Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Aleksander Styrmoe

Speeding Up k-Means-Clustering-Based Anomaly Detection Systems

Master's thesis in Communication Technology and Digital Security
Supervisor: Slobodan Petrović
June 2022

NTNU
Norwegian University of
Science and Technology

Aleksander Styrmoe

Speeding Up k-Means-Clustering-Based Anomaly Detection Systems

Master's thesis in Communication Technology and Digital Security
Supervisor: Slobodan Petrović
June 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Title:	Speeding Up k-Means-Clustering-Based Anomaly Detection Systems
Student:	Aleksander Styrmoe

Problem description:

The k-means algorithm is a popular clustering algorithm widely used in unsupervised machine learning. Anomaly-based intrusion detection systems (IDS) can use it to detect attacks on hosts and networks in situations where traditional signature-based IDS are not effective. However, the original k-means algorithm may be too slow for application in anomaly-based IDS due to high data rates in today's networks. Certain improvements to the original k-means algorithm have been proposed, yet few of these improvements have been applied in intrusion detection.

This thesis aims to build a more effective anomaly-based intrusion detection system by implementing some of the proposed improvements of the k-means algorithm on the distributed computing platform Apache Flink. These improvements exploit the triangle inequality property of the distance measure used in clustering. The master thesis contributes to increasing the efficiency of IDS based on the k-means algorithm, bearing at the same time in mind the computational overhead that implementing these improvements of the algorithm can introduce.

Date approved:	2022-01-26
Responsible professor:	Slobodan Petrović, NTNU IIK
Supervisor(s):	Slobodan Petrović, NTNU IIK

Abstract

The k-means algorithm is a popular clustering algorithm widely used in unsupervised machine learning. Anomaly-based Intrusion Detection Systems (IDS) can use it to detect attacks on hosts and networks in situations where traditional signature-based IDS are not effective. However, due to high data rates in today's networks, the original k-means algorithm may be too slow for IDS applications. In this thesis, we look at ways to speed up k-means-clustering-based IDS and build improved IDS on the distributed streaming platform Apache Flink.

Firstly, one can operate the k-means algorithm in different modes that vary in speed and accuracy. We have tested offline/batch k-means and some online versions. A mode of operation (MoO) that we call *transforming k-means* was the fastest mode and had a surprisingly high accuracy. Secondly, improvements to the original k-means algorithm that affect the speed have earlier been proposed, yet few of these improvements have been applied in intrusion detection. Three improvements based on the triangle inequality have been tested in this thesis. One improvement, Philips' Compare-Means algorithm, provides a slight speedup. In this thesis, we also use triangle inequality in online k-means. The tests showed that we could increase speed slightly by utilising Philips' idea of comparing centroid-to-centroid distances in *transforming k-means*.

This thesis also introduces domains as a concept to handle non-numeric data in k-means-based IDS. A separate Euclidean plane and a pair of centroids are assigned to data that share the same values on all non-numeric features. As a result, we have formalised the pre- and post-processing of data in k-means-based IDS, and discussed the consequences. One consequence is that some domains may be malicious by nature, in which case a signature-based IDS should assist our anomaly-based IDS.

We implemented the improvements and MoOs of the k-means algorithm on the distributed streaming platform Apache Flink. This thesis also introduces a new MoO which may be a good fit for anomaly-based IDS. It combines the speed and the surprisingly good accuracy of *transforming k-means*, and batch mode to update the model along the way. We propose anomaly-based IDS that apply the k-means speedups and different modes on Apache Flink. By doing so, the efficiency of anomaly-based IDS using k-means clustering are improved.

Sammendrag

K-means-algoritmen er en populær grupperingsalgoritme som er mye brukt i ikke-veiledet maskinlæring. Anomalibaserte inntrengningsdeteksjonssystemer (eng: Intrusion Detection Systems, IDS) kan bruke den til å oppdage angrep på verter og nettverk i situasjoner der tradisjonelle signaturbaserte IDS ikke er effektive. På grunn av høye datahastigheter i dagens nettverk kan den originale k-means-algoritmen være for treg til bruk i IDS. I denne oppgaven ser vi på ulike måter å øke hastigheten til k-means-grupperingsbasert IDS, samt bygger en forbedret IDS på Apache Flink, en distribuert streamingplattform.

For det første kan en benytte k-means-algoritmen i forskjellige moduser som varierer i hastighet og nøyaktighet. Vi har testet offline/batch k-means og noen online versjoner. En modus som vi kaller *transforming k-means*, var den raskeste modusen og hadde overraskende god nøyaktighet. For det andre finnes det tidligere foreslåtte forbedringer av k-means-algoritmen basert på trekantulikheten, som påvirker hastigheten, men få av disse forbedringene har blitt brukt i inntrengningsdeteksjon. Tre av disse forbedringene er testet i denne oppgaven. En av forbedringene, Compare-Means-algoritmen til Philips, gir en liten økning i hastighet. I denne oppgaven bruker vi også trekantulikheten i online k-means. Testene viste at vi kunne øke hastigheten litt ved å bruke idéen til Philips om å benytte sentroide-til-sentroide-avstander i *transforming k-means.*

Denne oppgaven introduserer også domener som et konsept for å håndtere ikke-numeriske data i k-means-basert IDS. Et eget euklidsk plan og et par sentroider er tilordnet data som deler de samme verdiene på alle ikke-numeriske attributter. På den måten har vi formalisert før- og etterbehandlingen av data i k-means-basert IDS, samt diskutert konsekvensene. En konsekvens er at noen domener kan være ondsinnet av natur, i så fall bør en signaturbasert IDS støtte opp om vår anomalibaserte IDS.

Vi implementerte forbedringene og modusene til k-means-algoritmen på den distribuerte streamingplattformen Apache Flink. Denne oppgaven introduserer også en ny modus som kan passe godt til anomalibasert IDS. Den kombinerer *transforming k-means* sin hastighet og overraskende gode nøyaktighet, og batch k-means for å oppdatere modellen underveis. Vi foreslår effektive anomalibaserte IDS, som bruker forbedringene av k-means og forskjellige moduser, på Apache Flink. Ved å gjøre dette, forbedres effektiviteten til anomalibaserte IDS som bruker k-means-gruppering.

Preface

This master thesis concludes my 5-year MSc in Communication Technology and Digital Security at the Faculty of Information Technology and Electrical Engineering at the Norwegian University of Science and Technology (NTNU). The thesis was carried out in 2022 during the spring semester. Although this thesis will provide background material, the readers are expected to have some technical knowledge. Professor Slobodan Petrović proposed the topic of research in May 2021.

Trondheim, June 2022

Aleksander Styrmoe

Acknowledgments

Firstly, I want to thank my supervisor professor, Slobodan Petrović, for guiding me and giving me the support I needed to seek solutions to roadblocks on my way. He also inspired me to pursue new and exciting knowledge. I would also thank Drake and Hamerly for their excellent work in one of my sources. They did a great job systemising the knowledge about improvements to the k-means algorithm.

I also want to thank the Department of Information Security and Communication Technology (IIK) and the great people there for our cooperation during my years in Trondheim. Thank you also to my classmates at my master's office and my girlfriend who has helped and supported me in my work with this thesis.

Finally, I want to thank my parents for supporting and encouraging me throughout my education. I would not have taken a Master of Technology without them.

– Aleksander

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

BR	Base Rate.
DAG	Directed Acyclic Graph.
FN	false negative.
FP	false positive.
FPR	False Positive Rate.
FTP	File Transfer Protocol.
HTTP	Hypertext Transfer Protocol.
IDS	Intrusion Detection System.
IPS	Intrusion Prevention System.
ML	machine mearning.
MoO	Mode of Operation.
PPV	Positive Predictive Value.
ROC	Receiver Operator Characteristic.
SMTP	Simple Mail Transfer Protocol.
TI	triangle inequality.
TPR	True Positive Rate.

Chapter 1

Introduction

1.1 Problem description

Efficient systems to mitigate attacks on hosts and networks are crucial in today's society. Intrusion Detection Systems (IDS) and Intrusion Prevention System (IPS) play a significant role in protecting personal, cooperative and national assets by detecting, identifying and, if possible, stopping attacks. To ensure IDS quality, efficient detection algorithms should be used since the attacks should be detected while they are in progress, i.e., the IDS must operate in real-time.

IDS can either be signature-based, also called misuse-based, or anomaly-based. The first type of IDS scans for signatures, or fingerprints, of known attacks when looking for intrusions, while the second looks for anything that is not close to normal traffic and behaviour. As a result, anomaly-based IDS are efficient against zero-day attacks, which are a significant threat in today's networks. In this thesis, we concentrate on anomaly detection systems.

To detect anomalies, clustering is often used as an unsupervised machine learning method. Clustering algorithms in IDS should be effective, fast and need enough capacity to handle large traffic loads. The k-means algorithm is a clustering algorithm that is often used as it is straightforward and linear in computation time. However, the original k-means algorithm is sometimes too slow for application in IDS.

Improvements to the k-means algorithm have been proposed as the original algorithm sometimes performs unnecessary computations. We can skip multiple calculations by utilising the triangle inequality. As a result, the algorithm's efficiency can then be increased. Even though the literature can tell us a lot about the proposed improvements, few of the improvements have been put into IDS. Therefore, in this thesis, we implement multiple improvements of the k-means algorithm put in IDS context and compare them with each other and to the original algorithm to speed up k-means-clustering-based IDS.

Moreover, there are many different ways of operating the k-means algorithm in IDS as there are many proposed modes of the k-means algorithm. As a result, we also investigate different modes and variations of the k-means algorithm that may suit IDS.

The overall goal of this thesis is to build a better k-means implementation for application in IDS. By also utilising the potential parallelisation can provide, we believe we can create an even better IDS. More specifically, we implemented the proposed improvements and modes of the k-means algorithm on Apache Flink. This relatively new distributed computing platform supports interesting features we use considering IDS. Furthermore, this thesis also presents a proposal for implementing improvements to the k-means algorithm for application in IDS in Apache Flink, as it has not been done before. Moreover, the thesis also presents how to implement k-means-based IDS in Apache Flink in general. By doing so, we explore different ways to speed up anomaly-based IDS as the clustering problem at hand is hard and speed is crucial.

1.2 Research questions

We have defined some research questions (RQ) that we answer in this thesis. These questions will help us build a better IDS based on the triangle inequality on Apache Flink.

RQ1 How can the different improvements of the k-means algorithm be adjusted to operate with Apache Flink?
In this RQ, we address the changes that need to be made to the already proposed triangle-inequality-based improvements of the k-means algorithm to implement them in Apache Flink.

RQ2 How can k-means on Apache Flink be used in intrusion detection applications?
The RQ is about how a k-means-clustering-based IDS can be implemented on Apache Flink, with or without triangle inequality. We also explore different ways to operate the k-means algorithm in IDS, i.e., different modes of k-means.

RQ3 What is the accuracy of an IDS using the different versions of the k-means algorithm?
By accuracy, we mean the IDS' ability to do its job. This RQ help us verify the different triangle inequality-based improvements of the k-means algorithm. The RQ also help us compare different ways of operating the k-means algorithm in IDS, as we expect the modes of k-means to vary in accuracy.

RQ4 What is the speed of an IDS applying the different versions of the k-means algorithm?
This RQ addresses the efficiency of the IDS. In addition, this RQ help us compare the different triangle inequality-based improvements and modes of operation of the k-means algorithm.

1.3 Objectives

Related to the research questions stated above, we define the following objectives of the thesis.

1. Investigate what needs to be done to build a k-means-based IDS.
2. Investigate what needs to be done to implement k-means on a parallel streaming environment.
3. Implement different modes of k-means, incl. the sequential, on Apache Flink.
4. Implement different improvements of k-means utilising the triangle inequality on Apache Flink.
5. Test and compare different implementations of k-means put in IDS.
6. Propose a better IDS build with k-means on Apache Flink.

1.4 Hypothesis

We hypothesise that the improvements of the k-means algorithm based on the triangle inequality can be implemented on Apache Flink and that they perform better in case of speed than the original implementation. We also believe Apache Flink is an excellent system for implementing an IDS. The accuracy of IDS using the proposed improvements should be the same. However, the different modes of operation should vary in accuracy, as briefly discussed in RQ3.

1.5 Main contribution

The main contributions of this thesis are the following.

1. Adapting improved k-means to Apache Flink.
2. Adapting modes of operation of k-means to Apache Flink.
3. Adapting k-means-based IDS to process data sets with non-numerical data.
4. Comparisons of different modes of operation and improvements of k-means.
5. Propose one or multiple better IDS on Apache Flink by suggesting methods to improve k-means based on triangle inequality and modes of operation of k-means.

1.6 Thesis structure

Chapter 2 This chapter gives relevant background on the theory and technologies used in the thesis. This includes introducing Apache Flink and essential concepts in machine learning.

Chapter 3 This chapter presents some former studies and papers contributing to implementing k-means and its improvements in parallel computing environments.

Chapter 4 The scientific methods used and a more detailed description of what we do in this thesis are provided in this chapter.

Chapter 5 Multiple theoretical contributions are made and are outlined in this chapter. In addition, RQ1 and RQ2 are answered in this chapter.

Chapter 6 Some preliminary results to the different improvements and modes of operation of the k-means algorithm, as read in the literature, are provided in this chapter.

Chapter 7 This thesis includes experimental work that is presented in this chapter. Both setup and results are presented. RQ3 and RQ4 are answered by the work outlined in this chapter.

Chapter 8 Discussion of the obtained results and other contributions by this thesis are provided in this chapter.

Chapter 9 Finally, this chapter gives a conclusion summarising our findings. Also, recommendations for future work are provided.

Chapter 2 Background

2.1 Intrusion detection systems

As briefly described in Chapter 1, Intrusion Detection Systems (IDS) are systems to detect intrusions and attacks on either networks or hosts. These systems detect attacks by either inspecting logs or looking at network traffic (packets on the network) and then trigger an alarm. If the system also prevents attacks, the system can be called an IPS, see for example [Pet20a].

IDS can be categorised in many ways. One way to distinguish between IDS is to look at the scope. It can either be looking at attacks on a network or attacks on hosts (servers and computers) on a network. The first variant, network-based IDS, look at network traffic. The second variant is called host-based IDS and is only installed on one computer to detect malicious activity on this computer.

Also, network-based IDS can be further divided into systems deployed inline (all network traffic goes through the system, like in a router) or passive (where the system receives a copy of the network traffic). Finally, host-based IDS can look at memory usage, system logs, OS API calls or inbound network traffic.[YT11]

To classify different types of traffic (normal and malicious), one can either define what normal traffic is (and report any deviations) or store abnormal traffic (signatures or fingerprints of known attacks). This distinction gives us two categories of IDS that can be divided: signature-based, also called misuse-based, and anomaly-based. The first category is the most commonly seen in today's networks. However, with the evolution of today's networks, the last type should also be used. Both categories will be described here, but this thesis will mainly focus on anomaly-based IDS. The results in this thesis apply to both network and host-based IDS.

2.1.1 Signature-based IDS

Signature-based IDS store signatures of known attacks. These signatures can be written as regular expressions or strings that the logs or network traffic are matched against. If a match is found, the system reports an intrusion. The signatures must be defined in advance, and all known and relevant attacks must be specified. If new attacks are known, one needs to define the signatures of the different attack variations. If a slight variation of an attack is not defined in the database, the attack is not be detected. Therefore, signature-based IDS are not effective against zero-day attacks as they are not known in advance.[YT11]

2.1.2 Anomaly-based IDS

Anomaly-based IDS stores indications of normal behaviour and report if any activity stands out. Everything that is not "close" to normal traffic is then classified as abnormal. Anomaly-based IDS can detect zero-day attacks. One challenge is that normal behaviour varies a lot, especially if a new service or point is introduced into the network or the host. At first, before this new normality is defined, the IDS may report false positives[YT11]. One way to store normal behaviour and find deviation is to use machine learning, which is explained in the next section.

2.1.3 Quantitative comparison of IDS

The effectiveness of an IDS can be defined as how good it is at doing its job, meaning it can look at how accurate and fast the IDS is. The metrics for accuracy can be based on rates of true and false positives and negatives, as described in Table 2.1. The Bayesian detection rate also called Positive Predictive Value (PPV), and the Receiver Operator Characteristic (ROC) curve are examples of such metrics. They use the True Positive Rate (TPR) and the False Positive Rate (FPR) as defined in Section 7.2.[Pet20b]

Table 2.1: Confusion matrix

	Real attacks / Anomalies	Normal traffic
Positive (alarm)	True Positive (TP)	False Positive(FP)
Negative (no alarm)	False negative (FN)	True negative (TN)

Then talking about the capacity and the speed of an IDS, we look at how much incoming traffic or traffic/event intensity the IDS can handle. For example, if an IDS can process a bounded source of N records by using x time, the capacity of the IDS can be written as N/x.

2.2 Machine learning

Machine learning systems are techniques and algorithms that allow computers to autonomously acquire and integrate knowledge and automatically learn programs using data to finish tasks. This knowledge and programs can be continuously improved as well. [YT11; Dom12]

Machine learning can be categorised into supervised and unsupervised learning algorithms. Depending on the problem, different algorithms suit the situation and form different branches of machine learning. One of the branches of unsupervised learning is clustering, described in Section 2.2.1. Machine learning can be supervised, meaning it learns with the help of humans. Often, the machine is given labelled data to train. On the other hand, unsupervised learning is when unlabelled data trains the machine learning *model* without humans correcting the model it creates. If we train an IDS based on a supervised learning method, we would need loads of labelled network traffic, which would be very expensive. Instead, unsupervised methods, like cluster analysis, suit our case in a better way.

In many machine learning systems, a model is trained from either labelled or unlabelled training data. This process is sometimes referred to as *fitting* the model. When this model is used to work on actual data, the phase is called *transforming*. If the machine learning system is tested, it is essential to test it with data that the system has not seen before.

2.2.1 Cluster analysis

Clustering is the branch of machine learning that aims to cluster data points that belong together and is a method of unsupervised machine learning. The data can be clustered in two or more clusters, i.e., assign data to different *classes*. The number of clusters depends on the use case and can be decided in advance or set during the algorithm's run-time, like in hierarchical clustering[HSD00].

Cluster analysis can be used in anomaly detection where the algorithm clusters data points in two clusters, one for normal data and one for non-normal data (also called anomalies). All data points that fall into the anomaly cluster are reported. This thesis focuses on clustering as an unsupervised machine learning method and further dive into the k-means algorithm.

Since clustering is unsupervised and unlabelled by nature, and since we want to trigger an alarm each time a data point is assigned to the abnormal cluster, another labelling method is needed to label the clusters afterwards. Labelling algorithms are not in the scope of this thesis, but the cluster with the largest cardinality often gets labelled as "normal". However, this method is shown not to be very effective [PAOC06].

Although this thesis focuses on two clusters, multiple clusters are possible. For instance, one can have clusters for normal traffic and different types of attacks. However, the labelling may be tricky. Supervised labelling is also one method that can be used.

Feature engineering is selecting the relevant *features* (attributes) to be used in the machine learning system. What are the relevant features? What features should not be included? These questions are taken care of by feature engineering, which is often done by experts that perform feature selection.

2.2.2 Challenges with machine learning and cluster analysis

The *No Free Lunch* theorem[WM97] states that no machine learning algorithm is superior in all use cases. In other words, without any assumptions, there is no better algorithm than another algorithm when solving any problem. Likewise, the *Ugly Duckling* theorem states that without any assumptions, there are no better *features* to be included over other features. [HSD00]

One major challenge is *overfitting*, referencing to when the machine learning system is so adjusted to the training data that it fails when put in real-life environments. Another challenge has been called the *curse of dimensionality.* This challenge relates to an observation that any point will tend to be equally distant to any other point when increasing the dimensions, i.e., the number of features. [Dom12]

In some clustering algorithms, the number of clusters needs to be chosen. The number of clusters to select is not always obvious and may be a cause for a less precise clustering algorithm. As mentioned above, we would like to have two clusters in anomaly-based IDS, as we have two classes; normal and abnormal traffic. However, with the incoming data, two clusters may not be suitable.

2.3 The k-means algorithm

The k-means algorithm, first described by Edward W. Forgy [For65] and Stuart Lloyd [Llo82] in 1965 and 1982, respectively, is a widely used unsupervised clustering algorithm[Kog07]. k stands for the number of clusters, and the algorithm handles k vectors representing the mean of the points in the different clusters. The k-means algorithm is commonly used due to its low complexity and linear computation time. The algorithm involves no training phase, as described in Section 2.2.

The algorithm tries to minimise the sum-squared error, SSE. SSE is defined in Equation (2.1), where C_j is the set of data points, or points, assigned to cluster j, c_j is the mean vector of cluster j and $d(x, y)$ is the Euclidean distance between the data points x and y. This problem is NP-hard[ADHP09], meaning there is no known optimal solution to this problem in linear time. The algorithm will only manage to find a local optimum[HW79].

The mean of the cluster is called the centroid. The k-means algorithm works as stated in Algorithm 2.1. Different distance measures can be used[Kog07]. In this thesis, our distance measure will be Euclidean.

$$SSE = \sum_{j=1}^{k} \sum_{x_i \in C_j} d(x_i, c_j)^2 \tag{2.1}$$

Algorithm 2.1 The original k-means algorithm

```
Initialise k centroids at random
While algorithm has not converged:
    Assign each vector to its currently closest centroid.
    Move each centroid to the mean of its currently-assigned vectors.
```

For step 1, different initialisation methods are used. In the Forgy's original version of the k-means algorithm, k vectors are chosen at random. However, other methods have been proved to be more effective in making the algorithm converge faster[CKV13], such as the k-means++. In this version, k vectors are chosen using a special distribution [Art+06].

If d is the number of dimensions in each vector, i is the number of iterations before convergence, and n is the number of points, the running time is given by $O(kndi)$. This version of the algorithm is called "crisp" or "hard" k-means since the vectors are assigned membership to one cluster. Fuzzy k-means are when the vectors have a membership to all clusters with a weight[HSD00]. We will work with "crisp" k-means in this thesis.

2.3.1 Modes: Offline and online versions of the k-means algorithm

The k-means algorithm can work in many different modes. The naïve algorithm originally works in offline or batch mode, meaning it runs when all data points are presented in advance. When put in IDS, the batch mode accumulates N vectors at a time when network traffic arrives, adds them to the pool of vectors, and then runs the algorithm on the whole collection of vectors. Old batches can either be included or not. By contrast, the online versions do not accumulate batches but process one vector at a time when it arrives and assign this vector a cluster without first recomputing centroids.

MacQueen proposed an online mode of k-means [Mac+67] that recalculates centroids (step 2 of Algorithm 2.1) each time a vector gets assigned a (new) cluster [Cel15, ch. 2.2.2]. On the other hand, the offline/batch mode will recalculate centroids only once per iteration.

Another online version is called sequential k-means[Dud97]. Sequential k-means will assign a new vector, x_i, to the closest centroid and then update the mean, c_{i-1}, using Equation (2.2). The number of vectors in the cluster after the ith vector has arrived is given as n_i. Recursion can show that Equation (2.2) gives the updated mean when a new object is introduced. This version of k-means will not need to store former vectors in each cluster, only c_i and n_i. However, the updated model does not get the benefit of letting vectors change clusters as running the whole k-means algorithm gives. As a result, the order of events will influence the result.

$$c_i = \frac{c_{i-1} * n_{i-1} + x_i}{n_i}, n_i = n_{i-1} + 1 \tag{2.2}$$

Sequential k-means can be implemented with *forgetfulness*. In normal k-means, all vectors in a cluster are weighted the same when calculating the mean, c_i. However, the algorithm can be programmed to "forget" old means. In Equation (2.2), c_{i-1} may be weighted with a factor of 0.5. As networks evolve, old traffic may be irrelevant for the IDS, meaning forgetfulness may be a good solution when using sequential k-means. Forgetfulness in batch mode is solved by weighting old batches when running the algorithm using weights on the old batches.

2.3.2 Improvements to the k-means algorithm

The original k-means algorithm is called the naïve k-means algorithm since it sometimes does multiple unnecessary computations. Improvements to the algorithm utilising the triangle inequality (sometimes referred to as TI) and other techniques have been proposed. The triangle inequality holds if the distance measure used in clustering is a metric and states that for any three points, a, b and c, where $d(x, y)$ is the Euclidean distance between the points x and y, this is always true: $d(a, b) \leq d(a, c) + d(b, c)$.

By using the triangle inequality on either the distance between centroids or bounds on distances between points and centroids, Philips[Phi02], Elkan[Elk03], Hamerly[Ham10] and Drake[DH12] have proposed some improvements that speed up the k-means algorithm and reduce the number of distance calculations. The work by

Drake and Hamerly in [Cel15, ch. 2] gives a good comparison and overview of the different improvements of the k-means algorithm.

This section lists some of the improvements this thesis uses that utilise the triangle inequality to skip some re-computations. Below, x is a vector, c is its currently assigned centroid, and c' is another candidate centroid. The following descriptions of improvements to the k-means algorithm are taken from the pre-project report prior to this thesis [Sty].

Compare-means (Philips [Phi02]) By the triangle inequality, we can skip the calculation of $d(x, c')$, if $2d(x, c) \leq d(c, c')$. This means x cannot be assigned to c'. The reason for this condition is the following. The triangle inequality states that $d(c, c') - d(x, c) \leq d(x, c')$. If $2d(x, c) \leq d(c, c')$ we can write $2d(x, c) - d(x, c) \leq d(x, c') \implies d(x, c) \leq d(x, c')$. Now we know that c' is far enough from c. In this improvement, distances between centroids need to be calculated each time centroids move (after each iteration).

Sort-means (Philips [Phi02]) This variant uses the same condition as compare-means, but is faster than compare-means since it searches for centers in a different order. This variant has a larger overhead by maintaining a k x k matrix storing centroid-to-centroid distances each time a new centroid gets calculated. By sorting each row of the matrix, we can see if any other centroid, c', is far enough from c, meaning $d(c, c') \geq 2d(x, c)$, by searching the row of c in increasing distance to c. If one centroid is found, the search can stop, x gets a new centroid and more importantly, we have skipped looking at some far-away unnecessary centroids. Although sort-means is faster, the variant has some extra overhead of sorting the centroid-to-centroid distance matrix [Cel15].

Upper and lower bounds (Elkan [Elk03]) Elkan's idea is to use upper and lower bounds instead of exact values to skip some computations. Let c be a centroid before one iteration of the k-means algorithm. Let $c*$ be the new position after recomputing the cluster mean after the iteration. Multiple conditions are checked to skip computations. As Philips, Elkan also uses the following fact: $\frac{1}{2}d(c, c') \geq d(x, c) \implies d(x, c') \geq d(x, c)$, and $d(x, c')$ is not needed to be computed. An upper bound, $u(x)$, for $d(x, c)$ is maintained. At the beginning, the upper bound starts as $u(x) = d(x, c)$. After each iteration, the upper bound is maintained by adding $d(c, c*)$ to $u(x)$. If $u(x) \leq \frac{1}{2}d(c, c')$, by checking the minimum $d(c, c')$ over all $c' \neq c$, we do not need to compute $d(x, c')$ and x does not change the cluster. A lower bound for $d(x, c)$, $l(x, c)$, is also used in this modification of the k-means algorithm. If c did not move too much during the iteration, $l(x, c) - d(c, c*)$ is a good approximation for $d(x, c*)$. If

$u(x) \leq l(x, c')$ or $u(x) \leq \frac{1}{2}d(c, c')$, the calculation of $d(x, c')$ is unnecessary and x does not change cluster to c'.

One lower bound (Hamerly [Ham10]) This is a modification of Elkan's algorithm and uses only one lower bound, $l(x)$, instead of $k * n$ lower bounds. $l(x)$ represents the minimum distance any centroid, except for the currently-assigned, can be at that point. It maintains the maximum distance any centroid has moved and uses this to update $l(x)$. If $l(x) < u(x)$ happens to be true, more computation is needed and x might switch cluster. This algorithm uses less memory than Elkan's algorithm, but calculates distances more often. This algorithm works better in low dimension [Cel15].

***b* lower bounds (Drake and Hamerly [DH12])** This variant is a trade-off to reduce the overhead in Hamerly's algorithm and the memory usage in Elkan's algorithm. It maintains $1 < b < k$ lower bounds for each vector. The first $b - 1$ lower bounds represent the distance to the $b - 1$ closest centroids, except for its currently-assigned. The last lower bound represents a lower bound for the $k - b$ furthest-away centroids and is updated by either subtracting the distance the furthest-away centroid has moved, or by subtracting the longest distance centroid has moved (sometimes beneficial) [Cel15]. The rest of the lower bounds are maintained, as in Elkan's algorithm, by subtracting the movement of each corresponding centroid. By sorting the lower bounds, we may skip unnecessary comparisons when investigating if $l(x, c) < u(x)$. The number b can be chosen by forehand or adjusted by the algorithm after each iteration.

2.3.3 Practical use of k-means

As k-means works in the Euclidean space, non-numeric, symbolic or nominal attributes in records cannot be processed in the k-means algorithm. Boolean attributes can be included since the distance between true and false will be 1 if we encode true as 1 and false as 0[YT11, p. 65]. The same value at boolean attributes will have a distance of 0. However, suppose the symbolic attribute can have three or more values. In that case, we cannot assign each value to a number to include the attribute in the k-means algorithm, as they not have a natural ordering [LO13, p. 85]. For example, if an attribute for a record (network package) is the name of the protocol, it will not make sense to compare the distance between Hypertext Transfer Protocol (HTTP) and File Transfer Protocol (FTP) and the distance between HTTP and Simple Mail Transfer Protocol (SMTP). HTTP is not closer to SMTP than to FTP.

We can consider some alternatives for dealing with these attributes. Firstly, we could ignore the nominal points and not include them in the vectors. However, this could be problematic since the anomaly traffic of one service may be closer to the normal traffic cluster of another service. Secondly, we can do as [YT11, p.

65] suggests. The authors suggest solving this issue by splitting the datasets into partitions of records with the same values on all symbolic (not binary) fields and running the k-means algorithm on each partition divided. This means the groups will not share centroids, and we can have one pair of centroids (attack traffic and normal traffic) for each partition.

2.4 Parallelisation and distributed computing

Distributed computing, or cluster computing (not to be confused with clustering as described in Section 2.2.1), is the technique of dividing the computation and operations on large datasets among multiple computer nodes that form a cluster in a parallel fashion[DG04; Les+20]. This means the dataset is split among all computers in a cluster, and the same operations are performed in each node.

The MapReduce paradigm was the first widely used framework implemented in Apache Hadoop. MapReduce is the process of *mapping* the data to different operators by key (while filtering or sorting the data) and then performing a *Reduce* function on all operators (like a summary function) before the result is emitted. Unfortunately, while Apache Hadoop is known for its fault tolerance and scalability, the framework does not support iterative algorithms [Al +17]. Apache Spark is a framework that adds more functionality to the MapReduce paradigm, enabling more algorithms to be implemented and are better suited for lesser fault-tolerant applications. In addition, the framework supports both batch processing, meaning the operators operate on batches of data, and streaming by dividing the stream of data into small micro-batches that execute operators[Mac21].

2.4.1 Apache Flink

Apache Flink is a relatively new distributed computing framework that supports stateful operations and different APIs to operate on the different layers to handle both bounded and unbounded data sources[1]. The Table API and the SQL API operate on the highest layer, while the DataStreamAPI operates on the lower layer enabling more custom data operations. A programmer can change between the different APIs freely. Flink supports both batch operations and streaming natively[Mac21].

In Flink documentation, [Docs] we can read that the Flink architecture consists of one centralised JobManager connected to a cluster of worker nodes called TaskManagers. The user connects to the JobManager either through the Flink Client or the command line interface (CLI) to execute Jobs that are executable Java or Scala programs. The JobManager creates a Directed Acyclic Graph (DAG) of operators,

[1]Bounded means the data has an end, while unbounded is a continuous stream of data.

sends it to the TaskManagers and distributes the workload to the TaskManagers, where the operators are.

The Apache Flink community is very active. Some years ago, the DataStream API was supposed to cover streaming, while the DataSet API would take care of batch applications. However, the DataSet API is an API the community wants to phase out, and it is not recommended to use. The community is working on extending the DataStream API to fully support batch and streaming interchangeably. As seen in the roadmap of Apache Flink[2], the DataSet API approaches the end of life and should not be used anymore.

The legacy DataSet API supports iterative algorithms. However, only one stream of data is allowed in and out of the iterations. This feature is also put into the DataStream API. Other Flink features include advanced windowing as well.

A *datastream* (stream of data) can be connected and distributed evenly among the operators that work in parallel (TaskManagers), or a datastream can also be broadcasted. This is useful if network traffic to analyse is distributed among multiple operators, and the centroids in the k-means algorithm are broadcasted to each operator, ready to assign points to clusters.

Apache FlinkML

In January 2022, the community launched FlinkML 2.0. In a blog post, [LG22], they stated that they want to build a machine learning framework inside Apache Flink that also comes with some out-of-the-box machine learning algorithms. The machine learning library is marked as MVP on Apache Flink's roadmap, meaning it is still in early development.

In the FlinkML documentation, [MLDocs], we can read that the FlinkML library operates at different *Stages*. The first *Stage* is an *Estimator* responsible for training the machine learning model and implementing a *fit* method. This method takes in training data and outputs a *Model*. Next, the *Model* is another estimator that, among other things, implements a method called *transform*, which takes in data and uses the model data to produce a result or prediction. This corresponds to the training phase as described in Section 2.2.

The library also introduces a framework for iterative algorithms that supports multiple streams to be fed back into the iteration, emitted and put into the iteration like in Figure 2.1.

[2]https://flink.apache.org/roadmap.html

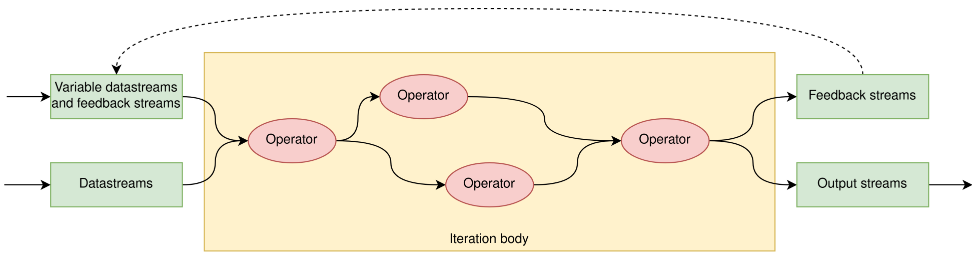

Figure 2.1: The iteration paradigm in Apache FlinkML. The figure is based on Apache FlinkML documentation[MLDocs].

KMeans in FlinkML

Offline and online implementations of the k-means algorithm have been added to the FlinkML library[3]. The online implementation was added at the end of March 2022. Note that the online implementation is not included in the current stable documentation of FlinkML[4] (May 2022).

Since the machine learning paradigm in Flink operates with a *fit* operation to create a model, the *OfflineKMeans* implementation in the machine learning library of Apache Flink is also split into two operations that derivate from the original k-means algorithm. The two operations are *fit* and *transform*. The *fit* operation creates a "KMeansModel" that essentially are centroids. In this operation, the offline k-means algorithm is performed. The second operation is the *transform* operation that takes input vectors and assigns them to their closest centroids from the "KMeansModel" in one iteration without adjusting the centroids. This paradigm works a lot like other machine learning systems, as described in Section 2.2. The *OfflineKMeans* implementation in FlinkML (the *fit* operation) terminates after a fixed number of iterations and will not look at when the algorithm has converged.

OnlineKMeans in FlinkML is a forgetful sequential k-means implementation as described in Section 2.3.1.

[3]https://github.com/apache/flink-ml/tree/master/flink-ml-lib/
[4]https://nightlies.apache.org/flink/flink-ml-docs-stable/

Chapter 3 Related work

3.1 Naïve k-means and Elkan's improvement on MapReduce

Despite Apache Hadoop not supporting iterative algorithms, AlGhamdi et al.[Mac+67] implemented k-means on Apache Hadoop by introducing a driver that took control over the iterations and fed the MapReduce framework for each iteration.

Along with implementing the naïve algorithm on MapReduce, KMMR-N, the authors also implemented Elkan's improvement of the k-means algorithm by using bounds to utilise the triangle inequality. The bounds were stored in two different methods, and both were tested. The first method, KMMR-EV, extended the point into an extended vector (EV), also containing the bounds. The second method, KMMR-BF, included storing this information in a separate file.

The results showed that KMMR-EV decreased the running time compared to KMMR-N with speedups of 4.5x. KMMR-BF was even faster, with speedups of 6.8x compared to KMMR-N. However, this varied with the number of dimensions, records and clusters. The overhead of KMMR-EV could outweigh the gained time from skipping distance calculations in high dimensions, high k and large number of records so much that it becomes slower than KMMR-N.

In future work, AlGhamdi et al. suggest testing other improvements of the k-means algorithm. For instance, Philips' Compare-Means or Hamerly's version using only one lower bound. The authors also suggest testing both methods to store the overheads.

3.2 Naïve k-means and Elkan's improvement on legacy Apache Flink

Ringdalen used version 1.9 of Flink when he, in his master thesis from 2020, implemented the naïve and Elkan's versions of the k-means algorithm on the Apache

Flink framework [Rin20]. The 1.9 version of Flink was quite different from today's stable version. Firstly, no machine learning library was available to support *Stages* and iterations with multiple datastreams. Besides, no k-means implementation was available out-of-the-box. As described in section 2.4.1, the earlier versions had other APIs, like the DataSet API. This API supported an iterative paradigm that only could handle one datastream in and out of the iterations. To mitigate this, Ringdalen introduced a tagged tuple to feed points, centroids and other information[1] to the next iteration. Apache Flink provides programmers with some examples, including an example showing the k-means algorithm implemented using the DataSet API. However, this example only supported two dimensions, so Ringdalen extended the example to support multiple dimensions and further extended that implementation to apply Elkan's improvements.

Ringdalen compared the naïve and Elkan's versions of the k-means algorithms on Flink and compared them to a naïve implementation using classical computing. Results showed that Elkan's version performed worse than the naïve version when increasing the number of clusters. Only increasing the parallelism and having a small k made Elkan's version perform better than the naïve version. However, the overhead of introducing tagged tuples to store bounds did outtake the gained time by skipping distance calculations. With parallelism of 4, Elkan's version had an increase in performance of just 1.8% with $k = 2$.

One could wonder if this result is because the datastream consisted of many tagged tuples that were not actual points and that these tagged tuples were quite large.

As further work, Ringdalen suggests looking into online versions of the k-means algorithm since input in IDS is unbounded by nature. Also, future work should work on only transferring information that will be altered during iterations and not the entire tagged tuple.

[1]Ringdalen called this information Carry Over Information (COI) and was sent as tuples containing information about bounds and more.

Chapter 4 Methodology

In this chapter, we look at how we solve the problem that *naïve k-means sometimes is too slow for application in IDS*, (see Section 1.1). We can formulate the goal of this thesis as *building a better k-means-clustering-based anomaly detection system based on already proposed improvements of the k-means algorithm utilising the triangle inequality on Apache Flink.* As the goal is to improve an artefact, this can be called a design process, as described in [Wie14]. The design process is described in Section 4.1. To help drive the thesis forward, as already described in Section 1.2, we have defined some research questions. The research questions help us compare the proposed improvements of the k-means algorithm to find the best solution and verify that we have achieved the goal and built a better IDS. They will also help us make actual artefacts. The experimental work that we performed in the thesis is described in Section 4.2.

Later in this thesis, we use the word *domain* as a concept to solve what was introduced in Section 2.3.3. (See Definition 4.1).

Definition 4.1. Domain Group of features with the same value on all non-numeric attributes (i.e., those attributes that cannot be included in the k-means algorithm). The k-means algorithm is performed on each domain separately. In other words, a domain is a separate Euclidean plane with a separate set of centroids where all points have the same value on all nominal attributes.

Mode of Operation

We will also use the term Mode of Operation (MoO). When putting k-means in an IDS context, as we have seen in Section 2.3.1, the algorithm can operate in different modes and be used in different ways. Three modes will be described here:

Batch When offline is put in IDS context, we operate k-means in batch mode as described in Section 2.3.1. We can vary the size of the batches ($batchSize$). However, in our experimental work we operated with $batchSize = 1000$.

Transform This mode, that we also call transforming k-means, is proposed by us and corresponds to the transform method in the KMeans implementation of FlinkML as described in Section 2.4.1. This mode means having a trained model, i.e., centroids and assigning vectors to the closest centroids without updating the centroids.

Sequential This mode is described in Section 2.3.1 as sequential k-means and allows the centroids to update. This mode can also be used on a trained model or by choosing the first points as initial centroids. In our experimental work we used a trained model as initial centroids.

4.1 Building a better IDS

As Wieringa presented in [Wie14], understanding the problem is central to the design process. By doing so, we can design different artefacts that could treat the problem. This phase includes software development in our case, where we make an IDS implementing different versions of k-means. We then validate the designed treatments (or artefacts) to see if they fulfil the requirements and solve our problem. This is done as part of the experimental work where we test each artefact.

4.1.1 Functional requirements

To design different artefacts that may solve the problem, we need to define the requirements for our IDS that determine the scope of the thesis. These requirements are specified in Table 4.1.

We set $k = 2$ in this thesis as we operate with two labels and clusters; one for normal traffic and one for anomalies (referencing FR1).

4.1.2 Designing artefacts

We introduce concepts, use literature, and adjust pseudo-code to design artefacts that may treat the problem and fulfil the requirements. This work is given in the theoretical contribution in Chapter 5. During this work, we answer RQ1 and RQ2.

Table 4.1: Functional requirements for our IDS.

ID	Functional requirement
FR1	The IDS should be able to cluster network traffic into two classes: One for normal traffic and one for abnormal traffic.
FR2	The IDS should be at least as accurate as naïve IDS meaning it should have the same T/FPR as naïve or better.
FR3	The IDS should be faster than a naïve version.
FR4	The IDS should implement an improvement of k-means based on triangle inequality.
FR5	The IDS should be implemented on Apache Flink.
FR6	The IDS should run k-means on all domains integrated, not in separate operators. Nor should it skip or drop the nominal fields.
FR7	The IDS should stop running the algorithm when the algorithm has converged and not when a fixed number of iterations has been performed.
FR8	The output of the IDS should be the points along with IDs of their assigned cluster so the clusters later can be labelled.

4.1.3 Validation and evaluation

To validate fulfilled requirements and compare the different artefacts, we ran statistical difference-making experiments to see the difference between our treatments on a test data set. The experimental work in Section 7 shows our results during this validation and comparisons. Our two RQs that were answered using experiments were RQ3 and RQ4.

The discussion in Section 8 evaluates the artefacts and provides solutions, based on the validation and theoretical contribution, where we describe a better IDS.

Software development

As part of our experimental setup, we perform software development. The software development was done in an agile fashion, where some of the requirements were taken care of for each iteration[Wie14]. We started with the KMeans implementation in FlinkML and changed that implementation in multiple iterations before we fulfilled all the requirements.

One challenge we faced is the researcher's knowledge about the platform Apache Flink, which may introduce unwanted interference in the results. One of the reasons we chose to start with the KMeans implementation that comes with FlinkML as a starting point was to minimise this effect. We focused on modifying the fit operation since this is the operation that is more like the original k-means algorithm. Then, we also needed to change the returned model to receive information about what vectors were assigned to which centroid.

4.2 Testing artefacts in experiments

We performed multiple experiments in this thesis. We used Wieringa's framework for the Empirical Cycle to get an understanding of our design research. As stated by Wieringa, design research is when we want to answer research questions about implementations or treatments to a problem.

Wieringa talks about different forms of treatments and variables when doing design research. Experimental treatments, by some known as independent variables, are the different ways we can treat the problem. It is the controllable input of the experiments we will perform. The output or results are studied when changing the treatments. Measured variables, by some known as dependent variables, are the measured result after performed experiments. Extraneous variables and confounding variables are variables that may influence the measured variables out of our control.

This section presents the different experimental treatments, i.e., experiments we performed. In addition, measurements regarding accuracy and speed are also outlined. The way the experiments were performed is outlined as well. All this leads to some requirements for our experimental work at the end of this section.

4.2.1 Experimental treatments

There are multiple variables we may change to find a better IDS. We concentrate on two of these that make up our two types of experimental treatments. We may vary the use of improvements of k-means based on triangle inequality (TI), from now called treatment/variable A, or we could change the mode of operation, from now called treatment/variable B.

Treatment A: Applying triangle inequality

The first treatment we apply is triangle inequality, i.e., testing different improvements of the k-means algorithm based on the triangle inequality. We use the improvements already introduced in Section 2.3.2. Table 4.2 shows the versions of treatment A that will be tested.

The naïve version is our reference treatment, i.e., the use of no triangle inequality. Since we only deal with $k = 2$, Sort-Means is not tested in these experiments. "Philips' algorithm" will then only refer to Compare-Means. Drake is not used since it requires $k > 2$. When applicable, we test treatment A on different MoO during the experiments, starting with offline.

Table 4.2: Treatments relevant to variable A to be tested.

TreatmentID	**Name**
A.1	Naïve k-means
A.2	Philips' Compare-Means
A.3	Elkan's algorithm
A.4	Hamerly's algorithm

In this treatment, we are interested in the difference in execution time and how the distance calculation reduction can increase the algorithm's speed. Therefore, when performing experiments on one specific MoO using different artefacts where treatment A is changed, it is important that we use the same data set, the order of events is the same and that the same initial centroids are chosen. The accuracy is only used for validation of A.1-A.4 as they all should perform the same when it comes to accuracy.

Treatment B: Using different modes of operation

Treatment B is about varying the different MoO that may be useful in IDS. Each mode that is tested and its parameters are described earlier in this chapter. The transform and sequential modes use trained models. To compare these MoO, both will use the same trained model, made from the outputting centroids from running k-means offline on a separate set of training data. Table 4.3 summarises the different MoO that we tested.

In the batch and sequential modes, the order of incoming events influences the result. Therefore, five different seeds are used to randomise the input order. All artefacts where treatment B vary are tested five times, one for each order of input, even the transform MoO. A mean from the five different orders of events is provided.

Table 4.3: Treatments relevant to variable B to be tested.

TreatmentID	**Name of MoO**	**Order of input...**	**Parameter**	**Trained**
B.1	Batch	...affect the result	*batchSize*	No
B.2	Transform			Yes
B.3	Sequential	...affect the result		Yes

Forgetful sequential k-means is not tested since that would be beyond the scope of this thesis. However, forgetful sequential k-means would be interesting in IDS as the internet is evolving. Moreover, test data sorted in order of event time should be used to test a forgetful algorithm. And we do not require this from our test data.

In this treatment, we are first and foremost interested in the accuracy of different MoO. How fast these different modes are also investigated. However, these modes differ widely in how they operate so there is significant differences. All artefacts are tested using the same dataset. No triangle inequality is applied when testing treatment B.

4.2.2 Measurements

This section outlines the different measurements we performed during the experiments.

Speed

To measure the calculation time for the different implementations, we looked at the execution time of the Flink Job by inspecting the logs afterwards. There, we could get a Job's *NetRunTime* in milliseconds.

The Flink Job, as already mentioned, includes pre-processing data and writing results to files. However, these operations were identical for all implementations we tested, meaning the results here are comparable.

Our calculations of execution time assume Flink's built-in timer is good. Regardless, there may be numerous extraneous variables that influence execution times. For example, other processes on the computer and memory state affect the results. Therefore, we also accumulated the number of distance calculations done. Flink comes with build-in accumulates we utilised for this purpose, whose output is given in the logs after running a Flink job.

Accuracy

We use PPV to compare different variants of IDS regarding accuracy. As a result, TPR/FPR were calculated. Each experiment's output was written to a file with the assigned cluster-ID and the correct label.

As part of the dataset we use, the correct class is included, i.e., the data is labelled. Then prepossessing the data, this label is included in the data points sent through the algorithm but is not used. Ideally, we would remove these labels when sending the data as point vectors into the algorithm and, in the end, join the resulted data points and their assigned cluster-id with the original dataset using the Base64 encoding of the point vector as the joined attribute, giving us the labels back. However, such joining may not be possible as multiple data points in the dataset may share the same value on the vectors.

We need to label the clusters after performing our experiments to calculate accuracy. As already mentioned, cardinality is often used. However, depending on the dataset used, this may not at all be the case. Therefore we assume a perfect labelling algorithm that is always correct and as good as supervised labelling. The cluster with the fewest normal data points is labelled positive. If a domain has the same number of normal data points in both clusters, the cluster with the highest cardinality is chosen as the "normal" cluster.

We accumulate the output data to measure TPR and FPR. This is done by accumulating the number of point vectors assigned to each cluster and the number of vectors labelled as normal in each cluster.

There are some important considerations when measuring accuracy when introducing domains. Firstly, we need to remember when calculating TPR/FPR that we cannot accumulate points in clusters 1 and 2 in different domains since the labelling probably would not be the same for cluster 1 in one domain and another. Therefore, the distinction between domains should remain until finished labelling each cluster in each domain. Secondly, some domains may have no data points with either normal or abnormal labels. This may be because the dataset has too few data points in each domain. Therefore, we measure accuracy without those domains, i.e., where the probability of an attack, Base Rate (BR), is 1 or 0.

4.2.3 Execution of experiments

Our experiments may have multiple extraneous variables that interfere with the measured variables. Wieringa states that randomisation and numerous attempts cancel out the effect of extraneous variables. The speed measurements are believed to be affected by extraneous variables the most. As a result, these experiments require us to perform multiple attempts and calculate a mean. The runtime we use is the mean of five executions.

We also need some test data for our experiments. As already stated, the test data should be the same for all artefacts where the treatment vary i.e., when comparing treatments A or B. The test data should be labelled so we can calculate the accuracy. We need two test sets for B.2 and B.3 that simulate network data — one for training and one for the testing.

4.2.4 Requirements for the experiments

The above considerations give us the requirements shown in Table 4.4. These requirements are included in the agile development process.

Table 4.4: Requirements for the experiments.

ID	**Requirement**
RE1	We need to accumulate the number of distance calculations.
RE2	We need to include the correct label in the outputted data.
RE3	We need a framework for testing different improvements without unnecessary differences between the artefacts interfering with the results.
RE4	We need an offline k-means implementation that outputs the centroids so we can use the centroids in B.2 and B.3 as initial centroids.

Chapter 5 Theoretical contribution

To fulfil some of the functional requirements and answer RQ1 and RQ2, multiple theoretical contributions have been made and that are outlined in this chapter. Where applicable, the theoretical contribution is presented for different MoO we use in the experimental work (referencing beginning of Chapter 4).

5.1 Introducing domains in k-means-based IDS

We need to solve the problem described in Section 2.3.3 regarding nominal features in k-means before implementing a k-means clustering-based IDS. The solution is to take what [YT11] said and introduce the concept of domains. Domains are already defined in Definition 4.1. The domain can be identified as a string being the conjunction of all non-numeric fields. The k-means algorithm is performed on each domain separately. Figure 5.1 shows what an IDS utilising the solution presented in [YT11] would look like.

By introducing the concept of domains, we have not altered the k-means algorithm but rather formalized the pre-processing and post-processing when applying k-means in IDS. Nevertheless, there are some essential considerations that we need to address.

In some domains, a clustering algorithm makes sense. For example, if one domain consists of HTTP traffic and another consists of SMTP traffic, both domains probably have normal traffic and anomalies. However, some domains may contain one class by nature. Some combination of flags and services may always be anomalies as they will never be normal behaviour. However, if a combination makes no sense now, it may be the root of a future zero-day attack. This depends on the dataset and the different features selected. Not taking care of this issue would lead to multiple false positive (FP) or false negative (FN) that should not happen in those domains. K-means does not solve this challenge. In such cases, an anomaly detection system should be assisted by signature-based detection by looking at those specific domains.

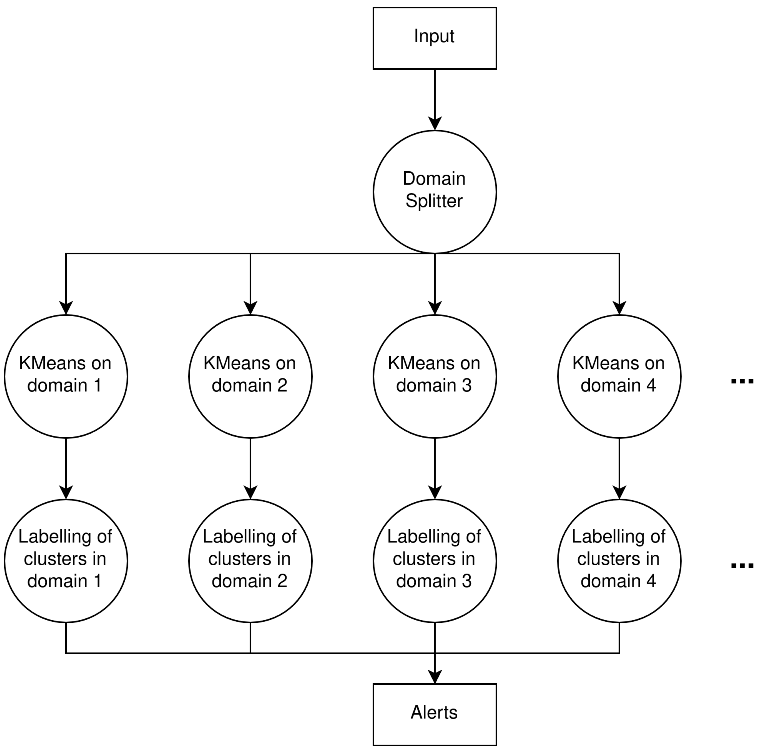
Input
Domain Splitter
KMeans on domain 1
KMeans on domain 2
KMeans on domain 3
KMeans on domain 4
...
Labelling of clusters in domain 1
Labelling of clusters in domain 2
Labelling of clusters in domain 3
Labelling of clusters in domain 4
...
Alerts

Figure 5.1: An anomaly-based IDS implementing the solution to nominal features as suggested by [YT11].

We suggest not automatically approving domains that seem only-normal domains as they may introduce zero-day attacks in the future.

If the number of vectors in a data set is less than k, there are not enough vectors to choose initial centroids, so k-means cannot be started, or we may need to wait for more data. In other words, k-means need to be dropped if there are too few vectors in offline mode. On the other hand, the algorithm needs to buffer the data point in online mode. This issue with too few vectors is likely when introducing domains in the IDS, as vectors in one domain may arrive less frequently.

5.1.1 Adjusting KMeans in streaming, like in FlinkML, to support domains

We run one instance of k-means while having domains as an integrated part in this thesis. Pseudo-code and description for this k-means algorithm to run on distributed streaming platforms (i.e., Flink) follows. Future work may adjust k-means to support domains on classical computing.

In the *KMeans* implementation that follows FlinkML, one datastream is used for only one element; an array holding the centroids. We have expanded the *KMeans* implementation to support multiple arrays of centroids, one for each domain. These elements are stored in *MapStates* at operators, and when a point comes, the correct array of centroids is retrieved. All centroid and point objects also have a domain attached as a field. Figure 5.2 shows an example of the centroid datastream after initial centroids have been chosen.

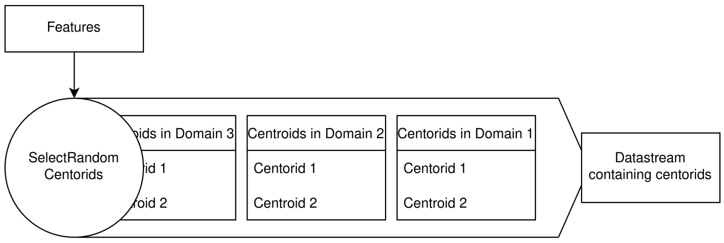

Figure 5.2: Diagram showing the centroid datastream after random centroids have been selected.

This adjustment has been made to the *Offline* and *Online fit* implementations in FlinkML and the *transform* method.

5.2 New framework for k-means on Apache Flink

This new framework is used in our experiments to make each artefact as similar as possible, and for adjusting k-means in FlinkML to support multiple improvements.

To make each implementation as similar as possible, as one of the requirements for the experiments states (Table 4.4), we placed each improvement's unique logic in objects. All points and centroid objects have information relevant to the improvement stored in them and have updating methods that update internal fields like bounds and other fields based on all centroids. In the framework, we combine *IterationBody* from the FlinkML library, different datastreams and some operators that call these unique methods in the objects.

The *Point* object has fields for the Euclidean vector, the ID of the assigned cluster (initial value: -1) and a field for indicating the domain. The *Point* object also has an *update* method that takes in the centroids and updates the assigned cluster-ID field and eventually other improvement-specific fields. The *Centroid* objects have fields for the mean vector of the cluster, the cluster-ID and the domain identification. The *Centroid* objects have two methods. The first is called *move* and moves the centroid to a new mean vector. The second is called *update* and takes in the other centroids and updates eventual improvement-specific fields.

The framework can be adjusted to support multiple modes of operation, and all adjustments to the modes of operations are given in this section. In the framework, we have three central operators that are used. These are *PointUpdater*, *CentroidUpdater* and *CentroidInitialiser*. Pseudo-codes for those operators are given in Algorithms 5.1, 5.2 and 5.3, respectively. Explanation of the different operators follows.

The **PointUpdater** takes broadcasted centroids and calls *update* on the points using the stored centroids in the correct domain.

Algorithm 5.1 Pseudo-code for PointUpdater

```
operator PointUpdater:
    processElement(point):
        centroids = getState()
        # If centroids not present, store point in buffer,
        # and process point when centroids arrive.
        point.update(centroids)
        emit point
    processBroadcastElement(centroids):
        storeInState(centroids)
```

The **CentroidUpdater** takes in centroid objects and vectors that are used to move the centroids. Firstly, each centroid is called with *move* using the corresponding updating value as an argument. After this, each centroid is called on with *update* using all centroids as arguments. This is done when both the values and centroids for the domain are present; otherwise, we buffer the objects. Please note that one of the streams needs to either be broadcasted or the stream has to be keyed for domain-pair to operate on each other.

The **CentroidInitialiser** operator does the same as the *UpdateCentroids* operator but it does not call *move* on the centroids. The operator has just one input stream.

5.2.1 Offline/batch MoO adjusted for our framework

Figure 5.3 shows the information flow in this implementation. Points and centroids that enter the iteration are used in the *PointUpdater*. The finished points and centroids are sent out of the iteration and are emitted as output (Section 5.3 goes into detail about the blue operators in the figure). The updated points split into two streams, one for the next iteration and one that goes through the *NCVOperator*. The *NCVOperator* stand for *NewCentroidValuesOperator*, It produces new values

Algorithm 5.2 Pseudo-code for CentroidUpdater

```
operator CentroidUpdater:
    processElement(value):
        centroids = getState()
        # If centroids not present, store value in buffer,
        # and process value when centroids arrive.
        for centroid in centroids:
            centroid.move(value)
        for centroid in centroids:
            centroid.update(centroids)
        emit centroids
    processBroadcastElement(centroids):
        storeInState(centorids)
```

Algorithm 5.3 Pseudo-code for CentroidInitialiser

```
operator CentroidInitialiser:
    processElement(centroids):
        for centroid in centroids:
            centroid.update(centroids)
        emit centroids
```

for the centroids that are input into the *CentroidUpdater* along with the centroids themself. The updated centroids are sent to the next iteration.

5.2.2 Transform MoO adjusted for our framework

Unlike in the offline version, an *IterationBody* is not used in this mode of operation. After the centroids have been initialised, the points are updated using the centroids. The finished points are emitted directly. The data flow can be seen in Figure 5.4.

5.2.3 Sequential MoO adjusted for our framework

In this mode of operation, a new field is included in the centroid objects. The field is called *weight* and essentially symbolises the cardinality of the cluster. The move method works a bit differently from the other centroids in this MoO. The input is not a new value of the cluster's mean but a new point that is included in the cluster. The new mean is calculated using the stored weight and the Equation (2.2).

Figure 5.5 shows that the centroids are initialised before entering the iteration. The *IterationBody* works much like the offline version but does not calculate new centroid values in a *NCVOperator*. It just emits updated centroids based on updated

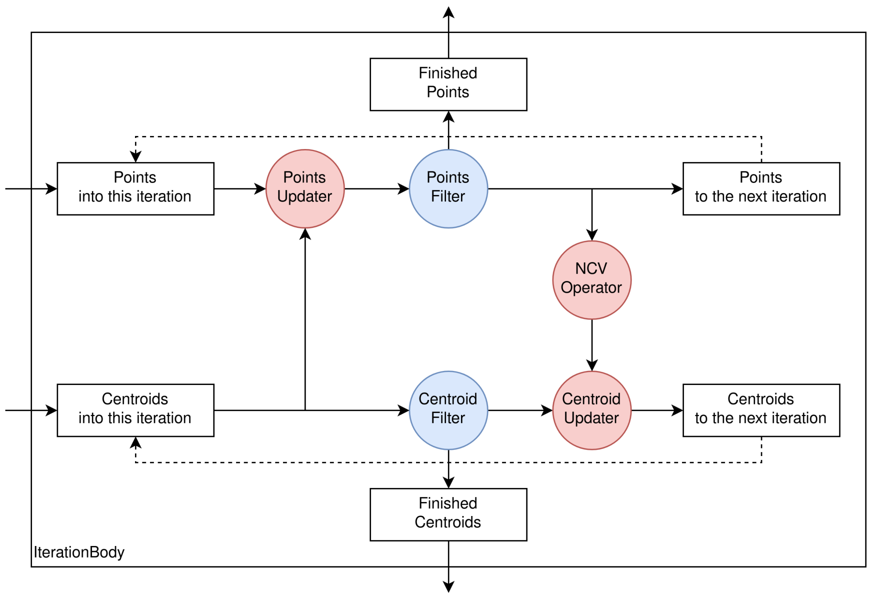

Figure 5.3: The data flow of offline k-means implemented in our framework.

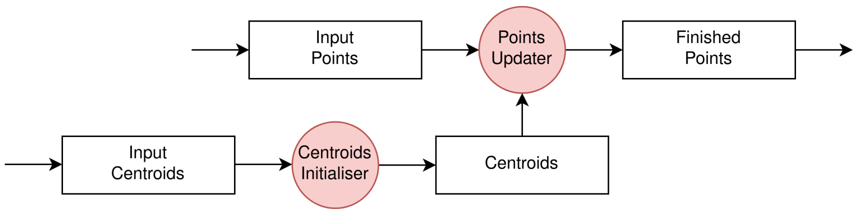

Figure 5.4: The data flow of transforming k-means implemented in our framework.

points directly. Also, no points are iterated multiple times, just emitted directly after being updated and assigned a cluster.

The initial centroids can be the first points arriving or two points randomly picked from the first 1000 points. Then, the initial weight will be 1. However, the initial centroid can also be based on centroids after training. In that case, the initial weight needs to be the cardinality of the clusters the corresponding centroids represent.

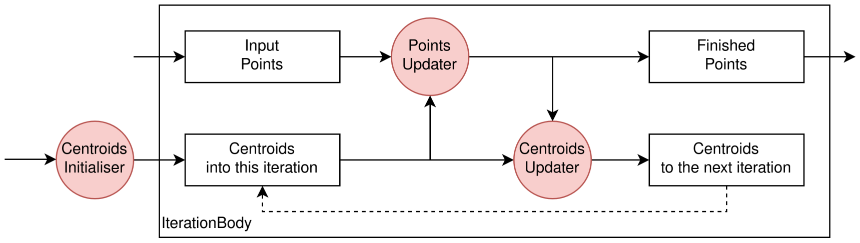

Figure 5.5: The data flow of sequential k-means implemented in our framework.

5.3 Adjusting offline k-means in FlinkML to stop when the algorithm has converged

The original offline k-means implementation in FlinkML has a fixed number of iterations that the user sets when initialising the KMeans object. The correct implementation should involve looking at when the algorithm converges, i.e., when points have stopped changing clusters and all centroids' movements are zero.

We have changed the data flow in the *KMeans* program in FlinkML to support convergence by introducing a filter that emits centroids and points when the algorithm has converged. The data flow is given in Figure 5.3.

All centroids in offline implementation have a field for the movement that is used to determine when the algorithm has converged. This field is initialised with the maximum value. This field is updated in the *move* method, where the distances between the new and old values are used.

After the centroids have been updated in the operator called *CentoridUpdater*, the centroids may have a movement of 0. If so, the centroids are fed into a new iteration so the *PointUpdater* can mark the points as finished. After this, the points with the finished flag and the centroids with $movement = 0$ are filtered out of the stream.

Note that domains can converge at different times. For example, if all centroids in a domain have stopped moving, the domain can be marked as finished, and the output of that particular domain can be emitted. It will save time ending calculations on domains that are finished.

The concept of convergence does not apply to online algorithms, so the *move* method in the *Centroid* objects does not update a *movement* field.

5.4 Adjusting improvements to fit into Apache Flink

By using the framework presented in Section 5.2, we have reduced the improvement-dependent side of the implementation to just *Centroid* and *Point* objects. In this section, we show the adjustments we make to the already-proposed improvements of the k-mean algorithm, so they fit Apache Flink and are adapted for our new framework for offline k-means. For each improvement, pseudo-code for the *update* functions of *Centroids* and *Points* are presented, as well as data structures inside these objects. A naïve version is also be given for completeness, adjusted for Flink and our framework.

The *Centroid* and *Point* objects in these adjustments extend those presented in Section 5.2, i.e., they have the same fields.

5.4.1 Naïve k-means adjusted for our framework

This implementation is relatively straightforward. The *update* method in *Point* and *Centroid* objects is given in Algorithms 5.4 and 5.5. As we can see, the *update* methods in *Centroid* objects are empty as there are no unique logic or fields in points or centroids that need to be updated. The *update* method in *Point* objects will just find the closest centroid and update the *assigned* field.

Algorithm 5.4 Naïve k-means: PointUpdater

```
update(centroids):
    minDistance = MAX
    assigned = -1
    for centroid in centroids:
        distance = distance(centroid)
        if (distance < minDistance):
            minDistance = distance
            assigned = centroid.ID
```

Algorithm 5.5 Naïve k-means: CentoridUpdater

```
update(centroids):
    empty
```

5.4.2 Philips' algorithm adjusted for our framework

Philips presented two versions of his improvement that reduce distance calculations [Phi02]. The first version, Compare-Means, calculates the inter-means distances for each pair of centroids (numbered $0 < i, j < k$) in a matrix, $D[i][j]$, and uses these when looping through the centroids when checking for closest centroid to a point.

The second version, Sort-Means, calculates one more matrix, $M[i][z]$, where $M[i]$ is an array representing centroids in increasing order of their distance to the ith centroid. For each point (whose assigned centroid has id *assigned*), the algorithm checks the centroids in the order given by $M[assigned]$ and skips to the next point if the distance between the currently-assigned centroid and the candidate centroid is larger than twice the distance to its currently-assigned centroid.

Since we adjust Philips' algorithm for application in streaming, we do not have a globally defined $D[i][j]$ but rather one array per centroid object holding distances to other centroids. Sort-Means is a good improvement over compare-means when $k > 2$. However, in our case, $k = 2$. Therefore, we do not present a Sort-Means version for Flink, but the adjustment presented here can be easily expanded to support Sort-Means.

Our algorithms for Philips' algorithm on Flink and the framework presented in Section 5.2 are presented in Algorithms 5.6 and 5.7. *Centroid* objects also has a field called *distanceArray* that holds distances to other centroids.

Algorithm 5.6 Philips' improvement: PointUpdater

```
update(centroids):
    minDistance = MAX
    assigned = -1
    for centroid in centroids:
        if (not first iteration):
            if (2*distance(centroids[assigned]) <=
            centroids[assigned].distanceArray[centroid]):
                continue
        distance = distance(centroid)
        if (distance < minDistance):
            minDistance = distance
            assigned = centroid.ID
```

Algorithm 5.7 Philips' improvement: CentroidUpdater

```
update(centroids):
    update distanceArray
```

5.4.3 Elkan's algorithm adjusted for our framework

The *Point* objects in Elkan's version have a field for *upperBound* and an array holding *lowerBounds* to other centroids. The *Centroid* objects have an array containing distances to different centroids called *distanceArray*. The *Centroid* objects also hold half the distance to the closest other centroid in a field called *halfDistToClosestCentroid*.

Pseudo-codes for *update* methods in the *Point* and *Centroid* objects are given in Algorithms 5.8 and 5.9. Central in work at making these pseudo-codes are the work done by [Al +17]. [Cel15] and [Elk03] are used as sources to the original improvements suggested by Elkan.

5.4.4 Hamerly's algorithm adjusted for our framework

Each *Point* object has fields called *lowerBound* and *upperBound*. The *Centroid* objects hold half the distance to the closest other centroid in a field called *halfDistTo-ClosestCentroid*. Pseudo-codes for *update* methods in the *Point* and *Centroid* objects are given in Algorithms 5.10 and 5.11. Both [Cel15] and [Ham10] were used as a source for the original improvement by Hamerly.

5.5 Triangle inequality in online k-means algorithms

Triangle inequality can also be used in the online mode of operations, like Transform and Sequential. However, not all proposed ideas to improve the k-means algorithm are applicable. For instance, extra information that involves points and are updated for each iteration, are not a good fit for online modes. Examples are Elkan's and Hamerly's algorithms, that have bounds between points and centroids. However, using the triangle inequality on distances between centroids will work and can be used in Transform and Sequential modes. This is the idea we utilise and stems from Philips' Compare-Means algorithm. The pseudo-codes for *update* methods in *Point* and *Centroid* objects is the same as in algorithms 5.6 and 5.7.

Algorithm 5.8 Elkan's improvement: PointUpdater

```
updateFirstIteration(centroids):
    lowerBounds = array with k values set to 0
    minDistance = MAX
    skipStatus[] = array with k boolean values set to false
    for (0 <= j < k):
        if skipStatus[j]: continue
        distance = distance(centroids[j])
        lowerBounds[j] = distance
        if (distance <= minDistance) {
            minDistance = distance
            upperBound = minDistance
            assigned = j
            for (j < z < k):
                if (centroids[j].distanceArray[z] != null):
                    distToZCentroid = centroids[j].distanceArray[z]
                else:
                    distToZCentroid = centroids[j].distance(centroids[z])
                    centroids[j].distanceArray[z] = distToZCentroid
                    centroids[z].distanceArray[j] = distToZCentroid
                if (distToZCentroid >= 2*distance):
                    skipStatus[z] = true
update(centroids):
    # Update bounds
    for (0 <= j < k):
        lowerBound[j] = max(lowerBound[j]-centroids[j].movement, 0)
    upperBound += centroid[assigned].movement
    updateUpperBound = true
    # Find closest centroid
    if (upperBound <= centroids[assigned].halfDistToClosestCentroid):
        return
    d1, d2 = 0
    for (0 <= j < k):
        if (j != assigned && upperBound > lowerBounds[j] &&
        upperBound>0.5*centroids[assigned].distanceArray[centroids[j].ID])):
            if (updateUpperBound):
                d1 = distance(centroids[assigned])
                upperBound = d1
                lowerBounds[assigned] = d1
                updateUpperBound = false
            else:
                d1 = upperBound
            if (d1 > lowerBounds[j] ||
            d1>0.5*centroids[assigned].distanceArray[centroids[j].ID]):
                d2 = distance(centorids[j])
                lowerBounds[j] = d2
                if (d2 < d1):
                    assigned = j
                    upperBound = false
```

Algorithm 5.9 Elkan's improvement: CentoridUpdater

```
update(centroids):
    minDist = MAX
    for centroid in centroids:
        if centorid.movement == 0 and this.movement == 0: continue
        if centroid.ID <= ID: continue
        dist = distance(centroid)
        distanceArray[centroid.ID] = dist
        centroid.distanceArray[ID] = dist
        if (dist < minDist): minDist = dist
    halfDistToClosestCentroid = minDist / 2
```

Algorithm 5.10 Hamerly's improvement: PointUpdater

```
update(centroids):
    # Update bounds
    upperBound = upperBound + centroids[assigned].movement
    biggestMovement = 0
    for centroid in centroids:
        if (centroid.movement > biggestMovement):
            biggestMovement = centroid.movement
    lowerBound = lowerBound - biggestMovement
    # Find closest centroid
    z = max(lowerBound, centroids[assigned].halfDistToSecClosest)
    if (upperBound <= z): return
        upperBound = distance(centroids[assigned])
    if (upperBound <= z): return
    bestDist, secBestDist = MAX
    bestC, secBestC = 0
    for (0 <= i < k):
        candidateDist = distance(centroids[i])
        if (candidateDist < bestDist):
            secBestDist = bestDist
            secBestC = bestC
            bestDist = candidateDist
            bestC = i
            continue
        if (candidateDist < secBestDist):
            secBestDist = candidateDist
            secBestC = i
    if (bestC != assigned):
        assigned = bestC;
        upperBound = distance(centroids[assigned])
    lowerBound = distance(centroids[secBestC])
```

Algorithm 5.11 Hamerly's improvement: CentoridUpdater

```
update(centroids):
    update halfDistToSecClosest
```

Chapter 6 Preliminary results

6.1 Treatment A

Literature can tell us about the differences between the improvements of k-means regarding speed. In [Cel15, ch. 2.7] we can see that Hamerly's algorithm is much faster than Elkan's algorithm in low and medium-dimension data sets. In data sets where distance calculations are costly, i.e., high dimensions ($d = 32$), Elkan's algorithm outperforms the other improvements. Philips' algorithm does not perform very well compared to naïve k-means when the dimensions are high in reducing distance calculations.

In Section 3, we looked at the work done in [Rin20]. There, Elkan's improvement would not be faster than naïve k-means unless the parallelism were four or higher. We will also probably achieve the same result since we in Section 5.2 have introduced significant overhead to implement the IDS in Flink.

For reference, the different improvements of the k-means algorithm utilising the triangle inequality are expected to be equally accurate and produce the same result as the naïve version (as already stated in Section 1.4).

6.2 Treatment B

As B.1 k-means runs the algorithm using multiple iterations and allows points to change clusters, this mode of operation is believed to be the most accurate. The larger the *batchSize*, the more accurate the algorithm should be, but the slower it should get.

Since the model does not change using B.2, this mode will probably perform the poorest. However, since the test data is not over a larger time frame where the centroids should have shifted, the resulted accuracy should be acceptable when using our test data. Sequential k-means are expected to perform better than the transform MoO since we update the model.

B.2 and B3 will have a number of distance calculations equal to the number of points times k and should be speedy. However, B.2 should be a bit faster since new centroids do not need to be calculated (even if the calculation is not very complex).

Chapter 7
Experimental work

In this section, the experimental work is presented. This chapter is split into the part where the results when applying treatment A are presented and the part where the results when applying treatment B are given. First, we look at the experimental setup that is common for both treatments and how the results are presented and calculated.

7.1 Experimental setup

This section describes the data set we used in our experiments for both treatments. Also, we present the experiment environment. The implementations of MoO for our experiments are also included.

7.1.1 Test data

The NSL KDD dataset includes 41 features and is a subset of the KDD CUP'99 dataset[DG17; UNB]. The original KDD CUP'99 data set had some problems, and NSL KDD (as proposed in [TBLG09]) is a subset trying to solve some of these problems related to KDD CUP'99. Although the authors in [TBLG09] still admit the NSL KDD data set "may not be a perfect representative of existing real networks, because of the lack of public data sets for network-based IDSs", they still believe it can be applied as an "effective benchmark data set to help researchers compare different intrusion detection methods".

Most of the features in NSL KDD are numbers that can be treated fine with Euclidean distance measure in the k-means algorithm. Four of the points are boolean according to the included ARFF file. We include these points in measuring distance by assigning *true* to 1 and *false* to 0. Two more Boolean points, *root_shell* and *su_attempted* exist according to [UCI99]. However, we only confirmed *root_shell* to be Boolean as *su_attempted* had rows containing the value "2". This feature is included as a numeric value.

The three remaining points, *protocol_type*, *service* and *flag*, are nominal. This challenge is already outlined in Section 2.3.3. The solution called domain is described in Section 5.1 and will be realized.

NSL KDD is labelled, meaning there is one more column in the dataset. This attribute is included in the *Point* objects to do calculations later but is not included in the k-means algorithm. Ideally, this would not be included in the *Point* objects, and for finding correct labels afterwards, unique IDs would be calculated from domain and vector. Still, a unique ID on domains and vectors did not exist.

One feature is connected to the "difficulty level" of the data point (in [UNB] called *successfulPrediction*) and was not included in our experiments.

The NSL KDD data set consists of two sets, one for training machine mearning (ML) algorithms and one for testing ML algorithms. For testing treatment A, we used the training data set for the offline mode and test data for the online modes. For treatment B, we used the test data set. The train data set is used if the mode of operation requires a trained model. Then, the initial centroids are the resulting centroids after running offline k-means on the training data.

Table 7.1 shows an analysis of the two data sets in NSL KDD when taking domains into account. Section 5.1 states that domains with too few points ($< k$) will not be considered for clustering. This means only 258 domains are processed in the training data set. 174 of the 258 domains processed in the train data set are also found in the test set, meaning only 174 pairs of centroids will be the initial centroids in B.2 and B.3. Domains with no initial centroids will not be included in the k-means algorithm in B.2 and B.3, or in the accuracy calculations afterwards. As mentioned in Section 4.2.2, domains with no points in one of the classes would not be included in the accuracy calculation.

7.1.2 Implementation

The implementation is performed using Flink 1.15.0 and FlinkML 2.0.0. The DataStreamAPI is used in the programs. The framework introduced in Section 5.2 is used to implement the different MoO and the improvements of k-means based on triangle inequality are implemented using pseudo-code from Section 5.4. In this section, we dive into some specific comments regarding the implementation of the artifacts to run them in experiments and to validate/evaluate them, as well as making the environment to run experiments in. Source code used in the experimental work can be found at GitHub[1].

[1] https://github.com/astyrmoe/MSc-thesis-experiments

Table 7.1: Analysis of NSL KDD data sets.

	Train set	**Test set**
Number of points	125973	22544
Number of domains	336	189
Number of domains with just one point	−78	−30
Remaining domains	= 258	= 159
Remaining points	125895	22514
Domains with no points in a class, of the remaining ones	−204	−130
Domains included in accuracy calculation	= 54	= 29
Points included in accuracy calculation	88461	17348
Nomalies included in accuracy calculation	66590	9666
Anomalies included in accuracy calculation	21871	7682

Two objects, *Point* and *Centroid*, are used in our k-means implementations, both inheriting from a *Vector* object. Both objects have a method called *distance* that is used to calculate the Euclidean distance to another *Vector*. This method includes a counter so we can get the number of distance calculations done later.

In some MoO, centroids and points have some additional fields. For example, in sequential MoO, the *Centroid* objects also have a *weight*. When implementing triangle inequality utilised in the k-means algorithm, we make objects inheriting from *Point* and *Centroid* objects to make sure we change them as little as possible, only adding the improvements. This means the Flink Job still makes the same method calls for each version, but the implementation of *move* and *update* methods inside the objects are overwritten. Sections 5.4 and 5.5 describe the logic inside the objects.

The data gets loaded into Flink from a CSV file. Flink comes with a datatype for double vectors called *DenseVector*, which we use to represent vectors in the Euclidean space. The domain is also stored as a string. The string (unique identifiers for domains) was made by conjugating all non-numeric fields in the dataset. The label is also stored in these objects for later pre-processing as is is not possible to derive a primary key from the domain and vector as multiple points shared the same values on these fields.

The Flink Job retrieves the test data, as described above, and runs the mode of operation. After the Flink Job has received the resulting points (and centroids) with its assigned clusters another FlinkJob calculates measurements as already explained in Section 4.2.2.

In online MoO, where trained centroids are needed, centroids are loaded from a file containing trained centroids during the start of the Flink Job.

Some comments regarding offline/batch MoO

As a starting point, we chose to use the *OfflineKMeans* implementation that comes out of the box of FlinkML. We changed the *KMeans* class to support our objects as described above, removed the *Estimator* interface and changed the return value of the *fit* method to be both the resulted *Centroid* object datastream and the resulted *Point* object datastream. Also, the API was changed from TableAPI to DatastreamAPI. The *KMeansIterationBody* inside the *KMeans* object was further changed to match what Figure 5.3 describes. The flow inside the *fit* method of the *KMeans* object is shown in Figure 7.1. The initial centroids are selected randomly. However, by using the same seed, we are able to use the same centroids each time to create a comparable accuracy measurement and a mean runtime. Batch mode is implemented by running multiple instances of our offline k-means implementation and assigning input points to different instances with fixed batch sizes. Resulted domains are not mixed between the instances before the labelling is complete, when calculating the accuracy.

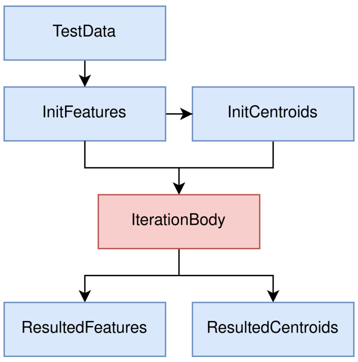

Figure 7.1: The data flow in our k-means implementation.

7.1.3 Environment

The experiments were performed by executing a fat JAR on a virtual computer from NTNU's Skyhigh solution. The VM had Debian 11 installed, had 8 GB of RAM and Java 11.0.15.

7.1.4 Reviewing the requirements for the experiments

RE1, in Table 4.4, is solved by introducing a counter on each distance call in the Vector objects. RE2 is solved by storing the label inside Point objects. RE3 is solved by using the framework presented in 5.2. Finally, RE4 is solved by making our offline MoO emit the final centroids for later usage.

7.2 Presentation of results

7.2.1 Speed in treatment A: Speed comparisons to A.1

The thesis use the relative speedup (*rel.speedup*) compared to the naïve implementation when presenting the speed of artefacts where treatment A is varied. To calculate the relative speedup, equation (7.1) is used. The times used in calculating rel.speedup are the mean of five executions. The percentage of skipped distance calculations compared to naïve k-means, Δd, are calculated using Equation (7.2), where d is the sum of distance calculations. Δd is also used when comparing artefacts where treatment A vary.

$$Rel.speedup = \frac{time_{naïve}}{time_{improvement}} \tag{7.1}$$

$$\Delta d = \frac{d_{naïve} - d_{improvement}}{d_{naïve}} * 100 \tag{7.2}$$

7.2.2 Speed in treatment B: Speed comparisons between artefacts

When comparing artefacts what vary in treatment B regarding speed, we provide the mean execution time, not the relative speed up, as there is no natural artefact to compare against. Also, the number of distance calculations is provided, not the percentage of skipped calculations, for the same reason.

7.2.3 Accuracy

As already mentioned, PPV is used to present the accuracy of the different artefacts. Equations (7.3) and (7.4) are used to calculate TPR and FPR, respectively. As given in Equation (7.5), BR tells us about the probability of an attack in our test data. PPV is calculated using Equation (7.6).

$$TPR = \frac{TP}{TP + FN} \tag{7.3}$$

$$FPR = \frac{FP}{FP + TN} \tag{7.4}$$

$$BR = \frac{TP + FN}{TP + FP + TN + FN} \tag{7.5}$$

$$PPV = \frac{BR * TPR}{BR * TPR + (1 - BR) * FPR} \tag{7.6}$$

7.3 Treatment A

In this section, we investigate the experiments run to explore treatment A, i.e., the use of triangle inequality to skip distance calculations. A.1-A.4 can be applied to Offline MoO, but as stated in Section 5.5, we cannot apply A.3 and A.4 to online k-means MoO. In Section 5.5, we suggest using Philip's idea (A.2) on the online k-means version. This is also what we have done.

Firstly, we tested A.1-A.4 on offline mode by testing each artefact five times and calculating the mean speed and number of distance calculations (d is the same for all artefacts). We used the NSL KDD Train data set in the same order, and the same initial centroids were chosen. The accuracy of each artefact was calculated to verify that A.1-A.4 provide the same result.

Secondly, the transform MoO was tested using A.1 (No TI) and A.2 (TI using Philip's idea). Also, here, we ran the experiments five times and calculated the mean speed. Finally, accuracy was calculated for validation. The NSL KDD Test data set was used in the same order for both artefacts and all executions. The initial centroids were the resulting centroids from running offline k-means on the NSL KDD Train data set.

Lastly, A.1 and A.2 were tested on sequential MoO in the same way as for the transform MoO.

7.3.1 Results of treatment A with Offline MoO

The results regarding the speed and capacity of the artefacts using different versions of treatment A in offline mode are presented in Table 7.2. The accuracy of the artefacts using different versions of treatment A is presented in Table 7.3. The table shows the TPR, FPR and the PPV of the various experiments performed. All artefacts perform the same when it comes to accuracy. Consequently, all four treatments are validated. Only A.2 shows a speedup compared to the rest despite reducing the number of distance calculations. In A.3 and A.4, the overhead may be too significant.

Table 7.2: Results of relevant speed calculations of treatment A in offline MoO.

Artefact	**Rel.speedup**	**Δd**
A.1 Naïve k-means	1.000	0.0%
A.2 Philips' algorithm	1.046	39.0%
A.3 Elkan's algorithm	0.923	81.1%
A.4 Hamerly's algorithm	0.983	59.8%

Table 7.3: Accuracy results for treatment A in offline MoO.

Artefact	**TPR**	**FPR**	**PPV**
A.1 Naïve k-means	0.641	0.078	0.729
A.2 Philips' algorithm	0.641	0.078	0.729
A.3 Elkan's algorithm	0.641	0.078	0.729
A.4 Hamerly's algorithm	0.641	0.078	0.729

Our implementation of k-means may be better in some domains than others. Therefore, in those domains, where both nomalies and anomalies exist, we can see the T/FPR and PPV for each domain in Appendix B when running our naïve implementation for reference.

7.3.2 Results of treatment A with Transform MoO

The results regarding the speed and capacity of the different artefacts using treatment A in the transform MoO are presented in Table 7.4. To validate the treatment (TI), both artefacts perform the same when it comes to accuracy. With $TPR = 0.323$ and $FPR = 0.075$, PPV becomes 0.774. A minor speedup is registered when using TI on the Transform MoO.

Table 7.4: Results of relevant speed calculations of treatment A in transform MoO.

Artefact	**Rel.speedup**	**Δd**
A.1 No TI	1.000	0.0%
A.2 TI	1.016	26.2%

7.3.3 Results of treatment A with Sequential MoO

The results regarding the speed and capacity of the different artefacts using treatment A in the sequential MoO are presented in Table 7.5. To validate the treatment (TI), both artefacts perform the same when it comes to accuracy. With $TPR = 0.323$ and $FPR = 0.074$ PPV becomes 0.775. The number of distance calculations increases when applying TI.

Table 7.5: Results of relevant speed calculations of treatment A in sequential MoO.

Artefact	**Rel.speedup**	**Δd**
A.1 No TI	1.000	0.0%
A.2 TI	0.991	−23.7%

7.4 Treatment B

In this part of our experimental work, we compare the different artefacts where treatment B vary, when the k-means algorithm are put in IDS. No triangle inequality is used.

As already stated, some artefacts testing treatments B differ in accuracy depending on the input order. Although this only applies to B.1 and B.3 (referencing Table 4.3), we test each artefact using five different input orders (the same five for all artefacts). Each order of input is further executed five times. This means that each artefact is executed $5 * 5 = 25$ times, and a mean is calculated after that.

7.4.1 Comparing the accuracy of different MoO

The accuracy of the different artefacts varying treatment B is presented in Table 7.6. The table shows the BR, TPR, FPR and the PPV of the various experiments performed. B.1 performs the poorest concerning PPV. However, the difference in PPV between the artefacts is tiny. B.3 performs the best. Note that B.1 has another BR than B.2 and B.3 since it have another starting point in terms of not having a trained model.

Table 7.6: Accuracy results for treatment B.

Artefact	**BR**	**TPR**	**FPR**	**PPV**
B.1 Batch	0.331	0.463	0.072	0.767
B.2 Transform	0.443	0.323	0.075	0.774
B.3 Sequential	0.443	0.323	0.074	0.775

7.4.2 Comparing the speed of different MoO

The results regarding the speed and capacity of the different artefacts varying treatment B are presented in Table 7.7. Both B.2 and B.3 have 44254 distance calculations. This corresponds to each point calculating distances to both centroids ($22127 * 2 = 44254$). B.3 has a mean execution time of about 6 minutes. However, it varied a lot with the order of input. For example, one seed gave an execution time of 1404340.4 milliseconds, while the four others had a mean of 82894.15 milliseconds.

Table 7.7: Results of relevant speed calculations of treatment B.

Artefact	**Mean milliseconds**	**Distance calculations**
B.1 Batch	13647.96	187834.8
B.2 Transform	2218.84	44254.0
B.3 Sequential	347183.40	44254.0

Chapter 8
Discussion

In this chapter, we discuss the work done in this thesis. We start with some general points regarding making a k-means-based IDS. After that, we will dive into the theoretical contributions and, by doing so, discuss how we have answered RQ1 and RQ2. Later in this chapter, we move on to our experimental work and answer RQ3 and RQ4. Finally, we discuss how one could build a better IDS based on the work done in this thesis. We also address limitations, drawbacks and implications in this chapter.

This thesis has explored different ways to speed up k-means-clustering-based IDS. A lot of effort is put into speeding up the k-means algorithm. However, not many of these improvements have been put in IDS, although efficient algorithms are crucial. Also, other forms of operating the k-means algorithm that may be faster but decrease the accuracy are interesting in IDS. As we can see, there is a trade-off between accuracy and speed. Also, we have explored the Apache Flink framework for application in IDS as it is a relatively new platform. Not much research has been put into studying the functionality that may support different modes of operation of the k-means algorithm.

8.1 Building an IDS

This section contains essential discoveries we have made while building an IDS.

8.1.1 Introducing domains

We have formalised the pre- and post-processing of data in k-means-based IDS by introducing domains in this thesis. Although the solution to non-numerical data in k-means was proposed in [YT11], we cannot see much research done on using the concept of domains in k-means-based IDS. Therefore, we have taken the idea presented in [YT11] and taken it further by integrating domains as a part of the k-means system on a streaming platform. Future work may address these ideas of integrating domains in the k-means system on classical computing platforms.

Domains were among the most significant contributions as they had many implications for interpreting results and operating the IDS. Therefore, some considerations are stated here. However, when diving into the experimental results, we will discover more implications later in this chapter.

First, we need to label clusters in each domain separately as two "normal" labelled clusters in two different domains may not correspond. This affects the process of measuring how accurate a k-means-based IDS using domains is and how these IDS will operate with the labelling algorithm.

Second, a signature-based IDS should assist an anomaly detection system in handling the domains that are malicious by nature. As discussed in Section 5.1, one should not automatically approve a domain since it may later be a source of a zero-day attack. At this point, we will revisit the *No Free Lunch* theorem that states that without any assumptions, there is no single algorithm that solves any problem. Assuming all domains have both anomalies and normal traffic, k-means-based IDS using domains may fit perfectly. However, this is not an assumption that holds for networks today, so we need signature-based IDS to support our system.

Third, points in one domain may arrive less frequently, making the domain wait (online) or being dropped (batch) in the clustering process. This also supports the statement that signature-based IDS should assist anomaly-based IDS.

8.1.2 Different modes of operation (MoO)

We have explored and concretised different modes of operation (MoO). We have also operated with two distinctions when talking about the different MoOs.

Offline k-means and online versions of k-means

Offline k-means works when all points are presented in advance. When put in IDS context, we call offline batch as we divide the continuous stream of data into batches on which the offline k-means algorithm performs.

On the other hand, online modes will not require all points to be known in advance and will assign a cluster to each point when it arrives. Two online modes have been used in this thesis.

Sequential k-means is a mode that updates the centroids each time an incoming point has been assigned a cluster. The transforming k-means mode is not a mode we can read about in the literature. Technically, transforming k-means is not a k-means algorithm, in the sense of not calculating centroids, just assigning points to their nearest centroid. It is just a classifier trained by running offline k-means that outputs centroids the transforming k-means will use. However, this mode may be interesting for application in IDS as it is fast.

Trained and not trained k-means

In k-means, there is nothing that is called training. However, we have introduced training to the k-means algorithm to explore different ways to operate the algorithm. Trained k-means means the initial centroids are calculated from another execution of the k-means algorithm, for example, the transforming k-means.

In batch k-means, this means the initial centroids are not chosen randomly from the pool of points in the batch or by using k-means++, but rather stem from another execution of the k-means algorithms—for example, the former batch. Sequential k-means' initial centroids may be the first points that arrive, chosen randomly from the first 1000 points (not trained) or stem from another execution of the k-means algorithm (trained). As already pointed out, the weight of each centroid must be set accordingly.

When introducing training in k-means, we also need to revisit the previously discussed challenges with machine learning as described in Section 2.2.2. *Overfitting* is not originally a problem in k-means-based machine learning systems as no training is involved. However, in this thesis, we have introduced training. So, *overfitting* is a problem in transforming k-means. Not as much in batch and sequential k-means since the centroids are allowed to change and update.

8.2 Apache Flink for IDS and k-means improvements

8.2.1 The new framework for implementing k-means

Using the framework introduced in Section 5.2 and the adjusted pseudo-codes in Section 5.4, the improvements to the k-means algorithm based on triangle inequality can be implemented in Apache Flink. There are many ways to do this. We introduced objects and may introduced further overhead to ensure the implementations were comparable. However, there is no doubt there exist more efficient ways to implement each improvement if it should not be compared.

Although we had $k = 2$ in our IDS, we have developed a framework that supports $k > 2$. If $k > 2$, Sort-Means and Drake's algorithm (see Section 2.3.2) would be interesting to test and compare. However, a method for labelling $k > 2$ clusters should be further investigated.

Also, different operation modes were built using building blocks from Section 5.2. These included operators such as *PointUpdater*, *CentroidUpdater* and *CentroidInitialiser*. These building blocks give the community a framework when discussing and describing improvements and modes of operation of the k-means algorithm.

8.2.2 Triangle inequality in online algorithms

In Section 5.5, we also introduced triangle inequality to online k-means, by taking Philip's idea of using centre-to-centre distances, into online modes.

8.2.3 Adjusting existing solutions

We also based our k-means implementation on the built-in *OfflineKMeans* Implementation in FlinkML. However, the convergence of this implementation needed to be fixed. By introducing filters after the *UpdatePoint* operator, we could filter out finished points and centroids, leaving points and centroids of domains that had not yet converged. One may argue our introduction of domains makes the algorithm converge faster, besides being a correct way to solve non-numerical attributes instead of ignoring these attributes or assigning these nominal values to numbers as other research may have done.

8.2.4 Answering RQ1 and RQ2

RQ1: How can the different improvements of the k-means algorithm be adjusted to operate with Apache Flink?

By introducing the framework presented in Section 5.2 with the adjusted improvements of the k-means algorithm presented in Section 5.4, we have made it possible to implement Philips', Elkan's and Hamerly's improvements in Apache Flink, as well as a naïve implementation.

RQ2: How can k-means on Apache Flink be used in intrusion detection applications?

Working with this research question has introduced multiple concepts to make an IDS in Flink. It gave rise to an analysis of some already known, and introduction of new, modes of operations in Chapter 4 and earlier in this chapter. It also gave us the concept of domains, and we altered the *OfflineKMeans* implementation in FlinkML to stop when the algorithm had converged.

8.3 Interpretations of the experimental results

8.3.1 Treatment A

In this part of the experiments, we wanted to test the different improvements of the k-means algorithm utilizing the triangle inequality to compare the speed when put in an IDS.

Offline mode

As we have seen, the improvements' overhead outweighs the gained speed from reducing distance calculations, at least when parallelism is 1. This is also what Ringdalen found out in [Rin20] when he only tested Elkan's algorithm. However, we also tested Hamerly's and Philips' algorithms and got a minor speedup using only centroid-to-centroid-distances (Philips' Compare-Means algorithm). Probably this is because Philips' algorithm does not have a very advanced logic for each point update, and the overhead, in general, is smaller than for Elkan's and Hamerly's algorithms (where also all points need to store extra information, not just the centroids). Therefore, future work can concentrate on one method and use Tuples instead of Objects in the datastream to decrease overhead.

Online modes

As already mentioned, we took triangle inequality to the online algorithms to test. However, as already stated, triangle inequality was only effective in the transform mode of operation. As we can see in the results for Transform MoO, not allowing centroids to move is beneficial when using TI in an online version, as opposed to Sequential MoO. Distance-to-distance calculations are only performed once in Transform MoO. However, the benefit of applying TI in online k-means are minor. Using triangle inequality in sequential k-means will not be more effective since the number of distance calculations increases when utilizing TI on Sequential MoO. This proves that TI in sequential k-means is not exactly a great idea. This is because new centroid-to-centroid-distances must be calculated each time a point leaves. In batch mode, this only happens after one batch/iteration.

8.3.2 Treatment B

In the second part of our experiments, we want to compare different modes of operation. Some modes work faster than others, and some modes are more accurate than others. The question we wanted to answer was how much decrease in accuracy we tolerate in our IDS to increase the speed?

Both B.2 and B.3 perform better than B.1 in terms of accuracy. This is surprising as batch mode allows points to change clusters and further adjust based on future points. However, the *batchSize* may be too small, or this results from different initial centroids as B.2 and B.3 share initial centroids (trained), and B.1 chooses centroids randomly from the test data set. As we know, k-means will only find a local optimum, meaning other initial centroids will provide a different result and is essential for the outcome.

Another reason may illustrate the difference between trained and untrained modes of operation in k-means. The trained model given B.2 and B.3 may be too well suited for our test data set and give us a fake good result. This is substantiated by the fact that B.2 and B.3 have a poorer TPR and FPR than B.1. To underline the difference, we can look at the BR. B.1 has a different BR than the others since it does not depend on training data and will therefore include all domains with sufficient cardinality (not just the domains that had initial centroids provided). When training a model, i.e., getting "trained" centroids after running offline k-means, the algorithm will not perform clustering on domains with cardinality less than k. The test data set have domains that were not accepted during the training. Consequently, as we can read in Section 7.1, 174 of the 189 domains in the test data set will be included, not the remaining 15. The difference in domains included in the test set affects the BR (the probability of an attack). If TPR decreases in Equation 7.6, PPV will decrease. However, in the case of B.2 and B.3, this is compensated with increasing BR. This illustrates the importance of the relation between the training data and the real-world (here: test data), especially when working with domains. When working with domains, the requirements for our training data gets stricter (also on a domain level).

B.3 is surprisingly slow. This may be due to a combination of events that make points block the pipeline since all data points go through one operator. If a new point of the same domain as the former point arrives, it has to wait before new centroids for this domain enter the iteration. This may be solved using higher parallelism. However, domains must be grouped to ensure all operators have correct centroids. The fact that different orders influenced the speed that much (as seen in Section 7.7) supports this.

The fastest MoO is B.2, and the accuracy is quite good, considering the fact that this mode does not update the centroids at all. This is, again, maybe because the data set may have a bad BR that leads to a fake good result. Also, the test data is not a stream of ordered events that may make centroids shift in batch (B.1) and sequential (B.3) mode.

8.3.3 Answering RQ3 and RQ4

RQ3 was about the accuracy of IDS using the different versions of the k-means algorithm. RQ4 was about the speed of IDS applying the various versions of the k-means algorithm. Both questions are answered in Chapter 7 and summarized above.

Generally, our IDS is quite bad in accuracy, especially when looking at TPR. This may be because NSL KDD has many features and our IDS suffers from the *curse of dimensionality*, as described in Section 2.2.2. Therefore, signature-based IDS should support our system.

8.3.4 Limitations and drawbacks

Our experimental work bears the mark that we have not run the experiments in parallel on a distributed Flink cluster. Instead, it was only run by executing fat JARs. This significantly influences our speed results (not distance calculations or accuracy). It would be interesting to run these experiments by increasing the parallelism and running the artefacts on an actual Flink cluster.

Future work can also test with data sets where events are ordered by timestamp, meaning we can observe a realistic drift in centroids in batch and sequential modes while the centroids stay the same in transforming k-means. How bad will the accuracy of transforming k-means be then?

As briefly discussed in Section 8.1.2, there are multiple ways to operate k-means regarding the MoO, choice of initial centroids (trained or not trained) and other parameters. When we compared B.1-B.3, these settings were not the same. For instance, we did not train B.1 from training data as the other two were. This also gave us the accuracy results as previously discussed. We chose the settings like how one would have used each artefact in an IDS setting. Future work may find combinations of settings, initial centroids and MoO that are more comparable.

8.4 Building a better IDS

We have tested different treatments to the problem that *naïve k-means sometimes is too slow for application in IDS*. If offline or batch mode, Philip's improvement to the k-means algorithm is a good choice if parallelism is 1. However, if parallelism increases, Elkan's or Hamerly's algorithms may be a good choice.

Since IDS work in real-time, online algorithms are desirable. Transforming k-means is very fast, even if parallelism is 1. Sequential k-means may be a good solution if the speed increases and gets better with higher parallelism. TI can be applied to transforming k-means, not sequential.

At this point, we revisit the functional requirements (FRs) stated in Table 4.1. FR3 is solved by using Philips' algorithm on offline/transform if the parallelism is one. Future work may give more options when parallelism is higher. The rest of the FRs are trivial and are answered in this thesis.

8.4.1 Proposal for new MoO: Online-Batch

To bridge the gap between the different MoOs, we present a new conceptual MoO that has not been tested, but the idea is shown here. This mode works in online mode by assigning incoming points to a cluster immediately from already-trained centroids and emitting this point (much like transform MoO). The point is stored, and for every 1000 points, the offline k-means algorithm is run until we have a new set of centroids that will be used to transform new points. Among the benefits, we get the speed of transformation MoO and can update the centroids used with the evolution of the network traffic.

Figure 8.1 shows the concept using the framework presented in Section 5.2. In the figure, we see an *IterationBody* that takes in new points, points from the last iteration and centroids from the previous iteration. The new points are updated in a *UpdatePoint* operator (as in Section 5.4) and then emitted. Each newly updated point is sent to a window operator that accumulates 1000 (can be another number) points before sending them and joins this datastream with a datastream in the orange area (offline k-means). These joint datastreams are sent to an *NCVOperator* and the next iteration. The rest is offline (referencing Section 5.3). However, the *FilterCentroid* operator will send finished centroids to the "transform part" of the algorithm, and the *FilterPoints* operator will not emit finished points, just terminate these points.

8.4.2 An IDS architecture

As k-means only will find a local optimum, we can use this fact to make an IDS architecture that consists of multiple k-means systems, each having different centroids. These systems may form a majority decision system or work like a Swiss cheese model[Rea00]. The initial centroids to these systems can be decided by giving each instance unique trained models (where the offline executions had different initial centroids or training data). The research on faster execution times for k-means may allow multiple instances of batch k-means to be part of this system. Here the initial centroids of a batch may be former resulted centroids from its own instance. This is an architecture that future work may look into to make anomaly-based IDS more robust.

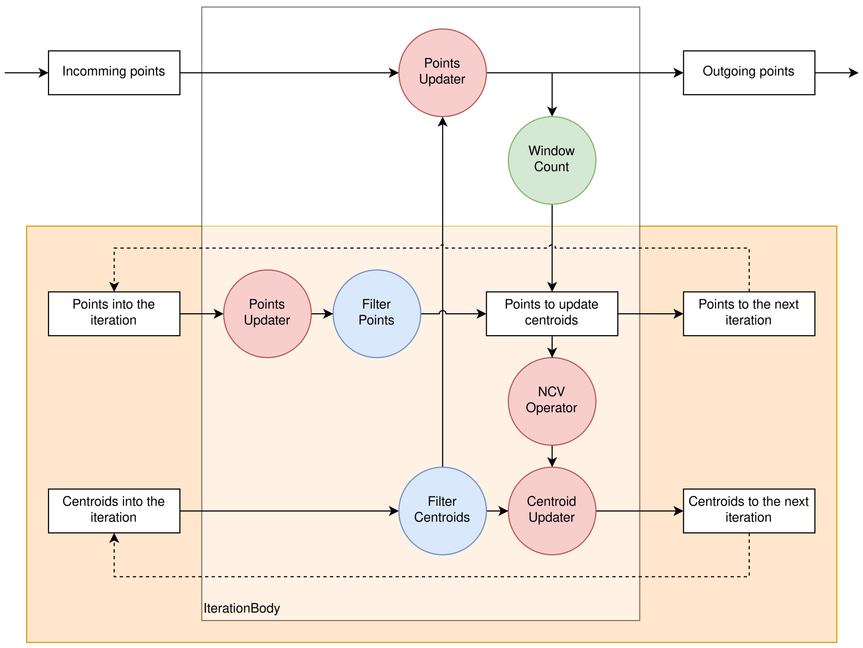

Figure 8.1: *Online-Batch* - Proposal for a new IDS bridging the gap between MoO.

8.5 Consequences

8.5.1 Consequences of the answers to RQ3 and RQ4

The clustering problem, to minimise the sum-squared error (SSE), see equation 2.1, is an NP-hard problem. The k-means algorithm is a widely used clustering algorithm in many fields that finds a local optimum in linear time (the optima depend on initial centroids, among other things). As a result, research on speeding up the k-means algorithm will have significant consequences for multiple fields.

The first method to speed up k-means-clustering we investigated, was the use of triangle inequality that we could not read much about in literature regarding IDS. Another method is varying the mode of operation. In some settings speeding up means lowering the accuracy as anomaly detection is a trade-off between speed and accuracy.

Speeding up the k-means algorithm may allow for finding multiple local optima in runtime (beneficial in the architecture proposed in Section 8.4.2). The findings in this thesis will be useful in the IDS community and other fields where clustering is used as an unsupervised machine learning method.

8.5.2 Consequences of the answers to RQ1 and RQ2

We have introduced a flexible framework for k-means on streaming platforms. Different improvements and operation modes are easy to implement and describe using this framework. The research community on the k-means algorithm may further use this framework when working with improving and exploring the possibilities and constraints in the k-means algorithm. Also, more effective initialisation techniques can be chosen and adjusted in Flink and our framework.

We have also contributed to the Apache Flink community by proposing methods to make the *OfflineKMeans* implementation converge and by implementing different improvements based on triangle inequality of the k-means algorithm and different modes of operation of the k-means algorithm.

Implications of introducing domains

One main contribution we have made is introducing domains. However, while doing this, we have seen multiple implications and considerations that need to be taken.

Labelling in cluster-based IDS is essential. Labelling of domains should be done separately, as already discussed. As pointed out in Section 2.2.1, the cluster with the highest cardinality often gets labelled "normal". When introducing domains, it is crucial to not use this labelling method as it is less likely that the highest-cardinality cluster is the "normal" cluster, as seen in Appendix B.

We also introduce multiple issues we cannot read about in literature by introducing domains. As domains should be included in a k-means-based IDS as a k-means-based IDS without domains should not be effective, we should address these issues. These issues are either solved by having a signature-based IDS or stricter requirements for our training data (if trained k-means).

As already discussed, some domains may be malicious by nature, giving the need for signature-based IDS to support our anomaly-based one.

Our k-means-based IDS perform poorly when tested on NSL KDD (in case of accuracy). This is because there should be enough normal and anomalies in each domain, but this is not the case in NSL KDD. Especially when testing trained k-means. The cut between domains in the training and test data sets with enough normal traffic and anomalies may be too small. And those domains that pass the training phase may give faulty results during the test phase. NSL KDD may still be a good data set for other detection systems, but when introducing domains in k-means, NSL KDD may not be an ideal data set. This emphasises the importance of using a data set that reflects the real world and using signature-based IDS on those domains where IDS performed poorly.

Also, a limitation to the results given during the experimental work of this thesis is due to the difference in domains included (which is an already described problem when using NSL KDD with domains). Therefore, it would be interesting to test a modified version of NSL KDD where small domains are not present in both sets or other data sets.

Some requirements for a trained k-means system follow. First, as already mentioned, 15 domains were not tested when testing B.2 and B.3 since we did not train our IDS for those domains. In our implementation, these domains would not be processed. If we meet a case where the trained algorithm does not have a model for a domain, signature-based IDS should assist. Secondly, our training data need enough normal and abnormal data points in all domains the k-means-system should process.

Chapter 9 Conclusion

This thesis explores new ways to think of k-means. We have built a better k-means-cluster-based IDS by investigating different options for speeding up the k-means algorithm and exploring different operation modes.

We have taken improvements of k-means and put them in the IDS context. We have also explored different modes of operation for k-means that may be suitable for IDS. And even further, we have implemented those improvements and different modes of operations of the k-means algorithm in Apache Flink. These improvements have then been compared in terms of speed, and the modes have been compared in terms of both speed and accuracy.

In this chapter, we will go through the answers to our research questions and describe what future work may be. Proposal for a new and better IDS is outlined in Section 8.4.

9.1 Answering RQs

RQ1: How can the different improvements of the k-means algorithm be adjusted to operate with Apache Flink?

By introducing the framework presented in Section 5.2 with the adjusted improvements of the k-means algorithm presented in Section 5.4, we have made it possible to implement Philips', Elkan's and Hamerly's improvements in Apache Flink, as well as a naïve implementation.

The framework consisted of *IterationBody*, datastreams, some new objects and operators. The operators were the *PointUpdater*, *CentoridUpdater* and *CentroidInitialiser*. The objects were the *Point* and *Centroid* objects that consisted of the unique logic of each improvement of the k-means algorithm. By using methods in those objects, we have made a flexible framework that may be used in the research and education area of k-means.

RQ2: How can k-means on Apache Flink be used in intrusion detection applications?

Working with this research question has introduced multiple concepts to make an IDS in Flink. It gave rise to an analysis of some already known, and introduction of new, modes of operations in Chapters 4 and 8. It also gave us the concept of domains as a solution to data sent to IDS that contain non-numeric features that cannot be included in the k-means algorithm. Also, we altered the *OfflineKMeans* implementation in FlinkML to stop when the algorithm had converged. The already mentioned framework enabled us to implement different modes of operation in Apache Flink easily.

RQ3: What is the accuracy of an IDS using the different versions of the k-means algorithm?

This question was answered by utilising and testing treatments A and B in Chapter 7 and discussed in Section 8.3. All improvements based on triangle inequality have the same accuracy as a naïve implementation, as already hypothesised.

Our experiments showed that sequential k-means perform better than transforming k-means in terms of accuracy. This was already hypothesised. However, transforming k-means performed surprisingly good. This may be due to the data set used (as already discussed). We have concluded that the data set in combination with domains is why batch k-means perform surprisingly poorly in the case of accuracy. The complete results are shown in Appendix A.

RQ4: What is the speed of an IDS applying the different versions of the k-means algorithm?

This question was answered by adjusting treatments A and B in Chapter 7 and discussed in Section 8.3. Philips's approach using centroid-to-centroid distances is a good choice if parallelism is 1. Philips' algorithm can be used offline/batch mode or transform mode. Elkan's and Hamerly's algorithms do not perform better due to the overhead introduced. However, this might change if parallelism is higher.

The modes of operation also differ in speed. For example, sequential may be too slow for application in IDS, especially if parallelism is 1. On the other hand, transforming k-means is a really fast MoO. The complete results are shown in Appendix A.

9.2 Future work

Much future work is proposed in Chapter 8. However, some key points are summarized here. First, future work may implement and test our newly proposed mode of operation called *Online-Batch*, as described in Section 8.4.1.

Multiple limitations of our experimental work have been pointed out in Section 8.3.4 that may give rise to future work. One of the most important ones is that we did not test with higher parallelism than one and did not execute the Flink Job in a Flink Cluster but instead as a fat JAR. This execution environment has influenced the speed comparisons in our experiments. Future work should test these artefacts with higher parallelism. Probably, some MoO and improvements of k-means may perform better. Especially sequential mode, as it still is a very interesting MoO to put in IDS. Another thing that may influence the experimental results is that different initial centroids were used when testing treatment B. Future work may explore different combinations of choosing initial centroids (referencing Section 8.3.4).

Regarding k-means in other applications, Drake's method is a significant improvement of the k-means algorithm that should be included. Sort-Means should also be tested. Treatments A and B with $k > 2$ should be investigated as well.

B.1 and B.3 artefacts will adjust with the change of networks. Transforming k-means will not. Future work may find test data ordered by timestamp and test B.1-B.3 to see how bad the accuracy is for transforming k-means when centroids are not allowed to drift.

References

[ADHP09] D. Aloise, A. Deshpande, *et al.*, «Np-hardness of euclidean sum-of-squares clustering», *Machine learning*, vol. 75, no. 2, pp. 245–248, 2009.

[Al +17] S. Al Ghamdi *et al.*, «Efficient parallel k-means on mapreduce using triangle inequality», in *2017 IEEE 15th Intl Conf on Dependable, Autonomic and Secure Computing, 15th Intl Conf on Pervasive Intelligence and Computing, 3rd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech)*, IEEE, 2017, pp. 985–992.

[Art+06] D. Arthur *et al.*, «K-means++: The advantages of careful seeding», in *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, 2006, pp. 1027–1035.

[Cel15] M. E. Celebi, «Partitional clustering algorithms», in Springer, 2015, ch. 2.

[CKV13] M. E. Celebi, H. A. Kingravi, and P. A. Vela, «A comparative study of efficient initialization methods for the k-means clustering algorithm», *Expert systems with applications*, vol. 40, no. 1, pp. 200–210, 2013.

[DG04] J. Dean and S. Ghemawat, «Mapreduce: Simplified data processing on large clusters», in *6th Symposium on Operating Systems Design and Implementation*, San Francisco, CA, Oct. 2004, pp. 137–150.

[DG17] D. Dua and C. Graff, *UCI machine learning repository*, Description of KDD Cup 1999 Data Data Set, 2017. [Online]. Available: http://archive.ics.uci.edu/ml (last visited: Jun. 13, 2022).

[DH12] J. Drake and G. Hamerly, «Accelerated k-means with adaptive distance bounds», in *5th NIPS workshop on optimization for machine learning*, vol. 8, 2012.

[Docs] *Apache flink documentation*, The Apache Software Foundation. [Online]. Available: https://nightlies.apache.org/flink/flink-docs-release-1.15/ (last visited: Jun. 12, 2022).

[Dom12] P. Domingos, «A few useful things to know about machine learning», *Communications of the ACM*, vol. 55, no. 10, pp. 78–87, 2012.

[Dud97] R. O. Duda, *Sequential k-means clustering*, Jun. 1997. [Online]. Available: https://www.cs.princeton.edu/courses/archive/fall08/cos436/Duda/C/sk_means.htm (last visited: Jun. 10, 2022).

[Elk03] C. Elkan, «Using the triangle inequality to accelerate k-means», in *Proceedings of the 20th international conference on Machine Learning (ICML-03)*, 2003, pp. 147–153.

[For65] E. Forgy, «Cluster analysis of multivariate data: Efficiency vs. interpretability of classification», *Biometrics*, vol. 21, no. 3, pp. 768–769, 1965.

[Ham10] G. Hamerly, «Making k-means even faster», in *Proceedings of the 2010 SIAM international conference on data mining*, SIAM, 2010, pp. 130–140.

[HSD00] P. E. Hart, D. G. Stork, and R. O. Duda, *Pattern classification.* Wiley Hoboken, 2000.

[HW79] J. A. Hartigan and M. A. Wong, «Algorithm as 136: A k-means clustering algorithm», *Journal of the royal statistical society. series c (applied statistics)*, vol. 28, no. 1, pp. 100–108, 1979.

[Kog07] J. Kogan, *Introduction to clustering large and high-dimensional data.* Cambridge University Press, 2007.

[Les+20] J. Leskovec *et al.*, *Mining of Massive Datasets, 3rd Edition.* Cambridge University Press, 2020.

[LG22] D. Lin and Y. Gao, *Apache flink ml 2.0.0 release announcement*, The Apache Software Foundation, Jan. 2022. [Online]. Available: https://flink.apache.org/news/2022/01/07/release-ml-2.0.0.html.

[Llo82] S. Lloyd, «Least squares quantization in pcm», *IEEE transactions on information theory*, vol. 28, no. 2, pp. 129–137, 1982.

[LO13] P. D. Leedy and J. E. Ormrod, *Practical Research: Planning and design, tenth edition.* Pearson, 2013.

[Mac+67] J. MacQueen *et al.*, «Some methods for classification and analysis of multivariate observations», in *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, Oakland, CA, USA, vol. 1, 1967, pp. 281–297.

[Mac21] *Apache spark vs. flink, a detailed comparison*, From Macrometa, Sep. 2021. [Online]. Available: https://www.macrometa.com/event-stream-processing/spark-vs-flink (last visited: Jun. 12, 2022).

[MLDocs] *Apache flinkml documentation*, The Apache Software Foundation. [Online]. Available: https://nightlies.apache.org/flink/flink-ml-docs-release-2.0/ (last visited: Jun. 12, 2022).

[PAOC06] S. Petrovic, G. Alvarez, *et al.*, «Labelling clusters in an intrusion detection system using a combination of clustering evaluation techniques», in *Proceedings of the 39th Annual Hawaii International Conference on System Sciences (HICSS'06)*, IEEE, vol. 6, 2006, 129b–129b.

[Pet20a] S. Petrovic, *IDS - Topic 1 - IDS/IPS definition and classification*, Lecture notes from the IMT4204 course at NTNU Gjøvik., 2020.

[Pet20b] ——, *IDS - Topic 4 - Testing IDS*, Lecture notes from the IMT4204 course at NTNU Gjøvik., 2020.

[Phi02] S. J. Phillips, «Acceleration of k-means and related clustering algorithms», in *Workshop on Algorithm Engineering and Experimentation*, Springer, 2002, pp. 166–177.

[Rea00] J. Reason, «Human error: Models and management», *Bmj*, vol. 320, no. 7237, pp. 768–770, 2000.

[Rin20] O. F. Ringdalen, «Applying k-means with triangle inequality on apache flink, with applications in intrusion detection», M.S. thesis, NTNU, 2020.

[Sty] A. Styrmoe, *Speeding Up Clustering-Based Anomaly Detection Systems*, Pre-project for the master thesis 2021.

[TBLG09] M. Tavallaee, E. Bagheri, *et al.*, «A detailed analysis of the kdd cup 99 data set», in *2009 IEEE symposium on computational intelligence for security and defense applications*, Ieee, 2009, pp. 1–6.

[UCI99] *Kdd-cup-99 task description*, From Donald Bren School of Information and Computer Sciences, University of California, Sep. 1999. [Online]. Available: http://kdd.ics.uci.edu/databases/kddcup99/task.html (last visited: Jun. 16, 2022).

[UNB] *Nsl-kdd dataset*, Canadian Institute for Cybersecurity, UNB. [Online]. Available: https://www.unb.ca/cic/datasets/nsl.html (last visited: Jun. 13, 2022).

[Wie14] R. J. Wieringa, *Design Science Methodology for Information Systems and Software Engineering.* Springer, 2014.

[WM97] D. Wolpert and W. Macready, «No free lunch theorems for optimization», *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 67–82, 1997.

[YT11] Z. Yu and J. J. Tsai, *Intrusion Detection: A Machine Learning Approach.* World Scientific Publishing Co. Pte. Ltd., 2011.

Appendix A
Results of experiments performed

The following table shows each experiment that was run. Each row is executed five times, and the mean speed is provided in milliseconds. The first column is for treatment A, i.e., show if triangle inequality was used and, if so, which variant. The second column is for treatment B or shows the mode of operation. For example, when comparing artefacts comparing treatment A in offline mode, B.1 (batch) was not used but rather an offline mode where the whole data set was run not in batches (Section 7.3.1).

The column *Data* shows whether the train or test data set of NSL KDD is used. The number after "test" in the *Data* column indicated which order of events was used (seed to generate a new random order).

In Section 7.3, we compared varying treatment A artefacts using different MoO (offline, B.2 and B.3). No variation in test data was used.

In Section 7.4, we compared varying treatment B artefacts without triangle inequality (just A.1). The mean of Test0-Test4 was used.

A	**B/MoO**	**Data**	**TPR**	**FPR**	**PPV**	**Mean speed**	**Dist. calcs.**
A.1	Offline	Train	0,641	0,078	0,729	34527,2	2549834
A.2	Offline	Train	0,641	0,078	0,729	33005,4	1556577
A.3	Offline	Train	0,641	0,078	0,729	37403,6	482019
A.4	Offline	Train	0,641	0,078	0,729	35129,2	1024901
A.1	B.1	Test0	0,475	0,071	0,769	13243,6	182410
		Test1	0,452	0,075	0,761	11967,8	178448
		Test2	0,478	0,078	0,752	15686,4	186510
		Test3	0,462	0,067	0,783	13629,8	195576
		Test4	0,446	0,070	0,771	13712,2	196230
A.2	B.1	Test0	0,475	0,071	0,769	12133,2	122131
A.3	B.1	Test0	0,475	0,071	0,769	12423,8	69477
A.4	B.1	Test0	0,475	0,071	0,769	12210,8	157265
A.1	B.2	Test0	0,323	0,075	0,774	2124,2	44254
		Test1	0,323	0,075	0,774	2166,0	44254
		Test2	0,323	0,075	0,774	2273,6	44254
		Test3	0,323	0,075	0,774	2217,8	44254
		Test4	0,323	0,075	0,774	2312,6	44254
A.2	B.2	Test0	0,323	0,075	0,774	2091,4	32675
A.1	B.3	Test0	0,323	0,074	0,775	1404340,4	44254
		Test1	0,322	0,075	0,775	93761,6	44254
		Test2	0,323	0,074	0,775	65114,4	44254
		Test3	0,323	0,075	0,773	73959,4	44254
		Test4	0,323	0,074	0,775	98741,2	44254
A.2	B.3	Test0	0,0323	0,074	0,775	1417594,6	54721

Appendix B
Accuracy results by domain

The following table shows the results for each domain that had both normal points and anomalies in test data after running offline mode. In addition, the table shows the number of points assigned to two clusters, A and B, and the number of normal points in each of the clusters (N+CA and N+CB).

Domain	CA	N + CA	CB	N + CB	TP	FP	TN	FN	BR	TPR	FPR	PPV
tcpftpSF	386	386	825	503	0	386	503	322	0,266	0,000	0,434	0,000
tcpsshSF	5	5	2	0	2	0	5	0	0,286	1,000	0,000	1,000
tcpIRCREJ	1	0	1	1	1	0	1	0	0,500	1,000	0,000	1,000
tcpX11REJ	8	6	8	6	2	6	6	2	0,250	0,500	0,500	0,250
tcpauthSF	119	115	103	101	2	101	115	4	0,027	0,333	0,468	0,019
tcphttpS0	403	0	979	261	403	0	261	718	0,811	0,360	0,000	1,000
tcphttpS1	162	160	10	10	0	10	160	2	0,012	0,000	0,059	0,000
tcphttpS2	1	1	91	85	0	1	85	6	0,065	0,000	0,012	0,000
tcphttpSF	34698	34692	979	120	859	120	34692	6	0,024	0,993	0,003	0,877
tcpsmtpS0	106	0	163	67	106	0	67	96	0,751	0,525	0,000	1,000
tcpsmtpSF	24	24	6862	6835	0	24	6835	27	0,004	0,000	0,003	0,000
tcptimeS0	358	0	88	8	358	0	8	80	0,982	0,817	0,000	1,000
tcptimeSF	72	61	2	0	2	0	61	11	0,176	0,154	0,000	1,000
tcptimeSH	1	0	2	2	1	0	2	0	0,333	1,000	0,000	1,000
tcpauthREJ	47	17	108	0	108	0	17	30	0,890	0,783	0,000	1,000
tcpftpRSTO	1	0	128	16	1	0	16	112	0,876	0,009	0,000	1,000
tcphttpREJ	809	615	2001	2000	194	615	2000	1	0,069	0,995	0,235	0,240
tcpimap4SF	8	2	2	1	1	1	2	6	0,700	0,143	0,333	0,500
tcpotherS2	3	2	1	1	0	1	2	1	0,250	0,000	0,333	0,000
tcpotherSF	2	0	302	299	2	0	299	3	0,016	0,400	0,000	1,000
tcppop_3SF	2	2	185	182	0	2	182	3	0,016	0,000	0,011	0,000

Domain	**CA**	**N + CA**	**CB**	**N + CB**	**TP**	**FP**	**TN**	**FN**	**BR**	**TPR**	**FPR**	**PPV**
tcpshellSF	3	2	3	0	3	0	2	1	0,667	0,750	0,000	1,000
tcpsmtpOTH	1	1	1	0	1	0	1	0	0,500	1,000	0,000	1,000
udpotherSF	2074	1837	414	414	0	414	1837	237	0,095	0,000	0,184	0,000
icmpeco_iSF	333	315	4253	182	4071	182	315	18	0,892	0,996	0,366	0,957
icmpecr_iSF	2197	0	880	190	2197	0	190	690	0,938	0,761	0,000	1,000
icmptim_iSF	7	4	1	1	0	1	4	3	0,375	0,000	0,200	0,000
icmpurp_iSF	307	307	295	292	3	292	307	0	0,005	1,000	0,487	0,010
tcpauthRSTO	1	0	3	3	1	0	3	0	0,250	1,000	0,000	1,000
tcpdomainS0	340	0	74	1	340	0	1	73	0,998	0,823	0,000	1,000
tcpdomainSF	32	32	12	5	7	5	32	0	0,159	1,000	0,135	0,583
tcpfingerS0	325	0	790	12	325	0	12	778	0,989	0,295	0,000	1,000
tcpfingerSF	267	234	293	286	33	234	286	7	0,071	0,825	0,450	0,124
tcphttpRSTR	76	0	24	3	76	0	3	21	0,970	0,784	0,000	1,000
tcpotherREJ	905	0	48	44	905	0	44	4	0,954	0,996	0,000	1,000
tcpsmtpRSTO	7	7	73	27	0	7	27	46	0,575	0,000	0,206	0,000
tcpsmtpRSTR	5	1	1	0	1	0	1	4	0,833	0,200	0,000	1,000
tcptelnetS0	332	0	908	5	332	0	5	903	0,996	0,269	0,000	1,000
tcptelnetS3	3	0	1	1	3	0	1	0	0,750	1,000	0,000	1,000
tcptelnetSF	790	740	20	20	0	20	740	50	0,062	0,000	0,026	0,000
tcptimeRSTO	64	0	55	1	64	0	1	54	0,992	0,542	0,000	1,000
tcppop_3RSTO	13	0	3	1	13	0	1	2	0,938	0,867	0,000	1,000

Domain	**CA**	**N + CA**	**CB**	**N + CB**	**TP**	**FP**	**TN**	**FN**	**BR**	**TPR**	**FPR**	**PPV**
tcpprivateSF	4	0	2	2	4	0	2	0	0,667	1,000	0,000	1,000
tcpshellRSTO	1	1	7	0	7	0	1	0	0,875	1,000	0,000	1,000
tcptelnetOTH	2	1	1	1	0	1	1	1	0,333	0,000	0,500	0,000
udpprivateSF	2118	21	1173	957	2097	21	957	216	0,703	0,907	0,021	0,990
tcpfingerRSTO	77	7	1	0	1	0	7	70	0,910	0,014	0,000	1,000
tcpfingerRSTR	3	2	4	0	4	0	2	1	0,714	0,800	0,000	1,000
tcpftp_dataS2	5	4	1	1	0	1	4	1	0,167	0,000	0,200	0,000
tcpftp_dataSF	269	213	5272	4714	56	213	4714	558	0,111	0,091	0,043	0,208
tcpprivateREJ	2535	1	1430	0	1430	0	1	2534	1,000	0,361	0,000	1,000
tcptelnetRSTO	188	53	3	3	0	3	53	135	0,707	0,000	0,054	0,000
tcptelnetRSTR	1	1	29	20	0	1	20	9	0,300	0,000	0,048	0,000
udpdomain_uSF	2556	2547	6487	6487	9	2547	6487	0	0,001	1,000	0,282	0,004

Aleksander Styrmoe

Speeding Up k-Means-Clustering-Based Anomaly Detection Systems

NTNU

Norwegian University of Science and Technology