Master's thesis

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

Ola Fløtlo Ringdalen

Applying K-means with Triangle Inequality on Apache Flink, with Applications in Intrusion Detection

Master's thesis in Information Security
Supervisor: Slobodan Petrovic
July 2020

NTNU
Norwegian University of Science and Technology

Ola Fløtlo Ringdalen

Applying K-means with Triangle Inequality on Apache Flink, with Applications in Intrusion Detection

Master's thesis in Information Security
Supervisor: Slobodan Petrovic
July 2020

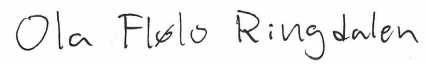
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

NTNU
Norwegian University of
Science and Technology

Preface

This master thesis project was carried out during the spring-semester of 2020, at the end of my fifth and final year at the Norwegian University of Science and Technology, at the Faculty of Information Technology and Electrical Engineering. The research topic in this thesis was proposed by professor Slobodan Petrović during a lecture in *Intrusion Detection in Physical and Virtual Networks* in the autumn of 2019. This thesis is written for an audience with some knowledge of technical topics; however, background material will be provided as part of this thesis.

June 29, 2020



Ola Fløla Ringdalen

Acknowledgement

First and foremost I would like to thank my supervisor professor, Slobodan Petrović, for his guidance and the expertise he provided during the work on this thesis. During this work he has always been helpful if I had any questions and still encouraged me to solve problems on my own. Furthermore, I would like to thank my girlfriend, Anine, for being supportive during this last semester. Lastly, I would like to thank my parents for always encouraging and supporting me in the many years of my education.

O. F. R.

Abstract

The well known clustering algorithm k-means is quite naive in its operation and many of the distance calculations it performs are unnecessary. Earlier research has shown that by exploiting the triangle inequality theorem the majority of distance calculations can be avoided, while still providing the same clustering result. In this thesis we aim to adjust k-means exploiting triangle inequality to operate on the parallel processing framework Apache Flink. Additionally, we evaluate the performance of both k-means and k-means with triangle inequality in an intrusion detection system environment.

The performance is evaluated by using a quantitative research approach where we apply a within-subjects research design to collect data. In the experiment we utilize the well known NSL KDD intrusion detection dataset. Two main experiments were performed. One where we kept the number of iterations static and varied the number of clusters, and a second experiment where we kept the number of clusters static and varied the number of iterations. Both experiments were repeated with a varying degree of parallelism.

Our results show that for a large number of clusters there are no increase in performance when clustering with k-means exploiting triangle inequality. A large number of clusters caused a large overhead in the iteration function of Apache Flink. However, with a lower amount of clusters and when performing the clustering over many parallel instances, a performance increase of up to 8.8% is observed. Furthermore, when evaluating the algorithms with a varying number of iterations we observed that there was a performance increase for all iteration values. The increase was most significant for a lower number of iterations. In an intrusion detection setting where a low number of clusters are used, the results are promising, but further research is needed in order to reduce the overhead and increase the performance further.

Sammendrag

K-means er en av de mest brukte kluster-algoritmene, men den fungerer på en relativt naiv måte og majoriteten av distansekalkulasjonene den utfører er unødvendige. Tidligere forskning har bevist at ved å utnytte triangelaksiomet er det mulig å unngå majoriteten av disse distansekalkulasjonene og samtidig ende opp med de samme klustrene. I denne avhandlingen justerer vi k-means algoritmen slik at den kan fungere med et rammeverket for parallell prosessering kalt Apache Flink. I tillegg til dette vil vi evaluere ytelsen til både den originale k-means algoritmen og k-means som utnytter triangelaksiomet, fra et inntrengings og deteksjonssystem perspektiv.

Ytelseevalueringen blir gjennomført ved å følge kvantitative forskningsmetoder og vi bruker et oppsett under eksperimentene hvor vi tester begge algoritmene på det samme datasettet. Under eksperimentene bruker vi et datasett med nettverkstrafikk laget for å evaluere inntrengings og deteksjonssystemer kalt NSL KDD. To eksperimenter ble gjennomført. Ett eksperiment hvor vi holdt antall iterasjoner likt, men endret på antall kluster. I det andre eksperimentet holdt vi antall kluster likt, men endret på antall iterasjoner. Begge disse eksperimentene ble gjennomført med varierende grad av parallellisme.

Resultatene viser at når man bruker k-means med triangelaksiomet og bruker mange klustre finnes det ingen ytelsesforbedringer, snarere tvert imot. Et stort antall klustre førte til mye ekstraarbeid for iterasjonsfunksjonen i Apache Flink. Ved å bruke et mindre antall klustre og kjøre programvaren med flere parallelle instanser så observerte vi en økning i ytelse på 8.8% med k-means med triangelaksiomet i motsetning til den originale k-means algoritmen. Videre, når vi evaluerte k-means med triangelaksiomet med et varierende antall iterasjoner observerte vi bedre ytelse for alle testede verdier i forhold til den originale k-means algoritmen. Fra et inntrengings og deteksjonssystem perspektiv er dette lovende, da det ofte er et mindre antall klustre som brukes. Vi anbefaler likevel videre forskning for å minimere ekstraarbeidet i iterasjonsfunksjonen og dermed øke ytelsen ytterligere.

Contents

Figures

Tables

Code Listings

Acronyms

API Application Programming Interface.

APT Advanced Persistent Threat.

DAG Directed Acyclic Graph.

HDFS Hadoop Distributed Filesystem.

HTTP Hypertext Transfer Protocol.

HUMINT Human Intelligence.

ICT Information and Communications Technology.

IDS Intrusion Detection System.

IPS Intrusion Prevention System.

OSINT Open Source Intelligence.

RAT Remote Access Trojan.

SQL Structured Query Language.

SSH Secure Shell.

Chapter 1

Introduction

In today's society vast amounts of information are being generated every day. Data from a huge number of applications, systems and networks are retained for a long period of time. With this increase in data volume, analysing the data for unwanted or malicious events becomes increasingly computationally demanding as well. During the last decade within the information technology sphere big data has emerged as a field to handle the vast amount of data that are being generated as efficiently as possible. In the security field enormous amounts of information are also being analyzed and traffic is being monitored for malicious activity. Machine learning methods have been introduced to extract anomalies that should be handled in order to avoid damages to an organization or an individual.

In this first chapter we introduce the topics covered in this thesis, together with a problem description. Moreover, research questions are presented and the scope and contributions for this thesis are given. Lastly, the structure of the thesis is described.

1.1 Keywords

A set of keywords is compiled for this project in order to describe the scope and make it easier to locate this project after its completion.

unsupervised machine learning, *clustering*, *anomaly detection*, *intrusion detection*, *k-means*, *triangle inequality*, *apache flink*

1.2 Problem description

K-means is one of the most used clustering algorithms used for unsupervised learning problems, or in other words, learning problems that does not rely on labeled data in order to make decisions. The original implementation of k-means is quite naive, as the algorithm calculates the distance for every object in the dataset to

each of the centroids. These calculations are being performed every iteration, but a large majority of these calculations are redundant when it comes to the final outcome of the clusters.

Triangle inequality has been proven to decrease the computational requirement with great success. In other works, k-mean has been implemented on distributed computing platforms such as Apache Hadoop and Apache Spark. However, to the best of our knowledge, such an implementation has not been adjusted to work with another up and coming platform called Apache Flink. One goal of this project is to adjust the k-means algorithm with triangle inequality on the Flink platform. Then we will research how this can be applied in intrusion detection use-cases. Thereafter, a study of the efficiency of the implementation can be performed. Circumstances where the clustering does perform well or does not perform well will be identified as well.

1.3 Research questions

Research questions have been formulated for this thesis and are forming the base of what is researched in this project.

- How could k-means clustering with triangle inequality be adjusted to operate with Apache Flink?
- How can we apply k-means clustering with triangle inequality on Apache Flink in intrusion detection applications?
- How much can k-means clustering with triangle inequality be optimized compared to regular k-means?
- Under what circumstances could the implementation of k-means with triangle inequality thrive?

Our hypothesis is that k-means with triangle inequality can be adjusted for Apache Flink. Furthermore we hypothesize that the average speedup for k-means with triangle inequality on Apache Flink compared to regular k-means on Apache Flink is greater than 50%, since the vast majority of the distance calculations being performed will be skipped.

1.4 Scope and contributions

The main focus of this thesis is to research how the performance of the k-means algorithm with triangle inequality can be increased when utilizing the Apache Flink framework. Additionally, we will investigate how any performance increases can be exploited in an intrusion detection system environment.

By the end of this project the contribution of this thesis will be a k-means algorithm employing triangle inequality that is adjusted in order to utilize the Apache Flink API for parallel processing. Furthermore, a comparison of the performance between k-means with and without triangle inequality will presented.

1.5 Thesis structure

The remainder of this thesis is structured as follows.

In **chapter 2** we cover background material relating to the scope of this thesis. First we present intrusion detection and prevention methods and then cover the topic of machine learning. Lastly, we present how modern cyber attacks are carried out by threat actors.

In **chapter 3** we present related work that this thesis is built upon, which include how the k-means algorithm work, how triangle inequality can be exploited decrease the number of distance calculations performed in said algorithm and then research on how this has been performed in other parallel frameworks.

With **chapter 4** we explain and discuss the choice of methods used in this thesis. This includes methods for how the software was developed and how the experiment is designed. Actions taken to ensure the validity of the research and dealing with bias is discussed as well. Additionally, we introduce the dataset used to evaluate the applications and a description of the experimental environment is also included.

In **chapter 5** we present the theoretical contribution of this thesis. With this chapter we introduce key concepts and methods we utilized to answer the research questions.

In **chapter 6** we present the results from the experiment. We evaluate the performance of regular k-means, versus the performance of k-means with triangle inequality. Only a subset of the full results are presented in this chapter, the full set of results can be found in the appendices.

In **chapter 7** we discuss and interpret the results presented in the previous chapter. Furthermore, we discuss the limitation and drawbacks of the selection solution.

With **chapter 8** a conclusion is given and and we end the thesis with suggestions for further work.

Chapter 2

Background

In this chapter we will present relevant background material to the reader of this thesis. The sections of this chapter will introduce different topics touched upon in the coming chapters. Topics presented here are not comprehensive, however a broad overview will help the reader put the subject of this thesis in to a broader context. We begin with presenting intrusion detection and prevention methods and then we introduce the topic of machine learning. Then we detail the composition of a cyber attack and end with discussing parallel processing and the MapReduce programming model.

2.1 Intrusion detection and prevention

Intrusion detection is a research field that have been studied quite a lot throughout the last couple of decades. With the ever growing amount of network traffic and security related events researchers in this field are still faced with new challenges as this increase continue. In this thesis we derive the definition of an intrusion [1] from the US agency NIST (National Institute of Standards and Technology), which states that an *intrusion* is:

> "*[..] any set of actions that attempt to compromise the integrity, confidentiality, or availability of a resource.*"

Furthermore, we derive NISTs definition of the term intrusion detection [2], which states that an *intrusion detection* is:

> "*[..] the process of monitoring the events occurring in a computer system or network and analyze them for signs of possible incidents*"

Systems with intrusion detection or prevention capabilities are often referred to as Intrusion Detection Systems (IDS) or Intrusion Prevention Systems (IPS). These are software systems that are designed to automate the process of detecting security incidents, and in some cases attempt to block these security incidents in

real-time. IDS and IPS include a feature set that is mostly overlapping with each other.

The main objective for such systems is to monitor and detect malicious activity, performed by a malicious actor. For instance, a malicious actor may compromise and gain access to a system within a monitored network by brute-forcing (the act of trying to log on using multiple passwords for an account) an account in the system. The IDS is designed to detect this incident and report it to a security resource within an organization (an analyst, system administrator, etc.), which will initiate the necessary actions in order to minimize damage to the compromised system.

There exists a wide variety of IDS and IPS implementations such as Snort[1], Suricata[2], Zeek (formerly Bro)[3], Fail2Ban[4] and many more. These are usually classified by both their detection method and the scope of protection (where the IDS or IPS are located). These two classes are outlined below.

2.1.1 Detection method

One can structure the detection method of an IDS/IPS into one of two main categories: signature-based detection and anomaly-based detection. The main distinction between those two categories is that the first one is best suited to detect known attacks, while the latter one is suited for detection unknown attacks. However, none of the methods are strictly constrained to detect either known or unknown attacks. This will be further detailed in the following subsections. In certain cases in the literature a third detection approach is mentioned and that is stateful protocol analysis detection [2]. With this approach the protocols themselves are profiled and any deviations from how the protocol normally intended to be used is alerted. Having said that, we deem this detection method more obscure and is therefore omitted from this overview.

Signature-based detection

An IDS/IPS with a signature-based detection approach tries to match the incoming events with a pre-defined set of signatures. This approach is sometimes in the literature called misuse detection. A signature is a pattern that is written prior to the analysis in order to match on and detect known attacks. Some examples of signatures could be:

A HTTP request that contains the characters "`'OR 1=1 --`". This could indicate a SQL injection attack, where the attacker exploits inadequate string sanitation of an application and gains direct access to the database.

[1] www.snort.org
[2] www.suricata-ids.org
[3] www.zeek.org
[4] www.fail2ban.org

A Windows Security log event with the ID 4719, which means that the audit policy on the system was changed. This could indicate that a malicious actor is trying to cover their tracks by disabling the auditing mechanism.

In general, signature-based detection has a higher accuracy and precision rate than anomaly-based detection approaches. Both the examples of signatures above can quickly detect an attack that consists of what the signature pattern is written to match with, by utilizing efficient search algorithms. One example of such a search algorithm is the Aho-Corasick search algorithm, that is included with the well known open-source network IDS Snort [3, 4]. This algorithm constructs a finite state pattern matching machine from the keyword to be searched. A finite state pattern matching machine will process the characters of the search string and when a exact string match is found, it is reported as a match.

However, the downside of using a signature-based detection approach is that only known attacks can be detected, if a signature have been written to detect the attack. Unknown attacks or so called zero-day attacks are very hard to detect with this approach. In addition to this, only small changes in the payload can render the signature useless. This is because the finite state machine will not report a match even with a one bit difference in the keyword and the payload, and thus make it possible for the attacker to avoid detection. For instance, an attack could simply add a space right before and after the equals sign in the first signature to avoid detection. The SQL injection attack would still work as intended with such a change.

If the signatures above were to match on certain events it does not necessarily mean that a real attack has been detected. Both of the signatures could trigger on benign events as well, for instance with the latter signature a system administrator could have changed the audit policy in order to perform maintenance. Such an alert will be deemed a false-positive.

Anomaly-based detection

An IDS/IPS with a anomaly based detection approach a profile or baseline of normal activity on the system or in the network must be established before monitoring can begin. The period that is used to record normal activity is often referred to as the training period. This training period should include as much normal and benign activity as possible, while avoiding malicious activity. Recording malicious activity in the training period will skew the profile and attacks may not be alerted as an intrusion. A profile may include the following metrics as a baseline:

User A does only log in to a specific system from the IP range 192.168.20.0/24. Any deviations from that will be alerted.

Host 1 only utilize 50% or less of the available CPU processing power. Any deviations from that will be alerted.

Activity that is deemed to be too far from the metrics collected in the training period will be alerted as an intrusion. This detection approach allows for the detection of so called zero-day attacks or attack that are not researched and written signatures for detection. However, by using this approach one assumes that all activity outside the regular baseline is an intrusion, which is an assumption that does not often hold in the real world with complex systems and networks.

Anomaly-based detection systems can further be grouped in to one of the following groups, based on the approach it uses to determine what is anomalous events or benign events. These approaches are either statistics-based, knowledge-based or machine learning-based [5]. An anomaly-based detection mechanism with machine learning methods is the approach that is explored in this thesis.

2.1.2 Scope of protection

When discussing the scope of protection of and IDS/IPS it is often structured in to one of two main categories: network-based or host-based. These categories are the most prevalent in the literature, however in some research application-based and target-based is covered as well. Application-based protection methods monitor a specific application for anomalies, while target-based methods monitor a specific file for unauthorized changes or verifies its integrity. In the following sections we are only going to cover the two most prevalent methods, network-based and host-based.

Network-based IDS/IPS

As the name implies, network-based IDS/IPS monitor network packets and its contents within a network. The IDS/IPS can either be installed in the network in a physical appliance designed to monitor network traffic or install a software version of the network-based IDS/IPS on a pre-existing host residing in the network. A network-based IDS/IPS requires a network card that support promiscuous mode, which essentially means that the card will accept all packets, even the packets that are not addressed to that card. Furthermore, the IDS/IPS can be configured in one of two ways, either with an inline deployment or as a passive deployment.

Inline deployment is a design where all the traffic has to pass trough the IDS/IPS on its route toward the final destination. This approach makes for a more efficient prevention of malicious traffic as the time between detection and response is lower than if the detection solution had to communicate with a third party prevention mechanism to block the traffic. However, placing a network sensor inline can greatly impact the latency of the traffic when the solution has to analyze large amounts of traffic [6, p. 174].

In contrast to an inline deployment, a passive deployment only receives a copy of the network traffic and the original traffic is being transmitted without any delays caused by the passive IDS/IPS solution. This copied traffic could be derived from a span port, which is a port on a network switch that outputs all traffic

passing the switch. Other solutions for sending the traffic to an inline IDS/IPS are to utilize a network tap which is a device that copies all traffic directly from the network media or to utilize a load-balancer that deliver that traffic to a sensor and the destination [2]. Passive deployments do not cause the same latency as inline deployments [6, p. 174].

Host-based IDS/IPS

Host-based IDS/IPS solutions reside on specific hosts and monitor it for malicious or unwanted activity. In order to monitor a fleet of hosts in an environment, one IDS/IPS is required to be installed on each host. The detection is usually based on data from either logs, contents found in the memory or the system calls that the various applications are making via the operating systems API [6, p. 55].

2.1.3 Challenges in intrusion detection

Different detection methods touched upon above all have their flaws and which method to be used in a ICT environment depends on a lot of factors. As already mentioned signature-based systems can produce alerts with few false positives, but are not able to detect novel attacks. With anomaly-based systems the false positive rate is often higher, but such systems are able to detect novel attacks. Furthermore, anomaly-based systems still suffer from low throughput because a high volume of traffic and data to analyze. [7] Advanced threat actors often perform their operations in a slow and steady manner as well, forcing the analysis of data from a even greater timeframe. Therefore research into optimizing the performance of anomaly-based detection system is important.

Additionally, a greater percentage of the traffic than earlier is encrypted which is positive for confidentiality of information, but an issue when analyzing the traffic for unwanted activity. [7] This makes the placement of network sensors even more crucial for detecting malicious activity. Host-based IDS/IPS systems will also become important.

2.2 Machine learning

Machine learning, which can be considered as a branch of artificial intelligence [8], is a research field that have gained a lot of traction in the last couple of years with the increase of available computational power. In the information security sphere it is being utilized to detect anomalous or unwanted events. Machine learning algorithms are usually split into two main categories; unsupervised algorithms and supervised algorithms. Supervised algorithms compile models that can be used to make a decision based on earlier data that has been collected and labeled with a correct outcome. Unsupervised algorithms on the other hand take in an unlabeled dataset and try to find patterns or a hidden structure in the data. In both categories there exists a plethora of algorithms and techniques that are

suitable for various learning problems. In the subsections below we provide some examples of the techniques that can be utilized.

2.2.1 Supervised learning

With supervised learning algorithms, pre-labeled data is used in order to infer a model that can be applied to non-labeled data. [8] The quality of the pre-labeled dataset is important in order to obtain a well generalized model and satisfactory results. Classification is one of the most well known problems that are often solved by using supervised learning algorithms. A classifier assigns objects in a dataset to a finite set of classes. The object, which can be seen as the dependent variable is analyzed in order to determine which class this object or independent variable belongs to. With regards to information security a common example often used to illustrate classification is mail filtering. Based on a set of attributes or features from the mail itself, it is classified either as spam or not spam. Such a problem where there are two classes is referred to as binary classification, while in cases that there are more than two classes it is referred to as multinomial classification. Well known classification algorithms include decision trees, k-nearest neighbours and Bayesian classifiers.

Besides classification, regression is another well known supervised learning technique. Instead of assigning each object to a class, regression is often used when predicting continuous values, sometime known as forecasting [8]. The independent variables can still be either continuous or discrete, but the dependent variable or target is continuous. This continuous variable is determined as a function of the independent variables, also known as the attributes. The function is inferred from the independent variables that is given in advance. Well known regressional methods are support vector machines, regression trees and linear regression. [9, p. 9]

When developing and evaluating machine learning algorithms with a pre-labeled dataset, the dataset is often split up in to two parts. One dataset used for training the algorithm, called the training set, and one dataset for evaluating the algorithm, called the testing set. Using the same data in the training phase as in the evaluation phase would skew the results and provide an inaccurate evaluation of the algorithm. In some cases where there are not a huge amount of labeled data, a technique called k-fold cross validation can be applied in order to utilize the whole dataset in training and testing. The dataset is first split into k subsets and then the machine learning algorithm is evaluated k times. For each time the algorithm is evaluated one new unique subset is utilized as the testing dataset, while the others are used as the training dataset. The final evaluation is then reported as the mean scores from the k evaluation iterations.

2.2.2 Unsupervised learning

Contrary to supervised learning, unsupervised learning takes in data that has not been assigned any labels. Instead of using the knowledge of these labels, unsuper-

vised algorithms try to find structures or relationships within the dataset that are given. Clustering is often the technique that comes to mind when first discussing unsupervised learning, as this method does not require the data to be pre-labeled. Clustering aims to find objects within the data that are as similar as possible, determined by a certain dissimilarity function. Some clustering algorithms determines the number of clusters as a part of the computation, while other algorithms such as k-means use a predetermined number of clusters before arranging the objects in to a number of clusters. When using a predefined number of clusters, this is often chosen based on domain knowledge for the problem at hand. However, there exists methods that can aid in selection of the number clusters. The elbow method is a well known technique for choosing the number of cluster for a given dataset. With this method the explained variance for an increasing number of clusters is plotted against the number of clusters. [10] The explained variance is a ratio that is calculated by taking the sum of squares between the cluster and dividing it by the sum of squares total.[5]

In order to be able to use clustering in a classification tasks, labeling of the clusters is required. This labeling can simply be performed by extracting the cardinality of the cluster and deriving some information from that or the utilization or more complex cluster evaluation methods. [11]. However, such labeling is not covered in this thesis.

The two most popular techniques when it comes to clustering are either hierarchical clustering or partitional clustering. [9, p. 15] Hierarchical clustering can either be performed in a bottom-up manner or in a top-down manner. When using a bottom-up approach each object in the dataset initially belongs to their own cluster with one object. The most similar clusters, based on some dissimilarity function, are then merged together. With the top-down manner, this process is reversed and the clustering begin with one large cluster of all the objects. Then for each iteration the cluster is split up into sub-clusters. [12] The correct number of clusters can then be selected, either by the algorithm itself or by a person with domain knowledge. Advantages of hierarchical clustering is that is can handle data with an arbitrary shape, as well as it support for arbitrary types of attributes. The disadvantage of these techniques is that the time complexity is relatively high, which is especially disadvantageous with large datasets. [12]

Partitional clustering techniques need to have the number of clusters c set in advance, before starting the clustering process. Then, based on some similarity metric the algorithm searches for the optimal solution objects are partitioned in to c clusters. The clustering process initially starts with c clusters that can either be randomly selected or selected by some selection algorithm and then the clusters are recomputed iteratively until the iteration outputs no new clusters that is improved with regards to the similarity metric. [9, p. 14] K-means is a partitional clustering algorithm and is very similar to what is described here. It will be explained more in depth in chapter 3.1. Advantages of partitional clustering

[5] www.davidmlane.com/hyperstat/B160638.html

techniques are the relatively low time complexity and the algorithms that fall into this category are quite efficient in general. When it comes to the disadvantages, as opposed to hierarchical clustering that can handle any data, partitional clustering cannot handle non-convex data. Furthermore, outliers in the data could easily impact the clustering result and the number c of cluster that is chosen will also heavily impact the end result. Partitional clustering techniques will also frequently output a local optimum, as opposed the global optimum that is desired. [12]

2.2.3 Challenges in machine learning

It is very rarely possible to apply some machine learning method on a dataset and retrieve the desired results. The fact that no single machine learning algorithm can be applied to any problem and yield usable results are illustrated with the *no free lunch* theorem. This theorem states that no single algorithm can beat a random guessing of the results for every given data. [13] Therefore, with machine learning we have a plethora of algorithms and parameters to differentiate on, based on the problem at hand.

However, there still exist many challenges that can negatively affect the end result that researchers and developers must be aware of. Overfitting is a common problem with supervised learning algorithms. This phenomenon occurs when the model is not generalized well enough and it will yield a very high accuracy in the training phase. However, when evaluating the model on test data the accuracy is considerably lower than in the training phase. The model is too tightly fit to the training data. k-fold cross validation, as described above, is a technique that can be utilized to overcome overfitting. On the contrary side, we also have underfitting where the trained model is too generalized and does capture the structure of the data in a satisfactory manner. [13]

Furthermore, we have the problem known as *curse of dimensionality*. It is easy to believe that more dimensions in the a dataset will yield better results, as there are more information. This is in many cases quite opposite of the truth and having more dimensions will often produce worse results than using fewer features or dimensions. Obtaining generalized models get exponentially harder with the number of dimensions used. [13] Distance metrics such as the Euclidean distance also suffers from this and the metric becomes less meaningful with the increasing number of dimensions. [14] More dimensions can cause a lot of noise and then the distance metric between each point becomes equally distant from each other. Lastly, whichever solution is chosen to solve a problem and especially with regards to classifying malicious events from a computer security perspective, there will always be some false positives and false negatives. When it comes to classification of such events is is important to assess the trade-off between a high false positive rate vs. a high false negative rate.

2.3 Composition of a cyber attack

Early on when research into the intrusion detection research field began, malware and computer worms were the main threat that defenders was worried about. It was widely believed that the threat actors were individuals, pursuing unauthorized access to computer networks in order to earn respect in technical hacking communities they were a part of. The consequence of attacks performed by such threat actors was mostly downtime, loss of data and the time and money spent to reinstall compromised systems [15].

In the last decade more motivated and skilled threat actors have emerged that pose a much greater threat to organizations than the threat actors mentioned above. Such threat actors are often referred to as *Advanced Persistent Threats*, or APTs for short. The characteristics that separate APTs from more traditional threat actors such as opportunists or script-kiddies is that APTs have clear objectives and targets for their attacks [16]. Often, the targets are organizations with valuable intellectual property or government organisations with classified information. Furthermore, APTs structure their attacks in a highly organized manner in order to maximise the possibility of successfully reaching their objective and it is mostly believed that such threat actors work in organized groups that are well financed. This financing often comes from governments, the military or in some cases private companies [16]. As these groups are well funded, they do not aim for short-term monetary gain. This means that their attack campaigns can be carried out over an extended period of time, while utilizing stealthy tools, techniques and procedures to avoid detection. APTs have the capacity to do a lot of research and development as well and therefore find and leverage zero-day vulnerabilities in their attacks.

Considering the great threat such highly organized attacks pose, it is important that defenders can engage with breaches in way that efficiently can terminate attacks as early as possible. In 2011, Hutchins *et. al.* [17] published a model called the *Intrusion Kill Chain* model, which aims to describe advanced attacks as an integrated process where each phase is dependent on the prior phase in order to succeed. For defenders, the main goal is to break this chain of events to terminate the attack. Early detection is key to reach this goal and intrusion detection systems is an important tool to facilitate this detection.

The seven different phases included in the intrusion kill chain model, is displayed in figure 2.1. Below follows a list of the phases with a corresponding explanation of actions performed in the phase, that is discussed in [17].

Reconnaissance In this first phase a threat actor will attempt to collect as much information as possible regarding the specified target. This information collection could for instance be performed by utilizing human intelligence gathering techniques (HUMINT), open-source intelligence techniques (OSINT)

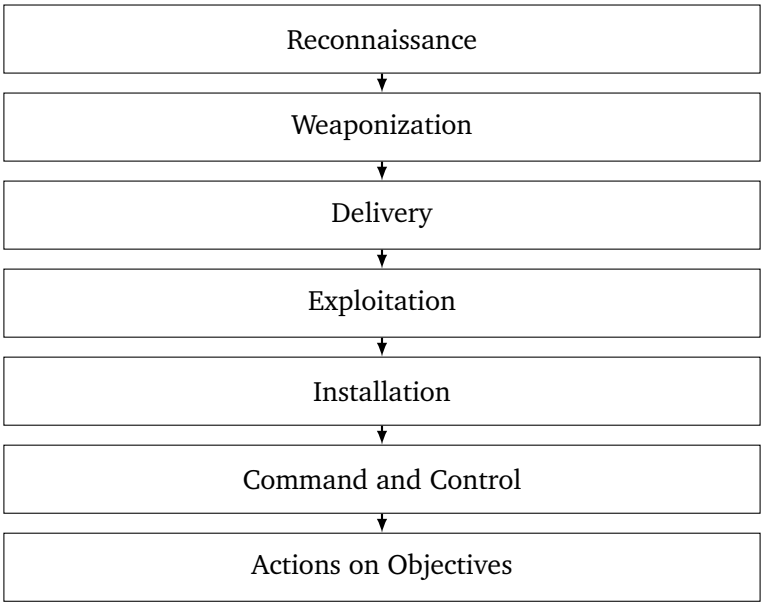


Figure 2.1: The intrusion kill-chain model

or technical reconnaissance tools used to map out the network and fingerprinting of systems used by the target.

Weaponization A threat actor use the information obtained in the reconnaissance phase to craft a specialized package that can exploit one or more vulnerabilities in the victims environment. This package can for instance consist of a remote access trojan (RAT), hidden within a Microsoft Office document or some other file that a victim is tricked in to opening.

Delivery In this phase the threat actor delivers the weaponized package crafted in the last phase to one or more victims. The transmission can either occur directly or indirectly [16]. With direct methods the threat actor often utilize e-mail or similar communication methods to deliver the package. While, with indirect methods the threat actor first compromise a trusted third party and utilize this third party when delivering the package. A third party could for example be a software company that regularly serves the threat actors target with patches for software and the malicious package could be hidden within such an update.

Exploitation After delivery of the package the malicious code inside is executed by some means. The execution could either be performed manually by tricking a user into executing it or a legitimate feature could be utilized in order to execute the code automatically.

Installation With successful exploitation in the last phase threat actors install malicious code in the system and ensure that this code is persistent on the

system, even after reboots. This persistence can be obtained by utilizing what is referred in the literature as Auto-Start Extensibility Points, which is features in a system used to run code at startup without any user interaction [18].

Command and Control After successful exploitation and installation the threat actors need to establish a communication channel in order to issue command to the installed RAT. This communication channel could be as simple as a regular SSH tunnel, however, more covert channels could be used instead. An example of such a covert channel is when threat actors leave pre-configured commands on social media sites and the RAT is then pulling the commands from those sites in order to determine what action should be performed [16].

Actions on Objectives In this final phase the threat actors perform the necessary actions in order to achieve their objectives. As mentioned above, the objectives often consist of exfiltrating intellectual property or classified information. An objective may also include to move laterally in the environment, to locate the desired information or goal. If that is the case, this whole process of reconnaissance to actions on objectives is performed again until the final goal is reached.

2.4 Parallel processing & MapReduce

Big Data is a term that has been used loosely over the past couple of years when discussing the influx of new data sources and the substantial volume of data derived from those sources. The data volume exceeds the capabilities of traditional technologies, both in terms of storage capacity and analysis. [19] As a result of those limitations within traditional technologies new models and frameworks have been developed in order to handle the large volume. These novel models and frameworks include distributed filesystems and parallel programming models such as MapReduce, which is discussed below. Distributed file systems are storage solutions that can be spread across hundreds of nodes. Distributing this storage need can therefore provide more reliable storage for large quantifies of data as well as cost-effective storage, compared to traditional single node storage solutions. [19]

In the sphere of information security, the claim that there is an influx of data holds true as well with large organizations onboard more employees and ICT systems. In order to comply with regulatory compliances and being able to monitor these systems for malicious activity event logging is enabled. Furthermore, the traffic from those systems and employees are captured and analyzed as well. At the beginning of the last decade it was estimated that, depending on the size of the organization, 10 to 100 billion events are generated by ICT systems every day. [20] One can safely assume that this number have risen significantly in the last ten years.

The following two section provide a brief overview of MapReduce and the Apache Flink framework utilized in this project work.

2.4.1 MapReduce

MapReduce is a programming model that has been proven to be efficient in use when dealing with large volumes of data in a parallelized environment spread across multiple nodes. [21] The programming paradigm is heavily utilized at Google in their processing operations of huge quantities of data. As the name implies, a MapReduce program consists of two operations; the *map* operation and then the *reduce* operation. The map operation take a key/value pair as the input and perform some type of custom operation on the value provided as input. The map operation then emits one or more key/value pairs as output. This output is referred to as a intermediate key/value pair as this is then used as input for the reduce operation. In the reduce operation all these pairs provided as input is processed and in most cases aggregated to a smaller set of key/value pairs than was provided as input to the reduce function.

Say for instance that there is a need to identify how many times a set of users has logged in to some system within the span of the last year. All this information exists in the retained log files, however the amount of data is to large in order to use traditional methods for analyzing. With a MapReduce program this task can be split up into smaller subtasks, in a parallel environment. The log files would then be split up into *n* partitions, which is processed by *n* nodes. A *map* function written for the purpose will emit a key/value record for each line of the log file where a user that existed in the set of desired users. These output key/value records from the *map* function would look something like `(user1, 1)`, `(user1, 1)` and `(user2, 1)`. The records will then be used as input for the *reduce* function, that will accumulate records with equal keys, resulting in an output of `(user1, 2)` and `(user2, 1)`.

2.4.2 Apache Flink

Apache Flink is an open source project that stems from research and developement at the Technical University of Berlin and is now managed by the Apache Software Foundation. Flink is a processing framework that can work with both stream and batch data, which in terminology used by Flink is referred to as unbounded and bounded streams. The framework is designed to be employed in a distributed manner over a large number of nodes, in order to process large amounts of data. Furthermore, the architecture of Apache Flink revolves around using in-memory structures, when performing computations which in turn offer better performance than writing and reading to disk.[6] The architecture of Flink differs from other similar frameworks, such as Apache Spark or Apache Hadoop, in that it use a true streaming engine for its execution, as opposed to treating

[6]https://flink.apache.org/flink-architecture.html

Transformation	**Explanation**
`Map`	The `map` transformation takes one element as input, applies the custom transformation and provides one element as output, a so called one-to-one mapping. This is performed for each element in the DataSet.[7]
`Filter`	The `filter` transformation takes on element as input and only output the element if the custom transformation function returns true.[7]
`Distinct`	The `distinct` transformation takes element from a DataSet and removes duplicate elements.[7]
`Aggregate`	The `aggregate` transformation provides built-in function to sum all elements or find the min / max of all the elements in the DataSet.[7]

Table 2.1: Overview of some Apache Flink transformations

streaming data as micro-batches. For both bounded and unbounded streams this streaming approach is used. [22] The programming model of Apache Flink is similar to the MapReduce model described above. However, Apache Flink includes additional transformation as well, besides the Map and Reduce transformation. Some of these transformations are detailed in table 2.1.

Apache Flink offer several APIs that can be used to make applications for a range of different use-cases. The applications, written in either Java or Scala, are compiled into a JAR file and then executed in a parallel environment. This environment consists of one jobmanager (master) and multiple taskmanagers (slaves) that work together in order to execute the application. [22] The APIs offered by Apache Flink is structured in three layers, where the lowest layer provide high expressiveness but it is not very concise. APIs at the top of this layered structure offer concise functions, but with lower expressiveness.[8] This top layer consists of two relational APIs, namely a SQL and Table API that can be used to make queries in the data that flows trough the Flink engine, much like in database systems. In the middle of the layered APIs we find the DataStream and DataSet API, which handle unbounded and bounded data respectively. We utilize the DataSet API in this thesis. The DataSet class in Flink is a special class which can be considered an immutable collection of the data the application is working with.[9] An important distinction to bear in mind in this thesis is the difference between *dataset* (lowercase letters) and *DataSet* (with capital letters). When using the latter we refer to the DataSet class, while using the former it refers to the dataset used to evaluate the algorithms.

[7] https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/batch/dataset_transformations.html
[8] https://flink.apache.org/flink-applications.html
[9] https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/api_concepts.html

When executed the data-flow of the application follows a directed acyclic graph (DAG), that begin with a data source, then transformations are applied the data and then the data is written to a datasink.[10] A datasink is where data is written to at the end of execution, such as to a web socket or a distributed filesystem.[11] Even though the data-flow follows a DAG, special forms of iterations are allowed. Unlike other similar frameworks such as Apache Spark and Hadoop, Flink include some custom operators that handle iterations. The two operators are named `bulkIteration` and `deltaIteration`. With the first operator the whole dataset is used in each iteration, while with the latter iteration operator this dataset is divided in to a working set and a solution set. [23] The working set contains data that is actively being worked with and forwarded to the next iteration, while the solution set contains data that no longer require any transformations.

[10] https://ci.apache.org/projects/flink/flink-docs-stable/concepts/programming-model.html
[11] https://ci.apache.org/projects/flink/flink-docs-stable/dev/connectors/

Chapter 3

Related Work

In this chapter we present the reader with related research that built the foundation for this thesis. First we present the original k-means algorithm and then we present the work of optimizing this algorithm by exploiting the triangle inequality theorem. Furthermore, research on parallel implementation of k-means with triangle inequality will be introduced as well.

3.1 K-means

k-means, is one of the most well known and used algorithms for clustering data. It is used within many different areas as it is not as complex as other algorithms and it is quite fast. The algorithm was published by two researchers independently in 1965 and 1982 by Edward W. Forgy and Stuart Lloyd respectively [24, 25]. In some cases the k-means goes by the name of these two inventors, and is therefore sometimes referred to a as the Lloyd-Forgy algorithm. In this thesis we simply refer to this algorithm as k-means or normal k-means, as we also use a modified version of the algorithm which is explained later.

As mentioned earlier k-means is categorized as a partitional clustering method, as the algorithm use a predefined number of clusters and iterates in a way to either minimize or maximize a numerical criterion. [9, p. 338] The k in k-means indicates the number of pre-defined clusters, while the means illustrates that the centroid for each cluster is the mean of all samples within that cluster. The centroid is the arithmetic mean point of a cluster and at the first iteration these centroids often are selected at random or pre-selected by some algorithm. It is important to remember that the end result of the clustering will vary depending on the initial centroids that was selected. Even though k-means is regarded as a fast algorithm the time complexity grows fast with very large datasets. Its time complexity can be described by $O(n^2)$, where n is the number of points to cluster. [26] This means that the time complexity is proportionate to the number of input points to group in to clusters.

In the steps below we illustrate in detail how k-means iterates in order to output the final cluster results of any given dataset.

1. First, k is pre-determined and given as an input parameter, either based on domain knowledge or based on some analysis performed beforehand. Additionally, the number of total iterations i is also given as an input parameter. In some cases a limit is set to determine when the algorithm has converged as well, meaning how small of a distance the centroids move between iterations.

2. Then initial centroid selection is performed. Initial centroids can either be picked randomly from the set of input points, be given in advance or determined by some other method such as k-means++.

3. Each point is assigned to the nearest centroid and becomes a part of the cluster for the given iteration. The distance between a point and a centroid is defined by a distance metric. Euclidean distance is most commonly used and this is what is shown in equation 3.1 with n dimensions.

$$d = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \ldots + (x_n - y_n)^2} \tag{3.1}$$

4. When all points are assigned to a centroid, new centroids are calculated. This is performed by taking arithmetic mean of the whole cluster. [9, p. 341]

5. A check is then performed to determine that the total amount of iterations has not been reached and that at least one centroid has moved more than the convergence limit. If both of these statements are true, the clustering process continues and the process repeats itself from step 3. If one of these statements is false, the clustering process ends and outputs k clusters.

3.2 K-means with triangle inequality

In any true triangle the length of one side is less than the length of the sum of the two other sides. This is known as the triangle inequality theorem. Phillips [27] published an article in 2002 that propose some modifications to the k-means clustering algorithm in order to reduce the computation time. In this article Phillips exploit the triangle inequality theorem to avoid provable redundant distance calculations. In an article by Elkan from 2003 [28], using the k-means with triangle inequality is once again proposed. In this article two lemmas are provided to show how the triangle inequality theorem is exploited to obtain both upper and lower bounds used to skip redundant distance calculations.

In these two lemmas x represent a point and c_i represents a centroid. $d()$ is a distance function which returns the distance between any two given points. Lemma 1, shown in equation 3.2, allows us to skip a distance calculation between x and c_1 when $d(x,c_1) \leq 0.5d(c_1,c_2)$, as illustrated by Elkan in [28].

$$If \;\; d(c_1,c_2) \geq 2d(x,c_1) \;\; then \;\; d(x,c_1) \geq d(x,c_2) \tag{3.2}$$

With lemma 2, shown in equation 3.3 we can skip some distance calculations for a point to a new centroid, based on knowledge of how much the centroid has moved compared to the last iteration.

$$d(x,c_2) \geq max\{0, d(x,c_1) - d(c_1,c_2)\} \tag{3.3}$$

Experiments performed in the article from Elkan [28] are using datasets with up to 1000 dimensions, while still benefiting from excluding the redundant distance computations and achieving better performance.

3.3 Parallel k-means with triangle inequality

Research performed by Al Ghamdi et. al. [29] in 2017, use this improved k-means clustering algorithm on the distributed computing framework Apache Hadoop. As the framework does not natively support the use of iterations, an iteration driver is used to control the iterative element. Two approaches are suggested in order to adjust k-means with triangle inequality to operate on Hadoop. In the first approach, an extended vector with the points and all necessary information for the triangle inequality to be exploited is included. These vectors are written to the Hadoop Distributed Filesystem (HDFS) at the end of an iteration and are read from the same filesystem again at the start of the iteration. With the second approach, the researchers use a method in which they only write the necessary information to *Bounds Files* in between iterations. With a parallelism of 16 the researchers observe a relative speedup, compared to normal k-means, of up to 6.8 times.

Lastly, what inspired this master thesis project is the research conducted by Chitrakar and Petrovic [30, p. 133]. K-means with triangle inequality, also referred to as k-meansTI, was in this work implemented on Apache Spark with the goal of analyzing digital evidence. They present a framework where the points are extended with the upper bound and lower bounds inside a resilient distributed dataset for Spark. The implementation was measured against a regular implementation of k-means included in Spark, and was evaluated with multiple datasets. These datasets had a number of attributes ranging from 41 up 962. One of the datasets they used was the KDDCup99, with both benign and malicious network

traffic from a military network environment. With this dataset no increase in performance was observed when using k-meansTI and the performance dropped drastically even though many distance calculations was skipped. For the larger dataset KDDCup98-Big, which they produced by doubling the amount of features by copying the dataset, a larger performance increase of 1.5 times was observed with k-meansTI when they utilized a cluster size of 500. None the experiments in the study was performed with a varying degree of parallelism. They concluded with that a performance increase is most likely to be observed when the data consisted of many dimensions and was not sparse in nature.

Chapter 4

Choice of Methods

This chapter describes the methods utilized to answer the research questions stated in 1.3, and justify the selection of these methods. The process followed in order to develop the applications used in the experiments are described first and then the selection of the dataset is justified. Furthermore, we describe how this dataset was pre-processed. In the end, the research approach and design utilized to collect and analyse data is laid out, as well as the environment we performed the experiments in.

4.1 Software development

In order to perform the experiment two applications were required, namely an application that can perform regular k-means clustering and an application that can perform k-means clustering with the triangle inequality. An agile development method was used in order to create these applications. In the book Software Engineering [31, p. 76] five principles define an agile development process. The first principle is *user involvement*, which states that the user of the application should be involved in the development process to provide requirements and evaluate each new iteration of the application. In our case, the user is the researcher which used the software to answer the research questions. Principle number two specifies that *change should be embraced* in the process as requirements often change trough out the development cycle. Third, the development should focus on *incremental delivery*. This means that for each iteration of the application should include a small number of new requirements. Furthermore, the fourth principle focus on *maintaining simplicity* and one should strive to eliminate as much complexity as possible. Lastly, the fifth principle say to focus on *people, not process*. This is an important principle forming an agile process, as it allows the people working on the software to use their own knowledge and experience without being impeded by processes.

For this research project an agile development method was suitable because only one person was involved when developing the software. In addition to this

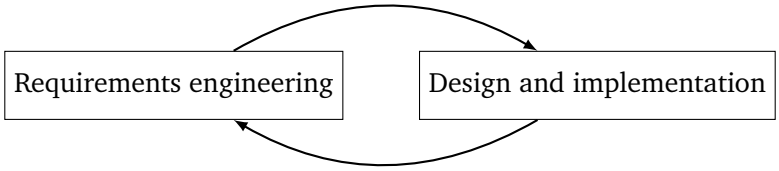


Figure 4.1: The agile development process

the project was relatively small compared to larger software systems, and the developed applications was not tightly coupled to other systems. This agile process we followed is illustrated in figure 4.1. The figure illustrates the cyclic process where requirements are being set and then implemented. After implementation testing was performed in order to verify that the features were implemented and worked as intended. Testing was performed manually by running the application and comparing the output to the expected output. Then this process began again, with further requirements until all requirements for the application was fulfilled.

Apache Flink includes several examples[1] of applications in their code base. One of these examples is a Java application performing k-means clustering. This example application served as a base for the development and was extended heavily with more features required for this research. First the application with normal k-means was developed, then the application with k-meansTI was developed further by forking the normal k-means application and extending it. This was done in order to ensure that the two applications were as similar as possible. The experimental work and setup is described in full in chapter 5.

4.2 Dataset

The dataset that was chosen for this research was the NSL KDD dataset, which is introduced in [32]. This dataset consists of records from the well known KDD Cup 1999[2] dataset. The authors [32] analyzed the original KDD Cup 1999 dataset and pointed out some inherent flaws in it that could affect classifiers that were using the dataset and therefore affecting the results of the classification. First, and most importantly a large percentage of the records was redundant. This duplication of records could cause a bias towards the records that occur more frequently in the training phase of a classifier. Secondly, in the original KDD dataset the test set did not reflect the training set in terms of distribution of records with different levels of difficulty. In the NSL KDD dataset these issues were resolved by removing redundant records and re-generating the training and testing sets. The full NSL KDD dataset consists of 41 different features.

[1] github.com/apache/flink/tree/master/flink-examples
[2] kdd.ics.uci.edu/databases/kddcup99/kddcup99.html

4.2.1 Pre-processing

In order to utilize the dataset for the application written, pre-processing steps were performed on the dataset. These pre-processing steps are outlined in the sections below.

Removing categorical features

Euclidean distance is the distance metric we utilized in the k-means algorithm for this work. This distance metric only support the use of continuous attributes [9, p. 326]. In the NSL KDD dataset there are some values that are discrete. The values that are two-valued (either 0 or 1) can be ordered and treated as a special case of continuous feature, according to Kononenko [9, p. 188]. Those are therefore kept. When it comes to the features that are discrete and consists of more than two values, are removed from the dataset. In this case that means that three features are removed, namely the protocol type, service type and the flag.

Label encoding

The applications that are written can only handle integers as labels for the records. Therefore class labels are translated to integers, where class 0 is normal traffic, and class 1 is abnormal traffic.

4.3 Experiment design

In order to answer the research questions for this project we applied a quantitative research approach. With this approach we aim to find explanations and make predictions from collected data that is inherently numerical of nature. Furthermore, as stated by Leedy and Ormrod [33, p. 98], the intent of quantitative research is to "*identify relationships among two or more variables and then, based on the results, to confirm or modify existing theories or practices*". This means that we are interested in finding a cause-and-effect relationship between some variables. To identify this relationship we used a true experimental design called within-subjects design which allowed us to compare the effectiveness of two different treatments on our selected dataset. In order to measure the effects we control the independent variable, while measuring the dependent variable. An independent variable is a variable that we as researchers can tweak in order to change the outcome of an experiment. The dependent variable is therefore the variable that is affected by any change in the independent variable. [33, p. 59]

In a within-subjects research design, which sometimes is also referred to as repeated measures design, different treatments are administered to the same subjects. The treatments should preferably be administered as close to each other in time as possible. Furthermore, treatments should be applied to the subject in a repeated manner, in a randomized order. An illustration of the experiment design

Subject	**Time →**			
Dataset	$Tx_{k-means}$	$Obs_{k-means}$	$Tx_{k-meansTI}$	$Obs_{k-meansTI}$

Figure 4.2: Within-subjects design

is displayed in figure 4.2, where Tx indicates the treatment applied, while Obs indicates the observation / collection of the dependent variable.

In our case when we say the subject, we mean the dataset that we are performing the calculations on and the treatment is either normal k-means or k-means with triangle inequality applied. We performed two different variations of the experiment in order to measure any change in speedup for the clustering. In the first experiment we varied the number of clusters (k), while using the same number of iterations (i). In the second experiment we varied the number of iterations (i), while using the same number of clusters (k). In both experiments we measured the total runtime in milliseconds that the algorithms used in order to cluster the data. This means that the independent variable in each of our experiments were the treatment, either normal k-means or k-means with triangle inequality. Additionally, the k and i can be seen as independent variables as well that we tweak and observe the results. The dependent variable was the execution time. In both experiments we also recorded the number of total distance calculations each time the algorithms was executed, and this measurement is also an dependent variable. For both the first and the second experiment we executed the the algorithms ten times, and alternated between the treatments. Besides this, all experiments where replicated running varying levels of parallelism (p), meaning the number of nodes that the Apache Flink application was allowed to utilize. p can be seen as a independent variable as well, that we control.

4.3.1 Validity and bias

Validity in research describes how reliable, accurate and meaningful the results of an experiment is. Validity can be divided in to two categories, internal validity and external validity.

In order to be able to draw reliable conclusions from our data and establish a cause-and-effect relationship it is important that the experiment has a high degree of internal validity. [33, p. 59] We primarily do two actions in order to maximise the internal validity, randomizing the order of the experiments and restarting the Apache Flink process between each execution. Randomizing the order of the experiments allows us to rule out any degradation of the system over time. Say for instance that we executed the experiments with an increasing number of k and some of the memory was not released after execution. This could lead us to a wrong conclusion than with a larger k, the execution time is greater, when in fact the application executed slower because of less memory. Furthermore, between

every execution we restarted the Apache Flink process in order to avoid any memory issues and to perform all experiments under as similar conditions as possible.

External validity say something about the extent the research can be generalized and replicated in a different setting. For this research project the external validity relies on the dataset utilized. It can be argued that the NSL KDD dataset mostly consists of traffic not found in modern networks and therefore is not applicable to most organizations traffic pattern today. However, when it comes to traffic patterns of different networks today there are no correct answer, as most networks are unique with various kinds of traffic. It would therefore be hard to find a dataset that could be generalized to any network. Furthermore, the main focus of this research is to measure the efficiency of the clustering and in a similar setting with the same amount of features the results can be generalized.

Bias is another element which can affect the result and conclusions drawn. Both instrumentation bias and researcher bias can come in to play. When it comes to the instrumentation bias we use Apache Flink's built in history server to collect the total execution time. Conclusions are drawn based on the assumption that this metric is accurate.

Moreover, when it comes to researcher bias, two elements can be considered. The researchers expectations of the outcome and programming competency on the platform. A researchers expectations and knowledge of previous results from the literature can influence the researchers objectivity. In order to counteract this bias, it is important that the conclusions is tightly coupled with the statistical observations. When considering the researcher programming competency a situation could emerge where certain optimization techniques are not utilized as the researcher is not aware of the techniques. However, this will not affect the results in a major way when comparing the two applications as measures are taken to make them as identical as possible. This is accomplished by first developing the first application, and then developing the second application by extending on a *fork* (identical copy) of the first one, as described in section 4.1

4.4 Collection of metrics

The metrics we collect for analysis is the execution time of the application, the total amount of distance calculations and the number of bytes transferred trough the iteration function (partial solution function), but only for the k-meansTI application. Very few bytes are transferred by the iteration function in the normal k-means application, as the points and their upper and lower bounds are not transferred to the next iteration.

After an execution of the application, called a *job*, metrics are stored in a JSON-file which can be retrieved via an API at a later stage. Both the execution time and the total amount of bytes sent through the iteration function for each specific job are collected via this API. Custom functionality, such as information about the dis-

Type	Number	RAM	Disk	Virtual CPUs
Master	1	8 GB	40 GB	2
Slave	8	4 GB	30 GB	2

Table 4.1: Overview of virtual machines

tance functions, are as default not included in the JSON-file. To overcome this and collect information about the number of distance calculations performed, accumulators were introduced in the source code. Accumulators are simple counters provided in Apache Flink that have an add operation and a final accumulated result. The add operation of these accumulators was executed each time a distance calculation was performed, making it possible to keep track how many distance calculations the two applications performed.

4.5 Logical experiment environment

Apache Flink was deployed in an OpenStack environment provided by NTNU. The full setup consisted of one master node and seven slave nodes. A full specification of the virtual machines that was deployed is displayed in table 4.1. These specifications were chosen in order to maximise the RAM for each host, while efficiently utilizing the available quota. All the machines used CentOS version 7.6 (x86 64-bit instruction set) as the operating system. The Java version on all machines was the OpenJDK version 1.8.0_242. Both the master node and the slave node utilized Apache Flink version 1.9.1, which was the latest version available when this project began.

We configured the environment with the following settings;
the `jobmanager.heap.size` was set to 6144mb, while the `taskmanager.heap.size` was configured with 3072mb. Slave hosts were built by first configuring one machine with the software mentioned above, and then a snapshot of this machine was taken. All the slave hosts were then replicated by using this snapshot.

Chapter 5

Theoretical Contribution

In the last chapter the overall process and method used to develop the two application were described. With this chapter we define key concepts and provide an explanation of the theoretical contribution of this thesis. Both applications are described and it is explained how they produce the output based on the input given. First we describe the inner workings on the extended regular k-means application, then we describe the inner workings of the k-meansTI application.

5.1 Adjusting k-means for Apache Flink

Apache Flink includes several example applications with the source code. One of these examples is a simple k-means implementation that support clustering of points with two dimensions, and a fixed number of iterations. In order to perform the experiments support for an arbitrary number of dimensions was added. Furthermore, support for stoppage of the algorithm when it had converged was added as well.

5.1.1 Support for n dimensions

We adjusted the application to allow for n dimensions as the input. The number of dimensions is given as an argument before the application is executed. Both the *Point* and *Centroid*, which are custom data types, are read from the rows of two separate files. These two classes extend a *Base* class, which includes common functionality of the two data types such as string representation and calculation of the distance between the point and any other point. These two data types are then stored in one DataSet containing points and one DataSet containing centroids.

5.1.2 Support for convergence criteria

Apache Flink offers two methods of terminating an iteration function, either when a pre-defined number of iterations is reached or if a specific DataSet is returned

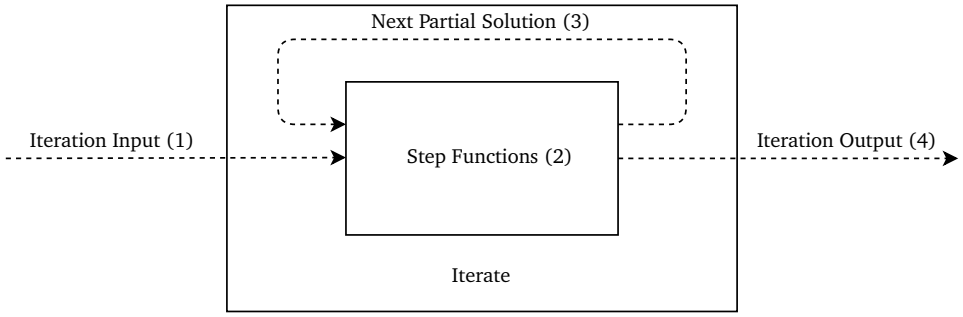


Figure 5.1: Illustration of the iteration process

empty. We utilize the latter method when executing the application with a specific convergence criterion. The iteration function of Apache Flink is illustrated in figure 5.1, and the dashed lines indicates how the data flows. This illustration is taken from the Apache Flink documentation.[1] In the first step of the iteration process the function takes a IterativeDataSet as input. Then one or more transformations are applied to this IterativeDataSet in each step of the iteration. After the transformations are applied, the data is then used as input for the next iteration, illustrated in figure 5.1 as the next partial solution. When the iteration reaches its termination point a final dataset is given as the output.

In this application, when executing a regular k-means algorithm, the next partial solution between each iteration consists of the centroids that was computed. Additionally, we also include an object named *COI (Carry Over Information)* in the next partial solution. In this object we store the euclidean distance between the centroids from the previous and current iteration. A check is performed at the end of each iteration if each of those distances are greater than the convergence criteria. In a situation where all the centroids have moved less than the convergence criteria, an empty DataSet is returned and the iteration process is halted.

5.2 Adjusting k-meansTI for Apache Flink

Using the application with regular k-means as a base, we adjusted it further in order to exploit the triangle inequality in the clustering process. In order to achieve this, the points with their respective upper and lower bounds from an iteration had to be transferred to the next iteration, along with the centroids as well. The iteration functionality in Apache Flink is limited in the sense that it only supports the transfer one dataset between each iteration. To overcome this we introduce an approach where a *tagged tuple* is utilized. Details regarding this approach is described in the sections below. The outline of the main function of the application is illustrated with pseudo-code in algorithm 1. In this code the while loop represent the iteration functionality of Flink, so the while loop does not function as a regular

[1] www.ci.apache.org/projects/flink/flink-docs-release-1.9/dev/batch/iterations.html

loop that one might expect. i represent the value of the iteration parameter and i_j represent counter that increases for each iteration. In Flink this is all handled by the iteration function itself. The full source code for the application with the adjusted k-meansTI can be found in appendix B.

Algorithm 1: Pseudocode of main method in k-meansTI application

```
Read.pointsFromFile();
Read.CentroidsFromFile();

ComputeCOI();

SelectInitialNearestCenter();
ProduceNewCentroids();
ComputeCOI();

AppendPointsToTaggedTuple();
AppendCentroidsToTaggedTuple();
AppendCOIToTaggedTuple();

while i_j < i and !converged do
    SeparatatePointsFromTaggedTuple();
    SeparatateCentroidsFromTaggedTuple();
    SeparatateCOIFromTaggedTuple();

    SelectNearestCenter();
    ProduceNewCentroids();
    ComputeCOI();

    AppendPointsToTaggedTuple();
    AppendCentroidsToTaggedTuple();
    AppendCOIToTaggedTuple();
end

FilterOutFinalPointsFromTaggedTuple()
```

5.2.1 Constructing the tagged tuple

To transfer all required data from one iteration to the next the Tuple7 datatype included with Apache Flink was utilized. A Tuple in Apache Flink is a class that can contain a fixed number of fields where objects can be stored and each tuple length is described in its own class, with the integer at the end of the class name designating the fixed length. This tuple is represented in table 5.1, illustrating the datatype stored in each field.

	Tuple7						
Field	F_0	F_1	F_2	F_3	F_4	F_5	F_6
Datatype	Integer	Integer	Point	Double	Double[k]	Centroid	COI

Table 5.1: Overview of the tagged tuple (tuple7)

The first field, F_0, in the tuple contains an integer indicating what kind of information is stored in the tuple. This is the key of the tuple and is also why we refer to this tuple as a tagged tuple. A value of 0 in the first field indicates that the tuple contains a centroid, 1 indicates a point, while 2 indicates that the tuple contain a custom COI (Carry Over Information) object.

Fields $F_{1,2,3,4}$ stores information regarding which cluster the point is currently assigned to, the point itself and the upper and lower bounds. Field F_5 holds the centroid, while field F_6 hold the COI object. When a certain type of information is stored in the tuple, the fields not in use are set to null. For instance, a key of 2 indicates that the tuple holds a COI object. In this case the fields $F_{1,2,3,4}$ are set to null, and only F_6 holds any information.

All these tuples are stored in one single DataSet in order to be transferred between iterations. At the beginning of an iteration, the points, centroids and the COI object is extracted to their respective DataSets. This extraction is performed by custom filter transformations. A filter transformation applies a custom function to each element in a DataSet and if the function return true the item is stored in a new DataSet. In our case the filter function return true only if the key of the tuple is equal to what the function is configured to extract. Then a custom map function is applied to the DataSets in order to remove all fields consisting of null in the tuple. As far as we know, this method for transferring multiple DataSets between iterations have not been mentioned in the literature.

5.2.2 Constructing the COI object

As mentioned earlier the COI object, short for Carry Over Information, hold information used when computing the upper bound and the lower bounds. This information is the distance between each centroid, minimum distance between any two centroids and the distance between centroids from the last and current iteration. In the DataSet transferred between iterations there is always one, and only one tagged tuple that contain the COI object.

All this information is calculated in a custom `reduceGroup` transformation. This transformation is utilized in order to collect all information on a single node. The transformation is applied on the DataSet with the centroids from the current iteration, while the centroids from the last iteration is included as a broadcast set. A broadcast set in Apache Flink allows us to transfer a DataSet to all active nodes in a Flink cluster. In this case, it is only one single node that receives the broadcast set.

In this custom `reduceGroup` transformation the centroid inter distances are

computed for every k where $k \neq k'$ in a nested loop. For each centroid the distance to its closest centroid is also determined within this nested loop. As the centroids from the last and current iteration is also available in the transformation these distances are computed. These distances are used later in order to determine convergence, but also in the triangle inequality theorem.

5.2.3 Mapping points to centroids

When executing the application, process of mapping a point to its nearest centroid is performed in two different transformations. For the initial mapping, outside of the loop, a transformation is applied for all points. Inside the loop, each iteration, a transformation re-configures the centroid to which a point is assigned to, only if required. These two transformations is described in detail below.

Mapping initial points

With the `SelectInitialNearestCenter()` class we initially map the points to the nearest centroid, while using the triangle inequality to avoid as many distance calculations as possible. This class extends the `RichMapFunction` provided by Apache Flink and allows us to apply the custom map function for all points in a DataSet. The `RichMapFunction` is required, as opposed to the `MapFunction`, it provides the ability to get the centroids and COI object from the runtime context. Full source code of this function can be found in appendix B.1. In order to execute this function only one time at the beginning of the execution, it is placed outside the iteration. The function take a tuple4 as input which consists of the point itself, upper and lower bounds that are not set and which centroid the point is assigned to. Points are assigned to a centroid with ID -1 (which does not exist) when read from file before being assigned to a real centroid. Output of the function is a similar tuple4, but with the correct values for the bounds and nearest centroid set. The initial centroids provided before execution that is utilized by `SelectInitialNearestCenter()` function is broadcasted to all parallel instances. Additionally the COI object is broadcasted in its own broadcast set.

An initial lower bound is set to the distance between the point received by the function and a centroid. These lower bounds are stored in an array referenced by the ID of the centroid, meaning that centroid with and ID of 1, place the lower bound in the first tray (tray 0) of the array. Then for each centroid the distance between the point and the centroid are calculated and compared to current lowest distance to any centroid. A distance calculation is skipped if $0.5d(c_1, c_2) \leq l$, where c_1 is the current closest centroid, c_2 is the centroid to which the distance calculation may be skipped and l is the current minimum distance to any of the centroids looped trough. Inter-centroid distance $d(c_1, c_2)$ is precomputed and fetched from the COI object. At the end the upper bound is set as the minimum distance between the point and a centroid.

Mapping points inside the iteration process

When all points are assigned to their initial centroid and both upper bound and lower bounds are calculated outside the iterations, the `SelectNearestCenter()` class operates inside the iterations in order to re-assign points to the nearest centroids if necessary. This class also extends the `RichMapFunction`. Full source code for this function is also appended in appendix B.1.

For each point received by the function new lower bounds are calculated such that for each centroid the lower bound is set to $max\{0, lb - d(c, c')\}$, where lb is the current lower bound, c is the centroid from the current iteration and c' is the centroid from the last iteration. $d(c, c')$ is precomputed before each iteration and stored in the COI object. This calculation is derived from lemma 2 (see section 3.2), which was introduced by Elkan. [28]

The upper bound is only updated if the current centroid that the point is mapped to has moved. If the centroid has moved, the upper bound is set to $ub + d(c, c')$ where ub is the upper bound. A flag is set to indicate that the upper bound has changed.

When these bounds are updated, the function proceeds to map each point to the nearest center. In this process as many distance calculations as possible are avoided. This is similar to the work presented by Ghamdi et. al. [29] with Apache Hadoop and Chitrakar [30] with Apache Spark.

Before looping trough all centroids and calculating the distances, a check is performed to determine if the whole loop can be skipped. If the upper bound for the point is greater than half of the distance between the currently assigned centroid and the second closest centroid, the point is not re-assigned to any centroid. If however, upper bound is less than that, we begin to loop through all centroids. Three main conditions must hold true in order to calculate a new distance between the point and a centroid. First, we check that $c_j \neq c_i$, where c_j is the centroid currently assigned to a point and c_i is the centroid for a given iteration in the loop. Measuring the distance between the same centroid twice is unnecessary. Secondly, we check that the $ub > lb_i$, where ub is the upper bound and lb_i is the lower bound for centroid i in the loop. The lower bounds used are the ones calculated and mentioned above. Third and lastly we check that $ub > 0.5d(c_j, c_i)$, where c_j is the centroid currently assigned to the point and c_i is the centroid for a given iteration in the loop. If one of these conditions is false, we skip the distance calculations and move on to the next point.

If all are true, we proceed to calculate the distance between point and c_j (currently assigned centroid) if the upper bound has been updated and update the upper bound and lower bound for the respective centroid with this distance. If the upper bound has not been updated, we already know the distance, which is equal to the upper bound. Then in the last if statement we have two conditions, and one of them must hold true in order to calculate the distance between the point and the current centroid for the iteration. First, we check that $d(p, c_j) > lb_i$,

where p is the point, c_j is the currently assigned centroid to this point as already described and lb_i is the lower bound for centroid i in the loop. Second, we check that $d(p, c_j) > 0.5d(c_j, c_i)$. If one of these hold true, we calculate the $d(p, c_i)$, which is the distance between the point and the current centroid for the iteration. The point is only re-assigned if $d(p, c_i) < d(p, c_j)$. Which translates to that the distance between the i-th centroid of the loop is closer than the currently assigned centroid, and then the upper bound is updated to $d(p, c_i)$ and the flag, indicating the upper bound is updated, is flipped.

Chapter 6

Results

In this chapter we provide the results from the experiments performed. The performance results are split up in to two parts, one where we vary the value k (number of clusters) and one where we vary the value i (number of iterations). We present the quantitative data collected with some values of parallelism. A complete set of all the results executing with a parallelism from 1 to 8 can be found in appendix A.

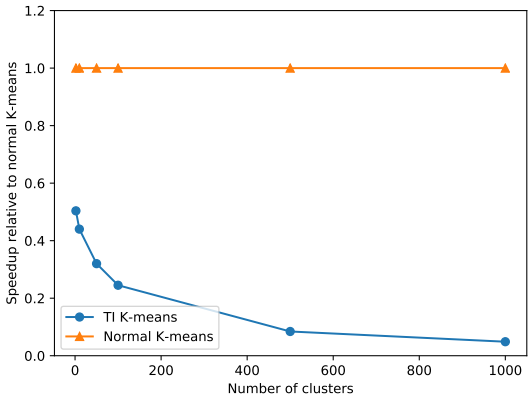
6.1 Performance results

In this section we show the performance of the two adjusted k-means clustering algorithms for Apache Flink, detailed in chapter 5. As in the work of Chitrakar [30] and stated in the introduction, we compare the two versions in order to investigate any changes in performance. The data for comparison are collected in two executions, by increasing the number of clusters k and by increasing the number of iterations i. These two executions are run multiple times, as described in chapter 4. Additionally, we document the change in the number of total distance calculations. The relative speedup is calculated by dividing the mean execution time of k-means by the mean execution time of k-meansTI, as shown in equation 6.1. Therefore the speed of normal k-means is considered as the baseline. In the equation, t represents the execution time. In this chapter we only include the results from executing with a parallelism of 1, 4 and 8.

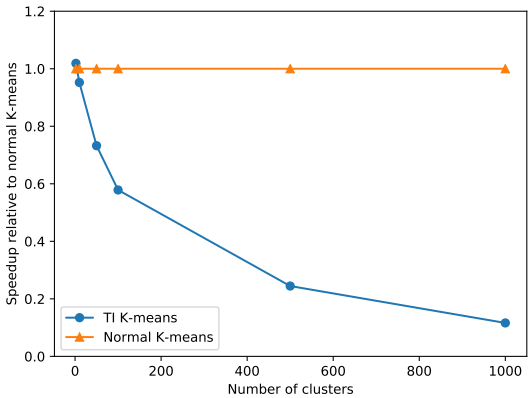
$$Relative\ speedup = \frac{t_{Kmeans}}{t_{KmeansTI}} \tag{6.1}$$

6.1.1 Performance results when varying k

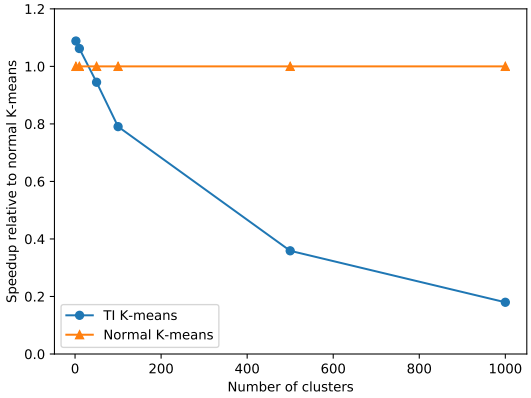
When executing with a varying k the iteration is configured as $i = 15$ and convergence stoppage is not considered. i was set to this value as testing showed that the algorithms converged at around that value. Having the algorithm stop at convergence would yield very inconsistent results and make it hard to compare



(a) Using a parallelism of 1



(b) Using a parallelism of 4



(c) Using a parallelism of 8

Figure 6.1: Speedup when executing with an increasing k

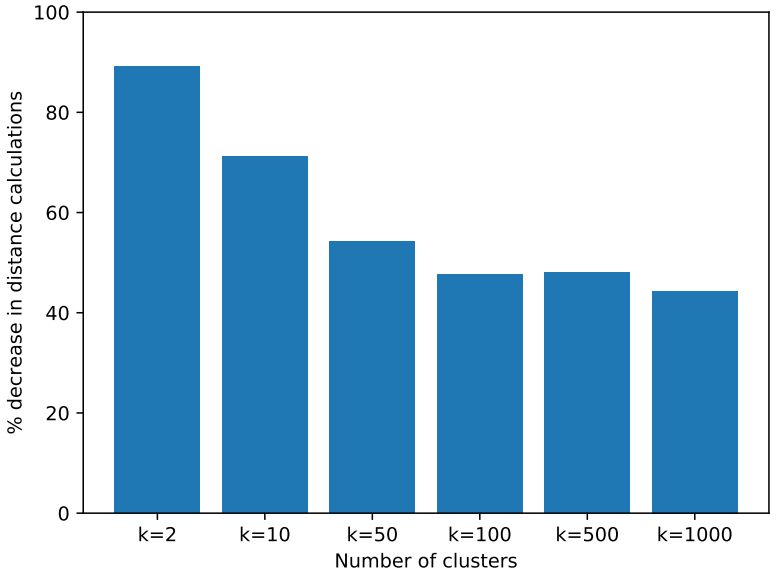


Figure 6.2: Change in distance calculations with an increasing k

the execution times as it would converge at different iterations with a varying k. We utilized the same initial centroids at each k for both normal k-means and k-meansTI.

In figure 6.1a we see that when using a parallelism of 1, the performance of k-meansTI is far slower than normal k-means for every k. At $k = 2$ the performance of k-meansTI has decreased 47.7% relative to the performance of k-means. As k increases the performance degradation becomes even more prominent. At $k = 1000$ the performance has decreased with 95.2%.

In figure 6.1b, when the parallelism is increased to 4 we see that the performance is slightly better at $k = 2$ with an increase in performance of 1.8%. However, when increasing k the performance degrades rapidly. At $k = 1000$ the performance has decreased with 88.4%, which is slightly better than when using a parallelism of 1.

When executing the algorithms with a parallelism of 8, as shown in figure 6.1c we see that for both $k = 2$ and $k = 10$ k-meansTI outperforms normal k-means. At $k = 2$ there is an 8.8% increase in performance, while at $k = 10$ there is a 6.2% increase in performance. We still see a significant drop off in performance after a further increase of k and at $k = 1000$ the performance decrease is 82%.

In figure 6.2, we see the decrease in distance calculations for k-meansTI rel-

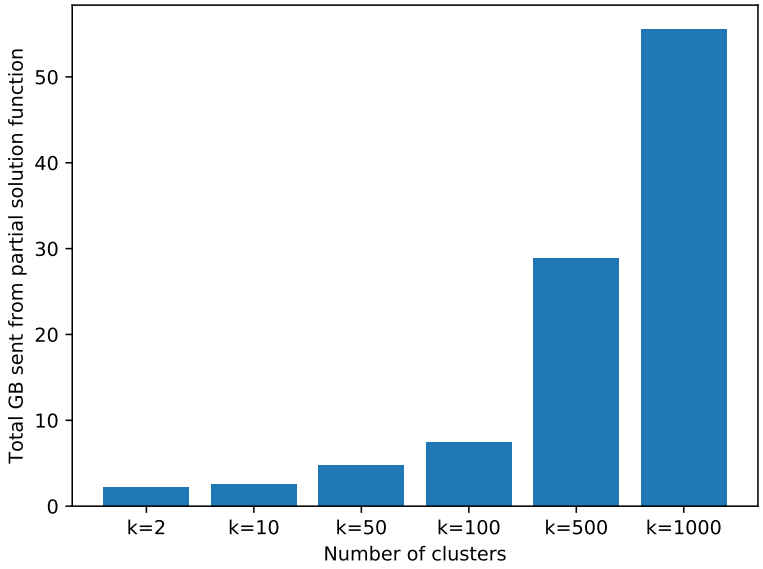


Figure 6.3: GB transferred between iterations, when increasing k

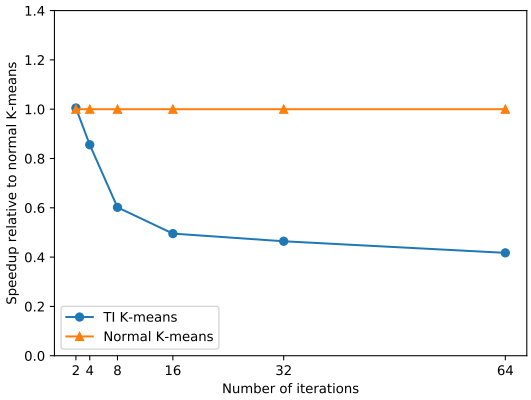
ative to normal k-means. At $k = 2$ about 90% of the total distance calculations are skipped compared to normal k-means. As k increase the amount of skipped distance calculations decrease and at $k = 1000$ only 44% of the distance calculations are skipped. The percentage of skipped distance calculations are calculated with the equation shown in 6.2, with d representing the total number of distance calculations.

$$\%\, skipped\ d\ =\ \frac{d_{Kmeans} - d_{KmeansTI}}{d_{Kmeans}} * 100 \tag{6.2}$$

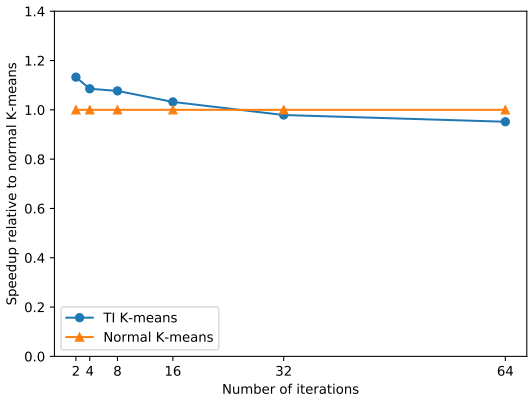
Figure 6.3 shows the increasing amount of data that is being sent trough the partial solution function, when executing k-meansTI. This is the total amount of bytes that is being transferred between each iteration. When $k = 1000$ we see that the total amount being transferred is 55.6 GB. The total amount of data being transferred is growing linearly, with respect to k.

6.1.2 Performance results when varying i

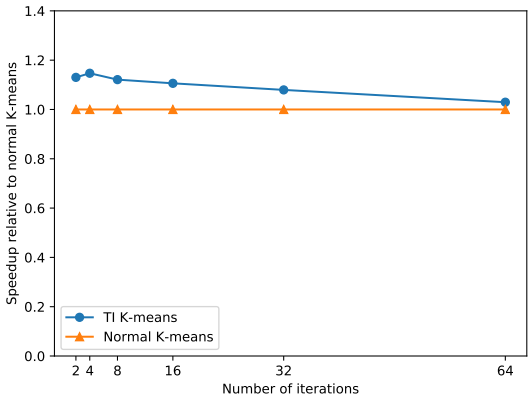
In the second batch of executions, we modified the number of i iterations and kept the number of clusters at $k = 2$. This value was chosen, as in a intrusion detection and prevention setting two classes are mostly used. One class for malicious events and one for benign.



(a) Using a parallelism of 1



(b) Using a parallelism of 4



(c) Using a parallelism of 8

Figure 6.4: Speedup when executing with an increasing i

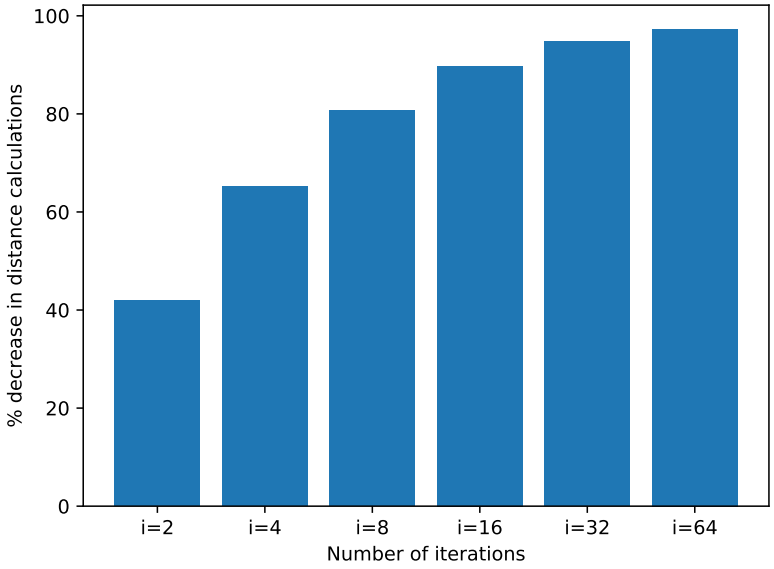


Figure 6.5: Change in distance calculations with an increasing i

In figure 6.4a we see the performance with an increasing number of iterations, using a parallelism of 1. When k-meansTI iterates for two iterations, it performs about the same as normal k-means. When increasing the number of iterations, the performance quickly drops by around 50% to 60%.

Increasing the parallelism to 4 we see that k-meansTI outperforms normal k-means when the number of iterations are less than about 20, as shown in figure 6.4a. With $i = 2$, k-meansTI performs 13.2% better than normal k-means. When k-meansTI iterates over 20 times, its performance is slightly worse than normal k-means.

When executing the applications with a parallelism of 8, k-meansTI outperforms normal k-means for every i that is included in these experiments. This is shown in figure 6.4c. When $i = 4$ k-meansTI performs 14.6% better than regular k-means and with an increasing number of iteration the performance gradually declines towards the benchmark performance that is k-means.

In figure 6.5 the decrease in distance calculations is plotted in a bar graph. We see that k-meansTI omits an increasingly larger number of distance calculations with the increase of i. With $i = 2$, 42% of the distance calculations are skipped, while at $i = 64$, 97.3% of the distance calculations are skipped when comparing to the number of distance calculation performed by k-means.

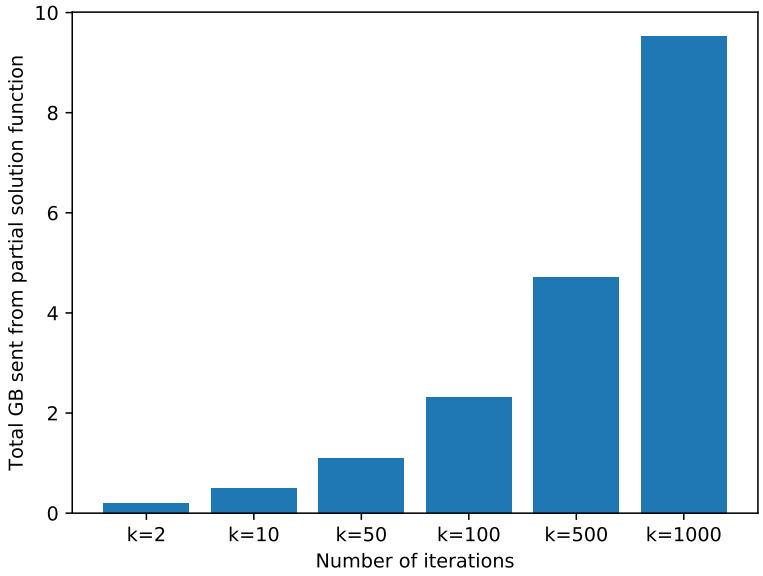


Figure 6.6: GB transferred between iterations, when increasing i

The total amount of bytes being transferred between iterations with different values of i when executing k-meansTI, is displayed in figure 6.6. We see from the figure that with $i = 64$ a total of 9.5 GB of data is transferred between iterations. Here the amount of data being transferred grows linearly as well, with with respect to i.

Chapter 7

Discussion

In this chapter we first provide a concise summary of the theoretical contribution presented in chapter 5 and the results in chapter 6. We then interpret what these results indicate and discuss how the results relate to the research questions posed in the beginning of this thesis. Lastly the limitations of this works are discussed, as well as the implications.

We have adjusted the normal k-means algorithm in such a way that with Apache Flink we were able to transfer data between iterations in order to utilize the triangle inequality theorem. This transfer of data was accomplished by a so called tagged tuple, where multiple DataSets was agglomerated in to a single DataSet and sent to the next iteration. Both the normal k-means algorithm and the adjusted k-means algorithm with triangle inequality was evaluated using a dataset consisting of network traffic, both benign and malicious. The execution time, as well as the number of distance calculations for both algorithms was recorded and evaluated. We saw an increase in execution time when the dataset was clustered in a low number of clusters, together with using a high number of parallel instances. Besides this, we saw that for executions with a larger number of iterations there are some performance benefit as well, if there are enough parallel instances.

7.1 Interpretations of the results

In this section we address the research questions individually and based on the results we provide an answer for the questions posed. At the end we re-visit the hypothesis stated in the introduction.

"*How could k-means clustering with triangle inequality be adjusted to operate with Apache Flink?*"

With the work presented in this thesis we show that it is in fact possible to adjust the k-means clustering algorithm with triangle inequality to operate on

Apache Flink by employing the DataSet API. Apache Flink offer native iterations, however with the drawback that only one DataSet can be transferred to the next iteration and made it harder to transfer all necessary information between iterations. This obstacle was solved by utilizing the tagged tuple, as outlined in chapter 5.2.1. The DataSet of tagged tuples was used to collect all necessary information in one DataSet, transfer it to the next iteration and then split it up to perform transformations on the points, centroids and the COI object individually. When introducing this tagged tuple a suspicion was that the overhead of collecting multiple DataSets into one and then unpacking it in the next iteration would be large, and have a huge impact on the overall performance. The results did not indicate that these operations did contribute to the the main performance degradation. This will be discussed further with research questions number three.

"*How can we apply k-means clustering with triangle inequality on Apache Flink in intrusion detection applications?*"

For the evaluation of k-means with triangle inequality we utilized a well known dataset used to evaluate intrusion detection applications. The NSL-KDD dataset consists of features collected from network connection, both malicious and benign. In an intrusion detection setting, the k-means algorithm can be configured to cluster the data in two clusters, one cluster for benign connections and one cluster for malicious or suspicious connections. Both network traffic or logs collected from systems in a large environment can be collected and then regularly analyzed by this clustering. In such a use-case the cluster with the largest amount of events would consist of mostly benign events.

"*How much can k-means clustering with triangle inequality be optimized compared to regular k-means?*"

The results indicate that the overhead of k-meansTI when having a large k is very high and even when executing the application with multiple parallel instances to distribute the overhead, no performance increase is detected. A rather large decrease in performance is detected instead. Only when executing the k-meansTI application with a low number of clusters and a high number of parallel instances, the overhead becomes manageable and we see a small performance increase. By only clustering with a $k = 2$, we see at most a 8.8% increase in performance. Furthermore, we see that with a low value of k the number of distance calculations are reduced drastically by up to 90% when $k = 2$, but this large reduction is just enough to see a performance increase with a large number of parallel instances that can distribute the overhead between multiple nodes. With an increasing k we see that the reduction in distance calculations drops to about 50% for when k is very large. This makes sense as there are more centroids that are placed within the range of the upper and lower bounds, forcing the algorithm to calculate the distances.

In the second experiment where the applications were executed with an increasing value of i, meaning that we tested with different number of iterations, a performance increase was detected as well for some configurations. When only iterating two times, and with a parallelism of 1 there where basically no changes in performance. With a higher number of iterations, the performance dropped. However, with a parallelism of 8 there was a performance increase up to 64 iterations. At most with an $i = 4$ there was an increase in performance of 14.6%. With the increase of i the number of distance calculations that are skipped also grows. This is because that the algorithm fully converges after about 14-15 iterations, and the centroids are then not moving as much as in the first iterations.

Only when using these parameters, k-means with triangle inequality is more optimal to use than normal k-means.

"*Under what circumstances could the implementation of k-means with triangle inequality thrive?*"

As the results show, even with a large number of skipped distance calculations, any large performance increase is absent. This attest that the distance calculations performed are not very computationally expensive. Instead the overhead of transferring the extra information from iteration to iteration becomes unmanageable for the infrastructure and the performance drops. In a circumstance where the distance calculation was in fact very computationally expensive, a larger increase in performance would likely have been detected with the use of k-means with triangle inequality. The distance calculations may become more expensive with an increase in total dimensions. K-means with triangle inequality will thrive with a low k and an i close to the convergence point.

Contrary to the hypothesis stated in chapter 1.3 we did not see a performance increase of greater than 50% when evaluating normal k-means against k-means with triangle inequality. However, we did show that is was possible to adjust the k-means algorithm with triangle inequality to operate on the Apache Flink framework.

7.2 Limitations and drawbacks

The results from this study only stems from the evaluation of one dataset with a set of features from network connections. Other datasets with more or different features may cause a greater change in performance. Using a dataset more aimed towards host intrusion events that may include more features, could be interesting to see. However, as discussed in chapter 2.2.3 the *curse of dimensionality* may become an issue and interfere with the usefulness of the clustering.

One inherent flaw with the solution chosen when the algorithm was adjusted to operate on Apache Flink is that the full points are being sent between iterations. These points are not altered in the clustering process, and therefore serve as "dead

weight" when transporting information from one iteration to the next. A better approach would have been to only transfer the upper and lower bounds relating to the points, while using the same points for each iteration.

Furthermore, the way the centroids are calculated is not ideal. When calculating the centroids all points are reduced and added to a single large point based on the centroid ID each point is assigned to. This large point is then divided by the number of points that was added. In cases where no points are assigned to a specific centroid, this centroid would become empty and disappear in the next iteration. This method of calculating centroids stems from the regular k-means example used a base. When adjusting the algorithm to operate on Flink with triangle inequality, the centroid ID referenced the place in the array where the lower bounds were stored. This means that with a missing centroid the k-meansTI application would crash, and that limited the evaluation of multiple intrusion detection datasets.

Lastly, a batch approach for analysing data is not ideal in a intrusion detection setting where the intrusions should be detected in real time or as close to it as possible.

7.3 Implications

Further research of methods to reduce the overhead is required before a this approach with k-means with triangle inequality can be utilized to cluster a large amount of security events in a shorter amount of time, than with normal k-means. However, with the results from the experiments performed we see that executing k-means with triangle inequality with a high degree of parallelism, overhead can be dealt with to a certain degree.

Also one should focus more on the real-time detection, instead of the bounded stream approach that was taken in this work. As the engine of Apache Flink treat bounded and unbounded streams similarly, the process of migrating the methods used here with the DataSet API, should be possible to migrate to the DataStream API that Flink offer to handle unbounded streams and continuous detection.

Chapter 8

Conclusion

This research aimed to investigate any increase or decrease in performance when executing k-means with triangle inequality compared to regular k-means, and how these algorithms can be applied in a intrusion detection setting. First the two applications was developed by employing the DataSet API provided by Apache Flink. One application utilizing the regular k-means algorithm and one application utilizing the k-means algorithm with triangle inequality. With respect to the first research question we provided a method to adjust k-means with triangle inequality for Apache Flink. A tagged tuple with all necessary information needed to exploit the triangle inequality was used to transfer information between iterations. This necessary information included the points, an upper bound and lower bounds. The inter centroid distances and the centroids themselves, was transferred by a special object we named *Carry-Over-Information*.

In the experiments we employed a within-subjects research design in order to collect quantitative data for analysis and answer research questions two and three. These questions revolved around applying the algorithms in an intrusion detection setting and how much we could improve the performance. Evaluation was performed with a dataset consisting of benign and malicious network connections. The applications was executed repeatedly every other time, with a predefined set of parameters. The order of which we executed the applications with the different parameters was randomized. An average of the execution time for each parameter set was recorded. Additionally, between each execution we restarted the Flink process on each of the slave nodes to assure that the memory was cleared between each run. These steps were taken in order to ensure the validity of the qualitative data.

Based on an analysis of the quantitative data collected we see that a performance increase is present for some configurations, but not greater than fifty percent as hypothesized. When it comes to research question number four, relating to under which circumstances the adjustment of the improved algorithm would thrive, we observed the following. For configurations often utilized in a intrusion detection application (low number of clusters, and a number of iterations close to the convergence point) the performance increase is at its best, which is promising.

This shows that with further work to reduce the overhead of dealing with transferring information that is static in the execution, k-means with triangle inequality can be used to increase the performance in an intrusion detection setting.

8.1 Further work

We suggest that future studies and research should work towards only transferring the information that possibly can be altered in an iteration, not the information that is kept static. The points themselves are not altered during execution and will only cause a large overhead when being transferred between iterations. Furthermore, it may be possible to avoid the splitting of the tagged tuple and instead just omit records in the transformations that does not handle a specific record.

In an intrusion detection setting where dealing with unbounded streams of data, an online version of k-meansTI using the DataStream API of Apache Flink is more preferable. Methods introduced in this work should be used to adjust the k-means algorithm with triangle inequality for an online version. It should be feasible to utilize the same approach when transferring information between iterations, as the engine Apache Flink use is the same for bounded and unbounded streams. Such a version would fit better for a intrusion detection application.

As already mentioned k-meansTI should be evaluated with multiple datasets, in order to further evaluate use-cases where it will perform well. Especially a dataset consisting used to evaluate host intrusion detection systems. Before this is performed for larger datasets however, the applications need to be adjusted further to handle empty centroids without having them disappear in the following iteration.

Additionally, a study on how various similarity measures affect the performance of clustering algorithms should be performed as well. In this work we only utilized the euclidean distance metric, however with a more computationally expensive distance function k-means with triangle inequality would benefit even more from a large percentage of skipped distance calculations.

Bibliography

[1] T. Grance, J. Hash, M. Stevens, K. O'Neal, and N. Bartol, "Guide to information technology security services," in, Special Publication 800-35. NIST (National Institute of Standards and Technology), 2003, ch. 5.

[2] K. Scarfone and P. Mell, "Guide to intrusion detection and prevention systems (idps)," in, Special Publication 800-94. NIST (National Institute of Standards and Technology), 2007, ch. 2.

[3] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *Commun. ACM*, vol. 18, no. 6, pp. 333–340, Jun. 1975, ISSN: 0001-0782. DOI: 10.1145/360825.360855. [Online]. Available: https://doi.org/10.1145/360825.360855.

[4] M. Norton, *Optimizing pattern matching for intrusion detection*, 2004.

[5] A. Khraisat, I. Gondal, P. Vamplew, and J. Kamruzzaman, "Survey of intrusion detection systems: Techniques, datasets and challenges," *Cybersecurity Springer Open*, vol. 2, no. 20, 2019. DOI: https://doi.org/10.1186/s42400-019-0038-7.

[6] A. A. Ghorbani, W. Lu, and M. Tavallaee, *Network Intrusion Detection and Prevention: Concepts and Techniques*. Fredericton, Canada: Springer, 2009, p. 174. DOI: https://doi.org/10.1007/978-0-387-88771-5.

[7] P. García-Teodoro, J. Díaz-Verdejo, G. Maciá-Fernández, and E. Vázquez, "Anomaly-based network intrusion detection: Techniques, systems and challenges," *Computers & Security*, vol. 28, no. 1, pp. 18–28, 2009, ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2008.08.003. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167404808000692.

[8] M. R. M. Talabis, R. McPherson, I. Miyamoto, J. L. Martin, and D. Kaye, "Chapter 1 - analytics defined," in *Information Security Analytics*, M. R. M. Talabis, R. McPherson, I. Miyamoto, J. L. Martin, and D. Kaye, Eds., Boston: Syngress, 2015, pp. 1–12, ISBN: 978-0-12-800207-0. DOI: https://doi.org/10.1016/B978-0-12-800207-0.00001-0. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780128002070000010.

[9] I. Kononenko and M. Kukar, *Machine Learning and Data Mining - Introduction to Principles and Algorithms*, 1st. Woodhead Publishing, 2007, ISBN: 978-1-904275-21-3.

[10] P. Bholowalia and A. Kumar, "Article: Ebk-means: A clustering technique based on elbow method and k-means in wsn," *International Journal of Computer Applications*, vol. 105, no. 9, pp. 17–24, Nov. 2014.

[11] S. Petrovic, G. Alvarez, A. Orfila, and J. Carbo, "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)*, vol. 6, 2006, 129b–129b.

[12] D. Xu and Y. Tian, "A comprehensive survey of clustering algorithms," *Ann. Data. Sci.*, vol. 2, no. 2, pp. 165–193, Jun. 2015. DOI: 10.1007/s40745-015-0040-1. [Online]. Available: https://doi.org/10.1007/s40745-015-0040-1.

[13] P. Domingos, "A few useful things to know about machine learning," *Commun. ACM*, vol. 55, no. 10, pp. 78–87, Oct. 2012, ISSN: 0001-0782. DOI: 10.1145/2347736.2347755. [Online]. Available: https://doi.org/10.1145/2347736.2347755.

[14] C. C. Aggarwal, A. Hinneburg, and D. A. Keim, "On the surprising behavior of distance metrics in high dimensional space," in *Database Theory — ICDT 2001*, J. Van den Bussche and V. Vianu, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 420–434, ISBN: 978-3-540-44503-6.

[15] C. Tankard, "Advanced persistent threats and how to monitor and deter them," *Network Security*, vol. 2011, no. 8, pp. 16–19, 2011, ISSN: 1353-4858. DOI: https://doi.org/10.1016/S1353-4858(11)70086-1. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1353485811700861.

[16] P. Chen, L. Desmet, and C. Huygens, *A study on advanced persistent threats*, ser. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). 2014, vol. 8735 LNCS, pp. 63–72.

[17] E. M. Hutchins, M. J. Cloppert, and R. M. Amin, "Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains," *Leading Issues in Information Warfare & Security Research*, vol. 1, no. 1, p. 80, 2011.

[18] D. Uroz and R. J. Rodríguez, "Characteristics and detectability of windows auto-start extensibility points in memory forensics," *Digital Investigation*, vol. 28, S95–S104, 2019, ISSN: 1742-2876. DOI: https://doi.org/10.1016/j.diin.2019.01.026. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1742287619300362.

[19] A. Oussous, F.-Z. Benjelloun, A. [Lahcen], and S. Belfkih, "Big data technologies: A survey," *Journal of King Saud University - Computer and Information Sciences*, vol. 30, no. 4, pp. 431–448, 2018, ISSN: 1319-1578. DOI: https://doi.org/10.1016/j.jksuci.2017.06.001. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1319157817300034.

[20] A. A. Cárdenas, P. K. Manadhata, and S. P. Rajan, "Big data analytics for security," *IEEE Security Privacy*, vol. 11, no. 6, pp. 74–76, 2013.

[21] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008, ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. [Online]. Available: https://doi.org/10.1145/1327452.1327492.

[22] G. Mon, M. Makkie, X. Li, T. Liu, and S. Quinn, "Implementing dictionary learning in apache flink, or: How i learned to relax and love iterations," in *2016 IEEE International Conference on Big Data (Big Data)*, 2016, pp. 2363–2367.

[23] N. Spangenberg, M. Roth, and B. Franczyk, "Evaluating new approaches of big data analytics frameworks," in *Business Information Systems*, W. Abramowicz, Ed., Cham: Springer International Publishing, 2015, pp. 28–37, ISBN: 978-3-319-19027-3.

[24] E. W. Forgy, "Cluster analysis of multivariate data: Efficiency versus interpretability of classifications," *Biometrics*, vol. 21, no. 3, pp. 761–777, Sep. 1965, ISSN: 0006341X, 15410420. [Online]. Available: http://www.jstor.org/stable/2528559.

[25] S. Lloyd, "Least squares quantization in pcm," *IEEE Transactions on Information Theory*, vol. 28, no. 2, pp. 129–137, Mar. 1982, ISSN: 1557-9654. DOI: 10.1109/TIT.1982.1056489.

[26] M. K. Pakhira, "A linear time-complexity k-means algorithm using cluster shifting," in *2014 International Conference on Computational Intelligence and Communication Networks*, 2014, pp. 1047–1051.

[27] S. J. Phillips, "Acceleration of k-means and related clustering algorithms," in *Algorithm Engineering and Experiments*, D. M. Mount and C. Stein, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 166–177, ISBN: 978-3-540-45643-8.

[28] C. Elkan, "Using the triangle inequality to accelerate k-means," *Proceedings of the Twentieth International Conference on Machine Learning (ICML-2003)*, 2003.

[29] S. A. Ghamdi and G. D. Fatta, "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)*, Nov. 2017, pp. 985–992. DOI: 10.1109/DASC-PICom-DataCom-CyberSciTec.2017.163.

[30] A. S. Chitrakar, "Constrained approximate search and data reduction techniques in cybersecurity and digital forensics," PhD thesis, Norwegian University of Science and Technology, NTNU i Gjøvik, Teknologiveien 22, 2815 Gjøvik, Norway, Jun. 2019.

[31] I. Sommerville, *Software Engineering*, 10th. Pearson, 2016, ISBN: 1-292-09613-6.

[32] M. Tavallaee, E. Bagheri, W. Lu, and A. A. Ghorbani, "A detailed analysis of the kdd cup 99 data set," in *2009 IEEE Symposium on Computational Intelligence for Security and Defense Applications*, 2009, pp. 1–6.

[33] P. D. Leedy and J. E. Ormrod, *Practical Research - Planning and Design*, 11th. Pearson, 2015, ISBN: 978-1-29-209587-5.

Appendix A

Results from Performance Evaluation

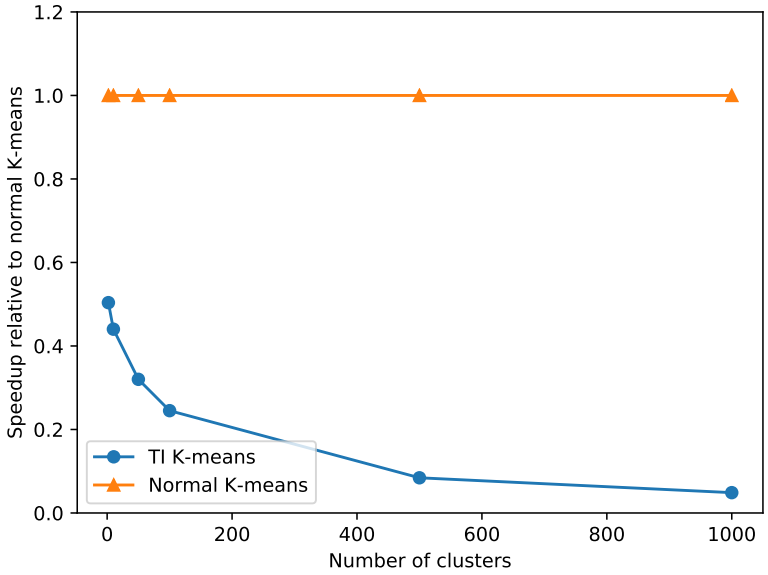


Figure A.1: Speedup with an increasing k, using a parallelism of 1

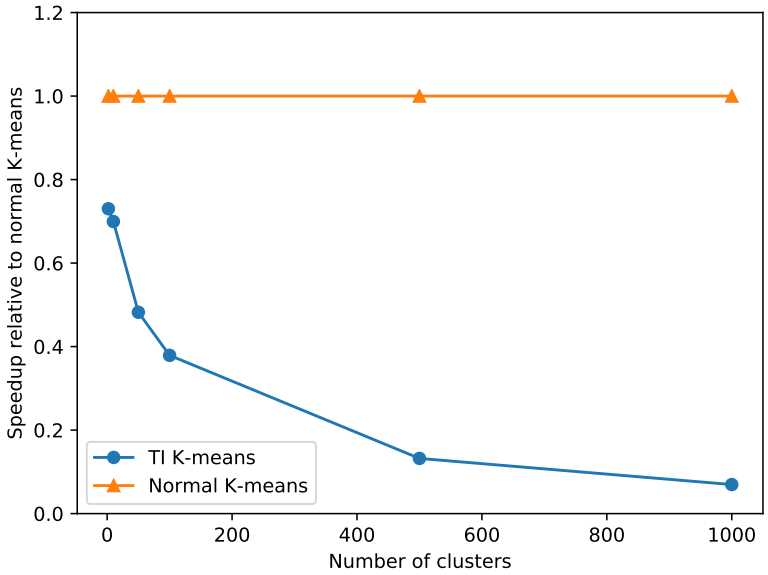


Figure A.2: Speedup with an increasing k, using a parallelism of 2

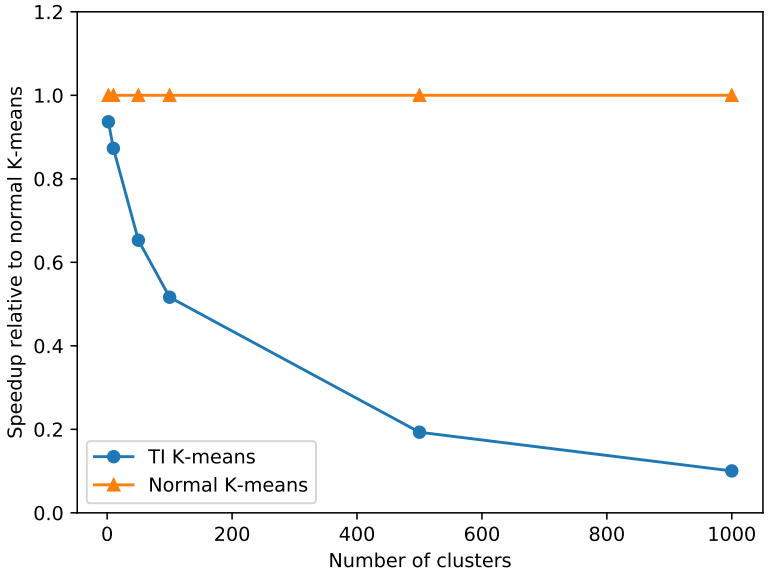


Figure A.3: Speedup with an increasing k, using a parallelism of 3

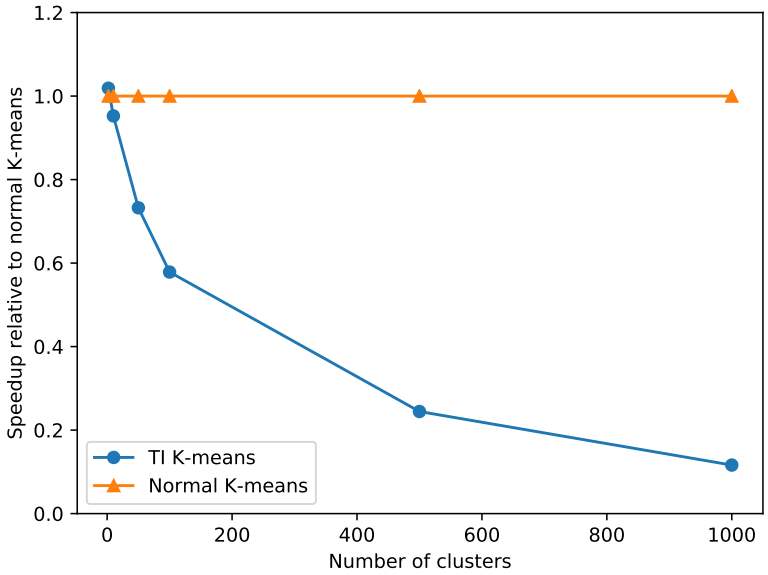


Figure A.4: Speedup with an increasing k, using a parallelism of 4

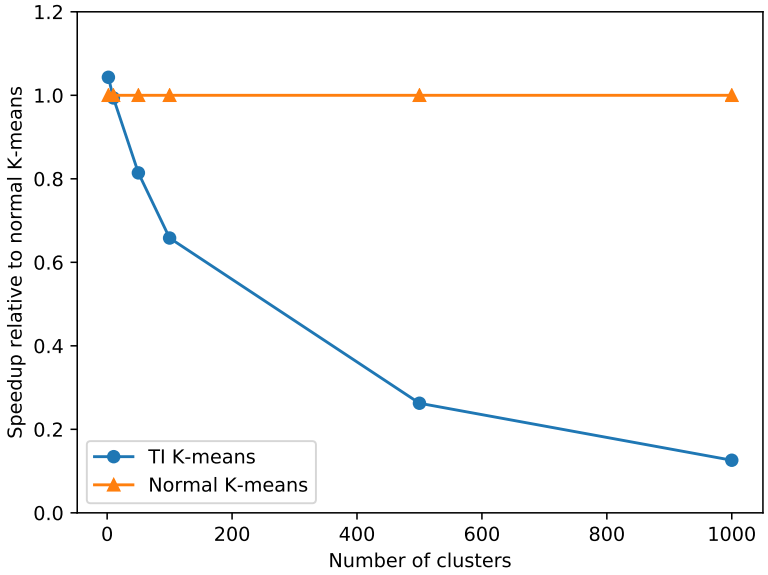


Figure A.5: Speedup with an increasing k, using a parallelism of 5

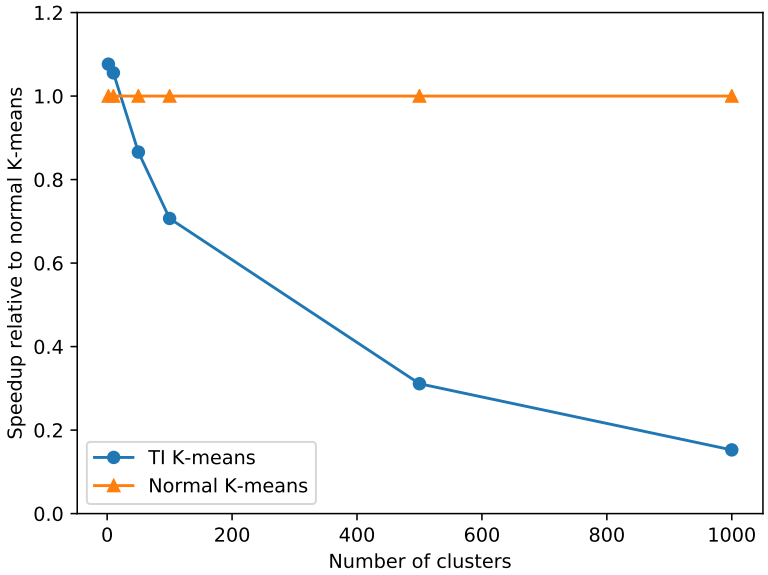


Figure A.6: Speedup with an increasing k, using a parallelism of 6

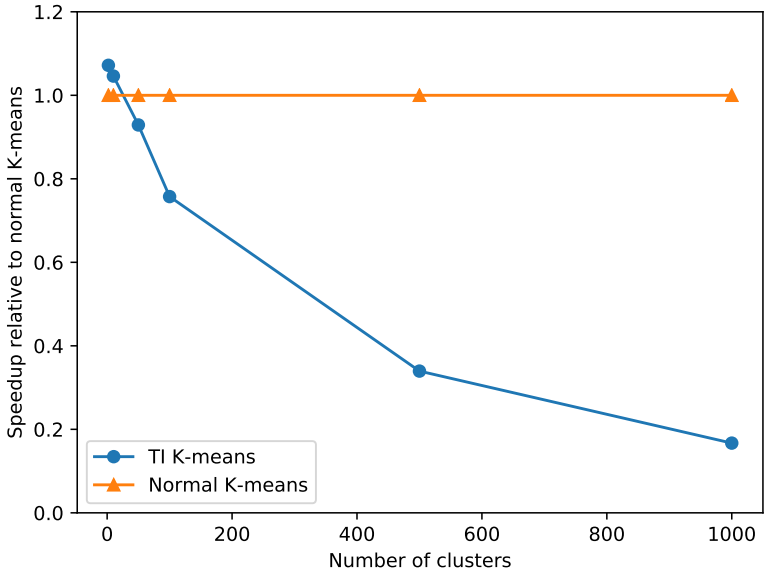


Figure A.7: Speedup with an increasing k, using a parallelism of 7

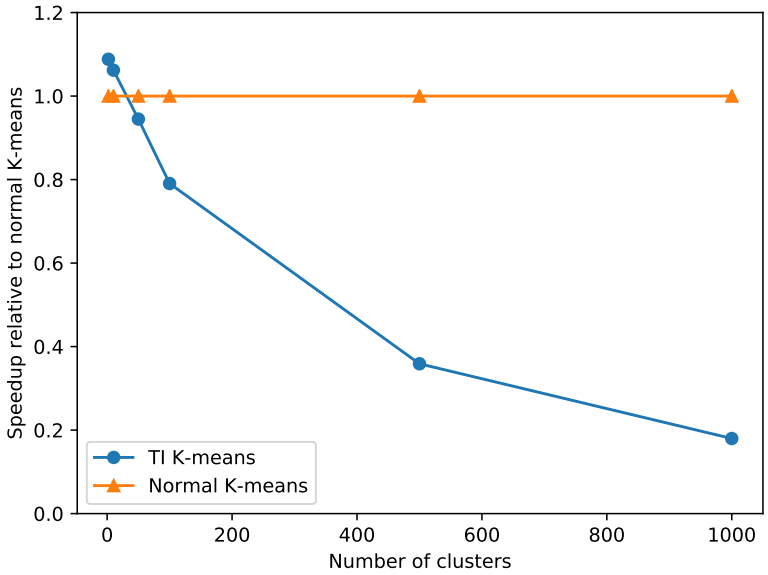


Figure A.8: Speedup with an increasing k, using a parallelism of 8

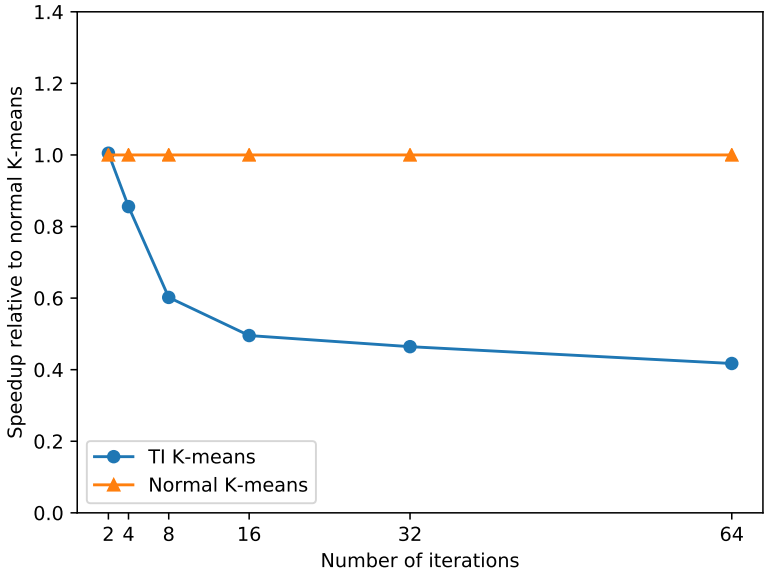


Figure A.9: Speedup with an increasing i, using a parallelism of 1

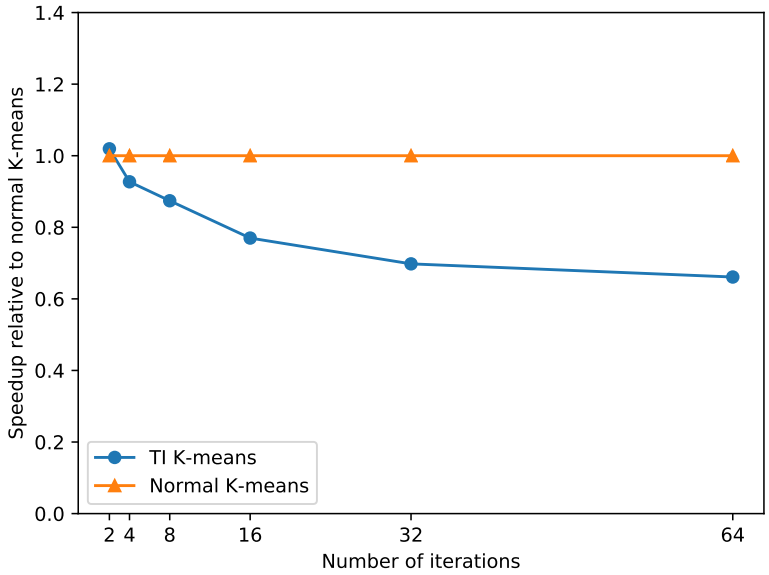


Figure A.10: Speedup with an increasing i, using a parallelism of 2

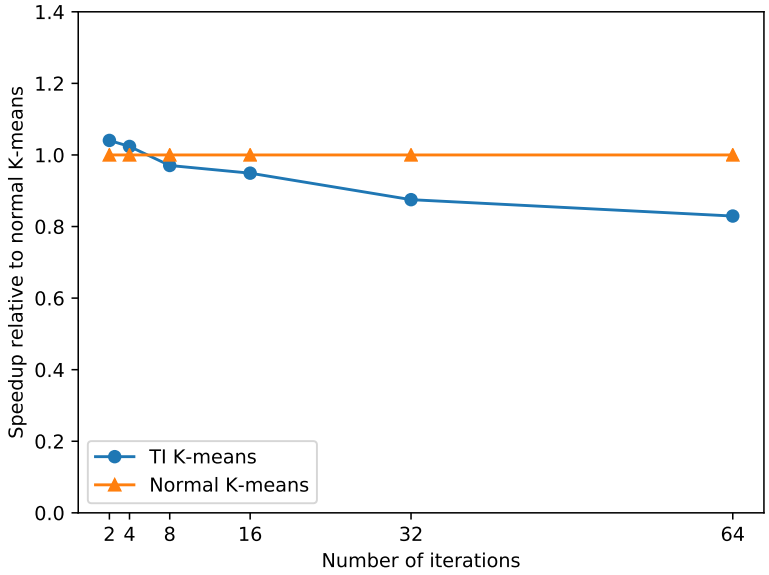


Figure A.11: Speedup with an increasing i, using a parallelism of 3

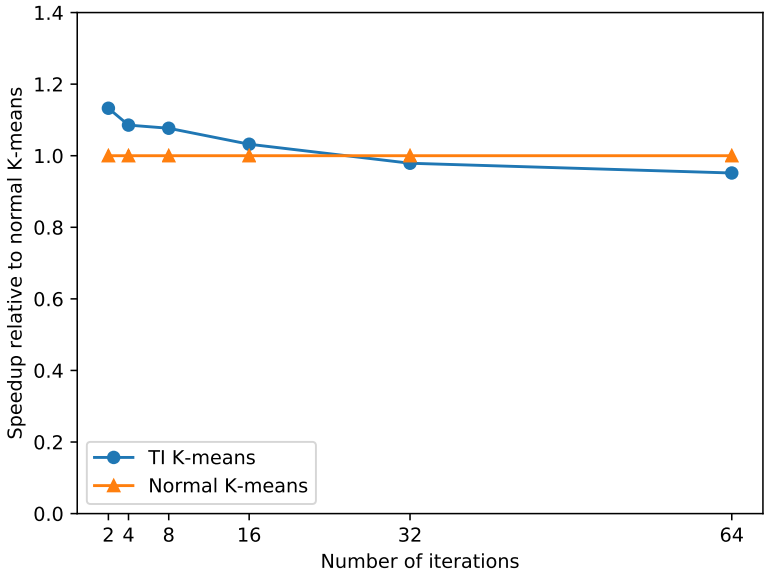


Figure A.12: Speedup with an increasing i, using a parallelism of 4

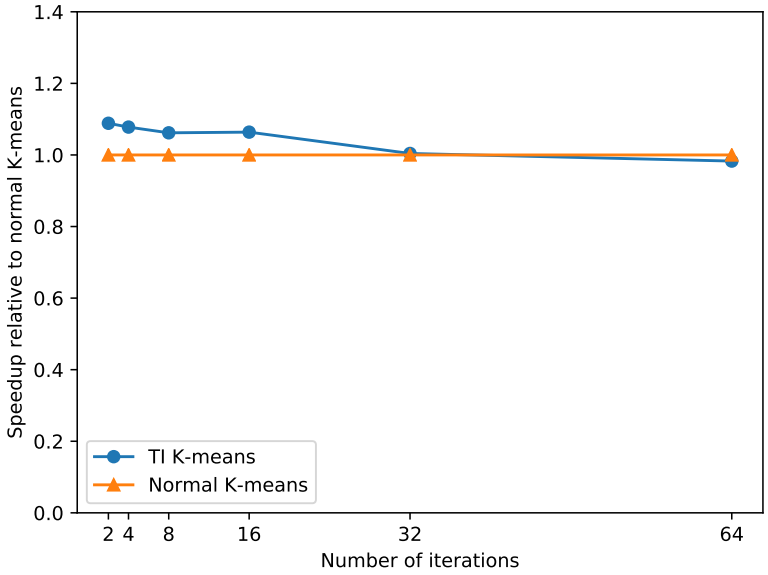


Figure A.13: Speedup with an increasing i, using a parallelism of 5

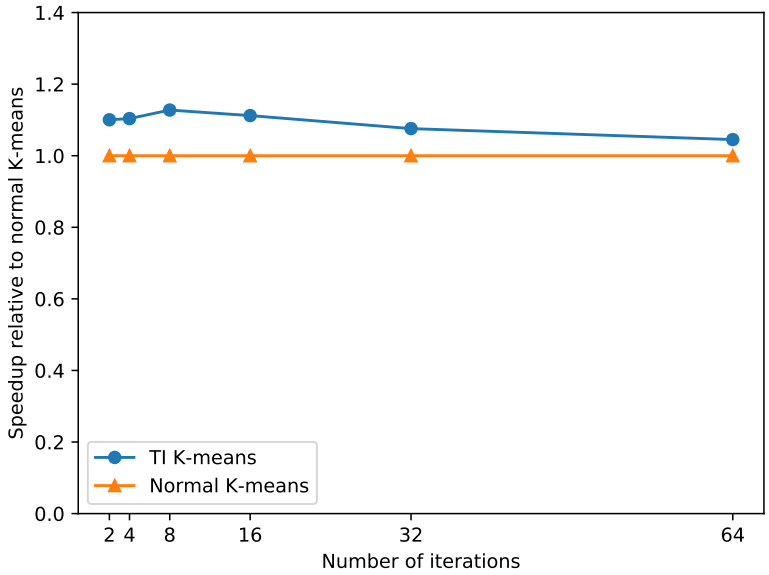


Figure A.14: Speedup with an increasing i, using a parallelism of 6

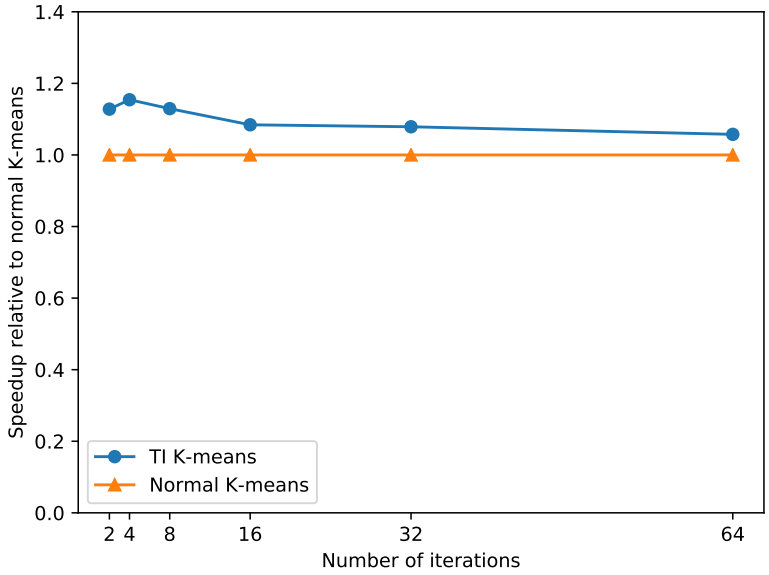


Figure A.15: Speedup with an increasing i, using a parallelism of 7

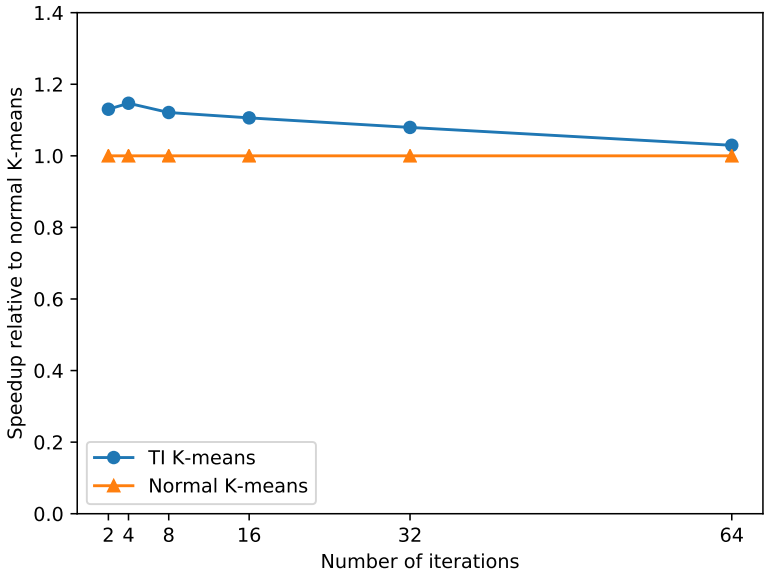


Figure A.16: Speedup with an increasing i, using a parallelism of 8

Appendix B

Source Code

B.1 KMeansTI.java

Code listing B.1: KMeansTI.java

```
1  package com.ringdalen.kmeansti;
2
3  import com.ringdalen.kmeansti.datatype.DataTypes.Centroid;
4  import com.ringdalen.kmeansti.datatype.DataTypes.Point;
5  import com.ringdalen.kmeansti.datatype.DataTypes.COI;
6
7  import com.ringdalen.kmeansti.util.Read;
8
9  import org.apache.flink.api.common.JobExecutionResult;
10 import org.apache.flink.api.common.accumulators.IntCounter;
11 import org.apache.flink.api.common.functions.*;
12 import org.apache.flink.api.java.DataSet;
13 import org.apache.flink.api.java.ExecutionEnvironment;
14 import org.apache.flink.api.java.functions.FunctionAnnotation.ForwardedFields;
15 import org.apache.flink.api.java.operators.IterativeDataSet;
16 import org.apache.flink.api.java.tuple.*;
17 import org.apache.flink.api.java.utils.ParameterTool;
18 import org.apache.flink.configuration.Configuration;
19 import org.apache.flink.core.fs.FileSystem;
20 import org.apache.flink.util.Collector;
21
22 import java.util.*;
23
24 /**
25  * This code is an extended version of the K-Means clustering algorithm provided as
        ↪  an example with Aapche Flink.
26  *
27  * Usage: KMeansTI
28  *           --points <path>
29  *           --centroids <path>
30  *           --output <path>
31  *           --iterations <n iterations>
32  *           --d <n dimensions>
33  */
34
35 @SuppressWarnings("serial")
36 public class KMeansTI {
```

```
37
38      public static void main(String[] args) throws Exception {
39
40          // Fetching input parameters
41          final ParameterTool params = ParameterTool.fromArgs(args);
42
43          // Set up execution environment. getExecutionEnvironment will work both in
                ↪ a local IDE as well as in a
44          // cluster infrastructure.
45          ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
46
47          // make parameters available in the web interface
48          env.getConfig().setGlobalJobParameters(params);
49
50          // Read both points and centroids form files
51          DataSet<Tuple4<Integer, Point, Double, Double[]>> points = Read.
                ↪ PointsFromFile(params, env);
52          DataSet<Centroid> centroids = Read.CentroidsFromFile(params, env);
53
54          // Fetching max number of iterations loop is executed with
55          int iterations = params.getInt("iterations", 10);
56          double convergence = params.getDouble("convergence", 0.0001);
57
58
59          ////// Initial mapping of points and computation of new centroids //////
60
61          // Computing the COI information
62          DataSet<Tuple7<Integer, Integer, Point, Double, Double[], Centroid, COI>>
                ↪ firstCoiTuple = centroids
63                  .reduceGroup(new computeCOI())
64                  .withBroadcastSet(centroids, "oldCentroids");
65
66          DataSet<COI> coi = firstCoiTuple.map(new ExtractCOI());
67
68          // Select the initial cluster the point is assigned to
69          DataSet<Tuple4<Integer, Point, Double, Double[]>> initialClusteredPoints =
                ↪ points
70                  .map(new SelectInitialNearestCenter())
71                  // Broadcast data that is needed in the initial clustering to each
                        ↪ node
72                  .withBroadcastSet(centroids, "centroids")
73                  .withBroadcastSet(coi, "coi");
74
75          // Producing new centroids based on the initial clustered points
76          DataSet<Tuple7<Integer, Integer, Point, Double, Double[], Centroid, COI>>
77                  initialCentroidsTuple = initialClusteredPoints
78                  // Count and sum point coordinates for each centroid
79                  .map(new CountAppender()).groupBy(0).reduce(new CentroidAccumulator
                        ↪ ())
80                  // Compute new centroids from point counts and coordinate sums
81                  .map(new CentroidAverager());
82
83          // Extracting centroids for use in calculation of COI
84          DataSet<Centroid> initialCentroids = initialCentroidsTuple.filter(new
                ↪ CentroidFilter()).map(new ExtractCentroids());
85
86          // Computing new COI information after new centroids have been produced
87          DataSet<Tuple7<Integer, Integer, Point, Double, Double[], Centroid, COI>>
                ↪ coiTuple = initialCentroids
88                  .reduceGroup(new computeCOI())
```

```
89                  .withBroadcastSet(centroids, "oldCentroids");
90
91          ////// Initial mapping of points and computation of new centroids //////
92
93          // Expand the points into a Tuple7 in order to union it with the other
                ↪ DataSets
94          DataSet<Tuple7<Integer, Integer, Point, Double, Double[], Centroid, COI>>
95                  initialPointsTuple = initialClusteredPoints.map(new
                        ↪ ExpandPointsTuple());
96
97          // Combine the points and centroids DataSets to one DataSet
98          DataSet<Tuple7<Integer, Integer, Point, Double, Double[], Centroid, COI>>
99                  unionData = initialPointsTuple.union(initialCentroidsTuple.union(
                        ↪ coiTuple));
100
101
102         // *******************************************************
103         //  Loop begins here
104         // *******************************************************
105
106         // Use unionData to iterate on for n iterations, as specified in the input
                ↪ arguments. This is the beginning
107         // of the loop
108         IterativeDataSet<Tuple7<Integer, Integer, Point, Double, Double[], Centroid
                ↪ , COI>>
109             loop = unionData.iterate(iterations);
110
111         // Separate points in a DataSet in order to use it later in the iteration
112         DataSet<Tuple4<Integer, Point, Double, Double[]>>
113                 pointsFromLastIteration = loop.filter(new PointFilter()).project(1,
                        ↪  2, 3, 4);
114
115         // Separate centroids in a DataSet in order to use it later in the
                ↪ iteration
116         DataSet<Centroid>
117                 centroidsFromLastIteration = loop.filter(new CentroidFilter()).map(
                        ↪ new ExtractCentroids());
118
119         // Separate COI in a DataSet in order to use it later in the iteration
120         DataSet<COI>
121                 coiFromLastIteration = loop.filter(new COIFilter()).map(new
                        ↪ ExtractCOI());
122
123         // Asssigning each point to the nearest centroid
124         DataSet<Tuple4<Integer, Point, Double, Double[]>> partialClusteredPoints =
                ↪ pointsFromLastIteration
125             // Compute closest centroid for each point
126             .map(new SelectNearestCenter())
127             .withBroadcastSet(centroidsFromLastIteration, "centroids")
128             .withBroadcastSet(coiFromLastIteration, "coi");
129
130         // Producing new centroids based on the clustered points
131         DataSet<Tuple7<Integer, Integer, Point, Double, Double[], Centroid, COI>>
                ↪ centroidsToNextIteration = partialClusteredPoints
132             // Count and sum point coordinates for each centroid
133             .map(new CountAppender())
134             .groupBy(0).reduce(new CentroidAccumulator())
135             // Compute new centroids from point counts and coordinate sums
136             .map(new CentroidAverager());
137
```

```
138         // Expand the tuples with points from a Tuple4 to a Tuple7 in order to
                ↪ union it with other datasets
139         DataSet<Tuple7<Integer, Integer, Point, Double, Double[], Centroid, COI>>
140                 pointsToNextIteration = partialClusteredPoints.map(new
                        ↪ ExpandPointsTuple());
141
142         // Separate out centroids in order to be used in calculation for COI
143         DataSet<Centroid> singleNewCentroids = centroidsToNextIteration.filter(new
                ↪ CentroidFilter())
144                 .map(new ExtractCentroids());
145
146         // Computing the new COI information
147         DataSet<Tuple7<Integer, Integer, Point, Double, Double[], Centroid, COI>>
148                 coiToNextIteration = singleNewCentroids.reduceGroup(new computeCOI
                        ↪ ())
149                 // Broadcast data that is needed in the initial clustering to each
                        ↪ node
150                 .withBroadcastSet(centroidsFromLastIteration, "oldCentroids");
151
152         // Check if the algorithm has converged. If no centroids has moved more
                ↪ than 0.001, an empty DataSet will be
153         // returned and that will halt the iteration.
154
155         DataSet<Tuple7<Integer, Integer, Point, Double, Double[], Centroid, COI>>
                ↪ converged = coiToNextIteration.filter(new checkConvergenceFilter(
                ↪ convergence));
156
157         // Combine points, centroids and coi DataSets to one DataSet
158         DataSet<Tuple7<Integer, Integer, Point, Double, Double[], Centroid, COI>>
159                 toNextIteration = pointsToNextIteration.union(
                        ↪ centroidsToNextIteration.union(coiToNextIteration));
160
161         // Ending the loop and feeding back DataSet
162         DataSet<Tuple7<Integer, Integer, Point, Double, Double[], Centroid, COI>>
                ↪ finalOutput = loop.closeWith(toNextIteration, converged);
163
164         // ********************************************************
165         //  Loop ends here. Feed new centroids back into next iteration
166         // ********************************************************
167
168
169         // Only preserve the information (ID of cluster point is assigned to and
                ↪ the point itself) that will be
170         // printed to file or the console
171         DataSet<Tuple2<Integer, Point>> clusteredPoints = finalOutput
172                 .filter(new PointFilter())
173                 .project(1, 2);
174
175         // Print the results, either to a file of the console
176         if (params.has("output")) {
177             clusteredPoints.writeAsCsv(params.get("output"), "\n", "␣", FileSystem.
                        ↪ WriteMode.OVERWRITE);
178
179             // Calling execute will trigger the execution of the file sink (file
                        ↪ sinks are lazy)
180             JobExecutionResult executionResult = env.execute("KMeansTI");
181
182         } else {
183             System.out.println("Printing␣result␣to␣stdout.␣Use␣--output␣to␣specify␣
                        ↪ output␣path.");
```

```
184
185             clusteredPoints.print();
186         }
187     }
188
189     // ************************************************************************
190     //     USER FUNCTIONS
191     // ************************************************************************
192
193     /**
194      * This class implements the FilterFunction in order to filter out Tuple7's
            ↪ with Centroid
195      */
196     public static class CentroidFilter implements FilterFunction<Tuple7<Integer,
            ↪ Integer, Point, Double, Double[], Centroid, COI>> {
197
198         /**
199          * Filter out Tuple7 that does not have the key 0, whcih means that the
                ↪ Tuple does not contain a
200          * Centroid object.
201          *
202          * @param unionData Tuple7 with all unionData
203          * @return boolean True if f0-field (key) is equal to 0, else return False
204          */
205         @Override
206         public boolean filter(Tuple7<Integer, Integer, Point, Double, Double[],
                ↪ Centroid, COI> unionData) {
207             return (unionData.f0 == 0);
208         }
209     }
210
211     /**
212      * This class implements the FilterFunction in order to filter out Tuple7's
            ↪ with Point
213      */
214     public static class PointFilter implements FilterFunction<Tuple7<Integer,
            ↪ Integer, Point, Double, Double[], Centroid, COI>> {
215
216         /**
217          * Filter out Tuple7 that does not have the key 0, whcih means that the
                ↪ Tuple does not contain a
218          * Point object.
219          *
220          * @param unionData Tuple7 with all unionData
221          * @return boolean True if f0-field (key) is equal to 1, else return False
222          */
223         @Override
224         public boolean filter(Tuple7<Integer, Integer, Point, Double, Double[],
                ↪ Centroid, COI> unionData) {
225             return (unionData.f0 == 1);
226         }
227     }
228
229     /**
230      * This class implements the FilterFunction in order to filter out Tuple7's
            ↪ with COI
231      */
232     public static class COIFilter implements FilterFunction<Tuple7<Integer, Integer
            ↪ , Point, Double, Double[], Centroid, COI>> {
233
```

```
234         /**
235          * Filter out Tuple7 that does not have the key 2, which means the Tuple
                 ↪ does not contain a COI object.
236          *
237          * @param unionData Tuple7 with all unionData.
238          * @return boolean True if f0-field (key) is equal to 2, else return False.
239          */
240         @Override
241         public boolean filter(Tuple7<Integer, Integer, Point, Double, Double[],
                 ↪ Centroid, COI> unionData) {
242             return (unionData.f0 == 2);
243         }
244     }
245
246     /**
247      * This class implements the FilterFunction in order to filter out a COI object
             ↪  based on convergence. A returned
248      * COI object means that the algorithm has not converged.
249      */
250     public static class checkConvergenceFilter extends RichFilterFunction<Tuple7<
             ↪ Integer, Integer, Point, Double, Double[], Centroid, COI>> {
251
252         // Accumulator used to track the total number of iterations
253         private IntCounter numIterations = new IntCounter();
254         double convergence_criteria;
255
256         public checkConvergenceFilter(double converge){
257             this.convergence_criteria = converge;
258         }
259
260         /**
261          * Fetched the accumulator from the runtime context
262          * @param parameters The runtime parameters
263          */
264         @Override
265         public void open(Configuration parameters) {
266             // Registering the accumulator object and defining the name of the
                     ↪ accumulator
267             getRuntimeContext().addAccumulator("numIterations", this.numIterations)
                     ↪ ;
268         }
269
270         /**
271          * Filter out and return the COI object if not all centroids has converged
                 ↪ (meaning that they have moved more
272          * than 0.01). A returned object will allows the iteration to continue. If
                 ↪ no object is returned and the
273          * DataSet is empty, the iteration will stop.
274          *
275          * @param COITuple Tuple7 with one COI object.
276          * @return boolean True if not converged, False if it has converged
277          */
278         @Override
279         public boolean filter(Tuple7<Integer, Integer, Point, Double, Double[],
                 ↪ Centroid, COI> COITuple) {
280
281             double[] oldNewCentroidDistances = COITuple.f6.distMap;
282
283             boolean hasConverged = false;
284
```

```
285             //System.out.println("Convergence criteria is: " + convergence_criteria
                    ↪ );
286
287             // Add one to the number of iterations
288             this.numIterations.add(1);
289
290             // Loop trough all oldNewCentroidDistances to check for convergence
291             for (double distance : oldNewCentroidDistances) {
292
293
294                 // Checking if one of the centroids has moved more than 0.01.
295                 // If one has, the algorithm has not converged
296                 if (distance > convergence_criteria) {
297                     hasConverged = true;
298
299                     // Jump out of the loop and return result
300                     break;
301                 }
302             }
303
304             return hasConverged;
305         }
306     }
307
308     /**
309      * This class implements the MapFunction to extract the COI object from a tuple
310      */
311     public static class ExtractCOI implements MapFunction<Tuple7<Integer, Integer,
            ↪ Point, Double, Double[], Centroid, COI>, COI> {
312
313         /**
314          * Takes a Tuple7 that includes the COI object and return only this object.
                ↪  In this implementation no more
315          * than one COI object should exist at any time.
316          *
317          * @param COITuple Tuple7 with a COI object
318          * @return COI, should only be one object that is returned.
319          */
320         @Override
321         public COI map(Tuple7<Integer, Integer, Point, Double, Double[], Centroid,
                ↪ COI> COITuple) {
322             return COITuple.f6;
323         }
324     }
325
326     /**
327      * This class implements the MapFunction to extract all Centroids objects from
            ↪ the tuples
328      */
329     public static class ExtractCentroids implements MapFunction<Tuple7<Integer,
            ↪ Integer, Point, Double, Double[], Centroid, COI>, Centroid> {
330
331         /**
332          * Takes a Tuple7 with all information and extracts only the Centroid
                ↪ object. Multiple centroids exists,
333          * however the number of centroids is usually relatively low.
334          *
335          * @param unionData Tuple 7 with all information
336          * @return Centroid
337          */
```

```
338         @Override
339         public Centroid map(Tuple7<Integer, Integer, Point, Double, Double[],
                ↪ Centroid, COI> unionData) {
340             return unionData.f5;
341         }
342     }
343
344     /**
345      * This class implements the MapFunction to expand the points tuple
346      */
347     public static class ExpandPointsTuple implements MapFunction<Tuple4<Integer,
            ↪ Point, Double, Double[]>, Tuple7<Integer, Integer, Point, Double,
            ↪ Double[], Centroid, COI>> {
348
349         /**
350          * Takes a Tuple4 and return a Tuple7 where the fields that are not used by
                ↪  points is empty. This Tuple7
351          * is used to union all data at the end of the iteration.
352          *
353          * @param point A Tuple4 with the ID the point is assigned to, the point
                ↪ itself, the upper bound and the lower
354          *              bounds
355          * @return Tuple7 with correct key identifier (1) and the rest of the
                ↪ information.
356          */
357         @Override
358         public Tuple7<Integer, Integer, Point, Double, Double[], Centroid, COI>
359             map(Tuple4<Integer, Point, Double, Double[]> point) {
360
361             return new Tuple7<>(1, point.f0, point.f1, point.f2, point.f3, null,
                    ↪ null);
362         }
363     }
364
365     /**
366      * This class implements the RichMapFunction to initially select the nearest
            ↪ centroid to a point. This class
367      * function is utilized once before the loop begins. Field f1 is forwarded to
            ↪ improve efficiency, as this field
368      * is not changed in the function.
369      */
370     @ForwardedFields("f1")
371     public static final class SelectInitialNearestCenter extends RichMapFunction<
            ↪ Tuple4<Integer, Point, Double, Double[]>, Tuple4<Integer, Point, Double
            ↪ , Double[]>> {
372         private Collection<Centroid> centroids;
373         private Collection<COI> coiCollection;
374
375         // Accumulator used to track the total number of distance calculations
                ↪ performed
376         private IntCounter distCalcSelectInitialNearestCenter = new IntCounter();
377
378         // DEBUGGING
379         private IntCounter initAss = new IntCounter();
380
381         /**
382          * Reads the centroid values from a broadcast DataSet and reads the COI
                ↪ value from a broadcast DataSet
383          *
384          * @param parameters The runtime parameters
```

```
385          */
386         @Override
387         public void open(Configuration parameters) {
388             this.centroids = getRuntimeContext().getBroadcastVariable("centroids");
389             this.coiCollection = getRuntimeContext().getBroadcastVariable("coi");
390 
391             // Registering the accumulator object and defining the name of the
                    ↪ accumulator
392             getRuntimeContext().addAccumulator("distCalcSelectInitialNearestCenter"
                    ↪ , this.distCalcSelectInitialNearestCenter);
393 
394             // DEBUGGING
395             //getRuntimeContext().addAccumulator("initAss", this.initAss);
396 
397         }
398 
399         /**
400          * This function select the initial centroids each point is assigned to. It
                ↪  also calculates the lower bounds
401          * and the upper bound.
402          *
403          * @param tuple A Tuple4 with 0 as the initial centroid ID the point is
                ↪ assigned to. This 0 will be overwritten
404          * @return A Tuple4 with nearest centroid ID
405          */
406 
407         @Override
408         public Tuple4<Integer, Point, Double, Double[]> map(Tuple4<Integer, Point,
                ↪ Double, Double[]> tuple) {
409 
410             // DEBUGGING
411             //this.initAss.add(1);
412 
413             // The COI object is extracted to a single object
414             COI coi = new ArrayList<>(coiCollection).get(0);
415 
416             Point point = tuple.f1;
417             Double[] lb = tuple.f3;
418 
419             Centroid c = centroids.iterator().next();
420 
421             // Calculating the distance between the first centroid in the
                    ↪ collection and this point
422             double minDistance = point.euclideanDistance(c);
423 
424             // Increasing the accumulator for number of distance calculations
425             this.distCalcSelectInitialNearestCenter.add(1);
426 
427             double dist;
428             int closestCentroidId = c.id;
429 
430             //System.out.println(this.initAss + ": Setting closestCentroidID to " +
                    ↪  closestCentroidId);
431 
432             lb[closestCentroidId-1] = minDistance;
433 
434             // Loop trough all cluster centers
435             for (Centroid centroid : centroids) {
436 
```

```
437                 if((0.5 * coi.iCD[closestCentroidId-1][centroid.id-1]) <
                        ↪ minDistance) {
438
439                     // Calculating the lower bound for this centroid and saving the
                            ↪  distance between the centroid
440                     // and this point
441                     lb[centroid.id-1] = dist = point.euclideanDistance(centroid);
442
443                     // Increasing the accumulator for number of distance
                            ↪ calculations
444                     this.distCalcSelectInitialNearestCenter.add(1);
445
446                     if(dist < minDistance) {
447
448                         minDistance = dist;
449                         closestCentroidId = centroid.id;
450                     }
451                 }
452             }
453
454             Double ub = minDistance;
455
456             // Emit a new record with the current closest center ID and the data
                    ↪ point.
457             return new Tuple4<>(closestCentroidId, point, ub, lb);
458         }
459     }
460
461     /**
462      * This class implements the RichMapFunction to select the nearest cluster
             ↪ center for a point. This class function
463      * is utilized within the iteration. Field f1 is forwarded as this is not
             ↪ changed.
464      */
465     @ForwardedFields("f1")
466     public static final class SelectNearestCenter extends RichMapFunction<Tuple4<
            ↪ Integer, Point, Double, Double[]>, Tuple4<Integer, Point, Double,
            ↪ Double[]>> {
467         private Collection<Centroid> centroids;
468         private Collection<COI> coiCollection;
469
470         // Accumulator used to track the total number of distance calculations
                ↪ performed
471         private IntCounter distCalcSelectNearestCenter = new IntCounter();
472
473         /**
474          * Reads the centroid values from a broadcast DataSet and reads the COI
                    ↪ value from a broadcast DataSet
475          * @param parameters The runtime parameters
476          */
477         @Override
478         public void open(Configuration parameters) {
479             this.centroids = getRuntimeContext().getBroadcastVariable("centroids");
480             this.coiCollection = getRuntimeContext().getBroadcastVariable("coi");
481
482             // Registering the accumulator object and defining the name of the
                    ↪ accumulator
483             getRuntimeContext().addAccumulator("distCalcSelectNearestCenter", this.
                    ↪ distCalcSelectNearestCenter);
484         }
```

```
485
486         /**
487          * This function takes a clustered point and assigns it to a new centroid
                 ↪ if a centroid is deemed to be
488          * closer than the already assigned centroid. The lower bounds and the
                 ↪ upper bound is updated as well.
489          *
490          * @param tuple A Tuple4 with an ID for assigned centroid, the point itself
                 ↪ , upper bound and the lower bounds
491          * @return A Tuple4 with possibly new ID for assigned centroid, the point
                 ↪ itself, upper bound and the
492          *          lower bounds
493          */
494         @Override
495         public Tuple4<Integer, Point, Double, Double[]> map(Tuple4<Integer, Point,
                 ↪ Double, Double[]> tuple) {
496
497             // Unpacking the COI object
498             COI coi = new ArrayList<>(coiCollection).get(0);
499
500             // Unpacking the centroids and sorting them
501             Centroid[] centroidArray =  centroids.toArray(new Centroid[0]);
502             Arrays.sort(centroidArray);
503
504             Point point = tuple.f1;
505
506             Integer closestCentroidId = tuple.f0;
507             boolean upperBoundUpdated = false;
508
509             Double[] currentLb = tuple.f3;
510             Double currentUb = tuple.f2;
511
512             Double[] newLb = currentLb;
513             Double newUb = currentUb;
514
515             // Calculating k new lower bounds
516             for (int i = 0; i < coi.k; i++) {
517                 newLb[i] = Math.max((currentLb[i] - coi.distMap[i]), 0.0);
518                 //System.out.println("Lower bound nr. " + i + " is " + newLb[i]);
519             }
520
521             // Checking if the upperBound need to get updated
522             if (coi.distMap[closestCentroidId - 1] > 0.0) {
523
524                 // Updating the upperBound by adding the distance the currently
                         ↪ assigned centroid has moved
525                 newUb = currentUb + coi.distMap[closestCentroidId - 1];
526                 upperBoundUpdated = true;
527             }
528
529             double dist1;
530             double dist2;
531
532             if (newUb > coi.minCD[closestCentroidId - 1]) {
533
534                 // check all cluster centers
535                 for (Centroid centroid : centroids) {
536
537                     // Check if this centroid ID is not current assigned centroid
                             ↪ ID
```

```
538                     // Check if upper bound is greater than this points lower bound
                            ↪  for centroid
539                     // Check if DISTANCE(THIS CENTROID AND P'S CURRENT ASSIGNED
                            ↪ CENTROID)  >= 2 * MIN_DIST
540                     if ((centroid.id != closestCentroidId) && (newUb > newLb[
                            ↪ centroid.id-1]) && (newUb > (0.5*coi.iCD[
                            ↪ closestCentroidId-1][centroid.id-1]))) {
541
542                         // Do only this if upper bound is updated
543                         if (upperBoundUpdated) {
544                             dist1 = point.euclideanDistance(centroidArray[
                                    ↪ closestCentroidId-1]);
545
546                             // Increasing the accumulator for number of distance
                                    ↪ calculations
547                             this.distCalcSelectNearestCenter.add(1);
548
549                             newUb = dist1;
550                             newLb[closestCentroidId-1] = dist1;
551                             upperBoundUpdated = false;
552                         } else {
553                             dist1 = newUb;
554                         }
555
556
557                         if (dist1 > newLb[centroid.id-1] || (dist1 > (0.5 * coi.iCD
                                ↪ [closestCentroidId-1][centroid.id-1]))) {
558                             dist2 = point.euclideanDistance(centroid);
559
560                             // Increasing the accumulator for number of distance
                                    ↪ calculations.
561                             this.distCalcSelectNearestCenter.add(1);
562
563                             newLb[centroid.id-1] = dist2;
564
565                             if(dist2 < dist1) {
566                                 closestCentroidId = centroid.id;
567
568                                 newUb = dist2;
569                                 upperBoundUpdated = false;
570                             }
571                         }
572                     }
573                 }
574             }
575
576             // emit a new record with the center id and the data point.
577             return new Tuple4<>(closestCentroidId, tuple.f1, newUb, newLb);
578         }
579     }
580
581     /**
582      * This class implements the MapFunction to append an integer to each point,
                ↪ which is used to count total
583      * occurrences of points assigned to one centroid. Field f0 (the ID of the
                ↪ cluster) and field f1 (the point)
584      * is forwarded directly to the input.
585      */
586     @ForwardedFields("f0;f1")
```

```
587     public static final class CountAppender implements MapFunction<Tuple4<Integer,
            ↪ Point, Double, Double[]>, Tuple3<Integer, Point, Long>> {
588
589         /**
590          * Takes in a Tuple4 with all point information and return a Tuple3 with
                ↪ the ID of the cluster the point is
591          * assigned to, the point itself and a counter variable. The upper and
                ↪ lower bounds are removed since they are
592          * not needed in the calculation of the new centroids (it would complicate
                ↪ the use of the reduce
593          * function later if they were preserved also)
594          *
595          * @param pointData A Tuple4 with all point information
596          * @return A Tuple3 with only necessary data and a counter
597          */
598         @Override
599         public Tuple3<Integer, Point, Long> map(Tuple4<Integer, Point, Double,
                ↪ Double[]> pointData) {
600             return new Tuple3<>(pointData.f0, pointData.f1, 1L);
601         }
602     }
603
604     /**
605      * This class implements the ReduceFunction in order to take in two points and
            ↪ reduce them to one, by adding the
606      * points and the counters. Field 0 is forwarded as this does not change.
607      */
608     @ForwardedFields("0")
609     public static final class CentroidAccumulator implements ReduceFunction<Tuple3<
            ↪ Integer, Point, Long>> {
610
611         /**
612          * Takes two points and add them together, in addition to adding the
                ↪ counters together. When all points are
613          * accumulated the new centroid can be calculated in a later function using
                ↪  output from this function.
614          *
615          * @param point1 First point to be reduced to one
616          * @param point2 Second point to be reduced to
617          * @return A single Tuple3 is returned with the total values from the two
                ↪ points.
618          */
619         @Override
620         public Tuple3<Integer, Point, Long> reduce(Tuple3<Integer, Point, Long>
                ↪ point1, Tuple3<Integer, Point, Long> point2) {
621             return new Tuple3<>(point1.f0, point1.f1.add(point2.f1), point1.f2 +
                    ↪ point2.f2);
622         }
623     }
624
625     /**
626      * This class implements the MapFunction in order to compute a new centroid
            ↪ based on a fully accumulated point
627      */
628     public static final class CentroidAverager implements MapFunction<Tuple3<
            ↪ Integer, Point, Long>, Tuple7<Integer, Integer, Point, Double, Double
            ↪ [], Centroid, COI>> {
629
630         /**
```

```
631         * Takes in a fully accumulated point and return the centroid which is the
                ↪ average of all current points
632         * assigned to the centroid.
633         *
634         * @param value The accumulated point
635         * @return Tuple7 with all centroid field filled and all other fields
                ↪ contain dummy data. The key for the tuple
636         *          is also set to 0, indicating that it holds a centroid.
637         */
638        @Override
639        public Tuple7<Integer, Integer, Point, Double, Double[], Centroid, COI> map
               ↪ (Tuple3<Integer, Point, Long> value) {
640            Centroid centroid = new Centroid(value.f0, value.f1.div(value.f2));
641
642            // Initilazing empty values to put in the tuple
643            Double emptyDouble = 0.0;
644            Double[] emptyDoubleArray = {0.0};
645
646            return new Tuple7<>(0, 0, null, emptyDouble, emptyDoubleArray, centroid
                   ↪ , null);
647        }
648    }
649
650    /**
651     * This class implements the RichGroupReduceFunction in order to compute the
           ↪ COI (Carry-Over-Information) object.
652     * The COI object contains information about the inter-centroid distances (
           ↪ distances between each centroid),
653     * how much each centroid has moved between last and current iteration, the
           ↪ minimum distance between each centroid,
654     * and what K is.
655     */
656    public static class computeCOI extends RichGroupReduceFunction<Centroid, Tuple7
           ↪ <Integer, Integer, Point, Double, Double[], Centroid, COI>> {
657        private Collection<Centroid> centroidCollection;
658
659        // Accumulator used to track the total number of distance calculations
               ↪ performed
660        private IntCounter distCalcComputeCOI = new IntCounter();
661
662        /**
663         * Reads the centroid values from a broadcast DataSet into a collection.
664         *
665         * @param parameters The runtime parameters
666         */
667        @Override
668        public void open(Configuration parameters) {
669            this.centroidCollection = getRuntimeContext().getBroadcastVariable("
                   ↪ oldCentroids");
670
671            // Registering the accumulator object and defining the name of the
                   ↪ accumulator
672            getRuntimeContext().addAccumulator("distCalcComputeCOI", this.
                   ↪ distCalcComputeCOI);
673        }
674
675        /**
676         * This function use the centroid from the last iteration and the centroids
                ↪  from this iteration in order to
```

```
677          * produce the COI (Carry-Over-Information) object. The old centroids are
                 ↪ passed to this function via a
678          * broadcast variable, while the new centroids are passed directly to the
                 ↪ function trough its input (since
679          * it is a GroupReduceFunction that reduce the input from a whole group).
680          *
681          * @param iterable This is the centroids from the current iteration, all
                 ↪ sent to the group reduce function
682          * @param collector Tuple7 where the COI object is stored, the correct key
                 ↪ is set and returned
683          */
684         @Override
685         public void reduce(Iterable<Centroid> iterable, Collector<Tuple7<Integer,
                 ↪ Integer, Point, Double, Double[], Centroid, COI>> collector) {
686
687             // Instantiate list to store current / new centroids
688             List<Centroid> newCentroids = new ArrayList<>();
689
690             // Convert Collection centroidCollection to ArrayList
691             List<Centroid> oldCentroids = new ArrayList<>(centroidCollection);
692
693             // Convert Iterable iterable to ArrayList by adding to existing list
                     ↪ instantiated above
694             for(Centroid c : iterable) {
695                 newCentroids.add(c);
696             }
697
698             // Store the size of the List, which is used in allocation of array
                     ↪ below
699             int dims = newCentroids.size();
700
701             // Ensure that the centroids are sorted in ascending order on their ID
                     ↪ in order to be able to use
702             // them in the for-loop below.
703             Collections.sort(newCentroids);
704             Collections.sort(oldCentroids);
705
706             // Allocate the multidimensional array
707             double[][] matrix = new double[dims][dims];
708             double[] minCD = new double[dims];
709             double[] distMap = new double[dims];
710
711             // Computes the distances between every centroid and place them in List
                     ↪  l
712             for(int i = 0; i < newCentroids.size(); i++) {
713                 double minVal = Double.MAX_VALUE;
714
715                 // This represents the outer centroid
716                 Centroid ci = newCentroids.get(i);
717
718                 for(int j = 0; j < newCentroids.size(); j++) {
719
720                     // Check that k != k'
721                     if (i != j) {
722
723                         // This represents the inner centroid
724                         Centroid cj = newCentroids.get(j);
725
726                         // Calculate the distance between the two centroids
727                         double dist = ci.euclideanDistance(cj);
```

```
728
729                         // Increasing the accumulator for number of distance
                                ↪ calculations
730                         this.distCalcComputeCOI.add(1);
731
732                         // Update the matrix with the distance
733                         matrix[i][j] = dist;
734
735                         // Look for the smallest value and update if smaller
736                         if (dist < minVal) {
737                             minVal = dist;
738                         }
739
740                     } else {
741                         // If k == k', distance is automatically 0
742                         matrix[i][j] = 0;
743                     }
744                 }
745
746                 // Update minCD with 0.5 of minVal
747                 minCD[i] = (minVal / 2);
748             }
749
750             // Produce the distMap
751             for (int i = 0; i < dims; i++) {
752                 distMap[i] = newCentroids.get(i).euclideanDistance(oldCentroids.get
                        ↪ (i));
753
754                 // Increasing the accumulator for number of distance calculations
755                 this.distCalcComputeCOI.add(1);
756             }
757
758             // Make the new COI object
759             COI coi = new COI(matrix, minCD, distMap);
760
761             // Initilazing empty values to put in the tuple
762             Double emptyDouble = 0.0;
763             Double[] emptyDoubleArray = {0.0};
764
765             Tuple7<Integer, Integer, Point, Double, Double[], Centroid, COI>
                    ↪ coiTuple = new Tuple7<>(2, 0, null, emptyDouble,
                    ↪ emptyDoubleArray,  null, coi);
766
767             // Add it to the collector which will return it
768             collector.collect(coiTuple);
769         }
770     }
771 }
```

B.2 Read.java

Code listing B.2: Read.java

```
1 package com.ringdalen.kmeansti.util;
2
3 import com.ringdalen.kmeansti.datatype.DataTypes.Centroid;
4 import com.ringdalen.kmeansti.datatype.DataTypes.Point;
```

```
5
6  import org.apache.flink.api.common.functions.MapFunction;
7  import org.apache.flink.api.java.DataSet;
8  import org.apache.flink.api.java.ExecutionEnvironment;
9  import org.apache.flink.api.java.tuple.Tuple4;
10 import org.apache.flink.api.java.utils.ParameterTool;
11
12 import java.util.Arrays;
13
14 public class Read {
15     /**
16      * Function to map data from a file to Centroid objects
17      */
18     public static DataSet<Centroid> CentroidsFromFile(ParameterTool params,
       ↪ ExecutionEnvironment env) {
19
20         DataSet<Centroid> centroids;
21
22         // Parsing d features, plus the ID (thats why the +1 is included) from file
           ↪  to Centroid objects
23         centroids = env.readTextFile(params.get("centroids"))
24                 .map(new ParseCentroidData(params.getInt("d")));
25
26         return centroids;
27     }
28
29     /**
30      * Function to map data from a file to Point objects
31      */
32     public static DataSet<Tuple4<Integer, Point, Double, Double[]>> PointsFromFile(
       ↪ ParameterTool params, ExecutionEnvironment env) {
33
34         DataSet<Tuple4<Integer, Point, Double, Double[]>> points;
35
36         // Parsing d features from file to Point objects
37         points = env.readTextFile(params.get("points"))
38                 .map(new ParsePointData(params.getInt("d"), params.getInt("k")));
39
40         return points;
41     }
42
43     /** Reads the input data and generate points */
44     public static class ParsePointData implements MapFunction<String, Tuple4<
       ↪ Integer, Point, Double, Double[]>> {
45         double[] row;
46         int k;
47         int trueClass;
48
49         public ParsePointData(int d, int k){
50             this.row = new double[d];
51             this.k = k;
52             this.trueClass = 0;
53         }
54
55         @Override
56         public Tuple4<Integer, Point, Double, Double[]> map(String s) {
57             String[] buffer = s.split("␣");
58
59             // Setting the true class for this point, used to check accuracy of
                   ↪ clustering later
```

```
60              trueClass = Integer.parseInt(buffer[0]);
61
62              // Extracting values from the input string
63              for(int i = 0; i < row.length; i++) {
64                  row[i] = Double.parseDouble(buffer[i+1]);
65              }
66
67              // Declaring the initial upper bound
68              Double ub = -1.0;
69
70              // Declaring the initial lower bounds
71              Double[] lb = new Double[k];
72              Arrays.fill(lb, 0.0);
73
74              return new Tuple4<>(-1, new Point(trueClass, row), ub, lb);
75          }
76      }
77
78      /** Reads the input data and generate centroids */
79      public static class ParseCentroidData implements MapFunction<String, Centroid>
            ↪ {
80          double[] row;
81
82          public ParseCentroidData(int d){
83              row = new double[d];
84          }
85
86          @Override
87          public Centroid map(String s) {
88              String[] buffer = s.split("␣");
89              int id = Integer.parseInt(buffer[0]);
90
91              // buffer is +1 since this array is one longer
92              for(int i = 0; i < row.length; i++) {
93                  row[i] = Double.parseDouble(buffer[i+1]);
94              }
95
96              return new Centroid(id, row);
97          }
98      }
99  }
```

B.3 DataTypes.java

Code listing B.3: DataTypes.java

```
1   package com.ringdalen.kmeansti.datatype;
2
3   import java.io.Serializable;
4
5   public class DataTypes {
6       /**
7        * A n-dimensional point.
8        */
9       public static class Base implements Serializable {
10
11          public double[] features;
```

```java
        public int dimension;

        /** A public no-argument constructor is required for POJOs (Plain Old Java Objects) */
        public Base() {}

        /** A public constructor that takes the features, represented as an array of doubles as the argument */
        public Base(double[] features) {
            this.features = features;
            this.dimension = features.length;
        }

        /** Function that divides this point with a given value */
        public Base div(long val) {
            for(int i = 0; i < dimension; i++) {
                features[i] /= val;
            }
            return this;
        }

        /** Function that return the euclidian distance between this point and any given point */
        public double euclideanDistance(Base other) {
            double dist = 0;

            for(int i = 0; i < dimension; i++) {
                dist += Math.pow((features[i] - other.features[i]), 2.0);
            }

            return Math.sqrt(dist);
        }

        /** Function to represent the point in a string */
        @Override
        public String toString() {
            StringBuilder s = new StringBuilder();

            for(int i = 0; i < dimension; i++) {
                if (i < dimension-1) {
                    s.append(features[i]).append("␣");
                } else {
                    s.append(features[i]);
                }
            }

            return s.toString();
        }
    }

    public static class Point extends Base implements Serializable {
        public int trueClass;

        /** A public no-argument constructor is required for POJOs (Plain Old Java Objects) */
        public Point() {}

        public Point(int trueClass, double[] features) {
            super(features);
            this.trueClass = trueClass;
```

```
68          }
69
70          /** Function that adds this point with any given point */
71          public Point add(Point other) {
72              for(int i = 0; i < dimension; i++) {
73                  features[i] += other.features[i];
74              }
75
76              return this;
77          }
78
79          /**
80           * Function to represent the point as a string
81           *
82           * @return String with true class, as well as string from base class
83           */
84          @Override
85          public String toString()  {
86              return (trueClass + "␣" + super.toString());
87          }
88      }
89
90      /**
91       * A n-dimensional centroid, basically a point with an ID.
92       */
93      public static class Centroid extends Base implements Comparable<Centroid>{
94
95          /** The ID of an centroid, which also represents the cluster */
96          public int id;
97
98          /** A public no-argument constructor is required for POJOs (Plain Old Java
                ↪ Objects) */
99          public Centroid() {}
100
101         /** A public constructor that takes an id and the features, represented as
                ↪ an array as the arguments */
102         public Centroid(int id, double[] features) {
103             super(features);
104             this.id = id;
105         }
106
107         /** A public constructor that takes an id and a Point as the arguments */
108         public Centroid(int id, Base p) {
109             super(p.features);
110             this.id = id;
111         }
112
113         public Integer getID() {
114             return id;
115         }
116
117         /** A method to allow for comparing the ID of two different centroids. Used
                ↪  for sorting */
118         public int compareTo(Centroid c) {
119             return this.getID().compareTo(c.getID());
120         }
121
122         /** Function to represent the point in a string */
123         @Override
124         public String toString() {
```

```
125             return id + "␣" + super.toString();
126         }
127     }
128
129     /**
130      * A class used to store information that will be carried over to the next
             ↪ iteration.
131      * COI is an abbreviation of "Carry-Over-Information"
132      */
133     public static class COI implements Serializable {
134         public double[][] iCD;
135         public double[] minCD;
136         public double[] distMap;
137         public int k;
138
139         public COI(double[][] iCD, double[] minCD, double[] distMap) {
140             this.iCD = iCD;
141             this.minCD = minCD;
142             this.distMap = distMap;
143             this.k = distMap.length;
144         }
145     }
146 }
```

Ola Flølo Ringdalen

Applying K-means with Triangle Inequality on Apache Flink

NTNU
Norwegian University of
Science and Technology