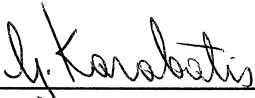


## APPROVAL SHEET

Title of Thesis: The effect of k-nearest neighbor's classifier for intrusion detection of streaming of Net-Flows in the Apache Spark Environment.

Name of Candidate: Muthukumar K. Thevar  
Master of Science in Information Systems, 2017

Thesis and Abstract Approved:   
Dr. George Karabatis  
Associate Professor  
Department of Information Systems

Date Approved: 4/13/2017

## ABSTRACT

Title of Document: THE EFFECT OF K-NEAREST NEIGHBORS  
CLASSIFIER FOR INTRUSION DETECTION  
OF STREAMING OF NET-FLOWS IN THE  
APACHE SPARK ENVIRONMENT

Muthukumar Thevar, Masters of Science  
Information Systems, 2017

Directed by: Associate Professor, Dr. George Karabatis  
Department of Information Systems

An Intrusion Detection System (IDS) is built with the purpose to detect normal and attack packets in network traffic data. Due to enormous amount of data present in the network traffic, analyzing all the individual packets present is both an impractical task which also increases the system performance overhead. To solve this problem, another technique is employed, which aggregates packet information into flows and reduces the amount of data to be examined from the network traffic. In addition, IDS efficiency is increased by the use of the k-NN classification algorithm to classify the incoming connections as normal or suspicious. Combining the flow based Intrusion detection approach and k-NN classifier in the Spark Streaming framework has helped develop a system which is able to detect attacks in real time. In this thesis, the KDD-99 data set has been used for testing the proposed approaches. Experimental results show that Apache Spark Streaming, a modern distributed stream processing system provides enough throughput to process large volumes of data in shorter span of time which is suitable for network traffic monitoring.

THE EFFECT OF K-NEAREST NEIGHBORS CLASSIFIER  
FOR INTRUSION DETECTION OF STREAMING NET-FLOWS  
IN APACHE SPARK ENVIRONMENT

by

Muthukumar Thevar

Thesis submitted to the Faculty of the Graduate School of the  
University of Maryland, Baltimore County in partial fulfillment  
of the requirements for the degree of  
Master of Science in  
Information Systems  
2017

Advisory Committee:

Dr. George Karabatis, Chair/Advisor

Dr. Jianwu Wang, Co-Advisor

Dr. Vandana Janeja

© Copyright by  
Muthukumar Thevar  
2017



# Acknowledgement

I would like to express my gratitude to my supervisor Dr. George Karabatis for the useful comments, remarks and engagement through the learning process of this master thesis. Furthermore I would like to thank Dr. Jianwu Wang for suggesting new ideas to the topic as well for the support on the way.

I would also like to thank the committee member Dr. Vandana Janeja for being on the examination committee and for providing invaluable feedback. I would like to thank Dr. Ahmed Aleroud for his range of ideas and helping me understand my research area better. I am grateful to Rishi Sankineni and Sai Chaithanya for their immense contribution in developing framework for Information gain and similarity calculation respectively.

I would like to thank my parents for their constant support and encouragement. I would also like to take this opportunity to thank my friends in the DATA SCIENCE lab @ UMBC and Abu Zaher Md Faridee particularly for their tips on research ideas and suggestion.

# Contents

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Introduction</b>                          | <b>1</b>  |
| 1.1      | Significance of the Problem . . . . .        | 2         |
| 1.2      | Summary of the Approach . . . . .            | 3         |
| 1.3      | Contribution of the Thesis . . . . .         | 4         |
| <b>2</b> | <b>Background and Related Work</b>           | <b>5</b>  |
| 2.1      | Snort . . . . .                              | 5         |
| 2.2      | NetFlow . . . . .                            | 6         |
| 2.3      | Packet to flow conversion . . . . .          | 7         |
| 2.4      | Classifier . . . . .                         | 8         |
| 2.4.1    | Evaluation of Classifier . . . . .           | 8         |
| 2.5      | Apache Spark . . . . .                       | 9         |
| 2.5.1    | Spark Stack . . . . .                        | 10        |
| 2.5.2    | Programming Model (RDD) . . . . .            | 11        |
| 2.6      | Related Work . . . . .                       | 12        |
| <b>3</b> | <b>Methodology</b>                           | <b>14</b> |
| 3.1      | Overview of the Approach . . . . .           | 15        |
| 3.2      | Snort Architecture . . . . .                 | 16        |
| 3.2.1    | Snort Alert Generation . . . . .             | 19        |
| 3.3      | Capturing via tshark/Wireshark . . . . .     | 21        |
| 3.4      | Flow Generation from Packet Traces . . . . . | 21        |
| 3.5      | Dataset and Source Description . . . . .     | 22        |
| 3.6      | Nearest Neighbor Classifier . . . . .        | 23        |
| 3.6.1    | k - Nearest Neighbor Functions . . . . .     | 25        |
| 3.7      | Map Reduce Paradigm . . . . .                | 26        |
| 3.8      | Spark Streaming Framework . . . . .          | 28        |
| 3.8.1    | Discretized Streams (D-Streams) . . . . .    | 29        |
| 3.9      | RDD to Data frame . . . . .                  | 30        |
| 3.10     | Spark Benchmarking . . . . .                 | 30        |
| 3.11     | AWK & Netcat Utility . . . . .               | 33        |
| <b>4</b> | <b>Implementation and Evaluation</b>         | <b>34</b> |
| 4.1      | Implementation . . . . .                     | 34        |
| 4.1.1    | Data set description . . . . .               | 35        |
| 4.2      | Evaluation . . . . .                         | 38        |
| 4.3      | Spark WebUI . . . . .                        | 40        |
| 4.4      | Spark Benchmarking . . . . .                 | 42        |
| 4.5      | Spark JSON log results . . . . .             | 44        |

|  |           |
|--|-----------|
| <b>5 Conclusion &amp; Future Works</b> | <b>46</b> |
| 5.1 Conclusions . . . . .              | 46        |
| 5.2 Future Works . . . . .             | 47        |
| <b>Bibliography</b>                    | <b>48</b> |



# List of Tables

|     |  |    |
|-----|--|----|
| 2.1 | Evaluation Measures . . . . .                                | 9  |
| 2.2 | Confusion Matrix . . . . .                                   | 9  |
| 4.1 | Features list and Description of KDDCup'99 Dataset . . . . . | 36 |
| 4.2 | Features list and Description of KDDCup'99 Dataset . . . . . | 37 |
| 4.3 | Attack Classification & data types . . . . .                 | 37 |
| 4.4 | Feature Score of Top 25 Attributes . . . . .                 | 38 |

# List of Figures

|      |  |    |
|------|--|----|
| 1.1  | System Overview . . . . .  | 4  |
| 2.1  | Snort Basic Software Component . . . . .   | 6  |
| 2.2  | NetFlow Overview . . . . .   | 6  |
| 2.3  | One NetFlow Record provides a Significant Amount of Information . . . . .          | 7  |
| 2.4  | Spark Stack . . . . .  | 10 |
| 3.1  | Architecture of Maya Cluster . . . . .   | 15 |
| 3.2  | Detailed Approach . . . . .  | 16 |
| 3.3  | Snort Architecture . . . . .   | 17 |
| 3.4  | Structure of Snort IDS Rule . . . . .  | 18 |
| 3.5  | Structure of Snort IDS Rule Header . . . . .                                       | 18 |
| 3.6  | Example of Snort IDS Rule . . . . .  | 19 |
| 3.7  | Alert Snippet . . . . .  | 20 |
| 3.8  | Packet aggregation to create flows . . . . .                                       | 22 |
| 3.9  | Data Flow Overview of MapReduce . . . . .  | 27 |
| 3.10 | High Level Overview of Spark Streaming System . . . . .                            | 30 |
| 3.11 | Map Reduce Engine . . . . .  | 32 |
| 4.1  | Evaluation Metrics of Classifier . . . . .   | 39 |
| 4.2  | Spark Jobs . . . . .   | 41 |
| 4.3  | Spark Stages . . . . .   | 41 |
| 4.4  | Spark RDD . . . . .  | 42 |
| 4.5  | Spark Executors . . . . .  | 42 |
| 4.6  | Spark Streaming . . . . .  | 43 |
| 4.7  | Average execution time in seconds with varying number of computing nodes . . . . . | 43 |
| 4.8  | Processing time for a batch . . . . .  | 44 |
| 4.9  | Elapsed Time for a batch . . . . .   | 45 |
| 4.10 | Throughput . . . . .   | 45 |

# Chapter 1

## Introduction

The advanced development in computer systems and internet has modified the way people think and do things. In the past, sending an email would take hours or even days, whereas now it can be delivered almost instantaneously just by a click of a mouse. Now, people can use Information Technology (IT) infrastructure to conveniently communicate with each other from different, even remote geographical locations through web chat or video conferencing. However, along with the development of Computer Systems and IT infrastructures large amounts of personal and sensitive information like SSN, phone number, address, bank accounts etc. are hosted on servers which are available through the world wide web [10]. New threats are developed by hackers each day, who aim to gain unauthorized access to computer systems and compromise the integrity, validity and confidentiality of stored data.

Malicious activities in the internet are also known as intrusions [10]. An intrusion or a threat can be defined as any action that attempts unauthorized access, information manipulation, or rendering the system unstable by exploiting the existing vulnerabilities of the system. These vulnerabilities used by the attackers weaken the security of the system [18]. An Intrusion Detection System (IDS) is a mechanism that tries to identify a set of actions or unallowed actions or behaviors in a computer system. There are two main types of intrusion detection techniques: misuse detection and anomaly detection. Misuse detection recognizes a suspicious behavior by comparing its signature with a stored database of attacks signatures;

the only drawback of this techniques is that it cannot detect new attacks. Snort is an example of an IDS using misuse detection techniques. Anomaly detection is another technique which creates a model of normal behavior, and tries to detect any abnormal deviation from that model resulting in generation of corresponding alerts [11].

## 1.1 Significance of the Problem

In most IDSs however, there are high rates of false positive and false negatives which can be difficult to deal with. “A false positive is an instance where an IDS incorrectly identifies a benign activity to be malicious while a false negative occurs when the IDS fails to detect a malicious activity” [10]. The number of alerts collected by an IDS per day can be more than 15000 and the number of false positives (FP) can be thousands per day which is large number of alerts that can cause the network administrator to lose confidence on the IDS[18].

Due to increase in Internet utilization, the size of network traffic data is growing exponentially and it is very difficult to process it, using the traditional data processing tools. Fast and efficient intrusion detection is a challenging problem due to the high volume and the complex nature of the network traffic data. A real time intrusion detection system should be able to process large volume of data in short span of time and detect the malicious traffic as early as possible [12].

“Cyber Security Intrusion Detection System commonly requires an efficient real-time storing and processing of large size of network traffic data as well as analysis to identify malicious network traffic”[12]. The Apache Spark is an open source clustering computing framework which processes vast amounts of data in short span of time.

Some of the notable features of Apache Spark are as follows:

- Speed: Spark runs 100 times faster than Hadoop. Hadoop is a well-known tool that is used for scalable computing which uses the concept of Map Reduce

which is discussed in details in the Methodology Chapter. It is an advanced DAG (Directed Acyclic Graph) execution engine for in-memory computing.

- Ease of use: Spark offers over 80 high-level operators that make it easy to build parallel applications (apps). These operators can be used in the Scala, Python, or R language, and they can be invoked interactively from the corresponding shells that are linked to Spark.
- Runs Everywhere: Spark runs on several platforms, such as Hadoop, Mesos, standalone, or in the cloud. It can access diverse data sources stored in a variety of repositories such as HDFS, Cassandra, HBase and S3

Combining the features of Apache Spark and Intrusion Detection techniques one can build a system that processes in real time a large volume of data and performs an analysis to discover malicious activity in the network traffic.

## 1.2 Summary of the Approach

In our approach, we have implemented a system using the Spark platform which uses the IP flows to detect attacks. “A flow is defined as a set of IP packets passing an observation point in the network during a certain time interval. All packets belonging to a particular flow have a set of common properties” [28]. In Figure 1.1, which gives the overview of our systems Network Traffic data (flows) arrives from the real-time networks or any network equipment’s which is then passed to our system (IDS) to detect the normal and attack packets.

We have used the Semantic Link Network (SLN) approach to predict the attacks. A Semantic Link Network is a graph that provides semantic relations between concepts. SLN is often used in Knowledge representation. It is represented as a directed or undirected graph consisting of vertices representing the concepts and edges connecting the vertices. To efficiently process the large volume of data we used the Apache Spark Streaming framework to process the incoming data in real on near real time and predict the attack labels of the incoming flows and passed it to SLN for

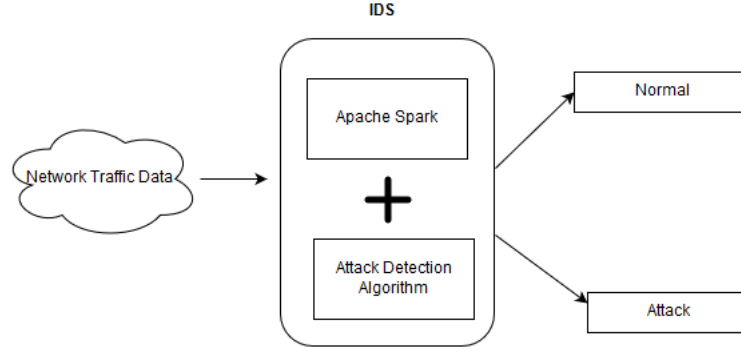


Figure 1.1: System Overview

further analysis. The incoming flow is given to the KNN classifier which is trained by the 70% of the KDDcup dataset to predict the attack labels. We used the Hadoop File System (HDFS) for the faster retrieval of the training dataset file.

### 1.3 Contribution of the Thesis

This thesis contains a number of unique contributions. They are summarized below:

- Creation of KNN classifier model in Apache pyspark module.
- Training the KNN classifier model with the 70% KDDcup dataset and testing the model with 30% of the KDDcup datasets.
- Calculation of the classification metrics like precision, recall, accuracy and F-Score of KNN classifier model.
- Capturing the Datasets in port which is then utilized by the Spark streaming context.
- Calculation of execution time taken by the system to predict dataset.

# Chapter 2

## Background and Related Work

In this chapter we will discuss the background relevant to Snort, packet to flow conversion, creation of a classifier, evaluation of classifier and Spark.

### 2.1 Snort

Network Intrusion detection systems (NIDS) are widely used tool to monitor the today's network security infrastructure. They act like a layer of defense which monitors the incoming network traffic for suspicious activities or patterns and they alert the system administrator when they detect potential hostile traffic[30].

Snort is a single-threaded user-level application that utilizes the transmission control protocol TCP/IP stack on a computer which captures and inspects packet payloads to identify possible intrusions[30]. Snort can be operated in four modes: sniffer, packet logger, NIDS, and intrusion prevention system. Snort captures raw packets with libpcap which then decodes and preprocess them before forwarding them to the detection engine. The preprocessing contains features like packet dropping, classifications, etc. The detection engine analyzes the packet headers and payloads against thousands of rules stored in a database of pre-defined attack signatures, as shown in Figure 2.1. If any one of the rule is matched, an action is taken depending on action specified in the rule configuration .

The detection engine is the most essential component of Snort, and very complex.

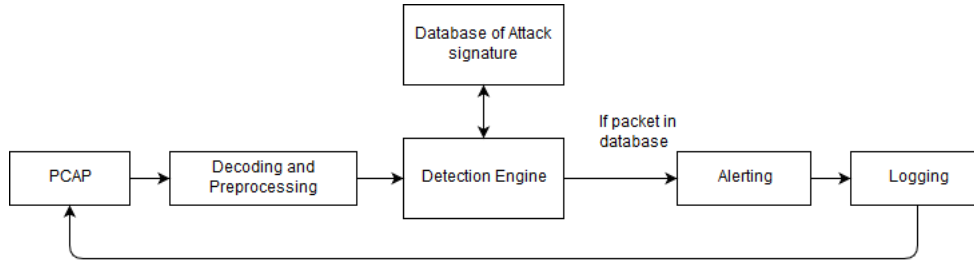


Figure 2.1: Snort Basic Software Component

It is responsible for analyzing every packet based on thousands of Snort rules that are loaded at runtime.

## 2.2 NetFlow

NetFlow is a network protocol developed by Cisco for the collection and monitoring of network traffic flow data which is created by the NetFlow-enabled routers and switches. It was developed by Cisco packet switching technology for Cisco routers. The idea behind NetFlow (Figure 2.2) is that the first packet of a flow would create NetFlow switching record on a switch or router, and subsequently, this record would be used for all the later packets of the same flow, until the expiration of the flow. Only the first packet of a flow would require an investigation for the route table to find the specific matching route[27].

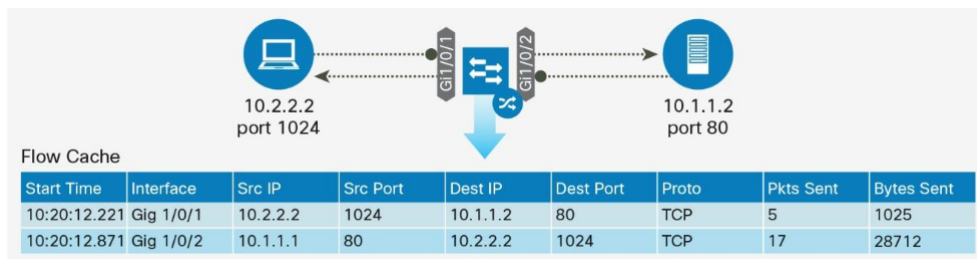


Figure 2.2: NetFlow Overview  
[27]

The flow information which is collected by flow enabled devices can be exported for network performance and behavioral analytics for security. The flows do not contain actual packet data, but rather communication metadata. It is a standard form of session data that details who, what, when and where of network traffic



(Figure 2.3). It is similar to the call records in a phone bill, but in real time. Every network transaction typically gets two flows, one in each direction[27].

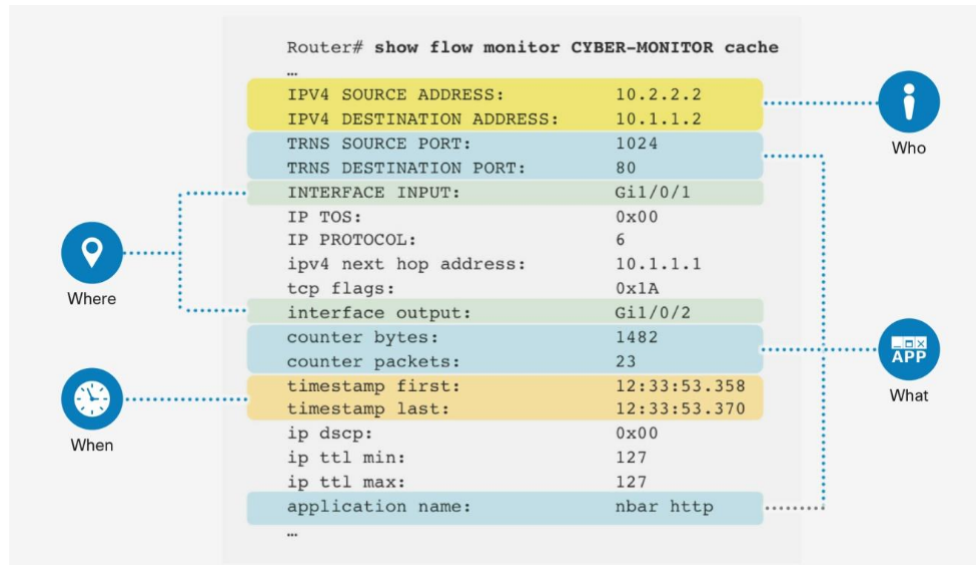


Figure 2.3: One NetFlow Record provides a Significant Amount of Information [27]

## 2.3 Packet to flow conversion

A Distributed Denial of Service (DDOS) attack is a method to make an online service unavailable by overwhelming it with traffic from multiple sources. The attacker targets a victim server by sending a large number of request in a short span of time [33]. Flow-based technology can be used to detect this kind of attacks. Flow export technologies such as Net flow and the IETF standardization effort IPFIX can be helpful to generate traffic aggregates from packets and create flows that contain aggregate information from packets without including payload information. This technique helps to reduce the amount of data that needs to be analyzed to identify attacks as well as to minimize the computation time since it is dealing with aggregate data and not with detailed information contained in packets [34]. The above technology is widely available on packet forwarding devices like Routers, Switches, and Firewalls. In this research, we have used Flow based techniques to convert the network tcpdump data into network flows.

## 2.4 Classifier

In today's world large amounts of data is getting accumulated and stored in the databases everywhere across the globe. These databases store lot of "hidden" knowledge and we need to find an automatic method for extracting this valuable information from databases. There are many algorithms which are created to extract this information from large sets of data. Classification, Association rules, and clustering are some examples of those algorithms [37]. A system that creates a classifier is one of the most widely used tools in data mining. Such systems take as input a collection of observations, each belonging to one of a small number of labels and describe its values for a fixed set of features, and output a classifier that can accurately predict the label to which a new observation belongs to [38]. In this thesis we used the k-Nearest Neighbor classification technique to find a group of k-observations in the training set that are closest to the test observation, and base the assignment of a label on the predominance of a particular label in the neighborhood. A k-NN classifier finds the similarity between an incoming observation of a flow and predicts whether this flow is normal or attack based on the similarity calculation.

### 2.4.1 Evaluation of Classifier

In this Section we present the metrics for assessing how good or how accurate a classifier is able to predict the class labels. The classifier evaluation measures include accuracy (also known as recognition rate), sensitivity (or recall), specificity, and precision. There is certain terminology for evaluating classifier which we need to understand.

#### Basic Terminology

- True positive ( $tp$ ): is the number of correct predictions that an instance is positive.
- True negative ( $tn$ ): is the number of correct predictions that an instance is negative.

- False positive ( $fp$ ): is the number of incorrect predictions that an instance is positive.
- False negative ( $fn$ ): is the number of incorrect of predictions that an instance negative.
- Positive Sample ( $p$ ): is the number of positive samples.
- Negative sample ( $n$ ): is the number of negative samples

| Measure                                 | Formula                                   |
|---|---|
| Accuracy, recognition rate              | $(tp+tn)/(p+n)$                           |
| Error rate, misclassification rate      | $(fp+fn)/(p+n)$                           |
| Sensitivity, true positive rate, recall | $tp/p$                                    |
| Specificity, true negative rate         | $tn/n$                                    |
| Precision                               | $tp/(tp+fp)$                              |
| F-Score                                 | $(2*precision*recall)/(precision+recall)$ |

Table 2.1: Evaluation Measures  
[13]

|               | predicted positive | predicted negative |
|---------------|--------------------|--------------------|
| true positive | $tp$               | $fn$               |
| true negative | $fp$               | $tn$               |

Table 2.2: Confusion Matrix

## 2.5 Apache Spark

The ever growing amount of data volumes in industry and research give us a tremendous amount of research opportunities and challenges. The sheer amount of data size has already outpaced the capabilities of single machines, creating a significant problem. To solve the above problem we need a new cluster programming models targeting the diverse computing workloads[41]. To distribute the workload across the multiple nodes, new models have been created for new workloads. For example, MapReduce supports Batch processing whereas Google has developed Dremel for interactive SQL queries and Pregel for interactive graph algorithms[41].

In 2009, at the University of California, Berkeley, the Apache Spark project was initiated to design a unified engine for distributed data processing. Spark supports a programming model similar to MapReduce but extends it with a data-sharing abstraction called “Resilient Distributed Datasets” or RDDs. Using RDDs Spark can capture a wide range of processing workloads that previously needed separate engines including SQL, streaming, machine learning and graph processing. Spark possesses several important benefits: applications are easier to develop since they use a unified API, it is more efficient to combine processing tasks, the older systems required writing data to storage and pass it to another engine whereas Spark has the capabilities to apply diverse functions over the same data, often in memory. Finally, Spark has the capabilities to apply interactive queries on a graph and Streaming machine learning[41].

### 2.5.1 Spark Stack

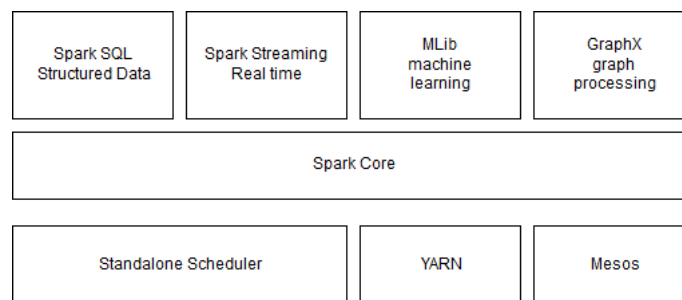


Figure 2.4: Spark Stack

#### Spark Core

The Spark Core includes the basic functionality of Spark components such as task scheduling, memory management, fault recovery, interacting with storages systems, and more. Spark core is also home to the API that defines resilient distributed datasets (RDD). RDD is the Spark’s main programming abstraction. RDDs is a collection of items distributed across many compute nodes that can be manipulated in parallel[19].

#### Spark SQL

Spark SQL is Spark’s package for working with structured data. It allows query-

ing data via SQL. Spark SQL allows developers to intermix SQL queries with the programmatic data manipulations supported by RDDs in Python, Java and Scala programming languages[19].

### **Spark Streaming**

Spark Streaming is a Spark component that enables processing of real time streams of data. For example, data stream can be log files generated by production web servers.

### **MLLIB**

Spark comes with a library containing common machine learning (ML) functionality call MLlib. MLlib contains various machine learning algorithms such as classification, regression, clustering and collaborative filtering.

### **GraphX**

GraphX is a library for manipulating graphs such a social networks friend graph and performing graph-parallel computations. GraphX also provides us various operations for manipulating graphs(e.g. subgraph and mapVertices) and common graph algorithms such as PageRank and triangle counting.

### **Cluster Manager**

Internally, spark is optimized to scale from one to many thousands of compute nodes. Spark can run on variety of cluster managers including Hadoop YARN, Apache Mesos and even it works as a standalone scheduler such as installing spark on an empty set of machines.

## **2.5.2 Programming Model (RDD)**

The key programming abstraction in spark is RDDs, which are fault tolerant collections of objects partitioned across a cluster that can be manipulated in parallel. RDDs can be created by using operation called “transformation” which includes functions like map, filter and group by to the data[41]. Let us see an example of how to create an RDD in Scala from a HDFS file.

```
lines= spark.textFile("hdfs://..")
```

```
errors=lines.filter(s=> s.startswith("ERROR"))  
println("Total Errors:" errors.count())
```

The first line defines an RDD backed by a file in the Hadoop Distributed File System (HDFS) as a collection of lines of text. The second line calls the filter transformation to derive a new RDD from lines. The last line calls count, another type of RDD operation called an “action” that returns a result to the program.

RDDs have the following advantages:

- Spark evaluates RDDs lazily, which helps to find an efficient path for the user computation.

For example, a transformation returns a new RDD object representing the result of a computation but it is not computed immediately. Spark performs computations only when an action is called by looking at the whole graph of transformations used to create an execution plan.

- RDDs provide explicit support for data sharing among computations
- Users can also persist selected RDDs in a memory for rapid reuse.
- RDDs are fault tolerant meaning that they are automatically recovered from failure.
- Spark uses the “lineage” approach for recovery process. Each RDD tracks the graph of transformations that was used to build it and reruns those operations on base data to reconstruct any lost partitions.

## 2.6 Related Work

Intrusion Detection Systems have become popular in the cyber security field. In [12] the authors proposed a framework in which they have employed a well-known feature selection algorithm to select the important features in the DARPA’s KDD’99 dataset and then they used classification based intrusion detection method for fast and efficient detection of intrusions in the massive network traffic. The authors in

[22] have employed five machine learning algorithms, Logistic regression, Support Vector Machines, Random forest, Gradient Boosted Decision trees & Naïve Bayes in Apache Spark for processing and detecting the attack traffic as fast as possible and finally they calculated the individual classifier in terms of training time, predicting time, accuracy, sensitivity and specificity on the KDD'99 dataset.

In [31] Sharma et al. have used the Map Reduce framework of Hadoop and implemented Machine learning algorithms like Naïve Bayes and K-Nearest Neighbor and evaluated their performance using WEKA concluded that MapReduce platform is faster than WEKA. In [1] they used network flow data and created semantic links between the suspicious flows forming a probabilistic semantic link network (SLN) to detect known attacks. In [7] they have implemented a streaming-based threat detection system which analyzes the highly intensive network traffic data in real-time using streaming based clustering to detect abnormal attacks.

In this thesis, we have used the Apache Spark framework and the flow based intrusion detection approach to detect the real-time attacks with the help of Spark streaming features. We have also utilized Minimum Redundancy Maximum Relevance (mRMR) feature algorithm to select the important features from the KDD'99 datasets. We also created a K- Nearest Neighbor classifier and trained it and used the Spark streaming to test testing data set and evaluated the classifier.

# Chapter 3

## Methodology

In this chapter, we will discuss in depth the different components that we used to classify whether a packet is a benign or an attack. We start with an overview of the architecture that we have proposed in Apache Spark Ecosystem in Section 3.1. Then we discuss the individual component of the architecture, such as Snort in Section 3.2. Subsequently we describeways to collect the Network tcpdump information using the packet analyzer tool such as tshark and Wireshark in Section 3.3.

We describe the dataset that we use in Section 3.5 and the small introduction about Spark Streaming which can be helpful in performing data mining tasks in Section 3.8. Then we discuss how the K-NN classifier works and the methodology to calculate the similarities in Section 3.6. In Section 3.7 we discuss the Map Reduce paradigm in Apache Spark and benefit of discretized streaming in Section 3.8.1, and we discuss the AWK and Netcat utilities in Section 3.11, which can help us send the data to a particular port. Finally, in Section 3.10, we discuss in depth the Hadoop ecosystem and how it distributes its job to the worker nodes and how it calculates the performance metrics of the individual jobs and tasks.

To implement our system, we used the UMBC High Performance Computing Maya cluster. The Maya cluster is a 324 node, 38 GPU, 38 Intel Phi coprocessor with over 8 TB of main memory. The Maya clusters contain several types of nodes that fall into four main categories:



- **Management Node:** they are reserved for administration of clusters and not available to users.
- **User nodes:** Users work on these nodes directly.
- **Compute nodes:** These nodes are where the majority of computing on the cluster takes place.
- **Development nodes:** These are special compute nodes which are dedicated to running code that is under development.

The UMBC Maya Cluster is an on demand clustering computing environment where a user has the flexibility to specify the number of clusters that he/she may require to perform computations. A brief overview of Maya Cluster can be seen in the below Figure 3.1

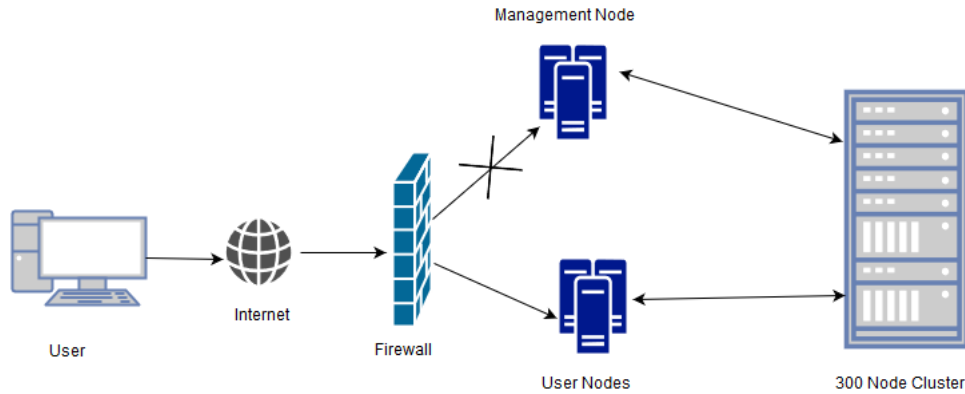


Figure 3.1: Architecture of Maya Cluster

### 3.1 Overview of the Approach

Figure 3.2 shows the overall approach of the system. The tcpdump file with packet information is given to the snort to generate the alerts and the tcpdump file is also given to the flow generator which generates flows from the packet information in the network tcpdump file. Then, the generated flows from the network tcpdump file and the alerts from Snort are merged to generate the Labelled Network Flow. The Labelled Flow is then passed to the Data preprocessing techniques like filling the

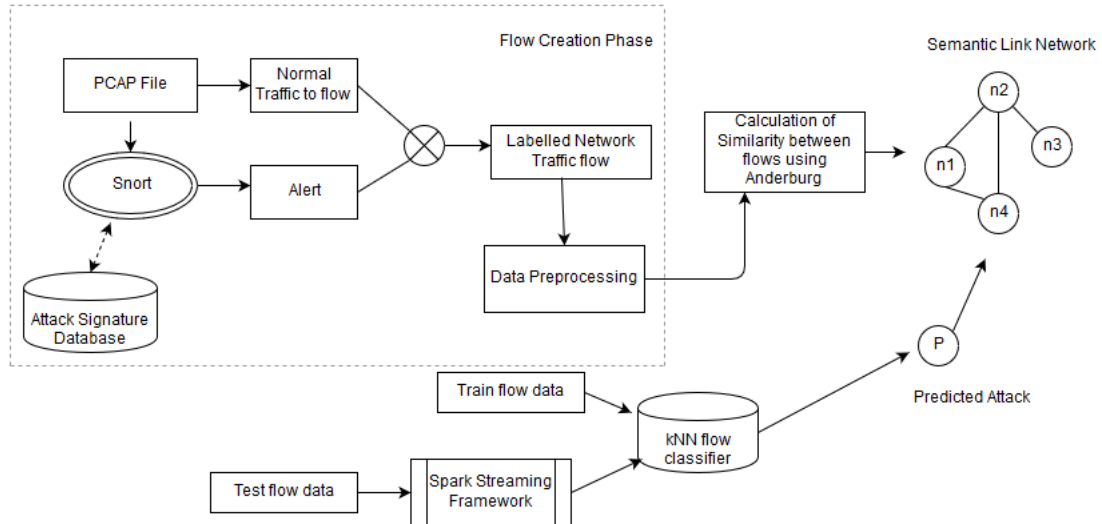


Figure 3.2: Detailed Approach

missing values and performing normalization. Once the preprocessing step is done then by using the Anderberg Similarity Method a Sematic Link network (SLN) is created, which helps us find the similarity and classify an incoming network activity as an attack or a normal activity. We have created a K-NN classifier to classify a packet as a benign or attacks by training the classifier with the previously captured tcpdump file. Finally, to test the classifier we have used the spark streaming framework to test the classifier by sending the data to particular port which is then captured by the classifier and predicts the label of the test data sets.

### 3.2 Snort Architecture

Snort is one of the useful software for security network. Snort can be installed on various platforms of operating systems such as Windows, Linux, etc. Snort has a real time alerting the traffic data network and analyzes capability. The alert can be sent to syslog or a separated ‘alert’ files. Snort is logically divided into various components[6]. These components work in a flow to detect a particular attack and generate output in required format. The components of Snort are packet decoder, preprocessors, a detection engine, logging and alerting system and output modules.

According to Figure 3.3 Snort modules perform the following tasks:

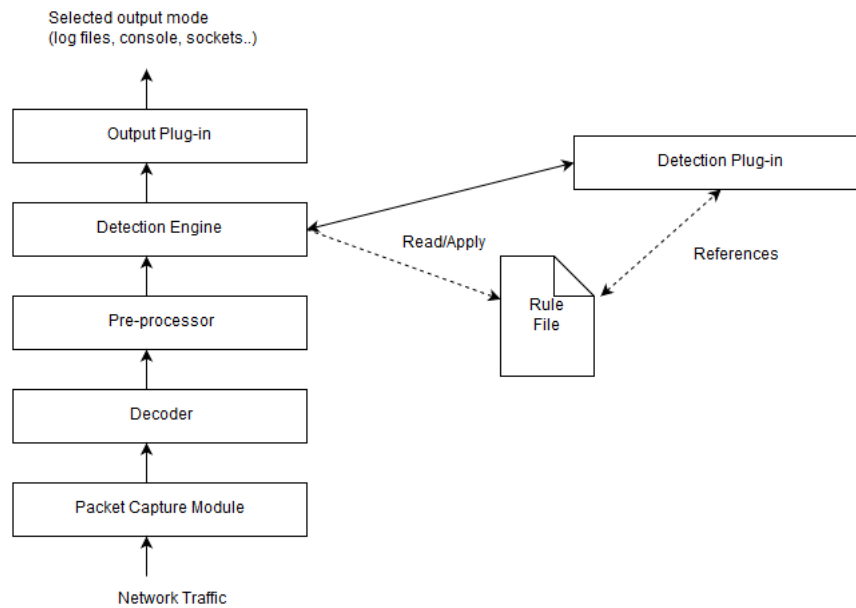


Figure 3.3: Snort Architecture

1. **Packet Capture Module:** This module is built on packet programming library libpcap, which provides implementation independent access to the packet capture facility provided by the operating system, to provide a high-level interface to capture packets.
2. **Decoder:** This module divides the captured packets into a various data structure and identifies the links to be checked in the next module, such as suspicious connection attempts to some TCP/UDP ports, or too many packets sent in a short period.
3. **Preprocessors:** SNORT's preprocessors fall into two categories. They can be used to either examine packets for suspicious activity or modify packets so that the next module can properly interpret them. The other preprocessors are responsible for categorizing traffic so that the next module can accurately match signatures. These preprocessors defeat attacks that attempt to evade Snort's detection engine by manipulating traffic patterns.
4. **Detection Engine:** This module uses the detection plug-ins and matches the packets against the rules loaded into memory during SNORT initialization.
5. **Detection Plug-ins:** The detection plug-in definition is in the rules files.

They are used to identify patterns.

6. **Rules Files:** These are text files containing a list of rules with a known syntax. The syntax includes protocols, addresses, and some other important data.
7. **Output Plug-ins :** This module formats the notifications (alerts, logs) for the user to access them in many ways (databases, console, and external files)[2].

Snort utilizes existing rules, which are patterns of known attacks for searching and matching the network traffic data. If any abnormal pattern is detected it generates an alert. The structure of Snort rules consists of two logical parts

1. Rule Header
2. Rule Option

The Rule Header contains the following field: action, protocol, source address, source port, direction, destination address, and destination port.  
 action field in a snort rule has 3 properties: alert, log and pass.  
 protocol field acts as criteria where to detect network traffic data, which include IP, TCP, UDP, and ICMP.



Figure 3.4: Structure of Snort IDS Rule  
[21]

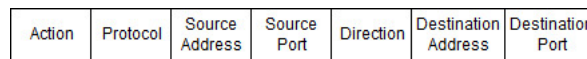


Figure 3.5: Structure of Snort IDS Rule Header  
[21]

The rule options of Snort consist of two parts: a keyword and an argument (defined inside parenthesis and separated by a semicolon). The keyword options are

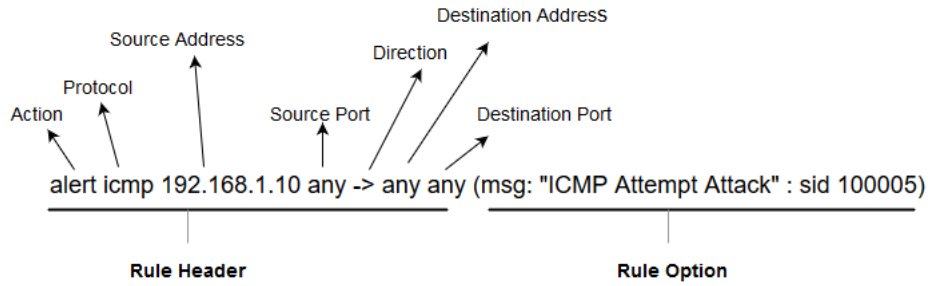


Figure 3.6: Example of Snort IDS Rule  
[21]

separated from the argument by a colon. Examples of keyword options are `msg`, `ttl`, `tos`, and `icode`[21].

An example of Snort IDS rule is shown in Figure 3.6. If it matches the network traffic data of an ICMP protocol field, such as source address, source port, destination address, and destination port, for example, 192.168.1.10, any, any, any, respectively, an alert is generated that outputs the message ICMP Attempt Attack with the signature ID 100005[21].

### 3.2.1 Snort Alert Generation

As we know from the previous section that Snort is an open source IDS to generate alert on the suspicious packets. We have installed Snort 2.9.7.0 on the Ubuntu 16.04.1 LTS to generate the alert for a particular tcpdump file using the following snort command:

```
snort -l ./log -b -c /etc/snort/snort.conf -r /home/muthu/outside.tcpdump
```

where

`-l logdir:` Sets the output logging directory to log-dir. All plain text alerts and packet logs go into this directory. If this option is not specified, the default logging directory is set to `/var/log/snort`.

`-b:` Log packets in a tcpdump formatted file.

*-c config-file:* Use the following Snort configuration file.

*-r tcpdump-file:* Read the tcpdump-formatted file. This will cause Snort to read and process the file fed to it.

Once the above command runs successfully in the log directory we can find the alert file.

```
[**] [1:553:7] POLICY FTP anonymous login attempt [**]  
[Classification: Misc activity] [Priority: 3]  
03/08-08:00:11.738571 206.48.44.18:1054 -> 172.16.112.100:21  
TCP TTL:127 TOS:0x0 ID:39680 IpLen:20 DgmLen:55 DF  
***AP*** Seq: 0x17AD29 Ack: 0x17AEB0 Win: 0x2209 TcpLen: 20  
  
[**] [1:3441:1] FTP PORT bounce attempt [**]  
[Classification: Misc Attack] [Priority: 2]  
03/08-08:00:13.194424 206.48.44.18:1054 -> 172.16.112.100:21  
TCP TTL:127 TOS:0x0 ID:40192 IpLen:20 DgmLen:66 DF  
***AP*** Seq: 0x17AD57 Ack: 0x17AF17 Win: 0x21A2 TcpLen: 20
```

Figure 3.7: Alert Snippet

Figure 3.7 is the short snippet of the alert file.

In the above the command we ran the snort in the default configuration. We can configure the snort to generate the output in csv or pcap with help of output plugin which is available inbuilt in snort. We need to make necessary configuration changes in the `/etc/snort/snort.conf` file. All the predefined snort rules can be found in the `/etc/snort/rules` directory Snort can be configured to listen on the real time packets that are coming to the networks using the following commands:

```
snort -i eth0 -c /etc/snort/snort.conf -l /var/log/snort/
```

where *-i interface:* Sniff packets on interface.

Once the pcap/tcpdump file given to the snort it will generates the alerts and respective packets attack classification label can be extracted.

### 3.3 Capturing via tshark/Wireshark

TShark is a terminal oriented version of Wireshark designed for capturing and display packets when an interactive interface is not available. In order to figure out what is happening in the network we can use tools like tcpdump, tshark or wireshark to sniff the packets in the network and troubleshoot the network problem. We have used the following command to extract the network packets:

```
tshark -i eth0 -c 10 -T fields -e frame.time -e ip.src -e tcp.srcport -e ip.dst -e tcp.dstport -e frame.protocols -E separator=, -E header=y
```

-T option to output data in different formats, this can be very handy when you need a specific format to your analysis.

If we choose fields to the -T option, we must set the -e option at least once, this will tell Tshark which field of information to display, we can use this option multiple times to display more fields Complete list of pcap reference field can be found in the following link: <https://www.wireshark.org/docs/dfref/#section-f>

Wireshark is a GUI version of a packet capture tool in which we can specify fields that we want to extract from the Pcap to CSV.

### 3.4 Flow Generation from Packet Traces

A flow consists of aggregated packets that have a set of common characteristics. IP flows have several characteristics such as source/destination IP, input/output router interfaces, protocol, type of service, packet count, octet count, start/end time, TCP flags, source/destination network mask, input/output interface encapsulation size and IP address of next hop within the network[8].

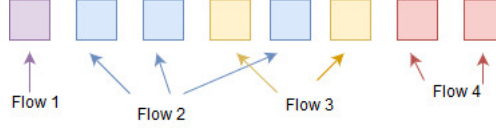


Figure 3.8: Packet aggregation to create flows

Formally a flow is define as  $fl = (I_{src}, I_{dst}, P_{src}, P_{dst}, Prot, Pckts, Octs, T_{start}, T_{end}, Flags)$

where  $I_{src}$  and  $I_{dst}$  are the features that identify source and destination IP addresses,  $P_{src}$  and  $P_{dst}$  are the source and destination port,  $Prot$  is the Protocol type;  $Pckts$  and  $Octs$  give the total number of packet and octets in the data exchange;  $Flags$  are the TCP header flags;  $T_{start}$  and  $T_{end}$  denotes the start and end time of the flow respectively[8]. For a flow, an alert is generated by sending its consisting packets to the Snort IDS for alert generation. Alerts generated for the packets of a flow are propagated to the flow itself and they include the timestamp of the alert, the alert description, and its category which identifies the type of security incident. Since each flow consists of several packets and each set of packets is labeled as either a specific type of alert or benign activity, the flow is labeled by using a majority rule. For example, if a flow consists of three records that generate three alerts, out of which two alert is `buffer_overflow`, the resulting flow is labeled as a `buffer_overflow`.

The KDDcup dataset does not contain the timestamp of the packets so to create the flows we removed all the duplicates packets from our entire datasets which results in the labeled flows containing aggregation of all packets.

### 3.5 Dataset and Source Description

The KDD'99 datasets are widely used for the anomaly detection methods. The original KDD dataset is built on the data presentation in DARPA'98 IDS assessment program. DARPA'98 which contains about 4 gigabytes of compressed binary data, tcpdump data of 7 weeks traffic of network which can be processed into about 5 million connection records, each containing about 100 bytes. A shorter version with two weeks of test data have around 2 million connection records[32]. KDD dataset



consists of connection vectors each of which encloses 41 features and is labeled as either normal or malicious. The simulated attacks fall in one of the following four categories:

1. **Denial of Service Attack (DoS):** A DoS is an attack in which an attacker sends too many requests which over utilize computing resources to handle genuine requests, or denies genuine users entrance to a system.
2. **User to Root Attack (U2R):** It is an attack in which an attacker logs to the system with normal access (perhaps gained by sniffing passwords, a dictionary attack, or social engineering) and is able to exploit some vulnerability to gain root access.
3. **Remote to Local Attack (R2L):** It is an attack in which an attacker who has the ability to send packets to a system over a network who doesn't have an account in the system exploits some vulnerability to gain access to a local system.
4. **Probing attack (PA):** It is a way to gather information about the network of computers with an intension to evade its security controls.

In this thesis, we use the  $k$  – nearest neighbor flow classifier model to classify the packets as benign or attacks and used the panda/spark dataframe.

### 3.6 Nearest Neighbor Classifier

The  $k$ -nearest neighbor algorithm( $k$ -NN) is one kind of distance- based algorithm[39]. It is one of the simple and straight forward lazy learning data mining techniques[17]. The  $k$ -NN is a supervised learning process where classified training samples determine the class of an unknown test samples[39].

To use this technique for us it is necessary to have a training set and a test sample, to know the  $k$  value (how many neighbors are needed to be used in classification) and the mathematical formula to calculate the distance between the instances[9].

The k nearest neighbor classifier is commonly based on the Euclidean or Manhattan distance formula between a test sample and the specified training samples[9].

Euclidean Distance is

$$d(x, y) = \sqrt{\sum_{i=0}^n (x_i - y_i)^2}$$

Manhattan Distance is

$$d(x, y) = \sum_{i=0}^n |x_i - y_i|$$

where  $x_i$  represents the test sample,  $y_i$  is the training data, n is the number of features.

k-nn is based on minimum distance from the test instance to the training samples to determine the k nearest neighbors. Once the value of k is selected, the majority of the k nearest neighbor decides the prediction of the new instances[9].

The general algorithm of computing the k-nearest neighbors is as follows[9]:

- Establish the parameter k that is the number of nearest neighbors.
- Calculate the Euclidean distance between the query instances and all the training samples.
- Sort the distances for all the training samples and determine the nearest neighbor based on the k-th minimum distance.
- Use the majority of nearest neighbors as the prediction value.

In this thesis, we have trained the classifier using training dataset stored on the HDFS (Hadoop Distributed File System). We have used the Spark Streaming framework to stream the test dataset to the classifier and predict test dataset label in streaming.

### 3.6.1 k - Nearest Neighbor Functions

First the available dataset is split into training and testing parts. We have splitted the k – nn classifier into three different functions such as Calculation of Similarity, Finding Nearest Neighbor and Getting the Responses from the Neighbor. The Algorithms below are applied to find the response of the testing labels.

---

**Algorithm 1** Similarity Calculation Using Euclidean Distance

---

Input: Instance1, Instance2, No.of features

Return: Distance score between Instance1 and Instance2

```
1: distance  $\leftarrow$  0
2: for all i in range(Number of Features) do
3:   distance + = pow((instance1[i] – instance2[i]), 2)
4: end for
5: Calculate the square root of distance
6: return distance
```

---

The above algorithm is the simple calculation of distance between the two instances. It takes input as two instances variable and outputs the distance score. Lines 2 – 4 calculate the distance between two instances using Euclidean distance formula.

---

**Algorithm 2** Finding the Neighbours

---

Finding the Neighbours

Return: k training instances similar to testInstance

```
1: Initialize a distances list
2: Initialize a no.of test Instances in length variable
3: for all x in range(len(trainingSet)) do
4:   dist = euclideanDistance(testInstance, trainingSet[x], length)
5:   #Calculating the EuclideanDistance for testInstance with all the trainingSet
6:   distances.append(trainingSet[x], dist) Storing the Euclidean Score
7: end for
8: distances.sort(key=operator.itemgetter(1)) # Sorting the Distance based on the score
9: # Creating the List to Store Neighbor
10: neighbors = []
11: for all x in range(k): do
12:   neighbors.append(distances[x][0])
13: end for
14: return neighbors # Return the k neighbors
```

---

The above algorithm finds the neighbor to the related test instances. It takes input training instances, testing instance and number of nearest neighbor (k) and it will output the k nearest neighbor to the testing instances. Lines 3-7 use Euclidean Distance to calculate the distance between the two instances and store all the distances score and the training instances in a list. Line 8 is simply sorts the distances list based on the score. Lines 10-13 filter only the k nearest neighbor and store it in a neighbor list and return the k neighbor list to the calling functions.

---

**Algorithm 3** Finding the Response

---

Input: Neighbor list containing Responses

Output: Response

```

1: Initialize an empty dictionary classVotes =
2: for all  $x$  in  $\text{range}(\text{len}(\text{neighbors}))$ : do
3:    $\text{response} = \text{neighbors}[x][-1]$ 
4:    $\text{response} = \text{neighbors}[x][-1]$ 
5:   if  $\text{response} = \text{neighbors}[x][-1]$  then
6:      $\text{classVotes}[\text{response}] += 1$ 
7:   else
8:      $\text{classVotes}[\text{response}] = 1$ 
9:   end if
10: end for
11: # Sort the dictionary based on the values
12:  $\text{sortedVotes} = \text{sorted}(\text{classVotes.items}(), \text{key} = \text{operator.itemgetter}(1), \text{reverse} = \text{True})$ 
13: return sortedVotes[0][0] # Return the first response label in the dictionary

```

---

The above algorithm is voting which calculates the number of occurrence of response in the neighbor list. In Line 1 we have initialized an empty dictionary. Lines 2 – 9 calculate the number of occurrence of response and store its values in the dictionary. Line 12 sorts the dictionary on the values hence the maximum occurrence of response will be at 0<sup>th</sup> index of the dictionary and in the Line 13 we are returning the 0<sup>th</sup> index response.

### 3.7 Map Reduce Paradigm

The MapReduce programming paradigm is a technique for data processing tool for Big data, designed by Google in 2003. MapReduce is based on two separate user-

defined primitives: Map and Reduce[23].

Map function reads the raw data in form of key-value ( $\langle \text{key}, \text{value} \rangle$ ) pairs and transforms them into a set of intermediate  $\langle \text{key}, \text{value} \rangle$  pairs, where both the key and value types must be defined by the user. In the next stage, MapReduce merges all the values associated with same intermediate key as a list which is called as Shuffle phase. In the last stage, reduce function takes the grouped output from the maps and aggregates it into a smaller set of pairs. This process can be visualized in the below diagram[23].

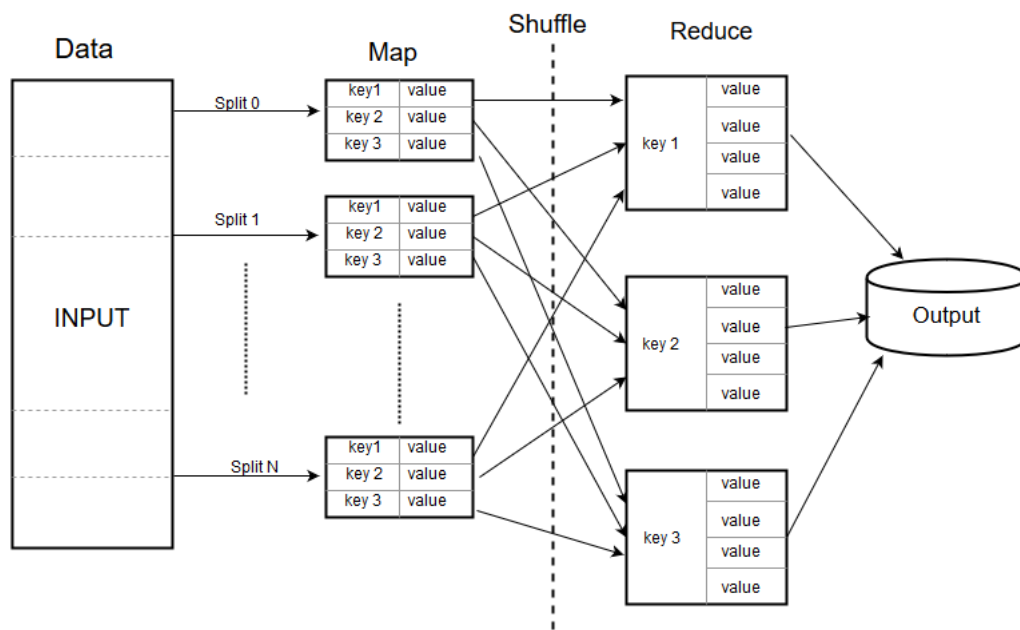


Figure 3.9: Data Flow Overview of MapReduce [23]

Figure 3.9 MapReduce Paradigm is transparent and scalable platform which automatically processes data in a distributed cluster, relieving the user from technical details such as data partitioning, fault tolerance or job communication.

Apache Hadoop is a well-known open source implementation of MapReduce for large scale data processing and storage of data across the cluster. Two main module of the Hadoop is Hadoop Distributed File System (HDFS) and MapReduce. HDFS is a distributed file system which enables the user to distribute the files across the several systems. The files in the HDFS are automatically synced throughout the distribution. Its inability to reuse data through in memory primitives makes the

application of Hadoop unfeasible for many machine learning algorithms.

Apache Spark, is a modified large scale data processing system which was developed to solve the problems of the Hadoop. Spark was introduced as the part of Hadoop ecosystem with all the advantages of Hadoop by using its distributed file system. The Spark framework proposed a set of in-memory computation and analysis with the aim of processing data more rapidly on distributed environments, up to 100 times faster than Hadoop. It provides the developer with an easy interface accessible through Scala, Java and Python and has complete machine learning library built-in.

Spark is based on Resilient Distributed Datasets (RDDs), a special type of data structure used to parallelize the computation across the cluster. These parallel structures let us persist and reuse results, cached in memory. A scalable machine learning library (MLlib) was built on top of spark. The spark MLlib contains a large set of standard learning algorithms and statistic tools which has many important functions for knowledge discovery process such as classification, regression, clustering, optimization or data preprocessing. It provides a high-level API that makes easier for the user to connect multiple machine learning algorithms.

### **3.8 Spark Streaming Framework**

In big data, large amount of data is received in real time which is more valuable at the time of its arrival. For example, in social network we can predict the trending conservation topics in minutes, a search site can create a model to predict which user visit a new page and a service operator may wish to monitor system logs to detect failures in seconds. To develop these low-latency applications, we need a streaming computation models that scale transparently to large clusters, in the same way that batch model like MapReduce simplified processing[40].

There are two main challenges in creating a Streaming application. The first is making the latency (interval granularity) low. Hadoop which is a traditional batch systems falls short because state information is kept in replicated, on-disk storage

systems between jobs. The second challenge is recovering quickly from faults and stragglers. Both problems are inevitable in large clusters, so streaming application must recover from quickly[40].

The above two problems can be solved by a data structure called Resilient Distributed Datasets (RDDs), which keeps data in memory and can recover it without replication by tracking the lineage graph of operations that were used to build it. With RDDs, we can attain sub-second end to end latencies. When a node fails, each node in the cluster works to recompute part of the lost node's RDDs which results in the faster recovery[40].

### **3.8.1 Discretized Streams (D-Streams)**

Discretized streams (D-Streams), a new stream processing model that overcomes above challenges. D-streams is a streaming computation structure that is a series of stateless, deterministic batch computations on small time intervals. For example, we can place the data received every second into an interval, and run a MapReduce operation on each interval to compute certain operation. It avoids the problems with traditional stream processing by structuring computations as a set of short, stateless, deterministic tasks instead of continuous state full operators. It then stores the state in memory across tasks as fault-tolerant data structures (RDDs) that can be recomputed deterministically. It also gives benefits of powerful unification with batch processing.

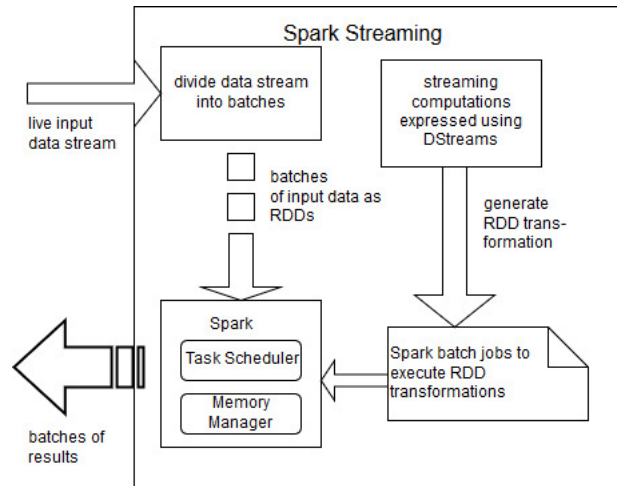


Figure 3.10: High Level Overview of Spark Streaming System

In Figure 3.10 Spark Streaming divides input data streams into batches and stores them in Spark’s memory. It then executes a streaming application by generating Spark jobs to process the batches.

## 3.9 RDD to Data frame

To convert the RDD to Data frame we use the below code

```

a1 = rdd.map(lambda w: w.split(","))
a2= [x for x in a1.toLocalIterator()]
pandadf=pd.DataFrame(a2)

```

First, we capture the RDD from streaming port and by using the map function we split our dataset by comma as a separator. The function `toLocalIterator()` returns an iterator over the dataset. Then by using the panda data frame function we convert our RDD into data frame which can be used for our analysis.

## 3.10 Spark Benchmarking

The Job Tracker and Task Tracker are coming into the picture when we require processing the data set. In Hadoop / Spark system there are five services always



running in the background (called Hadoop daemon services)[29].

Daemon Services of Hadoop are:

1. **NameNode:** The NameNode oversees the health of DataNode and coordinates access to the data stored in the DataNode.
2. **SecondaryNode:** The NameNode which keeps all the filesystem metadata in RAM has no capability to process the metadata on to the disk. So, if NameNode crashes, we may lose everything in RAM and we don't have the backup of the metadata. Therefore, the SecondaryNode contacts the NameNode in an hour and pulls the copy of metadata information out of NameNode. It shuffles and merge this information into a clean file folder and sent to back again to NameNode while keeping a copy for itself. In case of NameNode failure, saved metadata can rebuild it easily.
3. **Job Tracker:** The Job Tracker coordinates the parallel processing of data using MapReduce.
4. **Data Nodes (Executor Nodes):** The Data Nodes are the majority of the machine in Hadoop cluster and are responsible to store the data and process the computation.
5. **Task Tracker:** Task Tracker is the slave to the Job Tracker and resides in the Data Node and performs the necessary computation given by the Job Tracker and once the Task is completed respond back to the Job Tracker.

All the above services interact with each other.

On top of the file systems comes the MapReduce engine, which consists of one JobTracker, to which client applications submit MapReduce jobs. The JobTracker pushes work out to available TaskTracker nodes in the cluster, striving to keep the work as close to the data as possible. With a rack-aware file system, the JobTracker knows which node contains the data, and which other machines are nearby. If the work cannot be hosted on the actual node where the data resides, priority is given

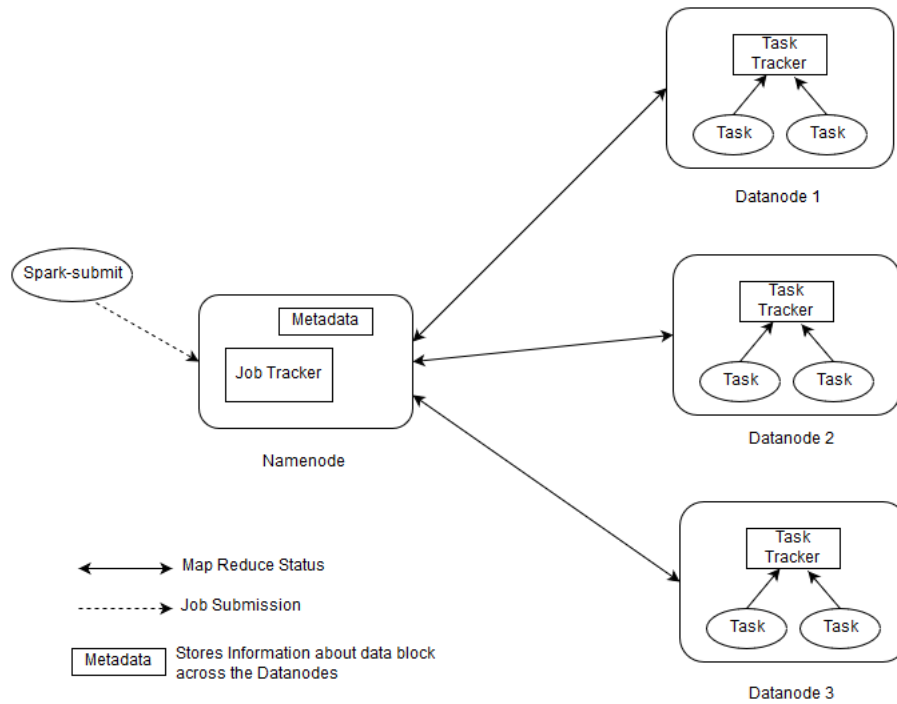


Figure 3.11: Map Reduce Engine  
[29]

to nodes in the same rack. This reduces network traffic on the main backbone network. If a TaskTracker fails or times out, that part of the job is rescheduled. The TaskTracker on each node spawns off a separate Java Virtual Machine process to prevent the TaskTracker itself from failing if the running job crashes the JVM. A heartbeat is sent from the TaskTracker to the JobTracker every few minutes to check its status[29].

The status of the ongoing JobTracker and TaskTracker can be viewed in the Spark WebUI during the job execution on the following link <http://localhost:4040> which gives the status of the individual batch processing time when the job execution is started and completion time and individual task launch and completion time[29].

### 3.11 AWK & Netcat Utility

AWK is a pattern scanning language designed for text processing and typically used as a data extraction and reporting tool. The name is the acronym of Aho, Weinberger, and Kernighan who created it at Bell Labs in the 1970s. It is data-driven scripting language and contains a set of actions taken against streams of textual data. AWK is basically a one-line programming tool. In this thesis, we have used AWK scripts to send data to a particular port by piping into the netcat utility. Netcat utility can open TCP connections, send UDP packets, listen on arbitrary TCP and UDP ports, do port scanning, and deal with both IPv4 and IPv6.

Using AWK and Netcat Utility we have sent the testing dataset to a particular port which acts as a Spark streaming sender whereas on the other side of the Spark Streaming program we have created a Spark streaming socket listener (receiver) which listens to a particular port and captures the data.

# Chapter 4

## Implementation and Evaluation

This chapter mainly focuses on the implementation and evaluation of the prototype system that detects intrusions. To carry out our experimentation we have used the dataset provided by UCI KDD Archive[14]. We have used this dataset to carry out experiments. It is a labeled dataset containing 494021 instances of packet flows. We have applied preprocessing techniques like removing duplicates, removing null values and normalization of our data set with help of python. Once the data set was preprocessed we created a KNN classifier with the help of the training instances and the respective labels. We also analyzed the performance of our system by running it in different number of clusters and we found out that as we increase the number of clusters the time taken to predict attack decreases.

### 4.1 Implementation

The dataset which we got from UCI KDD is a labelled comma separated file. With the help of Spark context, we converted the file into RDD (Resilient Distributed Dataset) which is the datatype which resides in memory for computation. Spark gives us the flexibility to convert RDD into data frames which helps us to perform computation in efficient manner. Once all our dataset is converted into Spark data frame it can be distributed across the worker nodes for computation.

### 4.1.1 Data set description

The dataset which we used in our experiment to access K-NN classifier for Network Intrusion Detection is KDDCup'99 dataset and it is developed by MIT at Lincoln's laboratory. This dataset is derived from the Defense Advanced Research Project Agency (DARPA) packet traces which comprises of variety military network territory simulated intrusions. The KDD dataset is also utilized in the Third International competition that happened on Knowledge Discovery and Data Mining Tools. The goal of this competition was to establish a network detector to find "good" connections and "bad" connections[26].

The entire KDDCup'99 dataset (extract the kddcup.data.gz file [26]) consist of 4,898,431 records in which every record is of 41 features which are detailed in the below table. We utilized only the 10% part (extract the kddcup.data\_10\_percent.gz file[26]) of KDD dataset for the purpose of training and testing. The 10% KDDCup'99 data consist of 494,069 records (each containing 41 features) which are categorized into 4 types of attack. The categories of attack and their distinct types are presented in Table 4.3 below.

| No | Features               | Description   |
|----|------------------------|---|
| 1  | duration               | Duration of the Connection  |
| 2  | protocol type          | Connection protocol (e.g. TCP, UDP, ICMP)   |
| 3  | service                | Destination service   |
| 4  | flag                   | Status flag of the connection   |
| 5  | source bytes           | Bytes sent from source to destination   |
| 6  | destination bytes      | Bytes sent from destination to source   |
| 7  | land                   | 1 if successfully logged in; 0 otherwise  |
| 8  | wrong fragment         | Number of wrong fragment  |
| 9  | urgent                 | Number of urgent packets  |
| 10 | hot                    | Number of “hot” indicator   |
| 11 | failed logins          | Number of failed logins   |
| 12 | Logged in              | 1 if successfully logged in; 0 otherwise  |
| 13 | num_compromised        | Number of “compromised” condition   |
| 14 | root shell             | 1 if root shell is obtained; 0 otherwise  |
| 15 | su_attempted           | 1 if “su root” command attempted; 0 otherwise   |
| 16 | num root               | Number of “root” accesses   |
| 17 | num file creations     | Number of file creation operations  |
| 18 | num shells             | Number of shell prompts   |
| 19 | num access file        | Number of operations on access control files  |
| 20 | num_outbound cmds      | Number of outbound commands in a ftp session  |
| 21 | is hot login           | 1 if login belongs to the “hot” list; 0 otherwise   |
| 22 | is guest login         | 1 if login is the “guest” login; 0 otherwise  |
| 23 | count                  | Number of connections to the same host as the current connection in the past 2 seconds      |
| 24 | srv count              | Number of connections to the same service as the current connection in the past two seconds |
| 25 | serror rate            | % of connections that have ”SYN” errors   |
| 26 | srv serror rate        | % of connections that have ”SYN” errors   |
| 27 | rerror rate            | % of connections that have ”REJ” errors   |
| 28 | srv rerror rate        | % of connections that have ”REJ” errors   |
| 29 | same srv rate          | % of connections to the same service  |
| 30 | diff srv rate          | % of connections to different services  |
| 31 | srv diff host rate     | % of connections to different hosts   |
| 32 | dst host count         | Count of connections have the same destination host   |
| 33 | dst host srv count     | Count of connections have the same destination host and using the same service              |
| 34 | dst host same srv rate | % of connections having the same destination host and using the same service                |
| 35 | dst host diff srv rate | % of different service on the current host  |

Table 4.1: Features list and Description of KDDCup’99 Dataset  
[26]

| No | Features                    | Description   |
|----|-----------------------------|---|
| 36 | dst host same src port rate | % of connections to the current host having the same src port                     |
| 37 | dst host srv diff host rate | % of connections to the same service coming from different host                   |
| 38 | dst host serror rate        | % of connections to the current host that have an S0 error                        |
| 39 | dst host srv serror rate    | % of connections to the current host and specified service that have an S0 error  |
| 40 | dst host rerror rate        | % of connections to the current host that have an RST error                       |
| 41 | dst host srv rerror rate    | % of connections to the current host and specified service that have an RST error |

Table 4.2: Features list and Description of KDDCup'99 Dataset  
[26]

Table 4.3: Attack Classification & data types

|          |  |        |       |  |       |
|----------|--|--------|-------|--|-------|
| [Normal] | Attack Types   | Class  | [U2R] | Attack Types   | Class |
|          | Normal<br>apache2<br>Back<br>land<br>mailbomb<br>neptune<br>pod<br>processtable<br>smurf<br>teardrop<br>udpstorm | Normal |       | buffer overflow<br>loadmodule<br>perl<br>ps<br>rootkit<br>sqlattack<br>xterm   | U2R   |
| [Probe]  | Attack Types   | Class  | [R2L] | Attack Types   | Class |
|          | ipsweep<br>mscan<br>portsweep<br>saint<br>satan<br>nmap  | Probe  |       | ftp write<br>guess passwd<br>sendmail<br>imap<br>multihop<br>named<br>phf<br>snmpgetattack<br>snmpguess<br>warezmaster<br>worm<br>xlock<br>httptunnel<br>xsnoop<br>wazerclient | R2L   |

| score  | Features                 | score  | Features                    |
|--------|--------------------------|--------|-----------------------------|
| 0.9630 | service                  | 0.5531 | logged_in                   |
| 0.9452 | same_srv_rate            | 0.4068 | dst_host_count              |
| 0.9119 | count                    | 0.3708 | dst_host_srv_diff_host_rate |
| 0.8747 | flag                     | 0.2134 | srv_count                   |
| 0.8498 | dst_host_diff_srv_rate   | 0.1999 | srv_diff_host_rate          |
| 0.8226 | dst_host_same_srv_rate   | 0.1546 | dst_host_error_rate         |
| 0.7932 | dst_host_srv_count       | 0.1489 | protocol_type               |
| 0.6735 | dst_host_serror_rate     | 0.1338 | dst_host_srv_error_rate     |
| 0.6554 | serror_rate              | 0.0959 | error_rate                  |
| 0.6302 | dst_host_srv_serror_rate | 0.0783 | hot                         |
| 0.6158 | srv_serror_rate          | 0.0704 | wrong_fragment              |
| 0.6158 | num_access_files         |        |                             |

Table 4.4: Feature Score of Top 25 Attributes

## 4.2 Evaluation

In the KDD'99 dataset we have applied Minimum Redundancy Relevance Feature selection algorithm to calculate which features give us more information about our datasets. After application of the above algorithm we found that the following columns give us the more information about our dataset as shown in Table 4.4. We used the above 25 columns to train our KNN classifier.

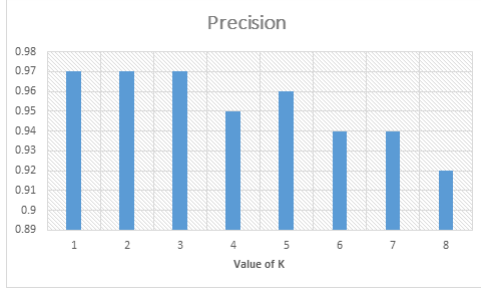
From the Figure 4.1, we can say that  $k = 3$  gives us a better accuracy. Since, Knn assumes that the data is in feature space more exactly the data points are in a metric space. Each training data consists of set of vectors and class label associated with each vector. The number 'k' in the classifier decides how many neighbors influences the classification[36].

Let us take case where  $k = 1$  which is the simplest scenario[36].

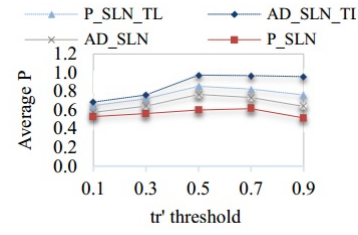
Let  $x$  be the point to be labeled. Find the point closest to  $x$ . Let it be  $y$ . Now nearest neighbor rule asks to assign the label of  $y$  to  $x$ . This seems too simplistic and sometimes even counter intuitive. This reasoning holds only when the number of data points is not very large.

If the number of data points is very large, then there is a very high chance that label of  $x$  and  $y$  are same. An example might help – Let's say you have a (potentially) biased coin. If one tosses it for 1 million times and one got head 900,000 times, most

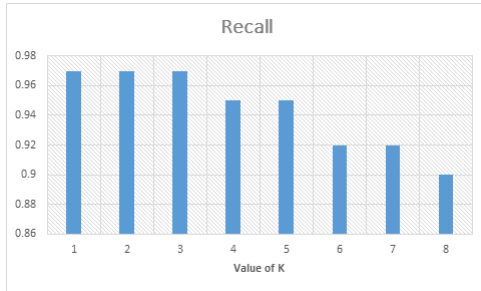




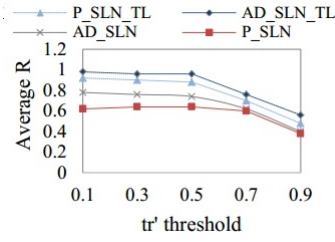
(a) k-NN Precision



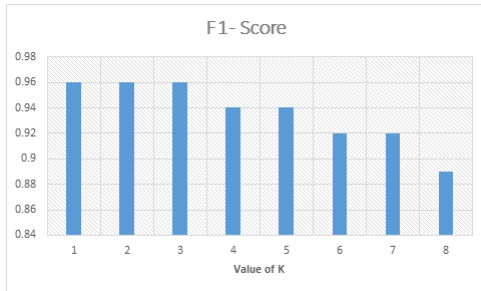
(b) Decision Tree Precision



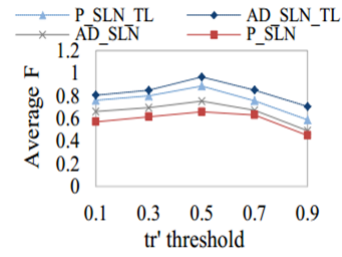
(c) k-NN Recall



(d) Decision Tree Recall



(e) k-NN F1 Score



(f) Decision Tree F1 Score

Figure 4.1: Evaluation Metrics of Classifier

likely the next call will be head. We can use a similar argument here.

Let us take another case where  $k=2$

This is a straightforward extension of 1NN. Basically, what we do is that we try to find the  $k$  nearest neighbor and do a majority voting. Typically,  $k$  is odd when the number of classes is 2. Let's say  $k = 5$  and there are 3 instances of C1 and 2 instances of C2. In this case, KNN says that new point must be labeled as C1 as it forms the majority. We follow a similar argument when there are multiple classes[36].

From the above three figures, we can conclude that feature selection is one of the critical and frequently used techniques in data preprocessing. It can reduce the number of features, remove irrelevant features and bring the immediate effects for intrusion detection. The performance of our algorithm can deteriorate if we select the wrong features while training our classifier[36].

The selected 25 features and the experimental results are listed in the figures. It shows that the performance of our method is good on original KDD. Experimental results also show that  $k=3$  gives us the accuracy of 97% which can be more effective in detecting intrusions with low false positive.

## 4.3 Spark WebUI

Spark Web UI helps us to view the performance and behaviour of our Spark applications. The behaviour of our spark application can be seen in the Spark web UI at <http://:4040>.

Here is a screen shot of the web UI after running the word count job. Under the "Jobs" tab, you see a list of jobs that have been scheduled or run, which in this example is the word count collect job. The Jobs table displays job, stage, and task progress[25].

Under the Stages tab, you can see the details for stages. Below is the stages page for the word count job, Stage 0 is named after the last RDD in the stage pipeline, and Stage 1 is named after the action in Figure 4.3.

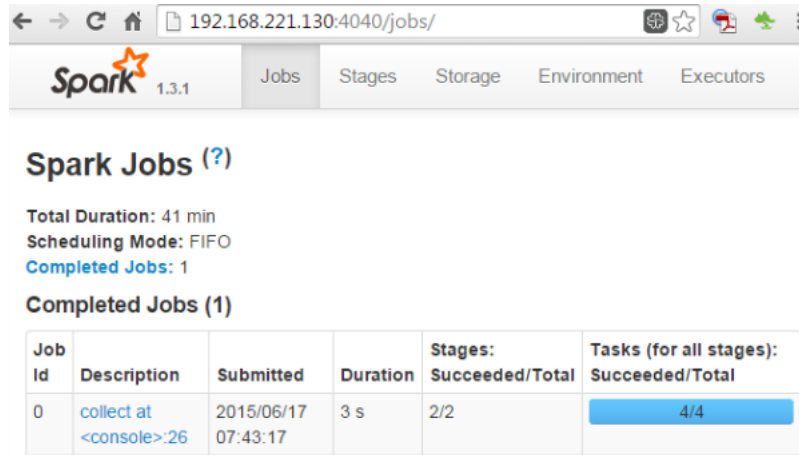


Figure 4.2: Spark Jobs [25]

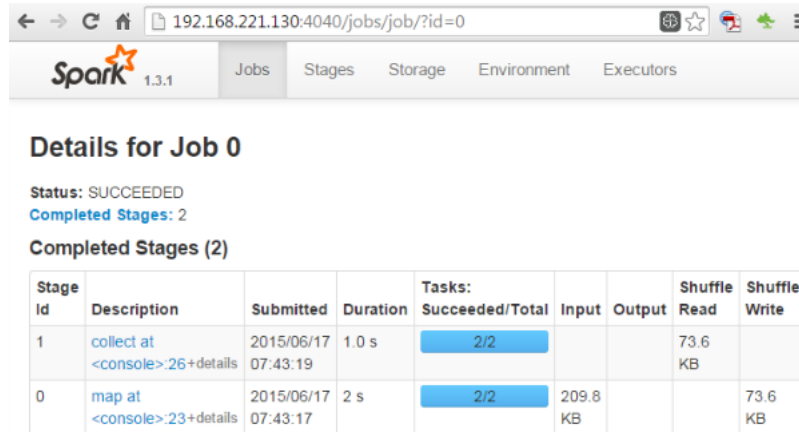


Figure 4.3: Spark Stages [25]

In Figure 4.4 We can view RDDs in the Storage tab.

In the Executors tab, we can see processing and storage for each executor. We can look at the thread call stack by clicking on the thread dump link[25].

In Spark streaming application, user is much interested in the rate at which data is being received and the processing time of each batch. The streaming tab in the UI makes it easy to see the current metrics as well as the trends over that past 1000 batches[35].

The first line (marked as [A]) shows the current status of the streaming application – in this example, the application has been running for almost 40 minutes at a 1-second batch interval. The timeline of Input Rate (marked as [B]) shows that the streaming app has been receiving data at a rate of about 49 events/second across

| RDD Name               | Storage Level                     | Cached Partitions | Fraction Cached | Size in Memory | Size in Tachyon | Size on Disk |
|------------------------|-----------------------------------|-------------------|-----------------|----------------|-----------------|--------------|
| /user/user01/alice.txt | Memory Deserialized 1x Replicated | 2                 | 100%            | 448.3 KB       | 0.0 B           | 0.0 B        |

Figure 4.4: Spark RDD  
[25]

| Executor ID | Address         | RDD Blocks | Memory Used         | Disk Used | Active Tasks | Failed Tasks | Complete Tasks | Total Tasks | Task Time | Input    | Shuffle Read | Shuffle Write | Thread Dump                 |
|-------------|-----------------|------------|---------------------|-----------|--------------|--------------|----------------|-------------|-----------|----------|--------------|---------------|-----------------------------|
| <driver>    | localhost:56501 | 2          | 448.3 KB / 246.0 MB | 0.0 B     | 0            | 0            | 4              | 4           | 4.6 s     | 209.8 KB | 0.0 B        | 73.6 KB       | <a href="#">Thread Dump</a> |

Figure 4.5: Spark Executors  
[25]

all its sources[35].

Further down in the page (marked as [D] in figure 4.6), the timeline for Processing Time shows that these batches have been processed within 20 ms on average. Having a shorter processing time comparing to the batch interval (1s in this example) means that the Scheduling Delay (defined as the time a batch waits for previous batches to complete, and marked as [E] in figure 4.6) is mostly zero because the batches are processed as fast as they are created[35].

## 4.4 Spark Benchmarking

We have calculated the Program Execution Time by collecting the JSON logs and by changing the number of nodes in the cluster. Below are the details of the execution times and we found that the if we increase the number of nodes in the cluster, as expected the program execution time decreases.

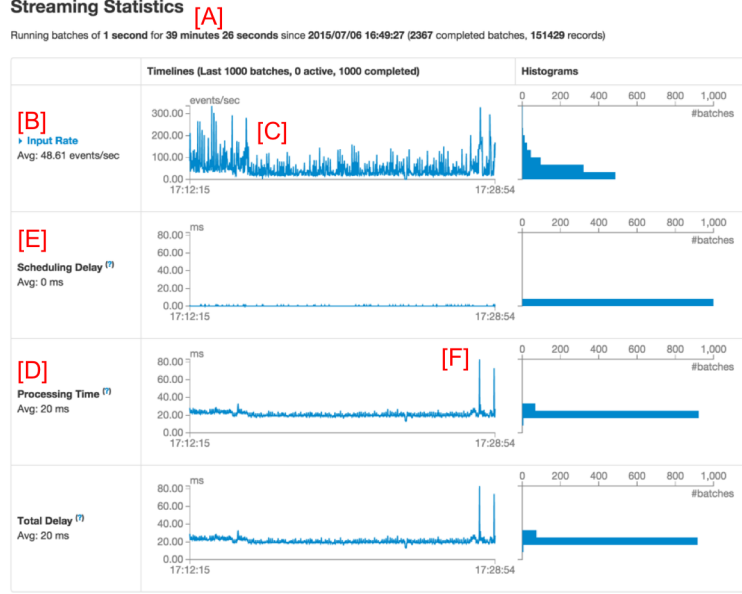


Figure 4.6: Spark Streaming  
[35]

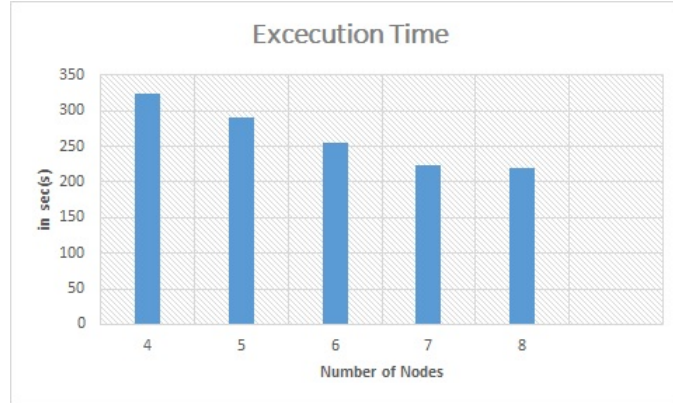


Figure 4.7: Average execution time in seconds with varying number of computing nodes

From the above figure 4.7 we can say that, as we increases the number of worker nodes in the spark environment the time taken to process our dataset decreases gradually. In our experiment, we have changed the number of nodes from 4, 5, 6,7 and 8 and we can observe that as we increase the number of worker/executor node, the average execution time decreases.

From the experimental results, we can clearly say that implementing a classifier with all the features with highest information gain can help us to create a classifier with highest accuracy. For k-nn classifier selecting the appropriate value of k

results with highest accuracy. We have also observed that in Spark environment detecting intrusion using the MapReduce Paradigm is much faster as compared to the traditional method.

## 4.5 Spark JSON log results

When a job is submitted by spark using a spark-submit command to the driver node, the driver node splits the jobs to individual stages and distributes it's across all the executor nodes for the parallel execution. When a stage is received by an executor node, it tries to divide the stages into individual tasks for computations.

In our spark benchmarking evaluation, we calculated the performance of individual batch on various number of nodes.

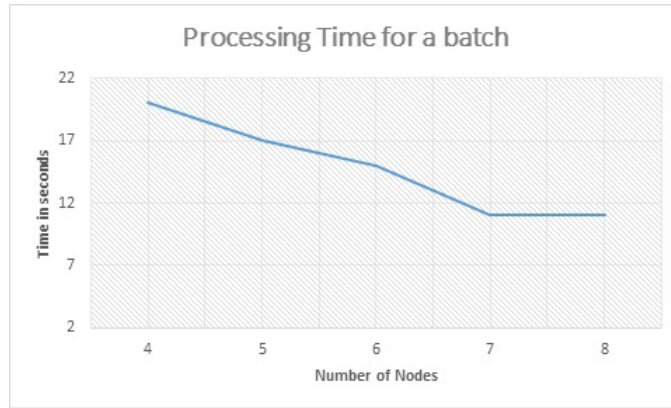


Figure 4.8: Processing time for a batch

In our implementation, it took around 20 seconds to execute a batch. As we increase the number of nodes the processing time for the individual batch decreases gradually.

When the spark streaming receives more batch in specific interval, it queues the additional batches. The time taken to execute the queued batches is called the Elapsed time. In the similar way, we calculated the elapsed time by varying the number of nodes in our environment as shown in the figure 4.9.

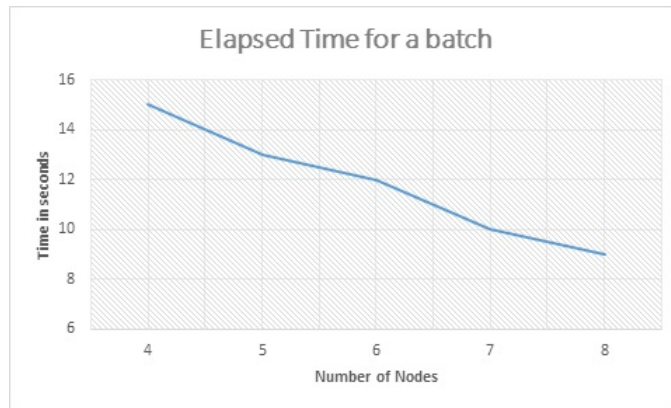


Figure 4.9: Elapsed Time for a batch

We also calculated the throughput of our systems which means number of batched processed per min which tells us about the scalability of our system which is shown in the below figure 4.10.

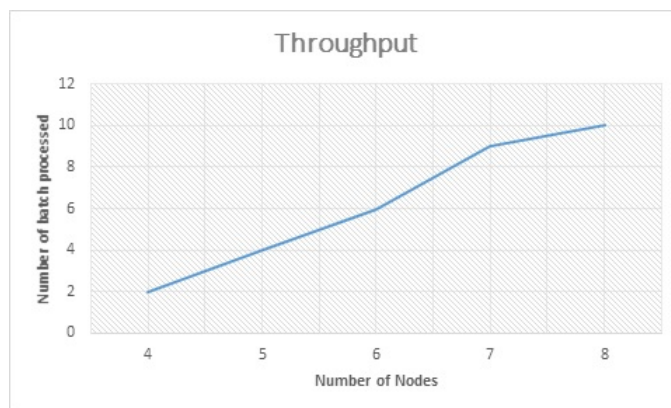


Figure 4.10: Throughput

As we increase the number of nodes in our spark environment, the number of batch processed per minute increases gradually.

# Chapter 5

## Conclusion & Future Works

### 5.1 Conclusions

The goal of an Intrusion Detection System is to detect attacks before they result in any damage to the organization. However, traditional IDSs are unable to handle large amount of traffic originating in short span of time. To handle the above problem, we have used a flow based IDS approach and combined it with Apache Spark ecosystem which provides parallelism through clustering to detect intrusions in a short time period.

In this research, we addressed the problem of handling large amounts of data by using a cluster environment which distributes the work across the clusters. We have also created the KNN classifier using Apache Spark and calculated the Euclidean distances from the test dataset with all the training data. Testing was performed using a test dataset to identify attacks by predicting the label of the test dataset using Spark Streaming.

We evaluated our system by varying the number of clusters and calculated the time taken for our system to predict attacks. Overall, we observed that as we increase the number of clusters in our environment, the time taken to predict attack decreases. We also evaluated the KNN classifier by changing the nearest neighbor value K in the KNN algorithm and we observed that in general, the error increases as we add more neighbors.



Spark Streaming in Apache spark helped us to stream the real time data in network. Internally the system creates batches and distributes the batches across the cluster for faster execution. We also evaluated the Spark Streaming by capturing its log in the json file and calculated the processing time of individual jobs. When a job is created in the Spark Driver node it is assigned an executor node to execute job in the executor memory. Processing time is the time taken to process all the jobs.

## **5.2 Future Works**

In this research, we used the KNN classifier to predict whether a packet is an attack or normal. We plan to use other efficient classifiers such as Logistics Regression, Support Vector Machine and J48 algorithms for the classification of packets. In the future, we plan to implement the system with the above different classifiers, calculate its efficiency and perform a comparison based on the results. We can perform tuning to the classifier with the highest efficiency for prediction of attacks.

In the current implementation we used Spark Streaming framework and the Socket Stream to stream the data in the network, whereas in the future we plan to use different Streaming frameworks, such as Kafka and Storm which are compatible with Apache Spark for real-time streaming of data and perform a comparison.

# Bibliography

- [1] Ahmed Aleroud and George Karabatis. “Context infusion in semantic link networks to detect cyber-attacks: a flow-based detection approach”. In: *Semantic Computing (ICSC), 2014 IEEE International Conference on*. IEEE. 2014, pp. 175–182.
- [2] Hussein Alnabulsi, Md Rafiqul Islam, and Quazi Mamun. “Detecting SQL injection attacks using SNORT IDS”. In: *Computer Science and Engineering (APWC on CSE), 2014 Asia-Pacific World Congress on*. IEEE. 2014, pp. 1–7.
- [3] Paul Barford and David Plonka. “Characteristics of network traffic flow anomalies”. In: *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*. ACM. 2001, pp. 69–73.
- [4] Stephen D Bay, Dennis Kibler, Michael J Pazzani, and Padhraic Smyth. “The UCI KDD archive of large data sets for data mining research and experimentation”. In: *ACM SIGKDD Explorations Newsletter 2.2* (2000), pp. 81–85.
- [5] Milan Čermák, Tomáš Jirsík, and Martin Laštovička. “Real-time analysis of NetFlow data for generating network traffic statistics using Apache Spark”. In: *Network Operations and Management Symposium (NOMS), 2016 IEEE/IFIP*. IEEE. 2016, pp. 1019–1020.
- [6] Youksamay Chanthakoummane, Saiyan Saiyod, Nunnapus Benjamas, and Natawat Khamphakdee. “Improving intrusion detection on snort rules for botnets detection”. In: *Information Science and Applications (ICISA) 2016*. Springer, 2016, pp. 765–779.
- [7] Zhijiang Chen, Hanlin Zhang, William G Hatcher, James Nguyen, and Wei Yu. “A streaming-based network monitoring and threat detection system”. In: *Software Engineering Research, Management and Applications (SERA), 2016 IEEE 14th International Conference on*. IEEE. 2016, pp. 31–37.
- [8] Tao Ding, Ahmed AlErroud, and George Karabatis. “Multi-granular aggregation of network flows for security analysis”. In: *Intelligence and Security Informatics (ISI), 2015 IEEE International Conference on*. IEEE. 2015, pp. 173–175.
- [9] Elia Georgiana Dragomir. “Air quality index prediction using K-nearest neighbor technique”. In: *Bulletin of PG University of Ploiesti, Series Mathematics, Informatics, Physics, LXII 1.2010* (2010), pp. 103–108.
- [10] Solane Duque and Mohd NIzam bin Omar. “Using data mining algorithms for developing a model for intrusion detection system (IDS)”. In: *Procedia Computer Science* 61 (2015), pp. 46–51.

- [11] Ziad Elkhadir, Khalid Chougali, and Mohammed Benattou. "Intrusion Detection System Using PCA and Kernel PCA Methods". In: *Proceedings of the Mediterranean Conference on Information & Communication Technologies 2015*. Springer. 2016, pp. 489–497.
- [12] Govind P Gupta and Manish Kulariya. "A Framework for Fast and Efficient Cyber Security Network Intrusion Detection Using Apache Spark". In: *Procedia Computer Science* 93 (2016), pp. 824–831.
- [13] Jiawei Han, Jian Pei, and Micheline Kamber. *Data mining: concepts and techniques*. Elsevier, 2011.
- [14] Seth Hettich and SD Bay. "The UCI KDD Archive [<http://kdd.ics.uci.edu>]. Irvine, CA: University of California". In: *Department of Information and Computer Science* 152 (1999).
- [15] Chang-Jung Hsieh and Ting-Yuan Chan. "Detection DDoS attacks based on neural-network using Apache Spark". In: *Applied System Innovation (ICASI), 2016 International Conference on*. IEEE. 2016, pp. 1–4.
- [16] Wei Huang, Lingkui Meng, Dongying Zhang, and Wen Zhang. "In-Memory Parallel Processing of Massive Remotely Sensed Data Using an Apache Spark on Hadoop YARN Model". In: *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* 10.1 (2017), pp. 3–19.
- [17] M Akhil Jabbar, BL Deekshatulu, and Priti Chandra. "Heart disease classification using nearest neighbor classifier with feature subset selection". In: *Anale. Seria Informatica* 11 (2013).
- [18] Omar Al-Jarrah and Ahmad Arafat. "Network intrusion detection system using neural network classification of attack behavior". In: *Journal of Advances in Information Technology Vol 6.1* (2015).
- [19] Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. *Learning spark: lightning-fast big data analysis*. " O'Reilly Media, Inc.", 2015.
- [20] Ahmad M Karimi, Quamar Niyaz, Weiqing Sun, Ahmad Y Javaid, and Vijay K Devabhaktuni. "Distributed network traffic feature extraction for a real-time IDS". In: *Electro Information Technology (EIT), 2016 IEEE International Conference on*. IEEE. 2016, pp. 0522–0526.
- [21] Nattawat Khamphakdee, Nunnapus Benjamas, and Saiyan Saiyod. "Improving Intrusion Detection System Based on Snort Rules for Network Probe Attacks Detection with Association Rules Technique of Data Mining". In: *Journal of ICT Research and Applications* 8.3 (2015), pp. 234–250.
- [22] Manish Kulariya, Priyanka Saraf, Raushan Ranjan, and Govind P Gupta. "Performance analysis of network intrusion detection schemes using Apache Spark". In: *Communication and Signal Processing (ICCSP), 2016 International Conference on*. IEEE. 2016, pp. 1973–1977.
- [23] Jesus Maillo, Sergio Ramirez, Isaac Triguero, and Francisco Herrera. "kNN-IS: An Iterative Spark-based design of the k-Nearest Neighbors classifier for big data". In: *Knowledge-Based Systems* 117 (2017), pp. 3–15.
- [24] Ilias Mavridis and Helen Karatza. "Performance evaluation of cloud-based log file analysis with Apache Hadoop and Apache Spark". In: *Journal of Systems and Software* 125 (2017), pp. 133–151.

- [25] Carol McDonald. *Getting Started with the Spark Web UI*. Jun 30, 2015. URL: <https://mapr.com/blog/getting-started-spark-web-ui/>.
- [26] Praneeth Nskh, M Naveen Varma, and Roshan Ramakrishna Naik. “Principle component analysis based intrusion detection system using support vector machine”. In: *Recent Trends in Electronics, Information & Communication Technology (RTEICT), IEEE International Conference on*. IEEE. 2016, pp. 1344–1350.
- [27] Cisco Publications. *Network as a Security Sensor Threat Defense with Full NetFlow*. URL: <http://www.cisco.com/c/en/us/solutions/collateral/enterprise-networks/enterprise-network-security/white-paper-c11-736595.pdf>.
- [28] Jürgen Quittek, Tanja Zseby, Benoit Claise, and Sebastian Zander. *Requirements for IP flow information export (IPFIX)*. Tech. rep. 2004.
- [29] Dinesh Rajput. *JobTracker and TaskTracker Design*. URL: <http://www.dineshonjava.com/2014/11/jobtracker-and-tasktracker-design.html#.WDEJelze0g5>.
- [30] K Salah and A Kahtani. “Improving snort performance under linux”. In: *IET communications* 3.12 (2009), pp. 1883–1895.
- [31] Rachana Sharma, Priyanka Sharma, Preeti Mishra, and Emmanuel S Pilli. “Towards MapReduce based classification approaches for Intrusion Detection”. In: *Cloud System and Big Data Engineering (Confluence), 2016 6th International Conference*. IEEE. 2016, pp. 361–367.
- [32] Preeti Singh and Amrish Tiwari. “An Efficient Approach for Intrusion Detection in Reduced Features of KDD99 Using ID3 and Classification with K-NGA”. In: *Advances in Computing and Communication Engineering (ICACCE), 2015 Second International Conference on*. IEEE. 2015, pp. 445–452.
- [33] Gaurav Somani, Manoj Singh Gaur, Dheeraj Sanghi, and Mauro Conti. “DDoS attacks in cloud computing: collateral damage to non-targets”. In: *Computer Networks* 109 (2016), pp. 157–171.
- [34] Daniël van der Steeg, Rick Hofstede, Anna Sperotto, and Aiko Pras. “Real-time DDoS attack detection for Cisco IOS using NetFlow”. In: *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*. IEEE. 2015, pp. 972–977.
- [35] Shixiong Zhu Tathagata Das and Andrew Or. *New Visualizations for Understanding Apache Spark Streaming Applications*. Jul 8, 2015. URL: <https://databricks.com/blog/2015/07/08/new-visualizations-for-understanding-apache-spark-streaming-applications.html>.
- [36] Saravanan Thirumuruganathan. *A Detailed Introduction to K-Nearest Neighbor (KNN) Algorithm*. March 2010. URL: <https://saravananthirumuruganathan.wordpress.com/2010/05/17/a-detailed-introduction-to-k-nearest-neighbor-knn-algorithm/>.
- [37] Fabricio Voznika and Leonardo Viana. *Data Mining Classification*. 2007.

- [38] Xindong Wu, Vipin Kumar, J Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J McLachlan, Angus Ng, Bing Liu, S Yu Philip, et al. “Top 10 algorithms in data mining”. In: *Knowledge and information systems* 14.1 (2008), pp. 1–37.
- [39] Moawia Elfaki Yahia and Badria Abaker Ibrahim. “K-nearest neighbor and C4. 5 algorithms as data mining methods: advantages and difficulties”. In: *Computer Systems and Applications* (2003), p. 103.
- [40] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. “Discretized streams: Fault-tolerant streaming computation at scale”. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM. 2013, pp. 423–438.
- [41] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. “Apache Spark: a unified engine for big data processing”. In: *Communications of the ACM* 59.11 (2016), pp. 56–65.

