POLITECNICO DI TORINO

Dipartimento di Elettronica e delle Telecomunicazioni
Corso di Laurea Magistrale in ”ICT for Smart Societies”

Tesi di Laurea Magistrale

Big Data Analytics for Network Traffic Monitoring and Analysis

Candidate:
Francesca Soro

Contents

List of Figures

List of Tables

Chapter 1

Introduction and problem description

1.1 Overview

Network Traffic Monitoring and Analysis (NTMA) has taken a paramount role to understand the functioning of the Internet, especially to get a broader and clearer visibility of unexpected events. One of the major challenges faced by large scale NTMA applications is the processing and analysis of large amounts of heterogeneous and fast network monitoring data. Network monitoring data usually comes in the form of **high-speed streams**, which need to be rapidly and continuously processed and analyzed. A variety of methodologies and tools have been devised to passively monitor network traffic, extracting large amounts of data from live networks. What is needed is a **flexible data processing system** able to analyze and extract useful insights from such rich and heterogeneous sources of data, offering the possibility to apply complex Machine Learning (ML) and Data Mining (DM) techniques. The introduction of Big Data processing led to a new era in the design and development of large-scale data processing systems. This new breed of tools and platforms are mostly dissimilar, have different requirements, and are conceived to be used in specific situations for specific needs. Each Big Data practitioner is forced to muddle through the wide range of options available, and NTMA is not an exception. A similar problem arises in the case of Big Data analytics through ML and DM based techniques. Despite the existence of ML libraries for Big Data Analysis Frameworks (BDAFs), there is a big gap to the application of such techniques for NTMA when considering fast online streams and massive offline datasets.

1.2 Goal of the thesis

Objectives

In such context, the final objective of this work is to test the performance of several supervised machine learning algorithms in terms of execution time and predictive capacity, to detect five types of network anomalies (DDoS, MPTP, Ping flood, Netscan-ACK and Netscan-UDP) on real network traffic. The tests are executed on an Hadoop-based distributed ecosystem using the Spark framework, and in particular Spark ML as a machine learning library.
The exploited packet traces are collected everyday for 15 minutes from the WIDE backbone. They are provided as an open repository by the MAWILab project, together with a label for each flow that states whether it is anomalous, suspicious, notice or benign, which is used to extract a reliable ground truth to train each model. The MAWI traces are further processed to extract relevant features (i.e. packet volume, total number of packets, empirical distribution of time to live, window size, fraction of flagged packets, fraction of IPv4 and IPv6 packets, etc.) to be given as an input to the tested classification algorithms. The instantiated models are: Decision Tree, Random Forest, Support Vector Machines, Naive Bayes and Neural Networks. To each of them 10-fold cross-validation is applied to reduce overfitting. Together with the test of basic classifiers with a full-feature setup, the case study involves the test of two feature selection procedures as well as the presentation and basic development of a sample of ensamble learning model.
Results are expressed in terms of relative execution time (computed with respect to the model having shortest execution time), and area under the Receiver Operating Characteristic curve. By this means we are allowed to draw conclusions on stregnths and weaknesses of all the models and choose which ones fit best our case study.

1.3 Thesis organization

These are the topics treated in each chapter.

Chapter 2: State of the art and contributions

This chapter discusses the most important studies from the literature that can be useful for the thesis. It delas with some of the traditional approaches used for NTMA, it lists some of the existing Big-Data frameworks for batch and stream analysis, focusing on how they can be correctly benchmarked and how are they currently applied to network security problems.

Chapter 3: The Big-DAMA project

In this section we discuss the project scientific challenges and aims, taking into account the work organization, the system configuration and the applied frameworks specifications in the batch and in the stream case, heading towards the future use of a single framework for handling the entire data stream warehouse system. The chapter also includes a basic description of the cluster environment involved in the case study, and its future improvements.

Chapter 4: The Machine Learning models

In this chapter the case study steps are illustrated. It presents the exploited dataset, the ground truth model and implementation details, of the chosen algorithms and metodologies.

Chapter 5: Conclusion and future work

The case study results are presented together with conclusions and considerations based on them. Possible future works are mentioned.

Chapter 2

State of the art and contributions

Before proceeding with the state of the art analysis, we should remark that what we call Big-Data cuts loose from the mere concept of very large amount of data; the following definitions, three among the many we find in literature, tackle the problem from aspects other than the data volume:

- The ***attributive definition*** characterizes big data through the ”4Vs”, i.e. volume, variety, velocity and value, referring to such technologies as ”designed to economically extract value from very large volumes of a wide variety of data, by enabling high-velocity capture, discovery, and/or analysis” [85];
- The ***comparative definition*** refers to big data as those datasets ”whose size is beyond the capability of typical database software tools to capture, store, manage and analyze” [86], providing an intentionally subjective definition lacking of precise metrics, since it supposes that the concept will adapt to the advancements of technology.
- The ***architectural definition*** states that ”Big data is where the data volume, acquisition velocity, or data representation limits the ability to perform effective analysis using traditional relational approaches or requires the use of significant horizontal scaling for efficient processing” [82].

In this chapter, we structure the state of the art analysis in three different sections: (i) Machine Learning, Data Mining and Big Data Analytics, (ii) Network Anomaly Detection and Security, and finally (iii) Benchmarks for Big Data Analysis Solutions.

For the sake of brevity we do not reference all the systems, solutions and algorithms we discuss next, but we mention them to give the reader an idea of the range of the problem. The following chapter will go deeper in covering the application of big data analytics for NTMA and will target the ones specifically used in the realized use case.

2.1 Machine Learning, Data Mining and Big Data Analytics

The fields of Machine Learning (ML) and Data Mining (DM) have been studied for more than 50 years, and today there is a comprehensive list of options [46]. Popular **supervised ML algorithms** include Locally Weighted Linear Regression, Naive Bayes, Gaussian Discriminative Analysis, Logistic Regression, Neural Networks, Principle Components Analysis, Independent Component Analysis, Expectation Maximization, Support Vector Machine, Decision Trees, and many more. **Clustering algorithms (unsupervised analysis)** [58] include partition-based clustering (K-means), density-based clustering (DBSCAN), hierarchical-clustering, spectral clustering [60], distribution-based clustering, etc... Two particularly promising unsupervised and supervised algorithms which exemplify the NTMA needs in terms of complex traffic analysis are Sub-Space Clustering [59] and Adaptive Trees [61, 62] respectively. The former because of its capabilities for exploring big dimensional data, even when working with Big Data, by allowing a dimensionality reduction of the problem through **feature transformation** (i.e. reducing the dataset dimension by combining together more features) and **feature selection**; the latter because of its direct applications in stream, supervised-based analysis. In the same context, work such as [63] shows potential and yet unexplored directions to perform clustering with stream data, which is also highly appealing for NTMA applications, such as those in our work on autonomous network security [44]. We can find today an increasing number of MapReduce-based implementations of the most important ML algorithms. Indeed, today there is a reasonably high number of ML libraries available, including **Spark ML** [78], **Apache Mahout** [74] and **MLlib** [75] (NoSQL), **MADlib** (SQL-based), as well as frameworks implementing machine learning and data mining algorithms (e.g., Weka, MOA, SAMOA, etc.). To describe some of them, Mahout is a scalable machine learning library with a relatively long history, containing implementations of algorithms in the areas such as classification,

clustering, recommendation systems, etc., whereas MADlib provides machine learning in SQL, including classification, regression, clustering, association rule mining, descriptive statistics, etc. Another field that is worth mentioning is the one of **Ensemble Learning**, whose goal is to combine the predictions of several base estimators to improve robustness over a single estimator [79]. We recognize three main Ensemble Learning techniques: **Bagging (Bootstrap Aggregating)**, **Boosting** and **Stacking**; the first one, by averaging several first level models, aims at reducing the variance of the final model while keeping the bias fixed, the second one, on the other hand, targets bias reduction, while the third one seeks for reaching a higher predictive capability of the final model originated by a given combination the basic ones.

2.2 Network Anomaly Detection and Security

The usual approach applied to network monitoring and anomaly detection is the so-called **knowledge-based** one, addressing all the techniques that require to be tuned by an external agent, usually a human expert, to detect attacks. Among such techniques we identify the **signature-based** detection and the **novelty detection**: the former is highly effective in the detection of attacks having a known set of characteristics, referred to as signature, while the latter is able to identify traffic events deviating from a baseline profile defined for normal traffic [44]. Anomaly detection provides the basis to detect novel incidents such as network failures, misconfigurations or network attacks that cannot be discovered by knowledge-based approaches. While many approaches exist today for applying a wide variety of different machine learning techniques to network intrusion detection [49, 50, 51, 52, 53, 44], we are still not able to cope with todays attack techniques. There are many reasons for this. While machine learning performs extremely well in other fields (such as SPAM detection), network traffic introduces much larger challenges to the detection task [54]. Reasons are the high dynamics of network traffic, the high costs of misclassifications and active attackers that adapt to detection techniques. The specific nature of network traffic confines the applicability of machine learning techniques. It requires very well focused problem statements and carefully tuned learning approaches. Also, while existing approaches make use of the whole range of available machine learning techniques, the extremely important steps of feature generation and feature selection is underrepresented in cur-

rent literature [55]. Feature generation and feature selection are essential to provide the input and significantly determine the detection performance [56]. Classical machine learning approaches are often computationally expensive. Data rates and the overall amount of network traffic is permanently increasing. In addition, network traffic analysis demands more and more detailed analysis results and online detection of anomalies requires high response times, i.e. fast processing and analysis, in order to react to severe situations and reduce the damage during critical incidents. With these characteristics, network traffic data analysis is a perfect candidate to profit from the benefits of big data analysis techniques [57, 64].

2.3 Benchmarks for Big Data Analysis Solutions

In order to facilitate the comparison of performances between existing solutions, testing benchmarks must be defined. The rise of new Big Data platforms and technologies calls for the definition of a standardized set of benchmarks, similar to the ones defined in the traditional relational databases field by TPC (Transaction Processing Performance Council), whose mission is to ”define transaction processing and database benchmarks and to disseminate objective, verifiable TPC performance data to the industry”[83]. According to [84], Benchmarking Big Data Analysis Frameworks (BDAFs) is not a trivial task, and it comes to be uniquely challenging for four main reasons: system complexity, use case diversity, data scale and rapid system evolution. The current research on big data benchmarks divides the field into two categories: **component level** benchmarks and **system level** benchmarks, the former targeting performance comparisons for stand-alone components, the latter addressing end-to-end testing. How to know if a certain BDAF is fast and accurate enough to tackle the specific needs of NTMA applications, and how to select the best BDAFs to perform complex analytics on the monitoring traffic? The literature offers many specifically tailored solutions: Gridmix [72], the Hive Benchmark [71], HiBench [69] and the Berkeley Big Data Benchmark [45], to name a few of them. The latter is based on a comparative study between MapReduce and parallel DBMSs frameworks conducted in [68], and consists of a carefully designed query workload, aiming to test the capability of data processing systems for online analytical processing queries. Other BDAFs (Spark, SimSQL, GraphLab and Giraph) have been benchmarked in terms of

completion-time when running complex, hierarchical Machine Learning models on very large datasets [47]. To the best of our knowledge, none of the available benchmarks addresses specifically NTMA applications. Another important aspect when working with Internet measurements is the need to integrate raw data with other data sources (e.g., for geo-location, routing and topology, classification, etc.), which is not addressed by those benchmarks.

Chapter 3

The Big-DAMA project

3.1 Big Data Analysis Frameworks

The introduction of Big Data processing led to a new era in the design and development of large-scale data processing systems [33]. This new breed of tools and platforms are mostly dissimilar, have different requirements, and are conceived to be used in specific situations for specific needs. Each Big Data practitioner is forced to muddle through the wide range of options available, and NTMA is not an exception. A basic yet complete taxonomy of Big Data Analysis Frameworks includes traditional **Database Management Systems** (DBMS) and extended **Data Stream Management Systems** (DSMSs), **noSQL** systems (e.g., all the MapReduce-based systems), and **Graph-oriented** systems. Generally speaking, the distinguishing characteristics whom one should refer to for choosing the most suitable framework for his needs are: **data volume, structure, speed, continuity, number of data sources, scalability requirements and application distribution**. Figure 3.1 below offers a first insight of time requirements according to the nature, structure and field of application of the analyzed data.

While the majority of the systems target the offline analysis of static data, some proposal consider the problem of analyzing data coming in the form of online streams, characterized by being continuous, rapid, time-varying and less predictable, and hence not compliant with a traditional DBMS framework not supporting continuous queries [76]. Data Stream Management Systems (DSMS) such as Gigascope, supporting an SQL-like language [34], and Borealis, exploiting Stream Query Algebra [35], are

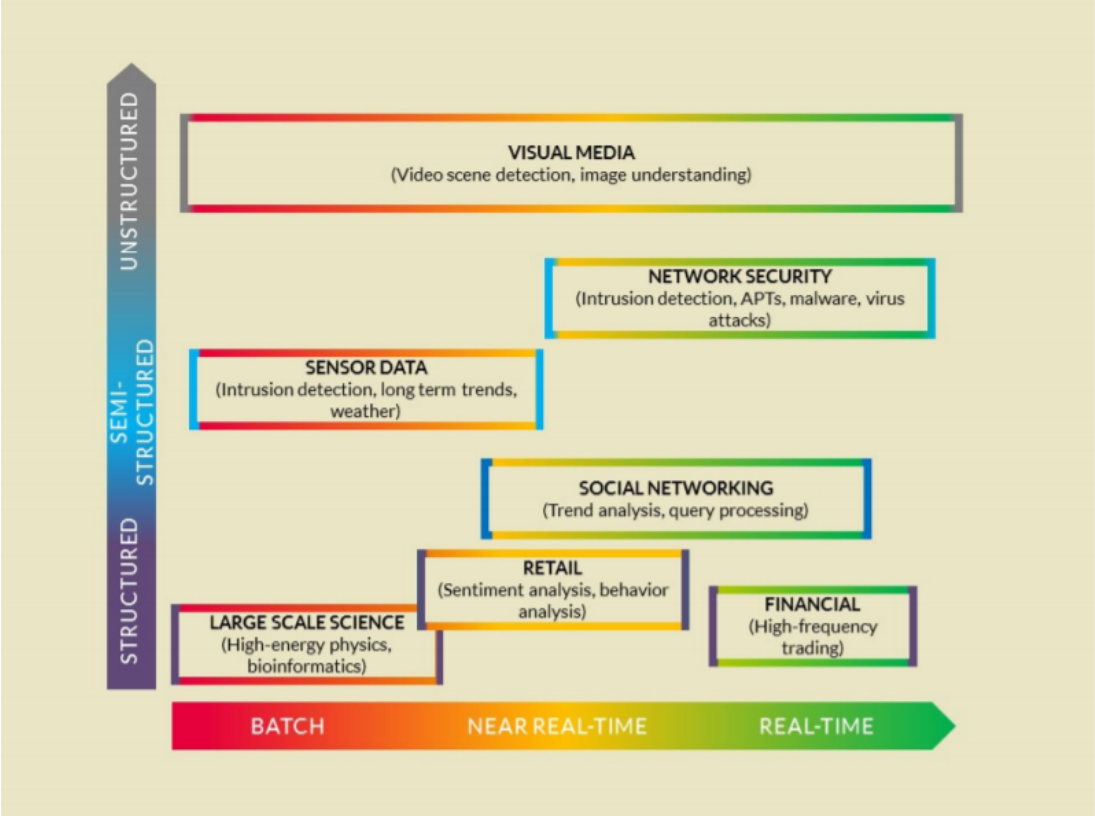


Figure 3.1: Time requirements and data structure for most common fields. From [87]

specifically designed for monitoring applications, they support continuous online processing of data coming from heterogeneous sources also containing incomplete, stale or intentionally omitted information, but they cannot run offline analytics over static data. The Data Stream Warehousing (DSW) paradigm provides the means to handle both types of online and offline processing requirements within a single system, loading continuous data in a streaming fashion, and warehousing them for a long time period (i.e. years or decades) [77]. DataCell and DataDepot are examples of this paradigm [36]. NoSQL systems such as e.g. MapReduce [40] have also rapidly evolved, supporting the analysis of unstructured data over distributed cluster architectures. Apache Hadoop [41] and Spark [42] are very popular implementations of MapReduce systems. These are based on offline processing rather than stream processing. There has been some promising recent work on enabling real-time and incre-

mental analytics in MapReduce-based systems, such as Spark Streaming [43], Incoop [37], Muppet [38] and SCALLA [39], but these remains unexploited in the NTMA domain. The offer of solutions available is overwhelming; more examples include Storm, Samza, Flink (NoSQL); Hawq, Hive, Greenplum (SQL-oriented); Giraph, GraphLab, Pregel (graph-oriented), as well as well known DBMSs commercial solutions such as Teradata, Dataupia, Vertica and Oracle Exadata (just to name a few of them).

3.2 Scientific Challenge

As highlighted in the above section, the application of Big Data Analysis Frameworks for NTMA tasks requires certain system capabilities:

- **scalability**: the framework must offer, possibly inexpensively, storage and processing capabilities to scale with huge amounts of data generated by in-network traffic monitors and collectors;
- **real-time processing**: the system must be able to ingest and process data in real-time fashion;
- **historical data processing**: the system must enable the analysis of historical data;
- **traffic data analysis tools**: embedding libraries or plugins specifically tailored to analyze traffic data.

The currently exploited systems are not able to cope with these requirements and present severe limitations when it comes to the application of Big Data Analytics to NTMA applications. Traditional SQL-like databases, for instance, are inadequate for the continuous real-time analysis of data. As we mentioned before, Data Stream Warehouses have been introduced to extend traditional database systems with continuous data ingest and processing, and they have been in some cases proven to be able to outperform in terms of processing speed Big Data technologies [32]. However, differently from Big Data technologies, they can not scale with the huge amounts of traffic data generated by nowadays networks. This represents a huge limitation to their applicability to NTMA purposes. Big Data Analysis Frameworks based on the MapReduce

paradigm have been recently started to be adopted for NTMA applications [65]. Considering the specific context of network monitoring, some solutions to adapt Hadoop to process traffic data have been proposed [66]. However, the main drawback of Big Data technologies in general is their inherent offline processing, which is not suitable for real-time traffic analysis, highly relevant in NTMA tasks. The only approach that leverages Hadoop for rolling traffic analysis is described in [67]. As we mentioned before, there have also been some Big Data Analysis Frameworks for online data processing, but none of these has been applied to the NTMA domain.

Moreover, Big Data Analytics results on NTMA applications are seldom available, especially when considering online, stream based traffic analysis. This creates a major gap between the developments of Big Data Analytics and Analysis Frameworks and the development of NTMA systems capable of analyzing huge amounts of network traffic. In addition, while there is a vast number of Big Data Frameworks, the offer is so big and difficult to track that makes it very challenging to determine which one to choose for the purpose of NTMA. Secondly, considering the theory of Big Data Analytics applied to the NTMA domain, most of the proposed Machine Learning frameworks and libraries do not scale well in fast big data scenarios, as their main target is offline data analytics.

It should also be noticed that, while some supervised and unsupervised learning algorithms are already available for Big Data Analytics, we are at a very early stage development and there is big room for improvement. The most notable example is explorative data analysis through clustering. Available algorithms are either too simple (e.g., no techniques such as Sub-Space clustering are available, most of the work done on traditional k-means), or too tailored to specific domains not related to traffic analysis. Clustering data streams is still an open problem, and a very useful one for unsupervised Anomaly Detection and Network Security. Similar unsolved problems such as unsupervised feature selection become more challenging as well, due to scalability issues in the Big Data scenario. Also when considering supervised approaches, we do not have today much evidence on how supervised online learning approaches perform with big stream-based traffic. There are also limitations in the analysis and comparison of different machine learning and data mining techniques running in Big Data Frameworks, because available benchmarks are very ad-hoc and tailored to specific types of systems (e.g., tailored for MapReduce-like frameworks). The Big-DAMA project will

advance many of this open issues, as we will futher discuss in the next Section.

3.3 Project objectives

Big-DAMA is a research project founded by Vienna Science and Technology Fund (WWTF), having as core partners AIT, Politecnico di Torino and TU Wien. One of the main questions it poses itself as research in the NTMA domain is straightforward: if one wants to tackle NTMA applications with (near) real-time requirements in current massive traffic scenario, which would be the best system one should use to the task? In addition, if the main target is to perform complex data analytics on top of this massive traffic, considering both supervised and unsupervised ML approaches, how should it be done? Which are the best ML algorithms for doing so? The Big-DAMA project will accelerate NTMA practitioners and researchers understanding of the many new tools and techniques that have emerged for Big Data Analytics in recent years. Big-DAMA will particularly identify and test the most suitable BDAFs and available Big Data Analytics implementations of ML and DM algorithms for tackling the problems of Anomaly Detection and Network Security in an increasingly complex network scenario. The Big-DAMA project proposes three main objectives:

- Explore, conceive and test scalable **online and offline Machine Learning and Data Mining-based techniques** to monitor and characterize extremely fast and/or extremely large network traffic datasets;
- Conceive novel frameworks for Big Data Analytics tailored to Anomaly Detection and Network Security, evaluating and selecting the best BDAFs matching NTMA needs. Such frameworks would target traffic stream data processing (online processing) and massive offline data processing (offline processing);
- Conceive a **novel benchmark for BDAFs** and Big Data Analytics tailored for NTMA applications, particularly focusing on stream analysis algorithms and online processing tasks.

The starting point of Big-DAMA is DBStream [32], a Data Stream Warehouse developed between FTW and the University of Waterloo, and benchmarked against new big data analysis platforms such as Spark, in collaboration with Politecnico di Torino, showing very promising results in the field of NTMA [32].

Specific research questions

While the proposed objectives of the Big-DAMA project might look a-priori more practical than theoretical, we will address several fascinating topics related to the application of ML and DM techniques in the Big Data domain, for the specific purposes of network traffic exploration and extraction of information. So besides the aforementioned objectives, the Big-DAMA project will provide answers to the following research questions:

- How to automatically construct proper data representations (i.e., computing good features or descriptors) given a certain ML algorithm and a huge dataset of unlabeled data?
- How to perform unsupervised feature selection?
- How to cluster big and fast evolving data streams? This is still an open problem, and result would be highly useful when thinking on unsupervised NTMA;
- Which supervised learning approaches can be applied in an online manner with big amounts of streaming data?
- Which are the statistical implications of divide and conquer algorithms when dealing with Big Data?
- Which are the impacts of traffic sampling and aggregation in the results of Big Data Analytics for NTMA?
- Is Big Data a curse when dealing with Anomaly Detection and Network Security, or it can be useful to improve traditional approaches?

We expect the outcomes of the project to have a direct impact on NTMA domain, being particularly beneficial for large network operators and network monitoring technology vendors, but also to be applicable with few adaptations to domains facing the same kind of challenges, such as online monitoring of M2M or IoT devices, and various smart cities scenarios (i.e. smart transport, smart grids or home automation applications).
The final platform is depicted in figure 3.2: the system is a **lambda architecture** recalling, as we said, the data stream warehouse paradigm. From the analytical point of

view what is developed is an **Automatic Network Anomaly Detection and Diagnosis system** to characterize symptomatic and diagnostic features in network traffic.

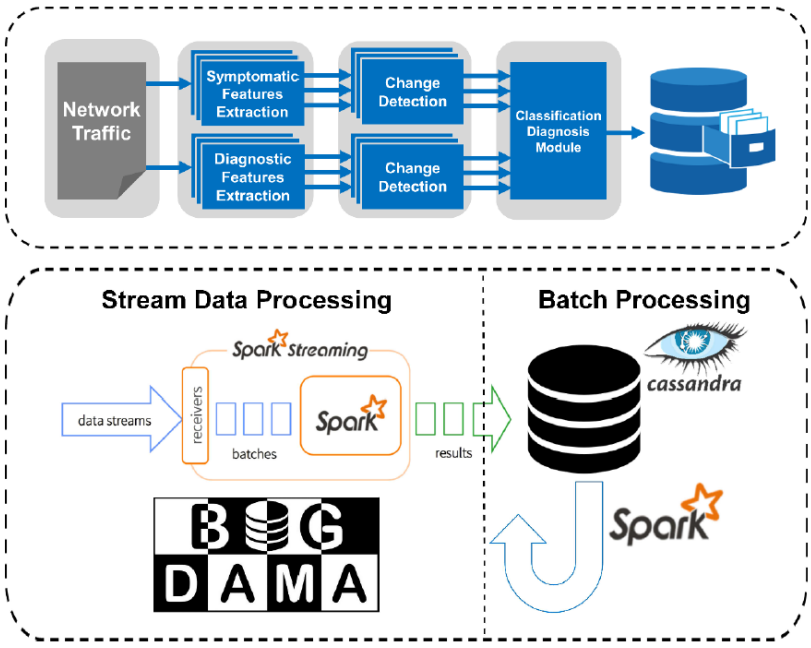


Figure 3.2: Big-DAMA system overview.

3.4 Big-DAMA cluster

The Big-DAMA cluster is **Hadoop based**, and managed through the **Cloudera Manager** interface [88]. The Hadoop framework is designed to work with very large datasets, distributed across a set of machines. In our specific case, the ecosystem includes the **Hadoop Distributed File System** (HDFS), the **YARN resource manager**, **Spark 2** as a computing system, on top of which we run **Hive**, **Oozie** and **Zookeeper** services (see figure 3.3). To better characterize the case study, we will proceed with a more detailed description of each service.

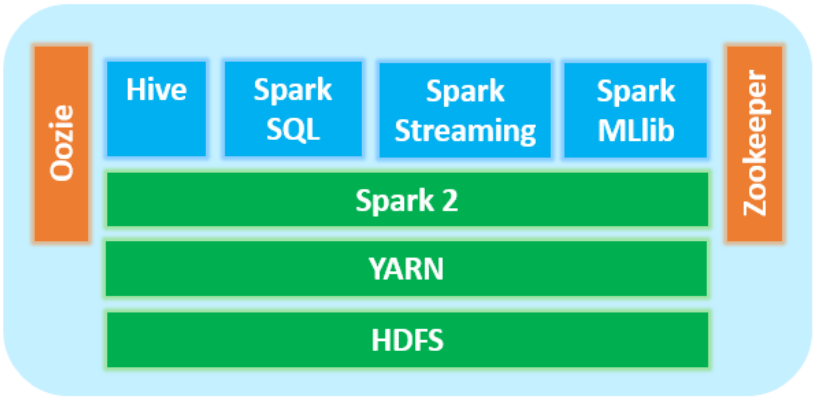


Figure 3.3: Service layers on the cluster.

3.4.1 HDFS

HDFS has been specifically designed to manage large files (in the order of gigabytes to terabytes) that cannot be handled through a traditional file system [89]; each file is splitted and scattered all over several machines, guaranteeing a **certain amount of replicated copies** all over the cluster, so that if a single copy gets corrupted, it will not affect the overall workflow. This file system has a master-slave architecture including a **NameNode (master)** and several **DataNodes (slave, one per node in the cluster)**; the former manages the namespaces and regulates the access to the clients handling operations such opening, closing, renaming files and directories, and determining the mapping of blocks to DataNodes, while the latter perform block creation, deletion and replication when instructed by the NameNode. Given the large amount of operations which the NameNode should provide, it may act as a single point of failure; that is the reason why also a SecondaryNameNode is deployed, to act as a backup and store periodic checkpoints for the main one. Figure 3.4 below shows the overall architecture structure.

3.4.2 YARN

YARN stands for Yet Another Resource Negotiator. As described in [90], its fundamental idea is to split up the functionalities of **resource management** and **job scheduling and monitoring**. Also in this case, the service is deployed as a master-slave archi-

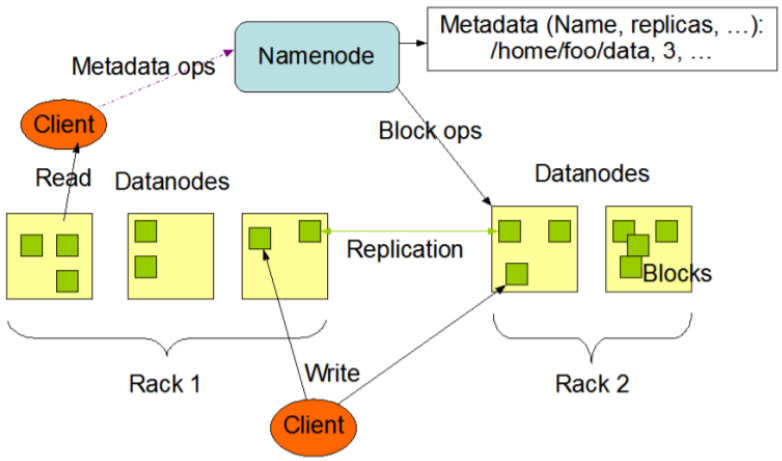


Figure 3.4: HDFS architecture. From [89]

tecture (as in figure 3.5), including a single **ResourceManager** which is aware of all the resources on the cluster, and one **NodeManager** per worker machine, responsible for Container handling and reporting usage data to the ResourceManager.

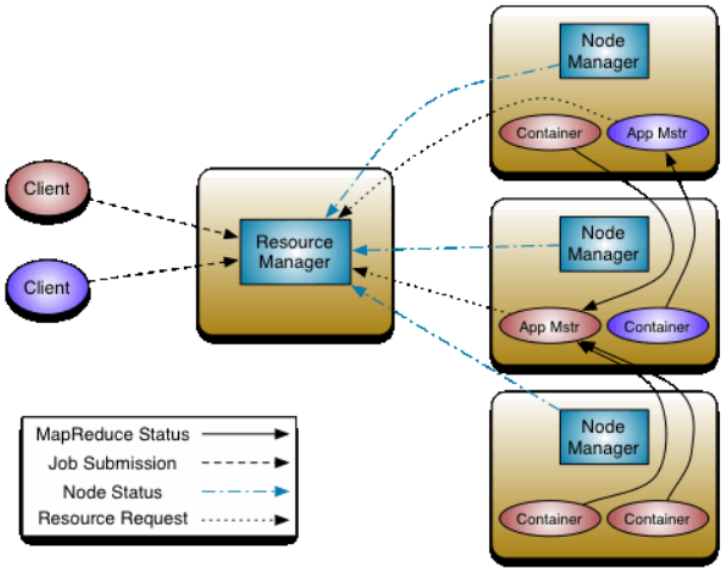


Figure 3.5: YARN architecture. From [90]

The concept of **Container** is really important to correctly configure the resources

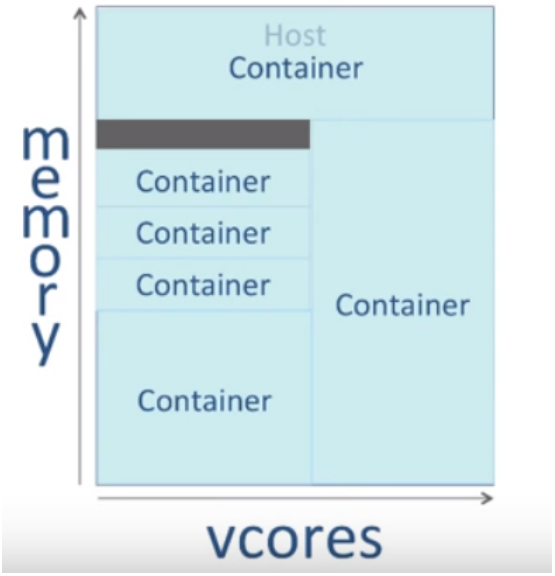


Figure 3.6: Example of YARN Container allocation. From [91]

allocation on YARN DataNodes. Given the available physical memory and vcores on a DataNode, we may build a container by allocating a subset of these two resources; according to the amount of memory and vcores allocated to each container, we will be able either to guarantee more resources to fewer job, allowing less parallelization of the existing tasks, or to define smaller containers on which more parallel jobs may run at the same time. YARN provides a spreadsheet to allow an optimal resource allocation, depending on the nature of the jobs that will run on the cluster. As shown in figure 3.6, the final aim of YARN is to allocate as much as possible of the available resources to running jobs.

3.4.3 Spark 2.0

Spark is defined as a general-purpose cluster computing system [42]. It was developed to overcome Hadoop MapReduce limitations when dealing with several kinds of applications, mostly iterative, such as many machine learning algorithms. To speed up the computational process Spark loads job data into memory, allowing them to be queried iteratively in a fast way; this same process, on the other hand, experienced a significant latency (tens of seconds according to [42]) with MapReduce, because data

were read from disk. The first version of Spark was mostly based on the definition of **RDD (Resilient Distributed Dataset)**, which is defined by its creators as a read-only collection of objects partitioned across a set of machines and that can be rebuilt if the partition is lost, meaning that the content of an RDD can be reconstructed in case of node failures or data losses starting from a given set of input data. The construction of an RDD may happen in several ways:

- by parallelizing existing collections of the hosting programming language (i.e. arrays, lists, etc.);
- by acquiring a file from a shared file system, like the above mentioned HDFS;
- by transforming an existing RDD into another one, through methods such as flatMap, filter, join, map, etc.

Each RDD can undergo two types of operations: **transformations** and **actions**. As we stated above, transformations are applied to an RDD to get another RDD, while actions (count, save, collect, etc.) aim at storing the content of an RDD into a local variable or in an output file.
Spark 2 is still based on the concept of RDD, but it allows the user to work at an higher level of abstraction by defining an environment based on the concepts of **DataFrame** and **Pipeline**. A DataFrame is defined as an immutable and distributed collection of data, organized into columns, similar to a table in a relational database [92]. The data types included in a DataFrame range from vectors to text, to a wide variety of structured data. Since, as we said before, the DataFrame object is immutable, it can only serve as an input to a Pipeline to generate another DataFrame; the Pipeline is organized in a sequence of stages, which are either **Transformers** or **Estimators** in a given order. Each Estimator is an object characterized by a .fit() method, that originates a Model, which is a Transformer, an object that calls a .transform() method on the input DataFrame, yielding the requested output DataFrame. To better distinguish among the two, the Spark 2.2.0 API documentation [93] provides a simple example of Logistic Regression applied to a given text. The Pipeline involves several stages:

1. splitting the text into single words;
2. converting the words into unambiguous numerical feature vector;

3. obtaining and applying a prediction model.

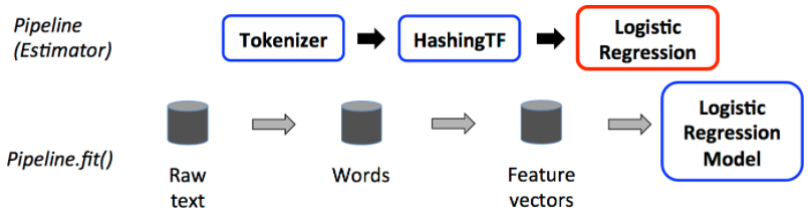


Figure 3.7: Pipeline model extraction. From [93]

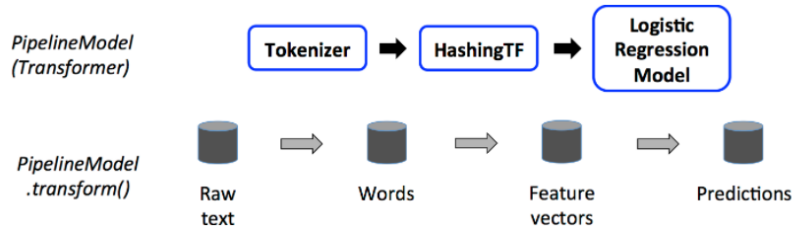


Figure 3.8: Pipeline model application. From [93]

Figures 3.7 and 3.8 illustrate respectively the training and testing phase of the model. Step 1 and 2 are in both cases performed by transformers (Tokenizer and HashingTF), but the two figures differentiate themselves in the last phase: in image 3.7, the .fit() method is called on the Pipeline, extracting the Model we would later need for making predictions on a new dataset. In image 3.8, the same model is then applied to the new raw data, finally yielding the required prediction result. This kind of operations is particularly important when it comes to data analytics applications, since often raw data are not properly structured or typed to be used as direct inputs for such problems and they need to undergo similar steps to be correctly processed.
Moreover, being Spark a general-purpose system, it is able to transparently give support to different libraries and frameworks that run on top of it, as shown in figure 3.9; some of these frameworks will be illustrated in detail in the next section.

3.4.4 Hive, Zookeeper and Oozie

Apache Hive provides a data warehouse system which facilitates reading, writing, and managing large datasets in distributed storage systems as HDFS [94]. It uses an SQL-

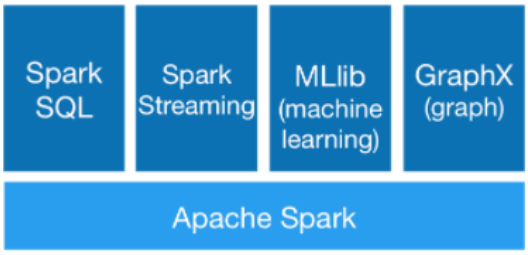


Figure 3.9: Set of frameworks running on top of Spark. From [93]

based language called **HiveQL**. Although being SQL inspired, the HiveQL dialect experiences some limitations with respect to the former one, as some standard SQL operations are still not possible or more difficult to execute (the INSERT INTO command, for instance, only accepts the insertion of ordered values, without allowing the user to specify the columns of interest); on the other hand, running on a distributed environment, it is more suited to handle big-data problems compared to a standard relational database system. On the Big-DAMA cluster, the Hive metastore is accessed through the **HUE** interface, that makes it possible to run queries interactively.
ZooKeeper is defined in [95] as a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. It aims at collecting and coordinating all distributed services on a centralized interface. Oozie is a workflow scheduler system to manage Apache Hadoop jobs. Its workflow jobs are Directed Acyclical Graphs (DAGs) of actions [96].

3.4.5 Spark MLlib

As mentioned before, we are able to run on top of Spark several modules providing more specific capabilities to the general system. Spark MLlib aims at solving machine learning problems on large datasets by efficiently exploiting Spark distributed environment. Also in this case, the first version of the API was RDD-based, but it switched to a DataFrame (and DataSet, when dealing with Java and Scala langauges) based API to allow an easier integration with an SQL-like environment such as the Hive one. This latter version was unofficially named ”Spark ML” to be distinguished from the former one, but it is still part of the same package. Among the tools the library provides we recognize not only common ML algorithms, but also tools for featurization (feature ex-

traction, selection, trasformation, etc.), evaluation and useful statistics extraction; the concepts taken into account when dealing with Pipelines in the above section are also strongly exploited also when dealing with the provided methods. Figure 3.10 below highlights, among the various modules, the methods used to implement the case study in chapter 4. The **Feature** module has mostly been used to transform and scale the existing features and prepare them to be correctly used as input for the classifiers, the **Classification** module provided the basic algorithms to be applied, the **Tuning** module allowed the construction of a parameter grid to be used for Cross Validation, while the **Evaluation** module was used to compute the Area Under Curve and other performances for each of the implemented models. The practical implementative details and results will be further discussed in the following chapters.

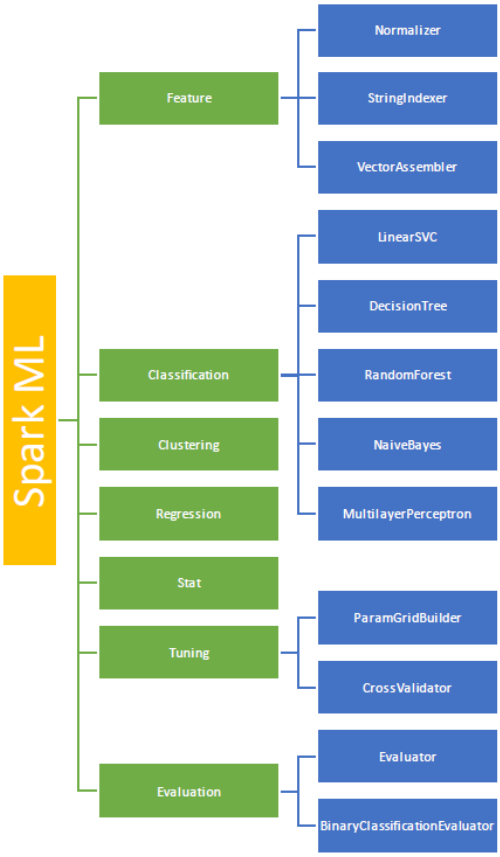


Figure 3.10: Modules of the Spark ML framework.

3.4.6 Future improvements

To provide our platform with online data processing capabilities, it must be equipped with services and frameworks supporting the streaming of data. For this purpose, **Kafka** service [100] in cooperation with the **Spark Streaming** framework [101] have been chosen.

Kafka

Kafka is defined as a distributed messaging system based on the **publish-subscribe** paradigm. Such paradigm identifies three main entities: the *producers*, that generate data associating them to specific categories, called *topics* (represented by the message icons in different colours in figure 3.11), the *consumers* that subscribe for a specific topic and wait for data to be forwarded to them, and the *broker*, the entity that handles the message forwarding.
Up to now, the structure may seem very similar to a centralized pub-sub system, but with the main difference that Kafka supports more than a single broker, allowing to build an entire cluster of brokers, and it is particularly resilient when it comes to messages handling. Kafka topics are in fact splitted into partitions, that allow their parallelization among different brokers, that may serve more consumers all at once. Each message within a partition has a unique identifier called *offset*, from which the consumer can start to read. Such partitions are then replicated over the different brokers, reducing the possibility for the message to be lost.

Figure 3.11: Kafka basic schema.

Spark Streaming

As reported in [101] Spark Streaming is an extension of the Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams. Data can be ingested from many sources, Kafka among the others, and can be processed using complex algorithms expressed with high-level functions (e.g. map, reduce or join) and machine learning algorithms provided by Spark ML. Finally, processed data can be pushed out to filesystems, databases, and live dashboards.
Spark Streaming work is internally based on the concept of discretized streams or **DStreams**, meaning that the arriving live streams are splitted into mini-batches, to be given as input to the standard Spark engine.

Chapter 4

Case study

This chapter will describe in detail the implemented use case targeting the application of different **supervised machine learning techniques** for anomaly detection to several months of network traffic. Each of the following sections will describe one of the steps involved in the process of knowledge discovery from data, as illustrated by figure 4.1

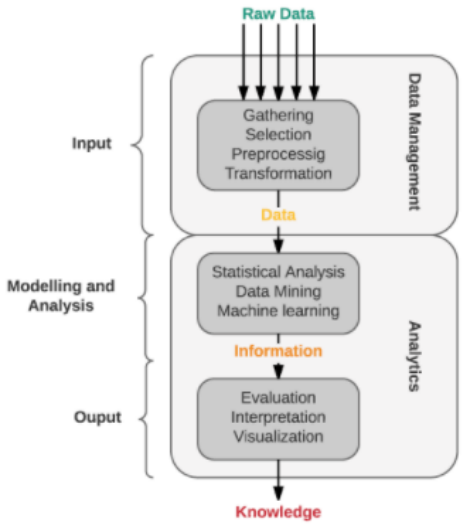


Figure 4.1: Process of knowledge discovery form data.

4.1 WIDE/MAWI Input data traces

This case study uses as input the traffic traces provided by the MAWI data repository provided by the WIDE project [97], collecting traffic flowing on a trans-pacific link since 2001. Traffic traces are collected on a daily base for 15 minutes using tcpdump, and then anonymized making them usable for research purposes. All the flows in the dataset have been classified through a **graph-based methodology** by the MAWILab project [9] to be distinguished in four macro-categories: **anomalous**, **suspicious**, **notice** and **benign**. The case study exploits this classification as a ground truth to train the machine learning algorithms and test their performances.

4.1.1 Data preprocessing

The daily traffic traces are fully available to be downloaded from the MAWILab website and come in **.pcap format**.The preprocessing phase is performed locally on a single node of the cluster, due to some problems in interfacing HDFS and the **Wireshark Network Analyzer** tools used to extract the preliminary features.
Each 15 minutes file is first splitted into 1 seconds chunks with the command:

```
editcap -i 1 <filename> <output folder>
```

for a total of 900 splits per day. Each file is autonomously named by editcap in the format: YYYYMMDDHHmmss.
The splits are then decoded in human readable format with tshark, to extract the basic features that will be used as input for the following steps, with:

```
tshark -T fields -n -r <filename> -E aggregator=,
-e <field> > <output folder>
```

Tshark yields as a result a file containing for each row the fields specified with the -e option: as we can see in more detail in table 4.1 the network analyzer was asked to extract the time at which the frame was captured in ISO format, the frame volume, some fields from the IP header (upper layer protocol, time to live, protocol version, packet length, source and destination addresses anonymized), either TCP or UDP source and destination ports, and, if the transport layer protocol is TCP, the flag values, the segment length and the window size. Each of those files is then uploaded to the distributed

filesystem and finally imported in the Hive database **mawi_traces** as a single table. Each table is named in the format **"a+splitNumber+a"**.

Layer	Feature	Description
Data Link	frame.time	Time of packet capture
	frame.len	Total length of the frame
IP	ip.proto	Upper layer protocol
	ip.len	Length of IP packet
	ip.ttl	Packet Time To Live
	ip.version	IPv4 or IPv6
	ip.src	Source IP address
	ip.dst	Destination IP address
UDP	udp.srcport	UDP source port
	udp.dstport	UDP destination port
TCP	tcp.flags	Number of flags in packet
	tcp.flags_ack	Presence of ACK flag
	tcp.flags_cwr	Presence of CWR flag
	tcp.flags_fin	Presence of FIN flag
	tcp.flags_ecn	Presence of ECN flag
	tcp.flags_ns	Presence of NS flag
	tcp.flags_push	Presence of PUSH flag
	tcp.flags_syn	Presence of SYN flag
	tcp.flags_urg	Presence of URG flag
	tcp.len	Length of TCP packet
	tcp.window_size	Size of receiver window
	tcp.srcport	TCP source port
	tcp.dstport	TCP destination port

Table 4.1: Preliminary features extracted with tshark

4.1.2 Feature extraction

Each table belonging to the above mentioned database is then further processed to extract the final set of features employed to feed the machine learning algorithm; the content of each mawi_traces table, corresponding, as we said, to one second of capture, will result in one row of the final input table: each table will represent one day of data with a total of 900 rows. Before proceeding with the feature extraction, the **mawi_features** database schema is defined and initialized to contain the final features

that we summarize in table 4.2. It should be noted that besides using traditional features such as min/avg/max values of some of the input measurements, we also consider the empirical distribution of some of them, sampling it at many different percentiles. This provides as input much richer information, as the complete distribution is taken into account. We also compute the empirical entropy of these distributions, reflecting the dispersion of the samples in the corresponding time slot. Each table in the

Field	**Feature**	**Description**
Tot. volume	#_pkts	Number of packets
	#_bytes	Number of bytes
PKT size	pkt_h	Packet size entropy
	pkt_{min,avg,max,std}	min, max, average, standard deviation on pkt size
	pkt_p{1, 2, 5, ..., 95, 97, 99}	Percentiles
IP proto	#_ip_proto	Number of different upper layer protocols
	ip_proto_h	Protocols entropy
	ip_proto_{min,avg,max,std}	min, max, average, standard deviation on protocols
	ip_proto_p{1, 2, 5, ..., 95, 97, 99}	Percentiles
	%ICMP/TCP/UDP	Share of upper layer protocols
IP TTL	ttl_h	Time To Live entropy
	ttl_{min,avg,max,std}	min, max, average, standard deviation on TTL
	ttl_p{1, 2, 5, ..., 95, 97, 99}	Percentiles
IPv4 - IPv6	%_IP_v4/v6	Share of IPv4/v6 packets
	#_ip_src/dst	Number of unique IPs
	top_ip_src/dst	Most used IPs
TCP/UDP port	#_port_src/dst	Number of unique ports
	top_port_src/dst	Most used ports
	port_h	Port entropy
	port_{min,avg,max,std}	min, max, average, standard deviation on ports
	port_p{1, 2, 5, ..., 95, 97, 99}	Percentiles
TCP flags	flags_h	Flags entropy
	flags_{min,avg,max,std}	min, max, average, standard deviation on TCP flags
	flags_p{1, 2, 5, ..., 95, 97, 99}	Percentiles
	% SYN/ACK/FIN...	Share of TCP flags
TCP WND	wnd_h	TCP window entropy
	wnd_{min,avg,max,std}	min, max, average, standard deviation on TCP WND
	wnd_p{1, 2, 5, ..., 95, 97, 99}	Percentiles

Table 4.2: Input features for classification algorithms

mawi_features database was named in the format **a+YYYYMMDD+a**, it consists of 900 rows, one per split of the original daily traffic file, and 117 columns (the percentiles data are stored in vectorial form as a single entry, when considering each percentile separately we get a total of 249 columns). The header schema was previously defined in an empty table called schema_ex, to ease the initialization process. Due to the massive amount of data exploited and to the large number of features to be extracted and written in the final database, several alternatives were tested to speed up the traces processing. A first version of the feature extraction procedure, implemented in **numpy** on a single machine of the cluster, took in average 2' 30" to process a single table; with a second implementation of the process, that exploited the **distributed computing capabilities** of the cluster by using RDD-related methods, almost 1' minute per table was gained. For the final implementation of the code, **threading** was used together with **Spark SQL**. To implement threading on the cluster, the first operation to be done is to set the **scheduler mode to FAIR** by inserting the following line in the code:

```
conf.set('spark.scheduler.mode', 'FAIR')
```

by default, the Spark job scheduler runs in **FIFO** fashion, hence the first job gets priority on all available resources. By changing this setting, the scheduler assigns tasks in a **Round Robin** fashion, so all jobs get a roughly even share of resources on the cluster during the execution. This operation allows the preprocessing to last in average about 1' per table. It should not be excluded that an enahncement in the cluster hardware may lead to a further improvement of the performances of the code in terms of time gain. The IDs of all the already processed traces tables are stored in the copied_tables database in the keep_track table. When launching the feature extraction script, we filter the IDs of the already processed tables from complete list of table IDs (in the mawi_traces database), so that, even if the process gets stopped, the extraction will be resumed from the last table processed. This operation generates an overhead of about 20 minutes before the actual start of the computation, overhead which will potentially increase in proportion to the further extension of the keep_track table. The result of each threading task is appended to a globally defined Python dictionary whose keys are named after the columns in the mawi_features tables. This detail will ease the construction the final INSERT query by iterating on the dictionary itself, because, as we stated when dealing with the Hive service in chapter 2, HiveQL requires the INSERT query to provide all fields in the correct order.

4.1.3 Labels extraction

As mentioned at the beginning of this section, MAWILab provides, together with the traffic traces, a database of labels locating traffic anomalies [9]. The project associates to each day of traffic an .xml file named YYYYMMDD_anomalous_suspicious.xml, in which every anomaly is represented in the following structure:

```
<anomaly type="T" value="Dn,Da,C0,V,C1">
   <description>
      "Structure of the community reporting
      the anomaly (in dot language)"
   </description>

  <slice>
      <filter "Traffic features describing the anomaly:
                     destination IP
                     and/or source IP
                     and/or destination port
                     and/or source port">
   </slice>
   <from "timestamp of the start of the anomaly">
   <to "timestamp of the end of the anomaly">
</anomaly>
```

The anomaly tag includes a type ”T”, expressed either as **anomalous**, **suspicious** or **notice**, and several pieces of information for the field value, among which we consider in particular ”V”, the parameter showing **which detector with which parameters found the anomaly**. It is a vector of binary values, 0 means the detector did not report the traffic whereas 1 means that the detector reported an alarm for the anomaly. There is four detectors (Hough, Gamma, KL, PCA) each using 3 different parameter tuning (sensitive, optimal, conservative), for a total of 12 cells in the resulting vector. For our label extraction also the fields ”from” and ”to” indicating the anomaly timestamp are crucial to associate each anomaly to the correct time bin. The anomaly taxonomy provided by MAWILab is really wide and diverse, but for our use case we focused only on 5 types of anomalies:

- Netscan-TCP-ACK;
- Netscan-UDP ;
- Distributed Denial of Service (DDoS);
- MultiPoint To Point (mptp or HTTP flashcrowds);
- Ping-flood.

The processing of each .xml file lead to the creation of a new table named with the format a+YYYYMMDD+a in the **mawi_anomalies** database. Each table has 900 rows, each one coinciding with a row in the feature table, and 5 columns, one per anomaly; the input values are binary: 1 if the split is affected by the specific anomaly, 0 viceversa. The label extraction script (see A.1.3 and A.1.4) is strongly based on the correct definition of time indexes; that is the reason why, before starting the proper file processing, the timestamp of the very first daily capture is extracted and stored to be used as a reference index. After filtering the anomalies according to the timestamp (all anomalies having length 0, starting at timestamp 0 and/or ending at timestamp 2147483645 are rejected), the script extracts the binary vector V, showing which detector found the anomaly. As we said before, V has 12 entries, corresponding to 4 detectors tuned with 3 different parameters. The final binary label is assigned through a majority voting technique: we exploit an auxiliary array having 900 entries, each of them storing the corresponding sum of V; the anomaly is marked as present (value 1) for all entries in the list being detected by the maximum number of detectors (i.e. by the highest value of sum of V in the currently examined array). The same operation is performed for all the anomaly types, which are then filtered to keep only the ones specified above. A query is then built to insert the final output in the correct table together with a numerical ID identifying each row, useful to join the labels to the correct feature row in the following steps.
Such process marked as anomalous, over a total of 37800 samples (42 days), a percentage of entries which is: 7.473% for the DDoS attack, 3.669% for ping flooding, 26.746% for netscan-ACK, 25.373% for netscan-UDP, and 35.986% for MPTP.

4.2 Modelling and analysis

4.2.1 Basic classifiers

The first phase of the case study involves the use of five different fully supervised ML models: **Decision Tree**, **Random Forest**, **Naive Bayes** and **Neural Networks**, **Support Vector Machine**, implemented by using the spark.ml package. Each classifier receives as input two columns, one containing the labels, and one containings the features to be used for training and testing. For this reason, the traffic features inserted in the database in separated cells need to be gathered together, each row in a single vector. A VectorAssembler object is used for this purpose. Such object is defined in the pyspark ML documentation as a feature transformer that merges multiple columns into a vector column; it receives as input parameters the list of columns to merge and the name of the output column, and yields as a result a single vector containing the whole set of features in each row. After this step, the transformed dataset is joined to the corresponding label table, and the final dataset is eventually split into two random non-overlapping training and testing sets with a ratio 80%-20%. In order to reduce overfitting, 10-fold Cross-Validation was applied to all models. The CrossValidator object provided by the pyspark.ml.tuning package depends on the paramGridBuilder, an entity which takes as input a set of different configurations to test on the basic classifier object. The grid builder is provided as input to the cross validator together with an evaluator for the current model, the model itself and the number of folds to be performed. The output of the fitting phase is the best model among the ones obtained by the various settings, and is then applied to the testing dataset for the final classification. All problems are threated separately as binary classification problems.
The following paragraphs will proceed with a description of the underlying idea of the implemented models, as reported in [78].

Decision Tree

Decision trees are widely used since they are easy to interpret, handle both categorical and continuous features, do not require feature scaling, and are able to capture non-linearities and feature interactions. The decision tree is a greedy algorithm that performs a recursive binary partitioning of the feature space. The tree predicts the same label for each leaf partition. Each partition is chosen greedily by selecting the

best split from a set of possible splits, in order to maximize the **information gain** at a tree node. The information gain is the difference between the parent node impurity and the weighted sum of the two child node impurities, where the **node impurity** can be calculated as:

$$Gini = \sum_{i=1}^{C} f_i(1 - f_i)$$

or as:

$$Entropy = \sum_{i=1}^{C} -f_i log(f_i)$$

Where C is the total number of labels, and f_i is the frequency of label i at the node. And the information gain is calculated as:

$$IG(D) = Impurity(D) - \frac{N_{left}}{N} Impurity(D_{left}) - \frac{N_{right}}{N} Impurity(D_{right})$$

Where D is the parent dataset, and N its size, while D_{left} and D_{right} are the child datasets, with N_{left} and N_{right} their respective sizes.
While instantiating the classifier, together with the above mentioned input columns, the parameter maxDepth is required; such specification is used as stopping condition for the algorithm.

Random Forest

Random Forests are ensambles of Decision Trees. They share their same basic properties and capabilities, and, moreover, the trees combination is helpful to **reduce overfitting**. The training of the set of used decision trees is done separately so that it can be executed in parallel with the others, but some randomness is injected in the training process to reduce the variance of the predictions. Randomness is injected by subsampling the original dataset on each iteration to get a different training set or considering different random subsets of features to split on at each tree node. To make a prediction on a new instance, a random forest must aggregate the predictions from its set of decision trees. In the case of classification, the aggregation is done by majority vote. Each trees prediction is counted as a vote for one class. The label is predicted to be the class which receives the most votes.

Naive Bayes

Naive Bayes is a simple multiclass classification algorithm with the assumption of independence between every pair of features. Naive Bayes can be trained very efficiently. Within a single pass to the training data, it computes the conditional probability distribution of each feature given label, and then it applies Bayes theorem to compute the conditional probability distribution of label given an observation and use it for prediction. The algorithm is suitable to perform both multiclass (multinomial Naive Bayes) and binary classification (Bernoulli Naive Bayes), provided that the features in input are non-negative.

Neural Networks

The Multilayer Perceptron Classifier (MLPC) provided by the Spark ML framework is a classifier based on the **feedforward artificial neural network**. MLPC consists of multiple layers of nodes. Each layer is fully connected to the next layer in the network. Nodes in the input layer represent the input data. All other nodes map inputs to outputs by a linear combination of the inputs with the nodes weights w and bias textitb and applying an activation function. This can be written in matrix form for MLPC with $K+1$ layers as follows:

$$y(x) = f_K(...f_2(w_2^T f1(w_1^T x + b_1) + b_2)... + b_K)$$

Nodes in intermediate layers use **sigmoid** function:

$$f(z_i) = \frac{1}{1 + e^{-z_i}}$$

Nodes in the output layer use **softmax** function:

$$f(z_i) = \frac{e^{z_i}}{\sum_{k=1}^{N} e^{z_k}}$$

The number of nodes N in the output layer corresponds to the number of classes. MLPC employs backpropagation for learning the model. When instantiating the MLPC classifier, we need to specify the dimensions of input (245 features), intermediate and output (2 classes) layers.

Support Vector Machine

A support vector machine constructs a **hyperplane** or set of hyperplanes in a high- or infinite-dimensional space, which can be used for classification, regression, or other tasks. Intuitively, a good separation is achieved by the hyperplane that has the largest distance to the nearest training-data points of any class (so-called functional margin), since in general the larger the margin the lower the generalization error of the classifier. LinearSVC in Spark ML supports binary classification with linear SVM.

4.2.2 Super Learner classifier

Each of the described basic classifiers is known to perform particularly well if applied to specific problems and datasets; by developing an **ensamble learning model** we want to combine the strengths of each classifier to overcome each one's limitations and increase their predictive force while reducing their overfitting tendencies. The theory behind the model is described in detail in [1], but in this section we want to describe the basic idea behind the algorithm to ease the comprehension of the folllowing implementstion steps.
The so-called **Super Learner** is based on the concept of cross-validation: given the usual labeled dataset of size N, we split it into K sub-sets of $\frac{N}{K}$ samples each, and we use at each iteration $K\text{-}1$ sub-sets as a **training set** and the remaining one as a **validation set**, so that at the end of the process each split is used once for the testing phase. At each iteration we train and test on J **first level learners**, and we append the prediction output to a new dataset called Z, having final size $N\ x\ J$. This new dataset is used as input for the second level training phase of the so-called **meta-learners**, from which the final predictions will be obtained. According to the different implementation of the meta-learners, the Super Learner can be adapted to perform regression or classification, and it is proved to be optimal, in the sense that it performs at least asymptotically as well as the best first level learners available.
In the binary classification example taken into account, the meta-learners implement a **weighted-majority-voting-classifier** where, being $h_h(X)$ the first level prediction for each learner, we compute:

$$H(X) = \sum_{j=1}^{J} w_j h_j(X)$$

And we define a threshold β such that, if $H(X) > \beta$ the sample is assigned to the positive class, or, if not, to the negative class. In this implementation, a special attention must be payed to the computation of the weigths w_j, which is done in three different ways:

- **Uniform**: $w_j = \frac{1}{J}$, all basic learners have the same weight;
- **Accuracy**: $w_j = \frac{\alpha_j}{\sum_{i=1}^{J} \alpha_i}$, where α_i is the accuracy of the i-th learner, i.e. the fraction of true classification on the whole training dataset;
- **Exponential**: $w_j = \frac{e^{\lambda \alpha_j}}{\sum_{i=1}^{J} e^{\lambda \alpha_i}}$, being λ a coefficient used to reduce the impact of predictors having low accuracy.

The practical implementation involves the code porting and testing of such model from a scikit-learn version running on a single node, whose performance and implementation are explained in detail in [98], to a distributed pyspark.ml version. This latter version exploits only three out of the five basic classifiers included in the centralized one and mentioned above, since the Support Vector Machine model is not available on Spark 2.0.0, and the output provided by the Multilayer Perceptron Classifier is not resembling the ones of the other classifiers, given that it is lacking of the "probability" column to be appended to the Z matrix.
Particular relevance should be given to the differences between the cross-validation implementations: while the scikit-learn cross validator (train_test_split object) only outputs the starting and ending indexes of the training and testing sets at each iteration, the pyspark cross validator directly tests all possible models and yields the best one. Thus, a manual implementation of the cross validation resembling the scikit-learn one is necessary: the matrix X, the normalized input dataset, is splitted into k subsets (the randomSplit method does not provide splits of exactly equal size, but it guarantees that they are not overlapping), and they are manually combined (*k-1* splits for training and 1 for testing) to iteratively undergo the .fit() and the .transform() method on the first level learners, and append to the matrix Z the prediction probability output.
To maintain compatibility with the basic learners, also the **MajorityVoting** class implements a fit() and a transform() method. The parameters required by the fitting operation vary according to the weight computation technique selected (uniform, accuracy or exponential, provided in string form by the user when instantiating the object). The

weights are computed and returned in form of a *K x 1* array. The transform operation multiplies every column of the Z matrix times the corresponding weight, and then rounds the obtained values to map the weighted probabilities either to class 0 or to class 1 (hence the β threshold is implicitly fixed to 0.5).

4.3 Classification results

4.3.1 Basic classifiers with full features set

The traffic processed for this case study (see also [99]) spans two months of packet traces collected in late 2015. The following graphs report the detection performance of each classifier using the **full set of input features**, depicted in terms of **Receiver Operating Characteristic** curve. The ROC curve depicts on the y-axis the True Positive Rate (or sensitivity) versus the False Positive Rate ($1 - specificity$). The performance of each model is tested against the *ideal classifier*, depicted in the upper-left corner of the graph, representing a TPR of 100%, against an FPR of 0%, and, on the other hand, the *random classifier*, identified by a line cutting the graph at 45 degrees. The points on the graphs are obtained by varying the false positive threshold (i.e. the probability of false positive the algorithm is not allowed to exceed), hence by defining a set of prior reasonable values for the specificity.
The **scikit-learn.metrics.roc_curve** method was used to extract the graphs. It takes as inputs the true labels (*y_true*), the probability of beign marked as anomalous, as extracted by the classification algorithm (*y_score*), and it returns the arrays containing the points for x and y-axis, together with the tested thresholds. Such thresholds are computed by the method *_binary_clf_curve* taking all the distinct values of the given y_score in decreasing order.
The results are also reported in terms of **Area Under Curve** (AUC) and **execution time** (relative with respect to the smallest execution time in the model benchmarking, SVM with CFS feature selection technique, lasting 110ms, and including also the time spent to perform 10-folds cross-validation).
In general, we can say that the results are really satisfying in terms of accuracy, with Random Forest and Multilayer Perceptron classifiers yielding the best performance

with the detection of almost the 80% of the attacks without false alarms (AUC in green in tables 4.4 and 4.6). Despite the very similar results, when choosing between the two a consideration on the execution time should be made: the execution time of the MLP (table 4.6) model turned out to be 3 orders of magnitude longer than the RF one, with a large impact on the total generated by the training of the neural network. This result makes the RF classifier particularly appealing when it comes to NTMA applications.
Furthermore, we should notice a general lower performance in the classification of the Distributed Denial of Service attack (blue dotted line in all graphs), with the Naive Bayes model yielding the lowest AUC value; on the other hand, the best classification performance is obtained when classifying Multipoint To Point or Netscan-UDP anomalies.
The next section will deal with the impact of several feature selection techniques on the classifiers results.

Decision Tree

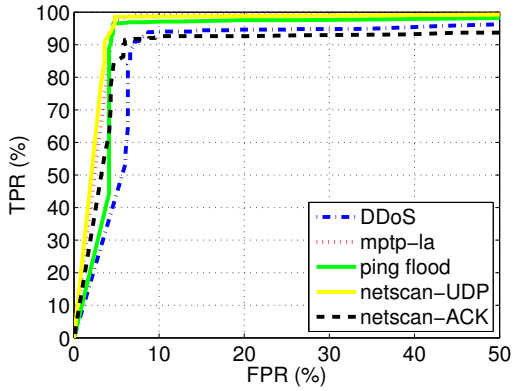


Figure 4.2: ROC for Decision Tree algorithm.

Anomaly	AUC	Relative ET
DDoS	0.922	27.8
MPTP	0.972	12.3
Ping-flood	0.952	20.8
Netscan-UDP	0.972	14.4
Netscan-ACK	0.918	22.3

Table 4.3: AUC and ET of Decision Tree.

Random Forest

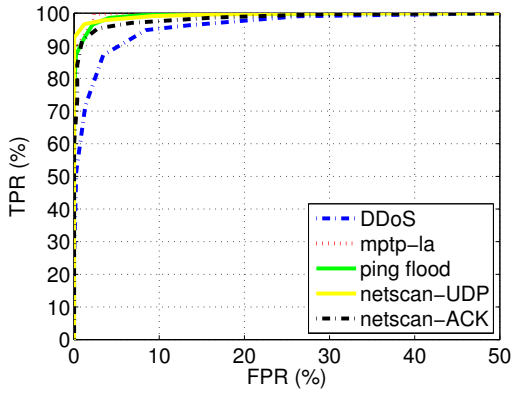


Figure 4.3: ROC for Random Forest algorithm.

Anomaly	AUC	Relative ET
DDoS	0.979	10.5
MPTP	0.998	4.6
Ping-flood	0.996	7.5
Netscan-UDP	0.995	7.2
Netscan-ACK	0.989	7.5

Table 4.4: AUC and ET of Random Forest.

Naive Bayes

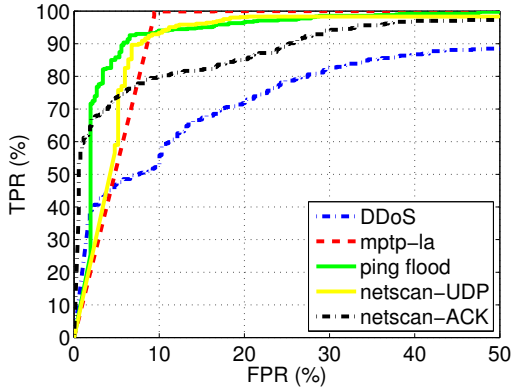


Figure 4.4: ROC for Naive Bayes algorithm.

Anomaly	AUC	Relative ET
DDoS	0.828	29.3
MPTP	0.952	27.4
Ping-flood	0.963	26.5
Netscan-UDP	0.944	26.4
Netscan-ACK	0.929	26.1

Table 4.5: AUC and ET of Naive Bayes.

Neural Networks

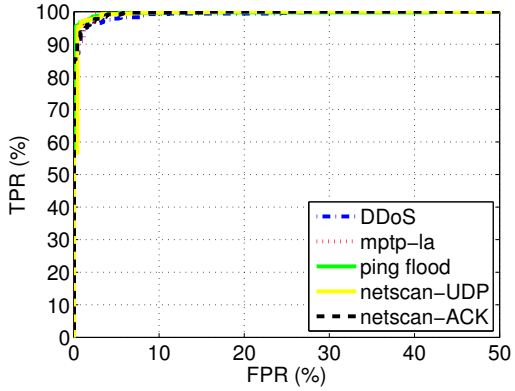


Figure 4.5: ROC for MLP algorithm.

Anomaly	AUC	Relative ET
DDoS	0.995	$27.4 * 10^3$
MPTP	0.998	$27.3 * 10^3$
Ping-flood	0.996	$27.3 * 10^3$
Netscan-UDP	0.997	$27.3 * 10^3$
Netscan-ACK	0.997	$27.4 * 10^3$

Table 4.6: AUC and ET for MLP.

Support Vector Machine

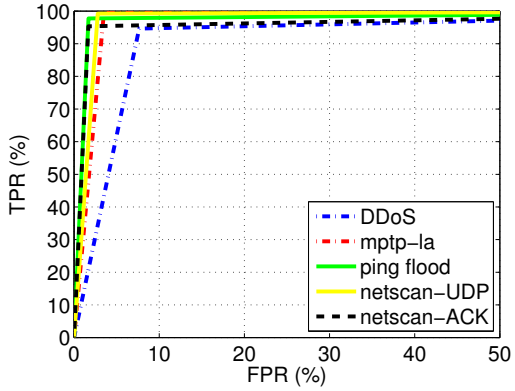


Figure 4.6: ROC for SVM algorithm.

Anomaly	AUC	Relative ET
DDoS	0.935	37.6
MPTP	0.982	5.5
Ping-flood	0.980	13.5
Netscan-UDP	0.982	11.2
Netscan-ACK	0.968	18.3

Table 4.7: AUC and ET for SVM.

4.4 Basic classifiers with feature selection

Despite being beneficial for some supervised learning techniques, having a large set of input features may lead to some drawbacks like over-fitting and, as we noticed from the results in the previuos section, an increase in the training time of the models. For these reasons, filtering the initial input to remove redundant or irrelevant features can be really helpful for the overall model performance. In [102], the problem is addressed by saying that "in the feature subset selection problem, a learning algorithm is faced with the problem of selecting a relevant subset of features upon which to focus its attention, while ignoring the rest. To achieve the best possible performance with a particular learning algorithm on a particular domain, a feature subset selection method should consider how the algorithm and the training data interact". The same work reports several definition of what **relevance** in a statistics concept is, stating that a feature V_i is relevant if and only if there exists some v_i and c for which $p(V_i = v_i) > 0$ such that:

$$p(C = c|V_i = v_i) \neq p(C = c).$$

We should keep in mind that feature selection does not involve the creation of new features by combining the already existing ones, but it only aims at selecting the best sub-set of input features. Such sub-set can be built by choosing different criteria. We consider correlation-based selection, following two different approaches: the **plain-top** and the **sub-set search selection**.

4.4.1 Plain-top feature selection

Such approach is also referred to as top-PLCC where PLCC stands for Pearson's Linear Correlation Coefficient. This procedure only takes into account the features which are mostly linearly correlated with the target, for each attack type. The linear coefficient r is computed as:

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

Where:

- n is the number of samples;
- x_i is a specific feature on the i-th sample;

- $\bar{x}$ is the average value of the specific feature;
- y_i is the label for the i-th sample;
- $\bar{y}$ is the average value of the label;

Figure 4.7 shows the absolute values of the linear correlation coefficients between features and attacks, separated by attack type. Features are sorted by decreasing correlation coefficient magnitude. A first observation is that features are in general poorly correlated to the attacks, with PLCC values generally below 0.5. Note that less input features are highly correlated to the DDoS class, which justifies the poorer performance obtained for this attack type. For the sake of top-PLCC feature selection, we keep features with a PLCC coefficient value above 0.2, resulting in 11, 29, 51, 45 and 47 features for the DDoS, HTTP flashcrowd, ping flood, UDP and TCP netscans attacks respectively.

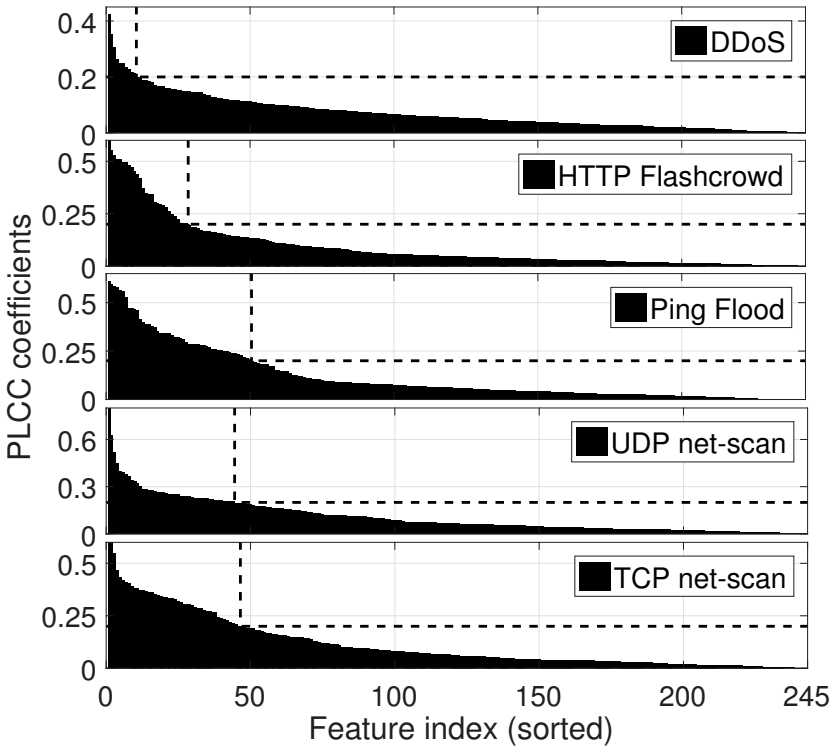


Figure 4.7: Linear correlation between features and attacks (absolute values).

Model	DDos	MPTP	Ping-flood	Netscan-UDP	Netscan-ACK
Decision Tree	0.874	0.965	0.956	0.970	0.920
Random Forest	0.942	0.988	0.991	0.996	0.988
Naive Bayes	0.824	0.982	0.981	0.925	0.959
Multilayer Perceptron	0.869	0.993	0.997	0.994	0.993
Support Vector Machine	0.755	0.982	0.948	0.938	0.900

Table 4.8: Detection performances of basic learners with top-PLCC feature selection in terms of AUC.

Model	DDos	MPTP	Ping-flood	Netscan-UDP	Netscan-ACK
Decision Tree	1.6	1.9	6.1	4.9	6.0
Random Forest	4.4	2.4	5.3	5.8	5.7
Naive Bayes	1.1	3.2	4.9	4.3	4.6
Multilayer Perceptron	25.6	21.6	$1.4 * 10^3$	$1.1 * 10^3$	$1.2 * 10^3$
Support Vector Machine	2.2	1.1	3.4	3.9	4

Table 4.9: Detection performances of basic learners with top-PLCC feature selection in terms of relative execution time.

4.4.2 Sub-set search selection

This methodology is presented in [103] as CFS, Correlation Feature Selection. Also in this work, the selected features are poorly correlated among each other, but highly correlated to the targets. The difference with respect to the previous case lays in the fact that the features are directly evaluated as a subset for the result they yield in term of *Merit* for the single subset S. This quantity is calculated as:

$$M_S = \frac{k\bar{r_{cf}}}{\sqrt{k + k(k-1)\bar{r_{ff}}}}$$

Where:

- k is the number of features in the considered subset;
- $\bar{r_{cf}}$ is the average feature-label correlation ($f \in S$);

- $\bar{r_{ff}}$ is the average feature-feature intercorrelation.

The above mentioned quantity is used as input for three different kinds of heuristics: the forward selection, the backward elimination, and the best first. The first starts with an empty sub-space, adding new features greedily, no new addition of features results in a better result; the second algorithm, on the other hand, starts from the full features set and removes one feature at a time until the model does not degrade; the third one can either start from no-features or from a full-features set, and it starts either adding or removing features until a stopping condition is met. In this case, we use best first search as search strategy, with the same stopping condition illustrated in [103]: the algorithm stops if five newly examonated subsets in a row give no better results with respect to the current best one. In the case of FS feature selection, the procedure selects 13, 19, 19, 21 and 19 features for the DDoS, HTTP flashcrowd, ping flood, UDP and TCP netscans attacks respectively.

Model	**DDos**	**MPTP**	**Ping-flood**	**Netscan-UDP**	**Netscan-ACK**
Decision Tree	0.924	0.944	0.972	0.958	0.938
Random Forest	0.984	0.987	0.995	0.995	0.990
Naive Bayes	0.863	0.993	0.969	0.969	0.950
Multilayer Perceptron	0.947	0.979	0.993	0.994	0.989
Support Vector Machine	0.803	0.916	0.932	0.933	0.877

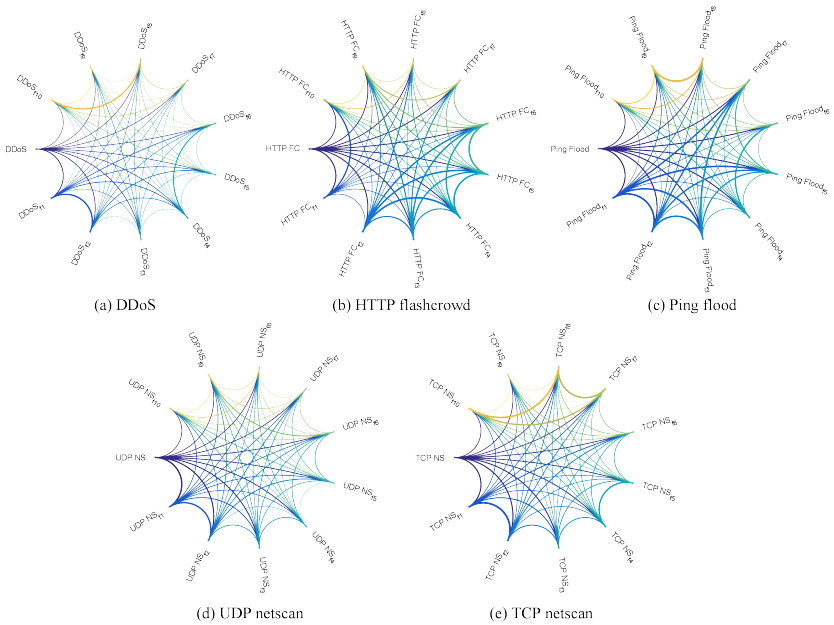
Table 4.10: Detection performances of basic learners with CFS in terms of AUC.

Model	**DDos**	**MPTP**	**Ping-flood**	**Netscan-UDP**	**Netscan-ACK**
Decision Tree	1.9	1.4	1.7	1.8	2.2
Random Forest	5	2.6	3.8	4	4.5
Naive Bayes	1.36	2	2	2.1	2
Multilayer Perceptron	59.5	76.1	74.6	13.8	72.8
Support Vector Machine	3	1	2.8	2	2.2

Table 4.11: Detection performances of basic learners with CFS in terms of relative execution time.

4.4.3 Results

Both tables 4.9 and 4.11 show a significant reduction on the execution time for each model. Among the other results should be particularly highlighted the reduction on the training time of the Multilayer Perceptron in the top-PLCC scenario when dealing with the DDoS and mptp-la attacks. From tables 4.8 and 4.10 we can see that feature selection does not impact on the quality of the models, which are still highly accurate, with Random Forest and MLP yielding again the best results. Particularly interesting is the case of the Decision Tree model, for which performance even slightly increases for some types of attacks, with a relevant reduction on the execution times.
To get a better understanding on which are the best features to detect the studied attacks, table 4.12 reports the top-10 correlated features per attack type, and figure 4.8 shows the inter-feature correlations among each set of 10 features, in the form of a circular graph. Features are coherent with the characteristics of each attack type, e.g., having a large number of packets towards a top targeted destination IP and destination port in the case of a DDoS attack, or taking into consideration the percentage of ICMP packets in the case of ping-flood. Note that in all cases, features derived from the empirical distributions are present in the top-10 features, suggesting that such types of features, are highly relevant for the sake of detection of network attacks.



(a) DDoS (b) HTTP flashcrowd (c) Ping flood

(d) UDP netscan (e) TCP netscan

Figure 4.8: Top-10 feature correlation graphs for the different types of attack.

	DDoS	**MPTP-LA**	**Ping-flood**
f 1	# pkts	% pkts → +TCPdst-port	% IPv4 pkts
f 2	% pkts → +IP	$p(TCP_{dst-port})$	% IPv6 pkts
f 3	tail $p(TCP_{src-port})$	head $p(TCP_{dst-port})$	$\bar{p}$(IPlen)
f 4	tail $p(UDP_{dst-port})$	head $p(TCP_{dst-port})$	% ICMP pkts
f 5	tail $p(TCP_{dst-port})$	tail $p(TCP_{dst-port})$	% pkts → +IP
f 6	head $p(UDP_{src-port})$	tail $p(TCP_{dst-port})$	# dst IPs
f 7	# TCPdst-ports	head $p(TCP_{win-size})$	head $p(IP_{len})$
f 8	head $p(IP_{TTL})$	% pkts → +IP	head $p(IP_{TTL})$
f 9	# src IPs	H($p(TCP_{dst-port})$)	head $p(IP_{TTL})$
f 10	head $p(IP_{TTL})$	tail $p(TCP_{src-port})$	# src IPs

	UDP netscan	**TCP-ACK netscan**
f 1	head $p(IP_{TTL})$	% IPv4 pkts
f 2	head $p(IP_{TTL})$	% IPv6 pkts
f 3	head $p(UDP_{dst-port})$	tail $p(TCP_{src-port})$
f 4	% pkts → +IP	head $p(TCP_{win-size})$
f 5	tail $p(UDP_{src-port})$	head $p(TCP_{win-size})$
f 6	$\bar{p}$(IPlen)	# TCPdst-ports
f 7	% UDP pkts	$\bar{p}$(TCPdst-port)
f 8	tail $p(UDP_{dst-port})$	tail $p(TCP_{dst-port})$
f 9	# dst IPs	head $p(TCP_{dst-port})$
f 10	# UDPdst-ports	$\hat{p}$(TCPdst-port)

Table 4.12: Top-10 correlated features per attack type.

To allow a better visualization of each algorithm performance, we have depicted in histogram form all the AUC and relative execution time data previously enumerated in tabular form. For the sake of readability, the relative execution times of the Multilayer Perceptron algorithm have been removed and substituted with a 0 when too high with respect to the other data.

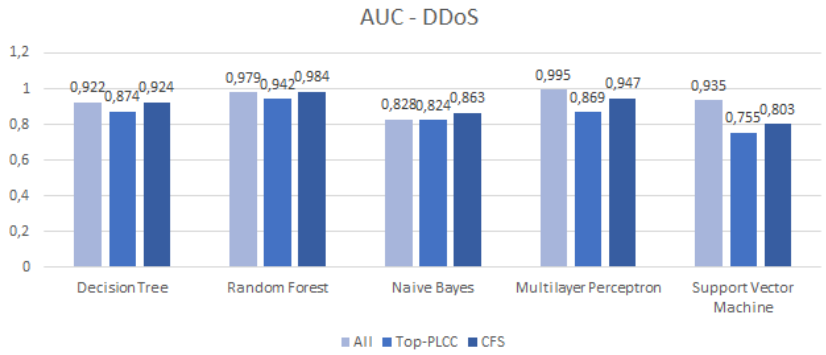


Figure 4.9: AUC results for DDoS attack in different configurations.

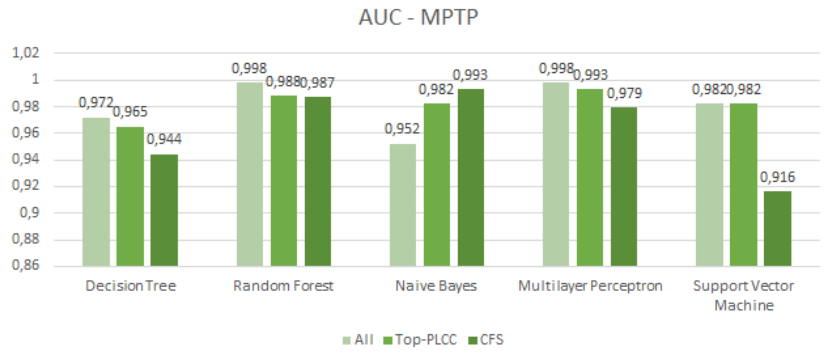


Figure 4.10: AUC results for MPTP attack in different configurations.

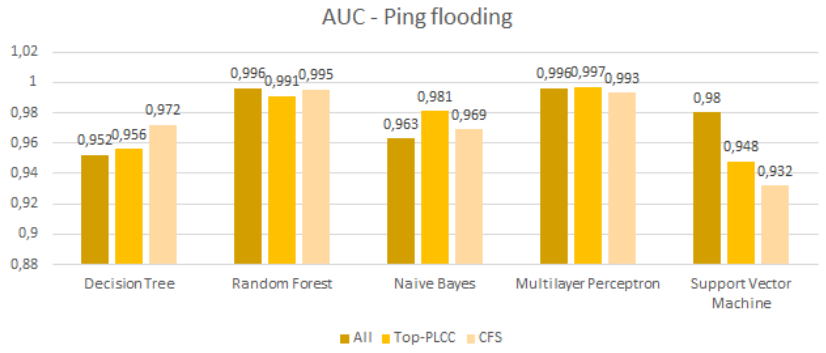


Figure 4.11: AUC results for ping flooding attack in different configurations.

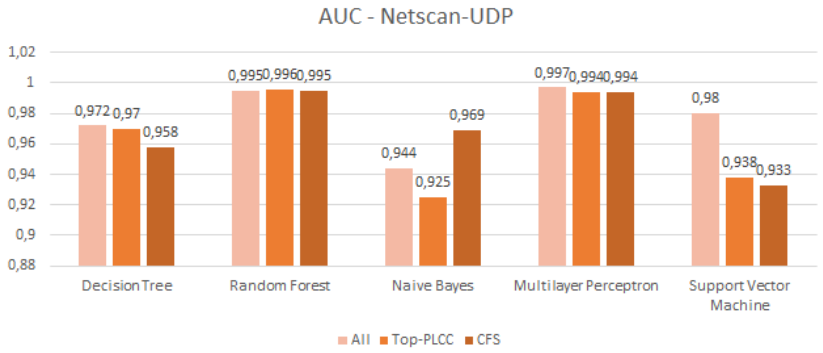


Figure 4.12: AUC results for netscan-UDP attack in different configurations.

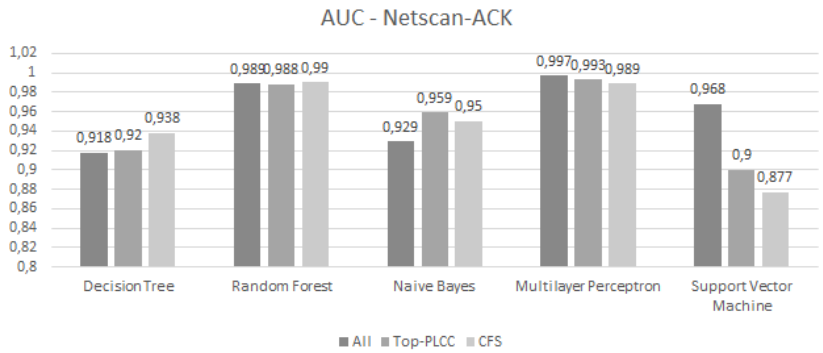


Figure 4.13: AUC results for netscan-ACK attack in different configurations.

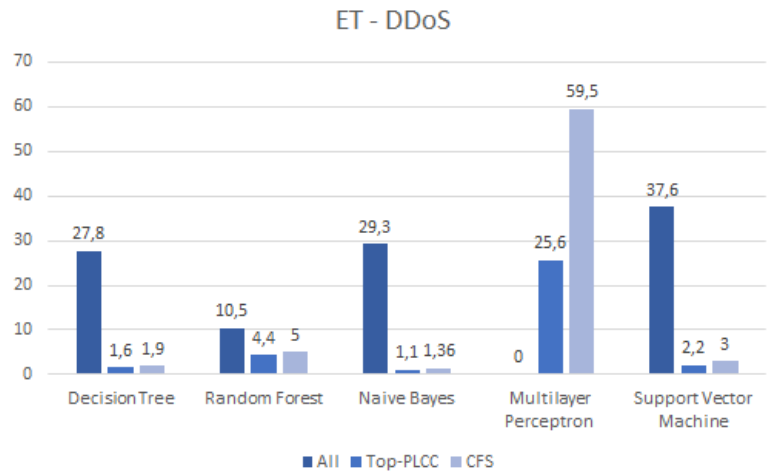


Figure 4.14: Relative execution time results for DDoS attack in different configurations.

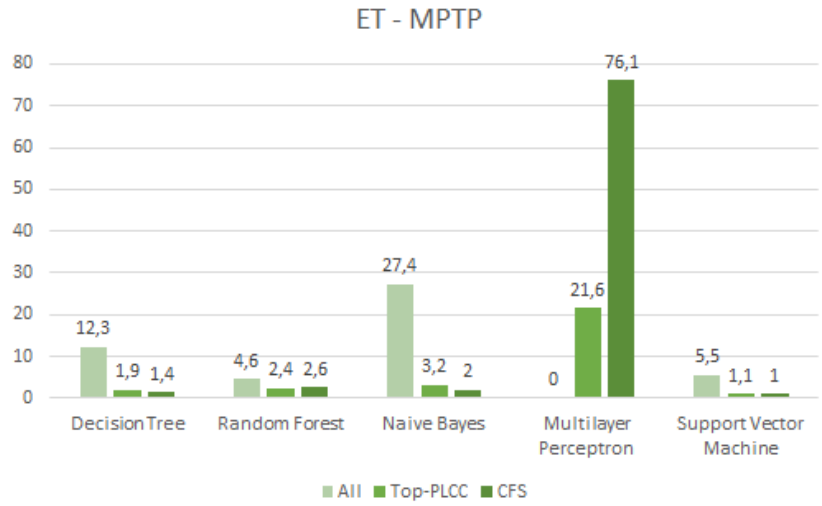


Figure 4.15: Relative execution time results for MPTP attack in different configurations.

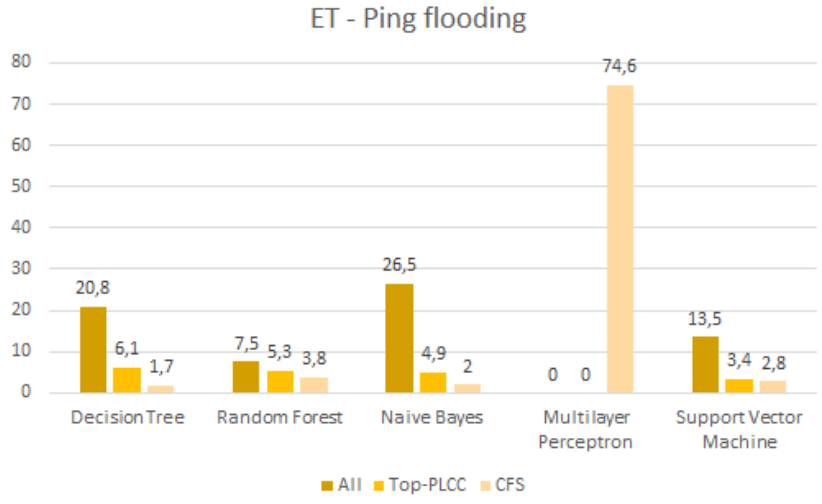


Figure 4.16: Relative execution time results for ping flooding attack in different configurations.

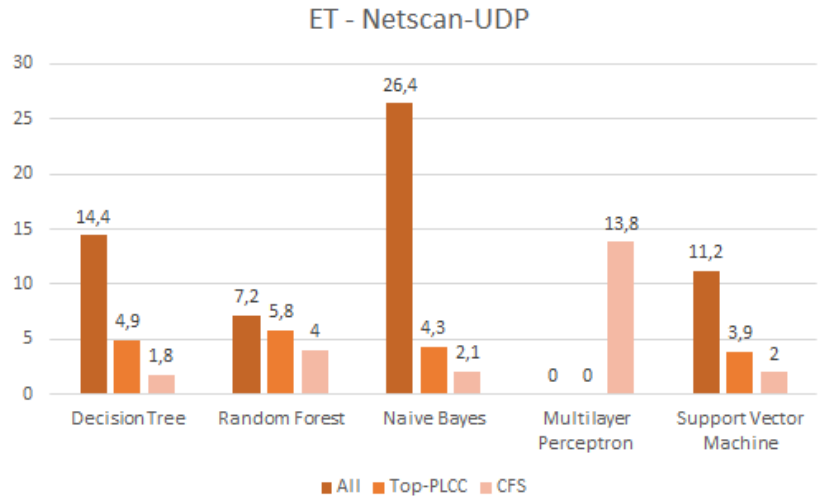


Figure 4.17: Relative execution time results for netscan-UDP attack in different configurations.

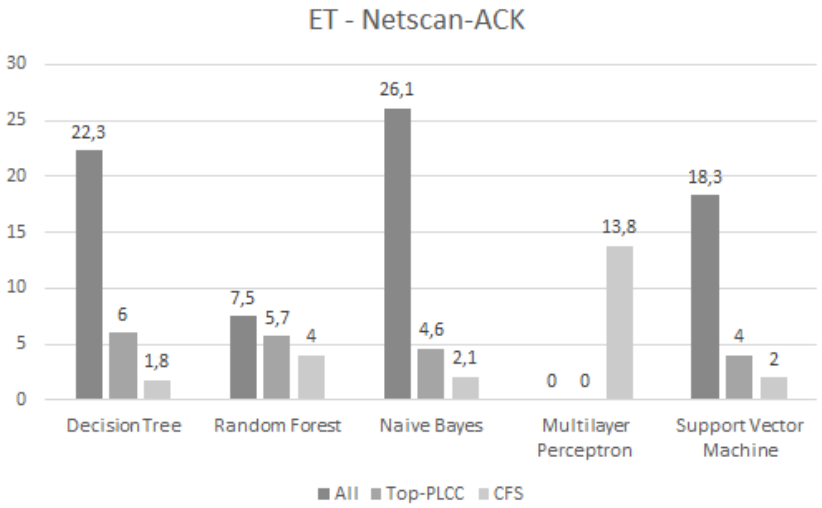


Figure 4.18: Relative execution time results for netscan-ACK attack in different configurations.

4.5 Super Learner classifier

Testing this ensamble learning implementation on the previously described environment highlighted the importance of a correct dimensioning of the YARN containers to fully exploit the available cluster resources. Both the Cross-Validator object and the manual implementation of the cross-validation phase required, in fact, a particularly high amount of memory, which was impossible to obtain due to limited hardware resources on the Big-DAMA cluster. Even if tested on the smallest possible portion of data, i.e. a single table spanning 900 rows, the executed code lead to errors like:

```
ERROR cluster.YarnScheduler: Lost executor 9 on
bigdama-vworker4-phy1.bigdama.local: Container killed by YARN
for exceeding memory limits. 1.5 GB of 1.5 GB physical memory
used. Consider boosting spark.yarn.executor.memoryOverhead.
```

The attempt to specify an higher value ('6G') for the suggested setting parameter **'spark.yarn.executor.memoryOverhead'** generated another error:

```
Required executor memory (1024+6144 MB) is above the max
threshold (2933 MB) of this cluster!
Please check the values of 'yarn.scheduler.maximum-allocation-mb'
and/or 'yarn.nodemanager.resource.memory-mb'.
```

The settings above can be modified through the YARN panel on the Cloudera Manager interface (YARN > Configuration > Resource Management), and will become active after having restarted all the services on the cluster. Unfortunately, in the specific case, the changes in setting were not supported by the scheduler, so, up to now, since the cross validation process is a fundamental phase for the development of this algorithm, we are not able to discuss any result.

Chapter 5

Conclusions and future work

5.1 Concluding remarks

We have presented Big-DAMA, a big data analytics framework specially tailored for network monitoring applications. Using off-the-shelf big data storage and processing engines, Big-DAMA is capable of analyzing and storing big amounts of both structured and unstructured heterogeneous data sources, with batch processing capabilities. We have shown the types of ML-based algorithms implemented in Big-DAMA for network security, using off-the-shelf ML libraries.

By applying Big-DAMA to the detection of different types of network attacks on top of real network measurements collected at the WIDE backbone network, we have also explored novel features to better and faster detecting common network attacks. The analysis of feature selection techniques also showed that it is possible to further reduce execution times by keeping only the most relevant and correlated-to-the-target features. As remarked by our case study analysis, the Random Forest algorithm turned out to be the most suited for NTMA purposes, yielding very good results both with a full features configuration and in the feature selection case.

Despite the promising results coming from the application of the described classification algorithms, the main weakness of such models should be pointed out: the need of a **ground truth**. When dealing with real-time network traffic, in fact, obtaining the true labels for training the model may result particularly difficult, and may impact on the speed at which the stream is classified. One of the most common objections moved to our case study concerns exactly this aspect: the labels use to train the models, even if particularly reliable, come from the application of another kind of classification algorithm, that may in some cases fail. One of the

suggested solutions to overcome this problem may be to investigate the results yielded by the application of some **clustering techniques** on the platform, to avoid the need for a ground truth.
Another important conclusion to be kept in mind is that the model development and application must go together with an adequate knowledge of the underlying environment (i.e. distributed file system, scheduler and other frameworks), in order to efficiently exploit all the tools and computational capabilities at one's disposal and give a correct dimensioning to the problem.

5.2 Future work and improvements

As future work, we plan to focus our research on the development of the streaming platform by adding **Kafka** as a resilient, distributed topic-based service [100] to the cluster. Such messaging system has already been tested on the as a standalone **message broker** supporting the publisher-subscriber paradigm (called respectively *producers* and *consumers* in the Kafka environment), but it must be further integrated with the **Spark Streaming** framework [101] to complete the development of the platform lambda architecture.
Once the data stream warehouse platform is completed, another interesting insight is represented by performing a periodic model evaluation aimed at detecting the **model decay** as time goes by, to properly understand the frequency at which the implemented models should be retrained.
Futhermore, an improvement in the cluster hardware will for sure make the test of the implemented ensamble learner possible on a wider dataset without any issue.

Appendices

Appendix A

Case study code and flow charts

A.1 Data preprocessing

A.1.1 Mawi traces upload

```
from pyspark import SparkConf
from pyspark import SparkContext
conf = SparkConf()
conf.setMaster('yarn-client')
conf.setAppName('anaconda-pyspark')
sc = SparkContext(conf=conf)

print("done with startup")

from __future__ import print_function
from pyspark import SparkContext
from pyspark.sql import Row
from pyspark.sql import SQLContext, HiveContext
from pyspark.sql.types import *
from pyspark.sql import SparkSession
import math
from datetime import datetime
import time
```

```
import subprocess

file_dir = subprocess.Popen(["hadoop", "fs",
"-ls", "/user/big-dama/mawi_traces/extracted"], stdout=subprocess.PIPE)

file_list = []

i = 0
for file_name in file_dir.stdout:
    #print(file_name)
    if i != 0:
        splits = file_name.split("/")
        file_list.append(splits[5].lstrip('\n').strip())
        #print(splits[5].lstrip('\n'))
    i = i+1

def get_table_name(x):
    split = x.tableName.split("a")
    return split[1]
import os

if __name__ == "__main__":
    spark = SparkSession\
        .builder\
        .appName("HiveDBCreation")\
        .getOrCreate()

    #iteration on all the content of the folder!


    sqlContext = SQLContext(sc)

    dblist = sqlContext.sql("show tables from mawi_traces")
    dblist = dblist.select("tableName")
        .rdd.map(lambda x: get_table_name(x)).collect()
```

```
#print(dblist.rdd.count())


file_list = list(set(file_list) - set(dblist))

for file_name in file_list:
    #file_name=file_name.lstrip('\n').strip()
    tablename = "mawi_traces.a"+str(file_name.strip())+"a"

    if file_name not in dblist:
        try:
            path = os.path
            .join("<path to HDFS>"
            , file_name.strip())
            packetdata_rdd = sc.textFile(path)
            df = packetdata_rdd
            .map(lambda l: l.split('\t'))
            .toDF(["timestamp", "frame_len",
            "ip_proto", "ip_len", "ip_ttl", "ip_version",
            "tcp_dstport", "tcp_srcport", "tcp_flags",
            "tcp_flags_ack", "tcp_flags_cwr",
            "tcp_flags_fin", "tcp_flags_ecn",
            "tcp_flags_ns", "tcp_flags_push",
            "tcp_flags_syn", "tcp_flags_urg",
            "tcp_len", "tcp_winsize",
            "udp_srcport", "udp_dstport",
            "ip_src", "ip_dst"])
            print(tablename)
            df.write.saveAsTable(tablename)

        except:
            print(tablename +" not correctly copied!")
```

A.1.2 Mawi features extraction

```
from pyspark import SparkConf
from pyspark import SparkContext
conf = SparkConf()
conf.setMaster('yarn-client')
conf.setAppName('process-matrices')
conf.set('spark.scheduler.mode', 'FAIR')
#conf.set("spark.executor.heartbeatInterval","3600s")
sc = SparkContext(conf=conf)

print("done with startup")

from pyspark import SparkContext
from pyspark.sql import Row
from pyspark.sql import SQLContext, HiveContext, DataFrameWriter
from pyspark.sql.types import *
from pyspark.sql import SparkSession
import math
import Queue
import subprocess
import threading
from datetime import datetime
from pyspark.sql import functions as F
from operator import add

def entropy(assoc, data_size):
    data_size = float(data_size)
    # Reduce the association information to (value, count) pairs
    assign = assoc.map(lambda parti: (parti, 1)).reduceByKey(add)

    # Compute the distribution
    dist = assign.map(lambda (x, y): y/data_size)

    # Compute the entropy of the distribution
    entropy = dist.map(lambda u: -u*math.log(u,2)).reduce(add)
```

```
    entropy = entropy/math.log(assign.count(),2)
    return entropy

def most_used(assoc, data_size):
    data_size = float(data_size)

    # Reduce the association information to (value, count) pairs
    assign = assoc.map(lambda parti: (parti, 1)).reduceByKey(add)


    # Compute the distribution
    dist = assign.map(lambda (x, y): (y/data_size, x)).max()

    return dist

def get_total(assoc, data_size):
    data_size = float(data_size)
    # Reduce the association information to (value, count) pairs
    assign = assoc.map(lambda parti: (parti.ip_proto, 1)).reduceByKey(add)
    return assign.collect()

def get_fraction(assoc, num_TCP, column):

    # Reduce the association information to (value, count) pairs
    assign = assoc.map(lambda parti : parti[column])
    .filter(lambda x: x == 1)

    #print(assign.collect(), column)

    return assign.sum()/num_TCP

def map_IPs_to_index(x):
    print(x)
    splits = x.split('.') #split IP address in its components
    res = float(splits[0])*math.pow(256,3)+float(splits[1])
```

```
    *math.pow(256,2)+float(splits[2])*256+float(splits[3])
    #print(res)

    return res

def get_fraction_ips(assoc, data_size):
    data_size = float(data_size)
    # Reduce the association information to
    #(value, count) pairs
    assign = assoc.map(lambda parti: (parti, 1))
   .reduceByKey(add)

    # Compute the distribution
    dist = assign
    .map(lambda (x, y): (x.ip_version,y/data_size))


    return dist.collectAsMap()

def task_sum(col, col_name):
    start = datetime.now()
    res = col.rdd.map(lambda x: x[col_name]).sum()
    key = "vol"
    dict_to_append[key] = res
    end = datetime.now()

def task_get_min(col, col_name):
    start = datetime.now()
    key = "min_"+col_name
    dict_to_append[key] = col.rdd
    .map(lambda x: x[col_name]).min()
    end = datetime.now()

def task_get_max(col, col_name):
    start = datetime.now()
    key = "max_"+col_name
```

```
    dict_to_append[key] = col.rdd
    .map(lambda x: x[col_name]).max()
    end = datetime.now()

def task_get_avg(col, col_name):
    start = datetime.now()
    key = "avg_"+col_name
    dict_to_append[key] = col.rdd
    .map(lambda x: x[col_name]).mean()
    end = datetime.now()

def task_get_var(col, col_name):
    start = datetime.now()
    key = "var_"+col_name
    dict_to_append[key] = col.rdd.map(lambda x: x[col_name]).variance()
    end = datetime.now()

def task_get_stdev(col, col_name):
    start = datetime.now()
    key = "stdev_"+col_name
    dict_to_append[key] = col.rdd.map(lambda x: x[col_name]).stdev()
    end = datetime.now()

def task_get_percentiles(col, col_name):
    start = datetime.now()
    key = "percentiles_"+col_name

    percentiles = [0.01, 0.02, 0.05, 0.10, 0.15,
    0.20, 0.25, 0.50, 0.75, 0.90, 0.95, 0.97, 0.99]
    res = col.approxQuantile(col_name, percentiles, 0)
    res = map(str, res)
    res = ", ".join(res)

    dict_to_append[key] = "array("+res+")"
    end = datetime.now()
```

```
def task_get_entropy(colrdd, count, name):
    start = datetime.now()
    key = "entropy_"+name

    res = entropy(col.rdd, col.count())

    dict_to_append[key] = res
    end = datetime.now()

def task_countunique(col, column_name):
    start = datetime.now()
    key = "num_"+column_name
    result = col.distinct().count()
    dict_to_append[key] = result
    end = datetime.now()

def task_mu(col, column_name):
    start = datetime.now()
    mu = most_used(col.rdd, col.count())
    end = datetime.now()
    key = "most_used_"+column_name
    key1 = "frac_most_used_"+column_name

    dict_to_append[key] = mu[1][column_name]
    dict_to_append[key1] = mu[0]

def task_fraction(df, column, tcptot):
    start = datetime.now()
    frac = get_fraction(df.rdd, tcptot, column)
    key = "frac_"+column
    dict_to_append[key] = frac
    end = datetime.now()

def task_fraction_ip(datadf):
    start = datetime.now()
    frac = datadf.select("ip_version")
```

```
    frac = frac.withColumn("ip_version",F
    .explode(F.split('ip_version',',')))
    resfr = get_fraction_ips(frac.rdd, frac.count())
    end = datetime.now()
    dict_to_append["frac_ipv4"] = resfr[str(4)]
    dict_to_append["frac_ipv6"] = resfr[str(6)]
import json
import time


if __name__ == "__main__":
    spark = SparkSession\
        .builder\
        .appName("ThreadingFeatureExtraction")\
        .getOrCreate()


    sqlContext = HiveContext(sc)
    kt = sqlContext
    .sql("select * from copied_tables.keep_track")
    .rdd.map(lambda x: x["tablename"]).collect()
    df_list = sqlContext.sql("show tables from mawi_traces")
    .rdd.map(lambda x: x.tableName).sortBy(lambda x: x)
    .filter(lambda x: x not in kt).collect()
    schema_df = sqlContext
   .sql("select * from mawi_schema.schema_ex").schema

    names = []

    for namedf in schema_df:
        names.append(namedf.name)




    get_dist = ["frame_len","ip_proto","ip_len", "ip_ttl",
```

```
 "tcp_dstport", "tcp_srcport", "tcp_len","tcp_winsize",
 "udp_dstport","udp_srcport", "tcp_flags"]
 countunique = ["ip_proto", "tcp_dstport",
 "tcp_srcport", "udp_dstport", "udp_srcport"]
 mostused = ["tcp_dstport", "tcp_srcport",
 "udp_dstport", "udp_srcport"]
 tcpfraction = ["tcp_flags_ack", "tcp_flags_cwr",
"tcp_flags_fin", "tcp_flags_ecn", "tcp_flags_ns",
"tcp_flags_push", "tcp_flags_syn", "tcp_flags_urg"]

 for table_name in df_list:
     start = datetime.now()
     datadf = sqlContext.sql("select * from mawi_traces."
     +table_name)
     end = datetime.now()
     dict_to_append = {}
     #print("dbaccess ", str(end-start))

     for name in get_dist:
         col = datadf.select(name)
         col = col.withColumn(name,F.explode(F.split(name,',')))


         if name == "tcp_flags":
             col = col.rdd.filter(lambda x: x.tcp_flags != "")
            .map(lambda x: int(x.tcp_flags, 16))
             dict_to_append["num_tcp_flags"]
            = col.distinct().count()
             row = Row("tcp_flags")
             col = col.map(row).toDF()
         else:
             col = col.withColumn(name, col[name]
            .cast(DoubleType())).dropna(how="any")
             if name == "frame_len":
                 t1a = threading
                 .Thread(target=task_sum, args=(col, name, ))
```

```
                t1a.start()

            t = threading
           .Thread(target=task_get_min, args=(col, name, ))
            t.start()
            t1 = threading
            .Thread(target=task_get_max, args=(col, name, ))
            t1.start()
            t2 = threading
            .Thread(target=task_get_avg, args=(col, name, ))
            t2.start()
            t3 = threading
            .Thread(target=task_get_var, args=(col, name, ))
            t3.start()
            t4 = threading
            .Thread(target=task_get_stdev, args=(col, name, ))
            t4.start()
            t5 = threading
            .Thread(target=task_get_percentiles, args=(col, name, ))
            t5.start()
            t6 = threading
            .Thread(target=task_get_entropy,
             args=(col.rdd, col.count(), name, ))
            t6.start()

        for name in countunique:
            col = datadf.select(name)
            col = col.withColumn(name,F
            .explode(F.split(name,',')))
            col = col.withColumn(name,
           col[name].cast(DoubleType())).dropna(how="any")

            t = threading.Thread(target=task_countunique,
            args=(col, name, ))
            t.start()
```

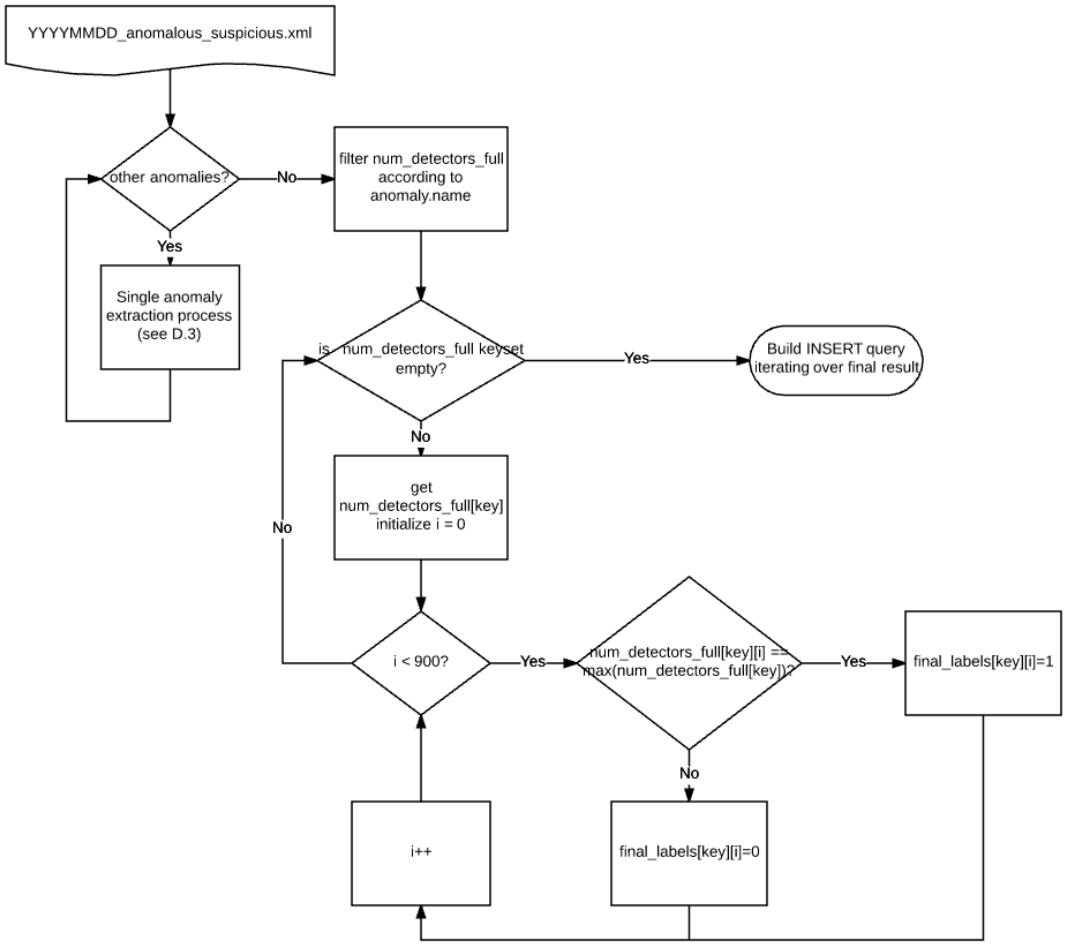
```
    if name != "ip_proto":
        t1 = threading.Thread(target=task_mu,
        args=(col, name, ))
        t1.start()


t3a = threading.Thread(target=task_fraction_ip, args=(datadf, ))
t3a.start()


start_nt = datetime.now()
ipproto = datadf.select('ip_proto')
ipproto = ipproto.withColumn('ip_proto',
F.explode(F.split('ip_proto',',')))
ipproto = ipproto
.withColumn("ip_proto", ipproto["ip_proto"]
.cast(DoubleType())).dropna(how="any")
flen = datadf.select('frame_len')
flen = flen
.withColumn("frame_len", flen["frame_len"]
.cast(DoubleType())).dropna(how="any")
num_pkts = flen.count()
dict_to_append["num_pkts"] = num_pkts
#res = dict(get_total(ipproto.rdd, num_pkts))

rddip = ipproto.rdd.map(lambda x: (x.ip_proto,1)).reduceByKey(add)
rddipfr = rddip.map(lambda (x, y): (x, float(y)/num_pkts))
res = rddip.collectAsMap()
#print(res)
resfr = rddipfr.collectAsMap()

num_TCP_pkts = res[6.0]

dict_to_append["num_icmp_pkts"] = res[1.0]
dict_to_append["num_tcp_pkts"] = res[6.0]
dict_to_append["num_udp_pkts"] = res[17.0]
```

```
dict_to_append["num_gre_pkts"] = res[47.0]

dict_to_append["frac_icmp_pkts"] = resfr[1.0]
dict_to_append["frac_tcp_pkts"] = resfr[6.0]
dict_to_append["frac_udp_pkts"] = resfr[17.0]
dict_to_append["frac_gre_pkts"] = resfr[47.0]


for name in tcpfraction:
    col = datadf.select(name)
    col = col.withColumn(name,F.explode(F.split(name,',')))
    col = col.withColumn(name, col[name]
   .cast(DoubleType())).dropna(how="any")

    t = threading.Thread(target=task_fraction,
   args=(col, name, num_TCP_pkts))
    t.start()

start_ipsrc = datetime.now()
ipproto = datadf.select('ip_src')
ipproto = ipproto.withColumn('ip_src',
F.explode(F.split('ip_src',','))).dropna(how="any")
dict_to_append["num_ip_src"] = ipproto.distinct().count()
indexed = ipproto.rdd.filter(lambda x: x.ip_src != "")
.map(lambda x: map_IPs_to_index(x.ip_src))
most_used_src = most_used(indexed, ipproto.count())

dict_to_append["most_used_ip_src"] = most_used_src[1]
dict_to_append["frac_most_used_ip_src"] = most_used_src[0]

end_ipsrc = datetime.now()

start_ipdst = datetime.now()
ipproto = datadf.select('ip_dst')
ipproto = ipproto.withColumn('ip_dst',
F.explode(F.split('ip_dst',','))).dropna(how="any")
```

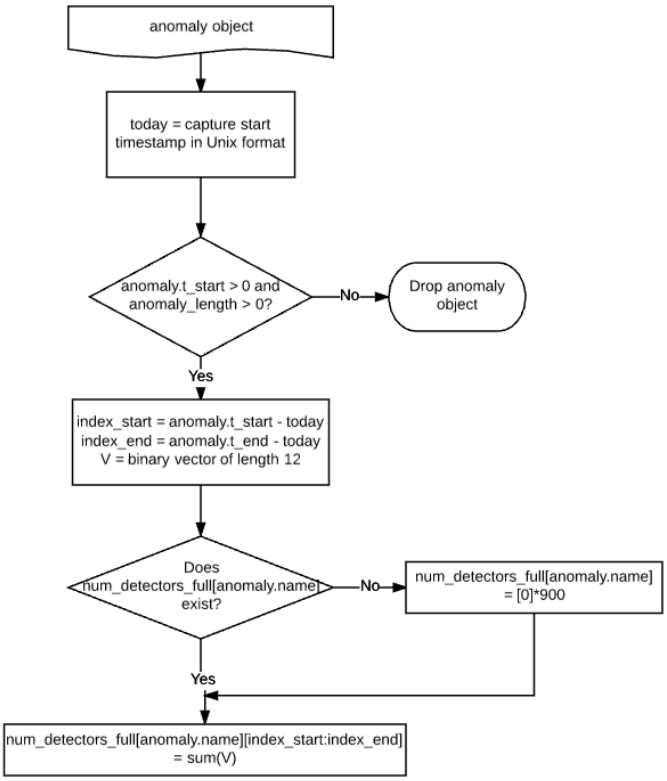
```
  dict_to_append["num_ip_dst"]
 = ipproto.distinct().count() #stampa
  indexed = ipproto.rdd.filter(lambda x: x.ip_dst != "")
 .map(lambda x: map_IPs_to_index(x.ip_dst))

  most_used_dst = most_used(indexed, indexed.count())
  dict_to_append["most_used_ip_dst"] = most_used_dst[1]
  dict_to_append["frac_most_used_ip_dst"]
  = most_used_dst[0]

  split = table_name.split("a")
  res = split[1]
  tablenamef = res[:8]

  query = "insert into table mawi_features.a"
+tablenamef+"a values ("

  first = True

  for n in names:
      if first:
          query = query+" "+str(dict_to_append[n])
          first = False
      else:
          query = query+", "+str(dict_to_append[n])

  query = query+")"

  query_track = "insert into table copied_tables.keep_track
 values ('"+str(table_name)+"')"


  try:
      sqlContext.sql(query)
      sqlContext.sql(query_track)
      print(table_name)
```

```
except:
    sqlContext.sql("insert into table
   copied_tables.keep_track_wrong
   values ('"+str(table_name)+"')")
    print(table_name+" not correctly copied\n")
```

A.1.3 Flow chart for anomaly processing - outer cycle



A.1.4 Flow chart for anomaly processing - inner cycle



A.1.5 Mawi anomalies processing

```
from pyspark import SparkConf
from pyspark import SparkContext
conf = SparkConf()
conf.setMaster('yarn-client')
conf.setAppName('anaconda-pyspark')
sc = SparkContext(conf=conf)
```

```
print("done with startup")

import os
import xml.etree.ElementTree as ET
from datetime import datetime
import time
import numpy as np

directory = "./"
date_dict = {}
for filename in os.listdir(directory):
    if filename.endswith(".xml"):
        splits = filename.split("_")
        datestr = splits[0]
        try:
            tree = ET.parse(filename)
        except:
            print(filename+" file not correctly processable")
        root = tree.getroot()

        for anomaly in root.iter('anomaly'):

            t_start = anomaly.find('from').attrib
            t_start = datetime.fromtimestamp(int(t_start["sec"]))

            if t_start > datetime.fromtimestamp(0):
                date_dict[datestr] = t_start.strftime("\%H")
                break

print(date_dict)

from pyspark.sql import Row
from collections import OrderedDict
from pyspark.sql import SparkSession, DataFrameWriter
import json
```

```
def convert_to_row(d):
    return Row(**OrderedDict(sorted(d.items())))

#Table schema initialization - Data insertion should follow the same schema!
from pyspark.sql import SQLContext, HiveContext
spark = SparkSession(sc)

sqlContext = SQLContext(sc)
hiveCtx = HiveContext(sc)

directory = "./"
for filename in os.listdir(directory):
    if filename.endswith(".xml"):
        splits = filename.split("_")
        datestr = splits[0]

        statement = "CREATE TABLE IF NOT EXISTS mawi_anomalies.
       a"+splits[0]+"a like mawi_anomalies.a20160103a"
        spark.sql(statement)

copied= sqlContext.sql("select * from mawi_anomalies.copied")
copied = copied.rdd.map(lambda x: x.copied).collect()
print(copied)

from operator import add

directory = "./"
for filename in os.listdir(directory):
    if filename.endswith(".xml"):
        splits = filename.split("_")
        datestr = splits[0]
        if datestr not in copied:

            datestr = datestr+str(date_dict[datestr])
```

```
 today = time.mktime(datetime.
strptime(datestr, "\%Y\%m\%d\%H").timetuple())
 print(filename, datestr, today)
 try:
     tree = ET.parse(filename)
 except:
     print(filename+" file not correctly processable")
 root = tree.getroot()
 labels = np.zeros(900)
 #name = {}
 #tax = {}
 #nd = {}
 num_detectors_full = {}

 for anomaly in root.iter('anomaly'):

     vals = anomaly.attrib
     label_name = vals["type"]
     info = vals["value"].split(",")

     dn = float(info[0])
     da = float(info[1])
     tax_1 = float(info[2])
     tax_2 = info[4]

     detectors = info[3].split(" ")
     detectors = map(int, detectors)

     t_start = anomaly.find('from').attrib
     t_end = anomaly.find('to').attrib
     t_start = float(t_start["sec"])
     t_end = float(t_end["sec"])


     index_aux_start = t_start - today
     index_aux_end = t_end - today
```

```
        index_aux_start = int(max(index_aux_start, 1))
        index_aux_end = int(min(index_aux_end, 900))
        anomaly_length = t_end-t_start


        if t_start > 0:
            labels[index_aux_start: index_aux_end] = 1
            s = sum(detectors)

        if t_start > 0 and anomaly_length > 0:

            if tax_2 not in num_detectors_full:
                num_detectors_full[tax_2] = [0]*900

            d_full = [0]*4
            for k in range(4):
                ind1 = k*3
                ind2 = (k+1)*3-1
                aux = sum(detectors[ind1:ind2])

                if aux > 0:
                    d_full[k] = 1


            for jj in range(index_aux_start, index_aux_end):
                #name[i][jj] = tax_2
                #tax[i][jj] = tax_1
                num_detectors_full[tax_2][jj] = sum(d_full)
                #nd[tax_2][jj] = s


final_labels = {}
keyset_detfull = num_detectors_full.keys()

print(keyset_detfull)
```

```
#EASY CASE - AGGREGATION OF MULTIPLE ROWS IS NOT REQUIRED
# - catch exception for key not present
find_anomalies =
[item for item in keyset_detfull if (item == ("ntscACK"))]
final_labels["ntscACK"] = [0]*900

for key in find_anomalies:
    maxval = max(num_detectors_full[key])
    final_labels[key]
    = [1 if x == maxval else 0 for x in num_detectors_full[key]]

#Possible entries for mptp - flashcrowd
#(the prefixes related to the anomaly are
#reported on the MAWILab doc page)
keyset_mptp = [item for item in keyset_detfull
if (item.find("mptp") != -1)]
final_labels["mptp"] = [0]*900
for key in keyset_mptp:

    final_labels["mptp"] = map(add,
   final_labels["mptp"], num_detectors_full[key])
 #all related anomalies are aggregated into
 # a single row for which we search the maximum

maxval = max(final_labels["mptp"])
final_labels["mptp"] = [1 if (x == maxval
and maxval != 0)
else 0 for x in final_labels["mptp"]]

#Possible entries for netscan_UDP
#(the prefixes related to the anomaly
#are reported on the MAWILab doc page)
keyset_udpscan = [item for item in
num_detectors_full.keys()
if (item.find("ntscUDP") != -1 or
item.find("ptpposcaUDP") != -1)]
```

```
final_labels["netscan_UDP"] = [0]*900
for key in keyset_udpscan:

    final_labels["netscan_UDP"] =
   map(add, final_labels["netscan_UDP"],
   num_detectors_full[key])

maxval = max(final_labels["netscan_UDP"])
final_labels["netscan_UDP"] =
[1 if (x == maxval and maxval != 0)
else 0 for x in final_labels["netscan_UDP"]]

#Possible entries for DDoS
keyset_DDoS = [item for item
in num_detectors_full.keys()
if (item.find("distributed_dos") != -1 or
item.find("DDoS") != -1
and item.find("DDosIC") == -1)]
final_labels["DDoS"] = [0]*900
for key in keyset_DDoS:

    final_labels["DDoS"] = map(add, final_labels["DDoS"],
    num_detectors_full[key])

maxval = max(final_labels["DDoS"])
final_labels["DDoS"] = [1 if (x == maxval and maxval != 0)
else 0 for x in final_labels["DDoS"]]

#Possible entries for ping flooding
keyset_pingfl = [item for item in num_detectors_full.keys()
 if (item.find("DDoSIC") != -1)]
final_labels["ping_flood"] = [0]*900
for key in keyset_pingfl:

    final_labels["ping_flood"] = map(add,
    final_labels["ping_flood"],
```

```
            num_detectors_full[key])

        maxval = max(final_labels["ping_flood"])
        final_labels["ping_flood"] =
        [1 if (x == maxval and maxval != 0)
        else 0 for x in final_labels["ping_flood"]]


        tosort = final_labels.keys()

        for iii in range(900):
            first = True
            query = "insert into table
           mawi_anomalies.a"+splits[0]+"a values ("
            for key in sorted(tosort):
                if first:
                    query = query+" "+str(iii)+",
                    "+str(final_labels[key][iii])
                    first = False
                else:
                    query = query+", "+str(final_labels[key][iii])

            query = query+")"
            sqlContext.sql(query)
            print(query)

        query_track = "insert into table
       mawi_anomalies.copied values ("+splits[0]+")"
        sqlContext.sql(query_track)
    copied.append(splits[0])
```

A.2 Modelling and Analysis

A.2.1 Decision Tree

```
dt = DecisionTreeClassifier(labelCol="label",
featuresCol="features", maxDepth=3)

    # Train model with Training Data
dtModel = dt.fit(trainingData)

print "numNodes = ", dtModel.numNodes
print "depth = ", dtModel.depth

predictions = dtModel.transform(testData)
    #predictions.printSchema()

selected = predictions.select("label",
"prediction", "probability")
selected.show()
```

A.2.2 Random Forest

```
# Create an initial RandomForest model.
rf = RandomForestClassifier(labelCol="label",
featuresCol="features")

# Train model with Training Data
rfModel = rf.fit(trainingData)

predictions = rfModel.transform(testData)
selected = predictions.select("label", "prediction",
"probability")
```

A.2.3 Naive Bayes

```
nb = NaiveBayes(labelCol="label", featuresCol="features")
# Train model with Training Data
nbModel = nb.fit(trainingData)
predictions = nbModel.transform(testData)
#predictions.printSchema()
selected = predictions.select("label", "prediction",
"probability")
#selected.show()
```

A.2.4 Multilayer Perceptron

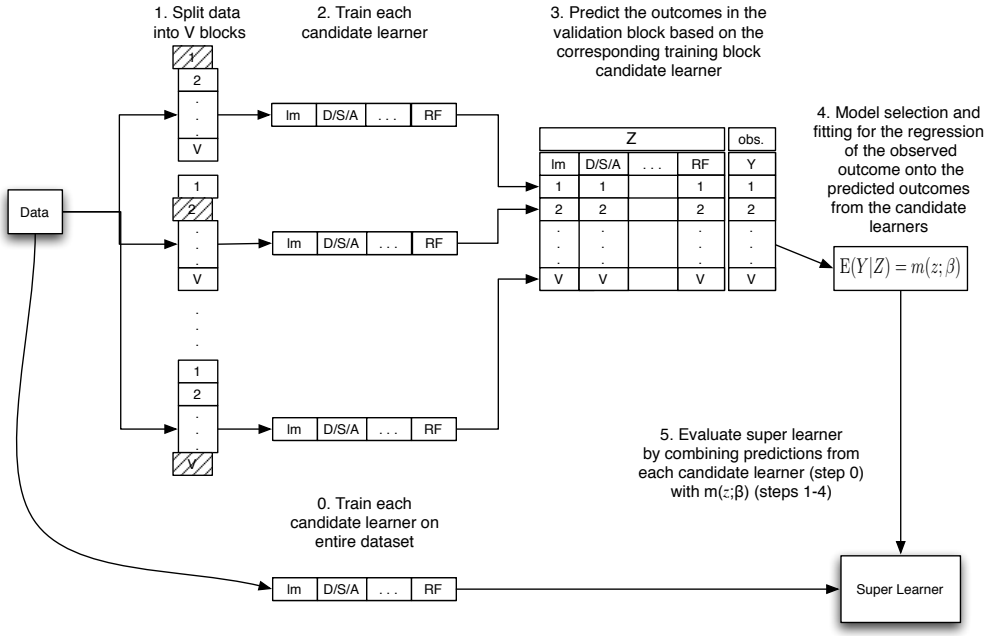
```
mlp = MultilayerPerceptronClassifier(labelCol="label",
featuresCol="features", layers=[2, 2, 2])

# Train model with Training Data
mlpModel = mlp.fit(trainingData)

#mlpModel.explainParams()

predictions = mlpModel.transform(testData)
predictions.printSchema()
```

A.2.5 Cross Validation

```
from pyspark.ml.evaluation import BinaryClassificationEvaluator

# Evaluate model
evaluator = BinaryClassificationEvaluator()
```

```
evaluator.evaluate(predictions)

# Create ParamGrid for Cross Validation
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator

paramGrid = (ParamGridBuilder()
             .addGrid(dt.maxDepth, [1,2])
             .build())

# Create 10-fold CrossValidator
cv = CrossValidator(estimator=dt,
estimatorParamMaps=paramGrid,
evaluator=evaluator, numFolds=10)

# Run cross validations
cvModel = cv.fit(trainingData)
# Takes ~5 minutes

predictions = cvModel.transform(testData)
```

A.2.6 Ensemble Learning (from [1])



Bibliography

[1] M. Van der Laan, et al., "Super learner", in Statistical applications in genetics and molecular biology, vol. 6, no. 1, 2007.

[2] P. Casas, et al., "Big-DAMA: Big Data Analytics for Network Traffic Monitoring and Analysis", in *ACM SIGCOMM LANCOMM Workshop*, 2016.

[3] P. Casas, et al., "POSTER:(Semi)-Supervised Machine Learning Approaches for Network Security in High-Dimensional Network Data", in *ACM CCS*, 2016.

[4] P. Casas, et al., "Machine-learning based approaches for anomaly detection and classification in cellular networks", in *TMA*, 2016.

[5] Y. Freund, et al., "Using and combining predictors that specialize", in *ACM STOC*, 1997.

[6] J. Hansen, "Combining predictors: Some old methods and a new method", available online at Citeseer, 1998.

[7] T. Dietterich, "Ensemble learning", The handbook of brain theory and neural networks, vol. 2, pp. 110–125, MIT Press: Cambridge, MA, 2002.

[8] P. Sollich, A. Krogh, "Learning with ensembles: How overfitting can be useful", Advances in neural information processing systems, pp. 190–196, MORGAN KAUFMANN PUBLISHERS, 1996.

[9] R. Fontugne, et al., "MAWILab: Combining Diverse Anomaly Detectors for Automated Anomaly Labeling and Performance Benchmarking", in *ACM CoNEXT*, 2010

[10] T. Nguyen, G. Armitage, "A survey of techniques for Internet Traffic Classification using Machine Learning", in IEEE Comm. Surv. & Tut., vol. 10, no. 4, pp. 56–76, 2008.

[11] A. Ghosh, A. Schwartzbard, "A Study in Using Neural Networks for Anomaly and Misuse Detection", in *USENIX Security Symposium*, 1999.

[12] A. Mitrokotsa et al., “Detecting denial of service attacks using emergent self-organizing maps”, in *IEEE ISSPIT*, 2005.

[13] M. Ostaszewski et al., “A non-self space approach to network anomaly detection”, in *IEEE IPDPS*, 2006.

[14] W. Chimphlee, et al., “Integrating genetic algorithms and fuzzy C-means for anomaly detection”, in *IEEE Indicon*, 2005.

[15] G.Prashanth, et al., “Using random forests for network-based anomaly detection”, in *IEEE ICSCN*, 2008.

[16] Y. Li et al., “An efficient network anomaly detection scheme based on TCM-KNN algorithm and data reduction mechanism”, in *IAW*, 2007.

[17] P. Casas et al., “Unsupervised Network Intrusion Detection Systems: Detecting the Unknown without Knowledge”, in *Computer Communications*, vol. 35 (7), pp. 772-783, 2011.

[18] T. Ahmed, et al., “Machine Learning Approaches to Network Anomaly Detection”, in *USENIX SYSML Workshop*, 2007.

[19] V. Chandola, et al., “Anomaly detection: A survey”, ACM Comput. Surv., vol. 41, no. 3, pp. 1–58, 2009.

[20] M. Ahmed, et al., “A Survey of Network Anomaly Detection Techniques”, J. Netw. Comput. Appl., vol. 60, pp. 19–31, 2016.

[21] W. Zhang, et al., “A Survey of Anomaly Detection Methods in Networks”, in *CNMT Symposium*, 2009.

[22] A. Moore et al., “Internet Traffic Classification using Bayesian Analysis Techniques”, in *Proc. ACM SIGMETICS*, 2005.

[23] M. Roughan et al., “Class-of-Service Mapping for QoS: a Statistical Signature-Based Approach to IP Traffic Classification”, in *IMW*, 2004.

[24] N. Williams el al., “A Preliminary Performance Comparison of Five Machine Learning Algorithms for Practical IP Traffic Flow Classification”, in *ACM CCR*, vol. 36 (5), pp. 5-16, 2006.

[25] S. Valenti et al., “Accurate, Fine-Grained Classification of P2P-TV Applications by Simply Counting Packets”, in *TMA*, 2009.

[26] J. Erman et al., “Traffic Classification using Clustering Algorithms”, in *MineNet*, 2006.

[27] P. Casas et al., “MINETRAC: Mining Flows for Unsupervised Analysis & Semi-Supervised Classification”, in *ITC*, 2011.

[28] J. Erman et al., “Semi-Supervised Network Traffic Classification”, in *ACM SIGMETRICS*, 2007.

[29] T. Nguyen et al., “A Survey of Techniques for Internet Traffic Classification using Machine Learning”, in *IEEE Comm, Surv. & Tut.*, vol. 10 (4), pp. 56-76, 2008.

[30] R. Ravinder, et al., “Real Time Anomaly Detection Using Ensembles”, in *ICISA International Conference*, 2014.

[31] M. Ozdemir, I. Sogukpinar, “An Android Malware Detection Architecture based on Ensemble Learning”, in Transactions on Machine Learning and Artificial Intelligence, vol. 2, no. 3, pp. 90–106, 2014.

[32] A. Baer, A. Finamore, P. Casas, L. Golab, M. Mellia, “Large-Scale Network Traffic Monitoring with DBStream, a System for Rolling Big Data Analysis,” in *Proc. IEEE International Conference on Big Data*, 2014.

[33] M. Stonebraker, “SQL Databases vs. noSQL Databases,” in *Communications of the ACM*, vol. 53(4), pp. 10-11, 2010.

[34] C. Cranor, T. Johnson, O. Spataschek, V. Shkapenyuk, “Gigascope: A Stream Database for Network Applications,” in *Proc. of the ACM SIGMOD International Conference on Management of Data*, pp. 647-651, 2003.

[35] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, S. Zdonik, “Aurora: A New Model and Architecture for Data Stream Management,” in *The VLDB Journal*, vol. 12(2), pp. 1020-1039, 2003.

[36] L. Golab, T. Johnson, J. Seidel, V. Shkapenyuk, “Stream Warehousing with DataDepot,” in *Proc. of the ACM SIGMOD International Conference on Management of Data*, pp. 847-854, 2009.

[37] P. Bhatotia et al., “Indoop: Mapreduce for Incremental Computations,” in *Proc. of the ACM Symposium on Cloud Computing*, pp. 7-14, 2011.

[38] , W. Lam, L. Liu, S. Prasad, A. Rajaraman, Z. Vacheri, A. Doan, "Muppet: Mapreduce-style processing of fast data," in *Proc. VLDB Endow.*, vol. 5(12), pp.1814-1825, 2012.

[39] B. Li, E. Mazur, Y. Diao, A. McGregor, P. Shenoy, "Scalla: A platform for scalable one-pass analytics using mapreduce," in *ACM Trans. Database Syst.* 37(4), pp. 27-43, 2012.

[40] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Communications of the ACM*, vol. 51(1), pp. 107-113, 2008.

[41] T. White, "Hadoop: the Definitive Guide," *O'Reilly Media, Inc.*, ISBN:0596521979 9780596521974, 2009.

[42] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, I. Stoica, "Spark: Cluster Computing with Working Sets," in *Proc. of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, pp. 10-16, 2010.

[43] M. Zaharia, T. Das, H. Li, S. Shenker, I. Stoica, "Discretized Streams: An Efficient and Fault-tolerant Model for Stream Processing on Large Clusters," in *Proc. of the 4th USENIX Conference on Hot Topics in Cloud Computing*, pp. 10-16, 2012.

[44] P. Casas, J. Mazel, P. Owezarski, "Knowledge-Independent Traffic Monitoring: Unsupervised Detection of Network Attacks," in *IEEE Network Magazine*, vol. 26(1), pp. 13-21, 2012.

[45] Berkeley AMPLab. Big Data Benchmark. https://amplab.cs.berkeley.edu/benchmark/, 2014.

[46] C. Bishop, "Pattern Recognition and Machine Learning", 2007.

[47] Z. Cai, Z. Gao, S. Luo, L. Perez, Z. Vagena, C. Jermaine, "A Comparison of Platforms for Implementing and Running Very Large Scale Machine Learning Algorithms," in *Proc. of the ACM SIGMOD International Conference on Management of Data*, pp. 1371-1382, 2014.

[48] F. Huici, A. di Pietro, B. Trammell, J. Gomez Hidalgo, D. Martinez Ruiz, N. d'Heureuse, "Blockmon: A High-Performance Composable Network Traffic Measurement System," in *Proc. of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pp. 79-80, 2012.

[49] A. Sperotto, G. Schaffrath, R. Sadre, C. Morariu, A. Pras, B. Stiller, "An Overview of IP Flow-Based Intrusion Detection," in *IEEE Communications Surveys Tutorials*, vol. 12(3), pp. 343-356, 2010.

[50] M. Panda, A. Abraham, S. Das, M.R. Patra, "Network Intrusion Detection Systems: A Machine Learning Approach," in *Int. Decision Technologies*, vol. 5 (4), pp. 347-356, 2011.

[51] I. Syarif, A. Prugel-Bennett, G. Wills, "Data Mining Approaches for Network Intrusion Detection: from Dimensionality Reduction to Misuse and Anomaly Detection," in *Journal of Information Technology Review*, vol. 3(2), pp. 70-83, 2012.

[52] L. Khan, M. Awad, B. Thuraisingham, "A New Intrusion Detection System Using Support Vector Machines and Hierarchical Clustering," in *The VLDB Journal*, vol. 16(4), pp. 507-521, 2007.

[53] A. Marnerides, A. Schaeffer-Filho, A. Mauthe, "Traffic Anomaly Diagnosis in Internet Backbone Networks: A Survey," in *Computer Networks*, vol. 73, pp. 224-243, 2014.

[54] R. Sommer and V. Paxson, "Outside the Closed World: On Using Machine Learning for Network Intrusion Detection," in *Proc. IEEE Symposium on Security and Privacy*, pp. 305-316, 2010.

[55] M. Tavallaee, N. Stakhanova, A.A. Ghorbani, "Toward Credible Evaluation of Anomaly-Based Intrusion-Detection Methods," in *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, vol. 40(5), pp. 516-524, 2010.

[56] F. Iglesias and T. Zseby, "Analysis of Network Traffic Features for Anomaly Detection," in *Machine Learning*, pp. 1-26, 2014.

[57] S. Sagiroglu and D. Sinanc, "Big Data: A Review," in *Proc. Int. Conf. on Collaboration Technologies and Systems*, pp. 42-47, 2013.

[58] H. Kriegel, P. Kroger, A. Zimek, "Clustering high-dimensional data: A survey on subspace clustering, pattern-based clustering, and correlation clustering," in *ACM Transactions on Knowledge Discovery from Data*, 2009.

[59] L. Parsons, E. Haque, H. Liu, "Subspace Clustering for High Dimensional data: a Review,", in *SIGKDD Explor. Newsl.*, 2004.

[60] W. Chen et al., "Parallel Spectral Clustering in Distributed Systems Pattern Analysis and Machine Intelligence," in *IEEE Transactions*, vol 33(3), pp. 568-586, 2011.

[61] P. Domingos et al., "Mining High-Speed Data Streams," in *Proc. of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 71-80, 2000.

[62] P. Biswanath, J. Herbach , S. Basu , R. Bayardo, "PLANET: Massively Parallel Learning of Tree Ensembles with MapReduce", in *Proc. VLDB Endow.*, pp. 1426-1437, 2009.

[63] C. Aggarwal et al., "A Framework for Clustering Evolving Data Streams," in *Proc. of the International Conference on Very Large Data Bases*, pp. 81-92, 2003.

[64] P. Costa, A. Donnelly, A. Rowstron, G. O'Shea, "Camdoop: Exploiting In-network Aggregation for Big Data Applications," in *Proc. of the 9th USENIX Symposium on Networked Systems Design and Implementation*, pp. 29-42, 2012.

[65] R. Fontugne, J. Mazel, K. Fukuda, "Hashdoop: A MapReduce Framework for Network Anomaly Detection," in *Proc. of the IEEE Conference on Computer Communications Workshops*, pp. 494-499, 2014.

[66] Y. Lee et al., "Toward scalable internet traffic measurement and analysis with Hadoop," in SIGCOMM Comput. Commun. Rev., 43(1), pp. 5-13, 2012.

[67] J. Liu, F. Liu, N. Ansari, "Monitoring and analyzing big traffic data of a large-scale cellular network with Hadoop," in IEEE Network, 28(4), pp. 32-39, 2014.

[68] A. Pavlo, E. Paulson, A. Rasin, D. Abadi, D. Dewitt, S. Madden, M. Stonebraker, "A Comparison of Approaches to Large-Scale Data Analysis," in *Proc. of the ACM SIGMOD International Conference on Management of Data*, pp. 165-178, 2009.

[69] S. Huang, J. Huang, J. Dai, T. Xie, B. Huang, "The HiBench Benchmark Suite: Characterization of the MapReduce-based Data Analysis," in *Proc. of the IEEE SIGMOD International Conference on Data Engineering Workshops*, pp. 41-51, 2010.

[70] Y. Chen, A. Ganapathi, R. Griffith, R. Katz, "The Case for Evaluating MapReduce Performance Using Workload Suites," in *Proc. of the IEEE Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*, pp. 25-27, 2011.

[71] Y. Jia, "A Benchmark for Hive, PIG and Hadoop," available at https://issues.apache.org/jira/browse/hive-396

[72] C. Douglas, H. Tang, "Gridmix3 – Emulating Production Workload for Apache Hadoop," available at https://developer.yahoo.com/blogs/hadoop/gridmix3-emulating-production-workload-apache-hadoop-450.html

[73] H. Karloff, S. Suri, S. Vassilvitskii, "A Model of Computation for MapReduce," in *Proc. ACM SODA*, 2010.

[74] Apache Mahout, http://mahout.apache.org.

[75] MLlib: https://spark.apache.org/mllib/

[76] B. Babcock, S. Babu, M. Datar, R. Motwani, J. Widom, "Models and Issues in Data Stream Systems," in *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pp. 1-16, 2002.

[77] L. Golab, T. Johnson "Consistency in Stream Warehouse," in *CIDR Vol.11*, pp. 114-122, 2011.

[78] Apache Spark ML, http://spark.apache.org/docs/latest/api/python/pyspark.ml.html.

[79] Scikitlearn Ensemble methods, http://scikit-learn.org/stable/modules/ensemble.html.

[80] J. Gantz, D. Reinsel, "Extracting value from chaos." IDC iview 1142.2011 (2011): 1-12.

[81] B. Brown, et al., "Big data: the next frontier for innovation, competition, and productivity." McKinsey Global Institute (2011).

[82] M. Cooper, P. Mell, "Tackling big data." Websitehttp://csrc.nist.gov/groups/SMA/forum/documents/june2012presentations/csm_june2012_cooper_mell.pdf, 2012.

[83] Council, Transaction Processing Performance. "Transaction processing performance council." Web Site, http://www. tpc. org (2005).

[84] Y. Chen, "We dont know enough to make a big data benchmark suite-an academia-industry view." Proc. of WBDB (2012).

[85] J. G. Gantz and D. R. Reinsel. "Extracting value from chaos".

[86] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. H. Byers. "Big data: The next frontier for innovation, competition, and productivity", May 2011.

[87] Cloud Security Alliance. "Big Data Taxonomy", September 2014.

[88] Cloudera Manager, https://www.cloudera.com/products/product-components/cloudera-manager.html.

[89] D. Borthakur. "HDFS architecture guide. Hadoop Apache Project, 53", 2008.

[90] Apache Hadoop YARN, https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[91] Cloudera YARN Container tuning guide https://www.cloudera.com/documentation/enterprise/5-3-x/images/yarn_mapreduce_tasks.jpg.

[92] Hortonworks, Inc. Hortonworks Data Platform: Apache Spark Component Guide, 2012-2017

[93] Apache Spark 2 API, https://spark.apache.org/docs/2.2.0/ml-pipeline.html.

[94] Apache Hive, https://hive.apache.org/.

[95] Zookeeper, https://cwiki.apache.org/confluence/display/ZOOKEEPER/Index.

[96] Oozie, http://oozie.apache.org/.

[97] MAWI MIDE, http://mawi.wide.ad.jp/mawi/.

[98] J. Vanerio, P. Casas, "Ensemble-learning Approaches for Network Security and Anomaly Detection", 2017.

[99] P. Casas et al., "Network Security and Anomaly Detection with Big-DAMA, a Big Data Analytics Framework", in *IEEE Cloudnet 2017*.

[100] Kafka, https://kafka.apache.org/documentation/.

[101] Spark Streaming, https://spark.apache.org/streaming/.

[102] R. Kohavi, H. J. George H. John, "Wrappers for feature subset selection." Artificial intelligence 97.1-2 (1997): 273-324.

[103] M. A. Hall. "Correlation-based feature selection for machine learning." (1999).