BRNO UNIVERSITY OF TECHNOLOGY

Faculty of Electrical Engineering
and Communication

MASTER'S THESIS

Brno, 2020 Bc. Miroslav Šiklóši

BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF TELECOMMUNICATIONS

ÚSTAV TELEKOMUNIKACÍ

SYSTEM LOG ANALYSIS FOR ANOMALY DETECTION USING MACHINE LEARNING

VYUŽITÍ STROJOVÉHO UČENÍ PRO DETEKCI ANOMÁLIÍ NA ZÁKLADĚ ANALÝZY SYSTÉMOVÝCH LOGŮ

MASTER'S THESIS

DIPLOMOVÁ PRÁCE

AUTHOR AUTOR PRÁCE	Bc. Miroslav Šiklóši
SUPERVISOR VEDOUCÍ PRÁCE	doc. Ing. Jiří Hošek, Ph.D.

BRNO 2020

BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

Master's Thesis

Master's study field **Communications and Informatics**

Department of Telecommunications

Student:	Bc. Miroslav Šiklóši	***ID:***	164414
Year of study:	2	***Academic year:***	2019/20

TITLE OF THESIS:

System Log Analysis for Anomaly Detection Using Machine Learning

INSTRUCTION:

The thesis deals with implementing an anomaly detection tool that can analyze the system log utilizing machine learning methods. First, student needs to compare between traditional way of log analysis and the machine learning one. Moreover, student should illustrate in detail the machine learning methodology used in the system log analysis. The proposed tool should be able to automatically learn log patterns and decide if new log is considered as anomalous or not. The tool should be able also to handle specific conditions, for example, system log is considered as anomalous if its pattern is more likely belong to anomalous area, plus the anomalous situation remains for some specific time.

RECOMMENDED LITERATURE:

[1] Shai Shalev-Shwartz, Shai Ben-David, Understanding Machine Learning: From Theory to Algorithms, ISBN:1107057132, 9781107057135, Pages: 397, Year: 2014.

[2] Adrian Mouat, Using Docker: Developing and Deploying Software with Containers, ISBN:1491915927, 9781491915929, Pages: 35, Year: 2015.

Date of project specification:	3.2.2020	***Deadline for submission:***	1.6.2020

Supervisor: doc. Ing. Jiří Hošek, Ph.D.
Consultant: Ing. Nabhan Khatib, Ph.D.

prof. Ing. Jiří Mišurec, CSc.
Subject Council chairman

ABSTRACT

This thesis deals with system log analysis for anomaly detection using machine learning models. The proposed models are based on supervised, unsupervised and deep learning algorithms. However, the functionality and behaviour of these algorithms have been clarified theoretically and practically in the thesis. Moreover, many preprocessing methods and logics were used to preprocess the data before it was fed to the machine learning model. At the end and to confirm the workability of proposed models, many metrics were calculated and unseen syslogs were fed to the best-proposed machine learning models to detect the anomalies. However, models Decision Tree Classifier, One-class SVM and Hierarchical Clustering demonstrated the best performance, correctly predicting 93.95%, 85.66% and 85.3% of all anomalies respectively.

KEYWORDS

Anomaly detection, Syslog messages, Python, Machine Learning

ABSTRAKT

Táto diplomová práca sa venuje problematike využitia strojového učenia na detekciu anomálií na základe analýzy systémových logov. Navrhnuté modely sú založené na algoritmoch strojového učenia s učiteľom, bez učiteľa a na hlbokom učení. Funkčnosť a správanie týchto algoritmov sú objasnené ako teoreticky, tak aj prakticky. Okrem toho boli využité metódy a postupy na predspracovanie dát predtým, než boli vložené do modelov strojového učenia. Navrhnuté modely sú na konci porovnané s využitím viacerých metrík a otestované na syslogoch, ktoré modely predtým nevideli. Najpresnejší výkon podali modely Klasifikátor rozhodovacích stromov, Jednotriedny podporný vektorový stroj a model Hierarchické zoskupovanie, ktoré správne označili 93,95%, 85,66% a 85,3% anomálií v uvedenom poradí.

KĽÚČOVÉ SLOVÁ

Detekcia anomálií, Systémové logy, Python, Strojové učenie

ŠIKLÓŠI, Miroslav. *System Log Analysis for Anomaly Detection Using Machine Learning*. Brno, 2020, 83 p. Master's Thesis. Brno University of Technology, Faculty of Electrical Engineering and Communication, Department of Telecommunications. Advised by doc. Ing. Jiří Hošek, Ph.D. and Ing. Nabhan Khatib, Ph.D.

ROZŠÍRENÝ ABSTRAKT

Diplomová práca sa zaoberá problematikou využitia strojového učenia na detekciu anomálií na základe analýzy systémových logov. Súčasný rast počítačových sietí distribuovaných systémov je časovo veľmi náročný na monitorovanie. Všetky zariadenia nonstop zasielajú stavové správy, tzv. systémové správy. Napriek tomu, že sú tieto správy veľmi dôležité, je veľmi náročné a neefektívne ich manuálne monitorovať a analyzovať. K zjednodušeniu spracovávania týchto logov existuje niekoľko desiatok rôznych programov, či už platených alebo open-source.

Tieto programy sú ale založené na tradičných metódach, ktoré často vyžadujú mnoho pravidelných úprav, aby boli schopné držať krok a analyzovať stále aktualizované správy. Využitie strojového učenie však môže zabezpečiť lepšiu robustnosť a zlepšiť schopnosti týchto programov zvládať spomenuté problémy.

Strojové učenie je vetva umelej inteligencie a umožňuje počítačom učiť sa bez toho aby k tomu boli explicitne naprogramované. K tomu využívajú algoritmy z rôznych oblastí ako napríklad štatistika, teória pravdepodobnosti a lineárna algebra. Strojové učenie využíva tieto poznatky k analýze historických dát a následne k predpovedaniu podobných situácií.

Algoritmy strojového učenia sú v tejto práci využité na detekciu anomálií, ktoré môžu znamenať bezpečnostné riziko. Anomália znamená odchýlku od normálne stavu, neočakávaný stav.

Táto práca aplikuje algoritmy na detekciu anomálií v sieťových paketoch a v systémových správach. Na detekcii sú využité viaceré metódy strojového učenia s učiteľom a bez učiteľa. Okrem toho je využitý aj algoritmus pre hlboké učenie.

Na začiatku práce je teoretická časť, obsahujúca dve kapitoly. Kapitola 1 popisuje teoretické a matematické základy všetkých metód strojového učenia a rozdiely medzi nimi. Kapitola 2 predstavuje teoretický základ predprípravy dát a spracovania prirodzeného jazyka.

Nasleduje praktická časť práce. Kapitola 3 popisuje praktickú implementáciu generátora systémových správ, jeho funkčnosť a využitie. Kapitola 4 popisuje logickú štruktúru a implementáciu navrhnutých súčastí na spracovanie dát a modelov strojového učenia. Navrhnuté programy a ich funkčnosť sú popísané v kapitole 5, spolu s ukážkou ich spustenia.

Na konci praktickej časti práce sa nachádza kapitola 6, ktorá navzájom porovnáva jednotlivé navrhnuté modely strojového učenia. Tieto porovnania sú zobrazené ako v príslušných tabuľkách, tak aj v graficky. Nachádza sa tu tiež krátka ukážka využitia navrhnutých programov v reálnych situáciách.

Navrhnuté programy boli naprogramované v jazyku *Python*. Modely strojového učenia s učiteľom a bez učiteľa využívajú voľne prístupnú knižnicu *Scikit-learn*. Model

hlbokého učenia využíva taktiež voľne prístupnú knižnicu *Keras.*

Jednotlivé modely boli najskor natrénované na označených dátach a následne otestované na dátach, aké dovtedy nemali k dispozícii. Tým došlo k overeniu nielen ich presnosti pri detekcii anomálií, ale aj k overeniu robustnosti v prípade, že vkladané dáta sa budú meniť. Pri testovaní bol taktiež porovnaný vplyv množstva trénovaných dát na výslednú presnosť jednotlivých modelov strojového učenia.

Ako najpresnejší model sa ukázal *Klasifikátor rozhodovacích stromov*, ktorý bol schopný správne označiť až takmer 94% anomálií. Veľmi tesne za ním sa nachádzali modely *Jednotriedneho podporného vektorového stroja* a *Hierarchického zoskupovania.*

Tieto výsledky boli dosiahnuté vďaka manuálnej úprave ich parametrov, na základe predchádzajúcich znalostí a náhodných testov. Jednotlivé modely sa môžu v budúcnosti vylepšiť využitím automatizácie úpravy týchto parametrov. Jednou z možností je využitie tzv. *Algoritmu mriežkového vyhľadávania*, ktorý využíva preddefinované mriežky parametrov a hrubou silou navzájom porovnáva ich presnosť a výkonnosť. Jedná sa ale o veľmi zdĺhavý a náročný proces, ktorý musí byť navyše aplikovaný na každý model strojového učenia samostatne.

Ďalšou možnosťou môže byť inšpirovanie sa programom *H20 AutoML*, ktorý automatizuje vytváranie a vzájomné porovnávanie väčšieho množstva modelov.

DECLARATION

I declare that I have written the Master's Thesis titled "System Log Analysis for Anomaly Detection Using Machine Learning" independently, under the guidance of the advisor and using exclusively the technical references and other sources of information cited in the thesis and listed in the comprehensive bibliography at the end of the thesis.

As the author I furthermore declare that, with respect to the creation of this Master's Thesis, I have not infringed any copyright or violated anyone's personal and/or ownership rights. In this context, I am fully aware of the consequences of breaking Regulation § 11 of the Copyright Act No. 121/2000 Coll. of the Czech Republic, as amended, and of any breach of rights related to intellectual property or introduced within amendments to relevant Acts such as the Intellectual Property Act or the Criminal Code, Act No. 40/2009 Coll., Section 2, Head VI, Part 4.

Brno . . . .

author's signature

ACKNOWLEDGEMENT

I want to thank my supervisor doc. Ing. Jiří Hošek, Ph.D. and consultant Ing. Nabhan Khatib, Ph.D. from AT&T Global Network Services Czech Republic s.r.o. for providing valuable knowledge, guidance, patience and helpful suggestions. I would like to extend my sincere thanks to my parents for moral and financial support throughout the studies.

This thesis was created as part of the key activity KA6 - Individual teaching and involvement of students of bachelor's and master's degree programs in research within the project OP RDE Creation of a double-degree doctoral study program Creating double-degree doctoral study program Electronics and Information Technology and creating doctoral study program Information Security reg. no. CZ.02.2.69/0.0/0.0/16_018/0002575.

EUROPEAN UNION
European Structural and Investment Funds
Operational Programme Research,
Development and Education

MINISTRY OF EDUCATION,
YOUTH AND SPORTS

The project is co-financed by the European Union.

Contents

List of Figures

List of Tables

List of Listings

Introduction

With current growth of computer networks and distributed systems it is very challenging and time-consuming to monitor them. All these devices are flooding servers with their runtime information, reporting their current state 24/7. This runtime information is so-called syslogs. However, even though such information is very important, it is not efficient to use human power to monitor and analyze such large datasets. To ease the analysis of such logs, there are dozens of different kinds of software available on the market, either open source or paid.

However, software based on traditional programming might require many periodical adjustments in their codes to be able to analysis new pattern of system logs. In other words, traditional analyzing process of syslogs is complex, prone to human errors and time-consuming. With the advent of machine learning methods, they can be more robust and more convenient to deal with the aforementioned difficulties.

Machine learning is a subset of artificial intelligence that enables computers to learn without being explicitly programmed with predefined rules. It has borrowed concepts from different fields like Statistics, Probability Theory and Linear Algebra and applied to solve practical problems. It focuses on the development of computer programs that can learn from the historical data and formulate a solution which can be used to solve similar problems in the future [1].

One of the biggest features of machine learning algorithms is their ability to improve over time. Machine learning technology can improve their efficiency and accuracy due to the increasing amounts of data that are processed. This gives the algorithm more experience which results in making better decisions or predictions.

Anomaly detection is a technique used to identify or detect unusual patterns that differs majorly from the rest of the data. It is critical in detecting a rare data pattern or potential problem in the form of financial frauds, medical events, system logs and many others. However, anomaly detection can use the machine learning and deep learning algorithms to achieve its tasks [2].

In this thesis, anomaly detection algorithms are implemented on network packets and syslog messages, looking for anomalous behaviours that may indicate a security threat. Several machine learning methods were utilized, e.g. supervised and unsupervised. Moreover, deep learning algorithm was utilized as well. Furthermore, the thesis illustrates which group of algorithms are the most suitable and have the best performance for anomaly detection.

The thesis is organized as follows:

Chapter 1 presents theoretical and mathematical behaviour of machine learning techniques and the difference between them. Chapter 2 deals with the theory of preprocessing the dataset through Natural Language Processing. In chapter 3 a practical implementation of syslog generator is shown and explained. Chapter 4 presents the logical structures and implementation of the proposed tools of data preprocessing and the machine learning models.

In chapter 5 the functionality of the proposed tools is demonstrated. The comparison of the proposed machine learning models is illustrated in the chapter 6. In the last chapter, the conclusion and future work are summarized.

Part I

Theoretical Part

1 Machine Learning

Machine learning is a branch of Artificial Intelligence for data analysis, that gives computers ability to learn and improve from experience [3]. Machine learning algorithms are built on mathematical models, which are searching for patterns and relations in given data. These models are then capable of making decisions and predictions, even though they were not explicitly programmed to do so [4].

Machine learning is widely used across different fields - from weather forecasting, through finances, computer sciences, image and language processing to tumor detection in biology. Utilization options for machine learning are almost endless and it can be used in almost every aspect of our lives [5].

1.1 Machine Learning For Anomaly Detection

Anomalies are events or items that differ from standard behaviour. They are rare and do not fit into normal pattern [6]. Anomalies can indicate whether there is some kind of problem, for example finance fraud, medical complication or cyber attack. Anomaly detection is a task to identify such events [7].

Figure 1.1 a) illustrates example of point not fitting into cluster and figure 1.1 b) illustrates example of a spike in a time-series data.

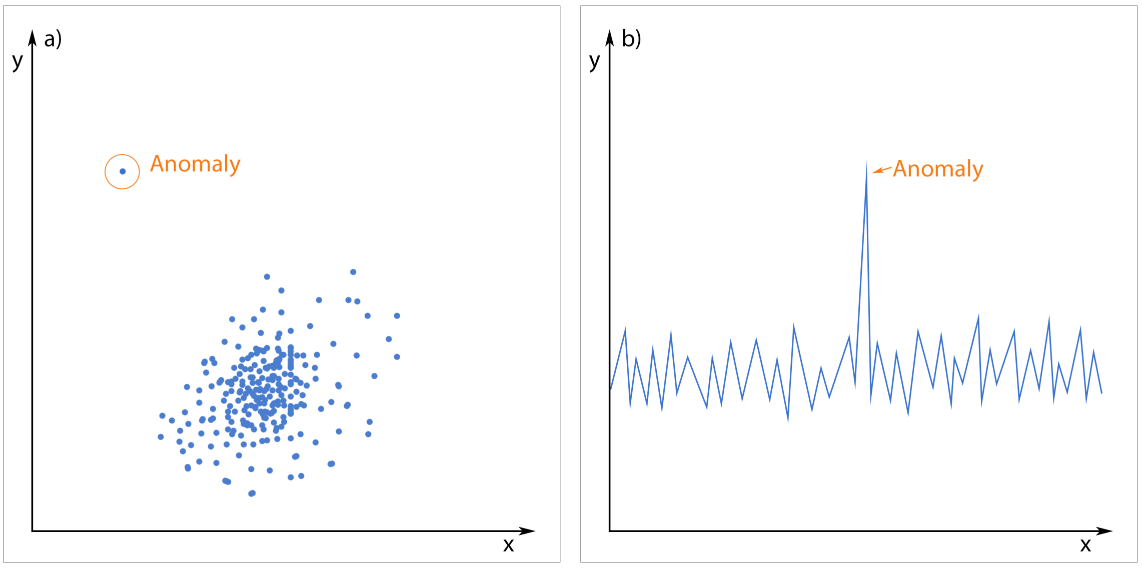

Fig. 1.1: Example of Anomaly

There are dozens of machine learning algorithms. Each algorithm has its own specifications, due to which some algorithms are better in solving certain problems than others and vice versa.

Some of the algorithms suitable for anomaly detection are Isolation Forests, One-class SVMs and Local Outlier Factor [6].

This thesis deals with implementation of these algorithms and compare them to others.

1.2 Supervised Machine Learning

The majority of existing machine learning algorithms are using supervised learning.

Supervised learning algorithms are developing machine learning model from labelled data. In other words, process of learning from labelled data can be viewed as a teacher supervising the learning process. Learning process is iterative - is running repeatedly until algorithm accomplishes sufficient level of performance [8].

Learning algorithm is mapping input variables (X) and an output variable (y) to find pattern. However, the goal is to create mapping function that can predict the output variable as precisely as possible.

$$y = f(X) \tag{1.1}$$

Based on problems algorithm is dealing with, supervised learning can be divided into two groups - Regression and Classification.

Regression

Regression is a supervised learning algorithm used to predict continuous responses [9]. For example, it can be used in finding relationship between price of a flat and it's location, number of bedrooms, etc. Regression is one of the earliest machine learning algorithms and is still widely used [10]. There are multiple types of regression methods, such as Simple Linear Regression (SLR), Multiple Linear Regression (MLR), Polynomial Regression (PR), Support Vector Regression (SVR), Decision Tree Regression (DTR), Random Forrest Regression (RFR), etc.

Due to character of the data that this thesis deals with, none of the regression methods are used.

Classification

Classification is a supervised learning algorithm, which is grouping given set of data points into classes. Sometimes classes are called also categories, labels or targets [11].

Basic example of classification is spam detection in emails. A classifier trains to understand how given dataset associate with labels spam and non-spam [12].

Spam detection falls under binary classification, as there are only two labels (classes). To deal with problems with multiple classes, such as whether an image is a dog, a cat or a mouse, there are Multi-class classifiers [13].

There are multiple types of classification methods, such as Logistic Regression (LR), K-Nearest Neighbors (K-NN), Support Vector Machine (SVM), Kernel SVM (kSVM), Naive Bayes NB), Decision Tree Classification (DTC), Random Forrest Classification (RFC), etc.

1.2.1 Logistic Regression

Logistic Regression (LR) is extending Linear Regression model to be used for classification problems [14].

Logistic Regression is not predicting exact number values (like 0 or 1), but it is using logistic sigmoid function to generate a probability value - for example value between 0 and 1 [15].

To compress output between 0 and 1, function of logistic regression is defined as following [14]:

$$logistic(\eta) = \frac{1}{1 + exp(-\eta)}. \tag{1.2}$$

Visual representation of Logistic function is shown in figure 1.2.

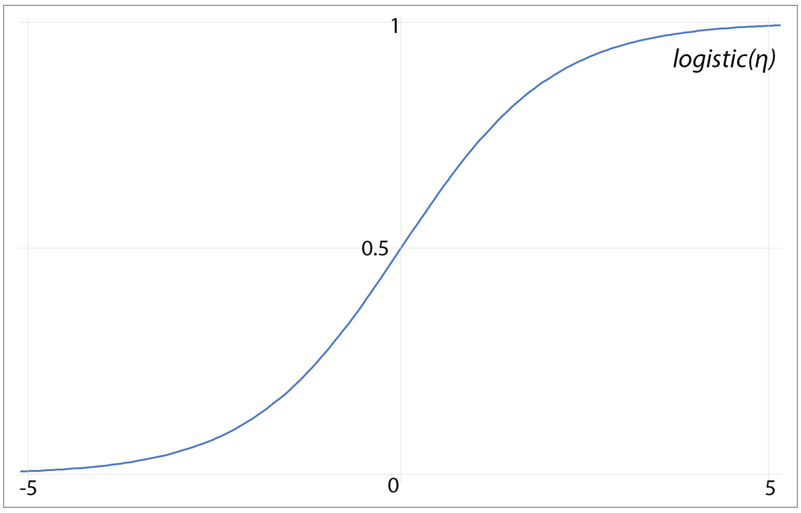

Fig. 1.2: Function of Logistic Regression

As mentioned above, logistic regression is extending linear regression. Linear regression model is used for regression problems and defined by following equation:

$$\hat{y}^{(i)} = \beta_0 + \beta_1 x_1^{(i)} + ... + \beta_p x_p^{(i)}. \tag{1.3}$$

As logistic regression is compressing outputs between values 0 and 1, function of logistic regression model is defined as:

$$P(\hat{y}^{(i)} = 1) = \frac{1}{1 + exp(-(\beta_0 + \beta_1 x_1^{(i)} + ... + \beta_p x_p^{(i)}))}. \quad (1.4)$$

Probability P will be a value between 0 and 1. As the output of the model needs to be either 0 or 1, threshold have to be set up. Usually, threshold is set up to be in the middle of values as following:

$$p \geq 0.5, class = 1 p < 0.5, class = 0. \quad (1.5)$$

In case threshold is set up as above and prediction model returns value 0.67, it will be classified as 1. If returned value is for example 0.48, it will be classified as 0 [15].

Implementation of Logistic Regression model in Python is described in the Section 4.3.1.

1.2.2 K-Nearest Neighbors

K-Nearest Neighbors (KNN) is simple, yet powerful classification algorithm. However, it can be used for regression problems as well [16]. The KNN algorithm is based on assumption that similar data points lay close to each other [17].

Decision to which class should a node be assigned to is based on majority of votes of its neighbors [18]. Number of neighbors depends on chosen value of parameter K. Nearest neighbors are then calculated by one of the following distance functions:

Euclidean function:

$$D(x, y) = \sqrt{\sum_{i=1}^{k} (x_i - y_i)^2}. \quad (1.6)$$

Manhattan function:

$$D(x, y) = \sum_{i=1}^{k} |x_i - y_i|. \quad (1.7)$$

Minkowski function:

$$D(x, y) = (\sum_{i=1}^{k} (|x_i - y_i|)^q)^{1/q}. \quad (1.8)$$

However, these functions are accurate only for continuous variables. For classification problems is more suitable to use Hamming distance function:

$$D_H = \sum_{i=1}^{k} |x_i - y_i|. \quad (1.9)$$

$$x = y \Rightarrow D = 0. \tag{1.10}$$

$$x \neq y \Rightarrow D = 1. \tag{1.11}$$

There is no specified method to choose the best value of K, but the following recommendations should be taken into consideration [19]:

- Large value of K means more precise model.
- Choose an odd value of K for a problem with two classes - if $K = 2$ for such a problem, there might be a tie of what should be a class of the node.
- Multiple of number of classes should not be chosen.

$K = 1$ is a special case of KNN, when node is assigned to class of its closest neighbor [20].

Implementation of K- Nearest Neighbors model in Python is described in the Section 4.3.2.

1.2.3 Support Vector Machine

Another supervised learning algorithm that can be used for both classification or regression problems is Support Vector Machine (SVM). However, when it is used to solve regression problems, it is mainly referred as Support Vectore Regression [21]. Same as KNN, SVM is mostly used for classification problems [22]. SVM's goal is to find and "draw" the best hyperplane that can divide n-dimensional dataset into classes [23]. Hyperplanes are basically boundaries separating different classes. They can be multidimensional and it is all depending on amount of input variables. If there are only two input variables (for example x and y), hyperplane is going to be just a line. If there are three input variables, hyperplane is going to be a 2D plane, etc.

- Support vectors "support" the hyperplane. They are the closest data point to the hyperplane and are affecting position of it.
- Positive and negative hyperplanes are "touching" border points of each classes as shown in figure 1.3 c).
- Margin is a distance between positive and negative hyperplanes [24].

Figure 1.3 a) ilustrates two classes. Figure 1.3 b) ilustrates same two classes separated by hyperplanes with small margin, which is not ideal, and with large margin. Ideal hyperplane with maximum margin (large margin multiplied by 2) is shown in figure 1.3 c).

Figure 1.4 a) shows one class "besieged" by the other class. When the visualisation is taken into third dimension as shown in figure 1.4 b), classes are now separable

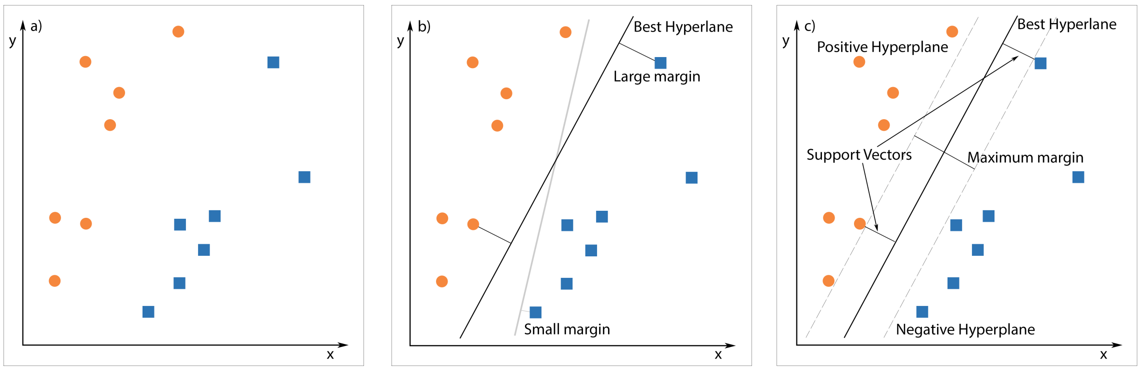

Fig. 1.3: Example of Support Vector Machine in 2D [25]

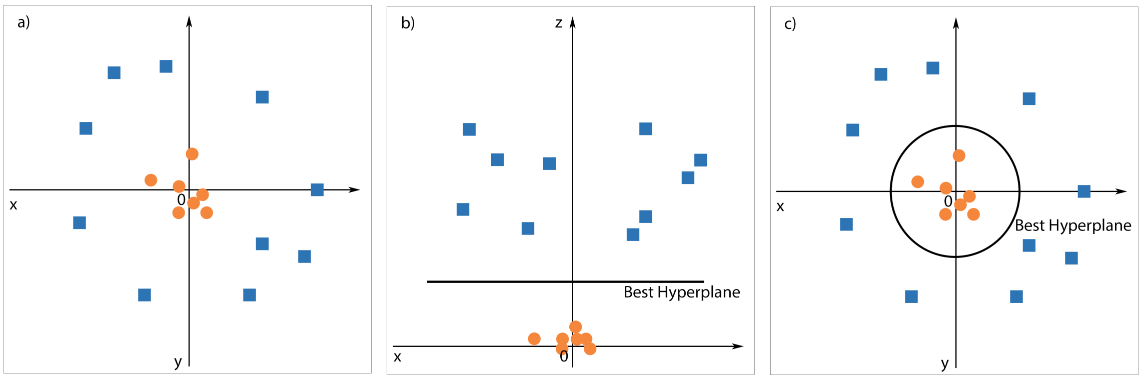

Fig. 1.4: Example of Support Vector Machine in 3D [25]

more easily. How hyperplane from figure 1.4 b) will look in initial view is shown in figure 1.4 c).

Kernel SVM

Calculating nonlinear data into multi-dimensional spaces can be pretty demanding on computer performance. Fortunately, there is a smart solution - kernel trick, sometimes referred to as a Kernel SVM. Instead of working with actual vectors, it can work with only dot products[1] in between them [25]. Due to this advantage, this thesis implements Kernel SVM instead of classic SVM.

Implementation of Kernel SVM model in Python is described in the Section 4.3.3.

[1]Dot Product of two vectors is equal to the cosine of the angle between them, multiplied by the lengths of each of the vectors [26].

One-class SVM

Even though One-class SVM (OC-SVM) is unsupervised machine learning algorithm, it is extension of SVM, hence it is placed in this section. Description of unsupervised machine learning algorithms can be found in the Section 1.3.

As it is clear from it's name, One-class SVM is a classification algorithm used on datasets that should contain only one class - normal data. Such algorithm can be used to find anomalies. It is designed to find patterns and relations in one class and based on this knowledge, it marks all data points that do not fit [27].

One-class SVM algorithm creates function which returns *+1* for data points that resides in area of most of the data points and *-1* for data points from elsewhere [28].

Implementation of One-class SVM model in Python is described in the Section 4.3.4.

1.2.4 Naive Bayes

Naive Bayes (NB) is an algorithm calculating probability of class for given data based on learned knowledge [29].

Bayes' Theorem

Bayes' Theorem is a mathematical formula describing conditional probability - probability of an event based on knowledge of other related conditions [30]. For example, an internet search for "movie with a yellow car" brings up "Transformers". How the search engine knows this? Has it watched the movie? No it hasn't, but it learned it from similar searches of other people who probably were looking for it. Search engines calculated this probability using Bayes' Theorem.

Formula of Bayes' Theorem is defined as following [31]:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}, \tag{1.12}$$

where A represents hypothesis and B represents observed evidence. $P(A|B)$ is called Posterior Probability, $P(B)$ is called Prior Probability and $\frac{P(B|A)}{P(B)}$ is called Likelihood Ratio [32].

Naive Assumption

Bayes' Theorem applied to dataset will be defined as following:

$$P(y|X) = \frac{P(X|y)P(y)}{P(X)}. \tag{1.13}$$

A is replaced by class variable y and B is replaced by n-sized vector of features X, defined as [33]:

$$X = (x_1, x_2, ...x_n). \tag{1.14}$$

Following on this, formula will be defined as:

$$P(y|x_1, ...x_n) = \frac{P(x_1|y)...P(x_n|y)P(y)}{P(x_1)...P(x_n)}. \tag{1.15}$$

After few adjustments, equation used by classifier will look like this:

$$\hat{y} = argmax_y P(y) \prod_{i=1}^{n} P(x_i|y). \tag{1.16}$$

Implementation of Naive Bayes model in Python is described in the Section 4.3.5.

1.2.5 Decision Tree

Decision Tree (DT) is a simple algorithm with tree-like structure, where attributes of dataset are represented by internal nodes of a tree, decision rules are represented by branched and outcome values are represented by leafs [34]. Internal nodes are called Decision Nodes and are used to make decision. They can have multiple branched going out of them. Leafs are called Leaf Nodes and are representing final output values. They do not have any brunches but the one incoming towards them from associated decision node [35]. DT algorithm can be used for either regression and classification problems. They can be called Decision Tree Regression (DTR) and Decision Tree Classification (DTC) algorithms respectively [34].

As it can be seen in figure 1.5, decision tree starts with the root node, expands into further branches and forms tree-like structure. Each branch represents decision rule which can have two or more outcomes, for examples Yes/No, Red/Green/Blue or numeric data as well.

Implementation of Decision Tree Classification model in Python is described in the Section 4.3.6.

1.2.6 Random Forest

Random Forest (RF) are multiple DT algorithms running at once - it is a "forest" of a "decision trees". As well as DT, RF is supervised learning algorithm which is being used for both, regression and classification problems [36]. When it is used to solve the regression problems, it is referred to as Random Forest Regression (RFR) and as Random Forest Classifier (RFC) when used to solve classification problems. Figure 1.6 illustrates main concept of a RF algorithm. Training dataset is randomly split

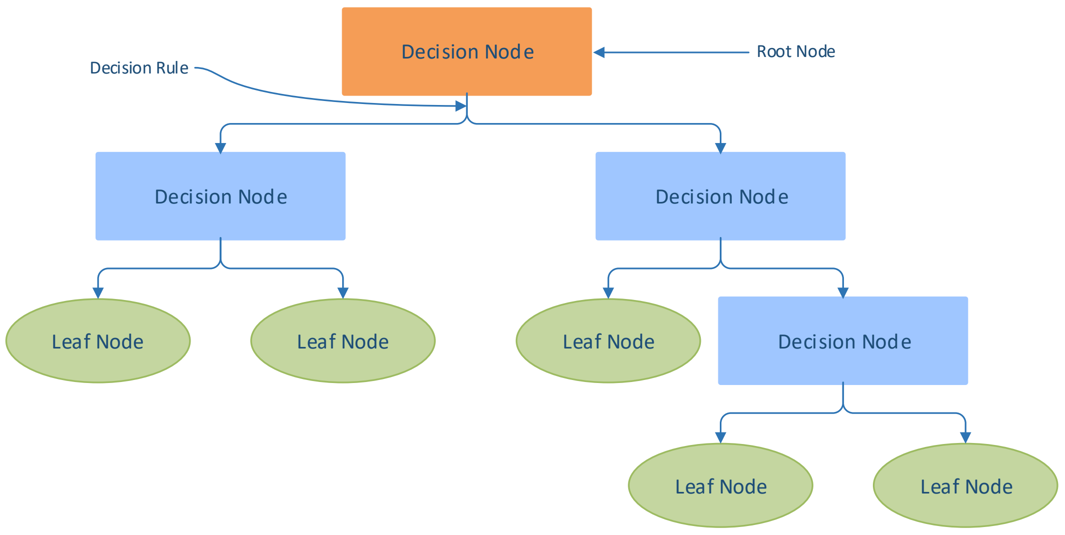

Fig. 1.5: Example of Decision Tree [35]

into multiple training samples, one for each tree. RFC is then collecting predictions from each decision tree and is selecting the best solution. Selection of the best solution is done by "voting". Prediction, on which most of trees have agreed on, is selected [37]. The more trees in forest, the more accurate the result will be [38].

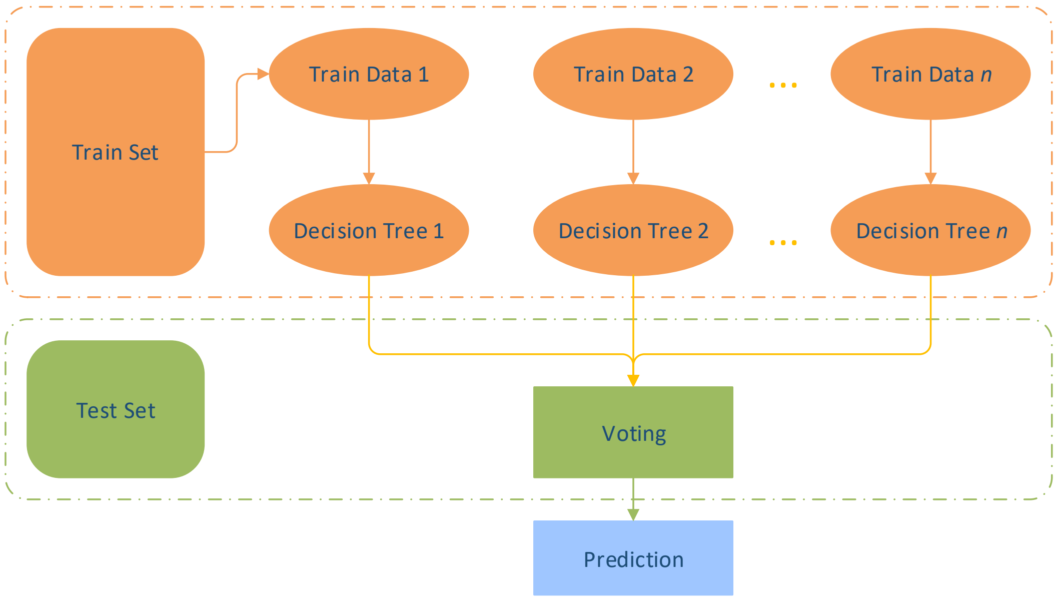

Fig. 1.6: Example of Random Forest [39]

Random Forest algorithm has two stages. In the first stage, algorithm creates random forest of n decision trees. In the second stage, algorithm makes predictions utilizing random forest created in previous stage. At the end of the second stage it is "voting" of each decision tree.

Process of the first stage is described as followin [36]:

1. Random number of features k from training dataset is selected.
2. Decision tree for selected features is created.
3. Steps 1 and 2 are repeated n times to create n number of trees.

Process of the second stage can be described as following:

1. Test dataset is taken and the features are inserted into each decision tree created in the first stage.
2. Predictions of each decision tree are taken and votes for each predicted target is calculated.
3. Target with highest votes (predictions) is selected as final prediction.

Implementation of Random Forest Classification model in Python is described in the Section 4.3.7.

1.3 Unsupervised Machine Learning

In some cases, the training dataset is containing only input variables (X) and no output variables (y). In such cases, the goal might be to find certain patterns [40]. To do so, unsupervised machine learning algorithms can be used. These algorithms can be used to find groups (clusters) based on similarities. Unsupervised machine learning algorithms are also often used for anomaly detection purposes. They can "group" normal behaviour data into one group and mark all data not fitting into this group as anomalies.

Unsupervised learning problems can be grouped into following groups [8]:

- Association mining is identifying associations in data points, for example people that buy A might buy also B.
- Clustering is grouping data points based on similarities.

Association Mining

Association mining is algorithm that is searching and uncovering relations between input variables [41]. These relations are formed into so-called association rules, which are then used to make predictions by calculating probabilities [42]. Example of association mining is Apriori algorithm [43].

Clustering

Clustering is the most common unsupervised learning method [44]. It is searching for similarities in uncategorized dataset and grouping (clustering) them based on found patterns. Clustering algorithms are simple and quite effective. Number of clusters can be modified. It can help to improve granularity of created clusters. Figure 1.7 illustrates simple example how certain data point can be formed into clusters.

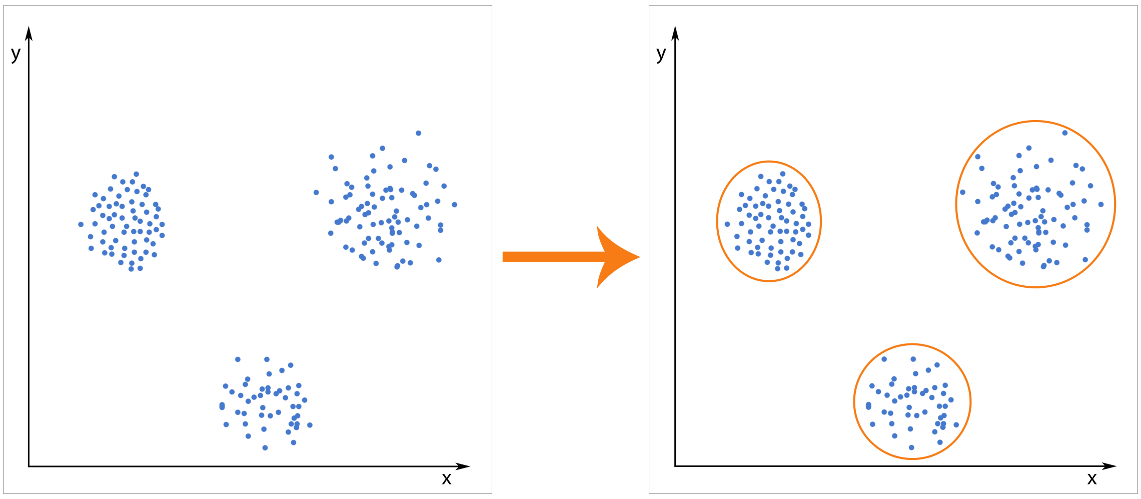

Fig. 1.7: Example of formation of clusters [43]

Most common clustering algorithms are K-Means clustering, Hierarchical clustering, Gaussian mixture models, Self-organizing maps and Hidden Markov models.

1.3.1 Isolation Forest

Isolation Forest is a machine learning algorithm used exclusively for anomaly detection. Instead of searching for pattern of normal data points as most of such algorithms do, Isolation Forest algorithm explicitly isolates anomalies [45]. Process of searching for anomalies is based on effectively constructed tree structure. In this tree, anomalies are closer to the root of the tree and normal data point lays deeper in the structure. Basically, it is harder to isolate normal data point than anomaly - it takes more time to find its exact location.

Set of these trees is called Isolation Forest. The only two parameters of this algorithm are number of trees and sub-sampling size.

Process of separating each point is shown in figure 1.8. Figure 1.8 a) illustrates process of separating of a not anomalous data point and figure 1.8 b) illustrates process of separating of an anomalous data point. It is iterative and pretty straightforward:

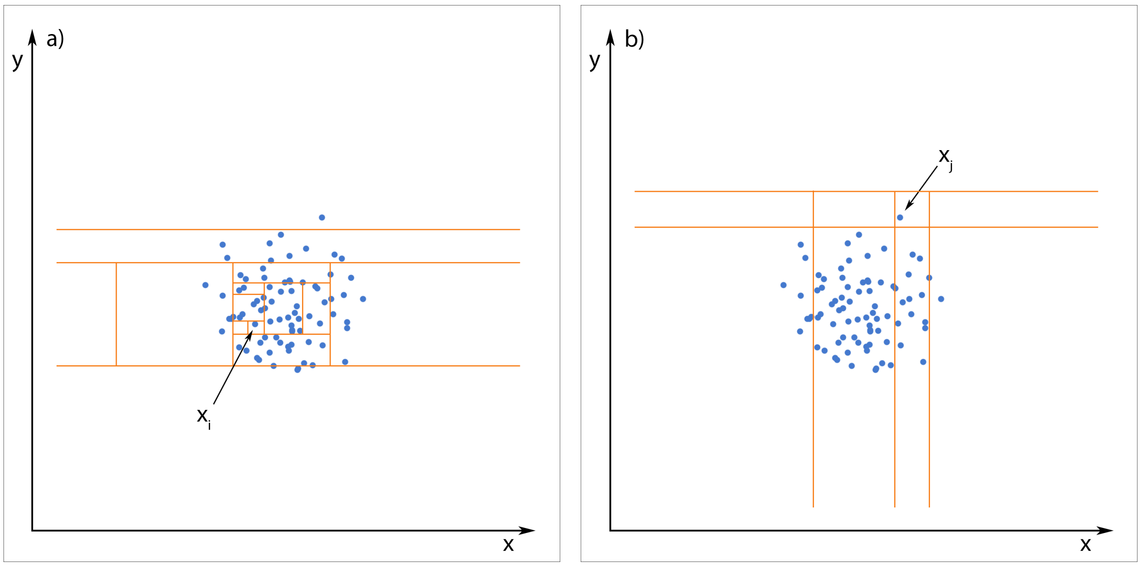

Fig. 1.8: Isolation Forest algorithm

1. Selected data point to be isolated.
2. Minimum and maximum range to isolate for each feature needs to be set.
3. Random feature needs to be chosen.
4. Chosen random value to be from the range:
   - If the point is above, replace minimum of the range with this value.
   - If the point is below, replace maximum of the range with this value.
5. Steps 3 and 4 need to be repeated until the point is isolated.
6. Amount of iterations of steps 3 and 4 return Isolation number.

Algorithm selects data point as anomaly if isolation number is low, meaning it is close to the root of the tree.

Implementation of Isolation Forest model in Python is described in the Section 4.3.8.

1.3.2 Local Outlier Factor

Local Outlier Factor (LOF) is a method for calculating neighbors density of certain data point and it is comparing it to neighbors density of other points. If the density of a point is lower than density of other points, it is marked as outlier (anomaly) [46].

Function of estimated density is defined as [47]:

$$\hat{f}(p) = \frac{k}{\sum_{x \in N(p)} d(p, x)}, \tag{1.17}$$

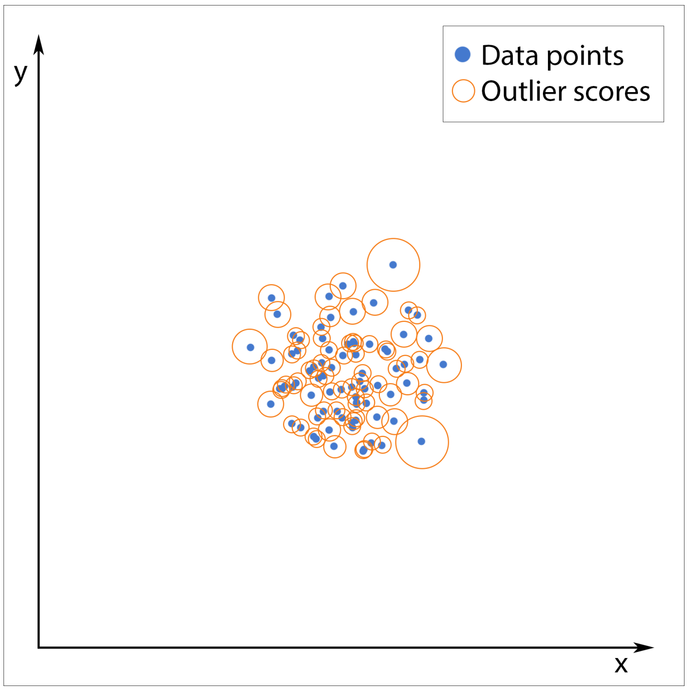

Fig. 1.9: Local Outlier Factor

where $N(p)$ are neighbors of data point p, k represents number of points in dataset and $d(p,x)$ represents distance between points p and x. Local Outlier Factor of data point p is then defined as:

$$LOF(p) = \frac{\frac{1}{k}\sum_{x\in N(p)} \hat{f}(x)}{\hat{f}(p)}. \tag{1.18}$$

Figure 1.9 illustrates example of how the LOF demonstrates the data points with corresponding outlier score. Implementation of Local Outlier Factor model in Python is described in the Section 4.3.9.

1.3.3 K-Means

K-Means is a simple and one of the most popular unsupervised learning algorithms [48]. It is creating groups (clusters) around centroids, which are basically defining the cluster. Centroids are also called means as they basically hold mean values of data points inside the cluster [49]. Data points are assigned to cluster based on distance between the data point and centroid. It is assigned to cluster of the closest centroid. Each data point can be part of only one cluster.

Process of how K-Means algorithm works is pretty straightforward [50].

1. Number of clusters K is specified manually.
2. K data points are randomly selected as centroids.

3. All data points are assigned to the closest centroid.
4. Centroids of newly created clusters are recalculated (centroids now might be "virtual" - not actual data points but real center of cluster).
5. Steps 3 and 4 are repeated until one of following criteria are matched:
    - Centroids are not changed.
    - Data points remain in same clusters.
    - Algorithm reaches maximum number of iterations.

Implementation of K-Means model in Python is described in the Section 4.3.10.

1.3.4 Hierarchical Clustering

Hierarchical Clustering (HC) is an unsupervised machine learning algorithm that creates clusters with predetermined order from top to bottom [51]. Main principle of HC algorithm is shown in figure 1.10.

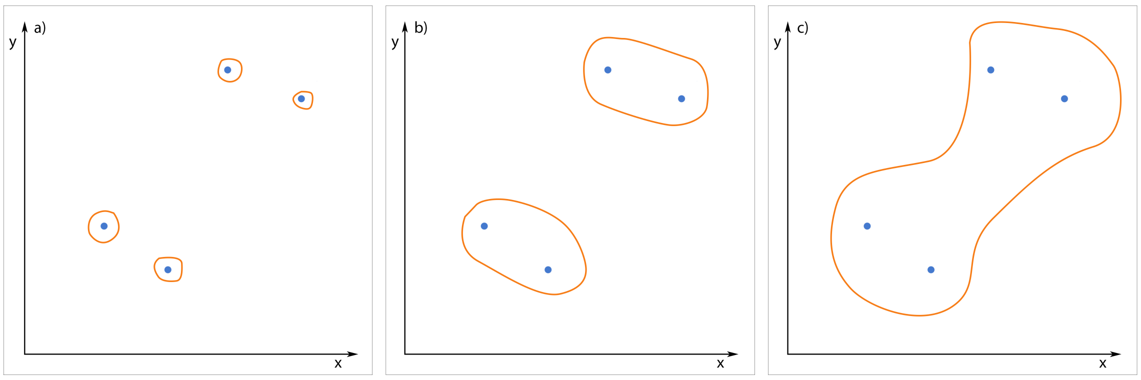

Fig. 1.10: Agglomerative Hierarchical Clustering

There are two types of hierarchical clustering approaches [52].

- Agglomerative hierarchical clustering.
- Divisive hierarchical clustering.

Agglomerative hierarchical clustering

Agglomerative method is also called bottom-up method and process of creating clusters is following:

1. Each data point is assigned into its own cluster.
2. Similarities between each cluster are calculated.
3. Two most similar clusters are merged together.
4. Steps 2 and 3 are repeated until n number of clusters is created or until only a single cluster is left.

Divisive Hierarchical clustering

Divisive method is also called top-down method and process of creating clusters is exactly opposite to Agglomerative method:

1. All data points are assigned into one cluster.
2. The cluster is divided into two least similar clusters.
3. Step 2 is repeated until n number of clusters is created or until each data point has its own cluster.

There are following methods to calculate distance between clusters [51]:

- Single Linkage: Distance between clusters is defined as the shortest distance between two points in each cluster.
- Complete Linkage: Distance between clusters is defined as the greatest distance between two points in each cluster.
- Average Linkage: Distance between clusters is defined as the average distance between all points from one cluster and all points from the second cluster.
- Centroid Linkage: Distance between clusters is defined as a distance between centroids of each cluster.

Implementation of Hierarchical Clustering model in Python is described in the Section 4.3.11.

1.4 Neural Networks and Deep Learning

Artificial Neural Network is a branch of machine learning modeled to mimic the network of neurons in a brain. On other hand, deep learning is an advanced subfield of machine learning that uses algorithms inspired by the structure and function of the Artificial Neural Networks. Deep learning is trained by using large sets of labelled data that learn features directly from the data without the need for manual feature extraction [53].

Common deep learning algorithms include convolutional neural networks and recurrent neural networks. Deep learning models are often referred to as deep neural networks. The term “deep” usually refers to the number of hidden layers in the neural network. Traditional neural networks only contain two to three hidden layers, while deep networks can have much more [54].

Neural Network algorithms are constructed with following connected layers:

- Input Layer: this layer accepts all provided inputs.
- Hidden Layer: this layer is between the input and the output layers where computations are performed. In case of deep learning then it means the network joins neurons in more than two to three layers.

- Output Layer: output is delivered via this layer.

The most basic unit of a Neural Network is a Perceptron. A Perceptron is a single layer Neural Network that is used to classify linear data. It has 4 important components:

1. Inputs.
2. Weights and Bias.
3. Summation Function.
4. Activation or transformation Function.

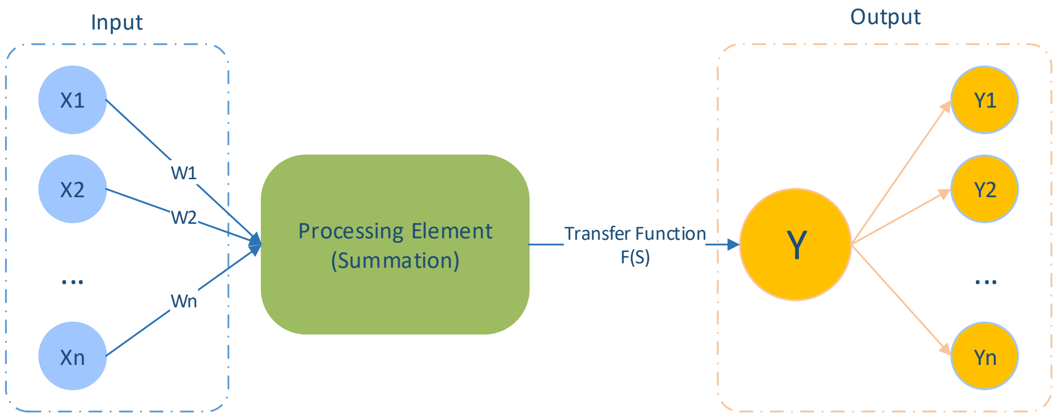

Fig. 1.11: Artificial Neural Network [56]

The inputs X received from the input layer are multiplied with their randomly chosen assigned weights w. While the weights determine the slope of the classifier line, bias allows to shift the line towards left or right. Normally, bias is treated as another weighted input with the input value x_0. However, the multiplied values are then added to form the *Weighted Sum*. The weighted sum of the inputs and their respective weights are then applied to a relevant Activation Function. The activation function maps the input to the respective output. This process is illustrated in figure 1.11.

Output of neuron can be expressed mathematically as [55]:

$$y = Activation(SUM(input * weight) + bias). \tag{1.19}$$

Activation function does a nonlinear transformation of the input data and thus enables the neurons to learn better. Some examples of activation functions are Sigmoid, ReLU and Softmax [56].

Sigmoid Function is defined as:

$$\frac{1}{1+e^{-x}}. \tag{1.20}$$

ReLU (Rectified Linear Unit) function is defined as:

$$f(x) = max(0, x). \tag{1.21}$$

Keras

Keras is the most used deep learning framework, it is written in Python and supports multiple back-end Neural Network computation engines.

The Model is the core Keras data structure. There are two main types of models available in Keras, the Sequential model, and the Functional one. The Sequential model is a linear stack of layers, and the layers can be described very simply. On other hand, The Keras Sequential model is simple but limited in model topology. The Keras Functional API is useful for creating complex models, such as multi-input/multi-output models, directed acyclic graphs (DAGs), and models with shared layers [57].

Implementation of Artificial Neural Network model in Python using Keras library is described in the Section 4.3.12.

2 Natural Language Processing

Syslog messages are sort of human language. However, human language is complex set of huge variety of words which are used to express huge amounts of information. Different combination of words, different shapes of chosen words and even different topics can add or change meaning of expressed information. Given the nature of computers, it is challenging to interpret such information correctly. However, to override these obstacles the Natural Language Processing (NLP) is used. NLP is a branch of artificial intelligence that helps to interpret human language to computers. The objective of NLP is to read, decode and understand human language [58].

There are several NLP techniques [59]:

- Named Entity Recognition.
- Tokenization.
- Stemming and Lemmatization.
- Word Embeddings.
- Natural Language Generation.
- Sentiment Analysis.
- Sentence Segmentation.

Named Entity Recognition (NER) is popular NLP, which is identifying names, nouns, etc. in given phrase or paragraph. It is widely used for example for news categorization, in search engines or to help categorize feedbacks provided by customers' reviews.

Tokenization technique, as it is obvious from its name, is splitting the text into tokens. Tokens can be understood as words, sentences, characters, numbers, etc.

Stemming is a process reducing words into their root shape. It basically removes the suffixes, for example, it adjusts word "denied" to "den".

Lemmatization is a process that returns the base form of the word. For example, it adjusts word "denied" to "deny".

Word Embeddings is the collective name for a set of language modeling and feature learning techniques in NLP, where words or phrases are mapped to vectors of real numbers. Such techniques are Bag of Words (BoW) and Term Frequency-Inverse Document Frequency (TF-IDF). However, BoW is the most used one. It creates "a bag" of all occurrences of each word, without caring about the position of the word in the text. Each word is basically represented by its own variable and the value of the variable means number of occurrences of the word.

To demonstrate this technique, the following two sentences are considered: *"Deny IP due to Land Attack."* and *"Deny protocol connection spoof."*. At first, it tokenizes the sentences and creates list of all words: *"Deny", "IP", "due", "to", "Land", "Attack", "protocol", "connection"* and *"spoof"*. Then words are mapped to vectors of real numbers. First sentence is transformed into vector *[1, 1, 1, 1, 1, 1, 0, 0, 0]* and second sentence is transformed into vector *[1, 0, 0, 0, 0, 0, 1, 1, 1].*

This technique is evaluated in this thesis and its implementation is described in section 4.2.1.

Natural Language Generation (NLG), as it is clear form the name, it is a technique to convert structured data into desired language. It is also called *data storytelling*, because it can help to understand patterns or insights in structured data. It can compose structured data into understandable reports, for example financial reports from stock market.

Sentiment Analysis is the most common preprocessing technique analysing "the emotion" of written text. Mostly it is analyzing subjective texts, such as reviews, whether they are positive, negative or neutral. That helps companies to better understand reputation of their products and services and understand customer experience.

The process of sentiment analysis is pretty straightforward: at first it breaks the text down into tokens (words, phrases, sentences, etc.), identifies token(s) carrying sentiment and assign an sentiment score to it from within the range of *-1* to *1* [60].

Sentence Segmentation is dividing text into sentences or phrases. It is also called *Sentence boundary detection* as it identifies sentence boundaries between words. Sentence Segmentation is one of the basic NLP techniques as it is quite easy to understand and use.

There are several libraries that support NLP such as Natural Language Toolkit (NLTK) and Spacy [61].

Part II

Practical Part

3 Implementation of Syslog Generator

Since machine learning algorithms need data to learn from, then syslog data needs to be available in our case. However, finding a suitable and labelled dataset of syslog messages was a big challenge. There are basically no such datasets available to download and use in this thesis.

Labelled dataset of syslogs from network devices of any vendor was crucial for this thesis.

Due all of this, a script generating desired dataset was developed for this thesis and can be found also on Github[1]. This script is generating syslog messages of Cisco ASA devices. Message templates are randomly chosen from official Cisco webpage [62]. Chosen messages are of two types - anomalous and informational. As anomalous messages are messages marked by vendor as a cause of an attack. However, informational messages are various messages that notify about common state update etc.

It is worth to mention here, that if we take the anomalous and informational messages as per vendor recommendation, then we might end in incorrect prediction for some type of messages. Reason of that, the message templates provided by vendor don't take into consideration the words in the message itself to express about anomalous or informational messages. For example syslog message *%ASA-6-778007: VXLAN: Packet from ifc-name:IP-address/port to IP-address/port was discarded due to invalid NVE peer.* is considered as "informational" by the vendor, while in reality it will be most probably considered as anomalous by the ML model. Due to that, machine learning models were trained on messages that contain *discarded* to be as anomalous message.

3.1 Structure of Syslog Generator

Structure of Syslog Generator is shown on figure 3.1. The tool first verifies how many syslog messages were requested to generate. The it checks whether the messages should be labelled or not. Each branch then checks whether the messages should be seen or unseen and based on it, it picks the messages from either templates of seen or unseen messages.

More detailed description of each used method can be found below.

[1]Github: https://github.com/miroslav-siklosi/Syslog-Generator

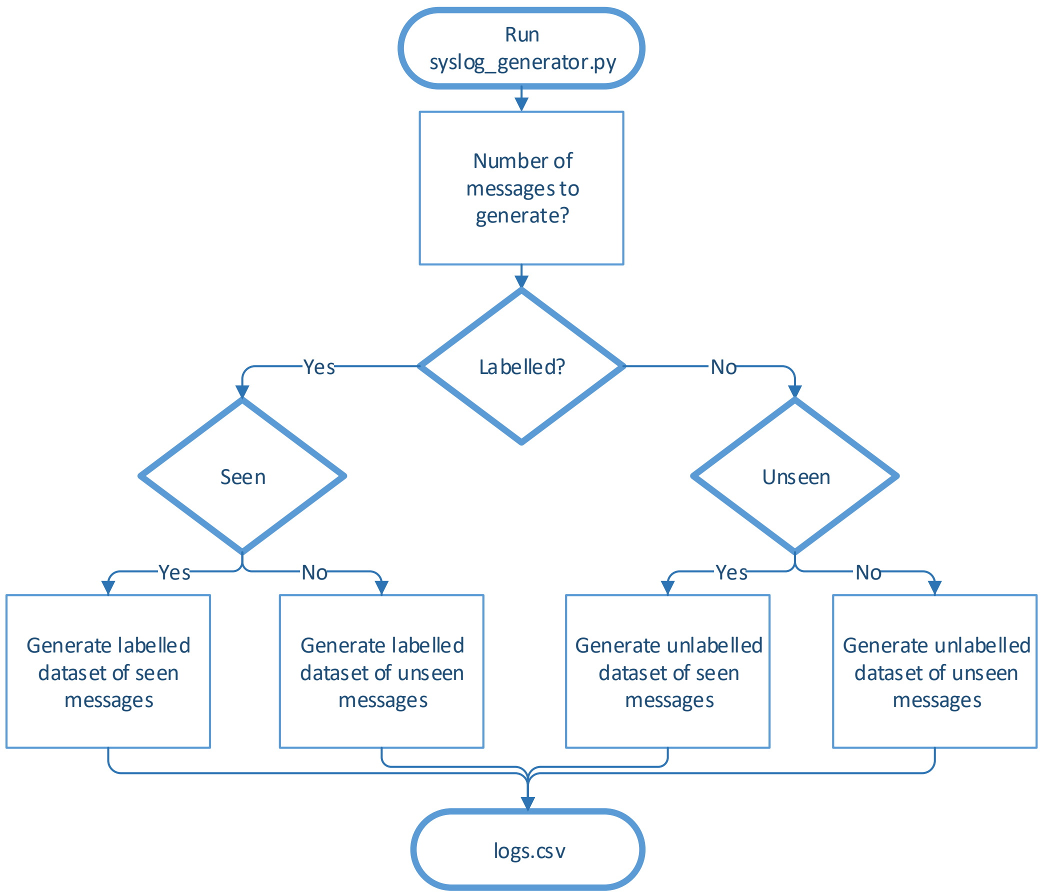
Run syslog_generator.py
Number of messages to generate?
Labelled?
Yes
No
Seen
Unseen
Yes
No
Yes
No
Generate labelled dataset of seen messages
Generate labelled dataset of unseen messages
Generate unlabelled dataset of seen messages
Generate unlabelled dataset of unseen messages
logs.csv

Fig. 3.1: Flowchart of Syslog Generator

Message Templates

At first, a class for templates needs to be created. There are four arrays containing 70 templates of anomalous and 86 templates of informational messages. Example are shown on listing 3.1.

First array, *anomalous_messages*, contains 50 "seen" anomalous messages. These are messages on which machine learning models will be trained. Second array, *anomalous_unseen*, contains 20 "unseen" messages - such messages will be used to test how well machine learning models are trained and how good are they adapting to new, unseen variables. Last two arrays, *informational_messages* and *informational_unseen*, are containing 60 and 26 messages respectively. Idea of seen and unseen messages is the same as with anomalous messages.

```python
class logTemplates:
    anomalous_messages = [
        "%ASA-2-106017: Deny IP due to Land Attack from {source_address} to
        ↪  {dest_address}.",
        ...
        ]
    anomalous_unseen = [
        "%ASA-1-106022: Deny protocol connection spoof from {source_address} to
        ↪  {dest_address} on interface {interface_name}.",
        ...
        ]
    informational_messages = [
        "%ASA-1-101001: (Primary) Failover cable OK.",
        ...
        ]
    informational_unseen = [
        "%ASA-1-101002: (Primary) Bad failover cable.",
        ...
        ]
```

Listing 3.1: *Message Templates*

Filling in Templates

There are brackets inside of the templates, which contain arguments such are *source_address*, *dest_address*, *interface_name*, etc. As expected, they are used to fit respective values. To generate as production-like messages as possible, these values are generated randomly, but with certain restrictions - private IP addresses are generated from respective IP pools etc.

First, there are defined so-called *generators*. They are used to match certain expressions inside the brackets. To find brackets inside the template, there is simple *for* loop to parse it, as shown on listing 3.2.

Values inside these brackets are generated by method specific for each generator. Most of the methods are using Python package *Faker*, which is generating random fake data [63].

For example, to fill in IP addresses into brackets containing generators such as *source_address* and *dest_address*, there is simple method using *Faker*, as shown on listing 3.3. Since there are templates that have to contain specific types of addresses, namely private and public, there are specific methods for such generators.

Faker is also used to generate random and fake URL links and MAC Addresses, using same logic as in previous methods.

```python
def fill_message(message):
    generators = {"source_address": generate_ip_address,
                  "dest_address": generate_ip_address,
                  "interface_name": generate_interface_name,
                  "source_port": generate_port,
                  "dest_port": generate_port,
                  "local_address": generate_local_address,
                  "remote_address": generate_remote_address,
                  "user": generate_user,
                  "url": generate_url,
                  "mac_address": generate_mac_address,
                  "number": generate_number,
                  "service": generate_service}
    parts = message.split("{")
    result = ""
    for part in parts:
        if "}" in part:
            value_type, rest = part.split("}")
            value = generators[value_type.lower()]()
            result += str(value) + rest
        else:
            result += part
    return result
```

Listing 3.2: *Method to fill in a message*

```python
def generate_ip_address():
    ip_addr = Faker().ipv4()
    return ip_addr
def generate_local_address():
    return Faker().ipv4(private=True)
def generate_remote_address():
    return Faker().ipv4(private=False)
```

Listing 3.3: *Generate random IP address*

Message Generating

Messages are generated based on requirements. Method to generate them is pretty straight forward and is shown on listing 3.4.

At first, date and time are generated, then a template is chosen from the needed array, and at the end, the device name is generated. These components compose the syslog message.

Method *pick_message* (shown on listing 3.5) is generating "seen" messages from

```python
def generate_log(add_label, gen_seen):
    now = datetime.datetime.now().strftime("%b %d %Y %H:%M:%S")
    if add_label:
        if gen_seen: # add_label == True and gen_seen == True
            message, is_anomaly = pick_message()
        else: # add_label == True and gen_seen == False
            message, is_anomaly = pick_unseen_message()
        filled_message = fill_message(message)
        filled_message = f"{filled_message}\t{int(is_anomaly)}"
    else: # add_label == False
        if gen_seen: # add_label == False and gen_seen == True
            message, is_anomaly = pick_message()
        else: # add_label == False and gen_seen == False
            message, is_anomaly = pick_unseen_message()
        filled_message = fill_message(message)
    device = f"FW{str(random.randint(0, 25)).zfill(2)}" # generate device name
    log = f"{now} {device} : {filled_message}" # compose syslog message
    return log
```

Listing 3.4: *Generate a log message*

both seen anomalous and informational templates. To ensure more informational messages than anomalous are in place, there is at first randomly generated number from 1 to 20. If the number equals 1, it will pick anomalous template, otherwise it will pick informational template. This gives us potential ration of generated messages to 1:20 for anomalous and informational messages respectively, meaning only 5% of messages should be anomalous.

Logical values *True* and *False* in here are used later on to label generated message if it is anomalous (True) or informational (False).

```python
def pick_message():
    x = random.randint(1, 20)
    if x == 1:
        return random.choice(logTemplates.anomalous_messages), True
    else:
        return random.choice(logTemplates.informational_messages), False
```

Listing 3.5: *Method to pick a seen message*

Method *pick_unseen_message*, as it can be clear from its name, is used to pick unseen templates. These are picked randomly from unseen templates. The rest of the code has same purpose as in method *pick_message*. This method is shown on lisitng 3.6.

```python
def pick_unseen_message():
    x = random.randint(1, 20)
    if x == 1:
        return random.choice(logTemplates.anomalous_unseen), True
    else:
        return random.choice(logTemplates.informational_unseen), False
```

Listing 3.6: *Method to pick an unseen message*

At last, whole log file is generated by method *generate_logs_file*, shown on listing 3.7.

```python
def generate_logs_file(log_count, add_label, gen_seen, filename):
    with open(filename, "w") as logs_file:
        for i in range(log_count):
            log = f"{generate_log(add_label, gen_seen)}\n"
            logs_file.writelines((log))
```

Listing 3.7: *Method generating log file*

Argument Parser

To obtain the desired functionalities of the tool such as changing amount of generated logs, efficient Argument Parser was implemented as shown on listing 3.8 [64].

```python
parser = argparse.ArgumentParser(prog="syslog_generator.py")
parser.add_argument("--number", dest="number", type=int, required=False,
↪   default=100)
parser.add_argument("--labelled", dest="labelled", choices=["yes", "no"],
↪   required=False, default="yes")
parser.add_argument("--seen", dest="seen", choices=["yes", "no"],
↪   required=False, default="yes")
```

Listing 3.8: *Argument parser of Syslog Generator*

There are three arguments. First one is *number* and it serves to adjust number of syslog messages to be generated. However, default number is set to *100*. Second argument is *labelled* and it serves to select whether generated dataset should be labelled or not. However, default value is set to *yes*. Last argument is *seen*. It serves to select whether generated messages should be chosen from templates of *seen* messages or from both templates, *seen* and *unseen*. However, default option is set to *yes*.

All three arguments are then put into method shown on listing 3.9 to generate logs file.

```
generate_logs_file(int(args.number), args.labelled == "yes", args.seen == "yes",
↪  "logs.csv")
```

Listing 3.9: *Calling method generating log file with arguments*

Script to generate unlabelled and unseen dataset of 25000 messages would then look like as shown on listing 3.10.

```
python syslog_generator.py --number 25000 --labelled no --seen no
```

Listing 3.10: *Example how to run Syslog Generator from Powershell*

4 Implementation of Analysis Tools

This thesis deals with implementation of two proposed tools for analysis, Traffic Analysis tool [1] and Syslog Messages Analysis tool[2]. Each can be found in its own repository on Github.

Traffic Analysis tool deals with implementing machine learning algorithms on network security traffic. While, Syslog Messages Analysis tool deals with implementing machine learning algorithms on syslog messages.

This chapter shows implementation of each tool with description of important methods and parts of the code.

4.1 Structure of Proposed Tools

Both proposed tools, Traffic Analysis tool and Syslog Analysis tool, are using argument parser to call methods based on required action. This parser is same for both tools and it is described in this section.

Arguments parser is created by the code shown on listing 4.1.

```python
# Create parser
parser = argparse.ArgumentParser(prog="IDS_traffic_analysis.py")
parser.add_argument("--mode", dest="mode", choices=["research", "prod"],
↪  required=True)
parser.add_argument("--command", dest="command", choices=["train", "predict",
↪  "trainandpredict"], required=True)
parser.add_argument("--model", dest="model", choices=models_flags,
↪  required=True)
parser.add_argument("--source", dest="source", required=True)
```

Listing 4.1: *Argument Parser of proposed tools*

There are four different arguments created, each serves different purpose.

- - -mode *<research/prod>*
- - -command *<train/predict/trainandpredict>*
- - -model *<LR/K-NN/kSVM/NB/DTC/RFC/ocSVM/iF/LOF/K-Means/HC /ANN>*
- - -source *<filename>*

Argument *mode* specifies in which mode the tool should run. Options are research(*research*) and production(*prod*). Each mode differentiates in output it returns. Research mode returns metrics of predictions of machine learning models,

[1]Github: https://github.com/miroslav-siklosi/Traffic-Analysis-Tool
[2]Github: https://github.com/miroslav-siklosi/Syslog-Messages-Analysis

such as confusion matrix, accuracy, precision, recall and F1-Score. Production mode labels imported messages whether they are anomaly (*1*) or not (*0*). Labelled dataset is then saved into text file in the folder *Results*.

Argument *command* chooses what action should the tool do. Options are to train the machine learning model (*train*), predict anomalies based on learned weights (*predict*) or train and predict machine learning model on the same dataset (*trainand-predict*). Train and predict is specific command usable only in mode *research*. Using it in mode *prod* will return an error.

Argument *model* chooses which machine learning model to use. Options are Logistic Regression(*LR*), K-Nearest Neighbors(*K-NN*), Kernel SVM(*kSVM*), Naive Bayes(*NB*), Decision Tree Classifier(*DTC*), Random Forest Classifier(*RFC*), One-class SVM(*ocSVM*), Isolation Forest(*iF*), Local Outlier Factor(*LOF*), K-Means(K-*Means*), Hierarchical Classifier(*HC*) and Artificial Neural Network(*ANN*).

Last but not least argument is *source*. This argument specifies which file (dataset) should be imported into the tool for training or predictions. If the file is not in the same folder as the tool, full filepath needs to be specified.

Flowchart of Analyzing Tools

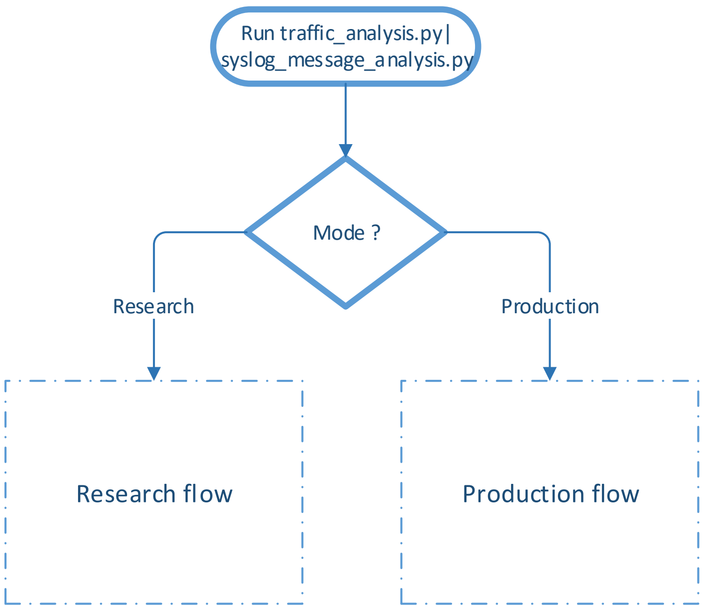

Fig. 4.1: Flowchart of the proposed tools

Depending on values of each argument, parser will decide which method should be called. At first, parser is looking at argument *mode* as shown by flowchart in figure 4.1. There are only few differences between modes *Research* and *Production*. Mode *Research* is returning performance metrics such as confusion matrix, accuracy, precision, etc. Hence, mode *Research* works only with labelled dataset. On other hand, mode *Production* is labelling the messages and returning labelled dataset. This mode is using labelled dataset for training and unlabelled dataset for predictions.

Depending on chosen mode, the tool will continue through respective flowchart. Flowchart of mode *Research* is shown in figure 4.2.

Command *Train* is for training of supervised methods on labelled dataset. In case a combination of mode *Research*, and command *Train* is chosen for unsupervised machine learning model, the tool will return an error as: *Unsupervised doesn't need training on labelled dataset*. If chosen model is supervised or deep learning, flow of the tool then continues by checking format of imported dataset. The tool will accept either *.csv* file as dataset or classifier (learned weights of a machine learning model) with extension of *.joblib* or *.h5*. If it is a classifier, it will immediately save it into folder *classifier* within the folder of the tool. If it is a dataset, it will import the dataset, preprocess it and send to a model of chosen machine learning model for training. Machine learning model will return trained classifier and it will be also saved into folder *classifier* within the folder of the tool.

Command *Predict* is to make the predictions based on classifier trained previously by command *Train*. The tool verifies whether the imported dataset has extension *.csv* and continues by importing and preprocessing it. Preprocessed dataset is then forwarded to chosen machine learning model. If the chosen model is *Unsupervised*, the model will train itself on matrix of independent variables X and predict the labels for the messages. If the chosen unsupervised machine learning models are *One-class SVM* or *Isolation Forest*, it will also edit label *1* to *0* and label *-1* to *1*. This is due to functionality of these two models, described in subsections 1.2.3 and 1.3.1.

If chosen model is *Supervised*, the tool will compare if it is a *Deep Learning* model or not. If yes, it will load classifier with extension *.h5*, makes the predictions, inverts predicted labels to either *0* or *1* and prints out performance results. If it is not a deep learning model, it will load classifier with *.joblib* extension, makes the predictions and prints out the performance results.

Third command called *Train and Predict* is specific only for *Research* mode. It is a specific command splitting the dataset into train set and test set. It is usable when there is one big labelled dataset and it would be uncomfortable to split it manually or when it is desired to quickly verify the functionality of a model (for example when tuning the hyper-parameters of the model).

Command *Train and Predict* will verify extension of imported dataset same way as command *Predict*. It continues by checking whether chosen model is *Unsupervised*

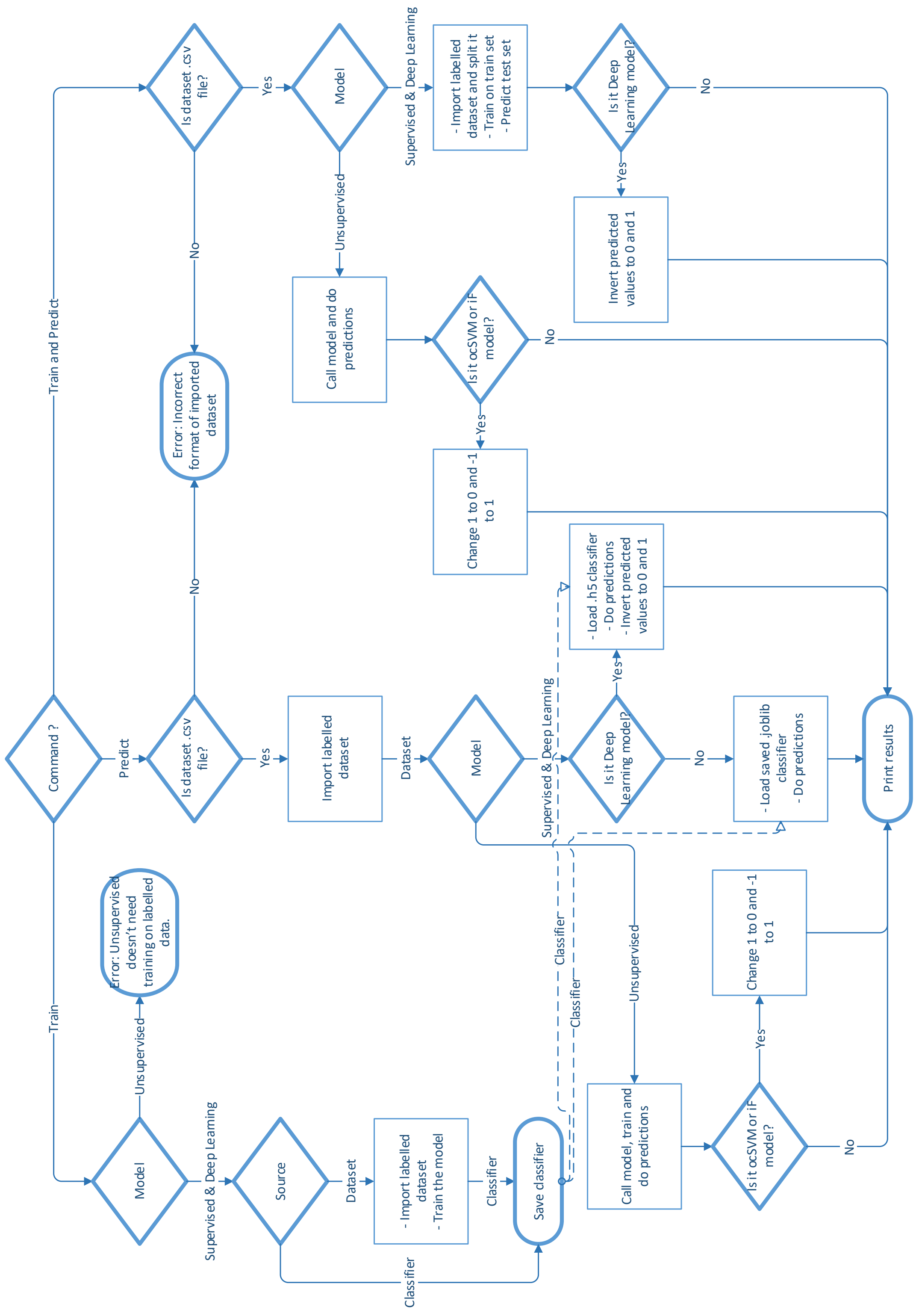
Command ?
Train and Predict
Predict
Train
Is dataset .csv file?
Yes
No
Model
Supervised & Deep Learning
Unsupervised
- Import labelled dataset and split it
- Train on train set
- Predict test set
Is it Deep Learning model?
No
Yes
Invert predicted values to 0 and 1
Call model and do predictions
Is it ocSVM or iF model?
No
Yes
Change 1 to 0 and -1 to 1
Error: Incorrect format of imported dataset
Is dataset .csv file?
Yes
No
Import labelled dataset
Dataset
Model
Supervised & Deep Learning
Is it Deep Learning model?
Yes
No
- Load .h5 classifier
- Do predictions
- Invert predicted values to 0 and 1
- Load saved joblib classifier
- Do predictions
Print results
Unsupervised
Classifier
Classifier
Call model, train and do predictions
Is it ocSVM or iF model?
Yes
No
Change 1 to 0 and -1 to 1
Model
Unsupervised
Error: Unsupervised doesn't need training on labelled data.
Supervised & Deep Learning
Source
Dataset
Classifier
- Import labelled dataset
- Train the model
Classifier
Save classifier

Fig. 4.2: Flowchart of Research mode

or not. If yes, the flow will follow similar path as in *Predict Unsupervised* mentioned above. If the chosen model is not unsupervised, it will continue by importing the dataset, splitting it into train and test set, training the model on training set and predicting the labels on test set. It then checks if the model is *Deep Learning*. If yes, it will invert predicted values of labels to *0* and *1* and prints out the performance results. If it is not deep learning model, it will print out the performance results right away.

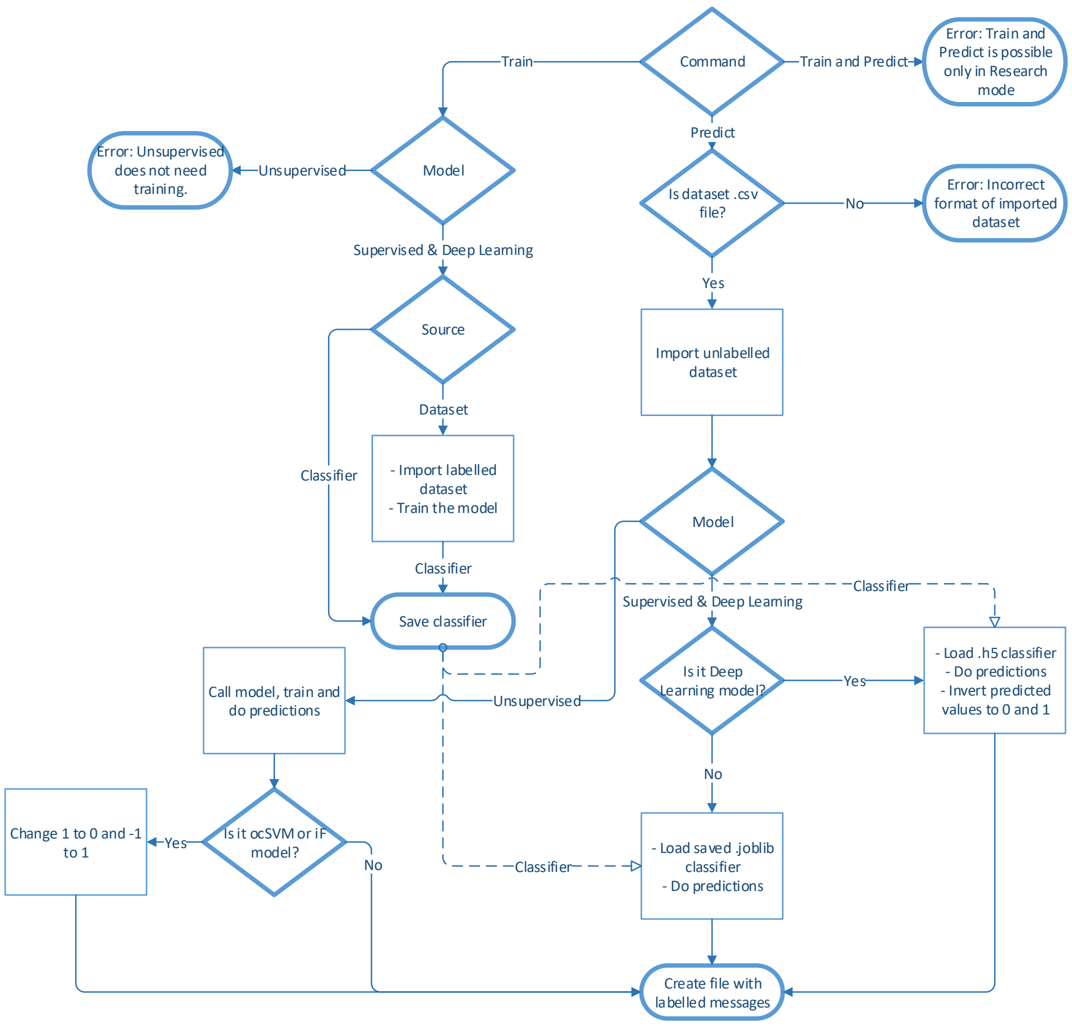
Command
Train
Train and Predict
Error: Train and Predict is possible only in Research mode
Predict
Model
Unsupervised
Error: Unsupervised does not need training.
Is dataset .csv file?
No
Error: Incorrect format of imported dataset
Supervised & Deep Learning
Yes
Source
Import unlabelled dataset
Dataset
Classifier
- Import labelled dataset
- Train the model
Model
Classifier
Save classifier
Classifier
Supervised & Deep Learning
Unsupervised
Call model, train and do predictions
Is it Deep Learning model?
Yes
- Load .h5 classifier
- Do predictions
- Invert predicted values to 0 and 1
No
Change 1 to 0 and -1 to 1
Yes
Is it ocSVM or iF model?
No
Classifier
- Load saved .joblib classifier
- Do predictions
Create file with labelled messages

Fig. 4.3: Flowchart of Production mode

The flowchart of mode *Production* is shown in figure 4.3. Command *Train* in this mode is pretty much the same as in mode *Research*. Command *Train and Predict* is disabled in this mode, hence it will return an error message. Command *Predict* is very similar to the one in mode *Research*. It verifies whether imported file has extension *.csv*. If not, it will return an error, if yes, it will continue by import of unlabelled dataset. This import is different than the one used in mode *Research* as it is returning only matrix of independent variables X, without the vector of dependent variables y as method to import labelled dataset.

When the dataset is imported and preprocessed, the tool will continue by checking if the chosen model is *Unsupervised* or not. If yes, it will follow similar path as in *Train* of *Research* mode. It will train the model and do the predictions. Then it will check if the chosen model is *One-class SVM*, *Isolation Forest* or neither of them. If one of them, it will change labels *1* to *0* and labels *-1* to *1*.

If the model is not unsupervised, it will check if it is *Deep Learning*. If yes, it will load classifier with extension *.h5*, makes the predictions based on weights in the classifier, inverts the labels to *0* and *1* and at the end, it will create file with labelled messages.

If the model is not deep learning, it will load the classifier with extension *.joblib*, predicts the labels and creates with labelled messages.

4.2 Data Preprocessing

Collected datasets can have various formats and contain various data. For example, there are two different datasets used in this thesis. First one is dataset CICIDS2017, which is containing benign traffic and some of the common attacks [65]. It is labelled dataset of generated traffic and it is parsed into multiple .csv files.

Second dataset used in this thesis is generated by tool *Syslog Generator*, mentioned above in section 3.

Data preprocessing is crucial for data before being applied into machine learning models. It contains many steps like Tokenization, removing stop-words, Normalization (Stemming and Lemmatization). Moreover, it also contains feature extraction (Word Embeddings) where the data are encoded into numerical feature vectors. However, data preprocessing is handled by NLP as described in subsection 4.2.1.

Further step that it is needed before applying the data into the machine learning model is the data splitting. Data splitting is a process that splits the data to training features, training labels and testing features, testing labels, as described is the subsection 4.2.2 below.

4.2.1 Natural Language Processing

All machine learning algorithms are based on certain mathematical calculations. In order for algorithm to be able to process dataset as a text, then this text has to be tokenized. Moreover, these tokens need to be normalized and then transformed into numbers. There are multiple options how to transfer words into numbers. However, a model Bag of Words was chosen. BoW is pretty easy to implement and it provides suitable outputs for machine learning, which are the reasons why this technique was chosen.

In Bag of Words model, each word is represented by its own variable (column). Set of all words used in dataset is then called Bag of Words.

If a word occurs in a certain log message, then its variable will have value 1, otherwise it will have value 0. If this word is in the log message twice, value will be 2, etc.

4.2.2 Data Splitting

Given the nature of this thesis, there are two types of datasets being used: labelled and unlabelled. To ensure proper data preprocessing and feature extraction, there are two methods created. Difference between labelled and unlabelled dataset is pretty clear: labelled dataset contains log message and label and unlabelled dataset contains only log message. However, since log message will be represented by matrix of independent variables X and label will be represented by dependent variable y, difference in importing and splitting of the data will be in returned variables: importing and splitting labelled dataset will return matrix X and vector y, importing unlabelled dataset will return only matrix X.

Since this thesis is dealing with two completely different datasets, there are two different methods to import and split data. Both methods are described bellow.

Preprocessing and Splitting CICIDS2017 Dataset

As mentioned above, there are two types of datasets: labelled and unlabelled. Method importing labelled dataset is described in listing 4.2. At first, dataset is loaded into memory and then it is splitted into matrix of independent variables X and vector of dependent variable y.

```python
# Load the dataset
dataset = pd.read_csv(filename)
# Splitting the dataset into independent and dependent variables
X = dataset.iloc[:, list(range(4, 6)) + list(range(7, 84))].values
y = np.array([0 if val == "BENIGN" else 1 for val in dataset.iloc[:,
↪   -1].values])
```

Listing 4.2: *Preprocessing labelled CICIDS2017 dataset*

To ensure only relevant values are imported into matrix X, there are few columns from dataset excluded. Namely, first column contains *Flow ID*, second one contains *Source IP address*, third one contains *Source port*, forth one contains *Destination IP address* and sixth one contains timestamp. All these values are irrelevant, hence are not imported into matrix X.

Importing values into vector y needs to be specified as well. Since dataset CICIDS2017 contains different written labels and this thesis is focusing on anomalies, anomalous labels are transformed into value *1* and benign traffic is transformed into value *0*.

Some of the values in this dataset are not processable. There are two columns which instead of numeric values contains nothing (*nan*) or infinity (*inf*). To take care of such data, there are two simple *for* loops, shown on listing 4.3. First one goes through both columns and calculates average value inside of them and also to find maximum value. Second one goes through the same columns again and replaces *nan* with average value and *inf* with maximum value.

```python
# Taking care of missing and incorrect data
SUM = 0
MAX = 0
COUNT = 0
# Count average values in columns 15 and 16
for i, row in enumerate(X):
    for j in [15, 16]:
        sx = str(float(X[i,j])).lower()
        if  (sx != "nan" and sx != "inf"):
            SUM = SUM + X[i,j]
            if X[i,j] > MAX:
                MAX = X[i,j]
            COUNT = COUNT + 1
AVERAGE = SUM/COUNT
for i, row in enumerate(X):
    for j in [15, 16]:
        sx = str(float(X[i,j])).lower()
        if  sx == "nan":
            X[i, j] = AVERAGE
        if  sx == "inf":
            X[i, j] = MAX
```

Listing 4.3: *Missing and incorrect data preprocessing*

Once the data are processable, they are ready to be forwarded into machine learning models. However, if there is chosen argument *mode* as *trainandpredict*, the data needs to be split into train and test set. This is done by function *train_test_split* from Sklearn library as shown on listing 4.4. To ensure model is trained properly, 80% of dataset is assigned to training and 20% to testing (a.k.a prediction).

```python
# Splitting the dataset into the Training set and Test set
if split:
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,
    ↪  random_state = 0)
else:
    X_train = X
    X_test = X
    y_train = y
    y_test = y
```

Listing 4.4: *Splitting the dataset into the Training and Test set*

Because this method is always returning variables (*X_train*, *X_test*, *y_train* and *y_test*), values from matrix X and vector y are allocated into them if chosen mode is not *trainandpredict*.

Preprocessing of unlabelled dataset is similar to preprocessing of labelled dataset. Missing and incorrect data are processed same way (see listing 4.3). Only difference is that this method returns only matrix X, so loading and spliting of the dataset is done by code shown on listing 4.5.

```python
# Load the dataset
dataset = pd.read_csv(filename)
# Load dataset into matrix of independant variables
X_test = dataset.iloc[:, list(range(4, 6)) + list(range(7, 84))].values
```

Listing 4.5: *Preprocessing unlabelled CICIDS2017 dataset*

Preprocessing and Splitting Dataset of Syslog Messages

Preprocessing of the dataset of syslog messages is similar to preprocessing of CICIDS2017 dataset. However, there are some key differences since both datasets are very different.

As shown in listing 4.6, dataset is loaded into memory by the same function *pd.read.csv*, but with different parameters. As a delimiter is chosen tabular, quoting is disabled, dates are parsed (to be removed by BoW more easily) and names of the columns are specified as *Syslog* and *Anomaly*. Matrix X is then created using method *extract_BoW*, described in section 4.2.1 and shown in listing 4.8. Vector y is created by taking out last column from imported dataset, also shown in listing 4.6.

```python
# Importing the dataset
dataset = pd.read_csv(filename, delimiter = "\t", quoting = 3, header = None,
↪  parse_dates = True, names = ["Syslog", "Anomaly"])
# Splitting the dataset into independent and dependent variables
X = extract_BoW(dataset["Syslog"])
y = dataset.iloc[:, -1].values
```

Listing 4.6: *Preprocessing labelled dataset of syslog messages*

Preprocessing of unlabelled dataset is practically the same as preprocessing of labelled dataset, but without importing vector y. See listing 4.7.

```python
# Importing the dataset
dataset = pd.read_csv(filename, delimiter = "\t", quoting = 3, header = None,
↪  parse_dates = True, names = ["Syslog"])
# Calling method BoW to create matrix X
X_test = extract_BoW(dataset["Syslog"])
```

Listing 4.7: *Preprocessing unlabelled dataset of syslog messages*

Matrix X and vector y are then split by same method as shown on listing 4.4 if chosen mode of the tool is *trainandpredict*.

Method creating Bag of Words shown on listing 4.8 creates corpus called *syslogs*. Before put into corpus, lines of the dataset are then processed one-by-one and stripped down of not needed symbols. At first, MAC Addresses, all symbols except letters and spaces are removed. The letters are then lower-cased. This creates string of lower case words divided by spaces. Words are then put in apostrophes and inserted into function *PorterStemmer* for normalization. The *PorterStemmer* adjusts words into their root shape, while stopwords such as *the*, *and*, etc. are being removed. Last step before adding a line into a corpus is merging words back into string divided by space.

Words *asa* and *fw* are removed from created corpus as they are in every single message and are irrelevant. The corpus is also reduced to contain only 200 of most used words. At the end, the *toarray* function is used to transfer the feature vector into two-dimensional array *[n_samples, n_features]* so machine learning model can accept it.

```python
def extract_BoW(syslogs_column):
    syslogs = []
    for line in syslogs_column:
        syslog = re.sub(r"(?:[0-9a-fA-F]:?){12}", "", line) # remove MAC
        ↪  Addresses
        syslog = re.sub('[^a-zA-Z]', ' ', syslog) # keep letters and spaces
        syslog = syslog.lower()
        syslog = syslog.split() # split text into words
        syslog = [PorterStemmer().stem(word) for word in syslog if not word in
        ↪  set(stopwords.words('english'))] # PS - keep to root of the words
        syslog = ' '.join(syslog) # merge words back into string
        syslogs.append(syslog)
    stop_words = text.ENGLISH_STOP_WORDS.union({"asa", "fw"}) # remove asa and
    ↪  fw from BoW
    cv = CountVectorizer(max_features = 200, stop_words = stop_words) # consider
    ↪  only 200 most used words
    X = cv.fit_transform(syslogs).toarray()
    return X
```

Listing 4.8: *Method Bag of Words*

4.3 Machine Learning Structure and Models

After the data are preprocessed, they are ready to be fed to the machine learning models. Almost all models are implemented using Python library for machine learning *Scikit-learn* [66]. Only Artificial Neural Network model is implemented using Python library *Keras* [67]. However, since the thesis deals with classification problem then the implementation of classification machine learning models will be described. As an unsupervised learning the clustering algorithms will be used. The machine learning models described below will cover both tools of Syslog Analysis tool and Traffic Analysis tool, just some parameters will be adjusted based on the need of every tool.

4.3.1 Logistic Regression Model

The model has slightly different parameters in Syslog Analysis tool than in Traffic Analysis tool. Parameters were chosen by manual tests and comparison of the precision. In Syslog Analysis tool, the model uses no penalization, random stat *0* and default number of iterations: *100*. In Traffic Analysis tool, the model uses penalization *l2*, random state *0* and *1000* iterations. This tool, when trained on *CICIDS2017* dataset, reached maximum number of iterations even when the number of iterations was *20000*, the tool was taking too much time to process and results were not different than with 1000 iterations. Due to that, the number of iterations was set back to *1000*. Implementation is shown on listing 4.9.

```
classifier_LR = LogisticRegression(penalty='none', random_state = 0)
classifier_LR.fit(data["X_train"], data["y_train"])
```

Listing 4.9: *Logistic Regression Model*

Predictions are then made using function *predict* as shown on listing 4.10.

```
y_pred = classifier.predict(data["X_test"])
```

Listing 4.10: *Prediction of supervised model*

4.3.2 K-Nearest Neighbor Model

The model uses function *Hamming* and utilizing 5 neighbors. Hamming function was chosen as it is recommended for classification problems as mentioned in subsection 1.2.2. Number of neighbors was chosen randomly, only following simple rule to not use multiple of number of classes. Implementation is shown on listing 4.11.

```
classifier_KNN = KNeighborsClassifier(n_neighbors = 5, metric = 'hamming')
classifier_KNN.fit(data["X_train"], data["y_train"])
```

Listing 4.11: *K-Nearest Neighbor Model*

Predictions are then made using function *predict* as shown on listing 4.10.

4.3.3 Kernel SVM Model

The model uses kernel function *Radial Basis Function* and random state 0. This function was chosen as it has lower performance requirements. Implementation is shown on listing 4.12.

```
classifier_kSVM = SVC(kernel = 'rbf', random_state = 0)
classifier_kSVM.fit(data["X_train"], data["y_train"])
```

Listing 4.12: *Kernel SVM Model*

Predictions are then made using function *predict* as shown on listing 4.10.

4.3.4 One-class SVM Model

The model uses kernel function *Radial Basis Function*. This function was chosen as it has lower performance requirements and it is also default function for this model. Fitting of features and predictions are made at once using function *fit_predict*. Implementation is shown on listing 4.13.

```python
ocSVM = OneClassSVM(kernel="rbf")
y_pred = ocSVM.fit_predict(data["X"])
```

Listing 4.13: *One-class SVM Model*

Since the model returns values *1* and *-1* for normal and anomalous data points respectively (see subsection 1.2.3 for explanation), these values need to be adjusted in order to be applicable in functions calculating metrics and performance. This is done by simple *for* loop as shown on listing 4.14.

```python
for i, row in enumerate(y_pred):
    if y_pred[i] == 1:
        y_pred[i] = 0
    else:
        y_pred[i] = 1
```

Listing 4.14: *Loop to adjust predicted values*

4.3.5 Naive Bayes Model

The model uses the Gaussian Naive Bayes algorithm, which is specifically written for classification problems, hence effective for challenge of this thesis. Implementation is shown on listing 4.15.

```python
classifier_NB = GaussianNB()
classifier_NB.fit(data["X_train"], data["y_train"])
```

Listing 4.15: *Naive Bayes Model*

Predictions are then made using function *predict* as shown on listing 4.10.

4.3.6 Decision Tree Classification Model

The model uses default number of samples for a node to be leaf of *1*, minimum of number of samples to split internal node of *2* and strategy to choose split of each node, splitter, is chosen as *best*. To measure the quality was chosen function *entropy* and it uses random state of *0*. These parameters were chosen by manual tests and comparison of the precision. Implementation is shown on listing 4.16.

```
classifier_DTC = DecisionTreeClassifier(criterion = 'entropy', random_state = 0)
classifier_DTC.fit(data["X_train"], data["y_train"])
```

Listing 4.16: *Decision Tree Classification Model*

Predictions are then made using function *predict* as shown on listing 4.10.

4.3.7 Random Forest Classification Model

The model uses default number of samples for a node to be leaf of *1* and minimum of number of samples to split internal node of *2*. To measure the quality was chosen function *entropy*, number of trees is *50* and random state *0*. These parameters were chosen by manual tests and comparison of the precision. Implementation is shown on listing 4.17.

```
classifier_RFC = RandomForestClassifier(n_estimators = 50, criterion =
↪  'entropy', random_state = 0)
classifier_RFC.fit(data["X_train"], data["y_train"])
```

Listing 4.17: *Random Forest Classification Model*

Predictions are then made using function *predict* as shown on listing 4.10.

4.3.8 Isolation Forest Model

The model in default utilizes 100 estimators with automatically choosing maximum number of samples. Random state was chosen *0*. These parameters are default ones for this model. Fitting of features and predictions are made at one once using function *fit_predict*. Implementation is shown on listing 4.18.

```
iF = IsolationForest(random_state=0)
y_pred = iF.fit_predict(data["X"])
```

Listing 4.18: *Isolation Forest Model*

Since model returns values *1* and *-1* for normal and anomalous data points respectively (see subsection 1.3.1 for explanation), values need to be adjusted in order to be applicable in functions calculating metrics and performance. This is done by same simple *for* loop as previously shown on listing 4.14.

4.3.9 Local Outlier Factor Model

The model uses function *Hamming* and utilizes 20 neighbors (default value). This function is chosen based on the same recommendation as model K-Nearest Neighbors, subsection 4.3.2. Fitting of features and predictions are made at one once using function *fit_predict*. Implementation is shown on listing4.19.

```python
lof = LocalOutlierFactor(metric = 'hamming')
y_pred = lof.fit_predict(data["X"])
```

Listing 4.19: *Local Outlier Factor Model*

4.3.10 K-Means Model

The model is using method *k-means++* to initial cluster centers, algorithm *full*, creating 2 clusters (normal and anomalous data points) and random state of *42*. The parameters were chosen by manual tests and comparison of the precision. Fitting of features and predictions are made at once using function *fit_predict*. Implementation is shown on listing 4.20.

```python
kmeans = KMeans(n_clusters = 2, init = 'k-means++', algorithm = 'full',
↪   random_state = 42)
y_pred = kmeans.fit_predict(data["X"])
```

Listing 4.20: *K-Means Model*

4.3.11 Hierarchical Clustering Model

The model's chosen approach is *Agglomerative* and creates 2 clusters (normal or anomalous data points). Metric used to calculate linkage between clusters is *euck-lidean* and linkage criterion used is *ward*. The parameters were chosen by manual tests and comparison of the precision. Fitting of features and predictions are made at once using function *fit_predict*. Implementation is shown on listing 4.21.

```python
hc = AgglomerativeClustering(n_clusters = 2, affinity = 'euclidean', linkage =
↪   'ward')
y_pred = hc.fit_predict(data["X"])
```

Listing 4.21: *Hierarchical Clustering Model*

4.3.12 Artificial Neural Network Model

Apart from the previous models, Artificial Neural Network model is implemented using Python library *Keras*. Implementation of this model is shown in listing 4.22.

```python
# Initialising the ANN
classifier_ANN = Sequential()
# Adding the input layer and the first hidden layer
classifier_ANN.add(Dense(activation="relu", input_dim=200, units=101,
↪  kernel_initializer="uniform"))
# Adding the hidden layers
h_layers = 10
for i in range(h_layers):
    classifier_ANN.add(Dense(activation="relu", units=101,
    ↪  kernel_initializer="uniform"))
# Adding the output layer
classifier_ANN.add(Dense(activation="sigmoid", units=2,
↪  kernel_initializer="uniform"))
# Compiling the ANN
classifier_ANN.compile(optimizer = 'adam', loss =
↪  'sparse_categorical_crossentropy', metrics = ['accuracy'])
# Fitting the ANN to the Training set
classifier_ANN.fit(data["X_train"], data["y_train"], batch_size = 10, epochs =
↪  10)
```

Listing 4.22: *Artificial Neural Network Model*

At first, the model needs to be initialized. Second, the first (input) layer needs to be created. In Syslog Analysis tool, this layer has *200* input dimension, because input matrix X has 200 columns. Parameter *units* specifies how many "neurons" will the input layer have and it is set to *101*, which is half of the number of input variables and +1. On other hand, in Traffic Analysis tool, this layer has *79* input dimensions and *39* units. Parameter *activation* was chosen to be *relu* and parameter *kernel_initializer* was chosen to be *uniform*.

After input layer the hidden layers appear. There are 10 hidden layers. In Syslog Analysis tool, all of them have *101* "neurons", same as input layer, and *39* units in Traffic Analysis tool. Parameters *activation* and *kernel_initializer* are same as in input layer.

Last layer is the output layer, which uses same *kernel_initializer* as all previous layers, but as an *activation* is chosen function *sigmoid*. This layer has only two neurons, since there are only two output values: 1 (as anomaly) and 0 (as not anomaly).

The model is then compiled using function *adam* as an *optimizer*. *Loss* is set as *sparse_categorical_crossentropy* and *metrics* is set to *accuracy*.

At the end, data are fit into the model for training. Number of epochs is chosen to *10*, same as batch size.

Predictions are then made by simple function *predict*, show on listing 4.23.

```
y_pred = classifier.predict(data["X_test"])
```

Listing 4.23: *Prediction of ANN model*

4.4 Machine Learning Metrics and Performance

To compare the performance of the machine learning models, there are different performance comparison methods implemented. First one is confusion matrix, which is a table that shows summary of correct and incorrect predictions of a model. Confusion matrix is printed out by simple line of code shown on listing 4.24 [68].

```
print(confusion_matrix(data["y_test"], y_pred))
```

Listing 4.24: *Print Confusion Matrix*

Next ones are metrics such as *accuracy, precision, recall* and *F1-score*. These are calculated and printed out by simple method shown on listing 4.25.

```
def print_metrics(model, data, y_pred):
    # accuracy: (tp + tn) / (p + n)
    accuracy = accuracy_score(data["y_test"], y_pred)
    print(f"Accuracy of Machine Learning model {model} is", accuracy)
    # precision tp / (tp + fp)
    precision = precision_score(data["y_test"], y_pred)
    print(f"Precision of Machine Learning model {model} is", precision)
    # recall: tp / (tp + fn)
    recall = recall_score(data["y_test"], y_pred)
    print(f"Recall of Machine Learning model {model} is", recall)
    # f1: 2 tp / (2 tp + fp + fn)
    f1 = f1_score(data["y_test"], y_pred)
    print(f"F1-Score of Machine Learning model {model} is", f1)
```

Listing 4.25: *Method printing metrics*

As a manual method to compare predicted labels with actual labels is shown on listing 4.26. This method prints out initial message with label. Below each labelled message is written whether the prediction was correct or not based on comparison with actual label.

```python
def print_prediction_result(data, y_pred, input_filepath):
    # [X_test, y_pred] Prediction is correct/Prediction is NOT correct
    y_test = data['y_test']
    np.set_printoptions(threshold=np.inf, linewidth=np.inf)
    with open(f"Results/prediction_result.csv", 'w') as f:
        with open(input_filepath) as input_file:
            for index, input_line in enumerate(input_file):
                if index == 0:
                    continue
                i = index - 1
                f.write(f"{input_line.rstrip()}, {y_pred[i]}, ")
                if y_test[i] == y_pred[i]:
                    f.write("Prediction is correct\n")
                else:
                    f.write("Prediction is NOT correct\n")
    print(f"Prediction results saved into prediction_result.csv")
```

Listing 4.26: *Print predictions into a text file*

Example of labelled output can be found on listing 4.27.

```
May 17 2020 16:44:07 FW15 : %ASA-4-400038: IPS:6100 RPC Port Registration
↪  150.60.80.114 to 21.56.146.99 on interface GigabitEthernet0/3 0
Prediction is correct
May 17 2020 16:44:08 FW01 : %ASA-6-304004: URL Server 175.136.4.15 request
↪  failed URL https://nguyen.com/ 1
Prediction is NOT correct
```

Listing 4.27: *Example of labelled messages with results*

Description of confusion matrix and metrics can be found in chapter 6.

5 Functionality of Tools

This chapter briefly describes how to run each of the proposed tools. Description shows examples on how to run the tools using command line and how to use arguments of each tool. There is also brief description of what all arguments are utilized for.

5.1 Syslog Generator

Syslog Generator is a tool to generate Cisco ASA system log messages. Generated messages can be either labelled or not, and can be generated from within seen or unseen message templates. Seen messages will be used to train machine learning models and unseen messages will be used to test how well machine learning models are trained and how good are they adapting to new, unseen variables.

Tool can be run by simple command from the tool's folder:

```
python syslog_generator.py
```

Listing 5.1: *Script to run Syslog Generator*

By default, generated dataset will contain 1000 messages, it will be labelled and it will use only "seen" message templates. However, these arguments can be changed. There are three different arguments as following:

- - -number <*number of lines*>; default *1000*
- - -labelled <*yes/no*>; default *yes*
- - -seen <*yes/no*>; default *yes*

So for example, unlabelled and unseen dataset of 25000 messages will be generated using following command:

```
python syslog_generator.py --number 25000 --labelled no --seen no
```

Listing 5.2: *Script to run Syslog Generator with certain parameters*

5.2 Traffic Analysis Tool

Traffic Analysis tool is analyzing CICIDS2017 dataset. This dataset is already parsed into .csv file so it is easier to process. However, dataset is already labelled. This tool is created to detect security anomalies using Machine Learning methods.

Tool can be run by command line from the tool's folder. There are no default arguments, so everything needs to be specified, as shown in listing 5.3.

```
python traffic_analysis.py --mode <> --model <> --command <> --source <>
```

Listing 5.3: *Script to run Traffic Analysis Tool*

As it can be seen in listing 5.3, there are four arguments and all of them are mandatory. The options are as following:

- - -mode *<research/prod>*
- - -model *<LR/K-NN/kSVM/NB/DTC/RFC/ocSVM/iF/LOF/K-Means/HC /ANN>*
- - -command *<train/predict/trainandpredict>*
- - -source *<filename>*

Meaning of each argument and their options are described in section 4.1.

5.3 Syslog Analysis Tool

Syslog Analysis tool is analyzing syslog messages created by Syslog Generator. The tool can be run by command line from the tool's folder. There are no default arguments, so everything needs to be specified, see listing 5.4.

```
python syslog_messages_analysis.py --mode <> --model <> --command <> --source <>
```

Listing 5.4: *Script to run Syslog Analysis Tool*

As it can be seen in listing 5.4, there are four arguments and all of them are mandatory. The options are as following:

- - -mode *<research/prod>*
- - -model *<LR/K-NN/kSVM/NB/DTC/RFC/ocSVM/iF/LOF/K-Means/HC /ANN>*
- - -command *<train/predict/trainandpredict>*
- - -source *<filename>*

Meaning of each argument and their options are described in section 4.1.

6 Comparison of Performance of Machine Learning Models

This chapter compares the performance of twelve different machine learning algorithms mentioned above. Each has a different approach. For results comparison, the confusion matrix and metrics calculated from it such as *accuracy*, *precision*, *recall* and *F1-score* are used.

Confusion matrix is a table showing summary of predicted labels compared to actual labels.

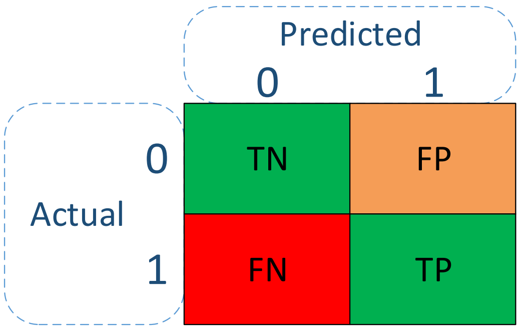

Fig. 6.1: Confussion Matrix

As shown in figure 6.1, confusion matrix's fields have certain names. As a *positive* result, the label is considered as an *anomaly*, and as a *negative* result as a label *not anomaly*. Explanation of each field is below:

- *True Positive* means predicted was positive label (*anomaly*) and it was correct.
- *False Positive* means predicted was positive label (*anomaly*), where actually was supposed to be negative label (*not anomaly*). It is also called *Type 1 Error* or *False alarm*.
- *False Negative* means that negative label (*not anomaly*) was predicted and it was incorrect. It is also called *Type 2 Error*. This one is critical as it shows anomaly was missed.
- *True Negative* means predicted negative label (*not anomaly*) was correct.

Metrics calculated from confusion matrix are *Accuracy, Precision, Recall* and *F1-Score*. Accuracy (also called Classification Rate) is a ratio of all correct predictions to all predictions and is calculated by equation shown below [68]:

$$Accuracy = \frac{TP + TN}{P + N}. \tag{6.1}$$

Precision is a ratio of how many predicted anomalies are correct and is calculated by following equation:

$$Precision = \frac{TP}{TP + FP}. \tag{6.2}$$

Recall (sometime also called Sensitivity) is a ratio of how many actual anomalies are predicted correctly. This metric will be key for comparison of implemented tools. Recall is calculated by following equation:

$$Recall = \frac{TP}{TP + FN}. \tag{6.3}$$

F1-Score (also called F-measure) is harmonic mean of Recall and Precision. It will be always closer to smaller value of either Recall or Precision. F1-Score is calculated by following equation:

$$F1\text{-}Score = \frac{2 * TP}{2 * TP + FP + FN} = \frac{2 * Recall * Precision}{Recall + Precision}. \tag{6.4}$$

Implemented machine learning models are tested on two different datasets: CICIDS2017 dataset and dataset of Cisco ASA syslog messages. However, CICIDS2017 was initially chosen only as temporary dataset to test models during their implementation.

6.1 Traffic Analysis Performance

Dataset CICIDS2017 contains generated benign traffic and some of the most common attacks. Dataset was generated for five days in a row, from Monday, July 3 to Friday, July 7 in 2017. On Monday, there was only benign traffic, while for the rest of the days there were different kinds of attacks simulated.

The Traffic Analysis tool was applied on logs from the day, when various kinds of DoS/DDoS attacks were caputured: *DoS Slowloris*, *DoS Slowhttptest*, *DoS Hulk* and *DoS GoldenEye*. As this dataset contains almost 700000 lines, is very demanding for computing power. Hence, only 100000 randomly chosen lines were used. The dataset was split to train and test set in ratio of 80% for train set and 20% for test set.

Results of the machine learning performance can be seen in Tab. 6.1. The most important metric for anomaly detection is *Recall*. The results show that the most accurate predictions were made by Supervised machine learning models. Almost 100% accurate was model *Decision Tree Classifier*, followed closely by models *Naive Bayes*, *Random Forest Classifier* and *K-Nearest Neighbors*. All were able to predict almost all anomalies correctly. Surprisingly, model *Logistic Regression* was pretty accurate as well.

		Metrics			
		Accuracy	Precision	**Recall**	F1-Score
Supervised Models	LR	0.9161	0.8945	**0.8696**	0.8818
	K-NN	0.9927	0.9835	**0.9964**	0.9899
	kSVM	0.9106	0.9673	**0.7783**	0.8626
	NB	0.5118	0.4236	**0.983**	0.592
	DTC	0.9992	0.9957	**0.999**	0.9988
	RFC	0.9988	0.9985	**0.9981**	0.9983
Unsupervised Models	ocSVM	0.6456	0.512	**0.6986**	0.5909
	iF	0.6417	0.5406	**0.1461**	0.2301
	LOF	0.59	0.1626	**0.0287**	0.0488
	K-Means	0.6873	0.613	**0.3972**	0.4821
	HC	0.2201	0.1786	**0.31**	0.2263
Deep Learning	ANN	0.6396	0	**0**	0

Tab. 6.1: Performance results of Machine Learning Models on CICIDS2017 dataset

Among the Unsupervised machine learning models was the most accurate model *One-class SVM*, correctly identifying almost 70% of anomalies. Model *K-Means* correctly identified almost 40% of anomalies and model *Hierarchical Clustering* only 31%, which are not very good performance results. Very poor performance in identifying anomalies was showed by models *Isolation Forest* and *Local Outlier Factor*.

However, the worst performance was showed by *Artificial Neural Network* model. This model labelled all test messages as not anomalies. For better results, there is a need of more "in-depth" tuning of parameters of this model.

Graphical comparison of results can be also seen in figure 6.2.

6.2 Syslog Messages Analysis Performance

Syslog Message Analysis tool was trained and tested on several various datasets generated by Syslog Generator, described in chapter 3. Generated datasets were containing 5000 or 50000 messages, and also datasets containing seen or unseen syslog messages.

There were three tests performed. The first test was performed on dataset containing 50000 seen messages and it was done in mode *Research* and command *Train and Predict*. That means that the dataset was split into train set of 80% of the messages and test set of 20% of the messages.

The results of the first test are shown in Tab. 6.2. All supervised machine learning models and also deep learning model predicted all anomalies correctly. Even

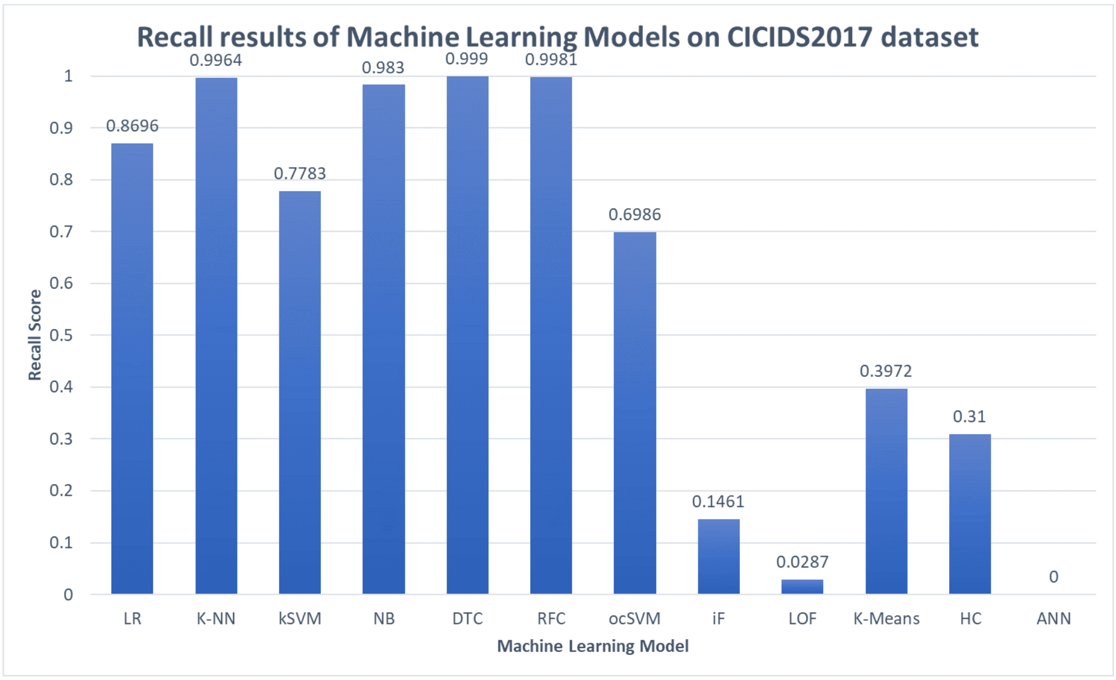

Fig. 6.2: Recall results of Machine Learning Models on CICIDS2017 dataset

		Metrics			
		Accuracy	Precision	**Recall**	F1-Score
Supervised Models	LR	1	1	**1**	1
	K-NN	1	1	**1**	1
	kSVM	1	1	**1**	1
	NB	0.9858	0.7852	**1**	0.8797
	DTC	1	1	**1**	1
	RFC	1	1	**1**	1
Unsupervised Models	ocSVM	0.4985	0.0595	**0.6003**	0.1081
	iF	0.9494	0	**0**	0
	LOF	0.9514	0.9813	**0.0415**	0.0796
	K-Means	0.5281	0.044	**0.4013**	0.0793
	HC	0.4624	0.0645	**0.7112**	0.1182
Deep Learning	ANN	1	1	**1**	1

Tab. 6.2: Performance results of syslog analysis on 50k of seen messages

though model *Naive Bayes* correctly predicted all anomalies, it also returned few false alarms. However, this is not a big issue as this tools is focusing on identifying anomalies.

Unsupervised machine learning models were not as successful as supervised models. Most successful in identifying anomalies was model *Hierarchical Clustering*. Its

Recall is *0.7112*, which means it correctly predicted *71.12%* of anomalies. However, as it can be seen from its Precision and F1-Score, which are very low, it returned a lot of false alarms. The second most successful unsupervised model was *One-clas SVM*, which correctly predicted *60%* of anomalies, but same as model *Hierarchical Clustering*, it returned a lot of false alarms. The third model is *K-Means*, which correctly predicted *40%* of anomalies, but also returned a lot of false alarms. Unsupervised models *Local Outlier Factor* and *Isolation Forest* totally failed in predictions.

The Recall results of the first test can be also seen on graph shown in figure 6.3.

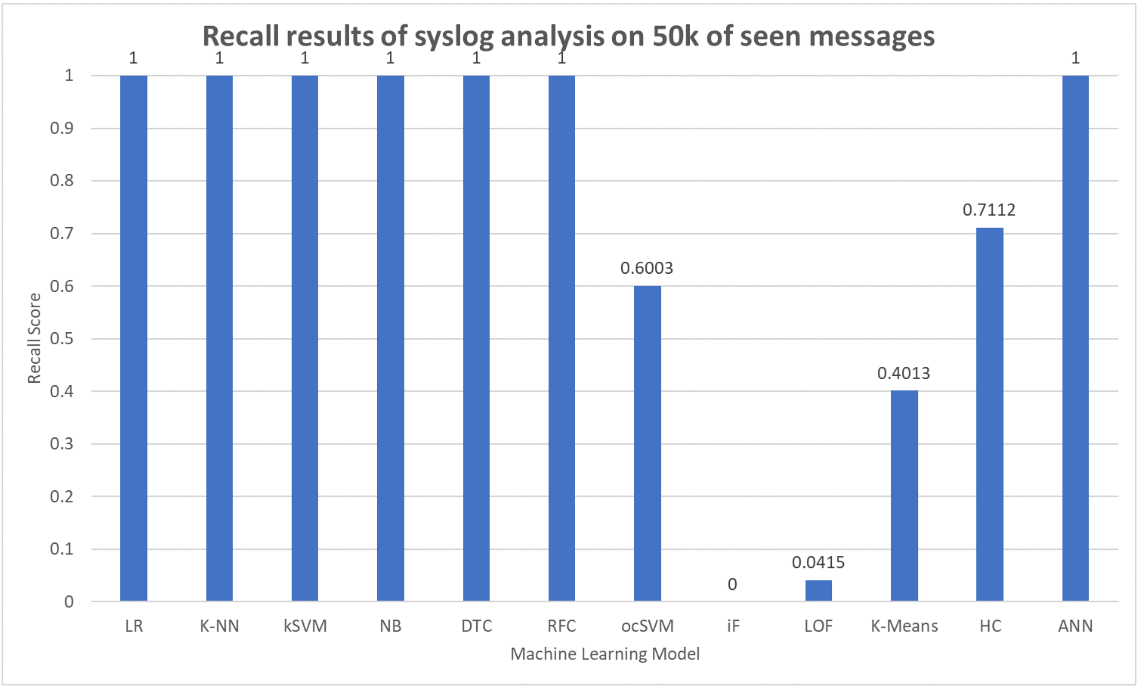

Fig. 6.3: Recall results of syslog analysis on 50k of seen messages

The second test was done to see how the machine learning models will perform when making predictions on messages they have not seen before. All models were trained on dataset of 5000 of seen messages and predictions were made on different dataset of 5000 of unseen messages.

The results of this test are shown in Tab. 6.3. The most successful in predicting anomalies were unsupervised machine learning models. Namely, it was model *Hierarchical Clustering*, which correctly predicted *87.1%* anomalies, and model *One-class SVM*, which correctly predicted *86.29%* anomalies. The third was model *K-Means*, which correctly predicted *73.39%* anomalies. Fourth most successful model was supervised *Decision Tree Classifier*, almost tied up by unsupervised model *Local Outlier Factor*, correctly predicting *47.98%* and *45.56%* anomalies respectively. However, these models also returned a lot of false alarms.

		Metrics			
		Accuracy	Precision	**Recall**	F1-Score
Supervised Models	LR	0.9208	0.1864	**0.1774**	0.1818
	K-NN	0.9556	0.7407	**0.1613**	0.2649
	kSVM	0.9532	1	**0.0546**	0.1069
	NB	0.9504	0	**0**	0
	DTC	0.7856	0.1121	**0.4798**	0.1817
	RFC	0.9216	0.2764	**0.3589**	0.3122
Unsupervised Models	ocSVM	0.5126	0.0818	**0.8629**	0.1439
	iF	0.9524	1	**0.0403**	0.0775
	LOF	0.945	0.4469	**0.4556**	0.4511
	K-Means	0.4916	0.0685	**0.7339**	0.1253
	HC	0.4984	0.0802	**0.871**	0.1469
Deep Learning	ANN	0.9504	0	**0**	0

Tab. 6.3: Performance results of syslog analysis trained on 5000 messages

The rest of the models were not very accurate in predicting anomalies. Models *Naive Bayes* and *Artificial Neural Network* did not even predict a single message as anomaly, which means complete failure in this test for these models.

Visualization of recall results for this test can be seen on graph shown in figure 6.4.

In the third test, machine learning models were trained on dataset of 50000 syslog messages and tested on the same dataset of 5000 unseen messages to see how increasing training dataset affects the performance.

The results of the third test are shown in Tab. 6.4. In this test, the most successful machine learning model was supervised model *Decision Tree Classifier* with *93.95%* correctly predicted anomalies. The second were two unsupervised models: *Hierarchical Clustering* with *85.66%* correctly predicted anomalies and *One-class SVM* with *85.3%* correctly predicted anomalies. However, all three models returned a lot of false alarms, as seen from their Precision.

Unsupervised model *K-Means* was fourth with only *46.1%* correctly predicted anomalies and precision only *5.61%*. It was almost tied up by supervised model *Random Forest Classifier*, with only *42.74%* correctly predicted anomalies and precision only *7.87%*. Supervised model *Logistic Regression* correctly predicted only *31.85%* with precision only *5.95%*.

The rest of the models failed the test, with models *Kernel SVM* and *Local Outlier Factor* correctly predicting *0* anomalies.

Visualization of recall results of the third test can be seen in figure 6.5.

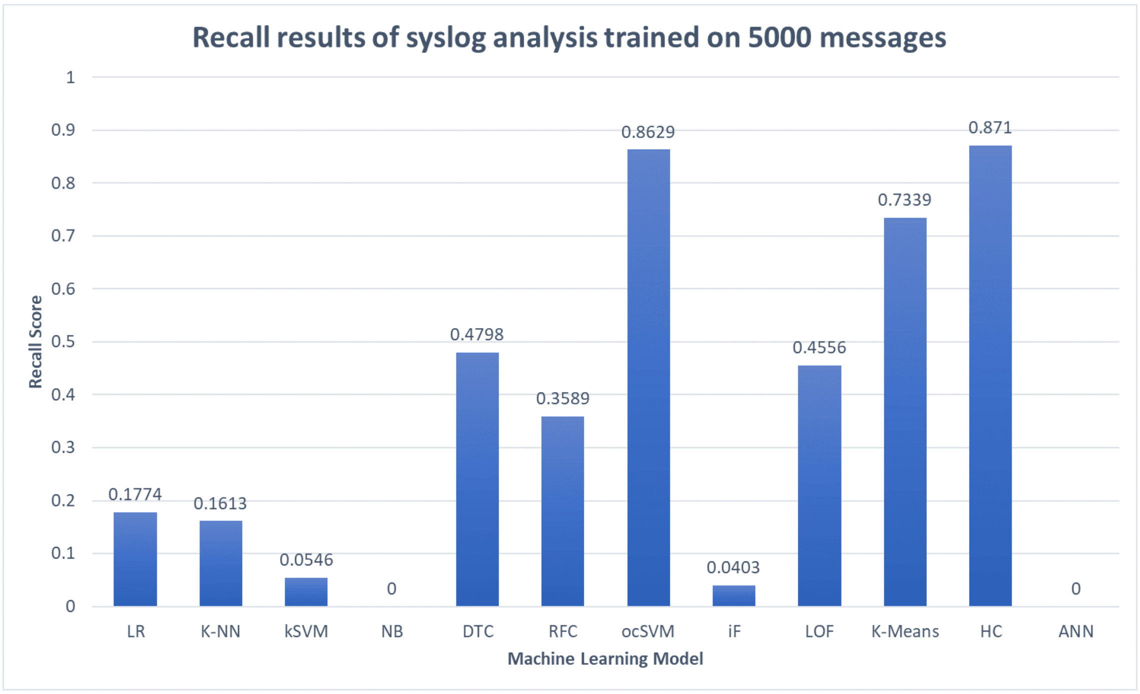

Fig. 6.4: Recall results of syslog analysis trained on 5000 messages

		Metrics			
		Accuracy	Precision	**Recall**	F1-Score
Supervised Models	LR	0.7164	0.0595	**0.3185**	0.1003
	K-NN	0.7962	0.0187	**0.0604**	0.0286
	kSVM	0.786	0	**0**	0
	NB	0.863	0.0822	**0.1734**	0.1115
	DTC	0.306	0.0632	**0.9395**	0.1184
	RFC	0.7232	0.0787	**0.4274**	0.1328
Unsupervised Models	ocSVM	0.4583	0.0729	**0.853**	0.1342
	iF	0.9562	1	**0.1101**	0.1983
	LOF	0.937	0	**0**	0
	K-Means	0.5915	0.0561	**0.461**	0.1
	HC	0.4956	0.0782	**0.8566**	0.1433
Deep Learning	ANN	0.7536	0.0205	**0.0847**	0.033

Tab. 6.4: Performance results of syslog analysis trained on 50000 messages

Comparison of how the size of the dataset affects training of machine learning model is shown in figure 6.6. Enlargement of the training dataset increased the performance of almost all implemented machine learning models. The biggest improvement was observed for supervised model *Decision Tree Classifier*, whose

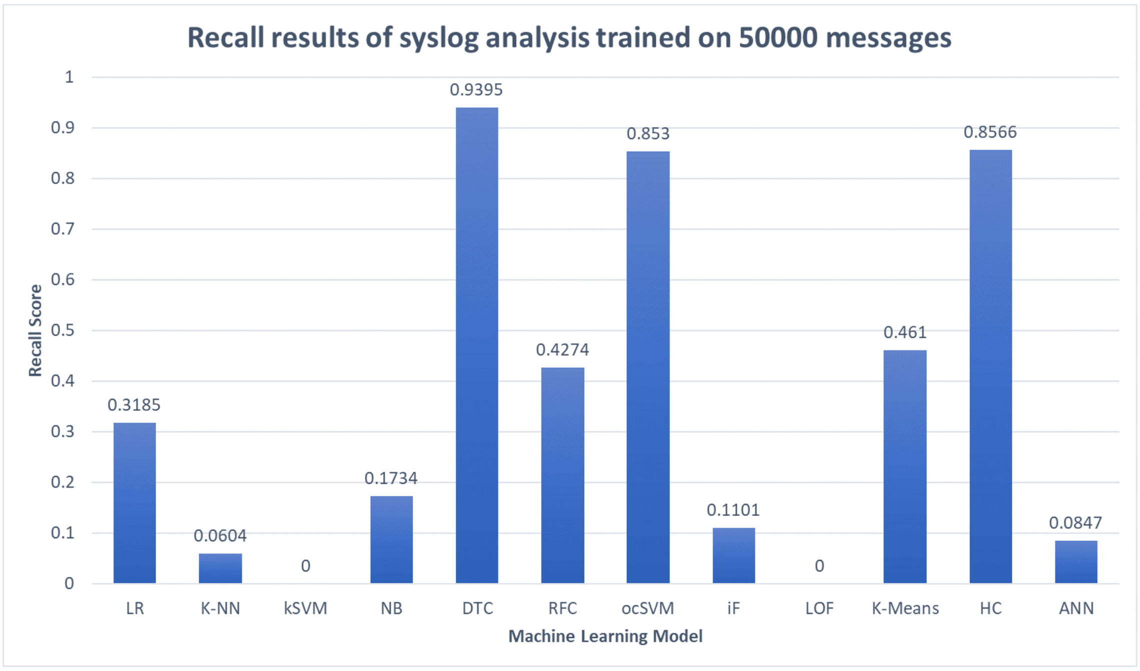

Fig. 6.5: Recall results of syslog analysis trained on 50000 messages

performance in identifying anomalies improved by almost 50%. Improvement was also observed for models *Logistic Regression* and *Naive Bayes*, and slightly also for models *Random Forest Classifier*, *Isolation Forest* and *Artificial Neural Network*. Models *One-class SVM* and *Hierarchical Clustering* stayed pretty much the same and enlargement of the dataset did not affect them. However, performance of models *K-Nearest Neighbors*, *Kernel SVM*, *Local Outlier Factor* and *K-Means* surprisingly decreased with enlargement of training dataset.

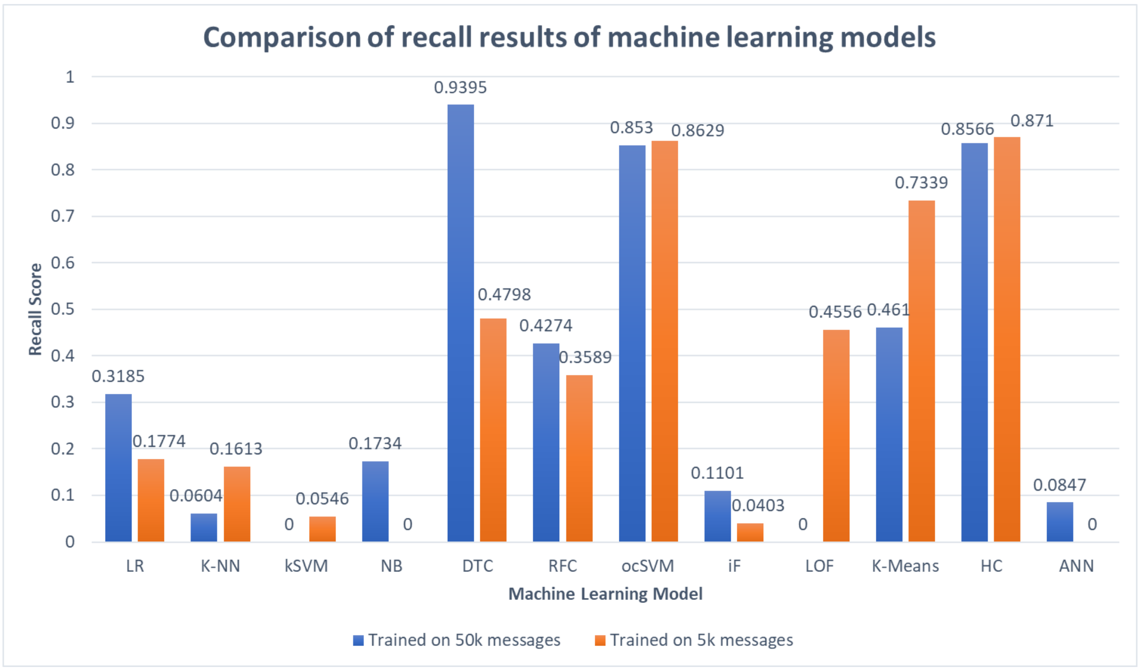

Fig. 6.6: Comparison of recall results of machine learning models

Live Testing of Analysis Tools

Both analysis tools were designed to be usable in real life packet and syslog prediction. They are able to label each log, whether it is anomaly or not. An example can be seen on listing 6.1.

```
May 17 2020 12:33:37 FW22 : %ASA-1-105004: (Primary) Monitoring on interface
↪  GigabitEthernet0/4 normal     Not anomaly
May 17 2020 12:33:37 FW25 : %ASA-4-733100: SYN attack drop rate 1 exceeded.
↪  Current burst rate is 425 per second, max configured rate is 0 ; Current
↪  average rate is 7532 per second, max configured rate is 0 ; Cumulative total
↪  count is 45783364     Anomaly
```

Listing 6.1: *Exampe of live analysis*

The tool will label each log message and prints it out into file *xxx_labelled.csv*, where *xxx* stands for the name of the machine learning model chosen for predictions.

Conclusion

Computer networks and distributed systems generate a big amount of logs every day. These logs have to be parsed and analyzed to keep the track of the system behaviour. Traditional analysis of system logs is complex, time consuming, and prone to human errors. On the other hand, machine learning methods can improve analyzing process and provide a solution to deal with complex and big amount of the log data. However, in this thesis, the anomaly detection using machine learning and deep learning algorithms were applied on network packets and syslog messages to detect security threats. Many steps were followed to achieve better accuracy of the proposed models, e.g., data preprocessing, algorithm tuning, and utilizing many machine learning models.

However, the supervised algorithm *Decision Tree Classifier* achieved the best accuracy in identification of anomalies, while unsupervised algorithms *One-class SVM* and *Hierarchical Clustering* achieved almost as good accuracy as *DTC*. The results were improved by algorithm tuning and increase of the training data set.

Interestingly, unsupervised algorithm *K-Means* and *Local Outlier Factor* were more accurate on smaller dataset than on the big one. Similarly, supervised algorithm *K-Nearest Neighbors* was little bit more accurate on smaller dataset.

Best algorithms of all was supervised algorithm *Decision Tree Classifier*, which was applied on live prediction of syslogs. It achieved stunning recall score of *0.9395*, which means it correctly predicted almost 94% of all anomalies.

However, tuning of the parameters was not automatized. Chosen parameters were either default or were tuned manually based on knowledge of the algorithm and brute-force tests. As future work, multiple automatized approaches for parameters tuning can be used. For instance, the *Grid Search* can be used to tune parameter of each model by exhaustively generating candidates from specified grid of parameters [69]. Since each model uses different parameters, each model needs its own grid search. This might be time-consuming, given the amount of machine learning models mentioned in this thesis.

Another approach can be inspired by *H20 AutoML* [70]. It automates process of building large number of models, with intention to find the best model.

Bibliography

[1] SHARMA, Avneesh: *How Different are Conventional Programming and Machine Learning?* [online; visited on 30. 05. 2020]. Available from URL: <`https://www.kdnuggets.com/2018/12/different-conventional-programming-machine-learning.html`>.

[2] *How Machine Learning Can Enable Anomaly Detection* [online]. Last update 13. 01. 2020 [visited on 30. 05. 2020]. Available from URL: <`https://medium.com/datadriveninvestor/how-machine-learning-can-enable-anomaly-detection-eed9286c5306`>.

[3] *What is Machine Learning? A definition* [online]. Last update 06. 05. 2020 [visited on 18. 05. 2020]. Available from URL: <`https://expertsystem.com/machine-learning-definition/`>.

[4] *Machine Learning* [online; visited on 18. 05. 2020]. Available from URL: <`https://www.geeksforgeeks.org/machine-learning/`>.

[5] *What Is Machine Learning? 3 things you need to know* [online; visited on 18. 05. 2020]. Available from URL: <`https://www.mathworks.com/discovery/machine-learning.html`>.

[6] ROSEBROCK, Adrian: *Anomaly detection with Keras, TensorFlow, and Deep Learning* [online]. Last update 02. 03. 2020 [visited on 18. 05. 2020]. Available from URL: <`https://www.pyimagesearch.com/2020/03/02/anomaly-detection-with-keras-tensorflow-and-deep-learning/`>.

[7] FLOVIK, Vegard: *How to use machine learning for anomaly detection and condition monitoring* [online]. Last update 31. 12. 2018 [visited on 18. 05. 2020]. Available from URL: <`https://towardsdatascience.com/how-to-use-machine-learning-for-anomaly-detection-and-condition-monitoring-6742f82900d7`>.

[8] BROWNLEE, Jason: *Supervised and Unsupervised Machine Learning Algorithms* [online]. Last update 12. 08. 2019 [visited on 18. 05. 2020]. Available from URL: <`https://machinelearningmastery.com/supervised-and-unsupervised-machine-learning-algorithms/`>.

[9] Hrishikesh D. Vinod, C.R. Rao: Handbook of statistics 42: *Financial, Macro and Micro Econometrics Using R*, North Holland, January 2020. 349 p. Hardcover ISBN: 9780128202500, eBook ISBN: 9780128202517.

[10] Mark Talabis; Robert McPherson; Inez Miyamoto; Jason Martin: *Information Security Analytics : Finding Security Insights, Patterns, and Anomalies in Big Data*, Syngress; 1. edition (December 10, 2014), 182 p. ISBN: 978-0-12-800207-0

[11] WASEEM, Mohammad: *How To Implement Classification In Machine Learning?* [online]. Last update 04. 12. 2019 [visited on 18. 05. 2020]. Available from URL:
<`https://www.edureka.co/blog/classification-in-machine-learning/`>.

[12] ASIRI, Sidath: *Machine Learning Classifiers* [online]. Last update 11. 06. 2018 [visited on 18. 05. 2020]. Available from URL:
<`https://towardsdatascience.com/machine-learning-classifiers-a5cc4e1b0623`>.

[13] LEONEL, Jorge: *Classification Methods in Machine Learning* [online]. Last update 09. 10. 2019 [visited on 18. 05. 2020]. Available from URL:
<`https://medium.com/@jorgesleonel/classification-methods-in-machine-learning-58ce63173db8`>.

[14] MOLNAR, Christoph: Interpretable Machine Learning: *A Guide for Making Black Box Models Explainable.* [online]. Last update 27. 04. 2020 [visited on 18. 05. 2020]. Available from URL:
<`https://christophm.github.io/interpretable-ml-book/`>.

[15] *Logistic Regression* [online; visited on 18. 05. 2020]. Available from URL:
<`https://ml-cheatsheet.readthedocs.io/en/latest/logistic_regression.html`>.

[16] *KNN(K-Nearest Neighbour) algorithm, maths behind it and how to find the best value for K* [online]. Last update 25. 10. 2019 [visited on 18. 05. 2020]. Available from URL:
<`https://medium.com/@rdhawan201455/knn-k-nearest-neighbour-algorithm-maths-behind-it-and-how-to-find-the-best-value-for-k-6ff5b0955e3d`>.

[17] HARRISON, Onel: *Machine Learning Basics with the K-Nearest Neighbors Algorithm* [online]. Last update 10. 09. 2018 [visited on 18. 05. 2020]. Available

from URL:
<`https://towardsdatascience.com/machine-learning-basics-with-the-k-nearest-neighbors-algorithm-6a6e71d01761`>.

[18] *K Nearest Neighbors - Classification* [online; visited on 18. 05. 2020]. Available from URL:
<`https://www.saedsayad.com/k_nearest_neighbors.htm`>.

[19] SUBRAMANIAN, Dhilip: *A Simple Introduction to K-Nearest Neighbors Algorithm* [online]. Last update 08. 06. 2019 [visited on 18. 05. 2020]. Available from URL:
<`https://towardsdatascience.com/a-simple-introduction-to-k-nearest-neighbors-algorithm-b3519ed98e`>.

[20] SRIVASTAVA, Tavish: *Introduction to k-Nearest Neighbors: A powerful Machine Learning Algorithm (with implementation in Python & R)* [online]. Last update 26. 03. 2018 [visited on 18. 05. 2020]. Available from URL:
<`https://www.analyticsvidhya.com/blog/2018/03/introduction-k-neighbours-algorithm-clustering/`>.

[21] SETHI, Alakh: *Support Vector Regression Tutorial for Machine Learning* [online]. Last update 27. 03. 2020 [visited on 18. 05. 2020]. Available from URL:
<`https://www.analyticsvidhya.com/blog/2020/03/support-vector-regression-tutorial-for-machine-learning/`>.

[22] GANDHI, Rohith: *Support Vector Machine — Introduction to Machine Learning Algorithms* [online]. Last update 07. 06. 2018 [visited on 18. 05. 2020]. Available from URL:
<`https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47`>.

[23] RAY, Sunil: *Understanding Support Vector Machine(SVM) algorithm from examples (along with code)* [online]. Last update 13. 09. 2017 [visited on 18. 05. 2020]. Available from URL:
<`https://www.analyticsvidhya.com/blog/2017/09/understaing-support-vector-machine-example-code/`>.

[24] *Support Vector Machine Algorithm* [online; visited on 18. 05. 2020]. Available from URL:
<`https://www.javatpoint.com/machine-learning-support-vector-machine-algorithm`>.

[25] STECANELLA, Bruno: *An Introduction to Support Vector Machines (SVM)* [online]. Last update 22. 06. 2017 [visited on 18. 05. 2020]. Available from URL: <`https://monkeylearn.com/blog/introduction-to-support-vector-machines-svm/`>.

[26] DUNLOP, Robert: *Understanding the Dot Product* [online]. Last update 26. 07. 2005 [visited on 18. 05. 2020]. Available from URL: <`http://www.mvps.org/DirectX/articles/math/dot/index.htm`>.

[27] BROWNLEE, Jason: *One-Class Classification Algorithms for Imbalanced Datasets* [online]. Last update 13. 03. 2020 [visited on 18. 05. 2020]. Available from URL: <`machinelearningmastery.com/one-class-classification-algorithms/`>.

[28] SCHOLKOPF, Bernhard; WILLIAMSON, Robert; SMOLA, Alex; SHAWE-TAYLOR, John; PLATT, John: *Support Vector Method for Novelty Detection* [online]. 1999 [visited on 18. 05. 2020]. Available from URL: <`http://users.cecs.anu.edu.au/~williams/papers/P126.pdf`>.

[29] BROWNLEE, Jason: *Naive Bayes Classifier From Scratch in Python* [online]. Last update 25. 10. 2019 [visited on 18. 05. 2020]. Available from URL: <`https://machinelearningmastery.com/naive-bayes-classifier-scratch-python/`>.

[30] HAYES, Adam: *Bayes' Theorem Definition* [online]. Last update 10. 04. 2020 [visited on 18. 05. 2020]. Available from URL: <`https://www.investopedia.com/terms/b/bayes-theorem.asp`>.

[31] *Bayes' Theorem Problems, Definition and Examples* [online; visited on 18. 05. 2020]. Available from URL: <`https://www.statisticshowto.com/bayes-theorem-problems/`>.

[32] ELLINOR, Andrew and others: *Bayes' Theorem and Conditional Probability* [online; visited on 18. 05. 2020]. Available from URL: <`https://brilliant.org/wiki/bayes-theorem/`>.

[33] *Naive Bayes Classifiers* [online; visited on 18. 05. 2020]. Available from URL: <`https://www.geeksforgeeks.org/naive-bayes-classifiers/`>.

[34] CHAKURE, Afroz: *Decision Tree Classification* [online]. Last update 06. 07. 2019 [visited on 18. 05. 2020]. Available from URL:

<https://towardsdatascience.com/decision-tree-classification-de64fc4d5aac>.

[35] *Decision Tree Classification Algorithm* [online; visited on 18. 05. 2020]. Available from URL:
<https://www.javatpoint.com/machine-learning-decision-tree-classification-algorithm>.

[36] *Classification Algorithms - Random Forest* [online; visited on 18. 05. 2020]. Available from URL:
<https://www.tutorialspoint.com/machine_learning_with_python/machine_learning_with_python_classification_algorithms_random_forest.htm>.

[37] POLAMURI, Saimadhu: *How the random forest algorithm works in machine learning* [online]. Last update 22. 05. 2017 [visited on 18. 05. 2020]. Available from URL:
<https://dataaspirant.com/2017/05/22/random-forest-algorithm-machine-learing/>.

[38] *How Random Forest Algorithm Works in Machine Learning* [online]. Last update 24. 10. 2017 [visited on 18. 05. 2020]. Available from URL:
<https://medium.com/@Synced/how-random-forest-algorithm-works-in-machine-learning-3c0fe15b6674>.

[39] *Random Forest Algorithm* [online; visited on 18. 05. 2020]. Available from URL:
<https://www.javatpoint.com/machine-learning-random-forest-algorithm>.

[40] MISHRA, Sanatan: *Unsupervised Learning and Data Clustering* [online]. Last update 19. 05. 2017 [visited on 18. 05. 2020]. Available from URL:
<https://towardsdatascience.com/unsupervised-learning-and-data-clustering-eeecb78b422a>.

[41] *Unsupervised Machine Learning: What is, Algorithms, Example* [online; visited on 18. 05. 2020]. Available from URL:
<https://www.guru99.com/unsupervised-machine-learning.html>.

[42] BILYK, Volodymyr: *Guide to Unsupervised Machine Learning: 7 Real Life Examples* [online; visited on 18. 05. 2020]. Available from URL:
<https://theappsolutions.com/blog/development/unsupervised-machine-learning/#contents_11>.

[43] *Apriori Algorithm* [online; visited on 18. 05. 2020]. Available from URL: <https://www.geeksforgeeks.org/apriori-algorithm/>.

[44] *Machine learning technique for finding hidden patterns or intrinsic structures in data* [online; visited on 18. 05. 2020]. Available from URL: <https://www.mathworks.com/discovery/unsupervised-learning.html>.

[45] LIU, Fei Tony; TING, Kai Ming; ZHOU, Zhi-Hua: 2008 Eighth IEEE International Conference on Data Mining: *Isolation Forest*, Pisa, 2008. pp. 413-422. ISBN: 978-0-7695-3502-9.

[46] BEUNIG, Markus M.; KRIEGEL, Hans-Peter; NG, Raymong T.; SANDER, Jörg: *LOF: Identifying Density-Based Local Outliers. Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pp. 93–104. ISBN 1-58113-217-4. Available from URL: <https://www.dbs.ifi.lmu.de/Publikationen/Papers/LOF.pdf>.

[47] *Local Outlier Factor* [online; visited on 18. 05. 2020]. Available from URL: <https://turi.com/learn/userguide/anomaly_detection/local_outlier_factor.html>.

[48] GARBADE, Dr. Michael J.: *Understanding K-means Clustering in Machine Learning* [online]. Last update 12. 09. 2018 [visited on 18. 05. 2020]. Available from URL: <https://towardsdatascience.com/understanding-k-means-clustering-in-machine-learning-6a6e67336aa1>.

[49] *K means Clustering – Introduction* [online; visited on 18. 05. 2020]. Available from URL: <https://www.geeksforgeeks.org/k-means-clustering-introduction/>.

[50] SHARMA, Pulkit: *The Most Comprehensive Guide to K-Means Clustering You'll Ever Need* [online]. Last update 19. 08. 2019 [visited on 18. 05. 2020]. Available from URL: <https://www.analyticsvidhya.com/blog/2019/08/comprehensive-guide-k-means-clustering/>.

[51] *Hierarchical Clustering* [online; visited on 18. 05. 2020]. Available from URL: <https://www.saedsayad.com/clustering_hierarchical.htm>.

[52] SHARMA, Pulkit: *A Beginner's Guide to Hierarchical Clustering and how to Perform it in Python* [online]. Last update 27. 05. 2019 [visited on 18. 05. 2020]. Available from URL:
<`https://www.analyticsvidhya.com/blog/2019/05/beginners-guide-hierarchical-clustering/`>.

[53] *Deep Learning Tutorial for Beginners: Neural Network Classification* [online; visited on 18. 05. 2020]. Available from URL:
<`https://www.guru99.com/deep-learning-tutorial.html`>.

[54] *What Is Deep Learning?* [online; visited on 18. 05. 2020]. Available from URL:
<`https://www.mathworks.com/discovery/deep-learning.html`>.

[55] *Keras Tutorial - Layers* [online; [visited on 18. 05. 2020]. Available from URL:
<`https://www.tutorialspoint.com/keras/keras_layers.htm`>.

[56] LATEEF, Zulaikha: *What Is A Neural Network? Introduction To Artificial Neural Networks* [online]. Last update 28. 08. 2019 [visited on 18. 05. 2020]. Available from URL:
<`https://www.edureka.co/blog/what-is-a-neural-network/`>.

[57] HELLER, Martin: *What is Keras? The deep neural network API explained* [online]. Last update 28. 01. 2019 [visited on 18. 05. 2020]. Available from URL:
<`https://www.infoworld.com/article/3336192/what-is-keras-the-deep-neural-network-api-explained.html`>.

[58] GARBADE, Dr. Michael J.: *A Simple Introduction to Natural Language Processing* [online]. Last update 15. 10. 2018 [visited on 18. 05. 2020]. Available from URL:
<`https://becominghuman.ai/a-simple-introduction-to-natural-language-processing-ea66a1747b32`>.

[59] KUMAWAT, Dinesh: *7 Natural Language Processing Techniques for Extracting Information* [online]. Last update 18. 11. 2019 [visited on 18. 05. 2020]. Available from URL:
<`https://www.analyticssteps.com/blogs/7-natural-language-processing-techniques-extracting-information`>.

[60] *Sentiment Analysis Explained: What is Sentiment Analysis?* [online; visited on 18. 05. 2020]. Available from URL:
<`https://www.lexalytics.com/technology/sentiment-analysis`>.

[61] BEDAPUDI, Praneeth: *DeepCorrection 1: Sentence Segmentation of unpunctuated text.* [online]. Last update 17. 11. 2018 [visited on 18. 05. 2020]. Available from URL:
<`https://medium.com/@praneethbedapudi/deepcorrection-1-sentence-segmentation-of-unpunctuated-text-a1dbc0db4e98`>.

[62] Cisco ASA Series Syslog Messages: *Chapter: Messages Listed by Severity Level* [online]. Last update 10. 04. 2020 [visited on 18. 05. 2020]. Available from URL:
<`https://www.cisco.com/c/en/us/td/docs/security/asa/syslog/b_syslog/syslogs-sev-level.html#con_1009233`>.

[63] *Welcome to Faker's documentation!* [online; [visited on 18. 05. 2020]. Available from URL:
<`https://faker.readthedocs.io/en/master/`>.

[64] argparse: *Parser for command-line options, arguments and sub-commands* [online; visited on 18. 05. 2020]. Available from URL:
<`https://docs.python.org/3/library/argparse.html`>.

[65] SHARAFALDIN, Iman; LASHKARI, Arash Habibi; GHORBANI, Ali A.: *Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization, 4th International Conference on Information Systems Security and Privacy (ICISSP)*, Portugal, January 2018. Available from URL:
<`https://www.unb.ca/cic/datasets/ids-2017.html`>.

[66] PEDREGOSA *et al.*: Scikit-learn: Machine Learning in Python: *Journal of Machine Learning Research*, pp. 2825-2830, 2011.

[67] CHOLLET, Francois and others: *Keras*, 2015. Available from URL:
<`https://keras.io`>.

[68] *Confusion Matrix in Machine Learning* [online; visited on 18. 05. 2020]. Available from URL:
<`https://www.geeksforgeeks.org/confusion-matrix-machine-learning/`>.

[69] *Tuning the hyper-parameters of an estimator* [online; visited on 30. 05. 2020]. Available from URL:
<`https://scikit-learn.org/stable/modules/grid_search.html`>.

[70] *H2O AutoML Tutorial* [online; visited on 30. 05. 2020]. Available from URL:
<`http://docs.h2o.ai/h2o-tutorials/latest-stable/h2o-world-2017/automl/index.html`>.

A Attachments

```
xsiklo00_thesis_attachments.zip/
└─ xsiklo00_thesis_attachments/
   ├─ Syslog-Generator/
   │  ├─ LICENSE.md ........................................... 2 KB
   │  ├─ README.md ............................................ 2 KB
   │  └─ syslog_generator.py ................................. 26 KB
   ├─ Syslog-Messages-Analysis/
   │  ├─ classifiers/
   │  ├─ Results/
   │  ├─ data_preprocessing.py ................................ 4 KB
   │  ├─ LICENSE.md ........................................... 2 KB
   │  ├─ ML_modules.py ......................................... 6 KB
   │  ├─ README.md ............................................ 3 KB
   │  └─ syslog_messages_analysis.py ......................... 12 KB
   └─ Traffic-Analysis/
      ├─ classifiers/
      ├─ Results/
      ├─ data_preprocessing.py ................................ 4 KB
      ├─ LICENSE.md ........................................... 2 KB
      ├─ ML_modules.py ......................................... 6 KB
      ├─ README.md ............................................ 3 KB
      └─ traffic_analysis.py .................................. 13 KB
```