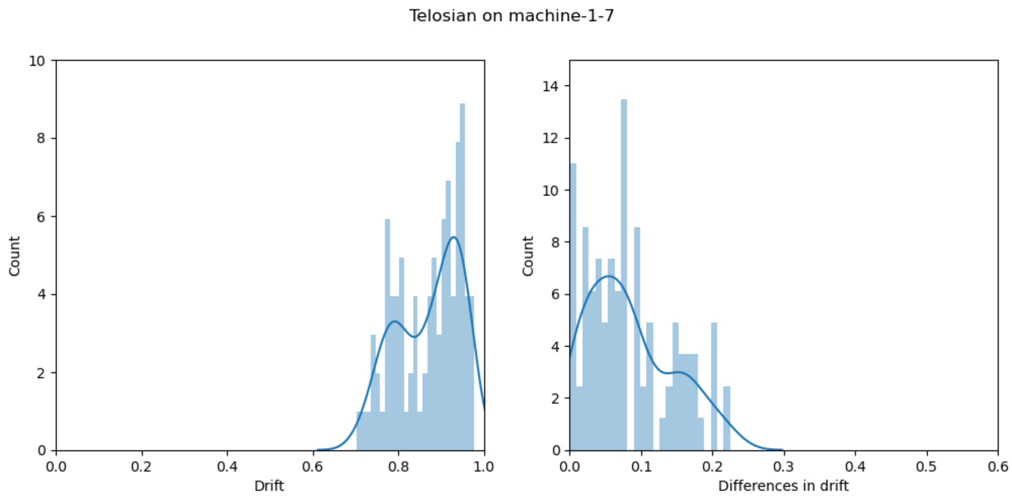

**Figure 5.6:** Drift an differences present in dataset `machine-1-7`.

shown in Table 5.2, along with the total mean difference and drift from all 28 sub-datasets.

| | **Mean difference** | **Mean drift score** |
|---|---|---|
| *Telosian* | 0.22 | 0.75 |
| *BWOAIF* | 0.19 | 0.71 |
| **All** | 0.20 | 0.73 |

**Table 5.2:** Mean difference and mean drift score when *Telosian* obtained higher results and when *BWOAIF* obtained higher results.

In the table we can see that when *Telosian* performed better, the mean drift was higher as well as the difference in drifts. This indicates that it did better when the level of drift was higher, but also when batches have a less uniform amount of drift between them. This is more clearly shown in Figure 5.6, where we see that most of the batches have a drift of over 0.8 and there is a spike of differences greater than 0, which shows that the drift levels changed throughout the batches.

In the case of *BWOAIF* we see in Table 5.2 that the mean drift and differences between sub-datasets are smaller. An example is shown in Figure 5.7, where the drift of each batch is more evenly distributed around 0.65 and the majority of the differences are closer to 0. This shows that the drift was not particularly high and that the batches showed similar amounts of drift.

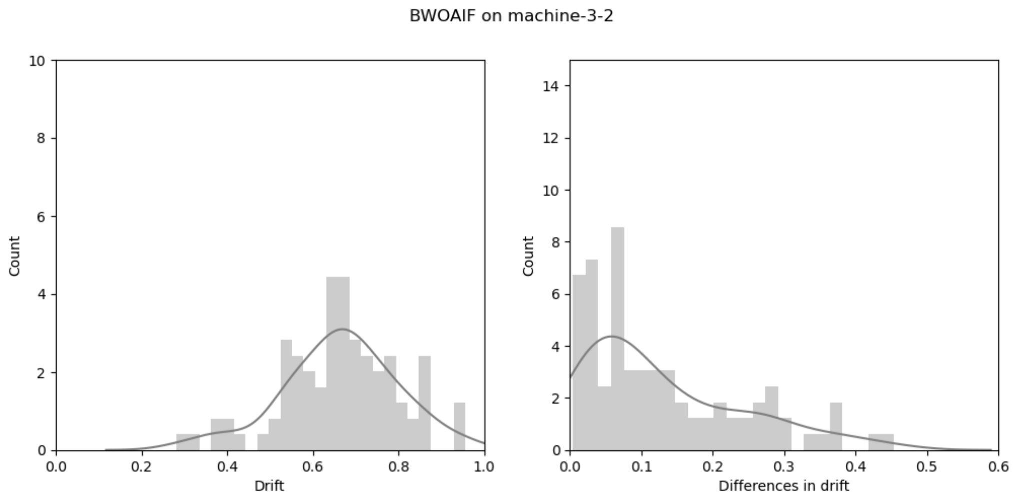

**Figure 5.7:** Drift an differences present in dataset `machine-3-2`.

In conclusion, we find that *Telosian* out performed *BWOAIF* when the drift was more noisy and higher, but the differences shown were still small, so both algorithms have similar performances. It would be interesting to investigate if this behaviour is maintained when testing on different datasets.

## 5.4 Real-time feasibility

In this section, we will review the execution times of the *Telosian* algorithm to assess its viability in a real-time scenario. It must be noted, however, that the implementation presented for this work is not time-performance oriented. Hence, the times presented here can be significantly improved. In [19], for example, the implementation of the *BWOAIF* algorithm was implemented in `Cython` and is considerably faster than our own `Python` implementation. A `Cython` implementation of the *Telosian* algorithm could be developed to further reduce the times presented here. Nevertheless, we consider that the execution times were satisfactory and show the viability of the algorithm in a real-time context. In the next subsections we will focus on the SMD dataset and review: first, the time it takes the algorithm to process each record, then, we will see the effect of the sub-sampling ($\psi$) and number of trees ($T$), parameters in the processing time, and finally, comment on the initial training time.

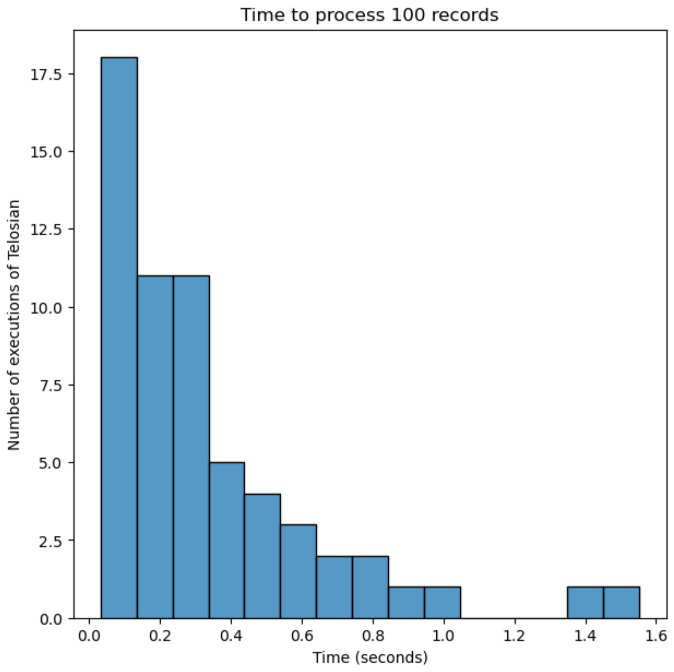

**Figure 5.8:** Time it takes to process 100 records with different executions of the *Telosian* algorithm in the SMD data, with different sub-sampling size, batch size and number of trees.

### 5.4.1 Processing time per second

In this part we will review the results of the time it takes the *Telosian* algorithm to process 100 records. This processing time includes the training of new trees, measuring of drift and the computation of the anomaly scores. Figure 5.8 shows the count of executions of the algorithm where 100 records were processed in the time range specified in the x-axis. Each of the executions were ran with different combinations of parameters, where the main variations were on the number of trees $T$, sub-sampling size $\psi$ and batch size. The figure shows that for most parameter combinations, the algorithm is able to process 100 records in less than 1 second. Furthermore, specific combinations of parameters take less than 0.2 seconds, which shows that changing the parameters of the algorithm could further reduce running times. In the next subsection we will discuss the effect of changing the sub-sampling size ($\psi$) and number of trees ($T$).

### 5.4.2 Effect of changing the parameters on the processing time

In this subsection we will review the effect of changing the number of trees ($T$) and sub-sampling size ($\psi$) which are the two parameters that have an impact on the execution time. The other parameters such as window size or weighting parameters have an effect on the AUC score, but do not change the number of operations performed, thus they are not included in this section. Although there are multiple possible combinations we chose some cases that are representative of the general behaviour of the algorithm. The results are summarized in Figure 5.9. Again, the processing time considers the time it took to train new trees, estimate the drift of the current batch, and compute the anomaly score for the data instances of the batch.

An initial observation we can make is that the batch size has a linear relationship with the processing time. This behaviour is the expected behaviour from the *iForest* algorithm and shows the scalability of *Telosian*. Additionally, the left plot shows that increasing the sub-sampling size $\psi$ does not have a significant effect of the slope of the duration but rather changes the intercept of the lines. This is mainly due to the fact that the sub-sampling size $\psi$ only has an effect on the training stage and not the computation of the anomaly score, which is more time consuming because it must be performed on all data instances rather than a subset of them.

On the other hand, the right plot shows that increasing the number of trees has a more evident effect on the slope of the execution time per batch. In this plot the sub-sampling size is fixed to $\psi = 256$. We can see that the greater the number of trees, the greater the processing time. Nevertheless, even with the largest number of trees ($T = 2048$), the number of records processed per second is approximately 300, so the real-time feasibility is maintained. Increasing the parallelism of the the computation of the anomaly score can further reduce the effect of increasing the number of trees.

### 5.4.3 Initial training of the algorithm

It is also important to consider how much data is needed to train the initial ensemble of the algorithm. This depends on the parameters chosen to run the algorithm, mainly $T$, the number of trees and $\psi$, the sub-sampling size. Since each tree is trained in a non-overlapping sample of the data, the total amount of points needed for the first ensemble must be equal or greater than $T\psi$. However, the data should be representative of the real-world application. For example, a very homogeneous data set (no anomalies) would not be useful to train the initial ensemble. Although *Telosian* is able to update itself, starting with a good ensemble will provide better performance in the earlier stages of the

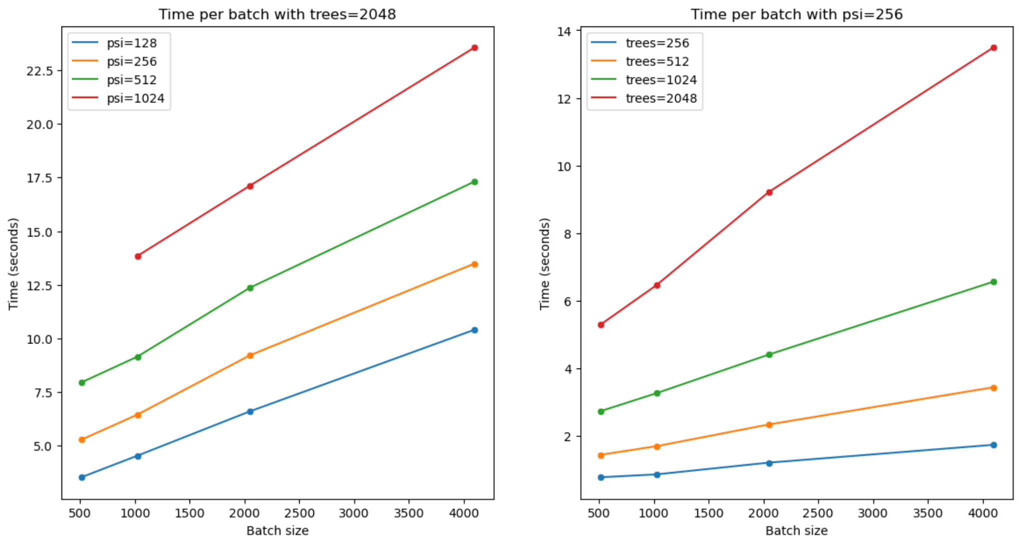

**Figure 5.9:** The effect of changing the number of trees ($T$) and sub-sampling size ($\psi$) in the processing time per batch.

algorithm.

All in all, the time analysis shows that the algorithm scales linearly with the number of records and also that even with a large amount of trees and sub-sampling size, *Telosian* is able to process a considerable amount of records in a short time, which makes it a feasible option for a real-time use case.

# 6

# Conclusion and future research

The goal of this research was to answer the research question: *How can anomaly detection algorithms adapt to concept drift in real-time unsupervised detection of unseen attacks in cyber security?* We answered this question by introducing *Telosian*, an unsupervised anomaly detection algorithm optimized for concept drift.

The components on which *Telosian* is built, which are inherited from the *iForest*, *BWOAIF* and *NNDVI* algorithms, as well as its new features such as the update scheme and the new tree estimation function, make it an algorithm that is able to scale, handle streaming data in real-time, leverage unlabeled data, account for concept drift, efficiently use resources and finally, perform a proportional update which were the main capabilities necessary to deal with cyber security data.

The algorithm was able to achieve comparable performance to the state of the art *BWOAIF* algorithm [19], and while there was no AUC improvement, it achieved similar results with fewer resources. The experiments showed that adding additional information about the concept drift within the data, could be leveraged to make custom updates to the model. The additional information was provided by the *NNDVI* algorithm [24], which provided a measure of the concept drift for each batch. Furthermore, the experiments also showed that the algorithm achieves good performance in various kinds of drifts and even in static data, having a slight advantage when concept drift was high or noisy. Additionally, the analysis of execution times confirmed that it is feasible to be used in a real-time context as it has low latency to process new data and is able to scale linearly with the amount of data.

We also identified some aspects of the algorithm which could be further investigated to improve the performance. For instance, new tree update functions could be tested, perhaps by using functions more sensitive to the drift or that take into account the drift of more batches instead of only the most recent one. Another idea would be to experiment with other algorithms to measure and perhaps try to identify the type of drift in order to perform a more tailored update. It would also be interesting to run a more extensive grid search to further understand the impact of the parameters. Additionally, running *Telosian* without the weighting scheme borrowed from *BWOAIF* could be a interesting way of comparing which method is more effective to adapt to concept drift. Furthermore, testing the algorithm on new cyber security data sets could prove of great value. Another interesting topic would be to find better ways to summarize the different types of drift within a data set rather than using only average drift. Finally, the trained trees could be leveraged to try to give explainability to the anomalies by using information gain or a similar metric to determine which features have a greater impact in a data instance being an anomaly.

To summarize, *Telosian* is an algorithm capable of achieving good detection performance on data sets from various sectors and with different types of drift, with a smaller resource requirement than comparable solutions. This makes it a versatile algorithm which can be applied to multiple types of applications and with a specific relevance to the cyber security sector.

# References

[1] Kdd cup 1999 data data set. https://archive.ics.uci.edu/ml/datasets/kdd+cup+1999+data. Accessed: 2023-03-10. 36, 37

[2] Menelaus. https://github.com/mitre/menelaus/tree/dev. Accessed: 2023-09-18. 41

[3] Olif - online iforest. https://ci.tno.nl/gitlab/iker.olarramaldonado-tno/olif. Accessed: 2023-03-10. 40, 42

[4] Statlog (shuttle) data set. https://archive.ics.uci.edu/ml/datasets/Statlog+(Shuttle). Accessed: 2023-03-10. 36

[5] Agrawal, S. and J. Agrawal (2015). Survey on anomaly detection using data mining techniques. *Procedia Computer Science 60*, 708–713. 6, 17

[6] Ahsan, M., K. E. Nygard, R. Gomes, M. M. Chowdhury, N. Rifat, and J. F. Connolly (2022). Cybersecurity threats and their mitigation approaches using machine learning—a review. *Journal of Cybersecurity and Privacy 2*(3), 527–555. 3, 5, 6, 7, 8, 9

[7] Alghushairy, O., R. Alsini, T. Soule, and X. Ma (2020). A review of local outlier factor algorithms for outlier detection in big data streams. *Big Data and Cognitive Computing 5*(1), 1. 7

[8] Andretta Jaskowiak, P., I. Gesteira Costa, and R. José Gabrielli Barreto Campello (2020). The area under the roc curve as a measure of clustering quality. *arXiv e-prints*, arXiv–2009. 43

[9] Bagui, S. S., D. Mink, S. C. Bagui, T. Ghosh, R. Plenkers, T. McElroy, S. Dulaney, and S. Shabanali (2023). Introducing uwf-zeekdata22: A comprehensive network traffic dataset based on the mitre att&ck framework. *Data 8*(1), 18. ii, 2, 38

[10] Benjelloun, F.-Z., A. A. Lahcen, and S. Belfkih (2019). Outlier detection techniques for big data streams: focus on cyber security. *International Journal of Internet Technology and Secured Transactions 9*(4), 446–474. 2, 3, 6, 7, 8, 9, 17

[11] Breunig, M. M., H.-P. Kriegel, R. T. Ng, and J. Sander (2000). Lof: identifying density-based local outliers. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pp. 93–104. 6

[12] Chitrakar, R. and H. Chuanhe (2012). Anomaly detection using support vector machine classification with k-medoids clustering. In *2012 Third Asian himalayas international conference on internet*, pp. 1–5. IEEE. 7

[13] Damasevicius, R., A. Venckauskas, S. Grigaliunas, J. Toldinas, N. Morkevicius, T. Aleliunas, and P. Smuikys (2020). Litnet-2020: An annotated real-world network flow dataset for network intrusion detection. *Electronics 9*(5), 800. 38

[14] Ding, Z. and M. Fei (2013). An anomaly detection approach based on isolation forest algorithm for streaming data using sliding window. *IFAC Proceedings Volumes 46*(20), 12–17. 2, 4, 8, 9, 10, 17, 19, 24, 35, 43

[15] Fawcett, T. (2006). An introduction to roc analysis. *Pattern Recognition Letters 27*(8), 861–874. ROC Analysis in Pattern Recognition. 43

[16] Fernando, T., H. Gammulle, S. Denman, S. Sridharan, and C. Fookes (2021). Deep learning for medical anomaly detection–a survey. *ACM Computing Surveys (CSUR) 54*(7), 1–37. 6

[17] Gemaque, R. N., A. F. J. Costa, R. Giusti, and E. M. Dos Santos (2020). An overview of unsupervised drift detection methods. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery 10*(6), e1381. 19

[18] Gomes, C., Z. Jin, and H. Yang (2021). Insurance fraud detection with unsupervised deep learning. *Journal of Risk and Insurance 88*(3), 591–624. 6

[19] Hannák, G., G. Horváth, A. Kádár, and M. D. Szalai. Bilateral-weighted online adaptive isolation forest for anomaly detection in streaming data. *Statistical Analysis and Data Mining: The ASA Data Science Journal*. ii, 2, 3, 4, 9, 10, 19, 20, 22, 23, 24, 31, 37, 41, 43, 56, 60

[20] Huasuya, T. (2019). Omnianomaly. `https://github.com/NetManAIOps/OmniAnomaly/tree/master`. 36

[21] Iwashita, A. S. and J. P. Papa (2018). An overview on concept drift learning. *IEEE access 7*, 1532–1547. 19

[22] Leenen, L. and T. Meyer (2021). Artificial intelligence and big data analytics in support of cyber defense. In *Research anthology on artificial intelligence applications in security*, pp. 1738–1753. IGI Global. 7

[23] Li, B., Y.-j. Wang, D.-s. Yang, Y.-m. Li, and X.-k. Ma (2019). Faad: an unsupervised fast and accurate anomaly detection method for a multi-dimensional sequence over data stream. *Frontiers of Information Technology & Electronic Engineering 20*(3), 388–404. 19

[24] Liu, A., J. Lu, F. Liu, and G. Zhang (2018). Accumulating regional density dissimilarity for concept drift detection in data streams. *Pattern Recognition 76*, 256–272. 2, 19, 25, 30, 31, 60

[25] Liu, F. T., K. M. Ting, and Z.-H. Zhou (2008). Isolation forest. In *2008 eighth ieee international conference on data mining*, pp. 413–422. IEEE. vi, 1, 3, 7, 9, 10, 13, 14, 15, 16, 37, 43, 47

[26] Lu, J., A. Liu, F. Dong, F. Gu, J. Gama, and G. Zhang (2018). Learning under concept drift: A review. *IEEE transactions on knowledge and data engineering 31*(12), 2346–2363. vi, 18, 19

[27] Lu, Y. and P. Xu (2018). Anomaly detection for skin disease images using variational autoencoder. *arXiv preprint arXiv:1807.01349*. 3, 18

[28] Maciá-Fernández, G., J. Camacho, R. Magán-Carrión, P. García-Teodoro, and R. Therón (2018). Ugr ‘16: A new dataset for the evaluation of cyclostationarity-based network idss. *Computers & Security 73*, 411–424. 38

[29] Moro, S., R. P. and P. Cortez (2012). Bank Marketing. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5K306. 36, 37

[30] Nazi, G. (2022). How the world became data-driven, and what’s next. 1

[31] Omar, M. and G. Sukthankar (2023). Text-defend: Detecting adversarial examples using local outlier factor. In *2023 IEEE 17th International Conference on Semantic Computing (ICSC)*, pp. 118–122. 3

[32] Parliment, E. (2022). Cybersecurity: why reducing the cost of cyberattacks matters. 2

[33] Perú, S. (2014). Club premier utiliza la analítica de sas para dar valor a su información y conocer mejor a sus clientes. 1

[34] Quinlan, R. (1987). Thyroid Disease. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5D010. 36

[35] Siddiqui, M. A., J. W. Stokes, C. Seifert, E. Argyle, R. McCann, J. Neil, and J. Carroll (2019). Detecting cyber attacks using anomaly detection with explanations and expert feedback. In *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 2872–2876. IEEE. 7, 8, 17

[36] Su, Y., Y. Zhao, C. Niu, R. Liu, W. Sun, and D. Pei (2019). Robust anomaly detection for multivariate time series through stochastic recurrent neural network. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pp. 2828–2837. 36

[37] Tan, S. C., K. M. Ting, and T. F. Liu (2011). Fast anomaly detection for streaming data. In *Twenty-second international joint conference on artificial intelligence*. Citeseer. 2, 19

[38] Togbe, M. U., M. Barry, A. Boly, Y. Chabchoub, R. Chiky, J. Montiel, and V.-T. Tran (2020). Anomaly detection for data streams based on isolation forest using scikit-multiflow. In *Computational Science and Its Applications–ICCSA 2020: 20th International Conference, Cagliari, Italy, July 1–4, 2020, Proceedings, Part IV 20*, pp. 15–30. Springer. 6, 7, 8

[39] Tufan, E., C. Tezcan, and C. Acartürk (2021). Anomaly-based intrusion detection by machine learning: A case study on probing attacks to an institutional network. *IEEE Access 9*, 50078–50092. 2, 3, 7

[40] Weisberg, H. I. and R. A. Derrig (1991). Fraud and automobile insurance: A report on bodily injury liability claims in massachusetts. *Journal of Insurance Regulation 9*(4). 8

[41] Yamanishi, K., J.-I. Takeuchi, G. Williams, and P. Milne (2000). On-line unsupervised outlier detection using finite mixtures with discounting learning algorithms. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 320–324. 37

[42] Zoppi, T., A. Ceccarelli, and A. Bondavalli (2021). Unsupervised algorithms to detect zero-day attacks: Strategy and application. *Ieee Access 9*, 90603–90615. 3, 6, 9, 17

# Appendix